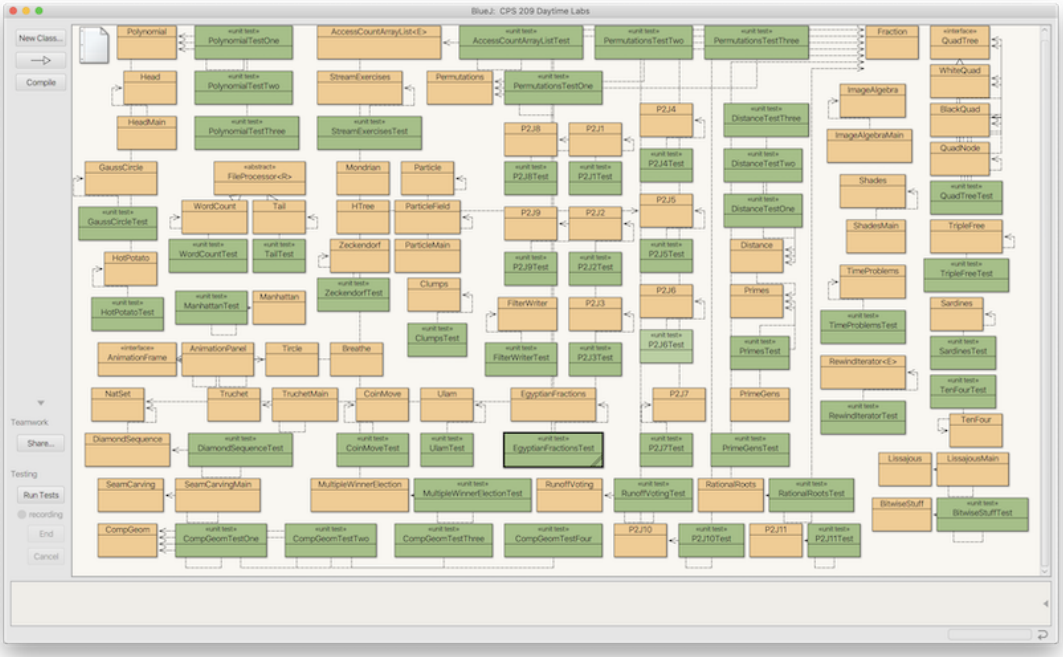
CCPS 209 Labs

This document contains the lab specifications for CCPS 209 Computer Science II, as compiled by Ilkka Kokkarinen. It currently contains **sixty-three problem specifications** for students to freely pick the problems they wish to tackle. These problems vary greatly in theme and spirit so that like *The Love Boat*, they promise something for everyone. These lab problems both drill in the Java language and its standard library, and introduce (although sometimes only between the lines) many classic ideas, concepts and techniques that will surely come handy in your future.

Working in BlueJ, **create one BlueJ project folder named "209 Labs", in which you write all of these labs.** That's right, **all your labs will go in one and the same project folder.** Into this same BlueJ project, you should also manually add the JUnit test classes provided by the instructor for each lab that you choose to implement. These fully automated JUnit tests for these labs are available in the instructor's GitHub repository `CCPS209Labs`.

The BlueJ project that you submit for grading at the end of the course should look something like the following low-resolution screenshot taken from the private model solution of the instructor, except that your project contains only the classes for the lab problems that you have solved, along with their automated testers. Positioning of individual classes inside this project window is entirely up to you, but make sure that the graders can tell how many lab problems you have successfully completed. (It might be a good idea to write this in the `README` text file.)



Pressing the button "Run Tests" on the left column will then verify that the classes and methods that you have written are working as expected. For the labs that ask you to write a Swing

component, the lab marks are given for these components passing the visual inspection of the behaviour specified in the lab. The other labs must cleanly pass the entire JUnit test to receive the marks for that lab. The lab points are **all or nothing** for each lab, and **no partial marks are given for labs that do not not pass these tests.**

In all these labs, silence is golden. Since many of these JUnit tests will test your code with a very large number of pseudo-randomly generated test cases, **your methods must be absolutely silent** and print nothing on the console during their execution. **Any lab solution that prints anything at all on the console during its execution will be rejected and receive a zero mark.** Make sure to comment out all your print statements used in debugging before submitting your project. Especially since BlueJ uses a graphical console to display the text, scrolling through millions of lines of debugging output horizontally and vertically will slow the entire test down to a painful crawl. Even if you put these debugging outputs behind some boolean verbosity toggle that decides whether the debugging output will actually get printed, composing the debugging outputs as text strings can significantly slow down the test.

For each lab that you choose to implement and submit, the project must also contain the JUnit test class so that the graders can evaluate your project with one click of "Run Tests" and watch the green checkmarks accumulate in the test report window. **You may not modify these provided JUnit tests in any way whatsoever** in your submission. Modifying any JUnit test to make it seem like that your class cleanly passes that test, even though it really does not do that, is considered **serious academic dishonesty** and will be penalized accordingly as defined in Ryerson policies.

Once you have completed all the labs that you think you are going to complete before the deadline, you will **submit all your completed labs in one swoop** as the entire "209 Labs" BlueJ project folder that contains all the source files and the provided JUnit tests, compressed into a zip file that you upload into the assignment tab on D2L.

Even if you write your labs working on Eclipse or some other IDE, it is compulsory to submit these labs as a single BlueJ project folder. No other form of submission is acceptable. (This rule is in effect if you are taking the course under the instructor Ilkka Kokkarinen. Other instructors can set their own requirements for lab submission. Their special instructions in their course sections will always supersede this document.)

ACCIDENTS HAPPEN, SO MAKE FREQUENT BACKUPS OF THE LAB WORK THAT YOU HAVE COMPLETED SO FAR. SERIOUSLY. THIS CANNOT BE EMPHASIZED ENOUGH NOT ONLY IN THIS COURSE, BUT IN ALL OUR INCREASINGLY DIGITAL LIVES.

Lab 0(A): Just Some Plain Integers

JUnit: P2J1Test.java

The transition labs whose numbers are of form 0(X) translate your existing Python knowledge into equivalent Java knowledge all the way between your brain and your fingertips. You may already be familiar with some of these problems from the same instructor's course CCPS 109 Computer Science I. Even if you are coming to this course via some other route, these problems are self-contained and require no particular background course. Each 0(X) lab consists of up to four required `static` methods that are effectively **functions** that involve no object oriented thinking, design or coding.

Inside your fresh new BlueJ project, create the first class that **must be named** exactly `P2J1`. If you name your classes and methods anything other than how they are specified in this document, the JUnit tests used to automatically check and grade these labs will not be able to even find your methods. Erase the nonsense template that BlueJ fills inside the class body in its misguided effort to "help" you, and in its place, write the following four methods.

```
public static long fallingPower(int n, int k)
```

Python has the integer exponentiation operator ** conveniently built in the language, whereas Java unfortunately does not have that operator, which would be mostly useless anyway in a language with fixed size integers. (In both languages, the **caret** character ^ denotes the **bitwise exclusive or** operation that has bupkis to do with integer exponentiation.)

In the related operation of **falling power** that is useful in many combinatorial formulas and denoted syntactically by underlining the exponent, each term that gets multiplied into the product is always one less than the previous term. For example, the falling power $8^{\underline{3}}$ would be computed as 8 * 7 * 6 = 336. Similarly, the falling power $10^{\underline{5}}$ would equal 10 * 9 * 8 * 7 * 6 = 30240. Nothing essential changes if the base `n` is negative. For example, the falling power $(-4)^{\underline{5}}$ is computed the exact same way as -4 * -5 * -6 * -7 * -8 = -6720.

This method should compute and return the falling power $n^{\underline{k}}$ where the base n can be any integer, and the exponent k can be any **nonnegative** integer. (Analogous to ordinary powers, $n^{\underline{0}} = 1$ for any n.) The automated tests are designed so that your method does not have to worry about potential integer overflow as long as you perform computations using the `long` kind of 64-bit integers. (However, if you use the bare 32-bit `int` type, a silent overflow will make your method silently return incorrect results and fail these tests.)

```
public static int[] everyOther(int[] arr)
```

Given an integer array `arr`, create and return a new array that contains precisely the elements in the even-numbered positions in the array `arr`. Make sure that your method works correctly for arrays of both odd and even lengths, and for arrays that contain zero or only one element. The length of the result array that you return must be exactly right so that there are no extra zeros at the end of the array.

```
public static int[][] createZigZag(int rows, int cols, int start)
```

This method creates and returns a new two-dimensional integer array, which in Java is really just a one-dimensional array whose elements are one-dimensional arrays of type `int[]`. The returned array must have the correct number of `rows` that each have exactly `cols` columns. This array must contain the numbers `start`, `start + 1`, ..., `start + (rows * cols - 1)` in its rows in order, except that the elements in each odd-numbered row must be listed in descending order.

For example, when called with `rows = 4`, `cols = 5` and `start = 4`, this method should create and return the two-dimensional array whose contents would look like

```
4     5     6     7     8
13    12    11    10    9
14    15    16    17    18
23    22    21    20    19
```

when displayed in the traditional matrix form that is more readable for humans than the more truthful form of a one-dimensional array whose elements are one-dimensional arrays of rows.

```
public static int countInversions(int[] arr)
```

Inside an array `a`, an **inversion** is a pair of positions `i` and `j` inside the array that satisfy simultaneously both `i < j` and `a[i] > a[j]`. In combinatorics, the inversion count inside an array is a rough measure of how "out of order" that array is. If an array is sorted in ascending order, it has zero inversions, whereas an n-element array sorted in reverse order has $n(n\text{-}1)/2$ inversions, the largest number possible. (You will encounter array inversions again if you work through the three labs 45 to 47 that deal with permutations and their operations.) This method should count the inversions inside the given array `arr`, and return that count. As always when writing methods that operate on arrays, make sure that your method works correctly for arrays of any length, including the important special cases of zero and one.

Once you have written all four methods, you can download and add the above **JUnit test** class P2J1Test.java to be placed separately inside the same BlueJ project. In the BlueJ project display window, the JUnit test classes show up as green boxes, as opposed to the usual yellow box like the ordinary classes. Unlike in Python with its more dynamic nature, **the JUnit test class cannot be compiled until your class contains all four methods exactly as they are specified.** If you want to test one method without having to first write also the other three, you can implement the other three methods as do-nothing **method stubs** that only contain some placeholder `return` statement that returns zero or some other convenient do-nothing value. Of course all these method stubs will fail their respective tests, but their existence allows the JUnit test to compile so that you can test the one method that you have properly written. Once that is done, move on to replace the placeholder body of the next method with its actual body.

Once successfully compiled, you can right-click the JUnit test class to run either any one test, or all tests at once. The methods that receive a green checkmark have passed the JUnit test and are complete. A red mark means that your method returned a wrong answer at some point, whereas a black mark means that your method crashed at some point and threw an exception.

Because these JUnit test methods, as implemented by your instructor, call your method with a large number of pseudo-randomly generated test cases, compute a checksum of the results returned by your method, and compare that to the expected checksum produced by the instructor's private model answer, it is impossible to point out precisely which particular test cases are different and failing for your method. You should therefore also create your own small test cases to find the errors in your code.

Lab 0(B): Duplications

JUnit: P2J2Test.java

Create a new class named `P2J2` inside the very same BlueJ project as you placed your `P2J1` class in the previous lab. Inside this new class `P2J2`, write the following four methods.

```
public static String removeDuplicates(String text)
```

Given a `text` string, create and return a new string that is otherwise the same as `text` except that every run of equal consecutive characters has been turned into a single character. For example, given the arguments `"Kokkarinen"` and `"aaaabbxxxxaaxa"`, this method would return `"Kokarinen"` and `"abxaxa"`, respectively. Note that only the consecutive duplicate occurrences of the same character are eliminated, but the later occurrences of the same character remain in the result as long as there was some other character between these occurrences.

```
public static String uniqueCharacters(String text)
```

Given a `text` string, create and return a new string that contains each character only once, with the characters given in order in which they appear in the original string. For example, given the arguments `"Kokkarinen"` and `"aaaabbxxxxaaxa"`, this method would return `"Kokarine"` and `"abx"`, respectively.

You can solve this problem with two nested loops, the outer loop iterating through the positions of the `text`, and the inner loop iterating through all previous positions to look for a previous occurrence of that character. Or you can be much more efficient, and use a `HashSet<Character>` to remember which characters you have already seen, so that you can determine in O(1) time whether you append the next character into the result.

```
public static int countSafeSquaresRooks(int n, boolean[][] rooks)
```

Some number of rooks have been placed on some squares of a generalized `n`-by-`n` chessboard. The two-dimensional array `rooks` of boolean truth values tells you which squares contain a rook. (This array is guaranteed to be exactly `n`-by-`n` in its dimensions.) This method should count how many remaining squares are safe from these rooks, that is, do not contain any rooks in the same row or column, and return that count.

```
public static int recaman(int n)
```

Compute and return *n*:th term of the Recamán's sequence, as defined on Wolfram Mathworld, starting from the term $a_1 = 1$. See the definition of this sequence on that page. For example, when called with $n = 7$, this method would return 20, and when called with $n = 19$, return 62. (More values for debugging purposes are listed at OEIS sequence A005132.)

To make your function fast and efficient even when computing the sequence element for large values of *n*, you should use a sufficiently large `boolean[]` (size `10*n` is certainly enough, and yet uses only a handful of bits of heap memory per each number) to keep track of which integer values are already part of the generated sequence, so that you can generate each element in constant time instead of having to loop through the entire previously generated sequence like some "Shlemiel" tiring himself by running back and forth across the same positions.

Lab 0(C): Reverse the Verse

JUnit: P2J3Test.java

One last batch of transitional problems taken from the instructor's Python graded labs. Only three methods to write this time, though. The automated JUnit test for these labs uses the `warandpeace.txt` text file as source of data, so ensure that this text file has been properly copied into your course labs project folder. **This text file, same as any other data files you would use, does not show up anywhere in the BlueJ project screen, even though it is part of the project directory underneath.**

```
public static void reverseAscendingSubarrays(int[] items)
```

Rearrange the elements of the given array of integers **in place** (that is, **do not create and return a new array**) so that the elements of every **maximal strictly ascending subarray** are reversed. For example, given the array `{ 5, 7, 10, 4, 2, 7, 8, 1, 3 }` (these pretty colours indicate the ascending subarrays and are not actually part of the argument), after executing this method, the elements of the array would be `{ 10, 7, 5, 4, 8, 7, 2, 3, 1 }`. Given another argument array `{ 5, 4, 3, 2, 1 }`, it would become `{ 5, 4, 3, 2, 1 }` since each element by itself is a maximal ascending subarray of length one and does not change.

```
public static String pancakeScramble(String text)
```

This nifty little problem is taken from the excellent Wolfram Challenges problem site where you can see examples of what the result should be for various arguments. Given a `text` string, construct a new string by reversing its first two characters, then reversing the first three characters of that, and so on, until the last round where you reverse your entire current string.

This problem is an exercise in Java string manipulation. For some mysterious reason, the Java `String` type does not come with a `reverse` method. The canonical way to reverse a Java string `str` is to first convert it to mutable `StringBuilder`, reverse its contents, and convert the result back to an immutable string, that is,

```
str = new StringBuilder(str).reverse().toString();
```

A bit convoluted, but does what is needed. Maybe one day the Java strings will have a `reverse` method built in, just like the string data types of all sensible programming languages. Or at least as a utility somewhere in the standard library.

```
public static String reverseVowels(String text)
```

Given a `text` string, create and return a new string of same length where all vowels have been reversed, and all other characters are kept as they were. For simplicity, in this problem only the characters `aeiouAEIOU` are considered vowels, and `y` is never a vowel. For example, given the text string `"computer science"`, this method would return the new string `"cempetir sceunco"`.

Furthermore, to make this problem more interesting and the result look more palatable, this method **must maintain the capitalization of vowels** based on the vowel character that was originally in the position that each new vowel character is moved into. For example, `"Ilkka Markus"` should become `"Ulkka Markis"` instead of `"ulkka MarkIs"`. Use the handy utility methods in the `Character` wrapper class to determine whether some particular character is in upper- or lowercase, and to convert a character to upper- or lowercase as needed.

Lab 0(D): Lists of Integers

JUnit: P2J4Test.java

The fourth transition lab from Python to Java contains four more interesting problems taken from the graded labs of the instructor's Python course. The four `static` methods to write here now deal with `List<Integer>` of Java (remember to `import java.util.*` at the top of the source code for your class) whose behaviour and operations are essentially the same as those of ordinary Python lists when used to store only integers, except with a more annoying syntax.

`public static List<Integer> runningMedianOfThree(List<Integer> items)`

Create and return a new `List<Integer>` instance (use any concrete subtype of `List` of your choice) whose first two elements are the same as that of original `items`, after which each element equals the **median** of the three elements in the original list ending in that position. For example, when called with a list that prints out as `[5, 2, 9, 1, 7, 4, 6, 3, 8]`, this method would return an object of type `List<Integer>` that prints out as `[5, 2, 5, 2, 7, 4, 6, 4, 6]`.

`public static int firstMissingPositive(List<Integer> items)`

Given a list whose each element is guaranteed to be a positive integer, find and return the first positive integer missing from this list. For example, given a list that prints out as `[7, 5, 2, 3, 10, 2, 9999999, 4, 6, 3, 1, 9, 2]`, this method should return 8. Given either the list `[]`, `[6, 2, 12345678]` or `[42]` as an argument, this method should return 1.

`public static void sortByElementFrequency(List<Integer> items)`

Sort the elements of the given list in decreasing order of **frequency**, that is, how many times each element appears in this list. If two elements appear in the parameter list the same number of times, those elements should end up in ascending order of values, the same way as we do in ordinary sorting of lists of integers.

For example, given a list object that prints out as `[4, 99999, 2, 2, 99999, 4, 4, 4]`, after calling this method that list object would print out as `[4, 4, 4, 4, 2, 2, 99999, 99999]`. Note that the return type of this method is `void`, so this method rearranges the elements of `items` in place instead of creating and returning a separate result list.

As in all computer programming, you should allow the language and its standard library do your work for you instead of rolling your own logic. The method `Collections.sort` can be given a `Comparator<Integer>` strategy object that compares two integers for their ordering. A good idea might be to start by building a local **counter map** of type `Map<Integer, Integer>` that you use to keep track of how many times each value appears in the list. Next, define a local class that implements `Comparator<Integer>` and performs integer order comparisons by consulting this map for the frequency counts of those elements and returns the answer from that, reverting to ordinary integer order comparison in the case where the frequencies are equal.

`public static List<Integer> factorFactorial(int n)`

Compute and return the list of prime factors of the **factorial** of `n` (that is, the product of all positive integers up to `n`), with those prime factors sorted in ascending order and with each factor appearing in this list exactly as many times as it would appear in the prime factorization of that factorial. For example, when called with `n = 10`, this method would create and return a list that prints out as `[2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7]`. (Multiplying together all those itty bitty

prime numbers would produce the result of 3628800, which equals the product of the first ten positive integers.)

The factorial function `n!` grows exponentially, and for `n > 11` its values would no longer fit inside the range of the `int` type of Java. Since this method must be able to work for values of `n` that are in the thousands, you cannot first compute the factorial of `n` and only then start breaking the resulting behemoth down to its prime factors. Instead, you need to build up the list of prime factors as you go, by appending the prime factors of the integer that you are currently multiplying into the factorial, and then sorting this entire list in the end before returning it.

Lab 0(E): All Integers Great and Small

JUnit: P2J5Test.java

Since the methods in this lab are a bit more complicated, there are only two of them to solve, unlike in the first four transition labs. The Python integer type is unbounded and limited only by your available heap memory, and the Python virtual machine silently switches between different efficient representations for small and large integers, letting us concentrate on the actual problem instead of the details of nitty gritty integer arithmetic, constantly having to worry about and protect against overflow errors.

The utility class `java.math.BigInteger` allows you to do computations in Java with similarly unlimited integers, but this again being Java as originally conceived in the early nineties, the syntax is not as nice as the ordinary and natural syntax for primitive `int` type. For example, to add two such integers `a` and `b`, we have to say `a.add(b)` instead of `a + b`. Consult the rest of the methods of `BigInteger` from its API documentation as needed to solve the following problems.

```
public static List<BigInteger> fibonacciSum(BigInteger n)
```

Fibonacci numbers, that dusty old example of a silly recursion still used in many introductory courses on computer science, turn out to have all kinds of interesting combinatorial properties that also make for more meaningful problems for us. For this lab, we look at Zeckendorf's theorem that proves that any given positive integer `n` can be exactly one way into a sum of distinct Fibonacci numbers, given the constraint that no two consecutive Fibonacci numbers appear in this sum. After all, if the sum contains two consecutive Fibonacci numbers F_i and F_{i+1}, these two can always be replaced by F_{i+2} without affecting the total. (To ensure that you have at most one of each F_i at any moment, this conversion should be done from higher indices downwards.)

The unique breakdown into Fibonacci numbers can be constructed with a **greedy algorithm** that simply finds the largest Fibonacci number `f` that is less than or equal to `n`. Add this `f` to the result list, and break down the rest of the number `n - f` the same way. This method should create and return an instance of some subtype of `List<BigInteger>` that contains the selected Fibonacci

numbers in **descending order**. For example, when called with `n = 1000000`, this method would return a list that prints as `[832040, 121393, 46368, 144, 55]`.

Your method must remain efficient even if `n` contains thousands of digits. To achieve this, maintain a list of Fibonacci numbers that you have generated so far initialized with

```
private static List<BigInteger> fibs = new ArrayList<>();
static { fibs.add(BigInteger.ONE); fibs.add(BigInteger.ONE); }
```

and then whenever the last Fibonacci number in this list is not big enough for your present needs, extend the list with the next Fibonacci number that you get by adding the last two known Fibonacci numbers. Keeping all your Fibonacci numbers that you have discovered so far in one sorted list would also allow you to do nifty things such as using `Collections.binarySearch` to quickly determine if some given integer is a Fibonacci number...

```
public static BigInteger sevenZero(int n)
```

Since seven is a famously lucky number in the Western culture, whereas zero is what nobody wants to be, let us bring these two opposites together for a moment and look at positive integers that consist of some sequence of sevens, followed by some (possibly empty) sequence of zeros. For example, 0, 7, 77777, 7700000, 77777700, or 700000000000000000000000000000000000. Note that all sevens must be in one consecutive bunch, followed by all zeros in another consecutive bunch.

One of the examples of combinatorial thinking with pigeonhole principle used in the MIT online textbook "*Mathematics for Computer Science*" (PDF link to the updated 2018 version, for any of you who are interested in that sort of stuff) points out that for any positive integer `n`, there exists at least one positive integer thus constrained to sevens and zeroes that is divisible by `n`.

This integer made of sevens and zeroes can get pretty humongous as `n` grows larger, but at least one such number must always exist for any integer `n`, and in fact it is easy to prove that infinitely many such numbers exist. This method should find and return the **smallest** such integer that is divisible by the given `n`. The easiest way to do this would be to use two nested loops. The outer `while`-loop iterates through all possible lengths (that is, how many digits the number contains) of the number, and for each length, the inner `for`-loop iterates through all legal sequences of sevens followed by zeros of that total length. Keep going up until you find such a number that is exactly divisible by `n`.

This process will always eventually terminate for any `n`, since such a number is always guaranteed to exist, although these numbers can easily become gargantuan. For example, for the argument value `n = 12345`, the resulting number consists of 822 copies of the digit seven followed by a single zero digit. To speed up the search, you can utilize an additional theorem given in the same book, that says that unless `n` is divisible by either 2 or 5, the sequence of sevens and zeros that is

divisible by `n` is guaranteed to contain only sevens but no zeros. This ought to speed up your search by at least an order of magnitude for such easy values of `n`, since the quadratic number of possibilities to examine is pruned down into a single linear branch 7, 77, 777, 7777, ...

Lab 0(F): Two Branching Recursions

JUnit: P2J6Test.java

In most mainstream computer science education, teaching of recursion tends to "nerf down" this mighty blade to cut through exponential possibilities by using it only as a toy to simulate some linear loop to compute factorials or Fibonacci numbers. This gains nothing over using ordinary for-loops the way that any reasonable person would have automatically done anyway. This also tends to leave the students understandably confused about the general usefulness of recursion, since from their point of view, recursion is only ever used to solve toy problems in a needlessly convoluted and mathematical manner.

However, as ought to become amply evident somewhere in the third or fourth year, the power of recursion lies in its **ability to branch to multiple directions one at a time**, which allows a recursive method to explore a potentially **exponentially large branching tree of possibilities** and **backtrack on failure to previous choice point** until it either finds one branch that works, or alternatively collect the results from all of the working branches. In this spirit, this last transition lab uses branching recursions to solve two interesting combinatorial problems. So without any further ado, create a new class `P2J6` to write the two methods in.

```
public static List<Integer> sumOfDistinctCubes(int n)
```

Determine whether the given positive integer `n` can be expressed as a sum of cubes of positive integers greater than zero so that all these integers are distinct. For example, `n = 1456` can be expressed as sum $11^3 + 5^3$. This method should return the list of these distinct integers as some kind of a list that prints out as `[11, 5]` with the elements given in descending order. If `n` cannot be broken into a sum of distinct cubes, this method should return the empty list.

Many integers can be broken down into a sum of cubes in several different ways. This method must always return the breakdown that is **lexicographically highest**, that is, starts with the largest possible working value for the first element, and then follows the same principle for the remaining elements that break down the rest of the number into a list of distinct cubes. For example, when called with `n = 1072`, this method must return the list `[10, 4, 2]` instead of `[9, 7]`, even though $9^3 + 7^3 = 1072$ just as well. This constraint might seem scary at first, but really just *fuhgettaboutit*, since it can be trivially enforced by arranging the loop to look for current possibilities always from highest to lowest. The easiest way to implement this recursion is again to use a private helper method that accepts additional parameters that were not present in the public top level method.

```
private static boolean sumOfDistinctCubes(int n, int c, LinkedList<Integer> soFar)
```

This method receives the two extra recursion parameters `c` and `soFar` from the original method. The parameter `c` gives you the highest number that you are still allowed to use (initially this should equal the largest possible integer whose cube is less than equal to `n`, easily found with a while-loop), and the parameter `soFar` contains the list of numbers that have already been taken in the list of cubes.

The recursion has two base cases, one for success and one for failure. If `n == 0`, the problem is solved and you can just return `true`. If `c == 0`, there are no numbers remaining that you can use, and you simply return `false`. Otherwise, try taking `c` into the sum (also adding `c` to `soFar`) and recursively solve the problem for new parameters `n-c*c*c` and `c-1`. If that one was not a success, remove `c` from `soFar`, and try to recursively solve the problem without using `c`, which makes the parameters of the second recursive call to be `n` and `c-1`.

Note how, since this method is supposed to find only one solution, once the recursive call returns `true`, the caller can also immediately return `true` without exploring the remaining possibilities. The first success will therefore cause the entire recursion to immediately roll back all the way to the top level where the breakdown of `n` into distinct cubes can then be read from the list `soFar`.

```
public static List<String> forbiddenSubstrings(String alphabet, int n, List<String> tabu)
```

Compute and return the list of all strings of length n that can be formed from the characters given in the list `alphabet`, but under the constraint that none of the strings listed in `tabu` are allowed to appear anywhere as substrings. For example, the list of all strings of length 3 constructed from alphabet `"ABC"` that do not contain any of the substrings `['AC', 'AA']` would be `['ABA', 'ABB', 'ABC', 'BAB', 'BBA', 'BBB', 'BBC', 'BCA', 'BCB', 'BCC', 'CAB', 'CBA', 'CBB', 'CBC', 'CCA', 'CCB', 'CCC']`. To facilitate automated testing, your method must return this list of strings in alphabetical order.

Following the exact same principle as the previous problem, this recursion is also easiest to implement as a private helper method

```
private static void forbiddenSubstrings(String alphabet, int n, List<String> tabu, String soFar, List<String> result)
```

that receives two additional arguments `soFar` and `result`, where `soFar` is the partial string generated so far (this should initially equal the empty string at the top level call) and `result` is the list of strings that the recursion has already discovered. The base case for failure is when the string `soFar` ends with one of the strings in the `tabu` list. The base case for success is when

`soFar.length() == n`, in which case the string soFar is added to `result`. Otherwise, the recursion should loop through the characters in the `alphabet`, appending each one to `soFar` for the recursive call.

Note how, unlike the previous method to find distinct cubes that was supposed to terminate after the first success, this private recursion method does not return anything to indicate success or failure, since even if it finds a solution right away, it still has to chug through the entire branching tree of possibilities to collect all the successes that can be found along the way.

Lab 0(G): ...Or We Shall All Crash Separately

JUnit: P2J7Test.java

The day of writing this lab, your instructor had to wait one hour in line outside Costco during the pandemic isolation. This wait gave him ample time to think up new good problems (this entire document consisted of only twenty problems before everything changed), and the situation naturally lended itself to the concept of queueing. Since various kinds of queues, along with the nifty data structures that efficiently implement them, are useful in many algorithms, this should be a perfect time to check out `LinkedList<E>`, the much faster alternative to `ArrayList<E>` for the operations needed to implement a **double-ended queue** (also called a "deque" for brevity) so that there is no need for random-access indexing inside the list, but the queue is accessed only from both ends.

As in all algorithmic problems, you should always choose the data structure that makes those operations fast that you are going to be doing a lot, whereas the cost of operations that you are not doing is irrelevant. (The same principle extends to all of engineering with the realization that non-existent parts waste no time or energy, and they will not be touched by either rust or moth.)

Often, such a double-ended queue is used as an ordinary "**first in, first out**" (FIFO) queue so that elements always enter the queue from the rear to then eventually exit from the front in the same order that they came in. But as long as it costs us nothing to have the ability to do something in reserve, we might as well have that possibility available for us.

The first problem (no doubt somewhat inspired, at least subconsciously, by the long wait in this line), takes us to the ancient world where a bunch of **zealots** (yes, literally, look it up), had been surrounded by Romans and trapped in a cave. Preferring mass suicide to capture and crucifixion, these men arranged themselves in a circle and drew lots for a random starting location. Counting from that location each time *k* steps ahead, the man...who...was...it! was killed and removed from this deadly game of eeny-meeny-miney-moe. The last man standing was then expected to fall on his own sword in a swift, sporting and gentlemanly fashion to join his fallen brothers. Josephus, our lovable rogue wandering hero who had somehow fallen into this bad crowd, had other plans of

escaping and, after some quick mental arithmetic, managed to finagle himself to this last position to live and tell us his tale.

Create a new class named `P2J7`, and inside that class, write the generic method

```
public static <T> List<T> josephus(List<T> men, int k)
```

that is given a list of men in order that they stand in the circle, the counting starting from the first man. This method should create another list that contains the men in the order that they were eliminated. Note that logic of this elimination does not depend on the element value or even its type but solely on its position, so we might as well make this method maximally generic by making the parameter type to be an arbitrary `List<T>`. (As we see in the lecture about generics, this is not at all the same thing as defining the parameter type to be `List<Object>` !)

As always, **this method should not modify the contents of its parameter list**, but create and return a new list object (you get to choose the concrete subtype yourself here) as the result. Note also that the step size `k` can be larger than the initial number of `men`, and yes, this does make a difference. For example, with `men = [ross, ted, ringo, alice, bob, rachel]` and `k=9`, the resulting elimination order is `[ringo, ross, ted, rachel, alice, bob]`.

Our second example of queueing comes from a more recent period in history, and is also an important algorithm from the era when humans still had to do the work of computers by hand. With slow and cranky humans playing computers (in fact, such a person is what the word "computer" originally meant as somebody's job title), we want to optimize every algorithm to reach the answer with the smallest number of computations.

Every ten years, the United States of America famously runs a nationwide census, and the seats to the congress (well, more accurately, to the House of Representatives) are distributed among the fifty states in proportion to their populations. The problem is rounding the number of seats that each state gets into an exact integer, since this task is not as clear-cut as it may initially seem. Simple rounding of each value to its closest integer does not work, because this method would not preserve the total number of congressional seats in the entire nation. How do you know which states are you supposed to round up, and which ones you round down? Paradoxically in situations like this, even though integers in general are much easier and more accurate to perform calculations than floating point numbers, the integer programming problem of finding a combination of integer values that satisfy the given constraints often turns out to be computationally far more complicated than solving that same problem while allowed to use arbitrary floating point numbers as part of the solution!

Fortunately, Huntington-Hill method will solve this integer programming problem the fairest possible way. Each state initially receives one seat, regardless of its population. The remaining seats are then distributed one seat at the time. Each seat always goes to the state whose **priority quantity** is currently the highest, as given by the formula $P^2 / (s(s+1))$, where P is the population of

that state and s is the number of seats the state has received so far. Since s is in the denominator of this fraction whereas the numerator P^2 remains constant, each additional seat decreases the priority of the state that receives it so that other states get ahead in this priority queue to receive seats after California, Texas and New York have taken the first seats.

(Yes, the Wikipedia page gives a bit different formula for priority quantity with a square root in it. But see what I did there, knowing that we do not actually care about the absolute values or the scales of these priority quantities but only about their mutual ordering, unaffected by the application of any monotonic function such as squaring to these quantities?)

Into the same class `P2J7` as the previous Josephus problem, write the method

```
public static int[] huntingtonHill(int[] population, int seats)
```

that, given the `population` in each state and the number of `seats` to distribute across these states, creates and returns a new array that contains the number of seats given to these states.

To write this method, first add the class `Fraction` from the class examples project into your labs BlueJ project so that you can perform your calculations with exact integer fractions without any possibility of integer overflow. Especially not even one floating point number or any calculation involving those abominations should appear anywhere inside your method.

(Would your instructor really be so mean as to design the tester method `testHuntingtonHill` to fuzz out populations that are too big to be accurately handled with floating point operations? Would that same instructor also ensure that some states end up having almost populations that differ only by one? Come on. At this stage of the studies, and especially if you have already taken my earlier course CCPS 109, do you really even have to ask?)

The best way to track of the relative priorities of the states (as identified by their position in `population` table) is to keep them inside a `PriorityQueue<Integer>` instance that uses a custom `Comparator<Integer>` strategy object whose method `compareTo(Integer a, Integer b)` makes its decision based on the current priorities of the states `a` and `b`. These priorities, of course, are kept up to date inside another array `Fraction[] priorities` that is defined as a local variable inside this method. To give out the next seat, `poll` the queue for the state whose priority is currently the highest. Update the priority for that state to the `priorities` array, and offer the state back into the priority queue.

As historically important as this algorithm is to the fate of the free world and our ability to keep rocking in it, it also has a more mundane everyday application in displaying rounded percentages from a list of numbers gleaned from some data. You know how you sometimes see a disclaimer "Due to rounding, displayed percentages may not add up to exactly 100" under some table of numbers and their computed percentages? Using the correct algorithm to display rounded

percentages eliminates any need for such disclaimers. (For example, to compute the percentages rounded to one decimal place, distribute 1,000 imaginary "seats" among your data items.)

Lab 0(H): More Integers Great and Small

JUnit: P2J8Test.java

In spirit of the earlier lab 0(E) that used the `BigInteger` type to represent solutions that would never fit inside an `int` or even a `long`, this week's lab brings you two more interesting problems dealing with "powerful" integer sequences.

```
public static void hittingIntegerPowers(int a, int b, int t, int[] out)
```

Let us define that two positive integers are "close enough for government work" if their absolute difference multiplied by the given **tolerance** level `t` is at most equal to the smaller of those two numbers. For example, the integers 2000 and 2007 are "close enough" when $t = 100$, since the difference 7 multiplied by 100 gives 700, which is less than 2000. On the other hand, the integers 2000 and 2050 would not be close enough for $t = 100$, although they would be close enough for a wider tolerance of $t = 10$. (Notice how $t = 100$ corresponds to the notion of being "within one percent", but without using any division or floating point numbers to compute this. The more narrow tolerance of $t = 100000$ would require these numbers to be within one thousandth of a percent of each other!)

Given two positive integers `a` and `b` and the desired tolerance `t`, this method should find and return find the smallest integer powers `pa` and `pb` so that when `a` is raised to the power of `pa`, and `b` is raised to the power of `pb`, the resulting two numbers are close enough for government work. Since these powers can get pretty big, as you can see in the table below, you need to perform these calculations using the `BigInteger` type. However, this is only for the actual powers that might end up having tens of thousands of digits; the exponents `pa` and `pb` are guaranteed to fit inside the `int` type for all our test cases.

a	b	t	Expected contents of out
2	4	100	{2, 1}
2	7	100	{73, 26}
3	6	100	{137, 84}
4	5	1000	{916, 789}
10	11	1000	{1107, 1063}
42	99	100000	{33896, 27571}

In Python, this function would simply return the answer as a two-tuple `(p1, p2)` of the exponents that fulfill the above requirement. Java does not have tuples in the language, so we have to return a two-element `int[]` object instead. Note that the return type of this method is `void` so that nothing actually is returned from this method. Instead, the method copies its answer into the two-element array given to it as the parameter `out`.

(This technique doesn't make much difference in this problem, but can be useful in methods that get called million, even billions of times. Instead of the method having to create a small new array for the result every time it is called, the same array object can be reused between all these calls, and this way lighten the load on the JVM garbage collector.)

```
public static BigInteger nearestPolygonalNumber(BigInteger n, int s)
```

As explained on the Wikipedia page "Polygonal Number", for every positive integer `s > 2`, the corresponding ***s*-gonal numbers** are an infinite sequence of integers whose *i*:th element is given by the formula $((s - 2)\ i^2 - (s - 4)\ i) / 2$. The Wikipedia formula uses the letter *n* instead of *i*, but *n* already means something else in this problem. (Besides, the letter *n* should only be used to refer to unknown fixed integers anyway, whereas the letters *i* and *j* are used for integers that advance as position counters.) For example, the sequence of "octagonal numbers" that springs forth from choosing *s* = 8 and plugging in the values 1, 2, 3, ... for *i*, starts with 1, 8, 21, 40, 65, 96, 133, 176...

Given `s` as an ordinary `int`, and `n` as a `BigInteger`, this method should find and return the *s*-gonal number whose value is closest to `n`. If `n` lies exactly halfway between two consecutive *s*-gonal numbers, this method should return the smaller of those two numbers.

`n`	`s`	Expected result (as `BigInteger`)
`5`	`3`	`6`
`27`	`4`	`25`
`450`	`9`	`474`
(ten billion)	`42`	`9999861561`
(one googol)	`91`	`100000000000000000000000000000000000000000000000000416332753518329478897755794704334003003544212420356`

This problem is best found with application of **repeated halving**, better known as **binary search**. First, find some position `b` in the sequence so that the `b`:th *s*-gonal number is greater or equal to `n`. You can just start with `b` equal to one, and keep multiplying it by ten until you reach your goal. Knowing some two positions `a` and `b` so that the correct answer is between these positions,

inclusive, compute the midpoint index `m = (a + b)` and look at the s-gonal number in that position. If that number is less than `n`, assign `a = m + 1`, and otherwise assign `b = m`. Either way, you are good. Once `a` and `b` are no more than one step apart, one more comparison gives you your answer. As you can see from the last line of the table, even the positions `a` and `b` can get so huge that they would never fit inside an ordinary `int`, let alone the polygonal numbers stored in those massive positions...

Lab 0(I): Squaring Off

JUnit: P2J9Test.java

The two problems in this lab produce results of type `boolean[]`, that is, arrays whose each element is a binary truth value. Since these individual bits can be packed eight per each memory byte, the storage of these truth bombs will be as compact as can be. Create the class named `P2J9` in your BlueJ labs project, and there the following two methods that each solve a problem that involves **squares** of positive integers. Given the upper limit `n` as a parameter, the result of both methods is a `boolean[]` array of `n` elements that reveals the answers to that problem for all natural numbers less than `n` in one swoop.

```
public static boolean[] sumOfTwoDistinctSquares(int n)
```

Determines which natural numbers can be expressed in the form $a^2 + b^2$ so that a and b are two distinct positive integers. In the `boolean` array returned as result, the `i`:th element should be `true` if and only if such a breakdown is possible. The infinite sequence of positive integers that allow such breakdown begins with 5, 10, 13, 17, 20, 25, 26, 29, 34, 37, 40, 41, 45, 50, 52, ...

You may have previously seen a version of this problem where the breakdown to two distinct squares was done separately for one number at the time. Given n, the question asked you to find positive integers a and b whose squares add up to n. However, since we are dealing with values in bulk, perhaps rather than looping through each n and find positive integers a and b whose squares add up to n separately for each n, it would be more productive to loop through all relevant pairs of a and b, and see which values of n can be built from those pairs...

```
public static boolean[] subtractSquare(int n)
```

Subtract a square is a simple impartial game of mental arithmetic. Two players take turns moving from the current number n into any smaller natural number that can be reached by subtracting some square of a positive integer from n without going below zero. For example, assuming the current n = 15, the player whose turn it is to play can and must move into one of the three possible numbers 14, 11, and 6 that respectively correspond to subtracting the square of one, two and three. The player who moves to zero wins the game, since the opponent can no longer make a move and therefore loses under the "normal play convention" of such games.

Using the standard terminology of this sort of impartial combinatorial games, a state is **winning** or **hot** if there exists a move into some cold state, and **losing** or **cold** if no such move exists. This rule makes $n = 0$ cold since no moves are possible there. The state $n = 1$ is hot since the move of subtracting 1 wins the game. The state $n = 2$ is again cold, since the only possible move from there leads to a cold state. The sequence of cold states in this game begins with 0, 2, 5, 7, 10, 12, 15, 17, 20, 22, 34, 39, ... and can be proven to continue infinitely from there. Some other games have only a finite number of losing states, since the moves of the game allow so much internal leeway that for any sufficiently large n past some threshold imposed by these rules, there must always exist some winning move. (For example, the trivial variation of this game called "Subtract any integer". But how about exciting variation where you subtract a prime? Subtract a Fibonacci number?)

This method should create and return an `n`-element array of `boolean` truth values so that the element in position `i` is `true` if the state `i` is hot, and false if it is cold. Notice that this method should be filling the result array from left to right, since to determine whether the current position is hot or cold, all the information needed to make that decision is already in the lower-numbered states whose heat values we have already filled in...

Lab 0(J): Similar Measures Of Dissimilarity

JUnit: P2J10Test.java

Consider two `boolean` arrays of equal length n, each filled with some combination of `true` and `false` elements. Various **dissimilarity** measures have been proposed to describe how "different" or "far apart" these two vectors of truth values stand from each other within the space of all 2^n possible such vectors of truth values. If both vectors are identical, these dissimilarity metrics will answer zero. Otherwise the answer will be some positive quantity whose magnitude corresponds to the dissimilarity between these vectors.

In this lab you are going to implement six such dissimilarity measures. All six methods are very similar, since they compute their results based on four counts n_{00}, n_{01}, n_{10} and n_{11} computed from the two argument vectors. The count n_{00} is the number of positions in which both arrays have the value `false`. The count n_{01} is the number of positions where the first array `v1` has the value `false` and the second array `v2` has the value `true`. The remaining two counts n_{10} and n_{11} are defined symmetrically, which entails that for any two n-element arrays `v1` and `v2`, these four counts must add to n. It would be good to extract the computation of these four counts into a separate helper method that these six methods then invoke as their first step. After that, each method is basically a one-liner that creates and returns the `Fraction` answer.

The formulas of how to compute each distance measure from the computed n_{ij} values can be found on the *Wolfram Mathematica* documentation page "Distance and similarity measures" section "Boolean data", in which you can click the Wolfram version of the function to read how that

particular dissimilarity measure works. (On the documentation pages for each separate function, click the section "Details" to expand it.) Alternatively, you can consult the page "Distance measures" that reveals how these six measures are merely the tip of the iceberg. (In the notation used in formulas on that page, $a = n_{00}$, $b = n_{10}$, $c = n_{01}$ and $d = n_{11}$.)

Each of these six measures will be a separate method inside the new class `P2J10` that you should create in your BlueJ labs project. Since the expected answer is always some integer fraction, we will again use our `Fraction` example type as the return type of these methods.

```
public static Fraction matchingDissimilarity(boolean[] v1, boolean[] v2)
public static Fraction jaccardDissimilarity(boolean[] v1, boolean[] v2)
public static Fraction diceDissimilarity(boolean[] v1, boolean[] v2)
public static Fraction rogersTanimonoDissimilarity(boolean[] v1, boolean[] v2)
public static Fraction russellRaoDissimilarity(boolean[] v1, boolean[] v2)
public static Fraction sokalSneathDissimilarity(boolean[] v1, boolean[] v2)
```

The following table lists a couple of argument arrays `v1` and `v2` with $n = 5$, along with their expected correct answers for each dissimilarity metric. To make this table a "bit" more compact, the truth value arrays are displayed as bit sequences so that `0` means `false` and `1` means `true`.

v1	v2	*Matching*	*Jaccard*	*Dice*	*RT*	*RR*	*SS*
01111	00100	3/5	3/4	3/5	3/4	4/5	6/7
01110	11100	2/5	1/2	1/5	4/7	3/5	2/3
10011	11100	4/5	4/5	2/3	8/9	4/5	8/9
01100	11010	3/5	3/4	3/5	3/4	4/5	6/7
11010	10100	3/5	3/4	3/5	3/4	4/5	6/7

Lab 1: Polynomial I: Basics

(Arooga! Arooga! Attention, young citizens! As an alternative to labs 1, 2 and 4 that make you implement the `Polynomial` data type and its arithmetic operations, some students might instead prefer to solve the similarly spirited labs 19, 20 and 21 to implement a `Distance` data type with its arithmetic operations. Experience over the past semesters has revealed that for some reason, getting all the little details of `Polynomial` correct so the code cleanly passes all of the JUnit tests with green check marks can sometimes be a frustrating slough. The operations for `Distance` are more straightforward, although they do require the use of the `Map<K,V>`, Java equivalent of Python dictionaries. Warnings for most of these pitfalls have also been added as clarifications to the specification of `Polynomial` below.)

JUnit: PolynomialTestOne.java

After learning the basics of Java language used in the imperative fashion, it is time to proceed to designing our own data types as **classes**, and writing the operations on these data types as **methods** that outside users of this abstract data type can call to get their needs fulfilled.

In the same spirit as the `Fraction` example class, your task in this (and the two following labs) is to implement the class `Polynomial` whose objects represent polynomials of variable *x* with integer coefficients, and their basic mathematical operations. If your math skills on polynomials have gone a bit rusty since you last had to use them somewhere back in high school, check out the page "Polynomials" in the "College Algebra" section of "Paul's Online Math Notes", the best and yet wonderfully concise online resource for undergraduate algebra and calculus that your instructor has ever found online.

As is the good programming style unless there exist good reasons to do otherwise, this class will be intentionally designed to be **immutable** so that `Polynomial` objects will not change their internal state after they have been constructed. Immutability in general offers countless advantages in programming, and is a goal worth pursuing wherever feasible, almost like that proverbial salmon whose price is so fantastically high that you will make a profit even if you don't happen to actually catch any! Even though those advantages might not yet be fully evident, that should not be a reason to stop you from benefiting from these gains! This is somewhat like how you still benefit from a glass of cool water on a hot day just the same, even if you could not begin to explain the biochemical effects that this water causes inside your body.

The `public` interface of `Polynomial` should consist of the following instance methods.

```
@Override public String toString()
```

Implement this method as your very first step to return some kind of meaningful, human readable `String` representation of `this` instance of `Polynomial`. This method is not subject to testing by the JUnit tests, so you can freely choose for yourself the exact textual representation that you would like this method to produce. Having this method will become **immensely** useful for debugging all the remaining methods that you will write inside `Polynomial` class!

```
public Polynomial(int[] coefficients)
```

The **constructor** that receives as argument the array of **coefficients** that together define the polynomial. For a polynomial of degree *n*, the `coefficients` array contains exactly *n* + 1 elements so that the coefficient of the term of order *k* is in the element `coefficients[k]`. For example, the polynomial $5x^3 - 7x + 42$ that will be used as an example in all of the following methods would be represented as the coefficient array `{42, -7, 0, 5}`.

Terms missing from inside the polynomial are represented by having a zero coefficient in that position. However, the **coefficient of the highest term of every polynomial should always be nonzero**, unless the polynomial itself is identically zero. If this constructor is given a coefficient array whose highest terms are zeroes, it should simply ignore such leading zero coefficients. For example, if given the coefficient array `{-1, 2, 0, 0, 0}`, the resulting polynomial would have the degree of only one, as if that coefficient array had been `{-1, 2}` all along without those pesky higher order zeros.

As the first warning of an unexpected pitfall lurking inside this problem, the zero polynomial should be encoded as the singleton coefficient array { `0` } instead of the empty array that the careless removal of all leading zeros will produce in this situation. The JUnit tester will produce zero polynomials by adding some randomly generated polynomials to their own negations. Even though the test class wasn't explicitly designed to do this, accidental **collisions** will happen even with perfect randomness! (In fact, especially with perfect randomness, unlike the "randomness" generated by humans that usually would not pass even the simplest statistical tests for randomness.)

To ensure that the `Polynomial` class is immutable so that no outside code can change the internal state of an object after its construction (at least not without resorting to underhanded Java tricks such as **reflection**), the constructor should not merely assign the reference to the `coefficients` array to the `private` field of coefficients, but it absolutely positively **must create a separate but identical *defensive copy* of the argument array, and store that defensive copy instead**. This technique ensures that the stored coefficients of the polynomial do not change if some outsider later changes the contents of the shared `coefficients` array that was passed as the constructor argument. The JUnit test `TestPolynomialOne` will intentionally try to mess up some coefficient arrays later to make your code fail.

```
public int getDegree()
```

Returns the degree of `this` polynomial, that is, the exponent of its highest order term that has a nonzero coefficient. For example, the previous polynomial has degree 3. All constant polynomials, including the zero polynomial, have a degree of zero.

```
public int getCoefficient(int k)
```

Returns the coefficient for the term of order *k*. For example, the term of order 3 of the previous polynomial equals 5, and the term of order 0 equals 42. This method should work correctly even when *k* is negative or greater than the actual degree of the polynomial, and simply return zero in such cases of nonexistent terms.

```
public long evaluate(int x)
```

Evaluates the polynomial using the value `x` for the unknown symbolic variable of the polynomial. For example, when called with `x = 2` for the previous example polynomial whose coefficients are `{42, -7, 0, 5}`, this method would return `68`.

Your method does not have to worry about potential integer overflows, but may assume that the final and intermediate results of this computation will always stay within the range of the primitive data type `long`. Interested students can perform this evaluation using the Horner's rule that is both faster and more numerically stable than the naive evaluation. However, make sure that even if you use the evaluation where each term is computed separately, make sure that you are not a "Shlemiel" who computes x^5 from scratch with a loop, even though x^4 from the previous round is right there and needs only one more multiplication to become the desired x^5...

Lab 2: Polynomial II: Arithmetic

JUnit: PolynomialTestTwo.java

This lab continues with the `Polynomial` class from the previous lab by adding new methods for polynomial arithmetic to its source code. (No inheritance or polymorphism is yet taking place in this lab.) Since the class `Polynomial` is designed to be immutable, none of the following methods should modify the objects `this` or `other` in any way, but always return the result of that arithmetic operation as a new `Polynomial` object created inside that method.

```
public Polynomial add(Polynomial other)
```

Creates and returns a new `Polynomial` object that represents the result of polynomial addition of the two polynomials `this` and `other`. Make sure that the coefficient of the highest term of the result is nonzero, so that adding two polynomials $5x^{10} - x^2 + 3x$ and $-5x^{10} + 7$, each having a degree of 10, produces the result $-x^2 + 3x + 7$ that has a degree of only 2, instead of 10. As the JUnit test class generates random polynomials and adds them together, occasionally some polynomial will be added to its own negation, and your code must handle this important edge case correctly.

```
public Polynomial multiply(Polynomial other)
```

Creates and returns a new `Polynomial` object that represents the result of polynomial multiplication of two polynomials `this` and `other`. You can perform this multiplication by looping through all possible pairs of terms between the two polynomials and adding their products together into the result where you combine the terms of equal degree together into a single term.

You should again note that multiplication can cancel out some internal terms whose coefficients were originally nonzero in both polynomials. For example, $x^2 + 3$ multiplied by $x^2 - 3$ gives the result $x^4 - 9$ whose second order term ends up with a coefficient of zero. Since the product of two nonzero

integers can never be zero, the highest term can never get cancelled out this way, so you always know the degree of the result right away once you know the degrees of the originals.

Lab 3: Extending an Existing Class

JUnit: AccessCountArrayListTest.java

In the third lab, you get to practice using **class inheritance** to create your own custom subclass versions of existing classes in Java with new functionality that did not exist in the original superclass. Your third task in this course is to use inheritance to create your own custom subclass `AccessCountArrayList<E>` that `extends` the good old workhorse `ArrayList<E>` from the Java Collection Framework. This subclass should maintain an internal data field `int count` to keep track of how many times the methods `get` and `set` have been called. (One counter keeps the simultaneous count for both of these methods together.)

You should override the inherited `get` and `set` methods so that both of these methods first increment the access counter, and only then call the superclass version of that same method (use the prefix `super` in the method call to make this happen), returning whatever result that superclass version returned. In addition to these overridden methods inherited from the superclass, your class should define the following two brand new methods:

```
public int getAccessCount()
```

Returns the count of how many times the `get` and `set` methods have been called for `this` object.

```
public void resetCount()
```

Resets the access count field of `this` object back to zero.

Lab 4: Polynomial III: Comparisons

JUnit: PolynomialTestThree.java

This lab continues modifying the source code for the `Polynomial` class from the first two labs to allow **equality** and **ordering** comparisons to take place between objects of that type. Modify your `Polynomial` class signature definition so that it `implements Comparable<Polynomial>`. Then write the following methods to implement the equality and ordering comparisons.

```
@Override public boolean equals(Object other)
```

Returns `true` if the `other` object is also a `Polynomial` of the exact same degree as `this`, and that the coefficients of `this` and `other` polynomials are pairwise equal. If the `other` object is anything else, this method should return `false`.

(To save you some time, you can actually implement this method after implementing the method `compareTo` below, since once that method is available, the logic of equality checking will be a trivial one-liner after the `instanceof` check.)

```
@Override public int hashCode()
```

Whenever you override the `equals` method in any subclass, you should also override the `hashCode` method to ensure that two objects that are considered equal by the `equals` method will also have equal integer hash codes. This method computes and returns the *hash code* of this polynomial, used to store and find this object inside some instance of `HashSet<Polynomial>`, or some other *hash table* based data structure.

You get to choose for yourself the hash function that you implement, but like all hash functions, the result should depend on the degree and all of the coefficients of your polynomial. The hash function absolutely **must** satisfy the contract that whenever `p1.equals(p2)` holds for two `Polynomial` objects, then also `p1.hashCode() == p2.hashCode()` holds for them.

Of course, since this entire problem is so common and it seems silly to force everyone to reinvent the same wheel once again, these days you can use the method `hash` in the `java.util.Objects` utility class to compute a good hash value for your coefficients.

```
public int compareTo(Polynomial other)
```

Implements the **ordering comparison** between `this` and `other` polynomials, as required by the interface `Comparable<Polynomial>`, allowing the instances of `Polynomial` to be *sorted* or stored inside some instance of `TreeSet<Polynomial>`. This method returns `+1` if `this` is greater than `other`, `-1` if `other` is greater than `this`, and returns a `0` if both polynomials are equal in the sense of the `equals` method.

A total ordering relation between polynomials can be defined by many possible different rules. We shall use an ordering rule that says that **any polynomial of a higher degree is automatically greater than any polynomial of a lower degree**, regardless of their coefficients. For two polynomials whose degrees are equal, the result of the order comparison is determined by **the highest-order term for which the coefficients of the polynomials differ**, so that the polynomial with a larger such coefficient is considered to be greater in this ordering.

Be careful to ensure that this method ignores the leading zeros of high order terms if you have them inside your polynomial coefficient array, and that **the ordering comparison criterion is precisely the one defined in the previous paragraph**. A common bug here is to loop through the

coefficients from lowest to highest, instead of looping them down from highest to lowest. This bug is particularly nasty, since looping through the coefficients in the wrong direction will still produce a perfectly legal total ordering between polynomials that the collections of type `TreeSet<Polynomial>` would be perfectly happy to use. It just would not be the total ordering that the JUnit test class is expecting to see, but how can the student know that if they missed it in the reading of the specification?

Lab 5: Introduction To Swing

Historically, **graphical user interface (GUI) programming** was the first **killer app** of object oriented programming that propelled the entire paradigm into the mainstream after two decades of purely theoretical academic research. (Most things in computer science are at least a decade or two older than what the majority of even working programmers might casually assume, and the author suspects the same principle to hold in all walks of life, not just coding.) Inheritance and polymorphism make the task of creating new customized GUI components so straightforward that in practice, any GUI programming done in a non-OO low level language first has to use that language to build up a crude simulation of these higher level mechanisms, nicely illustrating both Greenspun's Tenth Rule and Blub Paradox in practice.

In this lab, your task is to create a custom Swing GUI component `Head` that extends `JPanel`. To practice working with Java graphics, this component should display a simple human head that, to also practice the Swing *event handling* mechanism, reacts to the mouse cursor entering and exiting the surface of that component. You should copy-paste a bunch of boilerplate code from the example class `ShapePanel` into this class.

Your class should contain the field `private boolean mouseInside` that is used to remember whether the mouse cursor is currently over your component, and the following methods:

```
public Head()
```

The constructor of the class should first set the preferred size of this component to be 500-by-500 pixels, and then give this component a decorative raised bevel border using the utility method `BorderFactory.createBevelBorder`. Next, this constructor adds a `MouseListener` to this component, this event listener object constructed from the inner class `MyMouseListener` that `extends MouseAdapter`. Override both methods `mouseEntered` and `mouseExited` inside your event listener inner class to first set the value of the field `mouseInside` accordingly and then call `repaint`.

If you `implement MouseListener` instead of saying `extends MouseAdapter`, you need to provide the empty implementations also for all the other methods in that interface, even if those methods end up doing nothing at all. The `MouseAdapter` superclass provides these empty

implementations for its subclasses for free. (Nowadays all these methods could have been `default` methods with do-nothing empty bodies defined in the interface itself.)

`@Override public void paintComponent(Graphics g)`

Renders some kind of abstract cartoon image of a simple human head on the surface of this component. This is not an art class so this head does not have to look fancy, but a couple of simple rectangles and ellipses suffice. Of course, interested students with a more artistic bent might want to check out more complicated shapes from the package `java.awt.geom` to generate a prettier image, or perhaps even using `Image` objects whose contents are read from some GIF or JPEG file. (Back in the author's day and age, GIF and JPEG were the main image formats. You kids have your new fancy kinds of formats these days, while in our youth, by Jove, we had to grind our teeth through minutes of modem screech just to catch a glimpse of some nice lady's bare ankle...)

If the value of the field `mouseInside` equals `true`, the eyes of this head should be drawn to be open, whereas if `mouseInside` equals `false`, the eyes should be drawn to be closed. (As long as the eyes are somehow noticeably visually different based on the mouse entering and exiting the component surface, that is enough for this lab.)

To admire your interactive Swing component on the screen, write a separate `HeadMain` class that contains a `main` method that creates a `JFrame` that contains four separate `Head` components arranged in a neat 2-by-2 grid using the `GridLayout` layout manager. Wiggle your mouse cursor from top of one component onto another to watch the eyes open and close with these mouse movements.

Lab 6: Processing Text Files, Part I

JUnit: WordCountTest.java

This lab has you try out reading in text data from an instance of `BufferedReader`, and then performing computations on that data in the spirit of "poor man's data science". To practice working inside a small but entirely proper object-oriented **framework**, we shall design an entire class hierarchy that reads in text one line at the time and performs some operation for each line. Having read in all the lines, some kind of result is emitted in the end. The subclasses must then implement those operations by overriding these **template methods** that implement the **stages** of this abstract algorithm.

To allow this processing to return a result of arbitrary type that can be freely chosen by the users of this class, the abstract superclass of this little framework is also defined to be **generic**. Start by creating the class `public abstract class FileProcessor<R>` that defines the following three `abstract` methods:

```
protected abstract void startFile();
protected abstract void processLine(String line);
protected abstract R endFile();
```

The class should also have one concrete `final` method that is therefore guaranteed to be the same for all future subclasses in this entire class hierarchy:

```
public final R processFile(BufferedReader in) throws IOException
```

This method should first call the method `startFile`. Next, it reads all the lines of text coming from `in` one line at the time in some kind of suitable loop, and calls the method `processLine` passing it as argument each line that it reads in. Once all incoming lines have been read and processed, this method should finish up by calling the method `endFile`, and return whatever the method `endFile` returned.

This abstract superclass defines a template for a class that performs some computation for a text file that is processed one line at the time, returning a result whose type `R` can be freely chosen by the concrete subclasses of this class. Subclasses that purport different text processing operations can override the three template methods `startFile`, `processLine` and `endFile` methods in different ways to implement different computations on text files.

As the first application of this mini-framework (and indeed it is a framework by definition, **since you will be writing methods for this framework to call, instead of the framework offering methods for you to call**), you will emulate the core functionality of the Unix command line tool `wc` that counts how many characters, words and lines the given text file contains. Those who have taken the Unix and C programming course CCPS 393 should recognize this handy little text processing tool from there. But even if you have not yet taken that course, you can still implement this handy little tool based on the following specification of its behaviour.

The existence of `wc` in all Unix systems by itself proves its usefulness as it solves some important problem that tends to come up a lot in different problem domains. This deceptively simple piece of software therefore *must* contain some kind of important thing for us to learn, even though it is not immediately obvious what that thing might be. Our inability to detect its importance does not magically cause this importance to be non-existent! No matter which way you choose to design and implement this tool, something educational and computationally interesting will definitely happen during this process.

Create a class `WordCount` that `extends FileProcessor<List<Integer>>`. This class should contain three integer fields to keep track of these three counts, and the following methods:

```
protected void startFile()
```

Initializes the character, word and line counts to zero.

```
protected void processLine(String line)
```

Increments the character, word and line counts appropriately. In the given `line`, every character increments the count by one, regardless of whether that character is a whitespace character. To properly count the words in the given line, count the non-whitespace characters for which the previous character on that `line` is a whitespace character. To prevent the "loop and a half" situation where the first character has to be handled separately because it has no predecessor character, just imagine that the first character in `line` was preceded by an invisible whitespace character.

Furthermore, you should use the utility method `Character.isWhitespace` to test whether some character is a whitespace character, since the Unicode standard defines quite a lot more whitespace characters than most people could name, or even see any need for their existence. Since that mythical cadre of Top Men in their horn-rimmed glasses has already done all the legwork in tabulating all the properties of characters that exist for humanity, we might as well always reuse their work, instead of trying to badly reinvent the parts that we expect to see in the execution of our program. Since the classes and methods in standard libraries have been mercilessly pummeled by user code for decades for various purposes, all the edge case and corner case bugs lurking inside them have surely been brought to light by now.

```
protected List<Integer> endFile()
```

Creates and returns a new `List<Integer>` instance (you get to choose the concrete subtype of this result object for yourself) that contains exactly three elements; the character, word and line counts, in this order.

Lab 7: Concurrency In Animation

To paraphrase the immortal wisdom of Jack Handey, just as bees swarm about to protect their nests, so will I "swarm about" to protect my nest of knowledge eggs. This lab has you create a Swing component that displays a real-time animation using Java concurrency to execute an **animation thread** that runs at guaranteed constant pace independent of what the human user might happen to be doing with that particular component or elsewhere. This thread will animate a classic **particle field** where a **swarm** of thousands of independent little *particles* buzzes randomly around the component.

This same basic architecture could then, with surprisingly little additional modification, be used to implement some real-time game, with this animation thread moving the game entities according to the rules of the game 50 frames per second, and the event listeners giving orders to the game entity that happens to represent the human player. As seems to be the case in all walks of life, most things that you expect to be complex and difficult usually turn out to be surprisingly much simpler than

you would have assumed, whereas the things that you expected to be simple and easy often turn out to be dizzyingly complex once you crack them open and really try to implement them yourself properly without any shortcuts or hand waving! (This principle works even better in programming where some guy somewhere probably has solved the complex thing already, so we can just copy-paste this work and be done with that by the time for dinner.)

First, create a class `Particle` whose instances represent individual particles that will randomly around the two-dimensional plane. Each particle should remember its *x*- and *y*-coordinates on the two-dimensional plane and its heading as an angle expressed as **radians**, stored in three data fields of the type `double`. This class should also have a random number generator shared between all objects, and a shared field `BUZZY` that defines how random the motion is.

```
private static final Random rng = new Random();
private static final double BUZZY = 0.7;
```

The class should then have the following methods:

```
public Particle(int width, int height)
```

The constructor that places `this` particle in random coordinates inside the box whose possible values for the *x*-coordinate range from 0 to `width`, and for the *y*-coordinate from 0 to `height`. The initial heading is taken from the expression `Math.PI * 2 * rng.nextDouble()`.

```
public double getX()
public double getY()
```

The accessor methods for the current *x*- and *y*-coordinates of `this` particle.

```
public void move()
```

Updates the value of `x` by adding `Math.cos(heading)` to it, and updates the value of `y` by adding `Math.sin(heading)` to it. After that, the `heading` is updated by adding the value of the expression `rng.nextGaussian() * BUZZY` to it.

Having completed the class to represent individual particles, write a class `ParticleField` that extends `javax.swing.JPanel`. The instances of this class represent an entire field of random particles. This class should have the following `private` instance fields.

```
private boolean running = true;
private java.util.List<Particle> particles =
   new java.util.ArrayList<Particle>();
```

The class should have the following `public` methods:

`public ParticleField(int n, int width, int height)`

The constructor that first sets the preferred size of this component to be `width`-by-`height`. Next, it creates `n` separate instances of the class `Particle` and places them in the `particles` list.

Having initialized the individual particles, this constructor should create and launch one new `Thread` using a `Runnable` argument whose method `run` consists of one `while(running)` loop. (Don't even think about creating a separate thread for each particle: this is Java, not Go.) The body of this loop should first `sleep` for 20 milliseconds. After waking up from its sleep, it should loop through all the `particles` and call the method `move()` for each of them. Then call `repaint()` and go to the next round of the while-loop.

`@Override public void paintComponent(Graphics g)`

Renders `this` component by looping through particles and rendering each one of them as a 3-by-3 pixel rectangle to its current *x*- and *y*-coordinates.

`public void terminate()`

Sets the field `running` of `this` component to be `false`, causing the animation thread to terminate in the near future.

To admire the literal and metaphorical buzz of your particle swarm, write another class `ParticleMain` that contains a `main` method that creates one `JFrame` instance that contains a `ParticleField` of size 800-by-800 that contains 2,000 instances of `Particle`. Using the `main` method of the example class `SpaceFiller` as a model of how to do this, attach a `WindowListener` to the `JFrame` so that the listener's method `windowClosing` first calls the method `terminate` of the `ParticleField` instance shown inside the `JFrame` before disposing that `JFrame` itself.

Try out the effect that different values between 0.0 and 10.0 for `BUZZY` have to the motion of your particles. When `BUZZY` is small, the motion of each particle tends to maintain its current direction akin to the way actual physical particles behave, whereas larger values of `BUZZY` adjust the motion closer to random Brownian motion.

Particle systems are often used in games to create interesting animations of phenomena such as explosions or fire that would be otherwise difficult to simulate and render as a bunch of polygons. Giving particles more intelligence and mutual interaction such as making some particles follow or avoid certain other particles can produce surprisingly natural and lifelike emergent animations.

Lab 8: Processing Text Files, Part II

JUnit: TailTest.java

In this lab, we return to the `FileProcessor<R>` framework from Lab 6 for writing tools that process text files one line at the time. This time this framework will be used to implement another Unix command line tool `tail` that extracts the last `n` lines of its input and discards the rest. Write a class `Tail` that extends `FileProcessor<List<String>>` and has the following methods:

```
public Tail(int n)
```

The constructor that stores its argument `n`, the number of last lines to return as a result, into a private data field. In this lab, you have to choose for yourself what instance fields you need to define in your class to make the required methods work.

```
@Override public void startFile()
```

Start processing a new file from the beginning. Nothing to do here, really.

```
@Override public void processLine(String line)
```

Process the current `line`. Do something intelligent here. Be especially careful not to be a "Shlemiel" so that your logic of processing each line always has to loop through all of the previous lines that you have collected and stored so far. To avoid being a "Shlemiel", note that the `List<String>` instance that you create and return does not necessarily have to be specifically an `ArrayList<String>`, but perhaps some other subtype of `List<String>` that allows O(1) operations at both of its ends would be far more appropriate.

```
@Override public List<String> endFile()
```

Returns a `List<String>` instance that contains precisely the `n` most recent lines that were given as arguments to the method `processLine` in the same order that they were originally read in. If fewer than `n` lines have been given to the method `processLine` since the most recent call to the method `startFile`, this list should contain as many lines as have been given.

Lab 9: Lissajous Curves

In this lab you are going to write a Swing component that displays a Lissajous curve on its surface. The component should allow the user to control the parameters *a*, *b* and *delta* that define the shape of this curve by entering their values into three `JTextField` components placed inside this component. This lab is therefore an exercise in both component graphics and using other

specialized components such as text fields on the surface of a Swing component, and reacting to the events of the said components with an update of the display. Write a class `Lissajous extends JPanel`, and the following methods in it:

```
public Lissajous(int size)
```

The constructor that sets the preferred size of this component to be `size`-by-`size` pixels. Then, three instances of `JTextField` are created and added inside `this` component. Initialize these text fields with values 2, 3 and 0.5, with some extra spaces added to these strings so that these text fields have a decent initial size for the user to enter other numbers to try out. Define your own subtype of `ActionListener` as a nested class whose `actionPerformed` method simply calls `repaint` for this component. The same instance of this class can listen to all three text fields, since the reaction to the event is always identical for all three sources of said events.

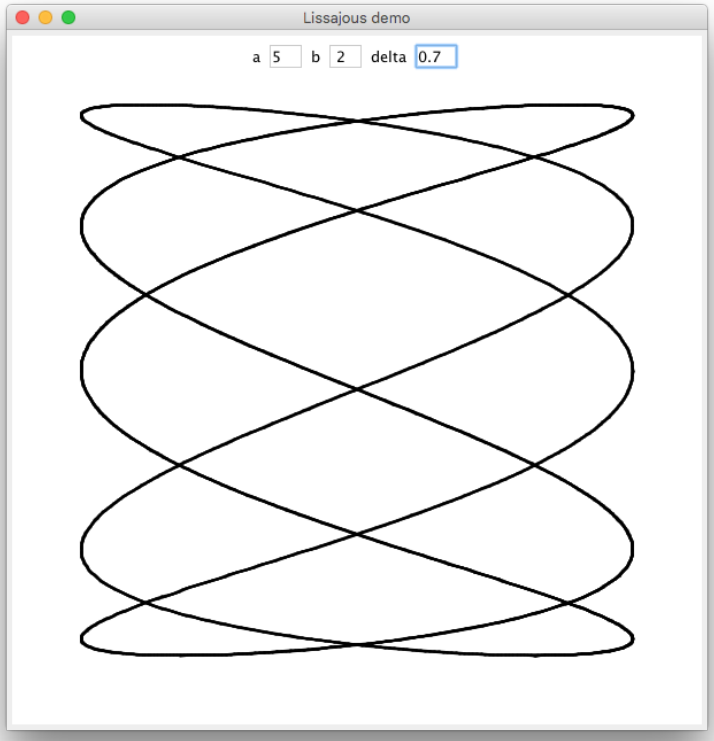
```
@Override public void paintComponent(Graphics g)
```

Renders the Lissajous curve on the component surface, using the current values for *a*, *b* and *delta* that it first reads from the previous three text fields. This method should consist of a for-loop whose loop variable `double t` goes through the values from `0` to `(a + b) * Math.PI` using some suitably small increments. In the body of the loop, compute the coordinates `x` and `y` of the current point using the formulas

```
x = size/2 + 2*size/5 * Math.sin(a * t + delta);
y = size/2 + 2*size/5 * Math.cos(b * t);
```

and draw a line segment from the current point to the previous point.

To admire your Lissajous curve and the effect *a*, *b* and *delta* on its shape, as if you were trapped inside the lair of professor Ilkkarius, the mad scientist of some 1970's dystopian science fiction movie, create a separate class `LissajousMain` whose `main` method creates a `JFrame` that contains your `Lissajous` component. The end result should look somewhat like this:



Motivated students can take on as an extra challenge to make the displayed image look smoother by eliminating some visual jagginess, and possibly even make this rendering look more artistic in other ways such as rendering more complex curves and shapes to the positions in the path carved into the plane by these equations. For example, instead of subdividing the curve into linear line segments, you could rather subdivide it into much more suave and smooth cubic curves that connect seamlessly at the point where the previous piece of the curve left off...

Lab 10: Computation Streams

JUnit: StreamExercisesTest.java

This lab teaches you to think about computational problems and then implement them in the functional programming framework of **computation streams** introduced in Java 8. Therefore in this lab, you are **absolutely forbidden** to use any conditional statements (either `if` or `switch`), loops (either `for`, `while` or `do-while`) or even recursion. Instead, all computation **must** be implemented using **only Java computation streams and their operations!**

In this lab, we shall also check out the **Java NIO framework** for better file operations than those offered in the old package `java.io` and the class `File`. Your methods receive a `Path` as an argument, referring to the text file in your file system whose contents your methods will then

process with computation streams. Write both of the following static methods inside a new class named `StreamExercises`.

```
public static int countLines(Path path, int thres) throws IOException
```

This method should first use the utility method `Files.lines` to convert the given `path` into an instance of `Stream<String>` that produces the lines of this text file as a stream of strings, one line at the time. The method should then use the method `filter` to keep only those whose `length` is greater or equal to the threshold value `thres`, and in the end, return a count of how many such lines the file contains.

```
public static List<String> collectWords(Path path) throws IOException
```

This method should also use the same utility method `Files.lines` to first turn its parameter `path` into a `Stream<String>`. Each line should be converted to lowercase and broken down to individual words that are passed down the stream as separate `String` objects (the stream operation `flatMap` will be handy here). Split each line into its individual words with the method `split` in `String` with the word separator regex `"[^a-z]+"`. During the stages of the computation stream that follow this one, discard all empty words, and `sort` the remaining words in alphabetical order. Multiple consecutive occurrences of the same word should be removed (you can simply use the stream operation `distinct`, or if you want to do this yourself the hard way as an exercise, write a custom `Predicate<String>` subclass whose method `test` accepts its argument if and only if it was the first one or was distinct from the argument of the previous call), and to wrap up this computation stream, `collect`ed into a `List<String>` as the final answer.

Lab 11: Prime Numbers and Factorization

JUnit: PrimesTest.java

"*All programming is an exercise in caching.*" (Terje Mathisen)

This lab tackles the classic and important problem of **prime numbers**, positive integers that are exactly divisible only by one and by themselves. Create a class `Primes` to contain the following three `static` methods to quickly produce and examine integers for their primality.

```
public static boolean isPrime(int n)
```

Checks whether the parameter integer `n` is a prime number. For this lab, it is sufficient to use the primality test algorithm that simply loops through all potential integer factors up to the square root of `n` and stops as soon as it finds one factor. (If an integer has any nontrivial factors, at least one of these factors has to be less than or equal to its square root, so it is pointless to look for any such factors past that point if you haven't already found some.)

```
public static int kthPrime(int k)
```

Find and return the k:th element (counting from zero, as usual in computer science) from the infinite sequence of all prime numbers 2, 3, 5, 7, 11, 13, 17, 19, 23, ... This method may assume that k is nonnegative.

```
public static List<Integer> factorize(int n)
```

Compute and return the list of **prime factors** of the positive integer n. The exact subtype of the returned `List` does not matter as long as the returned list contains the prime factors of n in **ascending sorted order**, each prime factor listed exactly as many times as it appears in the product. For example, when called with the argument `n = 220`, this method would return some kind of `List<Integer>` object that prints out as `[2, 2, 5, 11]`.

To make the previous methods maximally speedy and efficient, this entire exercise is all about **caching and remembering the things that you have already found out** so that you don't need to waste time finding out those same things later. As the famous space-time tradeoff principle of computer science informs us, you can sometimes make your program run faster by making it use more memory. Since these days we are blessed with ample memory to splurge around with, we are usually happy to accept this tradeoff.

This class should maintain a private instance of `List<Integer>` in which you store the sequence of the prime numbers that you have already generated, which then allows you to quickly look up and return the k:th prime number in the method `kthPrime`, and also to quickly iterate through the prime numbers up to the square root of n inside the method `isPrime`. You can use the example program `primes.py` from the instructor's Python version of CCPS 109 as a model of this idea. It would also be a good idea to write a `private` helper method `expandPrimes` that finds and appends new prime numbers to this list as needed by the `isPrime` and `kthPrime` methods.

To speed up the `factorize` method, you can use an instance of `Map<Integer, Integer>` that remembers, for each integer key n stored in it, some non-trivial factor of n that you have discovered earlier. If this map tells you that the integer n has a factor p, the prime factorization of n consists of p followed by the prime factorization of `n / p`. (To save memory, you can impose a **cutoff threshold** so that prime factors that are less than that threshold are not stored in this map, since they would be quickly found with the factor searching loop anyway.)

To qualify for the two lab marks, the automated test must successfully **finish all three tests within one minute** when run on the average off-the-shelf desktop computer from the past five years. Speed is the essence of this lab.

Lab 12: Interesting Prime Number Sequences

JUnit: PrimeGensTest.java

In Python, lazy sequences can be programmatically produced with **generators**, special functions that `yield` their results one element at the time and always continue their execution from the exact point where they left off in the previous call, instead of always starting their execution of the function body from the beginning the way ordinary functions do in both Python and Java. The Java language does not have generators (at least not around version 8), but the idea of lazy **iteration** over computationally generated infinite sequences is still very much worth learning, and fits snugly into the `Iterator<E>` class hierarchy of Java Collection Framework.

This lab continues the work from the previous lab by using its methods `isPrime` and `kthPrime` as helper methods to produce the subsequence of all prime numbers that satisfy some additional requirements. Create a new class `PrimeGens` inside which you write **no fields or methods whatsoever**, but three `public static` nested classes. Each of these classes acts as an **infinite iterator of integers** that are not explicitly stored inside some collection, but are generated **in a lazy fashion** by computational means only when the `next` number is requested by the user code.

Each of these three nested classes should have the two basic methods defined in the interface `Iterator<Integer>`. The method `public boolean hasNext()` should always return `true`, since all these sequences are infinite for our purposes. (It is currently unknown whether the twin primes sequence really is infinite, but at least this sequence is known to be longer than the proverbial year of famine so that we can here treat it as being infinite within the limited range of four-byte integers.) The method `public Integer next()` then always generates the next element of that sequence on command.

First, to help you get started with the required three nested classes, below is a complete model implementation of one such class that produces all palindromic prime numbers, for you to use as a model. Your three classes have the same structure, but of course use different logic inside the method `next` to find and return the next prime number that has the desired properties.

```
public static class PalindromicPrimes implements Iterator<Integer> {
  private int k = 0; // Current position in the prime sequence
  public boolean hasNext() { return true; } // Infinite sequence
  public Integer next() {
    while(true) {
      int p = Primes.kthPrime(k++);
      String digits = "" + p;
      if(digits.equals(new StringBuilder(digits).reverse().toString())) {
        return p;
      }
```

```
    }
  }
}
```

Having carefully studied that code until you can explain what each line does, the three nested classes that you write inside class `PrimeGens` for you to write are as follows. (You can quickly and easily test your implementations with a simple `main` method that first creates an instance of the said iterator, and then uses a `for`-loop to print out the first couple of dozen elements produced by that iterator, with you comparing the said sequence to the correct initial sequences given below.)

```
public static class TwinPrimes implements Iterator<Integer>
```

Generates all twin primes, that is, prime numbers `p` for which `p+2` is also a prime number. This sequence begins 3, 5, 11, 17, 29, 41, 59, 71, 101, 107, 137, 149, 179, 191, 197, 227, 239, 269, 281, 311, ...

```
public static class SafePrimes implements Iterator<Integer>
```

Generates all safe primes, that is, primes that can be expressed in the form `2*p+1` where `p` is also a prime number. This sequence begins 5, 7, 11, 23, 47, 59, 83, 107, 167, 179, 227, 263, 347, 359, 383, 467, 479, 503, 563, 587, 719, 839, 863, ...

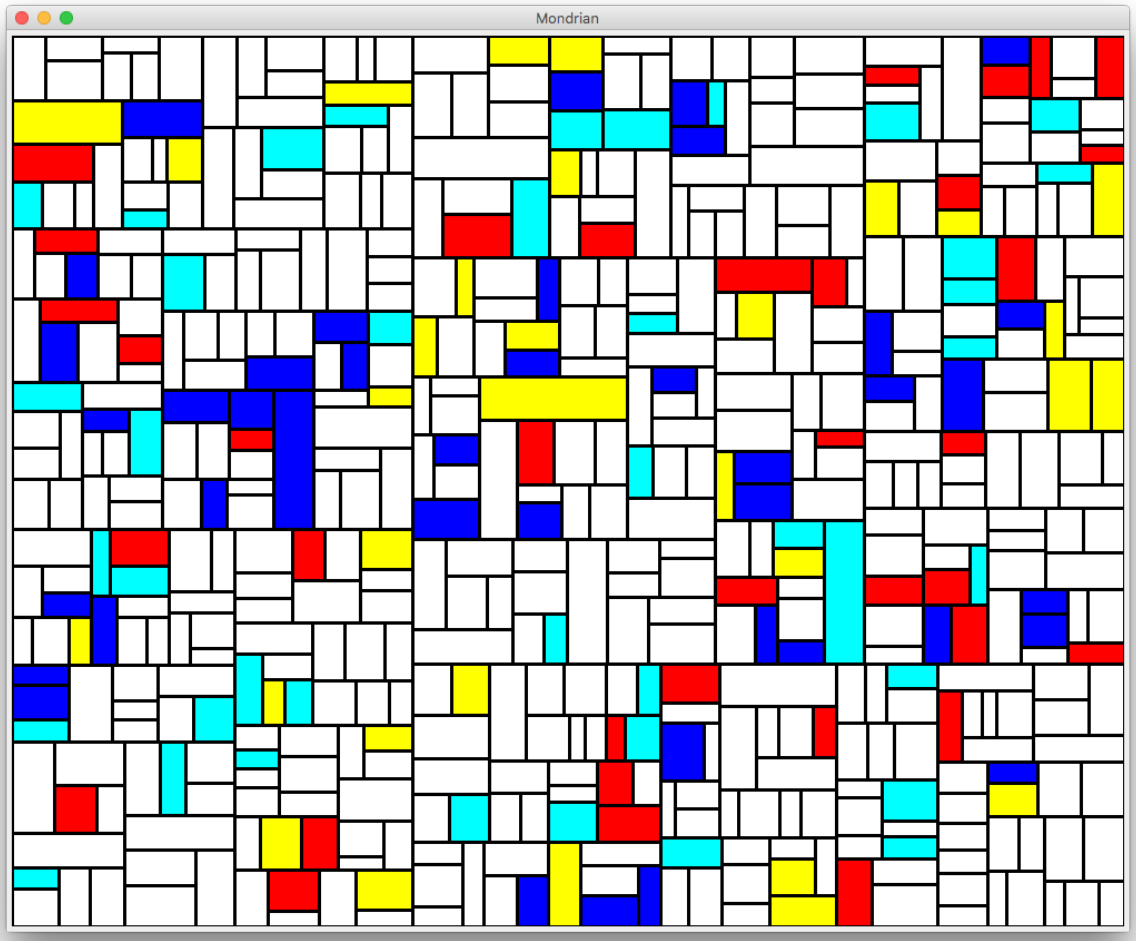
```
public static class StrongPrimes implements Iterator<Integer>
```

Generates all strong primes, that is, prime numbers `p` that are larger than the mathematical average of the previous and next prime numbers around `p` in the sequence of all prime numbers. This sequence begins 11, 17, 29, 37, 41, 59, 67, 71, 79, 97, 101, 107, 127, 137, 149, 163, 179, 191, 197, 223, 227, 239, 251, ...

Lab 13: Recursive Mondrian Art

Your instructor actually used this assignment in the old version of this course almost a decade ago, but now that this very problem has been included in Nifty Assignments, perhaps the time has come to revisit its timeless message, especially since during these years between, your instructor got himself severely bitten by the subdivision and fractal art bugs...

This lab combines graphics with recursion by constructing a simple **subdivision fractal** that resembles the famous works of abstract art by Piet Mondrian. An example run of the instructor's model solution produced the random image captured for eternity in the following screenshot, to which your images should be reasonably similar in style and spirit, using controlled randomness to generate a new piece of abstract visual art every time your program is run.



Write a class `Mondrian` that `extends JPanel`, with the following fields:

```
// Cutoff size for when a rectangle is not subdivided further.
private static final int CUTOFF = 40;
// Percentage of rectangles that are white.
private static final double WHITE = 0.75;
// Colours of non-white rectangles.
private static final Color[] COLORS = {
  Color.YELLOW, Color.RED, Color.BLUE, Color.CYAN
};
// RNG instance to make the random decisions with.
private Random rng = new Random();
// The Image in which the art is drawn.
private Image mondrian;
```

After this, the class should have the following methods:

`public Mondrian(int w, int h)`

The constructor for the class, with the desired width and the height of the resulting artwork given as constructor arguments. Initialize the field `mondrian` to a new `BufferedImage` of size `w` and `h`, and ask that image for its `Graphics2D` object to be passed as the last argument of the top-level call of the recursive `subdivide` method below.

`public void paintComponent(Graphics g)`

Draws the `mondrian` image created in the constructor on the surface of this component.

`private void subdivide(int tx, int ty, int w, int h, Graphics2D g2)`

Recursively subdivides the given rectangle whose top left corner is in coordinates `(tx, ty)` and whose width and height are `(w, h)`. The base case of the recursion is when either `w` or `h` is less than `CUTOFF`, in which case this rectangle is drawn on the `Graphics2D` object provided. (The top-level call in the constructor should ask the image object its `Graphics2D` object and pass that on to the recursive call.) This rectangle should be white with probability `WHITE`, and choose a random colour from `COLORS` otherwise.

Otherwise, this recursive subdivision method should make two recursive calls for the two rectangles that you split randomly from the given rectangle. Make sure that you always split each rectangle along its longer edge, and that the splitting line is not too close to either edge that has the same direction, to keep the subdivision reasonably balanced. Subdivisions that contain thin shards and forced sharp angles tend to be perceived as ugly and disharmonious, the same way like when somebody writes cursive by hand and has to squeeze the words to be more narrow at the end of the line to fit within the margins. (In all walks of life, that elusive sweet spot of perfect harmony between the extremes of predictably solid and boring order and the screeching cacophony of randomness will be perceived as the most aesthetic by actual humans.)

To admire your randomly generated pieces of art, write a `main` method that creates a `JFrame` instance that contains one instance of `Mondrian` that is 1,000 pixels wide and 800 pixels tall. You can also try varying the above parameters and the colour scheme to aim for an even more artistic and aesthetically pleasing result.

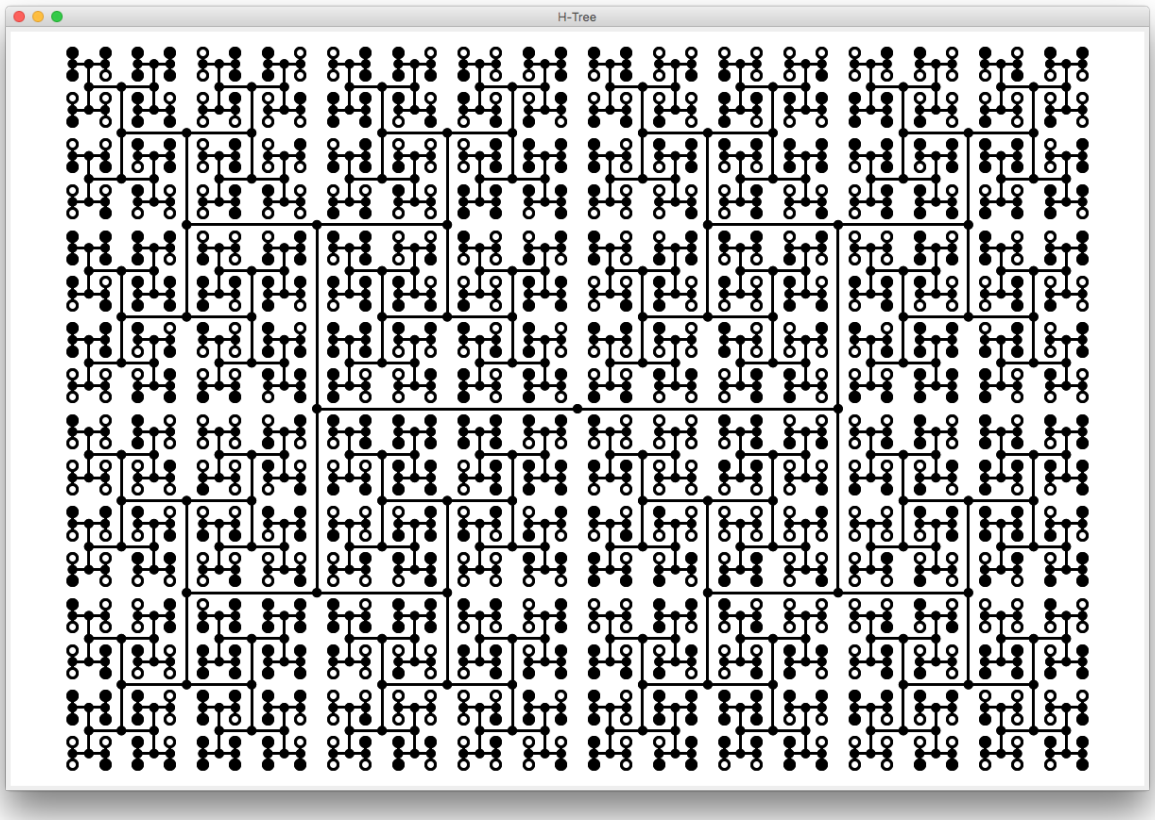
This random subdivision is not yet aesthetically the best possible due to a massive bias in the positioning of the subdivision walls inside the rectangle. Each subdivision will always contain exactly one straight **fault line** directly through that entire rectangle, which then continues to recursively hold for the two rectangles separated by this line. This is unconsciously perceived as being unnatural, even if the viewer does not consciously perceive the existence of these lines. (Having read that, the reader might enjoy finding these recursive fault lines from the previous example image, and deduce the order of the subdivision recursion from those lines.) A better

subdivision rule would break each rectangle into more than two smaller rectangles to eliminate the fault line through the rectangle, which makes the resulting subdivision structure seem more natural and generally pleasant.

After getting this lab to work, the student is invited to implement more organic subdivision rules as an alternative, and observe how some rules can make the other rules look like the proverbial dog's dinner. Similar random subdivision techniques are also seen in procedural level generation in certain roguelike adventure games.

Lab 14: H-Tree Fractal

In the spirit of the recursive Mondrian subdivision from the previous lab, here is another recursive fractal problem of rendering the H Tree fractal on the surface of a Swing component, with some additional decorative little dots that indicate the intersections and the caps at the end pieces of this exponentially branching structure. An example run of the instructor's private model solution produced the following result that your outcome should also resemble. (You may adjust the style to be more aesthetic, provided that you maintain the basic structure of this fractal.)



Start by creating the class `HTree` that extends `JPanel`. This class should have the following fields:

```
// Once line segment length is shorter than cutoff, stop subdividing.
```

```
private static final double CUTOFF = 10;
// Radius of the little dot drawn on each intersection.
private static final double R = 5;
// The image inside which the H-Tree fractal is rendered.
private Image htree;
// Four possible direction vectors that a line segment can have.
private static final int[][] DIRS = { {0, 1}, {1, 0}, {0, -1}, {-1, 0} };
// A random number generator for choosing the end piece colours.
private static final Random rng = new Random();
```

This class should then have the following methods.

```
public HTree(int w, int h)
```

The constructor receives the width and the height of the image as its arguments `w` and `h`, and should initialize `htree` to be a new `BufferedImage` instance of those dimensions. Acquire the `Graphics` object of this image and convert it to the more modern and powerful `Graphics2D` reference that allows you to first turn on **anti-aliasing**, and then later `draw` each line segment with non-integer precision inside the recursion. Before the recursive calls, make the image all white by filling it with a white rectangle whose area covers the entire image. Instances of `BufferedImage` start out being all black pixels since the bytes that store the colours of those pixels are guaranteed to be initially filled with zeroes. (This exact same guarantee is in effect for all objects in the Java heap memory.)

```
private static void render(double x, double y, int i, double len, Graphics2D
g2)
```

Render the H-Tree fractal starting from point `(x, y)` towards the direction given by the index `i` into the `DIRS` array that contains the four possible directions that each line segment can be heading, with the current line segment having a length equal to `len`.

The base case of the recursion is when `len < CUTOFF`, in which case this method should `fill` a disk of radius `R` centered at coordinates `(x, y)`, flipping a random coin to decide whether this disk should be black or white. (If it is white, you should still `draw` the black outline around it.) Otherwise, draw a black disk to the current coordinates `(x, y)`, and then calculate the endpoint `(nx, ny)` of the line segment that this fractal is currently heading. Draw the line segment from `(x, y)` to `(nx, ny)` on the image, and follow with two recursive calls at `(nx, ny)` using the directions `i + 1` and `i - 1`, making sure that these indices stay within the `DIRS` array. The new value of `len` in the recursive call should equal the current `len` divided by the square root of two, so that the branches become shorter down the line and now matter how deep you recourse, the fractal shape stays within the finite boundary.

Note how, due to the magic of subdivision fractals in general, these exponentially spreading branches never touch or overlap each other, no matter how deep you continue this recursion. In the

limit of `CUTOFF` equal to zero, as the recursion depth increases towards infinity, this **space-filling fractal** fills its entire two-dimensional bounding box perfectly not leaving any holes in which you could fit any circle with any arbitrarily small positive radius ε of your choice, despite consisting only of one-dimensional line segments and not even drawing the decorative disks on the intersections! (Students who happen to catch a case of "fractal fever" from this exercise, a once fashionable ailment presumed to be extinct since about the early nineties, might later want to dip into "Brainfilling Curves: A Fractal Bestiary", your instructor's personal favourite that teaches you how to render an infinite variety of such artistic fractals with **turtle graphics**.)

The constructor of `HTree` should call this `render` method twice, both times starting from the point that is in the middle of the image and with `len` set to equal to the minimum of the width and height of that image, divided by three. The first call should be heading left, and the second call should be heading right.

To admire your fractal artwork, you should again create a `main` method to open a new `JFrame` instance, inside which you add a new `HTree` instance of dimensions `(1000, 800)`. Again, the same way as with the recursive Mondrian subdivision, once you get this program to work, feel free to experiment to make the result look more aesthetic. This is perfectly fine with the instructor as long as you maintain the overall shape and spirit of the original H-Tree fractal. For example, if you happen to know how to compute arbitrary rotations to direction vectors using matrix algebra, you might want to add a touch of randomness to make the image seem more relaxed and natural so that instead of always turning exactly 90 degrees left or right, you instead turn 90 degrees plus or minus a small random amount, as if somebody were drawing this shape with a slightly trembling hand...

Lab 15: Zeckendorf Encoding

JUnit: ZeckendorfTest.java

Back in lab 0(E) we encountered the Zeckendorf theorem that says that every positive integer can be broken down into a sum of non-consecutive Fibonacci numbers exactly one way. This list of Fibonacci numbers can be encoded into bits so that the positions correspond to Fibonacci numbers so that 1 means that Fibonacci number is present in the sum, and 0 means its absence. For example, playing around with online conversion tools such as "The Fibonacci base system", we find out that 42 = 34 + 8. The Zeckendorf representation of this number is 10010000 to denote that the fifth (8) and eighth (34) Fibonacci numbers are present in the sum, the positions counted from right to left the same way as with ordinary positional representation.

The Zeckendorf representation is a bit (heh) inefficient way of representing integers compared to the ordinary binary number representation as a simple sum of powers of two. This is due to the wasted bit pattern 11 never appearing inside the Zeckendorf representation, for a roughly 25% loss of storage efficiency. However, this limitation suddenly turns into an advantage when we wish to encode not just one integer, but an entire sequence of arbitrary `BigInteger` values of unlimited

size! When a sequence of integers is encoded using the ordinary binary encoding for each individual element, this encoding must somehow be able to tell the reader code where each number ends and the next one begins. This can be solved by either agreeing that each integer is encoded with the exact same number of bits (which then enforces a maximum value that can appear inside the sequence), or by giving some high-end bits a role to indicate whether the encoding of the current number still continues, as is done in **UTF-8 character encoding** that we met back in the I/O streams lecture.

With Zeckendorf representation, telling where one number ends and the next one begins is automatic. The otherwise unused bit pattern 11 is used as an artificial **separator mark** that tells the decoder that the representation of the current number ends at that point, and the next number starts from the following position. The first 1 in the otherwise forbidden pattern 11 is the highest "zit" of the Zeckendorf encoding of that number, and the second 1 acts as a "comma" that separates that number from the next. Even better, this makes this encoding **self-synchronizing** and thus more **robust**; the reader can recover from any incorrect bits once it arrives at the next separator mark, from which the rest of the sequence will be faithfully restored. (UTF-8 also famously has the same property, as enforced by the high-order bits used as separator marks.)

Once we also realize that the convention of writing digits in descending order of positions is also but a societal agreement, and we all could have just as well agreed to write our bits, zits and other kinds of digits in ascending order of positions (for example, the number that we normally say out loud as "two hundred and eighty seven" would be written as 782 instead of 287), the Fibonacci coding of a sequence of integers consists of simply writing the Zeckendorf representations of each number left to right (so the last bit, being the highest bit, is always 1), and putting a 1 between any two numbers to create the impossible 11 to act as the "comma" to separate these numbers.

After that mouthful, it is finally time to start chewing. To celebrate the fact that the Zeckendorf encoding allows us to encode numbers that contain millions of digits, we shall use the `BigInteger` class in Java to represent our integers and the Fibonacci numbers used to break them apart, the same way as we did back in lab 0(E). Create a class `Zeckendorf` that contains the following two `static` methods:

```
public static String encode(List<BigInteger> items)
```

Given a sequence of `BigInteger` values as `items` so that each item is greater than zero, compute and return the Zeckendorf representation of that entire sequence as a single string whose each character is either `'0'` or `'1'`. For example, given the list `[4, 36, 127]`, this method would return the string "1011010000011010000101 1" (coloured for illustrative purposes), since 4 encodes to "1011", 36 encodes to "010000011", and 127 encodes to "10100001011".

If you have already completed the Lab 0(E), you might as well use the method `fibonacciSum` to do most of your heavy lifting here. Of course, in a real compression algorithm the result would be a sequence of bits instead of using one full Unicode character to encode each bit, but in this lab the

result is a string for demonstration purposes. (Inside this method, you will surely once again use the mutable `StringBuilder` to accumulate the zeros and ones, and convert the result into a `String` only at the end before returning it.)

```
public static List<BigInteger> decode(String zits)
```

Given the Zeckendorf decoding as `zits`, recreate the original sequence of positive integers. Your method may assume that `zits` is a complete encoding of an integer sequence that can contain characters `'0'` and `'1'` only.

Notice how, same as any two methods to encode and decode between two representations of something, these two methods are **inverses** of each other, so that applying one to the result of the other always produces the original starting value. Writing a **fuzz tester** is especially trivial for any such pair of functions. Even if test cases are produced in some pseudo-random fashion, the expected result of using the pair of methods in tandem is fully known, since it must always equal the original random value! Of course, to debug your code that is not passing this mass test, you can create yourself specific desired test cases with the above online Zeckendorf calculator.

Lab 16: Egyptian Fractions

JUnit: EgyptianFractionTest.java

As explained on the Wikipedia page "Egyptian fraction", ancient Egyptians had a curious way to represent positive integer fractions a/b by breaking them down into sums of distinct **unit fractions** of the form $1/n$. This breakdown can even be done in infinitely many different ways for any rational number, depending on what additional properties we wish to impose on this breakdown. However, the number of unit fractions needed to express a number as a sum of unit fractions tends to grow exponentially with respect to its absolute value. Even though the following construction algorithms would work for arbitrarily large rational numbers, we will only consider fractions $a/b < 1$ where $a < b$ to enable the JUnit test methods to terminate within a reasonable time and without running out of memory.

This lab has you implement two different algorithms to construct the Egyptian fraction breakdown. To ensure correct results even when they involve thousands of digits, you must do all your integer arithmetic using the `BigInteger` type from `java.math`. You probably should also add the `Fraction` example class into your labs project and use its operations to perform the arithmetic of fractions that you need here, instead of reinventing that rusty old wheel yourself, giving you more fun time to not merely walk but do arithmetic like an ancient Egyptian.

Create a new class called `EgyptianFractions` into your BlueJ labs project, and there a method

```
public static List<BigInteger> greedy(Fraction f)
```

that constructs the Egyptian fraction breakdown of the fraction $f = a/b$ with the **greedy algorithm** proven to terminate by Mr. Fibonacci himself! The simplest situation is when $a = 1$, so the result is $1/b$. Otherwise, compute the smallest positive integer n whose reciprocal still "fits inside" the fraction a / b . This can be done by dividing b by a using the **truncating** integer division, and then add one to the result of that division. Add the term $1/n$ to the result, and extend the result with the greedy breakdown of the remaining fraction $a/b - 1/n$ that is hopefully simpler. (Even if it is not, this algorithm has been proven to regardless terminate in finite time.)

To ensure unique results for the fuzz testing, this method should return the Egyptian fraction as a list of denominators of these terms in sorted ascending order each term of course represented as a `BigInteger` in our problem. For example, when called with the fraction 2/3, this method would return the list `[2, 6]`, and with the fraction 4/17, return `[5, 29, 1233, 3039345]`. As you can see, restricting the numerators to always equal one can cause quite a blowup in terms for even some seemingly simple fractions. As usual with greedy algorithms, the result is not necessarily the simplest or the most balanced possible subdivision into unit fractions, but the last term can be a rather thin sliver compared to the earlier terms.

The next method introduces another algorithm to construct an Egyptian fraction that usually produces a rather different result than the previous greedy method. Same as Fibonacci's greedy method, the following technique is not guaranteed to return the simplest or the most balanced breakdown. (It sure seems like some general pattern that underlies our visible fabric of reality is slowly emerging.) Write the method

```
public static List<BigInteger> splitting(Fraction a)
```

that maintains some `Set<BigInteger>` to keep track which terms have already been chosen to be part of the Egyptian fraction. Initially this set is empty. To break down a fraction a / b, we first realize that it trivially equals $(a - 1)/b + 1/b$. We can therefore "carve" unit fractions off the remaining fraction one at the time like carving slices from a Christmas ham. Slice off one unit fraction $1/b$ by adding b to your chosen set of terms, and construct the Egyptian fraction representation for the remaining fraction $(a - 1) / b$ until the entire number has been carved.

Note the difference between this method and the previous Fibonacci method in that here the denominators of the terms extracted are always some factors of b, whereas in the greedy Fibonacci method, n and b do not need to have any numerical relationship constraining them. If you compare the breakdowns for some randomly chosen simple fractions, the greedy algorithm usually produces fewer terms, but not always. For example, the fraction 5/91 breaks down into unit fractions `[19, 433, 249553, 93414800161, 1745264977814571645168l]` under the Fibonacci conversion, whereas the splitting method produces a far more balanced and pleasant subdivision `[23, 91, 2093]` into unit fractions.

To maintain the discipline inside the splitting method that each unit fraction can be used most once, we need to somehow handle the situation where the new slice $1 / b$ is already in our set of chosen terms. To do this in an orderly fashion, maintain a local variable `List<BigInteger> buffer` that contains the denominators of the unit fractions that are waiting to get into the set of chosen terms. Start by initializing this `buffer` to contain exactly one value b, the term that is currently waiting to enter the club. While this `buffer` is non-empty, pop out any one of the items there, let's call it n. (In the first round of popping, the value n must therefore be equal to b, but can and will equal other values during the later rounds of this corrective dance.)

If n is not already a member of the set of chosen terms, add it there. Otherwise you have a conflict of two equal terms $1/n$ trying to become a member of the set of chosen terms, even though only at most one of them can actually be part of the final set of chosen terms. Resolution of this conflict depends on the parity of n:

- If n is even, remove n from the set of chosen terms, and add $n/2$ in the `buffer`.
- If n is odd, add terms $n+1$ and $n(n+1)$ into the `buffer`.

For example, if the Egyptian fraction contains two equal unit fractions 1/10 and 1/10, these can be replaced by 1/5 without affecting the total. If the conflict is between two equal unit fractions 1/5 and 1/5 that unfortunately cannot be combined into 1/2.5, leave one 1/5 in the set of chosen terms as is, and replace the other term by pushing two new terms 1/6 and 1/30 into the `buffer`, the total again unaffected by this since 1/5 = 1/6 + 1/30. Rinse and repeat until all terms are distinct and the `buffer` is empty. Until then, the individual terms will enter, exit and re-enter the set of chosen terms and the waiting room of the `buffer`, the exact details of this back-and-forth rigmarole depending on the terms still waiting inside the `buffer` and the needs of the rest of the number $(a - 1) / b$ that is still waiting to be broken down.

We often like to think of integers being composed of prime factors multiplied together. For natural numbers, this **prime factorization** is unique, the result modestly known as the "Fundamental theorem of arithmetic" since this result is actually quite not as trivial as it might initially seem. But as we just saw, a whole another way to think of not just integers but all rational numbers is to see them as finite sums of tactically chosen subsets of unit fractions. However, unlike the unique prime factors that make up each integer, any rational number (and thus all integers) can be given an infinite number of different representations as subsets of unit fractions.

These Egyptian fractions boggle the mind as a way to express any one of the infinitely many positive rational numbers in an infinite number of ways of adding up smaller positive rationals. Kinda makes us wonder if those ancient Egyptians really were connected to something that we have forgotten... but up here in the present time, anybody interested to learn more about the topic can start on the page "Egyptian Fractions" by David Eppstein. Most of the external links from the page are long dead, of course, but "Algorithms for Egyptian Fractions" fortunately lives. (Note that on the page "Conflict resolution methods", Eppstein calls the algorithm "pairing" that we called "splitting", and the algorithm called "splitting" on that page is something else...)

Lab 17: Diamond Sequence

JUnit: DiamondSequenceTest.java

The `Set<E>` implementations provided in the Java Collection Framework are flexible, but not best for all situations due to their relatively high memory needs for each object stored in that set. Most of the time in practical programming, sets and maps are used to remember the things that we have seen and done during the method's execution. If those things can never become unseen or undone, the `remove` operation does not need to be supported, except possibly in the form of the `clear` method to evacuate the entire data structure at once. In the terminology of data structures and algorithms, collections that allow adding new elements but do not allow later removals are called **monotonic**, and can sometimes be implemented more efficiently than the full versions that allow `remove` that, since it decreases the entropy of the data structure, necessarily involves physical work making it inherently more complex than the methods `add` and `contains`.

Further optimizations are possible when members of the set are known to be **natural numbers**. This turns out to be quite a common case, given that everything inside a computer is made of integers anyway, as much as we like to otherwise pretend that the computer memory is full of strings, dictionaries and other entities that in reality are just as fictional as, say, hobbits and vampires. This becomes even more pronounced when these numbers tend to be added to the set in roughly ascend-ish order, which is often the case with many integer sequences whose rules to compute the next element depend on whether some particular integer has already been seen inside the sequence generated so far. For example, the Recamán's sequence that we met back in Lab 0(B). Membership information can then be packed as tight as sardines, so that each integer is represented as one bit stored inside a `boolean[]`. We will cleverly manage this boolean array as a "sliding window" so that the field `start` tells which position the zero index actually represents, and implement the methods `add` and `contains` to work relative to this fact.

Good news is that you don't need to do any of that, since for both this lab and your possible future endeavours in Java, your instructor provides a brand new class `NatSet` whose objects represent monotonic sets of natural numbers. These `NatSet` objects start their lives as representing the empty set. To make the set grow (but again, never shrink), this class has the public method `void add(int n)` that adds `n` into the set, and the public method `boolean contains(int n)` to query whether `n` is a member of this set. No `remove` method exists in its public interface.

Using the `NatSet` class as a tool to keep track of what numbers you have already seen during the construction, your mission is to implement another very interesting sequence of positive integers, found by your instructor from the book "Mathematical Diamonds" by Ross Honsberger. Since that book deals with mathematics instead of computer science, its conventional counting of sequence positions starts from one, not from zero the way it always does for us propeller beanie hat computer programmers. The first element in the position $k = 1$ equals one. Each position $k > 1$

contains the smallest positive integer that (a) so far has not appeared anywhere in the sequence and (b) makes the sum of the first *k* elements of the sequence to be some integer multiple of *k*.

For example, the second element for $k = 2$ is the smallest unseen integer that, when added to the first element 1, makes the total divisible by 2. The correct answer is 3, since $1 + 3 = 4$ is divisible by $k = 2$. After [1, 3], the third element is again the smallest unseen integer that, when added to the previous elements 1 and 3, makes the total divisible by 3. Correct answer this time is 2, since $1 + 3 + 2 = 6$, and two is the smallest of all unseen numbers that make it work. The sequence starts as 1, 3, 2, 6, 8, 4, 11, 5, 14, 16, 7, 19, 21, 9, 24, 10, 27, 29, 12, 32, 13, 35, 37, 15, 40,...

To practice writing virtual iterators that are not backed by any actual collections but who always produce the next element of the sequence in computational means, we implement this "diamond sequence" (named a bit unimaginatively due to the lack of further mathematical sophistication from this author's part), as a virtual iterator. Inside your BlueJ labs project, create the new class

```
public class DiamondSequence implements Iterator<Integer>
```

with the instance methods promised by the interface `Iterator`:

```
public boolean hasNext()
public boolean next()
```

Since this sequence is infinite, the method `hasNext` always returns `true`. For the computation performed inside the method `next`, this class should define an `int` data field `k` that keeps track of the position that the generation of this sequence is currently at. Then, there should also be another data field `sum` that contains the sum of the elements generated so far. The type of this field should be `long`, to ensure that adding up the elements of this sequence does not overflow even when we shall soon add up its first hundred million elements! Last, but definitely not the least, this object needs an instance of `NatSet` to keep track of elements generated so far.

The method `next` can should first increment `k` and perform some integer arithmetic to look for the smallest previously unseen `n` that, when added to the current `sum`, causes the total for the current position to be divisible by `k`.

The JUnit test class `DiamondSequenceTest` will push your code hard to its limits by making it generate the first one hundred million elements of the diamond sequence. Within this range, all values in sequence still fit inside `int` variables without possibility of overflow, but their sum certainly will not fit inside an `int`. The sequence does grow in a modest linear pace, but it bounces above and below this baseline in a seemingly chaotic fashion that somehow always manages to fill in the gaps that it has left behind. (Every filling of such a gap will then allow the `NatSet` data structure to move its sliding window into data that much more to the right...)

The fact that no integer appears twice inside follows trivially from the constructions, but it is not equally obvious that every positive integer will eventually appear in some position, as it will. Furthermore, this sequence has a most curious form of self-referential structure in that whenever the *k*:th element of this sequence equals *v*, its *v*:th element equals *k*. For example, since the seventh element of this sequence equals eleven, its eleventh element must therefore equal seven! Let's all try to wrap our heads around that fact for a moment. (In discrete math terms, when this sequence is viewed as an infinite **permutation** of natural numbers to themselves, all cycles of this permutation have length two!) The method `testSelfReferentiality` in the JUnit test class verifies this fact empirically for the first one million elements of this sequence.

Lab 18: Working Scale

This lab has you working with Swing again, but this time not just with inert graphics and boring GUI components such as buttons and text fields that people use at work, but animations so that the displayed content of your component is updated in real time fifty frames per second.

From the repository `ikokkari/CCPS209Labs`, add the interface `AnimationFrame` and the two classes `AnimationPanel` and `Tircle` into your labs project. The interface `AnimationFrame` represents the ability of an object to render itself into pixels on command, so that the contents of this image vary over time. This interface contains one method for your subclasses to implement:

```
public void render(Graphics2D g, int width, int height, double t);
```

Render the contents of the animation frame, as they are supposed to look like at time `t`, into the given `Graphics2D` object `g` so that the dimensions of this rendering are `width` pixels wide and `height` pixels tall. Time is not measured here in seconds of "wall clock time", but is only a number that can be positive, negative or zero. This way, the motion of the displayed animation could be sped up or slowed down arbitrarily with every individual frame still looking exactly like it should be. We could even find out with this one method what the animation frame would look like one year from now, without having to compute all the frames in between that far.

Regardless of the speed that this virtual time advances, the actual animation is rendered inside an `AnimationPanel` component exactly 50 frames per second to keep the visuals smooth enough to satisfy the human eye. The `AnimationPanel` objects are Swing GUI components customized to display the `AnimationFrame` object given to them. Time `t` always starts from 0 and increases by `step` after every animation frame. Whenever you create a new subclass of `AnimationFrame`, you can edit the `AnimationPanel[]` array in the `main` method to include an object of your new class to immediately try it out. As you admire your artwork, remember to also grab the window resizing handle and give it a good back and forth to ensure that your `AnimationFrame` properly renders as it should in any `width` and `height` and does not, for example, rely on the assumption that the animated image is a square.

(Also, from that whole deep philosophical and theoretical point of view why we use inheritance and polymorphism, do not fail to appreciate the isomorphism between the troika of concepts `Animal` / `Cat` / `getSound` versus `AnimationFrame` / `Tircle` / `render`.)

As an example of how to implement the `render` method, see the example class `Tircle` that renders a subdivision fractal that is built up from smaller copies of itself. Such **self-similarity** immediately suggests using recursive methods to compute and render such fractals. Since we cannot conveniently make the `render` method itself recursive (since it has no place for the additional parameters that our subdivision needs), we nimbly sidestep this limitation by extracting the recursion into a separate `private` method of its own (in the class `Tircle`, this recursive method is also called `tircle`) and make the. The arguments for this top level call from the `render` method depend on the parameters `width`, `height` and `t` to ensure that the rendering is correct for all image sizes.

To give this recursion a solid foundation with every branch reaching a **base case**, once the subdivision shape is small enough, it is rendered as some simple shape such as a circle or a rectangle in that position. In principle, this base case would be when the shape is smaller than one pixel. However, cutting off the subdivision once the radius of the shape is lower than some higher threshold such as ten pixels usually gives more aesthetic results. Armed with this general technique, you could render any subdivision fractal you can dream up. Note also that the laws of neither nature or man dictate that `t` has to remain fixed through all levels of recursion, but can be increased or decreased a little in each call to give the overall system a more organic appearance when its components are not acting in perfect lockstep with each other.

Your `render` method should render some kind of reasonable result for any legal value of the parameter `t`. The simplest way to achieve this is to **normalize** your animation time to always run from 0 to 1, regardless of the actual intended length of the animation. (The animation can always be stretched to arbitrarily length in the actual wall clock time by decreasing `step`.) For values of `t` outside the interval [0,1], the integer part of `t` is ignored and only the decimal part has an effect on the outcome. This makes the entire animation periodic so that at integer times t = 0, 1, 2, ... the animation has returned in its initial state. This animation can then be turned into an animated GIF that consists of `n` frames computed with `t` stepping through times 0, 1/n, 2/n, 3/n, ..., (n-1)/n. Note that the last frame is not `t = 1`, since this would be an exact duplicate of the first frame and show up as a small but annoying twitch in the animation. Since animated GIFs tend to use more space than is necessary (the format does not exploit any of the myriad statistical correlations between consecutive frames in the compression, but encodes each frame from scratch as a separate image), of course these days we convert any such animated GIF into an MP4 movie to squeeze out about half of its bytes. You kids these days have it so easy...

To make your periodic animation smooth so that there is no visible discontinuity at the integer times, define your animation paths so that the position, size and other aspects of its **appearance** are the same for every individual object for `t = 0` and `t = 1`, whichever way that object happens to move in times between these two endpoints. Alternatively, you can organize your objects so that

each object has a "successor" so that the appearance of this object at `t = 1` is the same as the appearance of its successor object at time `t = 0`, and this way "cheat" to create infinite forward motion. See, for example, "Frequipathy", "Cubenary" or "Stalka", some old creations of the author. The fact that these compute the objects in three dimensions makes absolutely no difference to anything relevant. (Objects that end up outside the rendering area don't need such a successor, since their "blip out of existence" never shows up on the screen...)

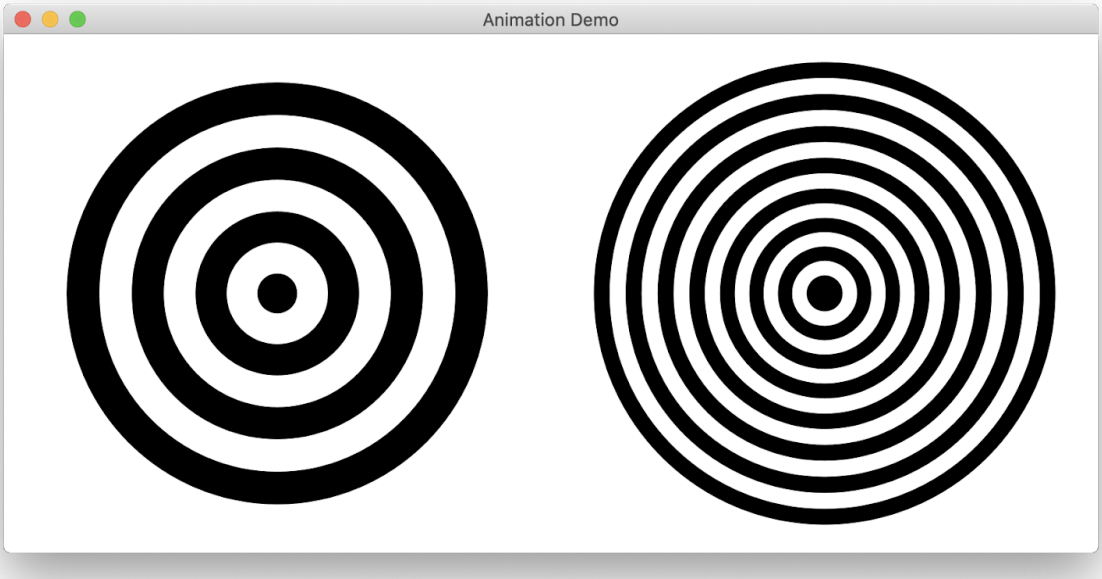
Calculations that consist of such periodic motions will greatly benefit from the existence of periodic functions, which most often are the basic trigonometric functions of sine and cosine. Since the purpose of this lab is not give you unpleasant flashbacks to trig back in high school, the math is conveniently provided to you in form of two periodic functions

```
default public double circleX(double cx, double r, double t)
default public double circleY(double cy, double r, double t)
```

already implemented in the interface `AnimationFrame` so that any class that implements `AnimationFrame` automatically inherits these methods. You can use these functions to implement smooth periodic motion in either one or two dimensions. To have some quantity `x` oscillate between two extreme values `cx+r` and `cx-r` around center point `cx`, simply compute it with the formula `x = circleX(t)`. For example, to have `x` oscillate between the values 4 and 12, the math says that `cx = 8` and `r = 4`. This oscillation is normalized to have a period

When used together to compute the *x*- and *y*-coordinates for the position of some object at time `t`, the two methods `circleX` and `circleY` give you the position of an object that does **circular motion** around the center point `(cx, cy)` with the radius `r`. (Internally, `circleX` uses the cosine function whereas `circleY` uses sine.) This approach can also be used to compute the points of a regular polygon centered at `(cx, cy)` with `n` sides (for example, a square for `n = 4`), with the *i*:th corner being in coordinates given by plugging in `t = (1.0 * i) / n`. (To rotate this polygon, add the same offset to `t` for each corner.)

Enough math and theory, let's finally get on with the practice! Create a class `Breathe` that implements `AnimationFrame`. Implement the `render` method in this class to render concentric circles alternating in black and white (the number of circles should be given as constructor parameter) from outside in so that each circle is centered at the same coordinates `(width/2, height/2)` with radius diminishing inwards. An example screenshot of the `main` method of `AnimationPanel` that creates one instance of `Breathe` with eight circles and another with sixteen, hopefully clarifies the intention. However, the radius of each circle should slightly oscillate in some smooth fashion so that **the entire animation gives an appearance of something organic that is breathing, has a pulse, or performs some muscular effort to move**. Adjust your equations for the radii until you are satisfied with the impression.



Once you get the shapes in their proper places on the componen, you can check out the author's old works "Frequestion", "Jobol" and "Zigwave" as examples of organic motion. The neat thing about periodic functions is that you can always add or subtract another periodic function, and the result remains periodic. Even better, as long as the period of the first function is an integer multiple of the period of the second function, the result will have the same period as the first function. This allows us to design our motion paths with linear thinking, and once the animation works, adjust it by adding suitably periodic small "wobblies" to them, possibly weighted with periodic functions that are also tactically chosen to be zero at both $t = 0$ and $t = 1$...

Lab 19: Symbolic Distances I: Representation

JUnit: DistanceTestOne.java

Even though Java expressions are in symbolic form inside the language, during the execution all these variables contain only the numerical results produced by evaluating some of those expressions. Numerical results usually suffice for our needs (you only need to "show your work" in school exams, whereas the rest of reality needs only the final result), but sometimes mere end results can be unsatisfactory. Especially when they get rounded to the `double` type that cannot express even simple numbers correctly, as we saw by trying to evaluate 0.1 + 0.2.

For this reason, we like to define new data types that can represent some restricted forms of symbolic expressions exactly. We already saw the `Fraction` type that allows us to keep rational numbers such as 1/3 in their exact form, instead of having to round this to 0.3333333333333333 so that this close but still fundamentally incorrect result can be assigned to some variable so that

the rest of the computation can proceed. Some languages define useful data types for fractions and other limited symbolic expressions. Some really high level languages such as *Wolfram Mathematica* keep every expression in its symbolic form and operate on those forms, so that only the final result needs to be converted to its approximate numerical approximation for human consumption.

If you can tolerate the memory and time cost in keeping expressions in their symbolic forms, such expressions are better than numerical approximations for the simple reason that you can always convert from symbolic to numeric, but once this approximation is done, there is no way to reverse it and get the original symbolic expression. Furthermore, symbolic expressions convey useful information about the structure of the data in a way that gets lost in conversion to numeric. (Of course, inside the computer there are only bytes filled with zeros and ones, and anything higher than that is only agreed fiction to help us organize our thoughts about computations in a more succinct form than the bare metal would allow.)

In spirit of the `Fraction` example we saw in the lectures, this lab and the two labs that follow it define a new data type to symbolically represent numbers of the form $a\sqrt{b}$ where a can be any integer, and b can be any **squarefree** positive integer. A positive integer n is said to be squarefree if n is not divisible by c^2 for any integer $c < n$. Some examples of such numbers that we wish to represent are $-7\sqrt{13}$, $42\sqrt{10}$, and $-1234\sqrt{101}$, where the requirement of being squarefree gives every such possible value one unique representation in our system. For example, $7\sqrt{48}$ is equal to $7\sqrt{3 * 4 * 4}$, which allows those two fours to be extracted from under the square root. This gives us a more elegant representation $28\sqrt{3}$ for that very same quantity.

(A relevant fact about integers is that the square root of an integer can never be a rational number of the form a / b unless $b = 1$, making the square root itself also an integer and the number under the root symbol a **square number**. Otherwise, that square root must be an irrational number.)

This representation generalizes integers, since every integer a can be trivially represented as $a\sqrt{1}$ inside this system. Furthermore, formulas that compute $a\sqrt{b} + c\sqrt{d}$, $a\sqrt{b} - c\sqrt{d}$ and $a\sqrt{b} * c\sqrt{d}$ will only produce results of that same form. The set of all such numbers is therefore **closed** with respect to these operations: no matter how you apply these operations to the values you have created so far, you will never get a value not representable in this system, excluding the considerations of arithmetic overflow. (To be able to allow division as the fourth arithmetic operator and still keep this system closed, we would have to extend the integers to rational numbers, perhaps with some sort of more sophisticated `Fraction` type that could handle symbolic expressions as numerators and denominators instead of mere integers...)

But that's again enough math, the time has come for some decisive action. Since these kinds of numbers often appear in distance calculations between points in the two-dimensional integer grid, create a class named `Distance` in your BlueJ labs window. Each instance of `Distance` represents a number that consists of a sum of terms of the form $a\sqrt{b}$, so that no b appears twice in the sum. (The number $a\sqrt{b} + c\sqrt{b}$ can always be written as $(a + c)\sqrt{b}$ to enforce this rule.)

Since the operation of extracting squares from the given integer is so important, our first order of business is to implement the method to do this. Since this method does not need anything from the underlying object, we might as well make it a `static` so that it can be used as a function:

```
public static int extractSquares(int n)
```

Given a non-negative integer *n*, this method should find and return the largest positive integer *c* so that *n* is divisible by c^2. (For any *n*, the value *c* = 1 will always work, if nothing higher does.)

The next thing we need to do is to choose the internal representation for this new data type. To keep everybody who works on this lab on the same page with respect to this design issue, please use an instance of `TreeMap<Integer,Integer>` to associate each integer that appears under the square root with its coefficient outside the square root. Declares the following field inside your `Distance` class:

```
private TreeMap<Integer, Integer> coeff = new TreeMap<Integer, Integer>();
```

Every term of the form $a\sqrt{b}$ from the pure Platonic plane of mathematics thus becomes an entry `b=a` inside `coeff` map in Java. For example, inside a `Distance` object that represents the number $3\sqrt{1} - 4\sqrt{7} + 5\sqrt{11} + 2\sqrt{101}$, contents of `coeff` would be `{1=3, 7=4, 11=5, 101:2}`, with exactly as many entries inside the map as there are terms in the original formula. (Note how Java even uses the curly brace syntax in the `toString` representation of a `Map` the same way as Python does for its equivalent `dict` data structure.)

So that we will be able to construct simple `Distance` objects, we need a constructor to initialize an object that consists of a single term, and a constructor to initialize an object from the arbitrary map that encodes the terms:

```
public Distance(int a, int b)
public Distance(Map<Integer, Integer> coeff)
```

These constructors should also call your `extractSquares` function to express each term in its simplest form, since the outside callers cannot be expected to have performed this simplification on your behalf. It will also make your life a lot easier later if you write these constructors to make sure that `coeff` will only contain terms $a\sqrt{b}$ for which *a* is nonzero. Any such term should simply be ignored during the object construction. So you might as well do that right now so that the rest of your methods can assume that every term has a nonzero coefficient.

Just like we did with `Fraction`, we are going to design the class `Distance` to be **immutable** so that it has no `public` methods that could modify the internal state of the object. The methods for arithmetic operators will always create and return a new object to represent the result, instead of

modifying either one of the originals that participate in that operation. Therefore, your second constructor should create a defensive copy of the `coeff` map object given to it and copy the coefficients there for safekeeping, instead of just storing a reference to the object that is still being used and therefore modified by the outside world.

One more method left to write in this lab, although that one is a biggie. The very first `public` method that you should implement inside any new data type, once you have chosen its representation, is surely `toString()`. Most of the time this is straightforward, but we might as well do all the work now to make the result look pretty, so that no user code ever has to do the work that clearly belongs to the designer of that data type. As they reputedly say in the military of some country, a good soldier will throw himself on the landmine to save his battle buddies. (Your instructor would not really know, since the only veteran status that he could claim is being a veteran reader of the "Humor in Uniform" column in *Reader's Digest*.)

```
@Override public String toString()
```

Because the automated JUnit test classes test the results by looking at this string representation, the string returned for each instance of `Distance` has to be unambiguous. Furthermore, making the string representation look pretty gives us an interesting computational exercise, which is why we are doing it this way here. The rules to guarantee that the outcome is both unambiguous and readable are as follows.

1. The terms must be listed in ascending order of their square roots. (This is why we use `TreeMap` instead of `HashMap`, so that looping through the elements of its `keySet()` view is guaranteed to produce them in ascending order.)
2. If the coefficient of some term is 1, it is left out, unless the square root is also 1.
3. Each individual term $a\sqrt{b}$ is transcribed as the string `"aSqrt[b]"`, with no space between the coefficient and the square root.
4. The consecutive terms are separated with either `" + "` or `" - "` depending on whether the coefficient of the next term is positive or negative, with exactly one whitespace around the operator symbol. If the `" - "` separator is used, the coefficient of the next term is transcribed without the minus sign.
5. If the coefficient of the first term is -1, it is transcribed as a minus sign, without the 1.

These rules produce the result that *Wolfram Mathematica* would use to display such terms. Given a choice between implementing the `toString` method some other way rather than the way that has been field tested over thirty hard years, is there any reason not to go with the tested and true? The following table shows some illustrative examples of the expected way to transcribe a `Distance` object, given its `coeff` map. Of course, your constructor of `Distance` will make sure to extract all possible squares out of these coefficients, rather than passing the buck to every future user of this class. (To add insult to injury, proclaim your class to be "simple" and "lightweight".)

coeff	Expected result of calling `toString()`
{}	"0"
{1:1}	"1"
{4:7}	"7Sqrt[4]"
{7:4}	"4Sqrt[7]"
{5:1,10:-1,15:1,21:-3}	"Sqrt[5] - Sqrt[10] + Sqrt[15] - 3Sqrt[21]"
{5:1,10:-1,15:1,20:-1}	"-2Sqrt[5] - Sqrt[10] + Sqrt[15]"
{3=-5,7=11,6=-17}	"-5Sqrt[3] + 11Sqrt[7] - 17Sqrt[6]"
{10=-1,43=99}	"-Sqrt[10] + 99Sqrt[43]"

Lab 20: Symbolic Distances II: Arithmetic

JUnit: DistanceTestTwo.java

The code written in the previous lab allows us to construct and print out symbolic expressions that are sums and differences of terms of the form $a\sqrt{b}$ so that a is any integer and b is any positive squarefree integer. The time has come to start doing computations on such numbers with addition, subtraction and multiplication. Since we intentionally design the class `Distance` to be immutable to get all the advantages of immutable data types, none of these methods should modify the internal state of either `this` or `other` object, but create and return a new `Distance` object for the result.

In writing this week's methods, you are allowed to assume that the JUnit test class gives you only argument values that never cause the computation to overflow when using the `int` type for all integer arithmetic. If you want to convince yourself that such normally unreported overflows do not happen, you can execute your `int` arithmetic with the methods `addExact` and all its brothers in the class `java.lang.Math`, instead of using the operators `+` and such already provided in the core language. These exact arithmetic methods in `Math` class are guaranteed to throw an `ArithmeticException` whenever they detect an overflow. The absence of the said exception makes you certain of the fact that no overflow has so far occurred during execution.

```
public Distance add(Distance other)
public Distance subtract(Distance other)
```

These methods implement the addition and subtraction between this and `other`. Since the logic of both methods is identical except for one small part of one statement, we strongly recommend implementing both of these operations in one swoop with a `private` helper method

```
private Distance arithmetic(Distance other, boolean adding)
```

where the parameter `adding` indicates whether the terms coming from `other` should be added or subtracted from the corresponding terms of `this`. The actual `public` methods `add` and `subtract` will then both be simple one-liners that call this method with the second argument set to `true` or `false`, respectively.

```
public Distance multiply(Distance other)
```

This method might be best implemented by first creating an empty `TreeMap<Integer, Integer>` used to build the result. Using that, the logic of the method is just two nested loops that iterate over all possible pairs that take the first term from `this` and the second term from the `other`. For every such pair of terms $a\sqrt{b}$ from `this` and $c\sqrt{d}$ from `other`, compute their product $ac\sqrt{bd}$, possibly extracting some square hiding under the square root made of the common prime factors of a and b. Then, either create a new entry for bd inside your result map, or update its existing entry for bd by adding the coefficient ac to its value.

Lab 21: Symbolic Distances III: Comparisons

JUnit: DistanceTestThree.java

The previous two labs granted us the ability to perform arithmetic on `Distance` objects and print out the results in a human-readable format. This week, it is time to make these `Distance` objects behave properly so that the existing classes in Java Collection Framework and elsewhere work properly with our new data type. In Python, this is done simply by defining the magic methods that correspond to the operators of the language. In Java, the following methods inherited from the universal superclass `Object` should be properly overridden for this purpose.

```
@Override public boolean equals(Object other)
```

Using the example class `Fraction` as a model, override this method to accept value equality only with another `Distance` object whose `coeff` map consists of the exact same terms as `this` one. The analogy to the famous Fundamental Theorem of Arithmetic applies in this situation in that every number that is representable in this system has exactly one such representation. If two `Distance` objects have different `coeff` maps, the numbers that these objects represent must surely be different.

```
@Override public int hashCode()
```

Override this method to somehow compute a hash code that depends on all of the terms in the `coeffs` map. In a sense there is no wrong answer in this one, but it is always better to make your hash functions spread the objects all along the possible range of all `int` values in a way that is as

"scrambly" as possible. **Bitwise operations**, especially the **bit shifts** and **exclusive or**, are often used in this kind of scrambling. See the `hashCode` method in the `Fraction` example class. Using the bit shifts differently for different components of the object guarantees that objects whose internal states are permutations of each other will still be hashed into different positions.

Since these `Distance` objects represent real numbers, they should also allow order comparison as such. Edit the first line of the class definition in your source code to now say

```
public final class Distance implements Comparable<Distance>
```

to guarantee the existence of `compareTo` method. The big question now is what kind of algorithm could compare two arbitrary `Distance` objects in such a manner. In absence of any fancier mathematics, we will simply evaluate each object as a decimal number and compare those for the answer. However, to make this lab more interesting, we will not be using the plain old `double` type for this, but might as well learn how to perform such calculations to arbitrary precision with the `BigDecimal` type from the `java.math` package! (At least when we are dealing with the `BigDecimal` type, 0.1 + 0.2 equals 0.3, instead of even one iota more or less...)

Analogous to the `BigInteger` type for integers limited only by your heap memory, `BigDecimal` objects represent decimal numbers. Their integer part is unlimited, same as in `BigInteger`, but their decimal precision is determined by some `MathContext` object that tells the `BigInteger` how far down it should keep computing these decimals. After all, even simple arithmetic expressions such as 1.0/3.0 would need an infinite series of decimals to be accurate! Whenever a `BigDecimal` operation produces an approximate result, the **rounding mode** parameter determines how the value of the last included digit is computed from the truncated digits.

```
public BigDecimal approximate(MathContext mc)
```

Return the approximate value of this `Distance` as a `BigDecimal` object with the settings as in the given `MathContext`. You should simply evaluate each term in turn and add it to the result.

```
public int compareTo(Distance other)
```

Performs the order comparison between `this` and `other` objects by comparing their `BigDecimal` approximations. Same as in equality comparison, two `Distance` objects that have any different terms must surely represent different numbers. To settle the issue of which one is greater than the other, how can we know how much precision the `MathContext` needs to have for this comparison to be accurate. Otherwise, two sums whose values differ by some microscopic amount would seem to be equal to us when viewed as their numerical approximations.

The lovely thing about this problem is that we don't need to know how much precision we need! Since we can always repeat the approximation using a higher precision, simply start with some small precision such as 10, and `approximate` both `this` and `other` with a `MathContext` of

precision. As long as these approximations do not reveal which number is greater, do it again using a higher precision such as 15. Eventually your precision will get surgical enough to tell apart these numbers even if they are separated by a razor-thin margin somewhere in the hundredth or such decimal place...

Students interested in seeing more `BigDecimal` computations can check out the `Mandelbrot` example. This fractal renderer performs its computations with the `BigComplex` example type defined to perform addition and multiplication of complex numbers of arbitrary precision. Akin to that old joke of how easy it is to sculpt a statue of an elephant ("Start with any block of stone, and then chip away every piece that doesn't look like it is a part of an elephant!"), the program sculpts the outline of the black Mandelbrot set by carving away all the pixels around it that do not belong to that black hole. Such pixels are coloured using some scheme that depends on how many steps were required for the iteration from that pixel to cross the boundary. Not even one processor cycle is wasted on pixels inside the Mandelbrot set, although the downside is that the computation never terminates as the border pixels are iterated forever in a vain hope of escape.

To zoom into the fractal whose rendering process never terminates as the outline gets chipped into a more accurate shape one pixel at the time, first press "Stop" to stop carving the current fractal, and then use your mouse to drag a new viewing window inside the display. Letting go of the drag initiates the new render zoomed into your drag window.

Lab 22: Truchet Tiles

For testing: TruchetMain.java

As explained in the Wikipedia page "Truchet Tiles", tiling the two-dimensional grid using four different but rotationally symmetric black-and-white tiles can produce surprisingly complex "op-art" geometric patterns. In their classic form, the four Truchet tiles are identified in this lab by two truth values `parity` and `diag`. The parameter `diag` determines whether the dividing line between black and white parts cuts through the main diagonal or the antidiagonal, and the parameter `parity` determines which side is black and which side is white. In your BlueJ labs project, create a new class with the signature

```
public class Truchet extends JPanel
```

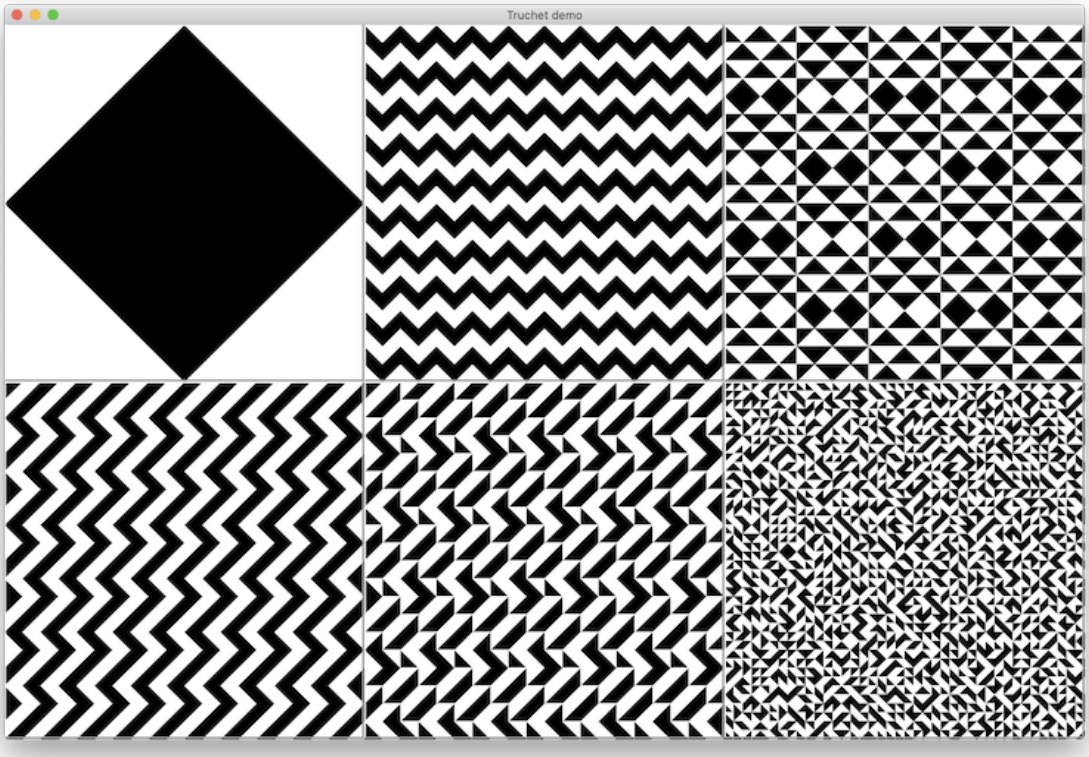
to make your class to be a full-fledged Swing component out of the box. The instances of this class should then display Truchet tilings defined by the five constructor parameters.

```
public Truchet(int s, int w, int h, BiPredicate<Integer, Integer> parity,
BiPredicate<Integer, Integer> diag)
```

Parameters `w` and `h` determine the width and height of the grid in tiles, and the parameter `s` determines the size of each tile in pixels. For example, an image made of values `w = 100`, `h = 60` and `s = 5` would be 500 pixels wide and 300 pixels tall. Parameters `parity` and `diag`, both of the same type `java.util.function.BiPredicate<Integer>` that encapsulates a function object that takes two `Integer` parameters and returns a `boolean` truth value, provide the renderer the parity and diagonal of each tile based on its tile coordinates `(x, y)`. This constructor should set the preferred size of this component to `new Dimension(s*w, s*h)`.

```
public void paintComponent(Graphics g)
```

Renders the individual Truchet tiles on the surface of this component, the shape of each tile as determined by the strategy objects `parity` and `diag` to make these decisions. There are four possible ways to determine the colours based on the parity and diagonal direction of the tile. However, to allow the student and the grader to agree that this lab has been implemented correctly, your code should make these decisions in the exact manner that makes running the automated test class `TruchetMain` produce the result shown in the screenshot below.



(Unlike with JUnit, human eyeballs will have to make the final determination whether the pattern is as expected. But at least our visual system offers a massive level of parallel processing that JUnit or other computer program could not even dream of...)

Especially important is the `Truchet` instance at the top left corner of this 2-by-3 component grid, the simple black lozenge against the white background (or is it four white corner triangles against the black background?), since that one confirms that your code uses the same scheme for colouring the tiles as in the instructor's private model solution. You might want to check out the modular arithmetic formulas used to produce the other instances where a pattern emerges from the local decisions that depend on the numerical properties of the tile coordinates. Sometimes even randomness can be made to look pleasant by adding enough nonrandom things to balance it out so that the human visual system has something to latch on and interpret.

Once you get this component to work, you can try out more artistic variations such as **Smith tiles** that use quarter circles instead of triangles to create a curved shape. Or try to create a labyrinth by using `diag` to draw a black diagonal line to separate the two white halves, and ignoring `parity` completely. You could even subdivide some tiles into smaller tiles for a fractal subdivision effect. Only our imagination shall be the ultimate limit of our art, and the only law that we accept to bind our hands in this is the Law of Awesome!

Lab 23: Ulam-Warburton Crystals

For testing: UlamTest.java

Game of Life is the most famous creation of John Conway who passed away during the COVID pandemic (xkcd tribute) one week before this lab was written. Game of Life is used in many introductory programming courses either as an example or a programming project for a good reason, since this deceptively simple setup contains ample educational value not just for the first coding course but far beyond that concerning the ultimate limits of computability.

However, just to be different from this beaten path, we shall instead implement a somewhat similar Ulam-Warburton automaton that produces its own distinctive fractal patterns of more crystalline nature. Besides, Stanislaw Ulam was also a pretty important figure in applied mathematics and computing in his time including the Manhattan Project, so his life and creations deserve at least some attention up here in the science fiction future timeline that we have found ourselves living in. Somehow most people just don't seem to realize this, almost as if some wizard had cast a spell to make humanity forget that this has happened. The term "science fiction" itself has become obsolete when even video phones, let alone face substitution and real time subtitles and translation to another language elicit mostly a bored yawn. (Any readers who have a better explanation for this mass amnesia than the meek "a wizard did it" then surely also know enough not to publicize this explanation to the wider world.)

The first new programming technique used in this lab shows how to explicitly encode two-dimensional arrays in just one dimension. Two-dimensional arrays in Java are actually just one-dimensional arrays whose elements are one-dimensional arrays. This "array of arrays" representations allows the individual rows to be **ragged** so that they don't necessarily all have the

same length. However, when dealing with regular grids where each row is known to have the same length, the two-dimensional structure can also be encoded into a one-dimensional array with a total of `rows*cols` elements, where each element in the two-dimensional position `(i,j)` is stored in position `i*cols+j`. (This is how numpy stores and represents its arrays that are known to be grids of uniform elements.)

The latter representation is especially useful if the values in that two-dimensional array are symmetric across the diagonal, as in problems where the underlying reality that creates the entries in this table is already symmetric with respect to these indices. To map both `(i, j)` and `(j, i)` into the same target position, first enforce `i <= j` with a swap if necessary, and then use the formula `j*(j+1)/2 + i` to find the target position in the one-dimensional array. Compared to the redundancy of storing every non-diagonal element twice, this arithmetic trick saves almost half of the memory used to store these elements. Cutting your memory use in half is nothing to sneeze at, since it allows you to solve instances twice the size and that way have twice the fun!

The Ulam-Warburton automaton is similar to Game of Life in style and spirit so that every **cell** in the board (now a simple two-dimensional grid, but could in principle be any **graph**) is always in exactly one of the two possible binary states "on" and "off". The automaton advances in discrete steps starting from zero, updating each cell logically simultaneously at each tick of time. In the original Game of Life the state of each cell at time $t + 1$ depends on the states of the up to eight cells in its 3-by-3 Moore neighbourhood (fewer for the cells on edges and corners) only at the previous time t, the rules of Ulam-Warburton automaton have the state at $t + 1$ depend only on its von Neumann neighbourhood (up, right, down, left) from the previous two steps t and $t - 1$, instead of just the immediate predecessor state. The rules are:

- If the cell (i, j) is **off** at time t, it is on at time $t + 1$ if **exactly one** of its four neighbours was on at time t, and otherwise it is off.
- If the cell (i, j) is **on** at time t, it is on at time $t + 1$ if that same cell (i, j) was off at time $t - 1$, and otherwise it is off.

In other words, cells turn on at the time when exactly one of their neighbours is on, and automatically turn off after having been on for exactly two time steps, regardless of the states of the neighbouring cells. Your method must, of course, be careful at the edges of the board.

In your BlueJ labs project, create a class `Ulam` and write the following method inside it:

```
public static void computeNext(int cols, boolean[] prev, boolean[] curr, boolean[] next)
```

The arrays `prev` and `curr` contain the two-dimensional boards at times $t - 1$ and t encoded into a single dimension, as explained above. (The number of `rows` is equal to `prev.length/cols`, and would thus be redundant to have as a parameter.) This method should not return anything, but compute the state of the board at $t + 1$ into the given parameter array `next`. Every time this method

is called, all three arguments `prev`, `curr` and `next` are **guaranteed to be proper array objects that contain enough elements**.

To allow the states of this automaton to be displayed as a Swing component, have your `Ulam` class extend `JPanel`. Define the field

```
private boolean[][] states;
```

that the component uses to store the computed states of the automaton, so that state at time *t* is stored in the one-dimensional boolean array `states[t]`. Then, define a constructor

```
public Ulam(int rows, int cols, int[][] start, int n)
```

that computes the `states` of this automaton for times 0 to `n` for later lookup. At time *t* = 0, every state is off. At time *t* = 1, every state is off except those whose positions are given in the array `start` that is guaranteed to contain two-element `int[]` objects as its elements. For example, in the `UlamTest` method `massTest`, the array

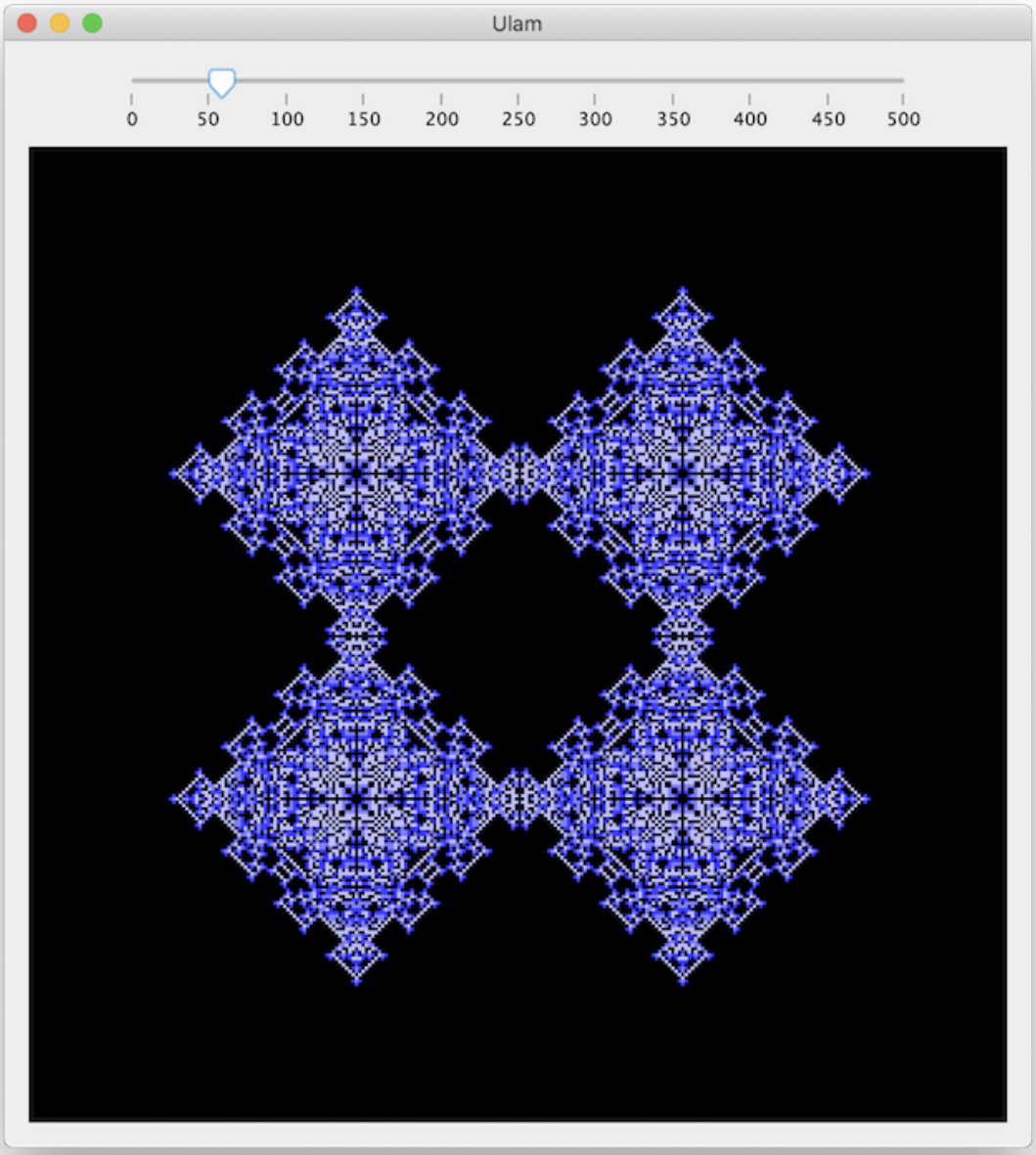
```
int[][] start = { {100, 100}, {100, 200}, {200, 100}, {200, 200} };
```

would have those four states on. To allow the user code to access the state of each cell at given times, your class should have the accessor method

```
public boolean getCellState(int i, int j, int t)
```

that returns the state of the cell (*i*, *j*) at time *t*. To verify that the logic of your `computeNext` method is correct according to the rules of this automaton, you should run the `massTest` method in the JUnit test class `UlamTest` to verify that it gives you a green checkmark.

Just so that we can try something new in this lab, instead of animating the evolution of the automaton states using some form of Java concurrency, the component should contain a `JSlider` instance whose values range from 0 to `n`. By grabbing and moving that slider, the user should be able to instantly skip to any particular time to see the board displayed in the `paintComponent` method in a manner that resembles the following low-resolution screenshot.



This display of a 300-by-300 board renders each cell as a 2-by-2 pixel box to make the result easier to see, but you can adjust this if you want to explore the evolution of the crystalline structure in larger game boards. You can also try seeding the automaton in a non-symmetric fashion to break the symmetry in the evolution of the automaton.

The colour scheme used in the above screenshot uses the colours listed in the following to render a cell based on its state at times t and t - 1, but you can also try different colour schemes for a more exciting result.

State at *t*	State at *t* - 1	Colour
off	off	black
on	off	dark blue
on	on	medium blue
off	on	light blue

To get back to Conway's Game of Life for a little bit, every second year computer science major should be able to write a simple brute force Game of Life component. However, try to imagine the algorithmic techniques needed to merely store the current state of a giant board whose side lengths measure in millions of tiles (data structures for **sparse matrices** suddenly become necessary to keep this Jacob's ladder from running out of memory), let alone simulate the effect of one time step on such a board, or coming up with mathematical techniques to simulate some part of the board more than one step forward in one swoop.

Another surprisingly difficult coding challenge is to implement the logic of Game of Life backwards so that the **previous** state is computed from the current state that it leads to. However, even if a computation is deterministic to the forward direction, it can still be nondeterministic to the backward direction so that the same current state could potentially have more than one possible predecessor, some of which leading to contradictory states that cannot be produced by any configuration in Game of Life. In general, if computations could somehow be reversed in some mechanistic fashion without this exponential blowup of possibilities of the past, any encryption algorithm would be trivial to crack simply by running the encryption algorithm in reverse from ciphertext output back to the original plaintext input...

Lab 24: Chips On Fire

JUnit: CoinMoveTest.java

Consider the following operation done for arrays whose elements are nonnegative integers, considered here to be the number of **chips**. Apologies for the inconsistent terminology used here. The first version of this problem was talking about "coins" until the author found out that in the literature of this field, the systems described below are known as chip firing systems. (As long as names are sufficiently memorable and unique to keep us from confusing their referents, it's all to-*may*-to, to-*mah*-to anyway, at least as far as this author is concerned.)

Each position in the array also has a list of **obligations**, each obligation being another position in the array. Each position can have any number of obligations to other positions, and can have multiple obligations to the same position. Time proceeds in discrete steps so that at time *t*, the

following operation is done in unison for every position of the array, to produce the chip distribution for the next time step $t + 1$.

- Every position that has at least as many chips as the total length of its list of obligations, gives out exactly one chip to each position as many times as it appears in that list.
- The remaining positions that did not have enough chips to fulfill their obligations do not give out any chips to anybody.

This operation does not change the total number of chips in the system, just moves these chips around the positions. The **states** of this chip-firing system are therefore the possible ways that the chips can be distributed among the positions, and the **transitions** lead from each state to the next state that results from the previous operation. To compute these transitions, create a class `CoinMove` and in that class, the method

```
public static void coinStep(int[] curr, int[] next, List<List<Integer>> obligations)
```

The current state is given in the parameter `curr`, and the list of obligations for each position is an element of the list of such lists named `obligations`. (Both `curr` and `obligations` are guaranteed to have the same length.) This method should not return any result, but instead write the next state into the array `next`, also guaranteed to be an `int[]` of sufficient length at the time of the call. The following table showcases some possible values for `curr` and `obligations`, and the expected `next` state that this method is expected to produce:

`curr`	`obligations`	Expected `next`
`[4, 5]`	`[[1], [0]]`	`[4, 5]`
`[3, 1]`	`[[1, 1], [0, 0]]`	`[1, 3]`
`[2, 3, 0, 1]`	`[[1], [3, 3, 0], [], [0]]`	`[3, 1, 0, 2]`
`[3, 3, 1, 1, 2, 0, 0]`	`[[6, 5, 1], [6, 4, 2, 2], [], [5, 0, 0], [], [], []]`	`[0, 4, 1, 1, 2, 1, 1]`
`[3, 2, 2, 0, 1, 2, 2]`	`[[3, 5], [6, 3], [0, 3, 4], [6, 2], [5], [2, 2, 3, 4, 2, 4, 1, 2, 1], []]`	`[1, 0, 2, 2, 0, 4, 3]`

Since the total number of chips in the system and the list of obligations for each position never change, iterating this system over the infinite sequence of time steps $t = 0, 1, 2, \ldots$ must eventually enter a **cycle** where the same sequence of states keeps repeating forever. The **period** of the cycle is the smallest positive integer $k > 0$ so that for every t in the cycle, states t and $t + k$ are equal.

```
public static int period(int[] start, List<List<Integer>> obligations)
```

From the given `start` state and the list of `obligations` for each position as in the previous method, this method should compute and return the period of the cycle reached from `start`. Iterate the sequence until you come to some state that you have already seen before. That state is going to be the starting point of the cycle, so we had better remember it. Initialize an integer counter to zero, and iterate the same cycle again incrementing the count in each step. When you inevitably return to the starting point of the cycle, return this count as the requested cycle length.

Ah... but one pesky little fly still remains in the ointment. Since the number of **reachable states** in the system can be exponentially large, so can also the cycle length and especially the length of the path leading there. Making the previous algorithm to explicitly store all the states it has seen into some kind of `Set` structure might therefore be prohibitively expensive. Fortunately, there exists an ingeniously simple algorithm for this exact type of situation where we have the ability to generate some sequence arbitrarily far one step at the time, and need to recognize the appearance of some previously seen state that marks the cycle. The Tortoise and Hare algorithm by Robert Floyd is an underappreciated classic in computer science due the astonishing savings of memory paid for by a mere constant factor of added time.

Instead of maintaining only one current state inside the sequence used to advance to the next state, maintain two such states separately, aptly titled "tortoise" and "hare", both initially at the `start`. The hare will take two steps for every step of the tortoise, thus iterating through the sequence twice as fast. When the current position of the hare reaches some cycle, as it eventually must (the `start` state could be inside some cycle to begin with), the hare will keep unknowingly racing around that cycle until that the slower-paced tortoise also enters that cycle. Regardless of their positions inside the cycle at that moment, the hare will inevitably catch up with the position of the tortoise. Once that happens, both animals stop and say their equivalent of "jolly good, old chap" to each other, knowing that the current state is inside the cycle. The hare can now take a well-deserved rest and stay in place, while the tortoise makes another trip around the same cycle, counting the steps needed to come back to the hare.

As revealed by the JUnit fuzz tester, the period of this cycle greatly depends on the structure of obligations that the positions have for each other. Most of the time the correct answer seems to be just one; such **self-cycles** are the equilibrium states where all obligations and chip constraints perfectly balance out each other. However, this period can also be much longer, such as well over one million for some of the randomly generated initial states and obligation lists. As with all exponential growth, increasing the array lengths and numbers of chips in the system would make these periods eventually dwarf our entire universe itself...

Lab 25: Manhattan Skyline

JUnit: ManhattanTest.java

The Manhattan skyline problem is a well-known classic in computational geometry with many educational and interesting variations. The interesting setup makes for a good illustration of the **sweep line algorithm** used to solve this problem efficiently. This general technique of sweep line will then solve many other problems that take place on the two-dimensional plane.

In the basic version of this problem, a number of tall buildings (perhaps built in the heyday of some midwestern city on a flat plain) are seen from a distance against the horizon far away. Our vantage point is far enough from these buildings so that each building is seen as a two-dimensional **silhouette** rectangle. From our point of view towards the horizon, each individual silhouette has **starting** and **ending** x-coordinates $s < e$, and the **height** of h. These projected silhouettes can overlap each other in part or whole even as the original buildings are physically disjoint. Create a new class `Manhattan` in your BlueJ labs project, and the method

```
public static int totalArea(int[] s, int[] e, int[] h)
```

The three arrays `s`, `e` and `h` (guaranteed to be of same length n and contain only positive integer values) contain the start coordinate, the end coordinate and the height for each individual building numbered 0, ..., n-1, respectively. The buildings are guaranteed to be sorted in order of their starting coordinate, with ties resolved by the ending coordinates. This method should return the area of the overall total silhouette so that the overlapping parts of the silhouettes of the individual buildings are included only once in the total, regardless of how many buildings happen to overlap at the given x-coordinate.

Conceptually, a sweep line algorithm "sweeps" through the horizon from left to right, noting all the points along the way where something "interesting" takes place in the sense that it can potentially affect the result of the algorithm. The algorithm should start by creating a list of these interesting **events**, and sort these events in the x-coordinate order. For the Manhattan skyline problem, the interesting events are "building b enters the view" and "building b exits the view" as our imaginary "eye" scans the horizon left to right. Start by defining a local variable

```
List<Integer> events = new ArrayList<>();
```

with these "enter" and "exit" events encoded so that the building `b` entering the view is encoded as `-1-b`, and the building `b` exiting is encoded as `b-1`. (Having made that choice, we can no longer use the value zero to encode any events, since we could not tell zero and minus zero apart!) Add these $2n$ event encodings into `events`, and sort them with a custom `Comparator<Integer>` whose `compareTo` method compares the events `a` and `b` by first decoding them into starting times (from array `s`) or ending times (from array `e`), and then compares those times.

The events sorted this way can now be processed in the order that they take place, updating the tally of the area. To keep track of the buildings that are currently in your active view (the so-called **active set** of the sweep line algorithm), define another local variable

```
Set<Integer> active = new HashSet<>();
```

that contains the buildings that are currently in the active view. Loop through the sorted `events`, also keeping track of the *x*-coordinate of the event processed during the previous round. To process an event, first compute its horizontal distance from the previous event, multiply that distance by the height of the active building that is currently tallest, and add that product to the total. Next, update the active set by adding or removing the building that enters or exits the view, and move on to the next round. Once all buildings have entered and exited the active view, the method can return the accumulated total.

s	e	h	Expected result
{1, 4, 11}	{5, 8, 13]	{2, 1, 2}	15
{2, 6, 9, 12, 15}	{3, 8, 10, 14, 20}	{3, 3, 4, 3, 2}	29
{0, 9, 11, 13, 13, 18, 31, 32, 35}	{4, 23, 21, 23, 24, 21, 38, 50, 45}	{3, 3, 1, 1, 2, 2, 3, 3, 1}	113

Lab 26: FilterWriter

JUnit: FilterWriterTest.java

Some class examples featured decorators for `Animal` and `Comparator` class hierarchies. The decorator pattern works the same for any other class hierarchy, except that the more methods the public interface of some class contains, the more annoying drudgery that class hierarchy becomes to decorate. The `Writer` hierarchy to write text composed of Unicode characters is more convenient for writing decorators. Only three of its methods have to be overridden in the subclasses, since the default implementations of all other methods call those three **template methods** to do their work. Subclasses can still override these template methods whenever they can provide them with a more efficient implementation, but they are never required to do this.

In your BlueJ labs project, create the class

```
public class FilterWriter extends Writer
```

with the following fields and constructor

```
private Writer writer;
private BiPredicate<Character, Character> pred;
private char previous;
```

```
public FilterWriter(Writer writer, BiPredicate<Character, Character> pred,
char prev) {
    this.writer = writer;
    this.pred = pred;
    this.previous = previous;
}
```

The functional interface `BiPredicate<T, U>` in the package `java.util.function` represents a predicate that takes two arguments. In `FilterWriter`, this predicate is used to filter the characters that this decorator lets through. Each `FilterWriter` instance remembers the `previous` character that the filter has let through, initialized from the constructor parameter to be used as the initial character. Override the methods `flush` and `close` to call the corresponding methods of the underlying `writer`, and then override the one required method

```
@Override public void write(char[] data, int off, int len) throws IOException
```

to write the characters from the subarray of `data` that contains the `len` characters from position `off`. To decide whether the current character `c` should be passed on to the method `write` of the underlying object, the method should call `pred.test(previous, c)`. If the predicate accepts the character `c` by returning `true`, `c` is passed on to the underlying writer and becomes the new `previous` character written. Characters rejected by the predicate are simply ignored.

The JUnit test class `FilterWriterTest` contains two test methods that both create a filtered version of *War and Peace*. The first test adds a bunch of Unicode combining characters to each letter of the original line to create some "zalgo text", but the predicate that rejects all such combining characters regardless of the previous character should have no problem restoring the original text. The second test uses a predicate that rejects every vowel that immediately follows some vowel, and every consonant that immediately follows some consonant.

```
Pire was unany. Sot, abot te average heg, bod, wit huge
red han; he did not kow, as te san is, how to ener a dawin
rom an sil les how to leve one; tat is, how to sa sometin
pariculary agebe before gon awa. Besides tis he was aben-
mined. Wen he rose to go, he tok up ined of his ow, te general's
te-corered hat, an hel it, pulin at te pume, til te general
ased him to resore it. Al his aben-minedes an inability to
ener a rom an conere in it was, however, redemed by his kiny,
sime, an modes exesin.
```

The resulting text has a curious rhythm and pattern of its own, leading us to wonder whether this operation could be applied to other natural languages to create a made-up but internally coherent sounding language for some fantasy universe. The algorithm demonstrated in the class example

DissociatedPress.java can be used to do the same job. That one can even mash together several languages to obfuscate the traces of the original languages.

Lab 27: Stacking Images

For testing: ImageAlgebraMain.java

Instances of the class `Image` represent two-dimensional pixel images in Java that can be read from files and URLs, and then rendered to `Graphics2D` canvases with the method `drawImage`, as illustrated in the example class `ImageDemo`. These images, no matter where they were acquired, can be further processed and transformed with various `ImageFilter` instances, as illustrated by our other example `ImageOpDemo`.

Normally we are accustomed to adding up numbers, but we can also "add" images to each other with concatenation, similarly to the way that strings are "added" to each other. However, since pixel raster images are two-dimensional, we can stack them up together horizontally or vertically provided that the dimensions of these images are compatible in that dimension. In your BlueJ labs project, create a new class `ImageAlgebra`, and there two static methods

```
public static Image hstack(Image... images)
public static Image vstack(Image... images)
```

for the horizontal and vertical stacking of an arbitrary number of `Image` objects. Both of these methods are **vararg** methods, meaning that they can accept any number of arguments of type `Image`, including zero. The **horizontal stacking** method `hstack` (these method names were chosen to same as in numpy) should create and return a new `BufferedImage` instance whose width is equal to the sum of the widths of its parameter images, and whose height equals the maximum of the heights of its parameter images. This image should then contain all the images together as one row. (Just draw the individual images one by one to an appropriate position of the result image.) The **vertical stacking** method `vstack` works the same but with width and height interchanged. We can immediately put both stacking methods in good use in some **recursive subdivision**. Define a third method in your class

```
public static Image halving(Image tile, int d)
```

This method produces the result image according to the following recursive rule. For the base case where the depth `d` equals zero, this method should simply return `tile`. The result for positive depths `d` is the horizontal stacking of `tile` with the vertical stacking of two copies of `halving(half, d-1)` where `half` is an image constructed from tile by scaling it to half of its `width` and `height`. (Of course, you will write your recursion to not have any branching, so that the level `d` activation of this method will create only one level `d-1` activation.)

The class `ImageAlgebraMain` uses the above three methods, along with the image file coffee.jpg, to create a display that should look like this to be accepted. To test your methods one at the time as you write them, first define all three methods as one-liner **stubs** that simply return the tile they were given. This allows you to run the `ImageAlgebraMain` program to test each method as soon as it compiles, even when you haven't implemented the other methods.

The 4-by-4 grid on the left of the above low-resolution screenshot is constructed by simple application of `hstack` to create the row, and then using `vstack` to duplicate that row into the final image. The image on the right side is the result of `halving` a 512-by-512 version of the original image. Since each tile gets cut in half down the recursion, the resulting image is always twice as wide as the original, minus the width of the last image on the right edge.

Lab 28: All The Pretty Hues

For testing: ShadesMain.java

Computers typically represent colours internally as **RGB triples of red, green and blue** so that each component can get values from 0 to 1 when represented as floating point, or from 0 to 255 when represented as an integer byte. Turning all three knobs to zero gives you the darkest black that your display device is capable of showing, whereas turning all three knobs to maximum gives you its brightest possible white colour.

Inside Java, objects of `java.awt.Color` represent individual RGB colours as objects. However, when we are dealing with images whose every pixel could potentially have a different colour, we wish to avoid filling the heap memory with objects. Instead, we pack four bytes of colour information into one `int` value whose four bytes represent the ARGB components of that colour. This fourth "alpha channel" has no fixed interpretation, but all image processing operations on Java treat it as the **opacity** value of that colour. When a pixel is rendered with a colour that is partially opaque, the resulting colour of each pixel is a mixture of the current pen colour and the previous colour of that pixel.

Most colours that our eyes can see can be broken down into these three components, although some might be a bit tricky. For example, how exactly would you represent the colour brown? By now you have surely seen lights of various colours through the rainbow, but have you ever seen a brown light in your life? What the heck exactly is that colour made of? As convenient as RGB encoding is for computer software and hardware, it is not the best way for humans to represent colours. The alternative HSB representation breaks each colour into three components of **hue**, **saturation** and **brightness**. Colour calculations and especially colour interpolation produce more aesthetic results when done in the HSB colour space.

To extract the HSB colour components of the RGB colour `rgb` given as an `int` into the first three elements of an existing 3-element float array `hsb`, you can copy-paste the method

```
private static void RGBtoHSB(int rgb, float[] hsb) {
    int a = (rgb >> 24) & 0xFF; // extract alpha
    int r = (rgb >> 16) & 0xFF; // extract r
    int g = (rgb >> 8) & 0xFF;  // extract g
    int b = rgb & 0xFF;         // extract b
    Color.RGBtoHSB(r, g, b, hsb);
}
```

The conversion from HSB to RGB is already available as the method `HSBtoRGB` in the `Color` class. (Of course the inverse method `RGBtoHSB` exists already in the `Color` class, but that one expects to receive the red, green and blue values as three separate parameters, not as packed into a single `int` like the above method.)

To play around with all these pretty colours, create a class `Shades` in your BlueJ labs project. In this class, write the method to create and return a modified copy of the given `Image`. The hue and value of each pixel are kept as they were, but the saturation of each pixel is decreased by multiplying it by the `factor` guaranteed to be between zero and one, inclusive.

```
public static Image desaturate(Image img, double factor)
```

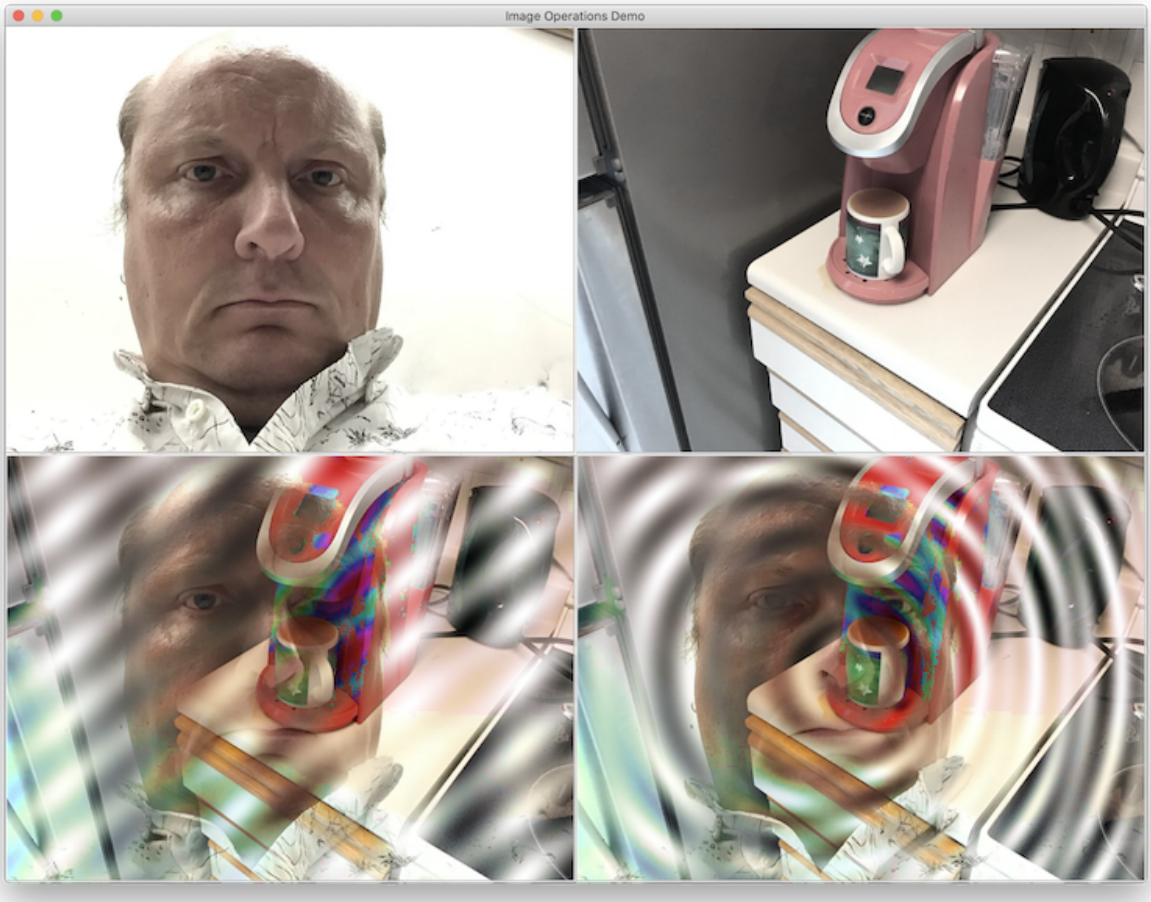
Compared to the original, the resulting image looks like somebody ran it through the hot wash a couple of times too many. The second method to write in this lab is

```
public static BufferedImage blend(BufferedImage img1, BufferedImage img2,
    DoubleBinaryOperator weight)
```

This method blends together two source images `img1` and `img2` to create a result image so that the hue, saturation and value of each pixel (x, y) of the result are the weighted averages of those things in the pixels (x, y) of the source images. The weight that is used to mix the pixel `(x, y)` is

determined by the `DoubleBinaryOperator` parameter `weight` that represents a function that takes the pixel coordinates `(x, y)` as parameters and returns the weight used in that pixel.

The following screenshot, produced by running `ShadesMain` with the example images coffee.jpg and ilkka.jpg included in the repository, shows the result that these methods are supposed to produce. The first row shows the desaturated versions of these source images. The second row shows two possible blends of these images. The bottom left image computes the blending weights from a periodic wave function directed at an angle through the two-dimensional plane, whereas the bottom right image uses a wave function emanating from the center with a superellipse shape instead of a regular circle.



A superellipse shape with parameter 2.1 or thereabouts is subconsciously perceived as more organic and pleasant than the ordinary circle whose main problem is that it is just too darn perfect to be perceived as a natural and organic thing. Few perfectly round things exist in nature, at least in the resolution that our eyes can discern; immediate examples would be the Sun and the Moon, and the iris and the pupil in the human eye. (It is probably not coincidental that humanity has historically ascribed preternatural properties to such objects.)

Lab 29: In Timely Manner

JUnit: TimeProblemsTest.java

Computations on times and dates are important in the real world, yet surprisingly difficult to get universally correct, since both of these concepts turn out much more complicated than they might seem in the everyday experience of our daily lives. However, just like our programs can no longer assume that all text is written in typewriter English so that every character can be stored in one byte, software that is going to be running around the globe 24/7/365 cannot blithely rely on any happy-go-lucky ideas of how time works. Everyone still reading this surely knows that during the year 2020 when this problem was written, those were actually 24/7/366. What other pitfalls with sharpened spikes at the bottom might be lurking in this important problem domain to catch those who try to patrol this enormous jungle, nervously scanning for traps laid by Time Cong?

To see the need for a standard package that cuts through this thicket and performs these calculations correctly in all possible cases, scan through "Falsehoods programmers believe about time" and tally up how many of the falsehoods listed there you would have also assumed to be true before doing this lab. The rest of the big curated list of "Falsehoods Programmers Believe In" gives you a glimpse of how both the world and human societies are chock full of messy edge cases and corner cases that trip unwary programmers who have to deal with these issues without any hand waving! Normal humans have the option of closing their eyes and hand-waving at such problems until they go away as all problems eventually do, but computer programmers do not have such luxury.

Before Java 8, the standard library offered the classes `Date` and `Calendar` to encapsulate such computations, but these classes were just defective in too many ways to count. The class `Date` is now officially **deprecated**, and `Calendar` remains as legacy for backwards compatibility. The far superior package `java.time` introduced in Java 8 offers a more intuitive and consistent design that is well worth studying just for the object oriented design standpoint from all perspectives in this course, once you have first learned to use its classes in methods in general.

The classes `LocalDate`, `LocalTime` and `LocalDateTime` represent dates and times without any time zone information. (Pythonistas may note that these roughly correspond to classes `date`, `time` and `datetime` in the `datetime` module.) Unlike the original `Date` class, all such data types in `java.time` are designed to be **immutable**, giving us a multitude of benefits that simplify our reasoning about programs that use them. Their methods are consistently named across these types, and they accept intuitive and human-readable arguments. For example, the method that creates a copy of an existing object with just one attribute changed and the rest kept as they were (such a method would, kind of, be a "mutator" for an immutable type) is always named `with`.

The first thing to note with these classes is that **none of them have a public constructor!** Instead, new instances are created using the **static factory method** that is always named `of`, usually **overloaded** to accept various types of arguments for object creation. For example, the expression `LocalDate.of(2020, 4, 23)` would return a `LocalDate` object that represents the date of April 23, 2020, the day that this lab specification was written.

Create a new class `TimeProblems` into your BlueJ labs project. The first of the three methods that you will be writing there has you perform some calculations on dates. The whole point of this entire lab is to teach you to read through the package and class documentation in the Java API Reference, usually googling through some Stack Overflow pages in another browser tab to get through the practical details and find the method that does the thing that you need at the moment. You should first think how you would solve this problem in real life, and then translate your ideas to code with the individual steps translated to appropriate method calls.

```
public static int countFridayThirteens(LocalDate start, LocalDate end)
```

This method should count and return the number of days between the `start` and `end` dates, both inclusive, that are Fridays whose date falls on the thirteenth day of that month. This particular combination used to be considered unlucky in the Western culture, at least back in the day when people were a bit easier to dazzle with numerological stuff. (Many such beliefs seem to be slowly fading into distant memories in our culture.)

The next problem was inspired by the "Gigasecond challenge" at CodeWars, another good online programming problem collection stone for anyone looking to sharpen their coding and algorithmic thinking skills against. To solve this problem, we need a new concept of `Duration` to represent a measured **passage of time** (for example, "two hours and five minutes", "three years and ten seconds", or "one nanosecond") that is not anchored to any particular time or place. (`Period` is similar in spirit and usage, but with a granularity of days instead of nanoseconds.)

To ensure that you understand the difference between the concepts represented by `Duration` and `LocalDateTime`, can you explain why adding or subtracting a `Duration` to a `LocalDateTime` or another `Duration` is a perfectly sensible thing for somebody to do, but trying to add up two `LocalDateTime` objects would be pure codswallop? Even more curiously, though dates cannot be meaningfully added to each other, the **difference** between two `LocalDateTime` instances is a perfectly meaningful operation that gives out a `Duration` result!

```
public static String dayAfterSeconds(LocalDateTime timeHere, long seconds)
```

Given the current date and time as one `LocalDateTime` object, return the day of the week (as one of the capital letter strings `"SUNDAY"`, `"MONDAY"`, `"TUESDAY"`, `"WEDNESDAY"`, `"THURSDAY"`, `"FRIDAY"`, `"SATURDAY"`) that would be the current date if the given number of `seconds` were added to the current date and time.

The previous two problems treated time as a local phenomenon, the way we do in our local everyday lives. Such a pragmatic approach to time is analogous to how, even though we understand that the Earth is a spheroid, all our decisions and actions in our daily lives occur as if we believed that the surface that we live and work on is an effectively flat plane with occasional minor bumps. (Students who work as airline pilots or artillery operators must use a different mental model while on the clock so that they don't miss their marks, of course.)

The spinning globe whose surface we live on is divided into **time zones** so that the same instant of time (we shall ignore the theory of relativity and any philosophical issues about what it means for two instants of time to be the "same" instant, with or without any "quantum" mumbo-jumbo) corresponds to a different time and date depending on your current location on Earth. Instances of `ZoneId` represent the recognized time zones, and are identified by standard strings such as `"America/Toronto"` or `"Pacific/Marquesas"`. However, before we casually extrapolate the general form of these strings from mere $n = 2$, note that `"Eire"`, `"Navajo"`, `"US/Hawaii"` and `"America/North_Dakota/New_Salem"` are also legitimate time zones! You should also note the use of underscores as artificial whitespace the same way as we do in C and Python with names made from multiple words, so that you don't get confused when there is no time zone of `"America/Los Angeles"`. The world sure can be a big and messy place! Fortunately, all this work has been done in the `ZoneId` and other classes, so that you don't have to repeat it.

```
public static int whatHourIsItThere(LocalDateTime timeHere, String here,
String there)
```

Assuming that you are currently `here` so that the clock on your wall says `timeHere`, compute and return the current hour that the people way over `there` see if they look at their clock, so you know whether you can make that important business call right now.

One last thing for you to ponder: why is the first parameter of this method a `LocalDateTime` instead of a mere `LocalTime`? In other words, in what situation would knowing the current local date, not merely the current local time, be necessary for returning the correct answer? The great thing about this library is that you don't need to know this! Instead of doing these tedious time calculations and doping research to determine all the lurking edge cases all by yourself, let the class `ZonedDateTime` take care of all such low level details, since some expert in this field already did all the necessary grunt work for you. This principle applies to all real world problem domains that we deal with in computational means. Note especially the `withZoneSameInstant` method as the "mutator" for immutable objects to access the same instant but in a different time zone...

Lab 30: Mark And Rewind

JUnit: RewindIteratorTest.java

Instances of `Iterator<E>` allow the elements to be iterated only in the forward direction with the method `next`, but not rewinding the sequence to an earlier position to repeat the elements of the sequence that we already saw the first time around. Since this ability would sometimes be useful in many algorithms that operate on sequences, we shall write a general purpose **decorator** for `Iterator<E>` instances that allows a position in any existing sequence to be **marked**, so that the decorated sequence can afterwards be **rewound** back to that position.

The mass test method of the JUnit test class `RewindIteratorTest` uses the simple sequence of natural numbers 0, 1, 2, ... as the underlying sequence, and then performs "mark" and "rewind" operations at pseudo-randomly chosen positions of the sequence. The three public test methods generate the first thousand, first million and first hundred million elements of this sequence. If you edit the test method for the first thousand elements to be `verbose`, the first eight lines of output (shown below as generated using the instructor's private model solution) help you trace what is supposed to happen, with `M` and `R` denoting marking and rewinding, and numbers denoting the element that pops out of the decorated sequence at that moment.

```
0 1 2 M R 3 4 5 6 M 7 R 7 8 9 10 11 12 13 14 15 16 M 17 R 17
18 19 20 M 21 22 23 24 25 26 27 M 28 M R 29 R 28 29 30 31 R 21 22 23 24
25 26 27 M 28 M 29 30 31 R 29 R 28 29 30 31 32 33 M R 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 M 52 53 54 55 56 57 58 R 52
53 54 55 56 57 58 59 60 61 62 63 64 M 65 66 M 67 68 69 M 70 M R 71 72
R 70 M R 71 72 73 74 75 76 R 67 68 R 65 66 67 68 69 70 71 72 73 74 75
76 77 78 79 M 80 81 82 83 84 85 86 87 R 80 81 82 83 M 84 85 86 87
R 84 85 86 87 88 89 90 M 91 92 93 94 95 96 97 98 99 100 101 M 102 103
```

The reader might want to trace the effect of the mark and rewind operations by hand for the first two or three lines. Since the original underlying sequence produces each natural number exactly once and in order, the repeated appearances of each number are due to the rewinding operations.

Create a new class with the following signature in your BlueJ labs project:

```
public class RewindIterator<E> implements Iterator<E>
```

Being a decorator, this class should have a private field

```
private Iterator<E> it;
```

that refers to the underlying iterator object being decorated, initialized in the constructor

```
public RewindIterator(Iterator<E> it) {  this.it = it; }
```

Since the underlying iterator does not remember the elements it has produced, your decorator has to store these elements into some kind of list from which they can be extracted for their repeat appearances. The simplest way to do this is to use two such lists, perhaps named

```
private LinkedList<E> buffer = new LinkedList<E>();
private LinkedList<E> emitted = new LinkedList<E>();
```

so that `emitted` contains the elements that have been given out by this decorator with its `next` method, and `buffer` contains the elements that have been rewound and are waiting to be given out again. For the iterator functionality, this class must have the methods

```
@Override public boolean hasNext()
@Override public E next()
```

The decorated iterator has a next element if either its `buffer` is nonempty or if its underlying iterator has a next element. The `next` element produced by this decorator is then the first element of the `buffer`, and in the case that the `buffer` is empty, the next element of the underlying iterator. If any marks are active at the moment (that is, they have been marked but not yet rewound), the element is pushed into the `emitted` list.

```
public void mark()
public void rewind() throws IllegalStateException
```

These two new methods implement the functionality of marking the current position in the sequence, and later rewinding the sequence back to that position to repeat its elements. It is your task in this lab to implement these methods so that they behave correctly in all possible situations. Note that multiple positions can be marked at the same time, and the same position can be marked multiple times. The method `rewind` always rewinds the sequence to the most recent marked position, or if there are no active marks at the moment, reports this to the caller by throwing an `IllegalStateException`. It might be a good idea to use a third instance field

```
private LinkedList<Integer> markPositions = new LinkedList<Integer>();
```

that contains all the positions in the `emitted` list that have been marked at the current time. (If this list is empty, the emitted elements do not need to be stored in `emitted` to waste memory, since such elements will never be needed again.)

Lab 31: Region Quadtrees

JUnit: QuadTreeTest.java

A two-dimensional image that consists of binary pixels (every pixel is either black or white) can be stored as a two-dimensional array of `boolean` truth values. This representation enables us to access and change the value of each pixel independently of other pixels. Since each pixel now takes exactly one bit of memory (plus some pocket change for the two-dimensional array structure), our available memory sets a limit to the size of the images that can be stored this way.

A simple combinatorial argument based on the pigeonhole principle, usually paraphrased in its bumper sticker form "There just aren't enough short bit patterns to uniquely encode not only themselves plus also some longer bit patterns", establishes that no general method can possibly compress data made up of arbitrary bits in a way that would **always** produce a smaller result. For compression to be possible, the data must be somehow distinguishable from random noise so that some kind of exploitable statistical dependencies and correlations exist between these bit values. Compressing that data then erases such dependencies; otherwise the compressed data could be run through another compression algorithm to squeeze it even tighter, rinse and repeat. You can observe this phenomenon in practice by trying to use your favourite compression algorithm to compress data that has already been compressed. Once all that metaphorical "air" has been squeezed out, the **entropy** of the data ultimately determines how many bits are needed to store it so that the original can still be unambiguously recovered.

Two-dimensional binary images that contain large uniform areas of either colour can sometimes be compressed into much smaller space with the **region quadtree** approach. Each quadtree object represents some square area of pixels so that the side is some power of two. If this square is all white or all black, the object is a leaf node of the quadtree structure. For square areas that contain both black and white pixels, the square is subdivided into four smaller squares that are recursively converted into region quadtree objects that become children of the quadtree object that represents the original square area. The subdivision ends at a single pixel, but hopefully much sooner in areas of uniform colour inside the image.

To do this lab, you should first copy the interface `QuadTree` and its two implementations `WhiteQuad` and `BlackQuad` into your BlueJ labs project. The interface `Quadtree` defines the functionality of the region quadtree as two methods

```
public boolean isOneColour();
public long computeArea(int scale);
```

The method `isOneColour` should return `true` if the region depicted by this `QuadTree` object represents an area of uniform colour, either all black or all white, and `false` otherwise. The method `computeArea(int scale)` should return the total area of the black pixels in the region under the assumption that the entire region is a square whose side length equals $2^{\texttt{scale}}$. Note that Java does not have integer exponentiation operator in the language similar to ** operator in Python. However, this particular case where the base equals two can be computed easily using the **bit shift** operator << with the expression `(1L << scale)` to compute $2^{\texttt{scale}}$. Note that use of `long` literal `1L` allows us to potentially use up to 63 bits of scale instead of 31 bits we would get from using the

bare `int` literal `1`. (Not 64 and 32 bits, as the highest bit stores the sign in these **signed integer** types.)

Before writing the class `QuadNode` of this lab, check out how these two methods of the `QuadTree` interface have been implemented in the classes `WhiteQuad` and `BlackQuad`. These classes have `private` constructors so that no new objects can be created from outside. Instead, the outside world must use the `get` method to access the **singleton** instances of these classes, so that we get to demonstrate the (in)famous **singleton design pattern** for the first time in this course. (Some schools of thought consider singleton to be an **anti-pattern** in object-oriented design. The debate continues.)

```
public final class QuadNode implements QuadTree
```

The instances of this class represent region quadtrees that are not of uniform but contain both white and black pixels. In the spirit of this course, this class to be **immutable** so that we get to practice **caching** of results the first time that they are computed, and the future calls of those methods will simply look up the previously computed result. This is an enormous advantage that immutable types enjoy over mutable types. This class should have the following three instance fields:

```
private QuadTree[] children;
private long area = 0;
private int lastScale = Integer.MAX_VALUE;
```

The four-element array `children` contains references to the four child quadtrees that represent the quarter squares (well, there is a name for some barbershop quartet) that this square is subdivided into. Since these children can be arbitrary region quadtrees, the type of this field is `QuadTree[]` instead of `QuadNode[]`.

The field `area` will cache the computed area of this quadtree, and `lastScale` will remember the scale in which that area was computed. Both fields are initialized with impossible placeholder values that allow us to recognize them as being uninitialized. Same way as the earlier example implementations `WhiteQuad` and `BlackQuad`, the class `QuadTree` should also have a private constructor to disallow any object creation from the outside. To ensure the immutability of the instance after creation, we make a **defensive copy** of the argument array for private storage, instead of just storing a reference to the array object that the outside world is still using and potentially modifying later. However, since the `QuadTree` objects in this array are immutable, we don't need to defensively copy them, but can sit back and enjoy the main benefit of immutable types in how they can be safely shared between any number of users!

```
private QuadNode(QuadTree[] children) {
    this.children = children.clone();
}
```

To allow the outside world to create new quadtree structures to their heart's content, we provide a **static factory method** instead:

```
public static QuadTree of(QuadTree... children)
```

Unlike a constructor, a static factory method is not under any obligation to return specifically a `QuadNode`, but can return a different subtype of `QuadTree` depending on the current situation. Since this constructor is intended to take a small array as parameter, we might as well make the parameter to be a vararg so that the array elements can simply be given as separate arguments to this method, instead of the caller having to first create a small array object explicitly. This factory method should recognize the special situations where all four children are either uniform white or uniform black, and return `WhiteQuad.get()` or `BlackQuad.get()` accordingly. Otherwise, this method creates and returns a new `QuadNode` object with the given `children`.

Provided that you do all that, the method `isOneColour` is trivial to implement:

```
public boolean isOneColour() { return false; }
```

Otherwise, this method would have to do the work of recursively testing if all its children are uniformly the same colour, potentially probing through the entire tree structure to come up with the answer. The same would happen also if the child trees were mutable. We get to enjoy the asymptotic running times of our methods jumping from O(n) all the way down to O(1) so very rarely that every occasion is worth celebrating at least a moment. Immutable types, huzzah!

```
public long computeArea(int scale)
```

Assuming that `this` quadtree represents a square whose side length is `(1 << scale)`, recursively compute and add up the areas of the child quadtrees. Before returning the answer to the caller, store the area and the `scale` used in its computation in the instance fields `area` and `lastScale` so that the next time this `computeArea` method is called, you can just look up the answer instead of computing it all the way down with recursion each time.

(What if the previously cached `area` was computed using a value for `lastScale` that is different from the `scale` in the current call? Well, assuming that `lastScale < scale`, you can simply adjust the previously computed `area` for the new `scale`, and even in the opposite cases where `scale < lastScale`, there are things you might do...)

Lab 32: Triplefree Sequences

JUnit: TripleFreeTest.java

Define a list of positive integers to be **triplefree** if it does not contain three elements `a < b < c` so that differences `c - b` and `b - a` are equal. Another way to express this requirement is that for any two elements `a < b` in the list, `a + 2*(b-a)` is not a member of the list. For example, the lists `[2, 3, 5, 6]`, `[1, 2, 7, 8, 10, 11]` and `[1, 2, 6, 9, 13, 18, 19, 21, 22]` are all triplefree. Try as you may, you won't find three elements forming a three-step arithmetic progression in any of them. For example, since the last list contains both elements 18 and 21, it cannot contain either one of the numbers 15 or 24.

Create a class `TripleFree` in your BlueJ labs project, and in that class, a public method

```
public static List<Integer> tripleFree(int n)
```

that finds and returns the longest possible triplefree list whose largest element is `n`. Since in general there exist several lists of the same length that each satisfy the triplefree constraint, this method must return the **lexicographically largest** such list when the elements are read down from highest to lowest. For example, the lists `[1, 3, 4, 6]` and `[2, 3, 5, 6]` are both triplefree and contain four elements, but the method must return the second list for `n = 6`, since for the highest position where these two lists are different, the value 5 in the second list is greater than the value 4 in the first list.

n	Expected result
2	`[1, 2]`
4	`[1, 3, 4]`
9	`[1, 2, 6, 8, 9]`
15	`[2, 3, 5, 6, 11, 12, 14, 15]`
42	`[2, 3, 5, 6, 11, 12, 14, 15, 29, 30, 32, 33, 38, 39, 41, 42]`

This problem is best solved bottom up, building the answer list with recursion that solves the problem for `n` with two recursive calls for two subproblems with `n-1`. Write a helper method

```
private static List<Integer> tripleFree(int n, Set<Integer> chosen)
```

that finds and returns the answer for integer `n` when the `chosen` numbers have already been taken into the sequence in the previous levels of recursion. Your public method can then make the top level initial call to this private recursive method with the second parameter initialized to some Set<Integer> object that the recursion fills in during its journeys down the search tree.

In each recursive subproblem, you have two choices: either you take `n` into your `chosen` set, or you do not. However, you can only take `n` into your `chosen` set if it does not create a triple with two

numbers that already exist in `chosen`. Whichever way gives you a better solution, return that list as the answer. If both ways are equally good, use the way that takes `n` in `chosen`, to ensure that you return the lexicographically largest solution.

The basic recursion described above solves the problem in reasonable time for `n` up to twenty or thereabouts, but for higher values on `n`, some branches that cannot possibly contain the best solution need to be mercilessly pruned on the spot, instead of recursing through that entire branch and coming up with a solution will eventually turn out to be suboptimal. For starters, the JUnit test class `TripleFreeTest` will loop through values of `n` in ascending order, so your class can remember the length of the best list for each `n`.

Since adding new constraints to any optimization problem can never improve the cost of solution, the stored optimum for unconstrained `n` will be an upper bound to the solution with any set of `chosen` elements. For example, suppose you have already found a 15-element solution to your original top-level value of `n`. If `chosen.size() == 5` in some branch down in the recursion, and the previously discovered optimal unconstrained solution for the current level `n` is 10 or less, this branch can be automatically dismissed as a failure. Any solution available in this branch contains at most 15 elements, and can therefore be no better than the lexicographically higher 15-element solution that you have already found.

Many other optimizations can be used to prune the unproductive branches, making this problem a good exercise in pruning down exponential recursions. If you really like to try out the value of your optimizations, uncomment the `testFirstHundred` method in the JUnit test class, and brace yourself to run it. Can you make your code complete that test within one hour?

A related and more famous problem of constructing the shortest possible Golomb ruler hardens these constraints so that the same difference between two element values may not occur anywhere in the array, not just in its two consecutive positions.

Lab 33: Sardine Array

JUnit: SardineTest.java

The primitive integer types `byte`, `short`, `int` and `long` of Java represent signed integers using one, two, four and eight bytes of memory, respectively. These primitive types are **signed** so that they allow both positive and negative values. Same as with other types in Java, we can create arrays to store a sequence of elements of the same type, although when the element type of an array object is a primitive type, these elements are stored in consecutive bytes with no additional bookkeeping or references needed.

However, in practice many programs operate on problem domains where the relevant integer values are known to be unsigned natural numbers. When such values are stored in Java primitive

integer types, the highest order bit that represents the sign of the integer in two's complement encoding is essentially wasted. Furthermore, the problem domain also occasionally inherently limits these unsigned integer values so that each such value is known to fit into *k* bits. When *k* is something convenient like 14 or 30, there is no problem since we can just use the appropriate primitive type that these values almost fill up with little ensuing "air".

However, for the sake of argument, suppose *k* is some more annoying number such as 17. We can't use the `short` type to store such values, since one `short` can carry only 16 bits of information. We are forced to use the next higher type `int` with 32 bits of information, of which 15 bits, almost half of them, end up being wasted air. Wasting almost half of your available bits of memory is not as big a deal as it used to be back in the day when the memory for each process was measured in kilobytes. Being able to use these bits more productively would allow our programs to solve bigger instances of the same problem within the same confines of heap memory given to that program.

In this lab, we shall design and implement a general purpose class whose instances behave as simulated "arrays" of unsigned *k*-bit integers. An "array" of *n* such numbers is stored inside an ordinary `boolean[]` that contains exactly *nk* bits of information, with less air inside than a shipping crate of Ikea furniture packed flat, or those titular sardines packed into a can. The computations in this lab are in fact so low-level that we don't even need to import any classes from the standard library! The only new operation that you need here is the **bitwise left shift** operator `(1 << k)` to quickly compute 2^k with only integers and no floating point arithmetic.

Create a new class `Sardines` that contains two instance fields `data` and `k`.

```
public class Sardines {
    private boolean[] data;
    private int k;
```

The class should have a public constructor that initializes these fields.

```
public Sardines(int n, int k)
```

Since not all possible values of `n` and `k` are meaningful, this lab also makes you throw exceptions to report an inevitable failure.. This constructor should throw an `IllegalArgumentException` whenever (a) the array size `n` is negative, or (b) the total number of bits `n*k` is not less than `Integer.MAX_VALUE`, or (c) the number of bits per element `k` is negative or greater than 31. The method `testExceptions` in the JUnit test class `SardinesTest` will make sure that your code enforces these restrictions gracefully and throws the correct kind of exception each time..

To read and write the value of an element in the simulated array, this class should provide the accessor and mutator methods

```
public void set(int idx, int v)
```

```
public int get(int idx)
```

to `set` and `get` the value of the element in the position `idx`. The *k* bits that constitute the binary value in position `idx` of this instance of `Sardines` are stored into the *k* consecutive positions starting from position `k * idx` in the underlying array `data` of `boolean` values. For example, when `k == 5`, the first element is stored in bits 0 to 4, the second element is stored in bits 5 to 9.

Same as in the constructor, these methods must throw an `ArrayIndexOutOfBounds` exception if `idx` is outside the range of legal indices `0`, ..., `n-1`, and throw an `IllegalArgumentException` if the element value `v` is not representable as *k* bits. Again, the test method `testExceptions` in the JUnit test class `SardinesTest` requires you to throw these exceptions correctly to pass the test.

Lab 34: Runoff Voting

JUnit: RunoffVotingTest.java

For elections that choose exactly one winner among *C* competing candidates, the Anglosphere nations tend to use the "first-past-the-post" system where each voter casts a ballot for exactly one candidate, and the candidate with the largest number of votes wins without necessarily achieving an actual majority of votes cast. This system is simple and predictable in its consequences, but has been criticized for its various shortcomings and paradoxes, such as "spoiler candidates" who induce **tactical voting** where some voters cast a ballot against their real preferences.

This lab examines two more advanced voting systems where each voter, instead of choosing just one candidate to vote for, must rank all *C* candidates in the preference ordering of that particular voter. For our purposes, each `ballot` is therefore a one-dimensional integer array that contains the numbers 0, ..., *C* - 1 in some permutation. For example, let *C* = 4 and number the candidates with 0, 1, 2 and 3. The ballot `{3, 1, 0, 2}` indicates that the voter wants to see the candidate 3 win, but if that candidate cannot win, the voter would then want to see the candidate 1 win, and so on. Both methods below to implement this week receive such `ballots` as a two-dimensional array of integers whose individual ballots are rows. When each ballot contains more information than merely identifying the voter's most favoured candidate, the voting systems can be modified to fairly use all this information in determining the winner.

Create a class `RunoffVoting` in your BlueJ labs project. This class will have two static methods to implement two different voting systems operating on these `ballots`, both returning the winner of that election as the result. For simplicity, the methods can assume that each individual `ballot` array has been completely filled so that it is a permutation of 0, ..., *C* - 1. To guarantee the uniqueness of expected results for the JUnit fuzz tests, all ties that occur during the calculation to determine the winner must always be resolved in favour of the higher-numbered candidate.

```
public static int condorcetMethod(int[][] ballots)
```

Resolve the winner of the election with the pairwise Condorcet method. In our variation to guarantee a winner within this same method without resorting to additional tie breaker methods, all candidates engage in a pairwise match against each other candidate. Each pairwise match between two candidates Alice and Bob counts how many ballots rank Alice above Bob, versus how many ballots rank Bob above Alice, temporarily ignoring all other candidates during the pairwise match between Alice and Bob. The winner of each pairwise match earns one "match point". The candidate with the largest total number of these match points wins the election.

To make your code pass the JUnit test within a reasonable time, make sure that you are again not being a "Shlemiel"; don't loop through each ballot any more times than you actually need to.

```
public static int instantRunoff(int[][] ballots)
```

Resolve the winner of the election with the instant runoff method. Initially, each candidate gets the ballots that placed him as the first choice. As long as no candidate has more than half of the ballots cast and thus immediately win, the candidate who currently has the fewest ballots is eliminated so that their current ballots are dealt between the remaining candidates so that each ballot goes to the next candidate down the preference line in that ballot.

Once a candidate has been eliminated, they are out for good, and no future adjustment can bring them back to the process. Some "apple pie" candidate (not as some sort of nostalgic Americana, but for being everybody's second choice but nobody's favourite; in other words, a neutral candidate who nobody loves but nobody hates either) will therefore be eliminated the very first round, even though in the grand scheme of things, that candidate would have been the optimal choice in the hypothetical situation of extremely fragmented partisan jingoism where every voter loves their own dear leader and hates every other candidate other than Mr. Apple Pie...

The critical step for performance of this method is the redistribution of ballots from the candidate being eliminated. To ensure that you can do this in a manner that does not resemble the way that "Shlemiel" paints the fence running back and forth at the parts of fence he has already painted before, you should maintain a list of lists of votes held by each candidate (some kind of `List<List<Integer>>` would probably be good for this) so that you can quickly loop through the ballots of the eliminated candidate without having to loop through all ballots and finding out for each one which candidate it currently belongs to.

Lab 35: Multiple Winner Election

JUnit: MultipleWinnerElectionTest.java

The Anglosphere countries tend to use voting systems where each district elects exactly one representative, typically done with the crude first-past-the-post method to determine the sole

winner of each district. Many other countries use some kind of party list proportional voting system where each electoral district elects multiple winners. Each voter casts a vote for a particular party, either directly (closed list) or indirectly (open list). The seats that are up for grabs in each district are dealt out to the parties in proportion to their votes in that district.

In this lab, you implement three methods to compute the distribution of `seats` to the parties according to the votes that these parties received. The methods of **D'Hondt, Webster and Imperiali**, each one a special case of the highest averages method, are otherwise identical, but use a different formula to compute the "priority quantity" of each party. You might wish to implement this as one private method that the following three `public` methods will then internally pass the buck to and pass its result to the caller.

```
public static int[] DHondt(int[] votes, int seats)
public static int[] webster(int[] votes, int seats)
public static int[] imperiali(int[] votes, int seats)
```

The parameter array `votes` contains the number of votes received by each party. The `seats` are given out one at the time the same way as in the Huntington-Hill method that we encountered back in Lab 0(G) so that each seat goes to the party that currently has the highest priority. To make the results unambiguous for the JUnit fuzz tests, all ties should be resolved in favour of the party whose number is higher, same way as in the "Runoff Voting" lab. The priority quantity of the party that received v votes and has so far been given s seats by this process is v / (s + 1) in the D'Hondt method, v / (2s + 1) in the Webster method, and v / (1 + s/2) in the Imperiali method. You should perform all these calculations with exact integer `Fraction` arithmetic, of course!

The returned `result` should be an `int[]` of the same length as the parameter array `votes`, each element indicating how many seats were given to that party. For example, in an election between four parties where `seats = 21` and `votes = {23, 26, 115, 128}`, the D'Hondt method would return `{1, 2, 8, 10}`, the Webster method would return `{2, 2, 8, 9}`, and the Imperiali method would return `{1, 1, 9, 10}`. These results illustrate nicely how the choice of the divisor sequence in the priority quantity calculation can make this process favour either the larger parties or the smaller ones, the Webster method being the most favourable for small parties, and the Imperiali method being the least favourable.

(What would we call a method where the priority quantity is computed as v / 2^{s+1} so seats are dealt logarithmically so that doubling your number of votes gives you one additional seat? Let some political science expert ponder that one and let us know. Some sort of monarchy where each social estate from peons to nobility gets essentially one seat of power?)

Lab 36: Rational Roots

JUnit: RationalRootsTest.java

Everybody has surely at least heard about the quadratic formula back in high school, but since that already brings up nasty square roots and can produce irrational and even complex numbers, we shall leave that problem for higher level languages that can handle algebraic computations on symbolic expressions. Instead, we shall implement a handy method, based on the rational root theorem for polynomials, that is guaranteed to find all rational roots of rational polynomials of arbitrary degrees, not merely for the quadratics. The method itself is explained clearly on the page "Finding zeroes of polynomials" of the course "Algebra" in "Paul's Online Notes", the best and most concise site among undergraduate calculus courses this author is aware of.

Same as in the three earlier labs that asked you to implement the `Polynomial` class, we represent a polynomial as an array of its coefficients so that the *i*:th element of that array contains the coefficient of its *i*:th order term. For example, the polynomial $5x^6 - 17x^3 + 8x^2 - 42$ would be represented as the integer array `{42, 0, 8, -17, 0, 0, 5}` reading the coefficients from the lowest order term to the highest. Terms that are missing from the polynomial (in this example, the exponents 1, 4 and 5) simply have a coefficient of zero.

As explained on the page linked above, the rational number *b* / *c* where *b* and *c* are integers and the fraction is in lowest terms can be a root of a polynomial whose lowest coefficient is *t* and the highest coefficient is *s* only if the numerator *b* is some factor of the lowest coefficient *t*, and the denominator *c* is some factor of the highest coefficient *s*. Since we can easily loop through all possible pairs (*b*, *c*) of integers that satisfy that condition, all that remains is to evaluate the polynomial at each such fractional point *b* / *c* and remember those points that make the polynomial evaluate to zero. For example, for *b* / *c* to be a rational root of the previous example polynomial, *c* must be one of the factors of 5 (therefore either 1 or 5) and *b* must be one of the factors of 42 (therefore one of 1, 2, 4, 6, 7, 14, 21, 42). This gives a total of 2*8*2 = 32 possible combinations for us to try; two for *c*, eight for *b*... and each such pair in two possible ways, since we have to try both possibilities *b* / *c* and -*b* / *c* for the sign of that root.

The operation to evaluate the given polynomial at given rational *x* is best written into a separate method of its own so that we can unit test and debug it. Once we are confident this method is working, it will be much easier to write the method that finds the rational roots of the polynomial, since the failures of that method to produce the correct answer for some test case can then be isolated to the body of that method for debugging.

```
public static Fraction evaluate(int[] coefficients, Fraction x)
```

This method evaluates the polynomial with the given `coefficients` at point `x`. The answer must be computed and returned as an exact `Fraction` object. However, please don't be a Shlemiel and evaluate each term separately, but realize that you can get the next term x^{k+1} with just one more multiplication from the previous term x^k.

```
public static List<Fraction> rationalRoots(int[] coefficients)
```

Finds and returns the list of all rational fractions of the polynomial with the given `coefficients`. To make the result unique and unambiguous for the JUnit fuzz tests, the list must contain the rational roots of that polynomial in sorted ascending order, with each root listed only once. Loop through the possible rational roots in two nested for-loops, using the previous method to `evaluate` the polynomial at that point, and storing the point to result if the polynomial evaluates to zero there.

For example, given the integer coefficients `{333, -432, -16}`, this method would find and return the list `[-111 / 4, 3 / 4]` that reveals two rational solutions for that polynomial. Given the coefficients `{319410, -207597, 48337, -4830, 176}`, this method would return four rational solutions in the list `[42 / 11, 13 / 2, 65 / 8, 9]`.

As a potentially important aside for your future, these kind of test cases for this problem with known rational solutions were trivial to create with *Wolfram Mathematica*, a very high level programming language that operates directly on symbolic expressions that are kept in their symbolic forms, and evaluated lazily according to the rules of mathematics. A screenshot from the session used to create the automated test class shows the idea of building up symbolic polynomials with desired rational roots:

```
In[67]:= p = Collect[(x - 13 / 2) (x - 27 / 3) (x - 42 / 11) (x - 65 / 8), x]

Out[67]= 159 705/88 - 207 597 x/176 + 48 337 x^2/176 - 2 415 x^3/88 + x^4

In[69]:= ReplaceAll[p, {{x → 13 / 2}, {x → 27 / 3}, {x → 42 / 11}, {x → 65 / 8}}]

Out[69]= {0, 0, 0, 0}
```

Even rudimentary knowledge of Mathematica would make all undergraduate level math courses immensely easier, which is why the author finds it rather odd that this wonderful system is not taught to everybody from day one. Plus, even though even many users of Mathematica don't seem to know this, Mathematica is a pretty awesome programming language also to actually solve real problems in all walks of life outside mere mathematics, especially when integrated with the Wolfram Alpha knowledge base. As high-level languages go, moving from Python 3 to Mathematica should feel pretty similar as moving from Java to Python 3. Interested students can create themselves a free account on Wolfram Cloud to try out the language, and check out either "An Elementary Introduction to Wolfram Language" or "Fast Introduction to Programmers" according to taste to find out what to do in, on, over, under and with that account.

Lab 37: Big Ten-Four

JUnit: TenFourTest.java

Standing on the integer a, you can step into either $10a$ or $10a + 4$, and whenever a is even, you can also step into $a / 2$. For example, if you are currently at 17, you can step into either 170 and 174. If you are currently at 42, you can step into 420, 424 or 21, which together sound like a fun weekend in Vegas. As the instructor learned from "In Polya's Footsteps", a collection of essays on entertaining mathematical topics by the late great Ross Honsberger, you could get from $a = 4$ to any other positive integer by using only these three moves in a suitably clever sequence!

This lab examines a mathematically less sophisticated but universally effective **systematic search** method of breadth-first search guaranteed to discover the shortest path of steps to get from the number 4 into any given n. Implementing this search for this problem teaches you the general idea of how to solve other problems of this nature with the systematic approach to searching all possibilities in such breadth-first manner. Create a class `TenFour` into your BlueJ labs project, and there the method

```
public static List<Integer> shortestPath(int n, int limit)
```

to find the shortest path from 4 to `n` under the additional constraint that you are not allowed to move into any number that is greater or equal to `limit`. If the goal number `n` is not reachable under this additional restriction, this method should return an empty list, and otherwise it should return the list of numbers encountered along the path from 4 to `n`. For the logic of breadth-first search, you should define the following local variables:

```
boolean[] seen = new boolean[limit];
int[] parent = new int[limit];
LinkedList<Integer> frontier = new LinkedList<>();
```

The array `seen` keeps track of which numbers we have already discovered in our search that will be spreading out from the starting number 4, the only number that we have seen so far. Every time that we discover some previously unseen number, we note its `parent` from which we came into this number. The `frontier` list contains the numbers that we have discovered at the moment but not yet **expanded** by looking at their neighbours. This `frontier` list should be an instance of `LinkedList<Integer>` instead of our golden child and the usual go-to guy of `ArrayList<Integer>`, because to implement a **first-in-first-out queue** this algorithm needs, we want to add elements at one end of the list, and remove them from the other end. Depending on which you slice it, either the addition or the removal operation for `ArrayList<Integer>` would necessarily take a linear time to restore the element structure to be consecutive without gaps, whereas a `LinkedList<Integer>` guarantees both operations to take only constant time.

To start this circus, mark the number 4 as having been `seen`, and add 4 into the `frontier` queue. Then, use a while-loop that keeps going until the `frontier` becomes empty. Pop out the first element of the `frontier`, the current number for this round that we shall call `v`. (During the first round of this loop, necessarily `v = 4`.) Look at the three numbers `10*v`, `10*v+4` and `v/2` (this last one only when `v` is even) that you could move from the current number `v` in one step. Those of the

three that haven't been `seen` yet, mark them as `seen`, make `v` their `parent`, and add them to the end of the `frontier` queue. To make sure that the returned shortest paths are unique (and not just their lengths) and allow automated testing with JUnit, your method **must** examine the three neighbours of `v` in this exact order of `10*v`, `10*v+4` and `v/2` when expanding `v`.

Keep doing this until you discover the goal number `n` during the expansion of some other number `v`. At that point you can construct the path from `n` back to 4 by following the `parent` information that you stored along the way, and then reverse that path from `n` to 4 to be returned as your final answer as the path from 4 to `n`. If the `frontier` becomes empty so that the while-loop terminates before finding the goal number, return the empty list as the artificial negative answer. As revealed by the instructor's private model implementation, the shortest paths from 4 to various goal numbers (with the `limit` incremented iteratively from `100*n` by factor of `2` until the goal was found via some path) are listed in the following table.

`n`	Expected result
`3`	`[4, 2, 24, 12, 6, 3]`
`13`	`[4, 40, 20, 10, 104, 52, 26, 13]`
`42`	`[4, 2, 24, 12, 6, 64, 32, 16, 8, 84, 42]`
`404`	`[4, 40, 404]`
`739`	`[4, 2, 1, 14, 144, 72, 36, 18, 9, 94, 944, 472, 236, 2364, 1182, 11824, 5912, 2956, 1478, 739]`
`740`	`[4, 2, 1, 14, 7, 74, 740]`

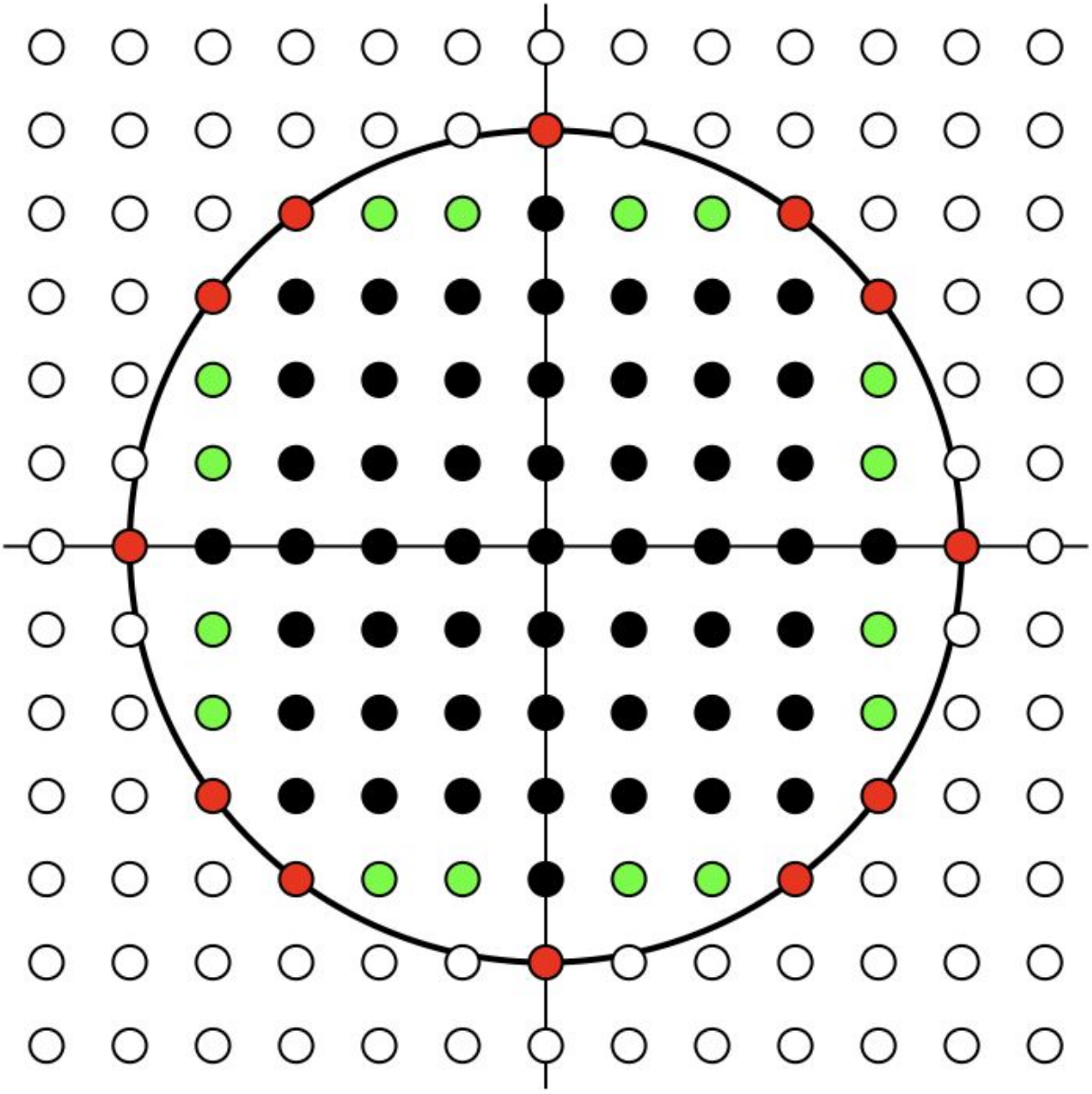
Lab 38: Euclid's Orchard

JUnit: GaussCircleTest.java

As can be easily seen from the Pythagorean theorem, a **disk** with the center at origin (0, 0) and radius r contains all points (x, y) that satisfy the inequality $x^2 + y^2 \leq r^2$. The points on the boundary **circle** make this equation an equality instead of inequality. As we all were taught back in school, the area of this disk is given by the famous formula $\pi\ r^2$. Note the correct terminology in this context in that circle is the one-dimensional boundary curve of the two-dimensional disk. (Next time somebody asks you whether you still remember the formula for the area of a circle, you can be a wise ass and remind them that the area of the *circle* around the disk is always zero...)

The Gauss circle problem asks how many such points (x, y) where both x and y are integers exist inside the disk of radius r. Our lab further breaks that count into three parts by defining an **edge point** to lie exactly on the edge of the disk. A **border point** lies inside the disk but is not on the

edge, and has at least one immediate neighbour $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$ or $(x, y - 1)$ that lies outside the disk. Points inside that are neither edge or border points are **proper** internal points.



As can be seen in the above image, the disk with $r = 5$ contains 12 edge points (red) that correspond exactly to the Pythagorean triples whose hypotenuse equals r. The same disk then contains 16 border points (green) and 53 proper internal points (black).

Create a class `GaussCircle` in your BlueJ labs project, and there the method

```
public static void classifyPoints(int r, long[] out)
```

Note that this method does not return anything, but fills in its three-part answer into the first three elements of the parameter array `out` so that `out[0]` is the number of proper internal points, `out[1]` is the number of border points, and `out[2]` is the number of edge points. This achieves the desired end because both the method and its caller share the same `long[]` object. Your method **must** perform all of its internal computations with the `long` type instead of `int` to guarantee that the squaring of distances does not overflow for large values of `r`. (Even though the value of `r` fits inside an `int`, its square `r*r` might not be as friendly.)

The following table lists a couple of values for r, and the expected numbers of internal, border and edge points. Due to symmetry, the number of edge points is always divisible by four, and the points (*r*, 0), (-*r*, 0), (0, *r*) and (0, -*r*) are edge points for every integer radius *r.* Of course the total number of points will increase with every step of *r*, but the distribution of those points into internal, border and edge points depends on the arithmetic properties of *r*. The three counts of points in the last row add up to 3141592649625, close enough to πr^2 for government work, but slightly overestimating the true value of π because the implicit sum of unit squares centered at all these lattice points will necessarily include some area from outside the original *r*-disk.

`r`	Internal	Border	Edge
`1`	`1`	`0`	`4`
`10`	`261`	`44`	`12`
`24`	`1661`	`128`	`4`
`118`	`43045`	`660`	`4`
`119`	`43797`	`660`	`12`
`12345`	`478705609`	`69820`	`12`
`1000000`	`3141586992773`	`5656800`	`52`

This problem could be solved with two nested loops that loop through all possible combinations of *x* and *y* getting values from -*r* to *r* and classifying each point separately. However, as you can see from the last line of the above table, this method has to work quickly even for large values of r, so you should rather add up these points to the tally one entire row at the time, utilizing the symmetries of the disk for maximal efficiency gain. So let us see how that could be done.

Place your finger on the topmost edge point (0, 5) of the above diagram and start tracing the red and green edge and border points of the upper right quadrant with your finger. Notice how from the current point (*x*, *y*) you always move to whichever is the first available point of the three adjacent points (*x*+1, *y*), (*x*+1, *y*-1) or (*x*, *y*-1) that is not outside the disk. In other words, when tracing the boundary of the upper right quadrant with your finger, you always move east when you

can do so. Otherwise you move southeast if you can do so, and move south only if there is no other choice. This approach is guaranteed to work any radius *r*.

Your method should therefore maintain three counters for the internal, border and edge points, and update these counters appropriately at each step depending on the direction of that step. Once you reach $y = 0$, at which point (heh) necessarily $x = r$, you can terminate the loop. Note that the center row $y = 0$ does not have a symmetric row, unlike the rows $y > 0$ for which the row $-y$ is always symmetric to row y.

This problem would become even trickier if the coordinates of the center point (c_x, c_y) can be arbitrary real values instead of being permanently nailed to the origin. An interesting theorem says that for any positive integer n, there exists some center point (c_x, c_y) and radius r so that the area πr^2 is exactly n, and that disk contains exactly n integer grid points, no more and no less. The Gauss circle problem asks for an exact closed-form combinatorial formula for the number of integer points inside the r-circle without any loops, but this problem still remains unsolved in mathematics. The titular Euclid's Orchard problem asks how many points inside the disk can be seen from the origin so that no other point blocks the view between those points and the origin.

Lab 39: Hot Potato

JUnit: HotPotatoTest.java

A group of n people numbered from zero to n-1 are standing around in a circle. Each person has an "enemies list" of other people in the circle. However, this enemy relationship is not symmetric so that Moe can be an enemy of Joe, even when Joe is not an enemy of Moe. Different people can also have a different number of enemies, but everyone has at least one enemy inside the room, and nobody is their own enemy. For the duration of this process, no new enemies are created or friendships forged inside the room.

All these people are assumed to be some kind of perfect Macchiavellian schemers who hatch all their plans in secret, but do not hold any grudges or desire for "payback" unless it benefits them. They have properly internalized the ideal Markovian view that once the present state of the world is known, the past does not exist in the sense that it could further affect the outcomes of future actions, so the only thing that matters for the future is accepting the present moment as it truly is, and acting in a way that maximizes their future gains in the future.

At time $t = 0$, the person numbered 0 is holding some kind of proverbial hot potato. At each discrete time step t, the person currently holding the hot potato will instinctively toss it randomly to one of his enemies who catches it. That person will then throw the hot potato randomly to one of his enemies at time $t + 1$, and so on. In this lab, you will create a method to investigate the probability distribution of the hot potato at a given time t, given the people and their list of enemies. Create a class `HotPotato` in your BlueJ labs project, and there a method

```
public static Fraction[] hotPotato(int[][] enemies, int tGoal)
```

The *i*:th row of the `enemies` array is a one-dimensional array of integers that lists the enemies of person *i* in ascending order. For example, if `enemies[2]` equals `{0, 5, 6, 9}`, that means that person 2 has four enemies, the persons numbered 0, 5, 6 and 9. Since Java does not actually have two-dimensional arrays, but every `int[][]` is in reality a one-dimensional array whose each element is a one-dimensional `int[]` that represents that row of elements, the rows of the array can be **ragged** so that they do not need to have the same length.

The probability distribution of the position of the hot potato at time *t* is expressed as an *n*-element array of probabilities. Since these probabilities will be integer fractions at all times, we shall again use the `Fraction` class from our class examples to represent these probabilities exactly without any rounding errors, so the type of the array is `Fraction[]`. The individual probabilities must always be between 0 and 1 and together add up to 1, since the potato must be held by exactly one person at any given time. For the initial state $t = 0$, this probability distribution is defined simply by $P(t, 0) = 1$ for the first person and $P(t, p) = 0$ for the other people $p > 0$, since we know that person 0 is initially holding the hot potato.

Given the probabilities $P(t, p)$ at time t as this array, you can compute each $P(t + 1, p)$ the following way. Find all the people p' who are enemies of p (since the enemy relationships never change during the execution of this method, it would be smart to precompute these in the beginning of the method so that you can then just look them up whenever you need them). For each such person p', compute $P(p', t) / m$, where m is the number of enemies of person p'. This term gives the probability of the event "Person p' is holding the potato at time t, and tosses person p." The desired $P(p, t + 1)$ can now be computed by adding up all such terms $P(p', t) / m$, since those are the only possible ways (and mutually exclusive, so that we can simply add up their probabilities) for the person p to receive the potato at time $t + 1$.

The following table again displays a couple of illustrative examples of the `enemies` structure and the resulting probability distribution at the given time *t*. In the situation given in the first row, person 0 will first throw the hot potato randomly to either person 1 or 2, resulting in a distribution where the person 0 cannot be holding the potato, whereas both persons 1 and 2 have the potato with the same probability 1/2. The second row shows the distribution that follows at time $t = 2$. This distribution is made more lopsided than the one expected in the first row by the difference in the tossing behaviour of persons 1 and 2.

n	t	`enemies`	Expected result (as `Fraction[]`)
3	1	`{ {1, 2}, {0, 2}, {1} }`	`{0, 1/2, 1/2}`
3	2	`{ {1, 2}, {0, 2}, {1} }`	`{1/4, 1/2, 1/4}`

4	2	{ {1, 3}, {0, 2, 3}, {0, 1}, {0, 1, 2} }	{1/3, 1/6, 1/3, 1/6}
5	3	{ {1, 2, 3}, {0, 4}, {1, 4}, {2, 4}, {1, 2} }	{1/12, 7/18, 11/36, 1/18, 1/6}
2	1000	{ {1}, {0} }	{1, 0}

Lab 40: Seam Carving

JUnit: SeamCarvingMain.java

The seam carving algorithm for **context-aware image resizing** is well-deservedly famous for the nearly magical outcome that it achieves. Modern image processing algorithms employ arduously trained **deep neural networks** that implicitly identify the semantic content of the image, whereas the following algorithm uses only simple arithmetic and tabulation of results to determine which pixels to eliminate to turn the original image into a smaller image. However, the seam carving algorithm needs none of that, but performs its magic with simple local calculations that need no semantic knowledge of the objects that appear in the image or their mutual ordering of importance in the image.

Instead of rescaling every pixel by the same amount, the scaling is done by repeatedly "carving" a one-pixel wide path through the image, and then erasing exactly those pixels from the image. Each individual carve is constrained to traverse from some top row pixel to some bottom row pixel so that the carve contains exactly one pixel from each row, and moves at most one pixel left or right in each individual step to the next row. Since the path contains exactly one pixel from each row, removing these pixels maintains the proper shape of the image as a rectangle whose height is unchanged and width has decreased by one pixel. The resizing can be performed arbitrarily deep by carving out these individual thin slices one slice at the time.

The details of energy calculation and other details of the seam carving algorithm are well explained with illustrative images in the project pages of various fine universities whose courses used seam carving as a programming project, such as Princeton, Brown and Berkeley. Now here in Ryerson, create a class `SeamCarving` in your BlueJ project window, and in there the method

```
public static Image carve(BufferedImage image, int width)
```

This method should create and return the seam carved version of the `image` horizontally scaled to have the desired `width`.

To allow you to concentrate on the seam carving algorithm itself instead of the nitty gritty details of bitwise arithmetic of colour manipulations, you can use the following method to compute the approximate gradient between two RGB colours packed into four-byte integers that you get from

the `getRGB` method of `BufferedImage`. The higher the gradient between the pixel and its chosen neighbours, the more energy there is in that pixel. (Energy, schmenergy. Everything is just numbers and whatever you choose to call them is your own business.)

```
private static float gradient(int rgb1, int rgb2) {
    int b1 = rgb1 & 0xFF, b2 = rgb2 & 0xFF;
    float b = Math.abs(b2 - b1);
    int g1 = (rgb1 >> 8) & 0xFF, g2 = (rgb2 >> 8) & 0xFF;
    float g = Math.abs(g2 - g1);
    int r1 = (rgb1 >> 16) & 0xFF, r2 = (rgb2 >> 16) & 0xFF;
    float r = Math.abs(r2 - r1);
    return b + g + r;
}
```

You can test your implementation of the algorithm with the SeamCarvingMain class provided in the repository. The outcome of carving the example images `coffee.jpg` and `ilkka.jpg` to half of their original width can be seen in the screenshot below.

The basic seam carving algorithm without further fine-tuning tends to have big problems dealing with sloping lines, as can be seen in the seam carved version of the coffee maker image. However, the important coffee maker remains almost as wide as in the original. For the portrait in the second row, even the most rudimentary version of this algorithm could not possibly fail to find the lowest-energy seams in the nearly uniform white background colour.

Lab 41: Geometry I: Segments

JUnit: CompGeomTestOne.java

At first impression, problems in **geometry** seem difficult to solve on a computer, at least in low-level languages such as Java or Python that do not allow expressions to be analysed in symbolic forms within the homoiconic language itself (no, that term does not refer to Cher in this context). Even worse, algebraic calculations that stem even from simple geometric shapes tend to be chock

full of square roots and other irrational quantities, especially once trigonometric formulas for arbitrary angles become involved. Even before that, the familiar junior high school mathematics notion of how a line is defined by its "slope" and "offset" goes kaput the moment the first vertical line segment appears in your world, since the slope of a vertical line would involve a division by zero. Our frustrated kids are gasping for something better, so it is our duty as parents and teachers to provide them just that.

Since geometric computations cannot be avoided in some applications, most exciting of which are games played in a simulated microworld on some part of the flat two-dimensional plane. These geometric computations become more manageable if restricted on the **lattice points**, that is, coordinates made out of integers. Our geometric objects are composed of straight **line segments** whose both endpoints are constrained to these lattice points. Instead of roots and trigonometric functions, we build our geometry on the vector operations of **cross product** and **signed area**. Even if you don't know what these operations are, no need to worry! Just like you are allowed to use a compiler even though you don't know how it works, or drive a car even if you don't know how a combustion engine works, you are still allowed to enjoy the existence and results of these functions even if you have never taken a vector mathematics course and be able to explain, standing on one foot, exactly what makes these functions tick.

Create a class `CompGeom` in your BlueJ labs project. You will be writing all methods from these computational geometry labs inside this same class, but each lab will have its own JUnit test class that depends only on the methods of that particular lab. To start, copy-paste the following implementations of methods `cross` and `ccw` inside your class exactly as given here:

```
public static int cross(int x0, int y0, int x1, int y1) {
    return x0 * y1 - x1 * y0;
}

public static int ccw(int x0, int y0, int x1, int y1, int x2, int y2) {
    return cross(x1 - x0, y1 - y0, x2 - x0, y2 - y0);
}
```

Without worrying about what these functions actually do and how they do it, you should now put on your Java programmer hat and spend a moment in silent appreciation of how these both methods use only basic arithmetic, and are well-defined for all possible argument values. (The JUnit tests are designed so that your code does not need to worry about effects of integer overflow.) Most importantly, **neither operation performs any divisions**, so we don't need to worry about dividing by zero the way people often end up doing in their slope calculations.

Analogous to the bumper sticker version of Liskov Substitution Principle that your instructor keeps parroting every chance he gets, you should also follow an analogous bumper sticker version of computational geometry. **The moment that any one of the words "angle", "slope" or "root"**

comes out in your thinking about your method to solve some computational geometry problem, that means that your method is automatically wrong!

Now that we go that out of the way, let us explain what the above two methods actually do. The `cross` method computes the **two-dimensional cross product** of two vectors (x_0, y_0) and (x_1, y_1). (Point (x, y) on the two-dimensional plane can be thought of as a vector from the origin (0, 0) to the point (x, y).) The cross product of two vectors is actually a three-dimensional operation that produces a three-dimensional vector result instead of just one scalar number, but since any two-dimensional vector (x, y) on the plane can be thought of degenerate three-dimensional vector $(x, y, 0)$, multiplications by these zeroes eliminate most terms. Only the z-component of the result needs to actually be computed with the remaining formula $x_0y_1 - x_1y_0$, and can be returned as a scalar instead of a redundant three-element vector with x- and y-components being zeros.

The cross product returns the **signed area** of the **parallelogram** defined by the vectors (x_0, y_0) and (x_1, y_1). Swapping the order of the arguments therefore negates the sign of the area. Since the signed area formula features only addition, subtraction and multiplication, it can be applied to integers, rational numbers and real numbers. Here, the real important feature of the signed area formula is the guarantee to produce an integer result when applied to integer arguments.

The concept of a negative area might first seem confusing, but it has an important purpose in the operation that in most online material is succinctly known as `ccw`, so we shall also follow that naming here. (In fact, just googling for "`ccw computational geometry`" gives you several good pages and sets of slides that illustrate how this operation can cut through seemingly complex geometry problems like a chainsaw cuts through butter!) This operation allows us to implement all methods from these four labs with absolute accuracy without any rounding errors or singularities. Among the subfields of computer science, computational geometry is especially notorious for how its problems have all kinds of tricky edge and corner (heh) cases that naive implementations of these algorithms have trouble dealing with, especially if implemented with floating point arithmetic in a happy-go-lucky attitude.

The name `ccw` is short for "counterclockwise". Given three points (x_0, y_0), (x_1, y_1) and (x_2, y_2) on the plane, the sign of the signed area of the triangle from (x_0, y_0) to (x_1, y_1) to (x_2, y_2) tells you the orientation of these three points with respect to each other.

- If the method `ccw` returns any **positive** number, these three points are oriented **counterclockwise**, so the turn in the middle when moving from (x_0, y_0) to (x_1, y_1) to (x_2, y_2) is **left-handed**.
- If the method `ccw` returns any **negative** number, these three points are oriented **clockwise**, so the turn in the middle when moving from (x_0, y_0) to (x_1, y_1) to (x_2, y_2) is **right-handed**.
- If the method `ccw` returns zero, the three points (x_0, y_0) to (x_1, y_1) to (x_2, y_2) are **collinear**.

In practice, all the problematic edge and corner cases in the computational geometric algorithms in these labs emerge in situations where three points are collinear, even overlapping. (Frankly, it is

not surprising that most word problems in geometry of points and line segments start with the disclaimer of "assume the general position" so that all points are distinct and no three points are collinear...)

And that's it! The method `ccw` is the fundamental building block of all computational geometry operations that we shall now implement. In fact, if you remember how the method `compareTo` for custom order comparison returns its result as an integer whose sign contains the entire answer and the magnitude of the returned number is meaningless, the `ccw` operation is the two-dimensional generalization of the one-dimensional order comparison operator! (In fact, checking whether $a > b$ in the one-dimensional line reduces to checking whether the two-dimensional turn defined by the three points $(a, 1)$, $(a, 0)$, $(b, 0)$ is left-handed.)

To practice using this wonderful `ccw` operation and appreciate it in action, here is finally your first method to write in this lab, in a new class named `CompGeom`.

```
public static int lineWithMostPoints(int[] xs, int[] ys)
```

For simplicity, a set of points on a two-dimensional plane is given to these methods as two arrays `xs` and `ys` (read these two names out as "exes" and "whys" so that the letter *s* denotes plural form) so that `xs` contains the *x*-coordinates of these points, and `ys` contains the *y*-coordinates. For example, the *x*- and *y*-coordinates of the point (x_4, y_4) can be found in the elements `xs[4]` and `ys[4]`. These points have not been otherwise sorted in any manner but can be given to your methods in any order.

Given a set of n points on the plane, find the line that contains the largest number of points from that set. Since there can exist several such lines for the given set of points, this method returns only the count of how many points are on that line, to make automated testing feasible.

Any two distinct points determine the (unique) line that goes through those two points. The `ccw` operation then tells you quickly whether some third point also lies on that same line. You can therefore solve this problem with two nested loops that examine all possible pairs of points (and therefore all possible lines defined by such pairs of points), with a third level of innermost loop finding all the other points in the same line. However, since the **cubic** running time of $O(n^3)$ caused by three levels of nested loops can be a bit heavy, you should do whatever you can to eliminate redundant comparisons to speed up the execution of this method. The details of this are left as an exercise for the reader.

With that out of the way, it is time to implement an important operation that will be used as building block for many other computational geometry algorithms: determine whether the line segment with endpoints (x_0, y_0) and (x_1, y_1) intersects another line segment with endpoints (x_2, y_2) and (x_3, y_3), that is, whether these two line segments have at least one point in common.

```
public static boolean segmentIntersect(
```

```
  int x0, int y0, int x1, int y1, int x2, int y2, int x3, int y3
)
```

(One of the famous programming epigrams by Alan Perlis points out that if your function takes ten parameters, you probably forgot some. We are two short of that, but the point is still valid.)

Once again: no angles, no slopes, no trig, no nothing that could ever produce anything that is not an exact integer, since we should solve this important problem with absolute accuracy even in all possible corner cases with the following approach. Start with a simple **quick rejection test** by comparing the **bounding boxes** of these line segments. Bounding box comparison is the basic computational geometry technique to speed up all intersection tests between complex objects, but it works just as well here with simple line segments. If the higher of the *x*-endpoints of either segment is strictly less than the lower of the *x*-endpoints of the other segment either way, these two segments cannot possibly intersect. The same test is also applied to y-coordinates of the endpoints. Most of the time in practice, the "no" answer comes out from these quick rejection tests, since most line segments on the plane are not anywhere near most other line segments.

The method moves to stage two only if the bounding boxes intersect. Then, the segment with endpoints (x_0, y_0) and (x_1, y_1) intersects the segment with endpoints (x_2, y_2) and (x_3, y_3) if the turns (x_0, y_0)-(x_1, y_1)-(x_2, y_2) and (x_0, y_0)-(x_1, y_1)-(x_3, y_3) do not have the same nonzero handedness. In other words, the points (x_2, y_2) and (x_3, y_3) are not on the same side of the line segment (x_0, y_0) and (x_1, y_1), but the line segment from (x_2, y_2) to (x_3, y_3) actually crosses that line so that an intersection can occur. This test also has to be repeated with the roles of two segments interchanged, since the segments must cross each other both ways.

This logic will take care of the intersection test between two arbitrary line segments, even in the degenerate case where one of the segments is a single point. To find all intersections within a set of n such line segments, you could of course loop through all $n(n - 1)/2$ pairs of segments, which would become a prohibitively inefficient "Shlemiel" algorithm as n gets large. Assuming that the segments are spread reasonably around the plane, a sweep line algorithm similar to the one that solved the earlier Manhattan skyline problem will do the job in expected linear time.

Lab 42: Geometry II: Polygons

JUnit: CompGeomTestTwo.java

The first lab of computational geometry introduced the `cross` product and `ccw` operations as the fundamental building blocks of our geometric algorithms. In this lab we move on from sets of n points to ordered lists of n points to define an n-sided polygon whose edges are the line segments between consecutive points in this list. The line segment from the last point in the list back to the first point closes the polygon edge. Depending on the needs of the current situation, we think of the polygon being made up of "edge segments" or made up of "corner points", but that is just a different

"point" of view (heh again, but it is probably not a coincidence that our everyday language is surprisingly rife with geometric metaphors) to the same integer data.

The line segments determined by an arbitrary list of points do not necessarily look like a "polygon" as we generally understand this term. For our purposes, a list of points defines a simple polygon if it satisfies the following two constraints:

1. No three consecutive corner points are collinear.
2. No two non-consecutive edge segments intersect, not even in a corner point.

Of course every two consecutive edge segments always meet in their common corner point so that there are no holes in the boundary of the polygon, but no other edge segment intersections exist except the *n* intersections at the corner points. The `ccw` and `segmentIntersect` methods from the previous lab should make short work for writing the following method:

```
public static boolean isSimplePolygon(int[] xs, int[] ys)
```

This method determines whether the list of points whose coordinates are given in two *n*-element arrays `xs` and `ys` form a simple polygon when their points are read in that order.

Technically, the first condition of non-collinearity of three consecutive corner points is not necessary for a polygon to be "simple" as that concept is generally understood. However, eliminating redundant corner points guarantees that every corner point is a literal "corner" so that a sudden discontinuous turn takes place in that point. (You could always modify this method to take an additional `boolean` parameter to indicate if the collinearity rule is in effect.)

Furthermore, note how any **cyclic shift** of these corner points changes nothing, but the shifted points still represent the exact same simple polygon as a geometric object. The same holds for reversing the list of corner points, but some material on computational geometry imposes an additional condition that the polygon corner points must be listed in the order that follows the boundary of the polygon in counterclockwise direction. Such guarantee would make some later operations on polygons easier to implement. (We have encountered this principle of unique canonical representation for objects a few times before; for example, in the `Fraction` class.)

We noted earlier how only the sign of the result of `ccw` has an effect on our decisions, and the magnitude of the result is irrelevant. The same way as our one-dimensional sorting algorithms only compare elements for order and never use subtraction to find out the absolute difference between elements, **topological** algorithms whose results are unaffected by arbitrary scaling, shifting and rotating of these points, should never need the actual magnitude of the result for anything. However, this magnitude does have a meaning that can be put to good use in another important problem, calculating the **area** of the given polygon.

Recall from the previous lab that the cross product of two vectors is the signed area of the parallelogram defined by those vectors. Therefore, the signed area of the triangle defined by three corner points (x_0, y_0), (x_1, y_1) and (x_2, y_2) is exactly half of `ccw(x0, y0, x1, y1, x2, y2)`. This leads to the immediate observation that **the area of a triangle made of lattice points can only be some integer or some integer plus one half**, regardless of how these three points lie on the plane, and whatever square roots their side lengths and other irrational numbers their angles and their trigonometric quantities happen to be.

Since any simple polygon can be broken down into disjoint triangles sharing edges and corner points (same as with many other theorems of computational geometry that somehow feel to our intuition as if they were "obvious", this result is far more complex than it might initially seem), the same immediately follows for areas of arbitrary simple polygons. Triangulation would then be one way to compute the area of any simple polygon. However, we don't actually need to perform any such triangulation, since we can compute the area in linear time with the shoelace formula, courtesy of the giant brain of Carl Friedrich Gauss himself.

```
public static int shoelaceArea(int[] xs, int[] ys)
```

Since the area of a lattice polygon can only ever be some integer or some integer plus one half, this method should return the area of the polygon multiplied by two, so that the result is always an integer. The caller can always divide the result by two to get the real area, but this way we can return 30 if the actual area is 15, and return 31 when the actual area is 15 1/2, so we can tell these situations apart while using only integer arithmetic.

(This idea of scaling fractional values to be integers generalizes for other computations that involve fractional quantities. For example, we can sometimes represent money as cents instead of dollars and not have to deal with the fractional parts that often don't have an exact representation in floating point anyway. Always remember to be careful with integer overflows.)

Back to the shoelace algorithm. Choose any reference point from the plane. It wouldn't even matter which point you choose, or whether that point lies inside or outside the polygon (coincidentally, the topic for our next lab after this). Given such unprecedented freedom, we might as well choose the origin (0, 0) as this reference point to simplify our calculations, but any other point would have worked just as well. Initialize the integer `area` variable to zero. Loop through each consecutive edge segment of the polygon with corner points (x_i, y_i) and (x_{i+1}, y_{i+1}), and add the result of `ccw` of the reference point and those two points to your `area`. After this one loop, return the final value of the `area` as your answer, wham, bam, thank you ma'am.

Honestly, it really is that simple. Since the `ccw` function returns a signed area whose sign depends on the direction of the edge segment relative to the reference point, the additional areas outside the polygon that get added to the `area` during the execution of this algorithm will be subtracted on the way back around the other side of the polygon. Regardless of how the polygon points lie and whichever way the polygon boundary might meander back and forth through the plane, these

positive and negative areas outside the polygon will always cancel each other out exactly. Only the area inside the polygon will not get cancelled that way, so the result is equal to the polygon area.

Lab 43: Geometry III: Points In Polygons

JUnit: CompGeomTestThree.java

Intersection and collision tests (the former in **static** worlds, the latter in **dynamic** worlds) are important in many games that take place inside some simulated two-dimensional world. This lab gets the ball rolling here with the simplest possible such intersection test of determining whether the given point is inside or outside the polygon. More advanced algorithms exist for intersections and collision between polygons, disks and other geometric objects.

A simple polygon is said to be **convex** if for any two points (x, y) and (x', y') inside that polygon, the entire line segment between these two points also lies fully inside the polygon. Intuitively, a convex polygon has no "pits" on its edges. Convex polygons are important in computational geometry, since many problems on polygons suddenly become simpler if the polygons involved are convex. Every **triangle** is automatically convex out of the box (the same does not hold for quadrilaterals of four points), which is one major reason why triangles and **triangle meshes** tend to be the fundamental shape primitive used in computer graphics in general.

During the traversal of the boundary of the convex polygon, every turn at every corner necessarily has the same handedness so that either every turn is left-handed, or every turn is right-handed, but both kinds of turns cannot exist on the boundary of the convex polygon. Under the additional voluntary constraint of counterclockwise listing of corner points, only left-handed turns can exist. These observations give us an immediate linear time algorithm to determine whether the given polygon is convex. The same idea can be used to determine whether the given point (x, y) lies inside the given convex polygon.

```
public static int pointInConvex(int[] xs, int[] ys, int x, int y)
```

The arrays `xs` and `ys` are guaranteed to be corner points of some convex polygon, and in this problem additionally guaranteed to be listed in counterclockwise direction. Instead of returning a mere yes/no answer as a `boolean`, it turns out to be more useful for our later purposes to define this method to make the distinction between different ways for the point to be inside the polygon. This method should return the answer as integer from zero to three, inclusive, so that the result of zero means that the point (x, y) is outside the polygon. Result of one means that the point (x, y) is one of the corner points. Result of two means that the point (x, y) lies on some edge segment (but not in any corner), and the result of three means that (x, y) lies properly inside the polygon.

As the number of corners n increases, the caverns and tendrils of that polygon can sprout into almost fractal-like complexity, so the question of point containment is not quite as trivial as it is for

the convex polygons. The first idea, same way as in the earlier problem of computing the area of a polygon, might be to break the polygon into disjoint triangles, and then test for point containment in these triangles with the previous method. If some triangulation is already available, there certainly is no harm in using it. Otherwise solving this problem via triangulation would be overpaying for overkill, since a far simpler algorithm can already solve this problem for arbitrary polygons in linear time around the edge segments!

```
public static int pointInPolygon(int[] xs, int[] ys, int x, int y)
```

Same as the previous method for point inside convex polygon, this method should return the answer zero for outside, one for corner, two for edge and three for inside. However, this method must work for any polygon, even those that are not convex or not even simple; the JUnit test will generate polygons that intersect themselves and degenerate to line segments and points. The first stage should be the loop around the polygon corner points to determine whether (x, y) is one of the corners, or is lying on one of the edge segments. Past that hurdle, the rest of the algorithm can safely assume that no edge segment of the polygon intersects the point (x, y) itself.

To visualize the ray casting technique used to determine whether the point (x, y) is inside the polygon, imagine standing on that point and then start walking along some straight **ray** that shoots from its starting point into some arbitrary direction. (Technically, a line segment has two endpoints, a ray has one, and a line has none. Most people erroneously say "line" when they "ackshually" mean "line segment", but this is a long lost linguistic hill for anybody to still die on.) Since we can choose this direction freely, we might as well choose one of the main axis directions where one coordinate is zero to simplify the formulas. Any one of the four compass directions would result in a symmetric method, but since we have to pick one, let us follow the usual convention and pick the ray shooting right towards infinity, eternally in parallel with the x-axis.

Since we don't have the math to deal with rays (this could be done with parameterized curves, but those quickly end up needing non-integer arithmetic), we realize that it is enough to look at the line segment that starts from point (x, y) and continues to the right just far enough so that its endpoint lies outside the polygon. Any x-coordinate larger than the maximum x-coordinate among the corner points of the polygon will do fine. Since the end position $(\max x_i + 1, y)$ of our rightward ray (there is another good name for an indie math rock band, for any nerdy hipsters with musical aspirations now reading this) is known to lie outside the polygon, and every edge crossing along the segment moves either from inside to outside, or from outside to inside. The parity of the number of polygon edge crossings from the starting point (x, y) to the end position $(\max x_i + 1, y)$ tells us the answer right away! Any odd number of crossings means that the starting point (x, y) was inside the polygon, whereas any even number means that it was outside.

A trivial modification of the previous "even-odd algorithm" to use the "nonzero winding rule" makes this algorithm even more general so that it can handle even non-simple polygons whose edges can freely intersect each other. In such a polygon, the issue of what the words "inside" and

"outside" even mean can become as thorny and frustrating as trying to separate the inside and outside of a Klein bottle, but the nonzero winding rule still handles such situations correctly.

Initialize the local `count` of crossings to zero. Whenever the ray segment intersects the polygon edge segment between two polygon corner points (x_i, y_i) and (x_{i+1}, y_{i+1}), the sign of a `ccw` check from point (x, y) to these two points tells us which direction we are crossing that edge. If the sign is negative, decrement the `count` by one, and if the sign is positive, increment the `count` by one. The point (x, y) is inside the polygon if and only if the final `count` is nonzero. For simple polygons, this rule gives the same answer as the previous even-odd rule, but for self-intersecting polygons, the results will be different so that the polygon has "holes" inside it that are not connected to the world outside the polygon. **Your `pointInPolygon` method should use the nonzero winding rule for counting the intersections, instead of the even-odd rule.**

Before anybody starts implementing this algorithm by actually traversing through that imaginary ray in their code, we must emphasize that the idea of "walking" along this ray is only an aid for your intuition. The actual algorithm should loop through the edge segments of the polygon in one linear-time for-loop. For each edge segment, check whether it intersects the ray segment between the points (x, y) and $(\max x_i + 1, y)$. We can do this with the `segmentIntersect` method from the first geometry lab.

However, as is typical for algorithms of computational geometry, there is a difficult edge case lurking in the grass to trap the unwary. Using concrete numbers to illustrate the danger, let (2, 3) be the point (x, y) whose containment inside the polygon we wish to find out, and let the polygon has two edge segments that connect the three consecutive corner points (5, 1), (7, 3) and (7, 5). The ray that we shoot right from the starting point (2, 3) will therefore intersect both of these line segments in their shared corner point (7, 3). The algorithm that looks at each edge segment separately will therefore count two edge crossings in that one corner point, which makes the parity of the final count to be the exact opposite of the correct answer!

In moments like this, it is good to step back for a moment. Many coders would start "fixing" their code by adding tests for special cases instead of solving the general problem for all cases in one swoop. **However, any such fix that talks about some special direction is automatically wrong, even if it solves the issue for that particular test case.** Since the result of any topological problem is unchanged by arbitrary planar rotation, a bug fix in any such problem can never depend on any particular "special" point or direction. The fix has to be more general so that it still works even if all points of the original problem are arbitrarily mirrored, rotated, translated or subjected to whatever wacky transformation, as long as that transformation does not affect the topological properties of the original set of points.

The correct fix is actually pretty clever in its simplicity once we realize it. Before applying the intersection test, we should do a quick rejection test to eliminate all polygon edge segments that cannot possibly intersect our rightward ray from the point (x, y). If both endpoints of the edge segment have a y-coordinate lower than our y, the segment lies fully below the ray and no

intersection can exist. The same happens when both endpoints have a *y*-coordinate higher than *y*. To fix the previous bug, **just treat the *y*-coordinate of the segment endpoint that is equal to *y* as if it were actually lower than *y* for the purposes of this quick rejection test.**

Using the previous numbers, the edge segment from (5, 1) to (7, 3) is now rejected as lying fully below the rightward ray from the starting point (2, 3), and therefore that segment does not increment the crossing count despite the fact that it technically intersects the ray at the corner point (7, 3). However, the next edge segment from (7, 3) to (7, 5) is not quick-rejected; the point (7, 3) is considered to lie below the ray whereas the other endpoint (7, 5) lies above it. That edge segment increases the crossing count by one, as it properly should.

Lab 44: Geometry IV: Convex Hull

JUnit: CompGeomTestFour.java

As we have seen in the previous three labs on computational geometry, a list of points can be thought of as just these points existing independently as points on the two-dimensional plane, or as the *n* corner points of a polygon so that the consecutive corner points are implicitly connected by the edge segments of that polygon. To conclude our discussion of basic computational geometry, we will tackle the problem of building a polygon out of the list of points that can be given in arbitrary order so that the resulting polygon is simple. In general, there exist exponentially many different simple polygons that could be built from the given set of distinct points. However, we might as well try to build a polygon that has some kind of useful properties for the rest of our program.

This fourth geometry lab is also a good place to introduce a technique of **indirection** that often comes handy not just in computational geometry, but in many other problems in data structures and algorithms. When the same points are shared between multiple polygons or other geometric objects (for example, when constructing a triangle mesh to represent a **polyhedron** surface where several triangles meet at their shared corner points), the ability to move these points without breaking the connections between these triangles would be highly desirable.

If every triangle is an island that knows only the absolute coordinates of its own corner points but not even about the existence of other triangles that share that point, moving these corner points will affect only that triangle, but not any other triangles. To avoid creating ugly cracks on the polygon mesh, we would have to somehow find all other triangles that use the same corner point, and perform the same update to that corner point in each triangle separately. To avoid this hassle, maintain the set of all relevant points separately, let's say for simplicity as the two arrays `xs` and `ys` we have already seen before. A polygon of *n* such points is encoded as an *n*-element array of integer indices to the arrays `xs` and `ys` that contain the actual point coordinates.

For example, a triangle encoded as the array `{42, 17, 99}` would consist of the corner points whose actual coordinates are stored in the positions 42, 17 and 99 of the arrays `xs` and `ys` for

lookup. These arrays `xs` and `ys` could potentially contain millions of other points, most of which are not part of this triangle. Another triangle encoded as the array `{42, 43, 77}` shares the corner point 42 with the previous triangle. Adjusting the coordinates of the point number 42 will then automatically adjust every triangle that uses that same corner point, without us even needing to do any extra updating at all! (In some problems, additional computation may be necessary to maintain additional constraints that prevent the mesh from becoming degenerate.)

Enough chit chat. Given a set of points as two arrays `xs` and `ys`, the first method in this week's lab will construct **a fan polygon** by sorting the given set of points in counterclockwise fashion, when viewed from the reference point with the lowest y-coordinate. If the set contains several points with the same minimum *y*-coordinate, use the point with the lowest *x*-coordinate among those points as the reference point. Note that neither method in this lab should change the contents of the `xs` and `ys` arrays given to them in any way, but create and return a new array of position indices that lists these points in the order in which they constitute the polygon corners.

```
public static int[] sortCCW(int[] xs, int[] ys)
```

Given a set of *n* points with coordinates given in two arrays `xs` and `ys`, this method should first create an *n*-element array `Integer[] result`, and fill it with the numbers 0, ..., *n* - 1. Next, define a local class for a `Comparator<Integer>` strategy objects that will be used to sort the array of indices.

```
class MyComp implements Comparator<Integer> {
    public int compare(Integer i, Integer j) {
      // Fill in this method body
    }
}
```

The method `compare` inside this local class should compare the two indices `i` and `j` according to the following ordering rules between indices:

1. The index of the reference point is always smaller than any other index.
2. If the turn from the reference point through point `i` into point `j` is left-handed, then `i` is considered lower than `j`.
3. If the turn from the reference point through point `i` into point `j` is right-handed, then `i` is considered higher than `j`.
4. If the reference point is collinear with the points `i` and `j`, whichever of the points `i` and `j` has a lower *y*-coordinate is considered lower. If both points have the same y-coordinate (another possible edge case that is not immediately obvious but can happen), the point with the lower x-coordinate is considered lower.

These rules allow the `compare` method to always return either `-1` or `+1` for any two distinct indices `i` and `j`, and the resulting ordering is total. Armed with this comparator, the `sort` method in the `Arrays` utility class can take care of all the hard work involved in the actual sorting:

```
Arrays.sort(result, new MyComp());
```

All that is left for you to do is to fulfill the letter of the law by creating an `int[]` that is going to contain the actual result, and copy the contents of the `Integer[]` into that array of the promised return type before returning it.

The fan polygon is not necessarily simple, since it can contain degenerate line segments without an area. This will happen, for example, with the points (0, 0), (1, 0), (2, 0), (2, 1), (1, 1), (2, 2) that become a fan polygon in this order, with the origin (0, 0) as the lowest point that becomes the reference point. The rest of the points end up listed in the order of increasing angl... aw shucks, pardon my French, as we were never supposed to use that bad word coined by Old Nick himself to lead us astray from the `true` path. Let's try again. The rest of the points end up sorted in counterclockwise order relative to the reference point in the lower left, which explains the name `sortCCW` of this method.

Since the boundary of the result polygon can contain both left and right turns, the fan polygon is not necessarily convex. It is guaranteed to be a star-shaped polygon, though, meaning that if that polygon is interpreted as the walls of a room, there exists at least one point you can stand on and be able "see" every other point inside that room by turning towards it. (The "star" shape of such a polygon can therefore be something that we would not really consider a cartoon blammo star, but all these names are made up anyway to bolster our intuitive grasp of the abstract subject and this way enlist the parts of our brain that deal with spatial intuition to aid in understanding such concepts.) Convex polygons are star-shaped polygons where every point works as that standing point. Since the constructed polygon contains all of the original points, it will also have redundant corner points in places where three consecutive points are collinear.

A fan polygon can be swiftly converted into a convex polygon with several different algorithms, of which we shall implement the one known as Graham scan. Many other algorithms also exist to find the convex hull of the given set of points. In fact, the convex hull problem is a two-dimensional generalization of one-dimensional **comparison sorting** problem, a fact humorously highlighted by how the convex hull algorithms "quickhull" and "mergehull" are pretty straightforward two-dimensional generalizations of the classic quicksort and mergesort algorithms for one-dimensional comparison sorting. (To sort a series of integers x_0, ..., x_n, simply construct the convex hull of the set of points (x_i, x_i^2) and read the answer in order around the hull. This reduction establishes that finding the convex hull cannot be performed faster within the same model of computation than one-dimensional sorting of numbers.)

In this lab, we choose to specifically implement Graham scan because the previous `sortCCW` method already did half of its work. Besides, we get to finally use a **last-in-first-out stack** in the

implementation of this algorithm, which makes this a more generally useful exercise in programming. Perhaps due to their deceptive simplicity, LIFO stacks are not always fully appreciated during the first and second year computer science curriculum, whereas queues and priority queues tend to find more use in algorithms during that time.

```
public static int[] grahamScan(int[] xs, int[] ys)
```

This method should start by calling the previous method `sortCCW` to acquire the fan polygon listing of the set of points given in the parameter arrays `xs` and `ys`. It should then create a stack (you can use the actual `Stack<Integer>` in the `java.util` package, or simply simulate a stack with an `int[]` and an index variable canonically named `top` that tells the current size of the stack) and push the first two indices of the fan polygon in this stack. The rest of the algorithm should then loop through the rest of the points `i` of the fan polygon and obey the following rules of what to do for each such point:

1. If the stack contains only one element, or if the turn from the current top two elements of the stack to the point `i` is left-handed, push `i` into the stack.
2. While the turn from the current top two elements to the point `i` is collinear or right-handed, pop the topmost element of the stack.

Note that the second rule must be a nested while-loop inside the outer for-loop through the fan polygon points, because in principle there is no upper limit on how many points the new point can theoretically eliminate from the stack. You might want to watch some animation of Graham scan in action to see for yourself why and how this happens. (Relaxing afterwards, you can also ponder why this algorithm works in guaranteed linear time despite the fact that it consists of two levels of nested loops, since the answer to this question generalizes all over the study of algorithms.) Once the outer for-loop has finished processing all corner points of the fan polygon, the stack contains precisely the points of the convex hull, so it should be trivial to construct the array that is returned as the result.

Each of the four labs of computational geometry has a separate JUnit test class to test only the methods of that particular lab. However, as an immaterial reward for all the students who complete all four geometry labs, check out the method `verifyPicksTheorem` in this fourth lab's JUnit test class. This pseudorandom fuzz testing method creates a bunch of sets of random points on the plane, and uses Graham scan to compute the convex hull polygon. The area of this polygon is then computed with the shoelace method, but also with an implementation of Pick's Theorem that computes the area of a simple lattice polygon as a function of how many lattice points fall inside that polygon and on its edges. Once both of these methods for area calculation return the same area for a large number of randomly generated convex hull polygons, our faith on the correctness of these methods is greatly increased.

Determining which points are inside and on the edges of the convex hull works as a mass test to verify that `pointInConvex` and `pointInPolygon` methods always return the same answer.

Whenever two totally different algorithms agree on all answers for the large number of randomly generated arguments given to these algorithms, our belief in the correctness of both algorithms is massively increased. Philosophically minded students should ask themselves why this is the case.

Lab 45: Permutations I: Algebraic Operations

JUnit: PermutationsTestOne.java

In this troika of lab problems about permutations, integer arrays of *n* elements that contain each natural number from 0 to *n* - 1 exactly once. All methods in these three labs that operate on permutations may assume that the arguments generated by the JUnit fuzz tests satisfy this definition. Since we are computer programmers, we start counting the elements and positions from zero, whereas real mathematicians prefer to define permutations using numbers 1, ..., *n*, which we need to keep in mind if reading about the theory of permutations.

Even though permutations are represented as arrays of integers, a permutation can conceptually be thought of as a "function" that can be applied to any array of *n* elements. This "function" copies each original element exactly once into the result, to the position given by the permutation array. For example, if `perm` is a permutation so that `perm[4] = 9`, applying it to an array `arr` where `arr[4] == "Bob"` will have that value `"Bob"` appearing in the position 9 in the result array.

Since permutations are "functions" that can be applied to arrays of *n* elements, while also being arrays of *n* elements themselves, permutations can be applied to permutations to create new permutations. Analogous to matrix multiplication in linear algebra where the product of any number of *n*-by-*n* square matrices is also an *n*-by-*n* square matrix, the combination of any number of permutations on *n* elements is itself a permutation of *n* elements. Whoa, as that Keanu fellow might say realizing this. (In fact, permutations can be implemented as binary permutation matrices so that each row and each column contains exactly one element that equals 1, so that known truths of linear algebra can be applied to the special case of permutations.)

Enough theory, let's see some action. Create a class `Permutations` in your BlueJ labs project window. You will be writing all methods in this set of three labs into this class. The methods in this first lab deal with the **algebra** of combining permutations to create new permutations.

```
public static int[] chain(int[] p1, int[] p2)
```

Combine the two permutations `p1` and `p2`, assumed to be of the same size, into a result permutation `p` that represents the operation of first applying permutation `p2` to the given array, followed by applying permutation p1 to this intermediate result. For example, if `p2[3] == 4` and `p1[4] == 9`, then `p[3] == 9` moving the element right there instead of taking it on a detour through the position 4 in between.

```
public static int[] inverse(int[] perm)
```

Given the permutation `perm`, construct its **inverse permutation** that, when combined with `perm`, produces the **identity permutation** for which `p[i] == i` for all positions `i`. For example, when called with a permutation array `{2, 1, 4, 0, 5, 3}`, this method would create and return the permutation array `{3, 1, 0, 5, 2, 4}`.

```
public static int[] square(int[] perm)
```

The square of the permutation is constructed by combining it with itself. This method should not be too difficult once you have written the method `chain`, a mere one-liner.

```
public static int[] power(int[] perm, int k)
```

Compute the *k*:th power of the given permutation `perm`, that is, the result of chaining `perm` with itself *k* times. For negative values of *k*, this method should return the -*k*:th power of the inverse of `perm`. With the rule that $k = 0$ always gives the identity permutation, this operation is well-defined for arbitrary integer powers *k*, both positive and negative.

Since *k* will get pretty large in the JUnit tests (permutations play nicer than integer matrices since the element values never change so there is no possibility of overflow), you should use the technique of exponentiation by squaring to compute the result much sooner for large *k* than you would get by chaining `perm` with itself *k* - 1 times. The base cases of this recursion are $k = 0$ where the result is the identity permutation, $k = 1$ where the result is just `perm`, and $k = 2$ where the result is the square of `perm`. For higher values of *k*, apply the algebraic identities $p^{2k} = (p^2)^k$ and $p^{2k+1} = p(p^2)^k$ that cut the exponent in half in each step, which ought to make this method blazingly fast even for *k* not just in the millions but even in the vigintillions, were the range of our integers that stratospheric.

`perm`	`k`	Expected result
`{1, 0, 2, 3}`	`4`	`{0, 1, 2, 3}`
`{1, 3, 6, 0, 5, 2, 4, 7}`	`98`	`{3, 0, 4, 1, 2, 6, 5, 7}`
`{0, 6, 3, 4, 8, 7, 2, 9, 5, 1}`	`-397`	`{0, 9, 6, 2, 3, 8, 1, 5, 4, 7}`

Students who aim to take math courses and especially a course on abstract algebra may note that permutations and their operations `chain` and `inverse` define an algebraic group, with the method `chain` standing for $p_1 + p_2$ and `inverse` standing for $-p$ in this group. Once it has been established that permutations form a group, all algebraic results and algorithms such as exponentiation by squaring work also for permutations out of the box. (Besides, permutation groups are by themselves pretty big shots in group theory, thanks to Cayley's theorem.)

Lab 46: Permutations II: Cycles

JUnit: PermutationsTestTwo.java

Continuing the previous lab, a permutation is an array of positions that describes where each element moves to when that permutation is applied to some array. The *n*-element array therefore already fully contains all information about that permutation. However, to allow easier computation of some interesting properties of the behaviour of this permutation, it is often handy to break down the permutation into disjoint cycles. To find the cycle of `perm` that the position `i` is part of, start a loop from position `i` and keep moving to the current position `j` into `perm[j]`, until you return to the starting position `i` having met all the other members of that cycle along the way. It is also possible that `perm[i] == i`, making that cycle a **singleton**.

```
public static List<List<Integer>> toCycles(int[] perm)
```

Since every cycle in `perm` is a list of integers, the list of such cycles must therefore be a list of lists of integers. This method should compute and return the **canonical cycle representation** of `perm` as disjoint cycles so that each individual cycle is listed starting from its largest element, and the cycles are listed in the ascending order of their first elements. The following table showcases the expected result for some random permutations for various *n*. As the second and third row of this table demonstrate, a permutation can consist of a single cycle that all these elements ride like some comically large tandem bike, or in the other extreme, every position rides its own unicycle.

`perm`	Expected result
`{1, 0, 2}`	`[[1, 0], [2]]`
`{2, 4, 1, 0, 3}`	`[[4, 3, 0, 2, 1]]`
`{0, 1, 2, 3, 4}`	`[[0], [1], [2], [3], [4], [5]]`
`{0, 5, 2, 3, 4, 1}`	`[[0], [2], [3], [4], [5, 1]]`
`{4, 6, 1, 5, 3, 0, 2}`	`[[5, 0, 4, 3], [6, 2, 1]]`
`{12, 7, 13, 1, 2, 5, 11, 0, 10, 14, 3, 6, 9, 4, 8}`	`[[5], [11, 6], [13, 4, 2], [14, 8, 10, 3, 1, 7, 0, 12, 9]]`

Since the permutations and their canonical cycle representations are one-to-one, we should naturally also write the inverse function that constructs the permutation from a list of cycles.

```
public static int[] fromCycles(List<List<Integer>> cycles)
```

In a fortunate situation like this where we have both a function and its inverse, an automated fuzz tester is trivial to implement. Repeatedly create a random permutation, convert it to cycle form, convert that cycle form back to a permutation, and verify that the result permutation and the original permutation are equal. This fuzz test could be left running overnight to strengthen our belief in the correctness of the methods, but a million random permutations should cover all possible edge cases and corner cases that this method can reasonably have lurking inside it.

One property of permutations that becomes easy to compute from the cycle representations is the **algebraic order**, that is, the smallest positive integer `k` so that `power(perm, k)` is the identity permutation. In fact, the `power` of a permutation from the previous lab would have been much easier to compute in the cycle form, since we could just shift each cycle `k % c` steps to the right, where `c` is the length of that cycle. Since the identity permutation keeps every element in its original position, the order `k` is simply the **least common multiple** of the cycle lengths inside the permutation. For example, if some permutation consists of cycles of length 6, 4, 1, 1, and 2, their least common multiple 12 is enough to rotate every cycle back to its original position.

The **parity** of the permutation is another important algebraic property that can be easily computed from the cycle representation. This parity tells us how many **pairwise swaps** of two distinct elements are necessary to actually perform the permutation. These swaps can be organized in a host of different ways, but whether there will be an odd or an even total number of swaps is an inherent fixed quantity for that permutation. (Try it for yourself, if you don't believe this. You will get stuck no matter which way you twist and turn.)

```
public static int parity(List<List<Integer>> cycles)
```

To compute the parity of the permutation given in its cycle form, simply count how many individual cycles have an even number of elements. If there is an odd number of such even-length cycles, return `-1` for the odd parity, and otherwise return `+1` for the even parity, hey bada bing, bada boom. This terminology where minus one stands for odd and plus one stands for even, despite the fact that both odd numbers, comes from the odd-looking term $(-1)^n$ that occasionally pops up in combinatorial formulas, especially those that involve the inclusion-exclusion principle where alternating terms are added and subtracted from the total. The sequence of values of $(-1)^n$ alternates between -1 and +1 depending on whether n is odd or even, hence this terminology. Of course, this little expression should always be evaluated with one simple conditional statement, instead of honestly computing out its value with n - 1 actual multiplications.

```
public static String cycles(List<List<Integer>> cycles, String alphabet)
```

Since the canonical cycle notation is more readable than Java code, the last method in this lab creates a human-readable representation of the cycles of the given permutation. Instead of using integers 0 to n - 1 separated by commas and spaces, the `alphabet` string tells which character to use for each such integer. Using the digits `0-9` as the first ten characters, the lowercase letters `a-z` as the next twenty-six and the uppercase letters `A-Z` as the next twenty-six allows us to express

these numbers without ambiguity or commas. Parentheses separate the cycles from each other. (Technically, as the closing parenthesis is redundant, this representation could be even more compact, but these parentheses make the expression easier for humans to digest in one glance.)

perm	Parity	As canonical cycles
{1, 0, 2}	-1	"(10)2"
{2, 4, 1, 0, 3}	+1	"(43021)"
{0, 5, 2, 3, 4, 1}	-1	"(0)(2)(3)(4)(51)"
{13, 10, 12, 1, 4, 6, 5, 11, 14, 7, 8, 0, 3, 2, 9}	+1	"(4)(65)(e97b0d2c31a8)"

Lab 47: Permutations III: Lehmer Codes

JUnit: PermutationsTestThree.java

Every first course on combinatorics starts by observing that there are exactly *n*! (read that out loud as "*n* factorial") permutations for *n* elements, the product of all positive integers up to and including *n*. For example, 5! = 1 * 2 * 3 * 4 * 5 = 120. Many well-known techniques exist to generate all these permutations one permutation at the time in various orders. (If you want to fool around a little with some of these techniques, check out the page "Generate Permutations" at Combinatorial Object Server.)

In this lab, we look at an interesting way to convert permutations to integers with Lehmer codes. Before we can explain how Lehmer codes work, let's think a little bit about how integers can be represented as sequences of digits. In ordinary **positional number systems**, the position of each digit indicates the power of ten that the digit should be multiplied by in the total sum of digits. In base ten, there is no need for digits greater than nine, since having ten of some power of ten is equal to **carrying** one to the next higher power instead. For example, we say "nine hundred" but not "ten hundred", since the latter carries to the next column to become "one thousand". Humanity has agreed to use base ten since time immemorial, but the same principle works for other bases, most importantly for base two for **binary representation** of integers.

However, few people seem to realize that the base does not need to be the same in every position. **Mixed-radix representations** allow different bases to be used for different positions. We all do this routinely without even realizing that we are doing so whenever we talk about times and dates expressed in mixed bases of 365/24/60/60/1000 depending on which combination of days, hours, minutes, seconds and milliseconds we are talking about. The base in each position still determines the largest digit that can appear in that position. For example, since every hour has sixty minutes, we say "one hour and fifteen minutes" instead of "seventy-five minutes".

Since there are exactly $n!$ different permutations of n elements, our process of encoding each permutation into an integer should utilize this fact to the maximum effect. This is achieved by using the factorial number system where the positions denote factorials 1, 2, 6, 24, 120, 720, ... (The Wikipedia page starts the factorials from zero, but this is redundant due to the duplication 0! = 1! = 1, so this position would be unused.) The largest digit allowed for the position that corresponds to $k!$ is k, since if you have the term $k!$ repeated $(k + 1)$ times in the sum, you actually have the next higher factorial $(k + 1)k! = (k + 1)!$ without realizing it.

The following two methods implement the conversion between long integers and their factorial number system (or "factoradic" for short) representations. Since these two functions are each other's inverses, the JUnit fuzz tester again repeatedly generates a random number and verifies that these functions cancel each other out.

```
public static void toFactoradic(long key, int[] coeff)
public static long fromFactoradic(int[] coeff)
```

Express the integer `key` as a sum of factorials so that $k!$ may be used at most k times. Note that this method does not return anything, but should write the answer into the array named `coeff` provided by the caller. (The caller is also responsible for this array object having sufficient length to contain the answer.) The method should fill in the given `coeff` array so that each individual element `coeff[k]` tells how many times the factorial $(k+1)!$ appears in the breakdown of the integer `key` as a sum of factorials. For example, 2168 breaks down to 2! + 3! + 3(6!) that you can verify is equal to to 2168 = 4 + 6 + 3*720.

key	Expected result (to be filled in the prefix of `coeff` array)
8	{0, 1, 1}
22	{0, 2, 3}
101	{1, 2, 0, 4}
2168	{0, 1, 1, 0, 0, 3}
61319700746071	{1, 0, 1, 1, 1, 6, 2, 7, 6, 8, 7, 4, 5, 13, 14, 2}

To see how we can use this representation to encode permutations of n elements uniquely into integers from zero to $n! - 1$, we next need the notion of an **inversion** inside a permutation. In your later courses on data structures and algorithms, this concept will come up again when analysing the lower bounds for the running time of a comparison sort algorithm. A **right inversion** inside the permutation `perm` consists of any two positions `i < j` so that `perm[i] > perm[j]`. Intuitively, the elements in positions `i` and `j` are "out of order" with respect to each other compared to the identity permutation where all elements are in order and therefore the permutation contains zero

inversions. An array sorted in descending order has the largest possible number of $n(n - 1)/2$ inversions, for the "n choose two" possible ways to choose two positions among n positions.

The **inversion count array** whose i:th element is the count of how many right inversions the position i participates in (that is, how many elements in positions $j > i$ are less than the element in position i), uniquely identifies each permutation. Even better, these inversion count arrays can be reversed and converted to integers with the previous two methods! The conversion back and forth between permutations and inversion count arrays will be performed by two methods for you to write in the `Permutations` class.

```
public static int[] toLehmer(int[] perm)
public static int[] fromLehmer(int[] inv)
```

The following table contains some randomly generated permutations and their expected inversion count arrays. Note that since the last element of the inversion count array is always zero, we will leave this element out of the inversion count array altogether, so the inversion count array always has one fewer element than the original permutation. For example, in the permutation `{5, 6, 2, 3, 1, 0, 4, 7}`, there are five elements after element 5 that are less than 5, five elements after element 6 that are less than 6, two elements after element 2 that are less than 2, two elements after element 3 that are less than 3, and so on.

perm	inv
{0, 1, 2}	{0, 0}
{0, 3, 4, 1, 2}	{0, 2, 2, 0}
{1, 3, 4, 5, 0, 6, 2}	{1, 2, 2, 2, 0, 1}
{6, 3, 2, 4, 1, 0, 5}	{6, 3, 2, 2, 1, 0}
{5, 6, 2, 3, 1, 0, 4, 7}	{5, 5, 2, 2, 1, 0, 0}
{3, 14, 13, 11, 4, 1, 5, 10, 9, 8, 12, 6, 2, 0, 7}	{3, 13, 12, 10, 3, 1, 2, 6, 5, 4, 4, 2, 1, 0}

The conversion from the permutation to the inversion count should be a pretty simple thing with two nested loops, but the reconstruction of the permutation from the inversion counts is a nice little programming exercise.

The last two methods in this lab combine these two operations into back and forth conversion between permutations and integer keys. Note that the method `fromKey` needs to know not only the key but the number of elements n in the permutation, since the same keys mean different permutations when applied to different values of n.

`public static long toKey(int[] perm)`

This method should first call the previous method `toLehmer` to compute the inversion count array for `perm`. This inversion count is then turned into a scalar integer value by applying the method `fromFactoradic` to the **reversed** inversion count array. (It is quite astonishing to realize that after almost three decades of existence, the `Arrays` utility class in the Java standard library still does not have a method to reverse an array.)

`public static int[] fromKey(long key, int n)`

This method should use the method `toFactoradic` to convert `key` into its factoradic form, which is then reversed and given to the `fromLehmer` method to produce the original permutation.

Lehmer encoding maps the permutations of *n* elements into integers from 0 to *n*! - 1 in **ascending lexicographic order**. For example, when *n* = 4, the lowest permutation `{0, 1, 2, 3}` maps to the lowest key 0, and the highest permutation `{3, 2, 1, 0}` maps to the highest key 23. Faster and more clever techniques exist to iterate through the *n*! permutations in lexicographic and many other useful orders, producing them one permutation at the time so that you won't run out of memory even trying them all for some *n* for which the iteration itself would be an overnight run. But at least our simpler method allows us to quickly jump to any arbitrary position in the sequence of permutations...

Readers whose interest in permutations has been sufficiently piqued by these three labs might want to check out the author's old example class `Permutations` that produces permutations one at the time as a lazy `Iterator<List<Integer>>` with a **backtracking** approach converted to iteration so that each call to the method `next` continues this simulated recursion from the place where it left off in the previous call. To save memory, the same `List<Integer>> object` is given out every time with its elements rearranged. (The object seen by the user code has been silently decorated with a `Collections.unmodifiableList` decorator to prevent the outside world from messing up the contents of this list, another nifty little OOP trick that can come handy when some internal implementation details need to be exposed for reasons of efficiency.)

This generator can be given an optional `Permutations.Predicate` object that acts as a **filter** to let through only the permutations that satisfy that predicate. For example, we can generate all permutations of *n* elements where each element is moved at most *k* steps away from its original location. Or generate only the alternating permutations where the element values zigzag up and down, such as `{1, 6, 3, 5, 2, 4, 0}`. The backtracking approach generates each permutation only as far until the generated prefix is rejected by the filter predicate, so it will not waste time iterating through all possible ways to complete the remaining elements that will not change the fact that the bad prefix has already violated the filter.

Lab 48: Disjoint Clumps

JUnit: ClumpsTest.java

All serious programming languages offer the important data structures such as sets and lists either in the language or its standard library, since these data structures are so immensely useful in solving problems that are universal through every problem domain. However, not every data structure appears in practical use often enough to earn its entry to this exclusive club. In this lab, we will implement the basic version of the disjoint set data structure, one of the lesser known data structures that never made the cut to the major leagues and eventually found itself working in his old man's used car dealership. However, its idea is elegant in its simplicity and thus worth knowing, even if we never go on to implement any of the classic algorithms that gain from the help of this plucky hometown champ.

Create a new class `Clumps` into your BlueJ labs project folder. Each object of this class represents some **partition** of numbers from 0 to *n*-1 into **disjoint clumps** so that at any moment, each number belongs to exactly one clump. Initially, every number belongs to its own singleton clump. Clumps are actually just disjoint subsets, but during this lab, we shall use the more colourful word "clump" to emphasize how these subsets will be crudely clumped together to create bigger clumps. Before looking at the implementation details, here are the public methods that define what this data structure does for its outside users.

```
public Clumps(int n)
```

The constructor of this class to initialize the data structure for integers 0, ..., *n* - 1.

```
public boolean sameClump(int a, int b)
```

Determines whether the integers `a` and `b` are currently in the same clump.

```
public boolean meld(int a, int b)
```

If `a` and `b` are already in the same clump, nothing happens. Otherwise this method combines the clumps that contain integers `a` and `b` into a single clump that contains all integers that were originally in the same clump as either one of the arguments `a` or `b`. This method should return `true` if the clumps of `a` and `b` were disjoint but became the same clump as a result of this operation, and return `false` if `a` and `b` were already part of the same clump so that nothing happened in this meld.

```
public int clumpSize(int a)
```

Counts how many numbers are currently in the same clump as the integer `a`, and returns this number. (Code inside this method will not actually be counting anything, but will simply look up the value from the `size` table.)

These clumps can only ever grow larger when they `meld`, and no clump is ever broken apart into smaller clumps. Eventually all numbers belong to the same clump, so nothing can change. Such **monotonicity** often allows liberties in the data structure design to allow a solution that would not bear the presence of **entropy-increasing** operations such as "remove" or "split". The disjoint set data structure allows the methods `sameClump`, `meld` and `clumpSize` to work in time that is, for all purposes both practical and impractical, a small constant that is independent of the value `n`. Even though one `meld` operation might theoretically combine two clumps of size n / 2 into a clump of size n, the internal updating of the data structure still happens in O(1) constant time!

It may seem strange that updating a data structure with billion elements needs no more time than updating a data structure with a hundred elements, so let us now pull away the cheap plywood and the black velvet cloth that has been to the laundry a couple of times too many to uncover the secret inside this magic trick. To allow the previous operations to be implemented efficiently for arbitrary clumps, the current partitioning of elements into clumps will be encoded in two arrays

```
private int[] parent;
private int[] size;
```

The values of these fields are initialized to array objects created in the constructor when we know the value of n that they need to be big enough to contain. The actual elements are initialized with assignments `parent[a] = a` and `size[a] = 1` for each number `a` from zero to n - 1.

Every clump has exactly one **representative** that can be any one of the numbers in that clump. (The lucky winner will be determined by the order and content of the `meld` operations performed to get to the current state.) The number `a` is the representative of its clump if and only if it satisfies `parent[a] == a`. Otherwise, to find the representative of the clump that the number `a` belongs to, continue looking for the representative in `parent[a]` the same way, until you arrive at the representative of the clump that equals its own parent.

This operation of looking for the clump representative of the given number should be implemented as a private helper method

```
private int findRepresentative(int a)
```

that finds the clump representative of `a` by following the parent information. After the desired representative has been found, this method should perform **path compression**, since this cleanup operation costs us essentially nothing and can massively speed up future queries. To perform this path compression, simply follow the parents from `a` to its representative the exact same way as before, and reassign `parent[x] = r` for every number `x` in this path.

The method `sameClump` is now easy to implement as a one-liner that checks whether its parameters `a` and `b` have the exact same representative.

The array `size` keeps track of how many numbers belong to each clump. However, this `size` information is maintained only for those numbers that are actually representatives of their own clumps. The value of `size[x]` for any other number `x` is never needed for anything.

The method `meld` should first check whether its parameters `a` and `b` are already in the same clump, and return `false` by doing nothing in that case. Otherwise, find the representatives of `a` and `b` in the structure, let us call these representative numbers `ra` and `rb`. In principle, either one of the two assignments `parent[ra] = rb` or `parent[rb] = ra` would combine these two clumps into one, but it is more efficient to reassign the parent of the smaller clump to rein in the growth of the implicit tree structure in these `parent` values. Remember to also update the `size` information for the representative that you chose to make the parent of the other representative.

Once the `Clump` object for partitioning `n` elements has been created, each operation `sameClump`, `meld` and `clumpSize` works in **time that is for all practical purposes a constant** independent of the total number of elements `n`. There are some technical details best left for the second course on algorithms, such as the fact that this "constant time" is not really a constant, but a constant plus a certain weird little function of "inverse Ackermann" that is difficult to explain. This function depends on *n* and will grow without bound as *n* increases, but grows so excruciatingly slowly that it would make even glaciers look like light beams in comparison. The author can guarantee without any hesitation that you will not find a slower growing function than this one anywhere in the materials that comprise your undergraduate computer science education. Treating this function as a minuscule rounding error in the constant running time can never give you any hassle in any possible counterexample, not just in our real world but in any imaginable one.

Lab 49: Bits Of This And That

JUnit: BitwiseStuffTest.java

Dealing with bits and bytes directly inside Java is the closest that we get to the true essence of the underlying machine that is executing our programs. We prefer to think about execution and the general meaning of a program using higher level languages and abstractions, so this can be considered character building even for students who have no plans to take courses on lower level coding. Furthermore, the knowledge of **bitwise arithmetic** in Java will directly translate to every other programming language and environment, so students need to learn this stuff only once to use it productively, no matter where their lives may take them after this course.

Depending on your attitude and the outcome of the random coin flips at the inflection points of your future life, this lab could either be the most important lab you ever complete, or the least important.

Unfortunately, we cannot know quite yet which one, but having come this far, are you willing to take that risk? (As you will learn in a future computer science course that you should take around the fourth year, whichever name that course might have in the institution of your studies, many computational problems have this kind of curious nature in that we cannot hurry to the result any faster than helplessly waiting for the system eventually produce the result for us.)

High-level languages such as Java and Python allow programmers to talk about integers and their arithmetic operations in their Platonic essence without knowing how these integers are actually stored, warts and all, inside a computer as aggregates of typically four bytes (`int`) or eight bytes (`long`), let alone requiring the programmer to perform bitwise arithmetic to juggle these bits and bytes around to achieve the desired end results. What a time it is to be alive despite everything else that is going on in the Current Year. However, just from the point of view of general knowledge of the reality underneath (in the same spirit of why in all fields of real life, we need to occasionally get own hands dirty instead of always delegating all our needs to the public interface that giant faceless utilities wait behind) and also many practical optimizations in time and space that this knowledge grants its holders, this lab teaches you to use bitwise arithmetic to access and manipulate the individual bits of the given integer value.

The same integer can be represented in different bases, usually 10 or 2, and it is still the same mathematical object underneath, silently and eternally floating in the Platonic realm in which all mathematical objects finger-quotes "exist". The integer that we normally write out in our universally agree base ten as `42` is the exact same integer that we would write out in binary as `0b00101010` to make it clear that 42 = 32 + 8 + 2, as you can see from how the logical expression `42 == 0b00101010` evaluates to `true` in Java. As a convenient shorthand, we say that bits whose value is 1 inside the number are **on**, whereas the bits whose value is 0 are **off**, analogous to some kind of light switches or signal flags.

Since the binary notation tends to get cumbersome once we get past our built-in mental limitation of seven plus minus two things to keep track of, the highly convenient **hexadecimal notation** groups four bits into one hexadecimal digit from `0` to `F`, where the letters `A` to `F` denote the six missing digits 10 to 15 that do not exist in base ten that goes only up to nine. (In retrospect, perhaps those limeys were on to something with the base 12 in the imperial system, seeing that the digits in that base can literally go up to eleven.) Since an `int` consists of 32 bits, and each hexadecimal digit encodes a group of exactly four bits, every `int` value can be given as eight hexadecimal digits, such as the famous hexadecimal literal `0xDEADBEEF` or its more juvenile incarnations such as `0xB00B1E5`. The prefix `0x` tells the parser of the Java compiler that the integer digits should be interpreted in hexadecimal even if they all happened to be in `0-9`, so there will never be any ambiguity between the numerical representations inside source code. The 64 bits inside each `long` value can similarly be encoded into sixteen hexadecimal digits.

Again, all these literals could have been written in base ten, but the use of hexadecimal notation makes it clear to the reader that the integers are supposed to be semantically thought of as bit

patterns instead of ordinary integers whose semantic meaning is normally associated with counting and arithmetic.

Bitwise arithmetic operators `&`, `|` and `~` correspond to the logical operators `&&`, `||` and `!` with the difference that the former operate on integers instead of `boolean` truth values where they perform the operations in parallel separately for each bit position.

The result of the **bitwise or** operator, denoted by `x|y` for two integers `x` and `y`, is on for all positions where **at least one** of the operands `x` and `y` is on. This operator can **turn on bits** of the first number `x` regardless of whether those bits were previously on or off, by making the second number `y` equal to a **mask** that is on in those positions that we want to turn on in the result.

The result of the **bitwise and** operator, denoted by `x & y` for two integers `x` and `y`, is on for all positions where **both** of the operands `x` and `y` are on. This operator can **turn off bits** in the first number `x` regardless of whether those bits were previously on or off, by making the second number `y` equal to a **mask** that is on in those positions that we want to turn off in the result.

The positions of bits inside a 32-bit `int` are numbered 0 to 31 from lowest to highest. Since all Java primitive integer types are signed, the highest bit is always the **sign** of the number so that 0 means that the number is positive or zero, and 1 means that the number is negative. The remaining bits in negative numbers do not have the same meaning as they have when the number is positive, but are given in **two's complement encoding** to guarantee that the number zero does not have two distinct representations as +0 and -0. This sign bit matters when the bit pattern is treated as an integer that performs integer arithmetic, but does not show up in the hexadecimal notation for integer literals, since the hexadecimal notation represents the underlying bit pattern directly instead of the semantic meaning of that bit pattern as an arithmetic integer.

To create a mask whose *k*:th bit is on and all other bits are off, use the expression `(1 << k)` where `<<` is the **left shift** operator for bits. Note that this is not a **cyclic** left shift, but the incoming bits from the right are initialized to zeros, and the outgoing bits to the left simply disappear. The **right shift** operator is also similarly not cyclic, but there are two versions `>>` and `>>>` with the difference that `>>` maintains the value of the highest order bit (since that bit denotes the **sign** in a signed integer representation) whereas `>>>` brings in a zero bit into that position. Your choice of which one of these operators you need to use therefore depends on whether you are treating that bit pattern as a signed integer in the rest of your code. (Bytes and aggregations of bytes do not inherently have any semantics, but the same 32 bits inside a computer are whatever you treat them as. If you use them as an unsigned integer, that is what they are then, and if you treat them as a signed integer, that is what they are. By themselves, all bits are just zeros and ones without any inherent meaning.)

You can use the left shift as a shorthand for multiplication by some power of two 2^k, but note that the results can be surprising if the sign bit of the result is different from that of the original number. This move allows us to create masks for an one bit position, and more complex masks can be

combined with the bitwise or operator. To create an integer mask whose *k*:th bit is off and all other bits are on, use the bitwise negation to the previous mask with `~(1 << k)`. Such a mask can often be thought to play the same fiddle as a **stencil** plays in spray painting (an analogy apt in more levels than just the surface one).

Enough theory. Before we can go to the four methods to be written in this lab, note that these methods have intentionally been defined to operate on 64-bit `long` parameters (all previous operators work the same way but make sure to use `long` values where appropriate) instead of 32-bit `int` values, since we live in the era of 64 bits nowadays anyway. (Some student could perhaps turn this into one of those "Virgin vs. Chad" memes, arguably the single greatest artistic achievement of Generation Z so far.) The utility class Long contains a bunch of `static` methods for useful bitwise operations that did not make the cut to the language itself, such as `bitCount`, `rotateLeft` and `parseUnsignedLong` that you might be useful when you write the following methods in the class `BitwiseStuff` in your BlueJ labs project.

```
public static int countClusters(long n)
```

Counts how many clusters of maximal consecutive blocks of ones the parameter value `n` contains, and returns that count. For example, for the value of `n` whose representation in binary would be `0b1001110111000111000000000110000100000110000000011111111110`, the correct result would be 8 for the eight clusters of consecutive ones inside the number. Note that each singleton one bit with zero bits on its both sides is still a maximal cluster that should be counted. Make sure to add up the clusters at the very beginning and the end of the number correctly.

```
public static long reverseNybbles(long n)
```

The term **nybble** (or "nibble" depending on which side of the pond you grew up on) is used to denote the four bits that comprise each half of a single byte but not, for example, the four bits that are in the middle of that byte. One hexadecimal digit therefore denotes exactly one nybble. Given a 64-bit long value that consists of 16 nybbles, compute and return the long value that contains these same nybbles but in reverse order. Note that the four bits inside each nybble should remain as they were. For example, when called with the value whose hexadecimal representation is `0x63e821ceea5afcc6`, the result would be `0x6ccfa5aeec128e36`, the original hexadecimal representation read backwards. (So as always, let George do it. Sergeant, erect that flagpole!)

```
public static long dosido(long n)
```

Swap each bit in every even position with the bit in the next higher off position, so that positions 0 and 1 are swapped with each other, positions 2 and 3 are swapped, and so on up to positions 62 and 63. For example, the value whose hexadecimal representation is `0x8074db18c339bafb` would produce the result `40b8e724c33675f7`. For example, the lowest nybble `0xb == 0b1011` turns into the nybble `0x7 == 0b0111` in this operation.

```
public static int bestRotateFill(long n)
```

Even though Java does not have a **cyclic left or right shift** operator in the language itself, these two operations can come handy in combinatorial problems so they are available as static methods `rotateLeft` and `rotateRight` in the wrapper class Long. This method should find and return the number of steps `k` so maximizes the `bitCount` of the result of the bitwise or operation `n | m` where `m` is the result of shifting `n` cyclically `k` steps to the left. If there are several equally good values for `k`, return the smallest such `k`.

With the same ideas needed to implement the previous four operations, many other bitwise operations can be implemented. In the real world, the necessity of bitwise arithmetic indicates some kind of low-level programming done close to the iron (hardware), making speed and efficiency the essence. Students who are interested in these kinds of more or less clever optimizations can check out the site "Bit Twiddling Hacks", the classic book "Hacker's Delight", or the free online textbook "Matters Computational" for an encyclopaedic presentation of all sorts of combinatorial and numerical techniques from the bit level and above.

To bring this lab to an end, you can still use the one-letter operators `&` and `|` also inside logical conditions. In that context, these operators denote the versions of logical operators that do not **short-circuit**; that is, they don't skip evaluating the second operand when the first operand already has made the result of the entire expression certain. The Java language specification explicitly guarantees ordinary the strict **left-to-right evaluation** and short circuit behaviour for logical operators `&&` and `||`.

As a final aside, the evaluation order of Java subexpressions is generally left for the compiler. This means, for example, that `rng.nextInt(42) * rng.nextInt(99)` is not guaranteed to produce the same result in every environment even with the same internal state of `rng`, since we cannot know which one of the two calls to `nextInt` will be evaluated first! To say nothing of Lovecraftian horrors such as `a = ++a + a--;` whose results the language standard says are undefined. To ensure your intended evaluation order, you must break such expressions into multiple statements and separate them by semicolon.

This invisible danger exists in basically all widely used programming languages, and yet no textbook or other material meant for a first year programming course seems to even acknowledge its existence. For every computer program that is currently used to make decisions that affect us, the version of the compiler that was current at the time that program was compiled and released generated an evaluation order that passed not only the unit and integration tests of that project, but more importantly, the test of time in actual use. One day these programs will be recompiled with our current year compiler that will generate a different evaluation order that *usually* won't make any difference to anything...

Lab 50: The Weight Of Power

JUnit: PowerIndexTest.java

Consider a weighted voting system such as a shareholder meeting where participants may have a different voting power, called the **weight** of that voter. These voters are convening to vote either "yes" or "no" for some motion. For a motion to pass, the weighted total of "yes" votes must be at least equal to the given **quota**. In a typical **majority vote** situation, more than half of the votes are necessary to pass the motion. However, this quota can be something else than one half in special situations such as requiring two thirds of the votes for some particularly drastic measure to be passed.

In a weighted voting system, the power of each voter is not necessarily proportional to the weight of that voter relative to the total weight. For example, weights `[9, 9, 7, 3, 1, 1]`, with the quota of 16 required to pass the motion. There are really only two relevant situations to consider. First, whenever the first two voters agree either for or against, that already determines the final outcome regardless of the votes of the four little guys. Second, whenever the first two voters disagree, the seven votes of the third voter act as a **kingmaker** to determine which one of the bigger guys gets their way this time, again rendering the three little guys irrelevant.

The three little guys are said to be **dummies** (used here in the sense of an inanimate doll that cannot affect the external world with its own decisions and actions, not in the sense of being some sort of imbecile) who might as well stay home, since the outcome tallied with their votes included can never differ from the outcome tallied without them. (As shareholders, these voters still enjoy their proportional share of the profits of that company, so their votes are worth their percentage of total shares for the purpose of divvying up the company's bottom line.)

To see even more clearly how the true voting power can be very different from the pure percentage of total votes, look again at the third player who acts as kingmaker in situations where the two big guys disagree. From this kingmaker's point of view, the bigger the better, since his position to extract outside concessions for his cooperation can only grow more juicy with the size of this kingdom! Macchiavellian operators in both fiction and real life throughout history have strived to find and latch onto such positions of leverage to move the world in ways that Archimedes could only dream of.

In this lab, we compute two different power index measures for weighted voting: first the Banzhaf power index, followed by the Shapley-Shubik power index. Both computations involve recursions through the combinatorial possibilities that are relevant for that index; all **subsets** for Banzhaf, and all **permutations** for Shapley-Shubik. Besides, the last names of these guys just somehow sound so gosh darn adorable, as if they were characters from some old *Mad Magazine* parody article by Don Martin that illustrates these algorithms as wacky machines with gears and whistles as the

individual ballots make sounds like "Fwababa dabada fwawaba dabada" and "Shtoink!" as they grind through this contraption.

Both algorithms operate with the concept of **coalition**, which here is a five-dollar word for what we normally call a **subset** of voters. A coalition is **winning** if the sum of weights of its members is at least equal to quota. In a winning coalition, an individual voter is **critical** if changing his vote would make the coalition no longer be winning. The winning coalition is **minimal** if every voter in that coalition is critical so that it contains no superfluous voters that just run up the score.

For example, consider the coalition of voters `{0, 2, 5}` in the previous situation. This coalition is winning, since 9 + 7 + 1 = 17 votes is enough to reach the quota of 16. Voters 0 and 2 both are critical for this winning coalition, since removing either one of them makes the total fall below the quota. Voter 5 with his single vote is not critical for the win, since the coalition `{0, 2}` would still be winning even without him. (At least that guy still gets the satisfaction of having rooted for the winning team this time, usually feeling the more triumphant the more impotent his actual vote was for this outcome. The human condition is generally mysterious.)

The Banzhaf power index for an individual voter is **the number of winning coalitions in which that voter is critical**. These numbers are usually normalized so that they add up to one, but in this problem we will keep these counts as integer counts for simplicity, trusting the user of this method to fill in the denominator of the fraction as needed. Alternatively, and potentially allowing the following recursion to be made more efficient, the Banzhaf power index for that voter equals the **number of minimal winning coalitions that they belong to.** To compute this quantity, you have to use recursion to loop through all possible subsets. To recurse through all possible subsets of *n* elements, you have the local two-way choice on whether you take the last element in the set. Both possibilities lead to recursions through all possible subsets of the remaining *n* - 1 elements.

Create the class `PowerIndex` in your BlueJ labs project, and there the method

```
public static void banzhaf(int quota, int[] weight, int[] out)
```

that computes the Banzhaf power index of voters with given `weights`. This method does not create and return a new array, but writes the results in the array `out`, guaranteed to be of sufficient length and filled with zeros at the time of call. The recursion is perhaps best extracted as a private helper method

```
private static void banzhaf(int quota, int[] weight, int[] out,
LinkedList<Integer> coalition, int sum, int n)
```

where the additional recursion parameter `coalition` contains the voters who have already been taken into the current coalition. The top level call should initialize this to a new and thus empty `LinkedList<Integer>` instance. The parameter `sum` contains the sum of weights of the elements in the chosen `coalition`. Technically the `sum` parameter is redundant, since we could always

compute this value by iterating through the `coalition`, but constant time is always better than linear time. Parameter `n` is the length of the prefix whose all possible subsets the recursion should visit.

Two separate base cases of this recursion are `n == 0` with nothing for us to do, and the case where `sum` is greater or equal than `quota`, at which point every critical voter of the `coalition` gets its counter in the `out` array incremented by one. (You need to check for criticality inside the loop where you give out these points, since the coalition in this base case is not necessarily minimal, but can theoretically contain any number of non-critical little guys.)

In other cases, the recursion should branch in two different directions; one for taking the voter numbered `n-1` into the `coalition`, and the other for leaving the voter `n-1` out of the `coalition`. Once both branches have returned and silently updated the `out` array along the way in their base case branches, this method can terminate and return the control to the previous level. Many legitimate recursions are of this nature in that the recursive method itself does not return anything, but some data structure that is shared between all levels of recursion has been incrementally updated during the exploration of the branches in this recursion.

The Shapley-Shubik power index is computed in a rather different fashion by summing over the $n!$ possible **permutations** of these n voters, instead of the previous summing over the 2^n possible **subsets** of these voters. When the weights of the voters arranged in one particular permutation are accumulated from left to right, the voter who makes the cumulative sum reach `quota` or higher is said to be the **pivotal voter** of that particular permutation. The Shapley-Shubik power index of that voter is the number of times that voter is pivotal over the $n!$ possible permutations.

The straightforward recursive method to compute the Shapley-Shubik power indices for these voters takes the same parameters as the previous method for Banzhaf power indices, of course.

```
public static void shapleyShubik(int quota, int[] weight, int[] out)
```

The simplest way to visit all permutations is to again define a `private` helper method that takes additional parameters that describe the current subproblem. Such recursion generates each one of the $n!$ possible permutations by extending the current **prefix** in all possible ways to complete the permutation. However, to avoid explicitly visiting all $n!$ permutations inside this method, which would be prohibitive for large n, we can be a bit more clever with pivotal voters.

```
private static void shapleyShubik(int quota, int[] weight, int[] out, int
level, int last, boolean[] taken, int sum)
```

The additional parameter `level` keeps track of the recursion level, initially zero in the top level call, and increases by one in each recursive call. This level of recursion is needed so that we can tell how far the prefix of the current permutation has been generated so far. Unlike we did in the Banzhaf recursion where the voters in the current `coalition` were explicitly stored in a list, the recursion

for Shapley-Shubik uses an array of truth values to keep track which voters have already been `taken` in the current prefix. The parameter `last` remembers which voter was added to the prefix in the previous level, and the parameter `sum` contains the total weight of the voters in the current prefix.

The base case of this recursion is when `sum >= quota`, which must happen in every branch before running out of voters to add to the current prefix. Since this makes the `last` voter the pivotal element of all permutations starting with that prefix, we can add up these permutations in one swoop by incrementing `out[last]` by the number of possible ways to complete that prefix into a complete permutation. This is simply the factorial of how many unassigned voters still remain, which we can easily compute from the formula `weight.length-level`. You might want to precompute all factorials that fit inside an `int` without overflowing into a `private static` array for quick lookup during this algorithm.

When total weight of the current prefix is less than `quota`, this method should use a for-loop to iterate through the voters who have not yet been `taken` into the prefix, and recursively complete all possible permutations that start with the previous prefix extended by the new voter with a recursive call with updated `level`, `last`, `taken` and `sum` for that extended prefix.

The following table contains some `weight` arguments along with the expected correct result from both methods. The `quota` is in all cases half the sum of weights, plus one. The first row is the example from the Wikipedia page for the Banzhaf method. The second row represents a **dictatorship** that results from one of the voters outweighing everyone else together, and both metrics correctly assign all power to this dictator. The third row shows how equalizing the weights of every voter gives each voter the same voting power, as expected when nothing comes in to break the symmetry between these voters.

`weight`	Banzhaf	Shapley-Shubik
`[4, 3, 2, 1]`	`[5, 3, 3, 1]`	`[10, 6, 6, 2]`
`[1, 2, 5, 9]`	`[0, 0, 0, 1]`	`[0, 0, 0, 24]`
`[42, 42, 42, 42]`	`[1, 1, 1, 1]`	`[6, 6, 6, 6]`
`[2, 13, 17, 26, 30]`	`[2, 2, 3, 3, 3]`	`[8, 8, 28, 28, 48]`
`[10, 24, 24, 32, 40, 42]`	`[2, 6, 6, 6, 7, 7]`	`[24, 120, 120, 120, 168, 168]`
`[3, 14, 24, 26, 27, 64, 67, 70]`	`[2, 6, 7, 7, 7, 10, 10, 11]`	`[288, 1824, 2592, 2592, 2592, 9888, 9888, 10656]`
`[2, 14, 52, 60, 64, 66, 84, 94]`	`[1, 7, 9, 12, 12, 10, 11, 12]`	`[144, 1872, 3216, 5520, 5520, 5904, 7248, 10896]`

We have already been pretty clever in optimizing these recursions, especially considering that this is supposed to be the second course on computer programming. The reader might still come up with even more clever speedups to compute these indices for larger sets of voters.

However, no matter how many orders of polynomials you manage to shave off from an exponential search as if carving slices out of a turkey, you still have a slightly shifted exponential search in your hands. Assuming that you had the patience to solve the problem for n voters before this optimization, you could probably solve it for $n + 2$ voters after your optimization. Meh. Such inchwormy increases rarely arouse great enthusiasm for the future of the algorithm. Problems with large n are best solved with randomized **Monte Carlo sampling** to evaluate each term only up to precision that we need to make the decision that we need these power indices for, but that's the cost of doing business in problems that are inherently exponential.

Exponential haystacks can be a bitch to even find one measly needle from, let alone enumerating every possible straw of hay inside that stack. It probably won't be worth your while to speed up your computations in this lab by cleverly utilizing the fact that two voters with equal weights must also end up with an equal voting power in both systems, since this special case will not appear in the automated tests often enough. Some examples on the web that perform these computations by hand like some animal use this trick to minimize the work.

Interested students can easily find plenty of online material about both methods to compute power indices. (This material is not all just math and computer science; for example, see the classic essay "The Inner Ring" by C. S. Lewis that uses no math, yet is still important for this general topic.)

Lab 51: Fermat Primality Testing

JUnit: FermatPrimalityTest.java

The **trial division** test used in an earlier lab to determine the primality of a positive integer n is a good exercise on basic looping, especially once we realize that only the known primes up to the square root of n need to be examined as potential divisors. However, this approach just is not feasible for efficient primality testing, except for relatively small values of n. Results of number theory allow faster algorithms for primality testing, usually harnessing the unpredictable power of **randomness** to navigate the way through some exponentially hairy thicket of possibilities whose promises we are unable to evaluate before being forced to commit into exactly one of them. If you don't know what you are going to be doing, at least no malicious **adversary** could exploit what you are doing, by setting up the problem to seem as if your reasonable decisions at each turn lure you towards the worst possible outcome!

In this lab, you will implement the Fermat primality test, mathematically surely the simplest of such primality testing algorithms. However, before we can implement the algorithm itself as a rewarding one-liner, we need a helper method to do all the heavy lifting behind the scenes.

```
public static long powerMod(long a, long b, long m)
```

This method should compute and return the **modular power** given by the formula $a^b \bmod m$, where the **modulus** m is some positive integer greater than one. Since this method needs to deal with positive values of `a` and `b` only, this modulus operator is simply the integer division remainder operator `%`. For negative numbers, modulus and remainder operators can give very different results. For example, `-7 % 3` equals `-1`, whereas `-7 mod 3` would equal `2` if Java language had the `mod` operator.

The massive advantage of doing your all integer arithmetic modulo some m is that the result of every intermediate arithmetic operation will always be less than m. This allows even humongous powers of integers to be computed without the need to use the `BigInteger` type to explicitly represent all their millions of digits. The final result will still be correct, thanks to the formulas

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$
$$(a * b) \bmod m = ((a \bmod m) * (b \bmod m)) \bmod m$$

that work for any prime or composite m. These formulas allow us to replace any integer with its value modulo m at any stage of evaluation, and the result will be equal to the original result when taken modulo m. Furthermore, although this is not needed in this lab but is interesting and important in general, precisely whenever the modulus m is a prime, every integer a in the range from 1 to m - 1 has a unique modular multiplicative inverse in that same range, denoted by a^{-1}, that satisfies the equation $(aa^{-1}) \bmod m = 1$ and therefore allows also division to be performed inside the field. This multiplicative inverse can even be efficiently computed using the extended Euclidean algorithm, everything taking place without any divisions or other problematic operations.

To make this method blazingly fast even for large values of a, b and m, it should use the technique of exponentiation by squaring that we previously encountered in the first permutation lab. When the exponent $b = 2k$ is even, use $a^{2k} = (a^k)^2$ to chop the exponent in half, and in the cases where the exponent $b = 2k+1$ is odd, use the subdivision $a^{2k+1} = a(a^k)^2$ for effectively the same outcome. Depending on how you organize the logic of your iteration either as a while-loop or the equivalent **tail recursion**, equivalent variations of these subdivision formulas $a^{2k} = (a^2)^k$ and $a^{2k+1} = a(a^2)^k$ can also be used.

Unlike Python and its dynamic nature that allows issues such as the behaviour of some function to be not set in stone at compilation, the static nature of the Java language allows its compiler to rewrite all tail calls into equivalent iterative structures that do not allocate a new stack frame, making linear recursions of this particular restricted form to be effectively just as good as iterative loops. Since the stack size in this problem grows only logarithmically anyways, the same way as it

does for all recursions whose logic is based on repeated halving, even Python can handle this one recursively within the usual recursion depth limits without tail call optimizations.

The following table showcases some values of *a*, *b* and *m* along with their expected results.

a	b	m	Expected result
17	30	9	1
8	9	3	2
60	1909	56	32
255	428433	307	101
1656	3647549	928	640

Fermat's primality testing is based on the idea of finding a **negative witness** for the positive integer *n* whose primality we wish to determine. A negative witness is any integer *a* for which the expression a^{n-1} mod *n* gives some other value than one. The existence of even one such witness... well, why don't we use a more colourful term "paper check" perhaps inspired by this author watching one too many Big Herc and Wes Watson videos about the hard reality of prison life on YouTube to help him "do his own time" during this pandemic lockdown. But now that the lockdown is hopefully almost over, it's time to get some yard time in!

Imagine the number *n* as a criminal who caught a case that he couldn't beat, so he has been sent away for a long stretch in prison. This man proclaims to be a member of the criminal underworld in good standing with his gang, so that he has never ratted on his fellow criminals or committed certain types of crimes that his fellow inmates would not look kindly upon. Any failure in this paper check when demanded (and as the hardboiled crime writer Andrew Vachss noted in one of his *Burke* novels, no matter how many years your sentence, certain parts of it will never take long) reveals him to be nothing but a rat faced punk and a cell gangster who had better crawl right into PC and never show his face in the mainline again.

However, this whole setup is very different when *n* is an "O.P.", a genuine Original Prime who has all his paperwork hooped to go since his name is probably carved deep into the cold hard streets that he came from. No matter which particular underling number *a* the shot caller of that particular modular gang will randomly send to check his heart and papers, this man keeps his head up high with the absolute certainty that the equation a^{n-1} mod $n = 1$ will stand up for him under any amount of pressure and interrogation under hot lights that the hacks can throw at it. An entire train of such challengers of both paper and heart can be thrown randomly at *n* without changing this outcome, since the genuine prime convicts will never break, no matter what they fall behind.

On the other hand, any composite number is known to eventually crack under sustained interrogation and spill their guts about their accomplices, so it is in the interest of the criminal fraternity to quickly reveal such snakes in the grass. Those punks will soon realize that their bad choices in life led them straight into a Level 4 yard that shows no mercy! Any composite number trying to pass itself as a prime has less than a coin flip's chance to survive even one paper check, let alone an entire gauntlet of checks coming randomly from every direction with the knowledge that if he fails any of these tests at any time, every solid con in the entire block will be jumping to split his wig with their Occam's shivs.

```
public static boolean isFermatNegativeWitness(long n, int a)
```

Determines whether `a` is a negative Fermat witness for the primality of integer `n`. This should now be a one-liner. You are welcome.

Safely back in The World again, we can try a large number of randomly chosen integers *a* to challenge the primality of integer *n* is the best known randomized algorithm for primality testing. Fermat's criterion is merely the simplest of such properties of primes that have been discovered over time. The same way as how barbed wire, guard towers and bloodhounds are combined to stop any prisoners from escaping so that each measure protects against the blind spots of the other measures, the results of different types of primality tests can be combined to ensure that pseudoprimes, composite integers that manage to slyly slide through the systemic weaknesses of one particular primality test, are caught by another test that is wise to their tricks. The randomized primality testing algorithm has been implemented in the `BigInteger` class as the instance method

```
public boolean isProbablePrime(int certainty)
```

where the parameter `certainty` determines how many challengers the integer has to survive to make its bones as a genuine prime number.

An interesting property of the `isProbablePrime` method is that **its errors are one-sided** so that it can never return a false negative. Returning `false` absolutely guarantees the number in question to be composite... although in the spirit of Cromwell's rule, we still ought to ascribe a small probability of erroneous bits in the hardware executing the algorithm. (And what about the possibility of an error occurring in the underlying reality that is "executing" the physical hardware that is simulating the execution of our virtual algorithm?)

The positive answer from primality can theoretically be a false positive when the underlying number is actually a very tricky composite, but this is utterly irrelevant. Since every individual challenger gives the composite number at most a random coin flip's chance of sneaking through, a gauntlet of hundred such challengers will certainly be enough certainty for you to bet all your chips and the kitchen sink on the primality of that number. The probability of erroneously treating a composite number as a genuine prime article is lower than the probability that the computer

performing this computation is destroyed by natural means before the result emerges, or that its hardware randomly flips some bit somewhere to make the result different.

The previous randomized algorithm for primality testing can be extended to generate a random prime number of given length. This classic algorithm that does this is, of course, already available in `BigInteger` as the `static` method

```
static BigInteger probablePrime(int bitLength, Random rnd)
```

The parameter `bitLength` is the number of bits the desired prime number should consist of, and `rnd` is the random number generator used to make the random choices during the execution of this method.

The method internally operates on the principle of rejection sampling. To generate a random member from the **subset** of primes of given length, generate a random member from the proper **superset** of positive integers of that bit length. In rejection sampling, this superset should always be something from which it is easy to sample a random element without bias. For the set of n-bit integers this is trivial by randomly flipping a coin for each of the *n* - 1 highest bits, as the lowest bit is of course hardwired to one. If the previous randomized primality check accepts this superset member as also being part of the subset of primes, stop and return that number as the answer, otherwise go back and try your luck with the next random *n*-bit integer.

If the selection of the random positive integer of given length is done uniformly, as it is clearly done in this case, the rejection sampling algorithm will also sample the subset uniformly without any additional effort from our part. The same principle can be extended from random sampling of integers to more complex combinatorial structures, the algorithm guaranteeing the uniformity of sampling by design regardless of the complexity of the subset we want to draw our samples from.

Such a scattergun approach might initially seem almost silly, and yet it works amazingly well in practice. This is due to the fact that the **density** of prime numbers among all integers is much higher than most people might intuitively assume. As the final philosophical point about what we have now witnessed in this lab, we tend to think of randomness as an enemy whose effect on us we try to minimize and preferably avoid. But as you will learn in many later courses, randomness can also be a good friend and ally in your corner if you let it. As you may appreciate by watching this method spit out random prime numbers with thousands of digits in less time than it takes you to snap your fingers, the sheer effectiveness of this algorithm is perhaps the closest thing to magic that we see in this course, which is why this author took over a page just to explain it. There is nothing more to do here.

Lab 52: Linus Sequence

JUnit: LinusSequenceTest.java

In this lab we go back to generating interesting integer sequences where some simple rule makes the sequence bounce around and evolve in some chaotic manner. The incongruence between the algorithmic simplicity of the rule, versus the complexity of the outcome that emerges from the repeated application of that rule, is often perceived as adorkably comical, as if "The Little Sequence That Could" is trying to elevate itself to something higher than it has any right to be.

The Linus sequence tries its best to be as unpredictable as possible, and yet this very attempt makes it completely predictable, analogous to the equivalent phenomena we sometimes see with actual people. Before starting coding, we can let the paradoxical nature of this sequence provide us a more general and potentially useful life lesson. The argument used by Linus in the *Peanuts* comic strip behind the link is the same basic train of thought that runs deterministically wherever its by now well-worn tracks lead. As every budding mentalist knows, most people would unknowingly repeat the same reasoning if asked to generate a "random" sequence of truth values in purely mental means without use of physical aids. It's pretty rare to find somebody whose "random" sequence starts with four equal elements, even though on average this should happen to one person in eight!

(Humans in general are notoriously bad at both emitting and detecting randomness. Fortunately, algorithmic techniques have been developed to convert bad and unbalanced sources of random bits, human or machine, into more stingy but higher quality sources of randomness.)

Hoping that the readers will excuse the slightly old-timey convention of using lowercase Greek letters to denote character strings (sometime around your third year you should be taking the course where this convention is common, so that it is not just "all Greek to you"), the **maximal repeated suffix** of the sequence *s* is its longest possible suffix β that allows the breakdown of sequence *s* into three substrings $s = \alpha\ \beta\ \beta$ where α is the prefix of *s* before the duplicated suffix β. The maximal repeated suffix exists and is well-defined for any finite string *s*, since the empty string always works as the suffix β, which in turn forces $s = \alpha$. For example, the maximal repeated suffix β of the string `"1100110101101"` is `"01101"`, where α equals the leftover prefix `"110"` that could not be used to extend the repeated suffix.

Create the class `LinusSequence` in your BlueJ labs project, and there the method

```
public static int maximalRepeatedSuffix(boolean[] bits, int n)
```

that computes and returns the length of the maximal repeated suffix of the sequence that consists of the first `n` elements of the boolean array `bits`. As we have seen earlier, it is often a good idea to have a method that operates on array arguments to take additional index parameters that make that method restrict its attention to some particular subarray. The following table showcases some *n*-bit `boolean` arrays, written here as character sequences of ones and twos instead of truth values for brevity and readability, and their corresponding maximal repeated suffixes. We might as well use this idiosyncratic convention since the construction of the Linus sequence consists of two

possible elements that, despite being represented in our code as `boolean` values, seem to be canonically called `1` and `2`. (This choice of symbols is to-*one*-to, to-*two*-to anyway.)

`bits` (of length `n`)	Expected result
`2211212`	`2`
`1212121212`	`4`
`2112121112121`	`6`
`222222121222221212`	`8`
`2121112211122221121`	`0`
`1121111211121122111`	`1`

The first element of the Linus sequence is defined to be `1`, after which the sequence must be constructed one element at the time. Let θ denote the Linus sequence of length k that has been constructed so far. The element $k + 1$ of this sequence is the one from our two possibilities `1` and `2` that minimizes the length of the maximal repeated suffix of the corresponding extended sequence θ `1` and θ `2`. It turns out that there is never a possibility of a tie, but the maximal repeated suffixes of the extended sequences given by `1` and `2` will always be different lengths.

```
public static boolean[] linusSequence(int n)
```

Computes the first n elements of the Linus sequence and returns the sequence as an array of `boolean` truth values so that `false` stands for `1`, and `true` stands for `2`.

The emergent behaviour of the Linus sequence is largely unknown, and chock full of mysteries far above your humble instructor's pay grade. Appropriately enough, your instructor first learned about this sequence when browsing the Futility Closet, an excellent blog that nicely covers the intersection of recreational mathematics and historical curiosities. The category "Science & Math" is probably the most relevant for the aims and spirit of this course.

Lab 53: Tchoukaillon

JUnit: TchoukaillonTest.java

Variations of Mancala have been played around the world since time immemorial. The simple structure of this game, relative to the rich complexity of its emergent situations, would certainly allow for many good exercises about integer lists and arrays, when the legal moves of the game are implemented as methods that operate on these arrays and lists that represent the current position of the game and the position that emerges as the result of some move. The first course on artificial

intelligence, usually taken some time around the third year, covers the **minimax algorithm** to find the best series of moves to defeat all of the opponent's potential responses. (All introductory AI course professors have secretly signed the pact to keep the minimax algorithm in the course outline.)

Until we have that algorithmic machinery available, we have to content ourselves with Tchoukaillon, one of the **solitaire** puzzle versions of this game. Conveniently for our coding purposes, especially compared to the more convoluted boards of some two-player variations of Mancala, the Tchoukaillon board is a simple one-dimensional row whose positions are numbered with natural numbers the same way as lists and arrays. Each position contains some number of discrete but identical seeds, coins, pebbles, men or whatever word you wish to use to refer to your game pieces. In all methods written in this lab, the game board is represented as some kind of instance of `List<Integer>` that contains the number of seeds in each position.

Position zero at the beginning of the row is the **safe home** where you are trying to bring as many of your coins as you can. A legal move in a state consists of choosing a position whose number *k* is equal to the number of pebbles in that position; for example, the position number five that currently contains exactly five men, no more or less. If no such positions exist on the board, the game is over. Those *k* pieces of seed are sown to the *k* preceding positions, exactly one coin per such position. Each move maintains the total number of pebbles but increases the number that are safely home by exactly one, the game for *n* men reaches a terminal state after at most *n* moves.

Okay, I will stop that. From now on, the game pieces will be called "seed". Were you one of the lucky quick few who read the previous paragraphs too fast to even notice? Either way, create a new class `Tchoukaillon` on your BlueJ labs project, and in that class its first method

```
public static boolean move(List<Integer> board, int k)
```

This method grabs the `k` seed from position `k` and sows them into the previous `k` positions, one seed each. Note that this method must update the state of the `board` argument object, shared by both this method and its caller the same way as all object arguments are in Java, to correspond to the reality after this move. For example, when called for a list `{3, 1, 2, 0, 4, 5}` and `k = 4`, the contents of that list would become `{4, 2, 3, 1, 0, 5}`. Redundant zeros, created from sowing the last position, should be trimmed from the end of the list before returning.

The return value of this method indicates whether the move from position `k` was legal and successful; that is, whether the `board` position `k` contained exactly `k` seed at the time of method call. If the position `k` contains any other number of seeds, this method should not modify the `board` in any way, but return `false` to inform the caller that the attempted move was not legal.

```
public static boolean undo(List<Integer> board, int k)
```

This method is the **inverse** of the previous `move` method. Given the state of the `board` after performing the move at position `k` in some unknown previous state of the board, restore the `board` to this unknown previous state. Note that it is possible for `k` to be past the end of the board, which would happen when trying to undo the sowing from the last position of the board.

The return value of this method again indicates whether the arguments `board` and `k` represent an undo of some legitimate move within the rules of Tchoukaillon. This happens when the position `k` on `board` contains zero seed, and every position before `k` contains at least one seed, so the undo can be performed by collecting these `k` seeds back to position `k` from the previous positions. Otherwise, this method should return `false` without modifying the board. For example, trying to undo the move `k = 3` at state `{3, 1, 0, 0, 4}` would be a no-op, since the position 2 doesn't have one seed to give back in undoing the hypothetical move from `k = 3`.

Your instructor certainly stands high and mighty on his bully pulpit every time he preaches a sermon about the fruits and blessings of immutable types in programming, and how accepting these types and their associated programming style in your hearts will make your code ascend closer to its abstract form residing eternally in the Platonic paradise. However, the serpent can now legitimately point out the one giant downside of immutable types; iterating through a large number of separate values within an immutable type necessitates the creation of a separate object in memory for each such value. Furthermore, such functional programming style is simply too alien and opposite to the usual way of thinking of even seasoned professionals.

Using mutable types, the same object or variable can just keep track of the current value, from which the next value is generated in computational means. For example, we can write arbitrarily long for-loops that use a mutable primitive `long` value as their loop counters, and count up to millions of trillions without ever running out of space when the eight bytes of that variable act as our "peephole" to the long sequence of values we are iterating.

Games that allow the methods `move` and `undo` to operate **in place** to the current mutable game state, instead of having to create a new object to represent that game state that results from each move, makes searching for solutions in such games much easier in recursive means. If we imagine the Tchoukaillon `board` as a crudely simulated war zone so that these seeds have sprouted into an army of identical little army men symbolically represented as some kind of tokens under your command, how many men could you bring to the safety of home before running out of moves?

This problem, exactly the same as other search problems that ask you to design a series of moves that solve the given puzzle, can be solved with a **recursive search** that receives the current state of the puzzle as an argument. The base cases of this recursion are the states that allow no legal moves, and their values are given by the number of men who are safe. Otherwise, the method should loop through the possible moves in the current state, and recursively compute the values for the states that result from each move, in this problem easily generated in place by using `move` and `undo` around the recursive call. Return the maximum value of these possible moves as the value of the current state. This recursion should be written as one method

```
public static int value(List<Integer> board)
```

that computes and returns the maximum number of men that can be brought safely home from the given starting `board`. Despite being exponentially branching in principle, this particular recursion manages to usually be quite fast, since most states of `board` contain only one or two possible moves, and most choices of moves will quickly lead to dead ends. The following table displays some expected results for some initial distributions of these men in the front lines.

`board`	Expected result
`[0, 1, 2, 3, 4]`	`9`
`[0, 1, 2, 3, 3]`	`5`
`[0, 1, 2, 0, 3, 4, 4]`	`3`
`[0, 0, 2, 3, 4, 5, 6, 5, 3]`	`16`
`[0, 1, 2, 1, 4, 3, 6, 7, 8]`	`27`

Whenever such a `value` method is available as a black box, we could easily use it to make the actual moves by consulting the value method for all possible successors of our present `board` and simply making the move that the black box claims to be the best. We shall not fear being led astray by the marching orders implied by this box, since the individual evaluations of that box will guide our steps all the way up to the best result possible, if not the best possible result.

However, when it comes to the puzzle being fully winnable, Tchoukaillon is nearly unique among all puzzles with the following curious combinatorial property. For each positive integer *n*, there exists **exactly one** possible way to place *n* men on the gameboard so that it is possible to bring all of them safely home using legal moves of Tchoukaillon! Any other way of placing these *n* men on the board will always see the battlefield eventually quiet down with at least one man stranded behind the enemy lines, no matter which way you play the individual moves.

Since each move will always decrease the number of men outside safety by exactly one, and these men must be arranged in the gameboard in a way that makes continuing to the solution possible, it follows that these unique solvable states form a linear chain that springs forth from the goal position where all pebbles are safe. From there, the chain reaches up towards infinity by moving from the unique solvable state for *n* men to the next higher unique solvable state for *n* + 1 men, easily generated by simply performing an `undo` to the leftmost empty position `k`, creating a new empty position to the end of the list when no earlier empty positions exist.

```
public static void previousSolvable(List<Integer> board)
```

This method should perform this operation on the given `board`. In the JUnit tests, this `board` is guaranteed to be some solvable state. Furthermore, to canonize the representations of possible boards, the number of men in the safe zero position is kept at zero. (The value of any future action can only be affected by men outside safety, but never by the **sunk reward** from any men who have already reached it.)