COMP 215 Assignment #4

Due October 15th (note this is the Monday after the break) at 11:55PM

1 Description

A4 is concerned with implementing a number of classes that simulate random variables. In this assignment, you are asked to implement five different random variables: a uniform, a "unit" gamma, a gamma, a multinomial, and a Dirichlet. For out purposes, a "unit" gamma is one with a shape of at most one, and a scale of one.

All of the random variables in A4 implement the following interface:

```
/**
 * Encapsulates the idea of a random object generation algorithm.  The random
 * variable that the algorithm simulates outputs an object of type OutputType.
 */
interface IRandomGenerationAlgorithm <OutputType> {
  /**
   * Generate another random object
   */
  OutputType getNext ();
  /**
   * Resets the sequence of random objects that are created.  The net effect
   * is that if we create the IRandomGenerationAlgorithm object,
   * then call getNext a bunch of times, and then call startOver (), then
   * call getNext a bunch of times, we will get exactly the same sequence
   * of random values the second time around.
   */
  void startOver ();
}
```

To make all of this a bit easier, in A4 we are defining for you precisely the set of methods that should appear in the abstract class that each of your five random variables should be derived from. This class is called `IRandomGenerationAlgorithm`. The key design decision that we have made is to move all of the functionality associated with creating and maintaining a pseudo random number generator into the abstract class. The class is defined as follows:

```
abstract class ARandomGenerationAlgorithm <OutputType> implements
               IRandomGenerationAlgorithm <OutputType> {
  /**
   * This constructor is called when we want an ARandomGenerationAlgorithm
   * who uses the default pseudo-random number generator; in our case, this
   * is the class PRNG, and it will be initialized with mySeed
   */
  protected ARandomGenerationAlgorithm (long mySeed) {}
  /**
   * This one is called when we want an ARandomGenerationAlgorithm who uses
   * a specific pseudo-random number generator
   */
  protected ARandomGenerationAlgorithm (IPRNG useMe) {}
  /**
   * Generate another random object
   */
  abstract public OutputType getNext ();
  /**
   * Resets the sequence of random objects that are created.  The net effect
   * is that if we create the IRandomGenerationAlgorithm object,
   * then call getNext a bunch of times, and then call startOver (), then
   * call getNext a bunch of times, we will get exactly the same sequence
   * of random values the second time around.
   */
  public void startOver () {}
  /**
   * Returns a reference to the PRNG that this guy is currently using
   */
  protected IPRNG getPRNG () {}
  /**
   * Generate a random number uniformly between low and high using the
   * appropriate PRNG (this object's or its parent's)
   */
  protected double genUniform(double low, double high) {}
}
```

Since there is only one abstract method in this class, it means that in the concrete classes you implement, you will only need to provide code for `getNext`, as well as two constructors (more on the constructors below) and any private methods you need.

Note that the above abstract class depends on the `IPRNG` interface. This is as follows:

```
/**
 * Interface to a pseudo random number generator that generates
 * numbers between 0.0 and 1.0 and can start over to regenerate
 * exactly the same sequence of values.
 */
interface IPRNG {
  /**
   * Return the next double value between 0.0 and 1.0
   */
  double next();
  /**
   * Reset the PRNG to the original seed
   */
  void startOver();
}
```

We will supply you with one particular implementation of `IPRNG`, called `PRNG`. `PRNG` has a constructor that takes a single `long`, where that long is the value of the seed to use. `PRNG` is really just a wrapper for the Java `Random` class, which itself implements the linear congruential method. One nice thing about this design is that if we wanted to be fancy, we could come up with other implementations for `IPRNG` that implement more complex and robust pseudo random number generation algorithms and we could easily use those instead.

Given this, the five concrete classes that you should implement are:

```
Uniform extends ARandomGenerationAlgorithm <Double>
UnitGamma extends ARandomGenerationAlgorithm <Double>
Gamma extends ARandomGenerationAlgorithm <Double>
Multinomial extends ARandomGenerationAlgorithm <IDoubleVector>
Dirichlet extends ARandomGenerationAlgorithm <IDoubleVector>
```

Each of these classes should have two constructors. The first constructor for each of these five classes should be as follows:

```
public Uniform (long mySeed, UniformParam myParam)
public UnitGamma (long mySeed, GammaParam myParam)
public Gamma (long mySeed, GammaParam myParam)
public Multinomial (long mySeed, MultinomialParam myParam)
public Dirichlet (long mySeed, DirichletParam myParam)
```

Note that each of these five constructors accepts parameters of type `UniformParam`, `GammaParam`, `MultinomialParam`, and `DirichletParam`, respectively. These are little classes that simply encapsulate the set of parameters that go into each distribution. For example, `GammaParam` wraps up a shape and a scale, as well as the leftmost edge of the leftmost step and the number of steps to use during the generation process. We have provided all of these `Param` classes for you. Each of the five constructors also accepts a seed. If this constructor is called, then the algorithm should use a `PRNG` object to go its pseudo random number generation, seeded with `mySeed`.

The second constructor for each of these five classes should be as follows:

```
public Uniform (IPRNG useMe, UniformParam myParam)
public UnitGamma (IPRNG useMe, GammaParam myParam)
public Gamma (IPRNG useMe, GammaParam myParam)
public Multinomial (IPRNG useMe, MultinomialParam myParam)
public Dirichlet (IPRNG useMe, DirichletParam myParam)
```

If this second constructor is used, then the resulting object should use exactly the provided `IPRNG` object to do its pseudo random number generation, as opposed to creating and using its own, default object.

Here are a few specific points regarding the assignment:

1. There are a lot of dependencies among the random variable types in the sense that you will build one on top of another. To make life a bit easier, we will provide for you a `.jar` file that contains implementations of all of the five concrete classes, as well as the one abstract class. Each of these implementations is pre-pended with `SC`. This means that in your code, you can (for example) use `SCUnitGamma` (almost) wherever you see fit—for example, you can use it to build the `Gamma` class. That way, you don't have to worry about bungling `UnitGamma` and then not being able to get `Gamma` to work. Just use `SCUnitGamma` inside of your `Gamma` implementation if you need to.

2. You can even include `SC` implementations in the code that you turn in, but there are two places where you will be penalized for this. One is within the class that it implements. For example, you can't use `SCUnitGamma` to implement `UnitGamma`. If you do this, you'll not get credit for the four test cases that cover `UnitGamma`. The other rule is that if you use `SCARandomGenerationAlgorithm`

anywhere in your final code, we'll "dock" you four test cases. So if you pass 18 test cases, but need to use `SCARandomGenerationAlgorithm`, you'll get credit for 14 test cases. The reason for this somewhat strange rule is that the abstract class is used all over the place in your code, though there is no test case that specifically targets it. So if you need to use `SCARandomGenerationAlgorithm` to get your stuff to work, that's fine, but there is a penalty associated with this.

3. Naturally, there is no requirement that you use the `SC` implementations. They are there to make development and debugging a bit easier. In theory, once you pass all of the test cases, you can take out all of the `SC` implementations, replace them with your own, and everything will work fine.

4. There is a lot to do here, but it should be manageable if you take things step by step. You should really start out by implementing the `Uniform` class, since this is the simplest. There is almost no code involved. Doing this first will help you understand the basic setup of the assignment.

5. To an extent, these classes should build upon each other. The `Dirichlet` class, for example, should fundamentally rely on the `Gamma` class. You should use the supplied `UnitGamma` class to build the `Gamma` class.

6. You should do any computational work associated with setting up the random variable in the constructor. For example, building the "stair steps" for the `UnitGamma` should happen in the constructor.

7. The vectors returned by the `Dirichlet` and `Multinomial` classes should match what was discussed in lecture. Briefly, the `Dirichlet` should return a normalized vector of values generated from Gamma distributions with the shapes given in the `shapes` vector in the `DirichletParam` class. The `Multinomial` should return a vector that indicates the number of times each element was selected, where the probability of selecting a particular element in any given trial is given by the `probs` vector in the `MultinomialParam` class.

8. Look over the test cases we will supply you with. The tests we'll run are a bit different this time around. Since you are implementing a bunch of random variables, it is not possible for us to look at the output of a single trial over the variable and determine if your code works. Instead, we will call your generation code thousands of times, and then make sure that the statistical properties of what you generate (the mean and variance, for example) are exactly what one would expect.

2 Grading

Passing all of the test cases is 80% of your grade. Documentation counts for 20% of your grade. The guidelines from A1 for commenting your code apply to A4 as well.

3 What You Need To Do

The first thing that you need to do is to download the `.jar` file that contains both `IDoubleVector` implementations, `PRNG`, as well as all of the `SC` types. Create a `HW4.drjava` project and add this `.jar` file to it. Also, download the four parameter classes, the `IRandomGenerationAlgorithm` interface, the test file, and add all of those to your `HW4.drjava` project. At that point, you are ready to go.

4 Turnin

You should add all of your source files and the `.jar` file into a DrJava project as described above. On Owlspace, you should turn in the `HW4.drjava` project file that DrJava creates when you create your project, as well as any and all `.java` source files or `.jar` files that you create or use in your project. The goal here is for us to be able to simply fire up DrJava using your `.drjava` project, and everything should work without having to move files around or add files to the project. Please comment out any debugging output from your code before you turn it in.

Please upload each file directly. Do not create an archive (zip, rar, etc.). Be super-careful and double check after you submit. Make sure you have turned in exactly what you thought you have turned in!

5 Academic Honesty

Please keep the academic honesty policy in mind as you work on the assignment.