# CSCI 1933 Project 1

## Mathematical Functions and Classes

## Instructions

Welcome to the first CSCI 1933 project! We recommend that you do *not* use an IDE when writing this project – the terminal and a simple text editor are sufficient. Students who learn to write and debug Java programs without the help of an IDE will gain fundamental experience as programmers and will likely do well on midterms and the final.

All students are expected to understand the rules listed below. While some rules may seem unforgiving, all guidelines are made to help the TAs grade efficiently and fairly. As a result, we will generally not make exceptions. Rules listed in the syllabus also apply but may not be listed here.

- **Due:** The project is due on **Friday, February 10th** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment in all files you submit. For example, `//Written by shino012 and hoang159`.
- **Submission:** Submit a `zip` or `tar` archive on Moodle containing all your `java` files. You are allowed to change or modify your submission, so submit early and often, and *verify that all your files are in the submission.*

  Failure to submit the correct files will result in a score of zero for all missing parts. Late submissions and submissions in an abnormal format (such as `.rar` or `.java`) will be penalized. Only submissions made via Moodle are acceptable.
- **Partners:** You may work alone or with *one* partner. **Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero**. Ensure all code shared with your partner is private.
- **Code:** You must use the *exact* class and method signatures we ask for. This is because we use a program to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 8, which is installed on the CSE Labs computers.
- **Questions:** Questions related to the project can be discussed on Moodle in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Moodle.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately*.

## Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.

- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName, functionName, variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments. `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

In further projects we will continue to expect a reasonable attempt at documentation and style as detailed in this section. If you are confused about the style guide, please talk with a TA.

## 1 Random Numbers

In this part you will program an instantiable `Random` class for generating random numbers. You are required to use a sequential random number generator (RNG). This is a function of form

$$\mathrm{r}_{\mathrm{new}} = ((P_1 * \mathrm{r}_{\mathrm{old}}) + P_2) \bmod M \tag{1}$$

In this function, $P_1$, $P_2$, and $M$ are large *constant* prime numbers (for example, 7919, 65537, and 102611). Random numbers generated by this function will be strictly less than $M$, and greater than or equal to zero. Choosing $P_1$, $P_2$, and $M$ can be more of an art than a science - some results seem random, and others are predictable. Therefore, our class will allow the programmer to specify these values in the constructor for each `Random` they create.

We will also need to specify an initial value $S$, since there will be no $r_{\mathrm{old}}$ the first time we invoke the function. This value is called the *seed*.

**Note:** Some students will notice that this is not inherently "random". What we are really doing is generating numbers with such a complicated function that it's impossible to predict the next one. However, if we set the seed to the same value, the exact same numbers will appear! This property will be useful for debugging.

Create a `Random` class with the following methods. You are allowed to create any member variables you want so long as they are all `private`. **You may not use another library, such as Java's Math module, to generate random numbers**.

- `public Random(int p1, int p2, int m)`: Set up a random number generator with the given constants. The constants must be in this order. Make sure not to confuse the role of $P_1$ and $P_2$ - check the description above if you are unsure. This should initialize the seed to 0. In general, $P_1$, $P_2$, and $M$ should *never* be changed after this constructor gets called.
- `public void setSeed(int seed)`: Set the seed of the random number generator.
- `public int getMaximum()`: Return the value of $M$ for this random number generator.
- `public int random()`: Use the sequential random number algorithm to generate the $r_{\mathrm{new}}$ value and return it. If this method is called again without resetting the seed, it should generate a different value.

  This should be the only function that applies the generation algorithm. All subsequent methods should be accomplished using only a call to this method.
- *Continued on the next page...*

- `public int randomInteger(int lower, int upper)`: Return a random integer in the range `lower` to `upper`. Possible values for `randomInteger(1, 5)` should be 1, 2, 3, 4 and 5; each of them should be equally probable. If `lower` > `upper`, swap them and don't cause an error.

  **Note:** We have the inequality $0 \leq$ `random()` $< M$ for all values. What happens when we add a constant to each element the inequality? What happens if we multiply each element by a constant? Will it still hold true?

- `public boolean randomBoolean()`: Randomly return `true` or `false`. This should simulate a 50% chance.

- `public double randomDouble(double lower, double upper)`: Get a random double in the range `lower` to `upper`. Possible values of `randomDouble(1.0, 5.0)` could be `1.0`, `5.0`, `1.265646948`, `4.9999999998`, etc... If `lower` > `upper`, swap them and don't cause an error.

  **Note:** In Java, if you try to divide a positive `int` by a larger positive `int`, the result is always 0. For example, `75 / 100` is 0. How can you avoid this problem?

- **A main method**: The main method should demonstrate sufficient testing to prove that each of the methods work. This should not just generate a bunch of random output; it should prove to the TA that you are certain the program is correct. This means *multiple* tests per method, and not just for cases that should succeed. Some things to try:

  – Ensure that an even distribution occurs – that is, every choice is equally probable.

  – Ensure that methods are robust – they do not throw exceptions on invalid input.

  – Ensure your generator works as a whole. Some functions should not break other functions depending on the order you call them in.

  The output of the main method should make the fact that the program is working quite obvious. This means printing useful information and human-readable text, not just a bunch of numbers or "true" statements.

*The rest of this page is intentionally left blank*

## 2 Prime Numbers

In this part you will program a `Prime` class for dealing with prime numbers. A prime number is an integer that is not evenly divisible by any integer other than itself and 1. 1, 0, and negative numbers are not prime. Create a `Prime` class with the following methods:

- `public static boolean isPrime(int number)`: This method should return `true` if the `number` is prime, and `false` otherwise. Use the definition of a prime number given above.
- `public static boolean isMersennePrime(int number)`: A *Mersenne prime* is a prime number that is equal to $2^k - 1$ for some integer $k$. This method should check if `number` is a Mersenne prime.
- `public static int prime(int n)`: This method should return the `n`th prime number. We will define 2 as the first (`n == 1`). This method should complete in under 5 seconds, even for large numbers. If `n` is less than 1, return `-1`.
- `public static int[] primeArray(int howMany)`: This method should return an array of the first `howMany` prime numbers. The array must be of length `howMany`. This method should also complete in under 5 seconds, even for large numbers. Return `null` if the `howMany` variable is less than one.
- `public static int[] primeFactors(int number)`: This method should return the prime factorization of `number`. The prime factorization of a number is a sequence of prime numbers whose product is the number. For example, the prime factorization of `7` is `[7]` and the factorization of `630` is `[2, 3, 3, 5, 7]`. The array you return can be in any order. Return `null` if the `number` is less than one.
- **A main method**: Like the previous class, demonstrate that all the methods work via well-documented, well-formatted tests.

**Note:** A `static` function is different from a regular instance method. To use an instance method, you would need to construct a `Class object = new Class()` and then invoke `object.method()`. Static methods do not require this; they can be called with `Class.method()`.

*The rest of this page is intentionally left blank*