Programming Assignment 1: Percolation

Write a program to estimate the value of the percolation threshold via Monte Carlo simulation.

Install a Java programming environment. Install a Java programming environment on your computer by following these step-by-step instructions for your operating system [Mac OS X · Windows · Linux]. After following these instructions, the commands javac-algs4 and java-algs4 will classpath in both stdlib.jar and algs4.jar: the former contains libraries for reading data from standard input, writing data to standard output, drawing results to standard draw, generating random numbers, computing statistics, and timing programs; the latter contains all of the algorithms in the textbook.

Percolation. Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as percolation to model such situations.

The model. We model a percolation system using an N-by-N grid of sites. Each site is either open or blocked. A full site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system percolates if there is a full site in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. (For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.)

Percolates

The problem. In a famous scientific problem, researchers are interested in the following question: if sites are independently set to be open with probability p (and therefore blocked with probability $1 - p$), what is the probability that the system percolates? When p equals 0, the system does not percolate; when p equals 1, the system percolates. The plots below show the site vacancy probability p versus the percolation probability for 20-by-20 random grid (left) and 100-by-100 random grid (right).

Percolation threshold for 20-by-20 grid Percolation threshold for 100-by-100 grid

When N is sufficiently large, there is a threshold value p* such that when p < p* a random N-by-N grid almost never percolates, and when p > p*, a random N-by-N grid almost always percolates. No mathematical solution for determining the percolation threshold p* has yet been derived. Your task is to write a computer program to estimate p*.

Percolation data type. To model a percolation system, create a data type Percolation with the following API:

```
public class Percolation {

   public Percolation(int N)              // create N-by-N grid, with all sites blocked

   public void open(int i, int j)         // open site (row i, column j) if it is not already

   public boolean isOpen(int i, int j)    // is site (row i, column j) open?

   public boolean isFull(int i, int j)    // is site (row i, column j) full?

   public boolean percolates()            // does the system percolate?

}
```

By convention, the indices i and j are integers between 1 and N, where (1, 1) is the upper-left site: Throw a java.lang.IndexOutOfBoundsException if either i or j is outside this range. The constructor should take time proportional to N^2; all methods should take constant time plus a constant number of calls to the union-find methods union(), find(), connected(), and count().

Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

Initialize all sites to be blocked.

Repeat the following until the system percolates:

Choose a site (row i, column j) uniformly at random among all blocked sites.

Open the site (row i, column j).

The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20-by-20 lattice according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.

Percolation 50 sites

50 open sites

Percolation 100 sites

100 open sites

Percolation 150 sites

150 open sites

Percolation 204 sites

204 open sites

By repeating this computation experiment T times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let xt be the fraction of open sites in computational experiment t. The sample mean μ provides an estimate of the percolation threshold; the sample standard deviation σ measures the sharpness of the threshold.

Estimating the sample mean and variance

Assuming T is sufficiently large (say, at least 30), the following provides a 95% confidence interval for the percolation threshold:

95% confidence interval for percolation threshold

To perform a series of computational experiments, create a data type PercolationStats with the following API.

```
public class PercolationStats {

   public PercolationStats(int N, int T)    // perform T independent computational experiments on an N-by-N grid

   public double mean()                     // sample mean of percolation threshold

   public double stddev()                   // sample standard deviation of percolation threshold

   public double confidenceLo()             // returns lower bound of the 95% confidence interval

   public double confidenceHi()             // returns upper bound of the 95% confidence interval

   public static void main(String[] args)   // test client, described below

}
```

The constructor should throw a java.lang.IllegalArgumentException if either $N \leq 0$ or $T \leq 0$.

Also, include a main() method that takes two command-line arguments N and T, performs T independent computational experiments (discussed above) on an N-by-N grid, and prints out the mean, standard deviation, and the 95% confidence interval for the percolation threshold. Use standard random from our standard libraries to generate random numbers; use standard statistics to compute the sample mean and standard deviation.

% java PercolationStats 200 100

mean = 0.5929934999999997

stddev = 0.00876990421552567

95% confidence interval = 0.5912745987737567, 0.5947124012262428

% java PercolationStats 200 100

mean = 0.592877

stddev = 0.009990523717073799

95% confidence interval = 0.5909188573514536, 0.5948351426485464

% java PercolationStats 2 10000

mean = 0.666925

stddev = 0.11776536521033558

95% confidence interval = 0.6646167988418774, 0.6692332011581226

% java PercolationStats 2 100000

mean = 0.6669475

stddev = 0.11775205263262094

95% confidence interval = 0.666217665216461, 0.6676773347835391

Analysis of running time and memory usage (optional and not graded). Implement the Percolation data type using the quick-find algorithm QuickFindUF.java from algs4.jar.

Use the stopwatch data type from our standard library to measure the total running time of PercolationStats. How does doubling N affect the total running time? How does doubling T affect the total running time? Give a formula (using tilde notation) of the total running time on your computer (in seconds) as a single function of both N and T.

Using the 64-bit memory-cost model from lecture, give the total memory usage in bytes (using tilde notation) that an N-by-N percolation system uses. Count all memory that is used, including memory for the union-find data structure.

Now, implement the Percolation data type using the weighted quick-union algorithm WeightedQuickUnionUF.java from algs4.jar. Answer the questions in the previous paragraph.

Deliverables. Submit only Percolation.java (using the weighted quick-union algorithm as implemented in the WeightedQuickUnionUF class) and PercolationStats.java. We will supply stdlib.jar and WeightedQuickUnionUF. Your submission may not call any library functions other than those in java.lang, stdlib.jar, and WeightedQuickUnionUF.

****

Programming Assignment 2: Randomized Queues and Deques

Write a generic data type for a deque and a randomized queue. The goal of this assignment is to implement elementary data structures using arrays and linked lists, and to introduce you to generics and iterators.

Dequeue. A double-ended queue or deque (pronounced "deck") is a generalization of a stack and a queue that supports inserting and removing items from either the front or the back of the data structure. Create a generic data type Deque that implements the following API:

```
public class Deque<Item> implements Iterable<Item> {
  public Deque()                  // construct an empty deque
  public boolean isEmpty()        // is the deque empty?
  public int size()               // return the number of items on the deque
  public void addFirst(Item item)  // insert the item at the front
  public void addLast(Item item)   // insert the item at the end
  public Item removeFirst()       // delete and return the item at the front
  public Item removeLast()        // delete and return the item at the end
```

```
  public Iterator<Item> iterator()   // return an iterator over items in order from front to end
}
```

Throw a java.lang.NullPointerException if the client attempts to add a null item; throw a java.util.NoSuchElementException if the client attempts to remove an item from an empty deque; throw a java.lang.UnsupportedOperationException if the client calls the remove() method in the iterator; throw a java.util.NoSuchElementException if the client calls the next() method in the iterator and there are no more items to return.

Your deque implementation should support each deque operation in constant worst-case time and use space proportional to the number of items currently in the deque. Additionally, your iterator implementation should support the operations next() and hasNext() (plus construction) in constant worst-case time and use a constant amount of extra space per iterator.

Randomized queue. A randomized queue is similar to a stack or queue, except that the item removed is chosen uniformly at random from items in the data structure. Create a generic data type RandomizedQueue that implements the following API:

```
public class RandomizedQueue<Item> implements Iterable<Item> {
  public RandomizedQueue()          // construct an empty randomized queue
  public boolean isEmpty()          // is the queue empty?
  public int size()                 // return the number of items on the queue
  public void enqueue(Item item)    // add the item
  public Item dequeue()             // delete and return a random item
  public Item sample()              // return (but do not delete) a random item
  public Iterator<Item> iterator()  // return an independent iterator over items in random order
}
```

Throw a java.lang.NullPointerException if the client attempts to add a null item; throw a java.util.NoSuchElementException if the client attempts to sample or dequeue an item from an empty randomized queue; throw a java.lang.UnsupportedOperationException if the client calls the remove() method in the iterator; throw a java.util.NoSuchElementException if the client calls the next() method in the iterator and there are no more items to return.

Your randomized queue implementation should support each randomized queue operation (besides creating an iterator) in constant amortized time and use space proportional to the

number of items currently in the queue. That is, any sequence of M randomized queue operations (starting from an empty queue) should take at most cM steps in the worst case, for some constant c. Additionally, your iterator implementation should support construction in time linear in the number of items and it should support the operations next() and hasNext() in constant worst-case time; you may use a linear amount of extra memory per iterator. The order of two or more iterators to the same randomized queue should be mutually independent; each iterator must maintain its own random order.

Subset client. Write a client program Subset.java that takes a command-line integer k, reads in a sequence of N strings from standard input using StdIn.readString(), and prints out exactly k of them, uniformly at random. Each item from the sequence can be printed out at most once. You may assume that $k \geq 0$ and no greater than the number of string on standard input.

```
% echo A B C D E F G H I | java Subset 3
C
G
A

% echo AA BB BB BB BB BB CC CC | java Subset 8
BB
AA
BB
CC

% echo A B C D E F G H I | java Subset 3
E
F
G

BB
CC
BB
```

Your client should use only constant space plus one object either of type Deque or of type RandomizedQueue; use generics properly to avoid casting and compiler warnings. It should also use time and space proportional to at most N in the worst case, where N is the number of strings on standard input. (For an extra challenge, use space proportional to k.) It should have the following API.

```
public class Subset {

   public static void main(String[] args)

}
```

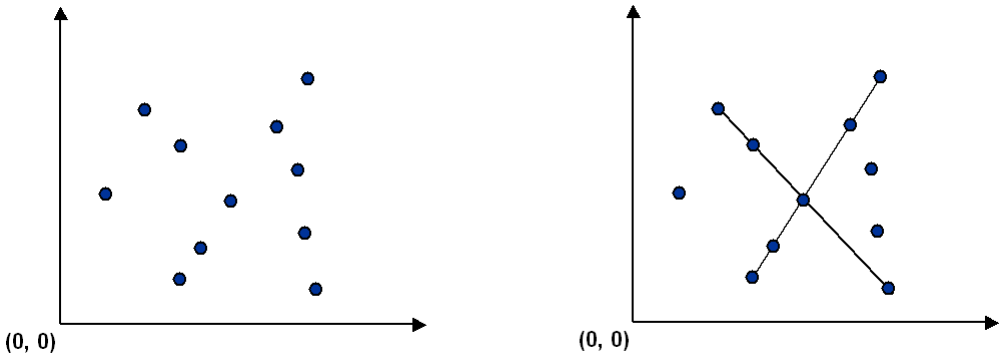
Deliverables. Submit only Deque.java, RandomizedQueue.java, and Subset.java. We will supply stdlib.jar. You may not call any library functions other than those in java.lang and stdlib.jar.

Programming Assignment 3: Pattern Recognition

Write a program to recognize line patterns in a given set of points.

Computer vision involves analyzing patterns in visual images and reconstructing the real-world objects that produced them. The process in often broken up into two phases: *feature detection* and *pattern recognition*. Feature detection involves selecting important features of the image; pattern recognition involves discovering patterns in the features. We will investigate a particularly clean pattern recognition problem involving points and line segments. This kind of pattern recognition arises in many other applications such as statistical data analysis.

The problem. Given a set of *N* distinct points in the plane, draw every (maximal) line segment that connects a subset of 4 or more of the points.



Point data type. Create an immutable data type Point that represents a point in the plane by implementing the following API:

```
public class Point implements Comparable<Point> {
   public final Comparator<Point> SLOPE_ORDER;       // compare points by slope to this point

   public Point(int x, int y)                        // construct the point (x, y)

   public   void draw()                              // draw this point
   public   void drawTo(Point that)                  // draw the line segment from this point to that
point
   public String toString()                          // string representation

   public    int compareTo(Point that)               // is this point lexicographically smaller than that
point?
   public double slopeTo(Point that)                 // the slope between this point and that point
}
```

To get started, use the data type Point.java, which implements the constructor and the draw(), drawTo(), and toString() methods. Your job is to add the following components.

- The compareTo() method should compare points by their *y*-coordinates, breaking ties by their *x*-coordinates. Formally, the invoking point (x_0, y_0) is *less than* the argument point (x_1, y_1) if and only if either $y_0 < y_1$ or if $y_0 = y_1$ and $x_0 < x_1$.
- The slopeTo() method should return the slope between the invoking point (x_0, y_0) and the argument point (x_1, y_1), which is given by the formula $(y_1 - y_0) / (x_1 - x_0)$. Treat the slope

of a horizontal line segment as positive zero [added 7/29]; treat the slope of a vertical line segment as positive infinity; treat the slope of a degenerate line segment (between a point and itself) as negative infinity.

- The SLOPE_ORDER comparator should compare points by the slopes they make with the invoking point (x_0, y_0). Formally, the point (x_1, y_1) is *less than* the point (x_2, y_2) if and only if the slope $(y_1 - y_0) / (x_1 - x_0)$ is less than the slope $(y_2 - y_0) / (x_2 - x_0)$. Treat horizontal, vertical, and degenerate line segments as in the slopeTo() method.

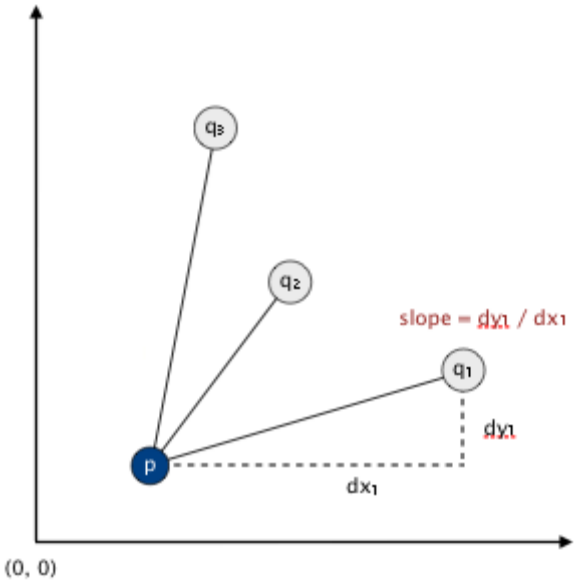
Brute force. Write a program Brute.java that examines 4 points at a time and checks whether they all lie on the same line segment, printing out any such line segments to standard output and drawing them using standard drawing. To check whether the 4 points*p*, *q*, *r*, and *s* are collinear, check whether the slopes between *p* and *q*, between *p* and *r*, and between *p* and *s* are all equal.

The order of growth of the running time of your program should be N^4 in the worst case and it should use space proportional to N.

A faster, sorting-based solution. Remarkably, it is possible to solve the problem much faster than the brute-force solution described above. Given a point *p*, the following method determines whether *p* participates in a set of 4 or more collinear points.

- Think of *p* as the origin.
- For each other point *q*, determine the slope it makes with *p*.
- Sort the points according to the slopes they makes with *p*.
- Check if any 3 (or more) adjacent points in the sorted order have equal slopes with respect to *p*. If so, these points, together with *p*, are collinear.

Applying this method for each of the N points in turn yields an efficient algorithm to the problem. The algorithm solves the problem because points that have equal slopes with respect to *p* are collinear, and sorting brings such points together. The algorithm is fast because the bottleneck operation is sorting.



Write a program Fast.java that implements this algorithm. The order of growth of the running time of your program should be $N^2 \log N$ in the worst case and it should use space proportional to N.

APIs. [Added 7/25] Each program should take the name of an input file as a command-line argument, read the input file (in the format specified below), print to standard output the line segments discovered (in the format specified below), and draw to standard draw the line segments discovered (in the format specified below). Here are the APIs.

```
public class Brute {
   public static void main(String[] args)
}

public class Fast {
   public static void main(String[] args)
}
```

Input format. Read the points from an input file in the following format: An integer *N*, followed by *N* pairs of integers (*x*, *y*), each between 0 and 32,767. Below are two examples.

% **more input6.txt**

```
6
19000  10000
18000  10000
32000  10000
21000  10000
 1234   5678
14000  10000
```

% **more input8.txt**

```
8
10000      0
    0  10000
 3000   7000
 7000   3000
20000  21000
 3000   4000
14000  15000
 6000   7000
```

Output format. Print to standard output the line segments that your program discovers, one per line. Print each line segment as an *ordered* sequence of its constituent points, separated by " -> ".

% **java Brute input6.txt**

```
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000)
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (32000, 10000)
(14000, 10000) -> (18000, 10000) -> (21000, 10000) -> (32000, 10000)
(14000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
(18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
```

% **java Brute input8.txt**

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

% **java Fast input6.txt**

```
(14000, 10000) -> (18000, 10000) -> (19000, 10000) -> (21000, 10000) -> (32000, 10000)
```

% **java Fast input8.txt**

```
(10000, 0) -> (7000, 3000) -> (3000, 7000) -> (0, 10000)
(3000, 4000) -> (6000, 7000) -> (14000, 15000) -> (20000, 21000)
```

Also, draw the points using draw() and draw the line segments using drawTo(). Your programs should call draw() once for each point in the input file and it should call drawTo() once for each line segment discovered. Before drawing, use StdDraw.setXscale(0, 32768) and StdDraw.setYscale(0, 32768) to rescale the coordinate system.

For full credit, do not print *permutations* of points on a line segment (e.g., if you output $p \to q \to r \to s$, do not also output either $s \to r \to q \to p$ or $p \to r \to q \to s$). Also, for full credit in Fast.java, do not print or plot *subsegments* of a line segment containing 5 or more points (e.g., if you output $p \to q \to r \to s \to t$, do not also output either $p \to q \to s \to t$ or $q \to r \to s \to t$); you may print out such subsegments in Brute.java.

Deliverables. Submit only Brute.java, Fast.java, and Point.java. We will supply stdlib.jar and algs4.jar. Your may not call any library functions other than those in java.lang, java.util, stdlib.jar, and algs4.jar.

This assignment was developed by Kevin Wayne.