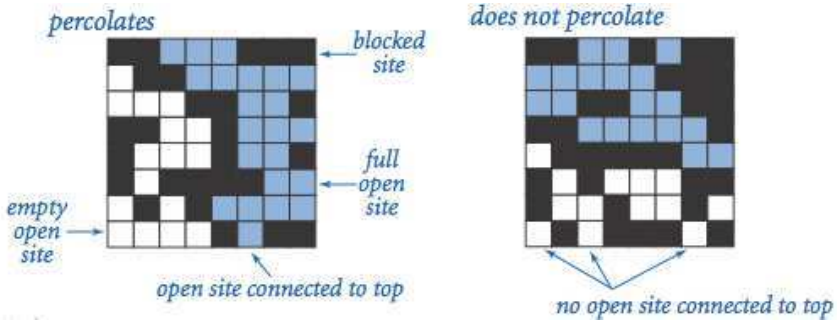
Programming Assignment 1: Percolation

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

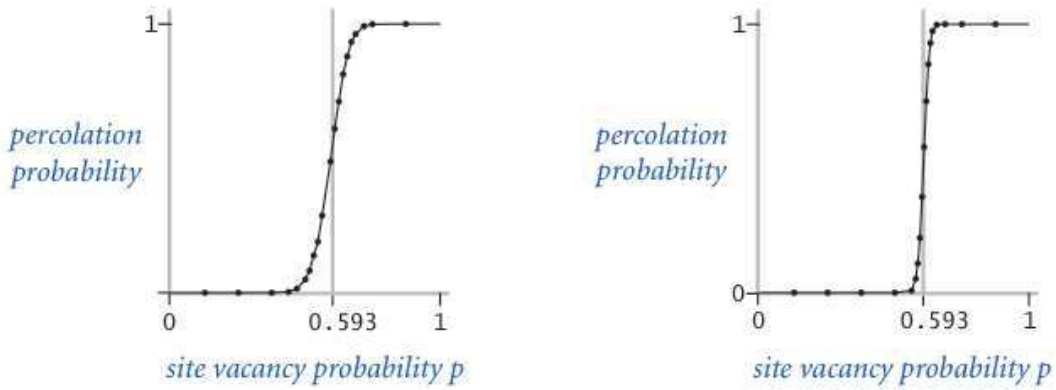
Install a Java programming environment. Install a Java programming environment on your computer by following these step-by-step instructions for your operating system [Mac OS X · Windows · Linux]. After following these instructions, the commands `javac-algs4` and `java-algs4` will classpath in both stdlib.jar 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.)



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).



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 open 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?

   public static void main(String[] args   // test client (optional)
}
```

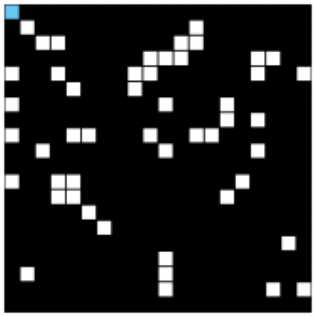
Corner cases. By convention, the row and column indices *i* and *j* are integers between 1 and *N*, where (1, 1) is the upper-left site: Throw a `java.lang.IndexOutOfBoundsException` if any argument to `open()`, `isOpen()`, or `isFull()` is outside its prescribed range. The constructor should throw a `java.lang.IllegalArgumentException` if $N \leq 0$.

Performance requirements. 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()`.

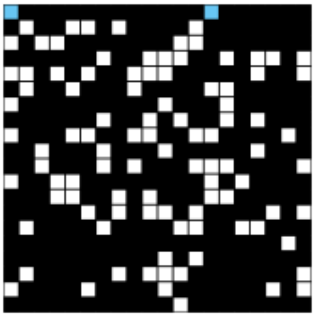
Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

- 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.

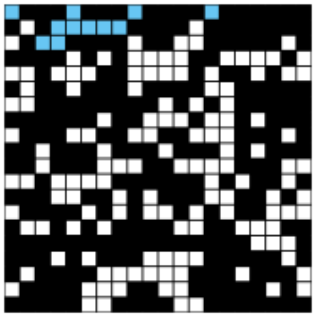
For example, if sites are opened in a 20-by-20 lattice according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.



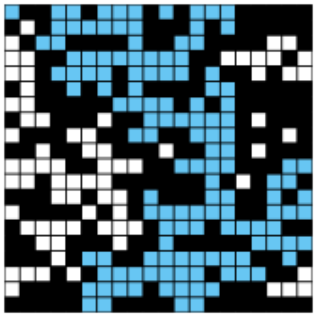
50 open sites



100 open sites



150 open 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 x_t 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.

$$\mu = \frac{x_1 + x_2 + \cdots + x_T}{T}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_T - \mu)^2}{T - 1}$$

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

$$\left[\mu - \frac{1.96\sigma}{\sqrt{T}}, \ \mu + \frac{1.96\sigma}{\sqrt{T}} \right]$$

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 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()              // low  endpoint of 95% confidence interval
```

```
   public double confidenceHi()             // high endpoint of 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 a `Percolation` object uses to model an N-by-N percolation system. 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`.

For fun. Create your own percolation input file and share it in the discussion forums. For some inspiration, see these nonogram puzzles.

Programming Assignment 1 Checklist: Percolation

Frequently Asked Questions (General)

What's a checklist? The assignment provides the programming assignment specification; the checklist provides clarifications, test data, and hints that might be helpful in completing the assignment.

Which Java programming environment should I use? For novices, we recommend the lightweight IDE DrJava along with the command line. If you use our Mac OS X or Windows installer, then everything should be configured and ready to go. If you prefer to use a different IDE (such as Eclipse), that's perfectly fine too—just be sure that you know how to do the following:

- Add `stdlib.jar` and `algs4.jar` to your Java classpath.
- Enter command-line arguments.
- Use standard input and standard output (and, ideally, redirect them to or from a file).

What are the input and output libraries? We have designed a set of easy-to-use Java libraries in `stdlib.jar` for input and output that you are required to use in this course. Here are the APIs.

Where can I find the Java code for the algorithms and data structures from lecture and the textbook? They are in `algs4.jar`. Here are the APIs.

How can I classpath in the textbook libraries from the command line? If you use our Mac OS X or Windows installer, then you can automatically classpath in the textbook libraries using the commands `javac-algs4` and `java-algs4` (instead of `javac` and `java`).

I haven't programmed in Java in a while. What material do I need to remember? For a review of our Java programming model (including our input and output libraries), read Sections 1.1 and 1.2 of *Algorithms, 4th Edition*.

Can I use various Java libraries in this assignment, such as `java.util.LinkedList`, `java.util.ArrayList`, `java.util.TreeMap`, and `java.util.HashMap`? No. You should not use any Java libraries until we have implemented equivalent versions in lecture. Once we have introduced them in lecture, you are free to use either the Java library version or our equivalent. You are welcome to use classes in the Java language such as `Math.sqrt()` and `Integer.parseInt()`.

How do I throw a java.lang.IndexOutOfBoundsException? Use a `throw` statement like the following:

```
if (i <= 0 || i > N) throw new IndexOutOfBoundsException("row index i out of bounds");
```

Your code should not attempt to catch any exceptions—this will interfere with our grading scripts.

How should I format and comment my code? Here are some recommended style guidelines. Below are some that are particularly important (though we will not deduct for style in this course).

- Include a bold (or Javadoc) comment at the beginning of each file with your name, date, the purpose of the program, and how to execute it.
- Include a bold (or Javadoc) comment describing every method.
- Include a comment describing every instance variable.
- Indent consistently, using 3 or 4 spaces for each indentation level. Do not use hard tabs.
- Do not exceed 80 characters per line. This rule also applies to the `readme.txt` file.
- Avoid unexplained magic numbers, especially ones that are used more than once.

Frequently Asked Questions (Percolation)

What are the goals of this assignment?

- Set up a Java programming environment.
- Use our input and output libraries.
- Learn about a scientific application of the union–find data structure.
- Measure the running time of a program and use the doubling hypothesis to make predictions.
- Measure the amount of memory used by a data structure.

Can I add (or remove) methods to (or from) Percolation? No. You must implement the `Percolation` API exactly as specified, with the identical set of public methods and signatures or your assignment will not be graded. However, you are encouraged to add private methods that enhance the readability, maintainability, and modularity of your program. The one exception is `main()`—you are always permitted to add this method to test your code, but we will not call it unless we specify it in our API.

Can my Percolation data type assume the row and column indices are between 0 and N-1? No. The API specifies that valid row and column indices are

between 1 and N.

Why is it so important to implement the prescribed API? Writing to an API is an important skill to master because it is an essential component of modular programming, whether you are developing software by yourself or as part of a group. When you develop a module that properly implements an API, anyone using that module (including yourself, perhaps at some later time) does not need to revisit the details of the code for that module when using it. This approach greatly simplifies writing large programs, developing software as part of a group, or developing software for use by others.

Most important, when you properly implement an API, others can write software to use your module or to test it. We do this regularly when grading your programs. For example, your `PercolationStats` client should work with our `Percolation` data type and vice versa. If you add an extra public method to `Percolation` and call them from `PercolationStats`, then your client won't work with our `Percolation` data type. Conversely, our `PercolationStats` client may not work with your `Percolation` data type if you remove a public method.

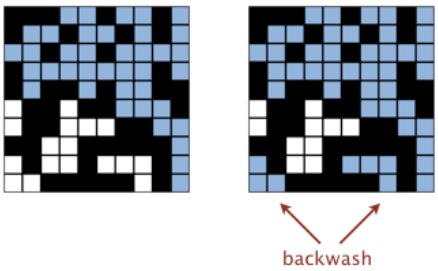
How many lines of code should my program be? You should strive for clarity and efficiency. Our reference solution for `Percolation.java` is about 70 lines, plus a test client. Our `PercolationStats.java` client is about 50 lines. If you are re-implementing the union-find data structure (instead of reusing the implementations provided), you are on the wrong track.

What assumptions can I make about the input to main() in PercolationStats? It can be any valid input: an integer $N \geq 1$ and an integer $T \geq 1$. In general, in this course you can assume that the input is of the specified format. But you do need to deal with pathological cases such as $N = 1$.

What should `stddev()` return if T equals 1? The sample standard deviation is undefined. We recommend returning `Double.NaN`.

After the system has percolated, my PercolationVisualizer colors in light blue all sites connected to open sites on the bottom (in addition to those connected to open sites on the top). Is this "backwash" acceptable? No, this is likely a bug in `Percolation`. It is only a minor deduction (because it impacts only the visualizer and not the experiment to estimate the percolation threshold), so don't go crazy trying to get this detail. However, many students consider this to be the most challenging and creative part of the assignment (especially if you limit yourself to one union-find object).

```
% java PercolationVisualizer input10.txt
```



How do I generate a site uniformly at random among all blocked sites for use in PercolationStats? Pick a site at random (by using `StdRandom` to generate two integers between 1 and N) and use this site if it is blocked; if not, repeat.

I don't get reliable timing information in PercolationStats when $N = 200$. What should I do? Increase the size of N (say to 400, 800, and 1600), until the mean running time exceeds its standard deviation.

Style and Bug Checkers

Style checker. We recommend using Checkstyle 5.5 (and the configuration file checkstyle.xml) to check the style of your Java programs. Here is a list of available Checkstyle checks.

Bug checker. We recommend using FindBugs 2.0.3 (and the configuration file findbugs.xml) to identify common bug patterns in your code. Here is a summary of FindBugs Bug descriptions.

Mac OS X and Windows installer. If you used our Mac OS X or Windows installer, these programs are already installed as command-line utilities. You can check a single file or multiple files via the commands:

```
% checkstyle-algs4 HelloWorld.java
% checkstyle-algs4 *.java

% findbugs-algs4 HelloWorld.class
% findbugs-algs4 *.class
```

Note that Checkstyle inspects the source code; Findbugs inspects the compiled code.

Eclipse. For Eclipse users, there is a Checkstyle plugin for Eclipse and a Findbugs plugin for Eclipse.

Caveat. The appearance of a warning message does not necessarily lead to a deduction (and, in some cases, it does not even indicate an error).

Testing

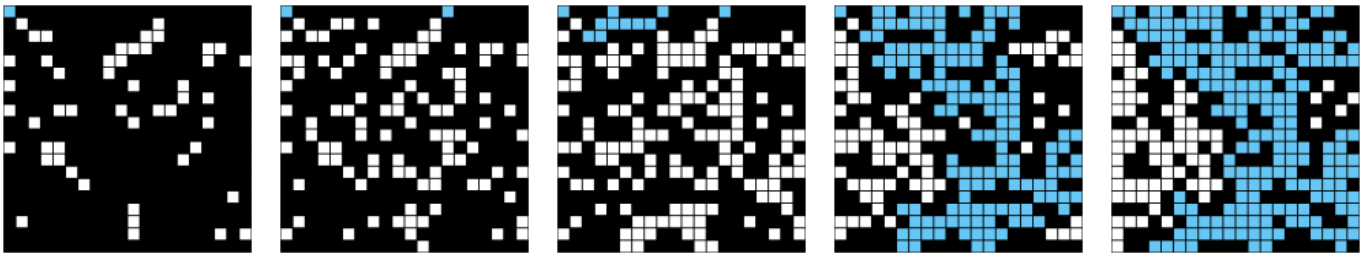
Testing. We provide two clients that serve as large-scale visual traces. We highly recommend using them for testing and debugging your `Percolation` implementation.

Visualization client. PercolationVisualizer.java animates the results of opening sites in a percolation system specified by a file by performing the following steps:

- Read the grid size *N* from the file.
- Create an *N*-by-*N* grid of sites (initially all blocked).
- Read in a sequence of sites (row *i*, column *j*) to open from the file. After each site is opened, draw full sites in light blue, open sites (that aren't full) in white, and blocked sites in black using *standard draw*, with with site (1, 1) in the upper left-hand corner.

The program should behave as in this movie and the following snapshots when used with input20.txt.

```
% java PercolationVisualizer input20.txt
```



50 open sites *100 open sites* *150 open sites* *204 open sites* *250 open sites*

Sample data files. The directory percolation contains some sample files for use with the visualization client. Associated with each input `.txt` file is an output `.png` file that contains the desired graphical output at the end of the animation. For convenience, percolation-testing.zip contains all of these files bundled together.

InteractiveVisualization client. InteractivePercolationVisualizer.java is similar to the first test client except that the input comes from a mouse (instead of from a file). It takes a command-line integer *N* that specifies the lattice size. As a bonus, it writes to standard output the sequence of sites opened in the same format used by `PercolationVisualizer`, so you can use it to prepare interesting files for testing. If you design an interesting data file, feel free to share it with us and your classmates by posting it in the discussion forums.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Consider not worrying about backwash for your first attempt.** If you're feeling overwhelmed, don't worry about backwash when following the possible progress steps below. You can revise your implementation once you have a better handle on the problem and have solved the problem without handling backwash.
2. **For each method in `Percolation` that you must implement (`open()`, `percolates()`, etc.), make a list of which `WeightedQuickUnionUF` methods might be useful for implementing that method.** This should help solidify what you're attempting to accomplish.
3. **Using the list of methods above as a guide, choose instance variables that you'll need to solve the problem.** Don't overthink this, you can always change them later. Instead, use your list of instance variables to guide your thinking as you follow the steps below, and make changes to your instance variables as you go. Hint: At minimum, you'll need to store the grid size, which sites are open, and which sites are connected to which other sites. The last of these is exactly what the union-find data structure is designed for.
4. **Plan how you're going to map from a 2-dimensional (row, column) pair to a 1-dimensional union find object index.** You will need to come up with a scheme for uniquely mapping 2D coordinates to 1D coordinates. We recommend writing a private method with a signature along the lines of `int xyTo1D(int, int)` that performs this conversion. You will need to utilize the percolation grid size when writing this method. Writing such a private method (instead of copying and pasting a conversion formula multiple times throughout your code) will greatly improve the readability and maintainability of your code. In general, we encourage you to write such modules wherever possible. Directly test this method using the main() function of Percolation.
5. **Write a private method for validating indices.** Since each method is supposed to throw an exception for invalid indices, you should write a private method which performs this validation process.
6. **Write the `open()` method and the `Percolation()` constructor.** The open() method should do three things. First, it should validate the indices of the site that it receives. Second, it should somehow mark the site as open. Third, it should perform some sequence of `WeightedQuickUnionUF` operations that links the site in question to its open neighbors. The constructor and instance variables should facilitate the `open()` method's ability to do its job.
7. **Test the `open()` method and the `Percolation()` constructor.** These tests should be in `main()`. An example of a simple test is to call `open(1, 1)` and `open(1, 2)`, and then to ensure that the two corresponding entries are connected (using `.connected()` in `WeightedQuickUnionUF`).
8. **Write the `percolates()`, `isOpen()`, and `isFull()` methods.** These should be very simple methods.
9. **Test your complete implementation using the visualization clients.**
10. **Write and test the PercolationStats class.**

Programming Tricks and Common Pitfalls

1. **Do not write your own Union-Find data structure. Use `WeightedQuickUnionUF` instead.**

2. **Your `Percolation` class should use the `WeightedQuickUnionUF` class.** If you submit with the `UF` class, your code will fail the timing tests.

3. **It's ok to use an extra row and/or column to deal with the 1-based indexing of the percolation grid.** Though it is slightly inefficient, it's fine to use arrays or union-find objects that are slightly larger than strictly necessary. Doing this results in cleaner code at the cost of slightly greater memory usage.

4. **Each of the methods (except the constructor) in `Percolation` must use a constant number of union-find operations.** If you have a for-loop inside of one of your `Percolation` methods, you're probably doing it wrong. Don't forget about the virtual-top / virtual-bottom trick described in lecture.

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 adding 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)          // add the item to the front
   public void addLast(Item item)           // add the item to the end
   public Item removeFirst()                // remove and return the item from the front
   public Item removeLast()                 // remove and return the item from the end
   public Iterator<Item> iterator()         // return an iterator over items in order from front to end
   public static void main(String[] args)   // unit testing
}
```

Corner cases. 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.

Performance requirements. Your deque implementation must 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 must support each operation (including construction) in *constant worst-case time*.

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()                    // remove and return a random item
   public Item sample()                     // return (but do not remove) a random item
   public Iterator<Item> iterator()         // return an independent iterator over items in random order
   public static void main(String[] args)   // unit testing
}
```

Corner cases. The order of two or more iterators to the same randomized queue must be *mutually independent*; each iterator must maintain its own 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.

Performance requirements. Your randomized queue implementation must 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 must support operations `next()` and `hasNext()` in *constant worst-case time*; and construction in *linear time*; you may use a linear amount of extra memory per iterator.

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 $0 \le k \le N$, where N is 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        BB
E                                               BB
F                                               CC
G                                               BB
```

The running time of `Subset` must be linear in the size of the input. You may use only a constant amount of memory plus either one `Deque` or `RandomizedQueue` object of maximum size at most *N*, where *N* is the number of strings on standard input. (For an extra challenge, use only one `Deque` or `RandomizedQueue` object of maximum size at most *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 `stdlib.jar`, `java.lang`, `java.util.Iterator`, and `java.util.NoSuchElementException`.

Programming Assignment 2 Checklist: Randomized Queues and Dequeues

Frequently Asked Questions

Should I use arrays or linked lists in my implementations? In general we don't tell you *how* to implement your data structures—you can use arrays, linked lists, or maybe even invent your own new structure provide you abide by the specified time and space requirements. So, before you begin to write the code, make sure that your data structure will achieve the required resource bounds.

How serious are you about not calling any external library function other than those in `stdlib.jar`? You will receive a substantial deduction. The goal of this assignment is to implement data types from first principles, using resizing arrays and linked lists—feel free to use java.util.LinkedList and java.util.ArrayList on future programming assignments. We also require you to use `StdIn` (instead of java.util.Scanner) because we will intercept the calls to `StdIn` in our testing.

Can I add extra public methods to the `Deque` or `RandomizedQueue` APIs? Can I use different names for the methods? No, you must implement the API exactly as specified. The only exception is the `main()` method, which you should use for unit testing.

What is meant by uniformly at random? If there are *N* items in the randomized queue, then you should choose each one with probability 1/*N*, up to the randomness of `StdRandom.uniform()`, independent of past decisions. You can generate a pseudo-random integer between 0 and *N*-1 using `StdRandom.uniform(N)` from the StdRandom.java library.

Given an array, how can I rearrange the entries in random order? Use `StdRandom.shuffle()`—it implements the Knuth shuffle discussed in lecture and runs in linear time. Note that depending on your implementation, you may not need to call this method.

What should my deque (or randomized queue) iterator do if the deque (or randomized queue) is structurally modified at any time after the iterator is created (but before it is done iterating)? You don't need to worry about this in your solution. An industrial-strength solution (used in the Java libraries) is to make the iterator *fail-fast*: throw a `java.lang.ConcurrentModificationException` as soon as this is detected.

Why does the following code lead to a `generic array creation` compile-time error when `Item` is a generic type parameter?

```
Item[] a = new Item[1];
```

Java prohibits the creation of arrays of generic types. See the Q+A in Section 1.3 for a brief discussion. Instead, use a cast.

```
Item[] a = (Item[]) new Object[1];
```

Unfortuantely, this leads to an unavoidable compiler warning.

The compiler says that my program uses unchecked or unsafe operations and to recompile with -Xlint:unchecked for details. Usually this means you did a potentially unsafe cast. When implementing a generic stack with an array, this is unavoidable since Java does not allow generic array creation. For example, the compiler outputs the following warning with ResizingArrayStack.java:

```
% javac ResizingArrayStack.java
Note: ResizingArrayStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked ResizingArrayStack.java
ResizingArrayStack.java:25: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: Item[]
        a = (Item[]) new Object[2];
                     ^
ResizingArrayStack.java:36: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: Item[]
        Item[] temp = (Item[]) new Object[capacity];
                               ^
2 warnings
```

You should not make any other casts.

Checkstyle complains that my nested class' instance variables must be private and have accessor methods that are not private. Do I need to make them private? No, but there's no harm in doing so. The access modifier of a nested class' instance variable is irrelevant—regardless of its access modifiers, it can be accessed anywhere in the file. (Of course, the enclosing class' instance variables should be private.)

Can a nested class have a constructor? Yes.

What assumptions can I make about the input to `Subset`? Standard input can contain any sequence of strings. You may assume that there is one integer command-line argument *k* and it is between 0 and the number of strings on standard input.

Will I lose points for loitering? Yes. Loitering is maintaining a useless reference to an object that could otherwise be garbage collected.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps. These same steps apply to each of the two data types that you will be implementing.

1. **Make sure you understand the performance requirements for both `Deque` and `RandomizedQueue`.** They are summarized in the table below. *Every detail in these performance requirements is important. Do not proceed until you understand them.*

	Deque	Randomized Queue
Non-iterator operations	Constant worst-case time	Constant amortized time
Iterator constructor	Constant worst-case time	linear in current # of items
Other iterator operations	Constant worst-case time	Constant worst-case time
Non-iterator memory use	Linear in current # of items	Linear in current # of items

Memory per iterator	Constant	Linear in current # of items

2. **Decide whether you want to use an array, linked list, or your own class.** This choice should be made based on the performance requirements discussed above. You may make different choices for `Deque` and `RandomizedQueue`. You might start by considering why a resizing array does not support *constant worst-case* time operations in a stack.

3. **Use our example programs as a guide when implementing your methods.** There are many new ideas in this programming assignment, including resizing arrays, linked lists, iterators, the *foreach* keyword, and generics. If you are not familiar with these topics, our example code should make things much easier. ResizingArrayStack.java uses a resizing array; LinkedStack.java uses a singly-linked list. Both examples use iterators, foreach, and generics.

4. **We strongly recommend that you develop unit tests for your code as soon as you've written enough methods to allow for testing.** As an example for `Deque`, you know that if you call `addFirst()` with the numbers 1 through N in ascending order, then call `removeLast()` N times, you should see the numbers 1 through N in ascending order. As soon as you have those two methods written, you can write a unit test for these methods. Arguably even better are randomized unit tests (which we employ heavily in our correctness testing). We recommend that you create a client class with a name like `TestDeque`, where each unit test is a method in this class. Don't forget to test your iterator.

Programming Tricks and Common Pitfalls

1. **It is very important that you carefully plan your implementation before you begin.** In particular, for each data structure that you're implementing (`RandomizedQueue` and `Deque`), you must decide whether to use a linked list, an array, or something else. If you make the wrong choice, you will not achieve the peformance requirements and you will have to abandon your code and start over.

2. **Make sure that your memory use is linear in the current number of items, as opposed to the greatest number of items that has ever been in the data structure since its instantiation.** If you're using a resizing array, you must resize the array when it becomes sufficiently empty. You must also take care to avoid loitering anytime you remove an item.

3. **Make sure to test what happens when your data structures are emptied.** One very common bug is for something to go wrong when your data structure goes from non-empty to empty and then back to non-empty. Make sure to include this in your tests.

4. **Make sure to test that multiple iterators can be used simultaneously.** You can test this with a nested *foreach* loop. The iterators should operate independently of one another.

5. **Don't rely on our automated tests for debugging.** You don't have access to the source code of our testing suite, so the *Assessment Details* may be hard to utilize for debugging. As suggested above, write your own unit tests; it's good practice.

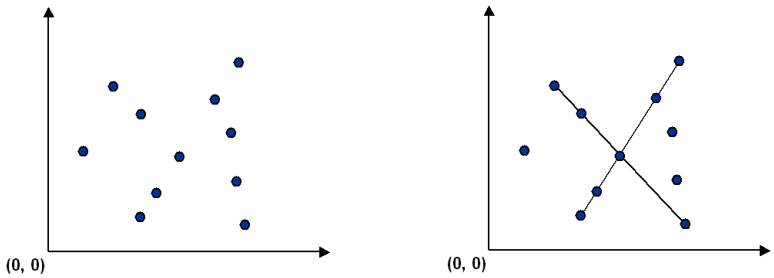
6. **If you use a linked list, consider using a sentinel node (or nodes).** Sentinel nodes can simplify your code and prevent bugs. However, they are not required (and we have not provided examples that use sentinel nodes).

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; 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.

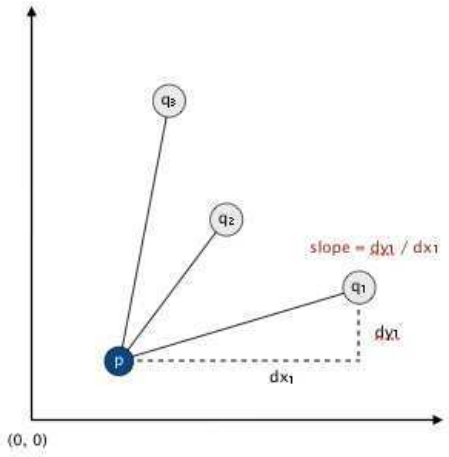
The order of growth of the running time of your program should be N^4 in the worst case and it should use space proportional to N.

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. 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 should 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.

Programming Assignment 3 Checklist: Pattern Recognition

Frequently Asked Questions

Can the same point appear more than once as input to Brute or Fast? You may assume the input to `Brute` and `Fast` are *N* distinct points. Nevertheless, the methods implemented as part of the `Point` data type must correctly handle the case when the points are not distinct: for the `slopeTo()` method, this requirement is explicitly stated in the API; for the comparison methods, this requirement is implict in the contracts for `Comparable` and `Comparator`.

The reference solution outputs a line segment in the order *p*→*q*→*r*→*s* but my solution outputs it in the reverse order *s*→*r*→*q*→*p*. Is that ok? Yes, there are two valid ways to output a line segment.

The reference solution outputs the line segments in a different order than my solution. Is that ok? Yes, if there are *k* line segments, then there are *k*! different possible ways to output them.

Can I draw a line segment containing 4 (or more) points by drawing several subsegments? No, you should draw one (and only one) line segment for each set of collinear points discovered: For example, you should draw the line segment *p*→*q*→*r*→*s*, with either `p.drawTo(s)` or `s.drawTo(p)`.

How do I sort a subarray? `Arrays.sort(a, lo, hi)` sorts the subarray from `a[lo]` to `a[hi-1]` according to the natural order of `a[]`. You can use a `Comparator` as the fourth argument to sort according to an alternate order.

Where can I see examples of Comparable and Comparator? See the lecture slides. We assume this is new Java material for most of you, so don't hesitate to ask for clarifications in the Discussion Forums.

My program fails only on (some) vertical line segments. What could be going wrong? Are you dividing by zero? With integers, this produces a runtime exception. With floating-point numbers, 1.0/0.0 is positive infinity and -1.0/0.0 is negative infinity. You may also use the constants `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`.

What does it mean for slopeTo() to return positive zero? Java (and the IEEE 754 floating-point standard) define two representations of zero: negative zero and positive zero.

```
double a = 1.0;
double x = (a - a) /  a;   // positive zero ( 0.0)
double y = (a - a) / -a;   // negative zero (-0.0)
```

Note that while `(x == y)` is guaranteed to be true, Arrays.sort() treats negative zero as strictly less than positive zero. Thus, to make the specification precise, we require you to return positive zero for horizontal line segments. Unless your program casts to the wrapper type `Double` (either explicitly or via autoboxing), you probably will not notice any difference in behavior; but, if your program does cast to the wrapper type and fails only on (some) horizontal line segments, this may be the cause.

Is it ok to compare two floating-point numbers a and b for exactly equality? In general, it is hazardous to compare a and b for equality if either is susceptible to floating-point roundoff error. However, in this case, you are computing b/a, where a and b are integers between -32,767 and 32,767. In Java (and the IEEE 754 floating-point standard), the result of a floating-point operation (such as division) is the nearest representable

value. Thus, for example, it is guaranteed that `(9.0/7.0 == 45.0/35.0)`. In other words, it's sometimes OK to compare floating-point numbers for exact equality (but only when you know exactly what you are doing!)

Note also that it is possible to implement `compare()` and `Fast` using only integer arithmetic, though you are not required to do so.

I'm having trouble avoiding subsegments Fast.java when there are 5 or more points on a line segment. Any advice? Not handling the 5-or-more case is a bit tricky, so don't kill yourself over it.

I created a nested Comparator class within Point. Within the nested Comparator class, the keyword `this` refers to the Comparator object. How do I refer to the Point instance of the outer class? Use `Point.this` instead of `this`. Note that you can refer directly to instance methods of the outer class (such as `slopeTo()`); with proper design, you shouldn't need this awkward notation.

Testing

Sample data files. The directory collinear contains some sample input files in the specified format. Assoicated with some of the input .txt files are output .png files that contains the desired graphical output. For convenience, collinear-testing.zip contains all of these files bundled together. Thanks to Jesse Levinson '05 for the remarkable input file `rs1423.txt`; feel free to create your own and share with us in the Discussion Forums.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Getting started.** Download Point.java and PointPlotter.java. The latter takes a command-line argument, reads in a list of points from the file specified as a command-line argument (in the format specified) and plots the results using standard draw. To plot the points, type the following at the command line

   ```
   % java PointPlotter input56.txt
   ```

2. **Slope.** To begin, implement the `slopeTo()` method. Be sure to consider a variety of corner cases, including horizontal, vertical, and degenerate line segments.

3. **Brute force algorithm.** Write code to iterate through all 4-tuples and check if the 4 points are collinear. To draw the line segment, you need to know the endpoints. One approach is to print out a line segment only if the 4 points are in ascending order (say, relative to the natural order), in which case, the endpoints are the first and last points.

   Hint: don't waste time micro-optimizing the brute-force solution. Though, if you really want to, there are two easy opportunities. First, you can iterate through all combinations of 4 points (N choose 4) instead of all 4 tuples (N^4), saving a factor of 4! = 24. Second, you don't need to consider whether 4 points are collinear if you already know that the first 3 are not collinear; this can save you a factor of N on typical inputs.

4. **Fast algorithm.**

- Implement the `SLOPE_ORDER` comparator in `Point`. The complicating issue is that the comparator needed to compare the slopes that two points `q` and `r` make with a third point `p`, which changes from sort to sort. To do this include a `public` and `final` (but not `static`) instance variable `SLOPE_ORDER` in `Point` of type `Comparator<Point>`. This `Comparator` has a `compare()` method so that `compare(q, r)` compares the slopes that `q` and `r` make with the invoking object `p`.

- Implement the sorting solution. Watch out for corner cases. Don't worry about 5 or more points on a line segment yet.

Enrichment

Can the problem be solved in quadratic time and linear space? Yes, but the only compare-based algorithm I know of that guarantees quadratic time in the worst case is quite sophisticated. It involves converting the points to their dual line segments and topologically sweeping the arrangement of lines by Edelsbrunner and Guibas.

Can the decision version of the problem be solved in subquadratic time? The original version of the problem cannot be solved in subquadratic time because there might be a quadratic number of line segments to output. (See next question.) The decision version asks whether there exists a set of 4 collinear points. This version of the problem belongs to a group of problems that are known as 3SUM-hard. A famous unresolved conjecture is that such problems have no subquadratic algorithms. Thus, the sorting algorithm presented above is about the best we can hope for (unless the conjecture is wrong). Under a restricted decision tree model of computation, Erickson proved that the conjecture is true.

What's the maximum number of (maximal) collinear sets of points in a set of N points in the plane? It can grow quadratically as a function of N. Consider the N points of the form: (x, y) for $x = 0, 1, 2$, and 3 and $y = 0, 1, 2, ..., N / 4$. This means that if you store all of the (maximal) collinear sets of points, you will need quadratic space in the worst case.

COS 226 Programming Assignment

8 Puzzle

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A* search algorithm.

The problem. The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks horizontally or vertically into the blank square. The following shows a sequence of legal moves from an *initial board* (left) to the *goal board* (right).

```
   1  3       1     3       1  2  3       1  2  3       1  2  3
4  2  5   =>  4  2  5   =>  4     5   =>  4  5     =>  4  5  6
7  8  6       7  8  6       7  8  6       7  8  6       7  8

initial       1 left         2 up          5 left         goal
```

Best-first search. Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the A* search algorithm. We define a *search node* of the game to be a board, the number of moves made to reach the board, and the previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to a goal board. The success of this approach hinges on the choice of *priority function* for a search node. We consider two priority functions:

- *Hamming priority function.* The number of blocks in the wrong position, plus the number of moves made so far to get to the search node. Intuitively, a search node with a small number of blocks in the wrong position is close to the goal, and we prefer a search node that have been reached using a small number of moves.
- *Manhattan priority function.* The sum of the Manhattan distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the search node.

For example, the Hamming and Manhattan priorities of the initial search node below are 5 and 10, respectively.

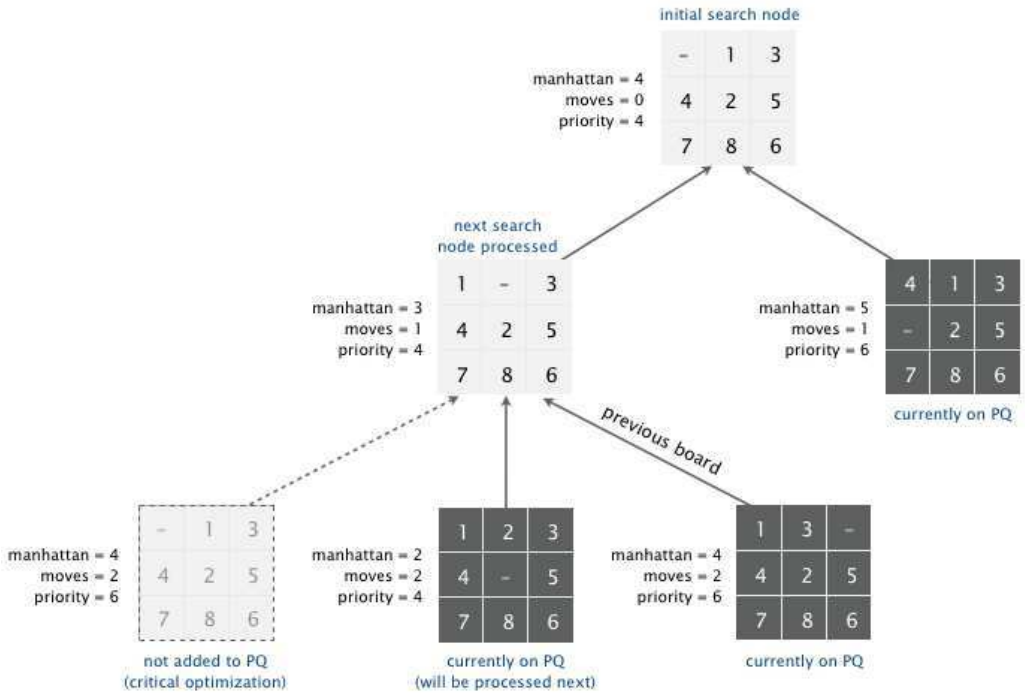
```
8  1  3       1  2  3     1  2  3  4  5  6  7  8     1  2  3  4  5  6  7  8
4     2       4  5  6     ----------------------     ----------------------
7  6  5       7  8        1  1  0  0  1  1  0  1     1  2  0  0  2  2  0  3

initial          goal          Hamming = 5 + 0           Manhattan = 10 + 0
```

We make a key observation: To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Hamming or Manhattan priorities.) Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

A critical optimization. Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the previous search node.

```
8  1  3       8  1  3       8  1       8  1  3     8  1  3
4     2       4  2          4  2  3    4     2     4  2  5
7  6  5       7  6  5       7  6  5    7  6  5     7  6

previous    search node    neighbor   neighbor    neighbor
                                     (disallow)
```

Game tree. One way to view the computation is as a *game tree*, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a priority queue; at each step, the A* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).



Detecting unsolvable puzzles. Not all initial boards can lead to the goal board by a sequence of legal moves, including the two below:

```
 1  2  3          1  2  3  4
 4  5  6          5  6  7  8
 8  7             9 10 11 12
                 13 15 14
unsolvable
                 unsolvable
```

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal board and (ii) those that lead to the goal board if we modify the initial board by swapping any pair of adjacent (non-blank) blocks in the same row. (Difficult challenge for the mathematically inclined: prove this fact.) To apply the fact, run the A* algorithm simultaneously on two puzzle instances—one with the initial board and one with the initial board modified by swapping a pair of adjacent blocks in the same row. Exactly one of the two will lead to the goal board.

Board and Solver data types. Organize your program by creating an immutable data type `Board` with the following API:

```
public class Board {
    public Board(int[][] blocks)           // construct a board from an N-by-N array of blocks
                                           // (where blocks[i][j] = block in row i, column j)
    public int dimension()                 // board dimension N
    public int hamming()                   // number of blocks out of place
    public int manhattan()                 // sum of Manhattan distances between blocks and goal
    public boolean isGoal()                // is this board the goal board?
    public Board twin()                    // a board that is obtained by exchanging two adjacent blocks in the same row
    public boolean equals(Object y)        // does this board equal y?
    public Iterable<Board> neighbors()     // all neighboring boards
    public String toString()               // string representation of this board (in the output format specified below)

    public static void main(String[] args) // unit tests (not graded)
}
```

Corner cases. You may assume that the constructor receives an N-by-N array containing the N^2 integers between 0 and $N^2 - 1$, where 0 represents the blank square.

Performance requirements. Your implementation should support all `Board` methods in time proportional to N^2 (or better) in the worst case.

Also, create an immutable data type `Solver` with the following API:

```
public class Solver {
    public Solver(Board initial)           // find a solution to the initial board (using the A* algorithm)
    public boolean isSolvable()            // is the initial board solvable?
    public int moves()                     // min number of moves to solve initial board; -1 if unsolvable
    public Iterable<Board> solution()      // sequence of boards in a shortest solution; null if unsolvable
    public static void main(String[] args) // solve a slider puzzle (given below)
}
```

To implement the A* algorithm, *you must use the `MinPQ` data type from `algs4.jar` for the priority queue(s).*

Corner cases. The constructor should throw a `java.lang.NullPointerException` if passed a null argument.

Solver test client. Use the following test client to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output.

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

    // create initial board from file
    In in = new In(args[0]);
    int N = in.readInt();
    int[][] blocks = new int[N][N];
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            blocks[i][j] = in.readInt();
    Board initial = new Board(blocks);

    // solve the puzzle
    Solver solver = new Solver(initial);

    // print solution to standard output
    if (!solver.isSolvable())
        StdOut.println("No solution possible");
    else {
        StdOut.println("Minimum number of moves = " + solver.moves());
        for (Board board : solver.solution())
            StdOut.println(board);
    }
}
```

Input and output formats. The input and output format for a board is the board dimension *N* followed by the *N*-by-*N* initial board, using 0 to represent the blank square. As an example,

```
% more puzzle04.txt
3
 0  1  3
 4  2  5
 7  8  6

% java Solver puzzle04.txt
Minimum number of moves = 4

3
 0  1  3
 4  2  5
 7  8  6

3
 1  0  3
 4  2  5
 7  8  6

3
 1  2  3
 4  0  5
 7  8  6

3
 1  2  3
 4  5  0
 7  8  6

3
 1  2  3
 4  5  6
 7  8  0

% more puzzle3x3-unsolvable.txt
3
 1  2  3
 4  5  6
 8  7  0

% java Solver puzzle3x3-unsolvable.txt
No solution possible
```

Your program should work correctly for arbitrary *N*-by-*N* boards (for any $2 \le N < 128$), even if it is too slow to solve some of them in a reasonable amount of time.

Deliverables. Submit the files `Board.java` and `Solver.java` (with the Manhattan priority). 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`. You must use the `MinPQ` data type from `algs4.jar` for the priority queue(s).

Programming Assignment 4 Checklist: 8 Puzzle

Frequently Asked Questions

Can I use different class names, method names, or method signatures from those prescribed in the API? No, as usual, your assignment will not be graded if it violates the API.

Is 0 a block? No, 0 represents the blank square. Do not treat it as a block when computing either the Hamming or Manhattan priority functions.

Can I assume that the puzzle inputs (arguments to the Board constructor and input to Solver) are valid? Yes, though it never hurts to include some basic error checking.

Do I have to implement my own stack, queue, and priority queue? You must use either `MinPQ` or `MaxPQ` for your priority queue (because we will intercept calls in order to do performance analysis). For the other data types, you may use versions from either `algs4.jar` or `java.util`.

How do I return an `Iterable<Board>`? Add the items you want to a `Stack<Board>` or `Queue<Board>` and return that.

How do I implement `equals()`? Java has some arcane rules for implementing `equals()`, discussed on p. 103 of Algorithms, 4th edition. Note that the argument to `equals()` is required to be `Object`. You can also inspect Date.java or Transaction.java for online examples.

Must I implement the `toString()` method for `Board` exactly as specified? Yes. Be sure to include the board dimension and use 0 for the blank square. Use `String.format()` to format strings—it works like `StdOut.printf()`, but returns the string instead of printing it to standard output. For reference, our implementation is below, but yours may vary depending on your choice of instance variables.

```
public String toString() {
    StringBuilder s = new StringBuilder();
    s.append(N + "\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            s.append(String.format("%2d ", tiles[i][j]));
        }
        s.append("\n");
    }
    return s.toString();
}
```

Should the `hamming()` and `manhattan()` methods in `Board` return the Hamming and Manhattan priority functions, respectively? No, `hamming()` should return the number of blocks out of position and `manhattan()` should return the sum of the Manhattan distances between the blocks and their goal positions. Recall that the blank square is not considered a block. You will compute the priority function in `Solver` by calling `hamming()` or `manhattan()` and adding to it the number of moves.

I'm a bit confused about the purpose of the twin() method. You will use it to determine whether a puzzle is solvable: exactly one of a board and its twin are solvable. A twin is obtained by swapping two adjacent blocks (the blank does not count) in the same row. For example, here is a board and its 5 possible twins. Your solver will use only one twin.

```
   1  3       3  1       1  3       1  3       1  3       1  3
 4  2  5    4  2  5    2  4  5    4  5  2    4  2  5    4  2  5
 7  8  6    7  8  6    7  8  6    7  8  6    8  7  6    7  6  8

  board      twin       twin       twin       twin       twin
```

How do I reconstruct the solution once I've dequeued the goal search node? Since each search node records the previous search node to get there, you can chase the pointers all the way back to the initial search node (and consider them in reverse order).

Can I terminate the search as soon as a goal search node is enqueued (instead of dequeued)? No, even though it does lead to a correct solution for the slider puzzle problem using the Hamming and Manhattan priority functions, it's not techincally the A* algorithm (and will not find the correct solution for other problems and other priority functions).

I noticed that the priorities of the search nodes dequeued from the priority queue never decrease. Is this a property of the A* algorithm? Yes. In the language of the A* algorithm, the Hamming and Manhattan distances (before adding in the number of moves so far) are known as *heuristics*. If a heuristic is both *admissible* (never overestimates the number of moves to the goal search node) and *consistent* (satisfies a certain triangle inequality), then this noticed property is guaranteed. The Hamming and Manhattan heuristics are both admissible and consistent. You may use this noticed property as a debugging clue: if the priority of the search node dequeued from the priority queue decreases, then you know you did something wrong.

Even with the critical optimization, the priority queue may contain two or more search nodes corresponding to the same board. Should I try to eliminate these? In principle, you could do so with a set data type such as `SET` in `algs4.jar` or `java.util.TreeSet` or `java.util.HashSet` (provided that the `Board` data type were either `Comparable` or had a `hashCode()` method). However, almost all of the benefit from avoiding duplicate boards is already extracted from the critical optimization and the cost of identifying other duplicate boards will

be more than the remaining benefit from doing so.

Can I put the logic for detecting whether a puzzle is infeasible in Board instead of Solver? There is a elegant algorithm for determining whether a board is solvable that relies on a parity argument (and occasionally we change our API to require this solution). However, the current API requires you to detect infeasiblity in `Solver` by using two synchronized A* searches (e.g., using two priority queues).

I run out of memory when running some of the large sample puzzles. What should I do? Be sure to ask Java for additional memory, e.g., `java -Xmx1600m Solver puzzle36.txt`. We recommend running from the command line (and not from the DrJava interaction pane). You should expect to run out of memory when using the Hamming priority function. Be sure not to put the JVM option in the wrong spot or it will be treated as a command-line argument, e.g., `java Solver -Xmx1600m puzzle36.txt`.

My program can't solve puzzle4x4-hard1.txt or puzzle4x4-hard2.txt, even if I give it a huge amount of space. What am I doing wrong? Probably nothing. The A* algorithm will struggle to solve even some 4-by-4 instances.

Testing

Input files. The directory 8puzzle contains many sample puzzle input files. For convenience, 8puzzle-testing.zip contains all of these files bundled together.

- The shortest solution to `puzzle[T].txt` requires exactly *T* moves.
- The shortest solution to `puzzle4x4-hard1.txt` and `puzzle4x4-hard2.txt` are 38 and 47, respectively.
- Warning: `puzzle36.txt` is especially difficult.

Test client. A good way to automatically run your program on our sample puzzles is to use the client PuzzleChecker.java.

Priority queue trace.

- Here are the contents of our priority queue (sorted by priority) just before dequeueing each node when using the Manhattan priority function on `puzzle04.txt`.

```
Step 0:    priority  = 4
           moves     = 0
           manhattan = 4
           3
            0  1  3
            4  2  5
            7  8  6

Step 1:    priority  = 4    priority  = 6
           moves     = 1    moves     = 1
           manhattan = 3    manhattan = 5
           3                3
            1  0  3          4  1  3
            4  2  5          0  2  5
            7  8  6          7  8  6

Step 2:    priority  = 4    priority  = 6    priority  = 6
           moves     = 2    moves     = 1    moves     = 2
           manhattan = 2    manhattan = 5    manhattan = 4
           3                3                3
            1  2  3          4  1  3          1  3  0
            4  0  5          0  2  5          4  2  5
            7  8  6          7  8  6          7  8  6

Step 3:    priority  = 4    priority  = 6    priority  = 6    priority  = 6    priority  = 6
           moves     = 3    moves     = 3    moves     = 2    moves     = 3    moves     = 1
           manhattan = 1    manhattan = 3    manhattan = 4    manhattan = 3    manhattan = 5
           3                3                3                3                3
            1  2  3          1  2  3          1  3  0          1  2  3          4  1  3
            4  5  0          4  8  5          4  2  5          0  4  5          0  2  5
            7  8  6          7  0  6          7  8  6          7  8  6          7  8  6

Step 4:    priority  = 4    priority  = 6    priority  = 6    priority  = 6    priority  = 6    priority  = 6
           moves     = 4    moves     = 3    moves     = 4    moves     = 2    moves     = 3    moves     = 1
           manhattan = 0    manhattan = 3    manhattan = 2    manhattan = 4    manhattan = 3    manhattan = 5
           3                3                3                3                3                3
            1  2  3          1  2  3          1  2  0          1  3  0          1  2  3          4  1  3
            4  5  6          0  4  5          4  5  3          4  2  5          4  8  5          0  2  5
            7  8  0          7  8  6          7  8  6          7  8  6          7  0  6          7  8  6
```

There were a total of 10 search nodes enqueued and 5 search nodes dequeued. In general, the number of search nodes enqueued and dequeued may vary slightly, depending the order in which the search nodes with equal priorities come off the priority queue, which

depends on the order in which `neighbors()` returns the neighbors of a board. However, for this input, there are no such ties, so you should have exactly 10 search nodes enqueued and 5 search nodes dequeued.

- The contents of our priority queue (sorted by priority) just before dequeueing each node when using the Hamming priority function on `puzzle04.txt` turns out to be identical to the results above: for this input file, throughout the A* algorithm, a block is never more than one position away from its goal position, which implies that the Hamming function and the Manhattan functions are equal.

- Write the class `Solver` that uses the A* algorithm to solve puzzle instances. -->

Enrichment

How can I reduce the amount of memory a `Board` uses? For starters, recall that an N-by-N `int[][]` array in Java uses about 24 + 32N + 4N^2 bytes; when N equals 3, this is 156 bytes. To save memory, consider using an *N*-by-*N* `char[][]` array or a length *N*^2 `char[]` array. In principle, each board is a permutation of size N^2, so you need only about lg ((N^2)!) bits to represent it; when N equals 3, this is only 19 bits.

Any ways to speed up the algorithm? Yes there are many opportunities for optimization here.

- Use a 1d array instead of a 2d array (as suggested above).
- Cache either the Manhattan distance of a board (or Manhattan priority of a search node). It is waste to recompute the same quantity over and over again.
- Exploit the fact that the difference in Manhattan distance between a board and a neighbor is either +1, -1, or 0, based on the direction that the block moves.
- Use only one PQ to run the A* algorithm on the initial board and its twin.
- When two search nodes have the same Manhattan priority, you can break ties however you want, e.g., by comparing either the Hamming or Manhattan distances of the two boards.
- Use a parity argument to determine whether a puzzle is unsolvable (instead of two synchronous A* searches). However, this will either break the API or will require a fragile dependence on the `toString()` method, so don't do it.

Is there an efficient way to solve the 8-puzzle and its generalizations? Finding a shortest solution to an *N*-by-*N* slider puzzle is NP-hard, so it's unlikely that an efficient solution exists.

What if I'm satisfied with any solution and don't need one that uses the fewest number of moves? Yes, change the priority function to put more weight on the Manhattan distance, e.g., 100 times the Manhattan distance plus the number of moves made already. This paper describes an algorithm that guarantees to perform at most N^3 moves.

Are ther better ways to solve 8- and 15-puzzle instances using the minimum number of moves? Yes, there are a number of approaches.

- Use the A* algorithm with a better admissible priority function:
  - *Linear conflict*: add two to the Manhattan priority function whenever two tiles are in their goal row (or column) but are reversed relative to their goal position.
  - *Pattern database*: For each possible configuration of 4 tiles and the blank, determine the minimum number of moves to put just these tiles in their proper position and store these values in a database. The heuristic value is the maximum over all configurations, plus the number of moves made so far. This can reduce the number of search nodes examined for random 15-puzzle instances by a factor of 1000.
- Use a variant of the A* algorithm known as IDA* (for iterative deepening). This paper describes its application to the 15-slider puzzle.
- Another approach is to use bidirectional search, where you simultaneously search from the initial board to find the goal board and from the goal board to find the initial board, and have the two search trees meet in the middle. Handling the stopping condition is delicate.

Can a puzzle have more than one shortest solution? Yes. See `puzzle07.txt`.

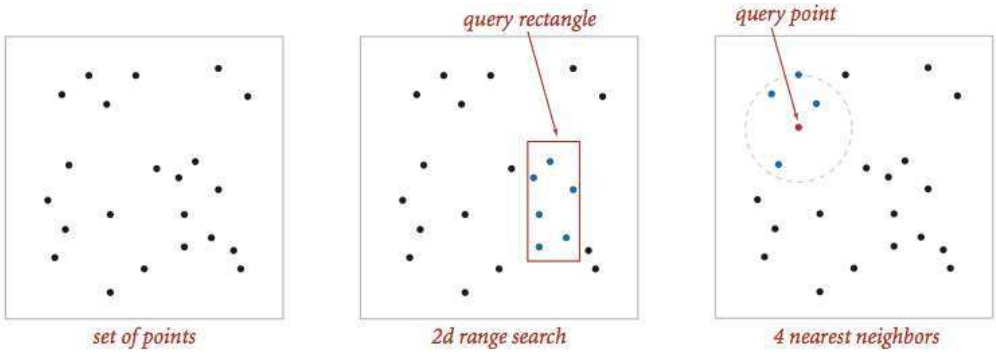
```
Solution 1
--------------------------------------------------------------------------------
1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3
   7  6    7     6    7  4  6    7  4  6       4  6    4     6    4  5  6    4  5  6
5  4  8    5  4  8    5     8       5  8    7  5  8    7  5  8    7     8    7  8

Solution 2
--------------------------------------------------------------------------------
1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3    1  2  3
   7  6    5  7  6    5  7  6    5     6       5  6    4  5  6    4  5  6    4  5  6
```

```
5  4  8        4  8     4      8     4  7  8     4  7  8        7  8     7      8     7  8
```

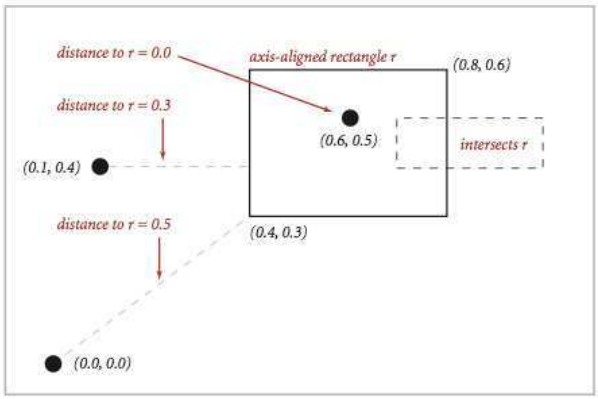
In such cases, you are required to output any one such solution.

Programming Assignment 5: Kd-Trees

Write a data type to represent a set of points in the unit square (all points have *x*- and *y*-coordinates between 0 and 1) using a *2d-tree* to support efficient *range search* (find all of the points contained in a query rectangle) and *nearest neighbor search* (find a closest point to a query point). 2d-trees have numerous applications, ranging from classifying astronomical objects to computer animation to speeding up neural networks to mining data to image retrieval.



Geometric primitives. To get started, use the following geometric primitives for points and axis-aligned rectangles in the plane.



Use the immutable data type Point2D.java (part of `algs4.jar`) for points in the plane. Here is the subset of its API that you may use:

```
public class Point2D implements Comparable<Point2D> {
   public Point2D(double x, double y)             // construct the point (x, y)
   public  double x()                             // x-coordinate
   public  double y()                             // y-coordinate
   public  double distanceTo(Point2D that)        // Euclidean distance between two points
   public  double distanceSquaredTo(Point2D that) // square of Euclidean distance between two points
   public     int compareTo(Point2D that)         // for use in an ordered symbol table
   public boolean equals(Object that)             // does this point equal that object?
   public    void draw()                          // draw to standard draw
   public  String toString()                      // string representation
}
```

Use the immutable data type RectHV.java (part of `algs4.jar`) for axis-aligned rectanges. Here is the subset of its API that you may use:

```
public class RectHV {
   public    RectHV(double xmin, double ymin,     // construct the rectangle [xmin, xmax] x [ymin, ymax]
                    double xmax, double ymax)     // throw a java.lang.IllegalArgumentException if (xmin > xmax) or (ymin > ymax)
   public  double xmin()                          // minimum x-coordinate of rectangle
   public  double ymin()                          // minimum y-coordinate of rectangle
   public  double xmax()                          // maximum x-coordinate of rectangle
   public  double ymax()                          // maximum y-coordinate of rectangle
   public boolean contains(Point2D p)             // does this rectangle contain the point p (either inside or on boundary)?
   public boolean intersects(RectHV that)         // does this rectangle intersect that rectangle (at one or more points)?
   public  double distanceTo(Point2D p)           // Euclidean distance from point p to closest point in rectangle
   public  double distanceSquaredTo(Point2D p)    // square of Euclidean distance from point p to closest point in rectangle
   public boolean equals(Object that)             // does this rectangle equal that object?
   public    void draw()                          // draw to standard draw
   public  String toString()                      // string representation
}
```

Do not modify these data types.

Brute-force implementation. Write a mutable data type `PointSET.java` that represents a set of points in the unit square. Implement the following API by using a red-black BST (using either `SET` from `algs4.jar` or `java.util.TreeSet`).

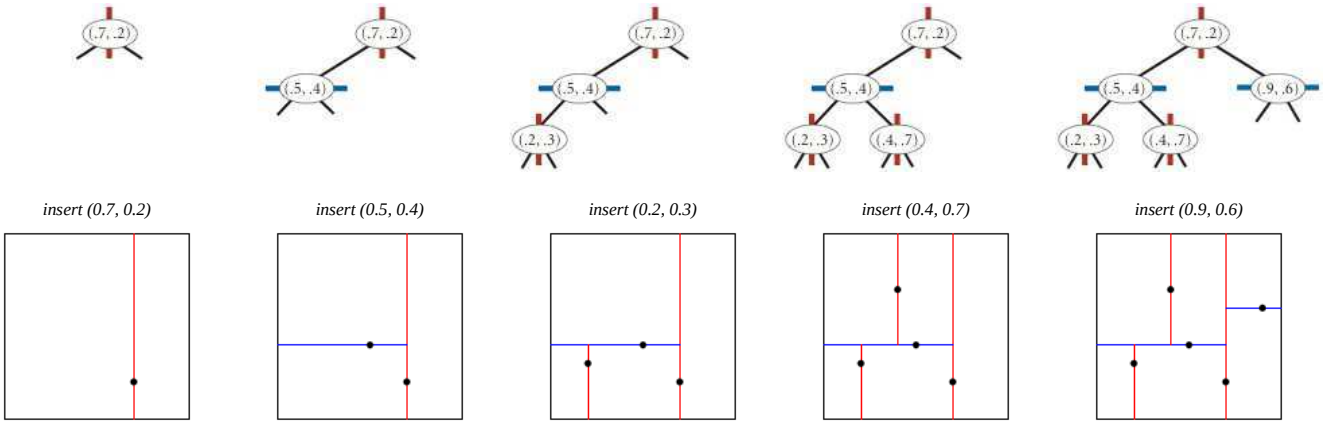
```
public class PointSET {
   public         PointSET()                               // construct an empty set of points
```

```
   public           boolean isEmpty()                    // is the set empty?
   public               int size()                       // number of points in the set
   public              void insert(Point2D p)            // add the point to the set (if it is not already in the set)
   public           boolean contains(Point2D p)          // does the set contain point p?
   public              void draw()                       // draw all points to standard draw
   public Iterable<Point2D> range(RectHV rect)           // all points that are inside the rectangle
   public           Point2D nearest(Point2D p)           // a nearest neighbor in the set to point p; null if the set is empty

   public static void main(String[] args)                // unit testing of the methods (optional)
}
```

Corner cases. Throw a `java.lang.NullPointerException` if any argument is null. *Performance requirements.* Your implementation should support `insert()` and `contains()` in time proportional to the logarithm of the number of points in the set in the worst case; it should support `nearest()` and `range()` in time proportional to the number of points in the set.

2d-tree implementation. Write a mutable data type `KdTree.java` that uses a 2d-tree to implement the same API (but replace `PointSET` with `KdTree`). A *2d-tree* is a generalization of a BST to two-dimensional keys. The idea is to build a BST with points in the nodes, using the *x*- and *y*-coordinates of the points as keys in strictly alternating sequence.

- *Search and insert.* The algorithms for search and insert are similar to those for BSTs, but at the root we use the *x*-coordinate (if the point to be inserted has a smaller *x*-coordinate than the point at the root, go left; otherwise go right); then at the next level, we use the *y*-coordinate (if the point to be inserted has a smaller *y*-coordinate than the point in the node, go left; otherwise go right); then at the next level the *x*-coordinate, and so forth.



- *Draw.* A 2d-tree divides the unit square in a simple way: all the points to the left of the root go in the left subtree; all those to the right go in the right subtree; and so forth, recursively. Your `draw()` method should draw all of the points to standard draw in black and the subdivisions in red (for vertical splits) and blue (for horizontal splits). This method need not be efficient—it is primarily for debugging.

The prime advantage of a 2d-tree over a BST is that it supports efficient implementation of range search and nearest neighbor search. Each node corresponds to an axis-aligned rectangle in the unit square, which encloses all of the points in its subtree. The root corresponds to the unit square; the left and right children of the root corresponds to the two rectangles split by the *x*-coordinate of the point at the root; and so forth.

- *Range search.* To find all points contained in a given query rectangle, start at the root and recursively search for points in *both* subtrees using the following *pruning rule*: if the query rectangle does not intersect the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). A subtree is searched only if it might contain a point contained in the query rectangle.

- *Nearest neighbor search.* To find a closest point to a given query point, start at the root and recursively search in *both* subtrees using the following *pruning rule*: if the closest point discovered so far is closer than the distance between the query point and the rectangle corresponding to a node, there is no need to explore that node (or its subtrees). That is, a node is searched only if it might contain a point that is closer than the best one found so far. The effectiveness of the pruning rule depends on quickly finding a nearby point. To do this, organize your recursive method so that when there are two possible subtrees to go down, you always choose *the subtree that is on the same side of the splitting line as the query point* as the first subtree to explore—the closest point found while exploring the first subtree may enable pruning of the second subtree.

Clients. You may use the following interactive client programs to test and debug your code.

- KdTreeVisualizer.java computes and draws the 2d-tree that results from the sequence of points clicked by the user in the standard drawing window.

- RangeSearchVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs range searches on the axis-aligned rectangles dragged by the user in the standard drawing window.

- NearestNeighborVisualizer.java reads a sequence of points from a file (specified as a command-line argument) and inserts those points into a 2d-tree. Then, it performs nearest neighbor queries on the point corresponding to the location of the mouse in the standard drawing window.

Analysis of running time and memory usage (optional and not graded).

- Give the total memory usage in bytes (using tilde notation) of your 2d-tree data structure as a function of the number of points *N*, using the memory-cost model from lecture and Section 1.4 of the textbook. Count all memory that is used by your 2d-tree, including memory for the nodes, points, and rectangles.

- Give the expected running time in seconds (using tilde notation) to build a 2d-tree on *N* random points in the unit square. (Do not count the time to read in the points from standard input.)

- How many nearest neighbor calculations can your 2d-tree implementation perform per second for input100K.txt (100,000 points) and input1M.txt (1 million points), where the query points are random points in the unit square? (Do not count the time to read in the points or to build the 2d-tree.) Repeat this question but with the brute-force implementation.

Submission. Submit only `PointSET.java` and and `KdTree.java`. We will supply `RectHV.java`, `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.

Programming Assignment 5 Checklist: Kd-Trees

Frequently Asked Questions

What should I do if a point has the same x-coordinate as the point in a node when inserting / searching in a 2d-tree? Go the right subtree as specified.

Can I assume that all x- or y-coordinates of points inserted into the `KdTree` will be between 0 and 1? Yes. You may also assume that the `insert()`, `contains()`, and `nearest()` methods in `KdTree` are passed points with x- and y-coordinates between 0 and 1. You may also assume that the `range()` method in `KdTree` is passed a rectangle that lies in the unit box.

What should I do if a point is inserted twice in the data structure? The data structure represents a *set* of points, so you should keep only one copy.

How should I scale the coordinate system when drawing? Don't, please keep the default range of 0 to 1.

How should I set the size and color of the points and rectangles when drawing? Use `StdDraw.setPenColor(StdDraw.BLACK)` and `StdDraw.setPenRadius(.01)` before before drawing the points; use `StdDraw.setPenColor(StdDraw.RED)` or `StdDraw.setPenColor(StdDraw.BLUE)` and `StdDraw.setPenRadius()` before drawing the splitting lines.

What should `range()` return if there are no points in the range? It should return an `Iterable<Point2D>` object with zero points.

How much memory does a `Point2D` object use? For simplicity, assume that each `Point2D` object uses 32 bytes—in reality, it uses a bit more because of the `Comparator` instance variables.

How much memory does a `RectHV` object use? You can look at the code and analyze it.

I run out of memory when running some of the large sample files. What should I do? Be sure to ask Java for additional memory, e.g., `java -Xmx1600m RangeSearchVisualizer input1M.txt`.

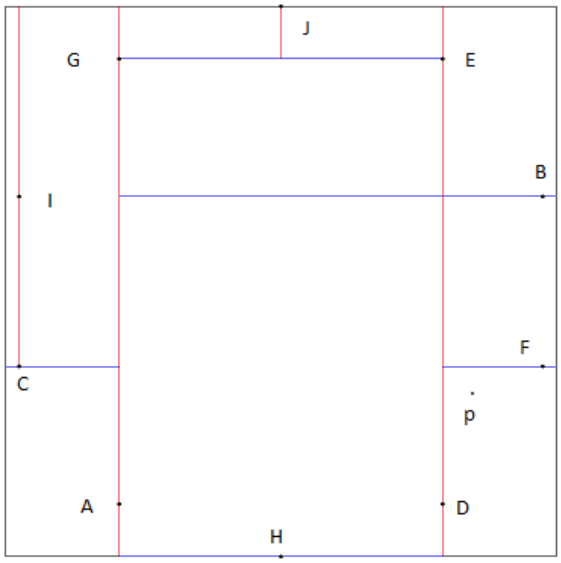
Testing

Testing. A good way to test `KdTree` is to perform the same sequence of operations on both the `PointSET` and `KdTree` data types and identify any discrepancies. The sample clients RangeSearchVisualizer.java and NearestNeighborVisualizer.java take this approach.

Sample input files. The directory kdtree contains some sample input files in the specified format. For convenience, kdtree-testing.zip contains all of these files bundled together.

- `circleN.txt` contains `N` points on the circumference of the circle centered on (0.5, 0.5) of radius 0.5.

The result of calling `draw()` on the points in `circle10.txt` should look like the following:



If `nearest()` is called with `p = (.81, .30)` the number of nodes visited in order to find that F is nearest is 5.

Starting with `circle10k.txt` if nearest is called with `p = (.81, .30)` the number of nodes visited in order to find that K is nearest is 6.

Possible Progress Steps

These are purely suggestions for how you might make progress. You do not have to follow these steps.

1. **Node data type.** There are several reasonable ways to represent a node in a 2d-tree. One approach is to include the point, a link to the left/bottom subtree, a link to the right/top subtree, and an axis-aligned rectangle corresponding to the node.

   ```
   private static class Node {
      private Point2D p;      // the point
      private RectHV rect;    // the axis-aligned rectangle corresponding to this node
      private Node lb;        // the left/bottom subtree
      private Node rt;        // the right/top subtree
   }
   ```

   Unlike the `Node` class for `BST`, this `Node` class is static because it does not refer to a generic `Key` or `Value` type that depends on the object associated with the outer class. This saves the 8-byte inner class object overhead. (Making the `Node` class static in `BST` is also possible if you make the `Node` type itself generic as well). Also, since we don't need to implement the rank and select operations, there is no need to store the subtree size.

2. **Writing KdTree.** Start by writing `isEmpty()` and `size()`. These should be very easy. From there, write a simplified version of insert() which does everything except set up the `RectHV` for each node. Write the `contains()` method, and use this to test that `insert()` was implemented properly. Note that `insert()` and `contains()` are best implemented by using private helper methods analogous to those found on page 399 of the book or by looking at BST.java. We recommend using orientation as an argument to these helper methods.

   Now add the code to `insert()` which sets up the RectHV for each Node. Next, write `draw()`, and use this to test these rectangles. Finish up KdTree with the nearest and range methods. Finally, test your

implementation using our interactive client programs as well as any other tests you'd like to conduct.

Optimizations

These are many ways to improve performance of your 2d-tree. Here are some ideas.

- **Squared distances.** Whenever you need to compare two Euclidean distances, it is often more efficient to compare the squares of the two distances to avoid the expensive operation of taking square roots. Everyone should implement this optimization because it is both easy to do and likely a bottleneck.
- **Range search.** Instead of checking whether the query rectangle intersects the rectangle corresponding to a node, it suffices to check only whether the query rectangle intersects the splitting line segment: if it does, then recursively search both subtrees; otherwise, recursively search the one subtree where points intersecting the query rectangle could be.
- **Save memory.** You are not required to explictily store a `RectHV` in each 2d-tree node (though it is probably wise in your first version).