CCPS 209 Computer Science II Labs

Ilkka Kokkarinen

Chang School of Continuing Education
Ryerson University

Version of February 18, 2022

Dramatis Personae

This document contains the lab specifications for the course CCPS 209 Computer Science II**,** as compiled and taught over the years by Ilkka Kokkarinen for the Chang School of Continuing Education, Toronto, Canada. It currently offers a veritable buffet of **95 problem specifications** for students to freely pick their battles from, with new and exciting problems added on a semi-regular basis as the author keeps coming up with ideas for them. These problems vary greatly in theme so that they promise something for everyone, in spirit of *The Love Boat*. Students who have previously taken this author's course CCPS 109 Computer Science I in Python should also note that unlike in the problem set used in that course, **these problems are not listed in their order of difficulty** but in the order in which they were created, to keep the numbering of these problems consistent for other documents outside this one. Have no fear to take a look around all over this document to see which topics interest you enough to raise the urge to hit the keyboard!

These problems drill in various aspects of the Java language and its standard library, and introduce (although occasionally between the lines) many classic ideas, concepts, techniques and algorithms that will surely come handy in your future coding tasks in both school and working life. The first half of these labs mostly contains problems about the straightforward application of basic data types such integers, arrays and lists, whereas the problems in the second half are inspired by discrete math, combinatorics and computational geometry. Occasionally these problems even reach out into more remote areas such as fractal art, game theory and political science. (This last topic is covered several problems about implementing and analyzing various voting systems.)

To work on these labs using IntelliJ IDEA, start by creating yourself new project named “209 Labs” in which all your source code for these labs should go. Into that project, you will then manually copy the automated **JUnit tests** from the instructor's GitHub repository `CCPS209Labs` for each lab that you attempt to solve. First download this entire folder somewhere in your computer, and whenever you embark on a new lab problem, copy the corresponding JUnit test class in your labs project folder. Once you have implemented all the methods of the labs so that they compile, IntelliJ IDEA will allow you to run either one test or all the tests for that particular lab, for you to determine whether your work passes the muster.

When IntelliJ IDEA initially complains about the import statements in the JUnit test class, you should correct the problem once and for all by clicking the red lightbulb symbol to open a drop-down menu from where you add JUnit 4 in your classpath. That is enough for IDEA to know to use JUnit 4 for all the tests in the same project. **Note that these tests are written in JUnit 4 framework, instead of JUnit 5**, so be sure to add the correct framework in your classpath!

The **fuzz acceptance tests** in the JUnit test classes will try out your methods with a large number of **pseudorandomly** generated **test cases**. These test cases are guaranteed to be exactly the same for everyone working on the Java language and its standard libraries, regardless of the particular computer environment they currently happen to be working on. Each pseudorandom fuzz test computes a checksum from the results that your method returns for the test cases that were given to it. Once this checksum equals the expected checksum computed and hardcoded into the test method from the instructor's private model solution, The Man from Del Monte says yes and grants you the green checkmark to celebrate your beating this level with a nice glass of pineapple juice. Your method is now correct and accepted as being **black box equal** to the instructor's private solution, without having to have this private model solution available for comparison.

Any discrepancy in the checksum reveals that your method returned a different answer than the instructor's private solution for at least for one test case. Unfortunately, such pseudorandom fuzzing scheme makes it impossible to pinpoint any one actual test case for which your method is failing, or determine for how many test cases your method disagrees with the instructor's private model solution. To alleviate this mechanism that may sometimes feel a tad bit passive aggressive, the JUnit test classes also feature conventional tests with explicit test cases aimed to flush out the most common bugs and edge case situations that seem to repeat in student solutions.

To test your code, you should also usually write your own `main` method in your classes and hardcode your own test cases there. (No law of either nature or man prevents your classes from having additional utility methods than merely those that our JUnit tests explicitly try out.)

All these JUnit fuzz tests are designed to run to completion within a couple of seconds at most per each green checkmark, when executed on a modern off-the-shelf desktop computer at most five years old. **None of these automated tests should ever require minutes, let alone hours, to complete.** If any JUnit test gets stuck that way, that means **your methods are doing something algorithmically very inefficient**, either in time or in space. You should mercilessly root out all such inefficient designs from your method, and replace them with better algorithms that get to the same end result at least an order of magnitude faster!

In all these labs, silence is golden. When it comes to your methods talking out of school all over the text console, the rules for *omertà* are as strict as they are in the thorny badlands of Sicily. Since these JUnit tests will pummel your code with a large number of pseudo-randomly generated test cases harder than an old time detective giving a suspect the third degree under the hot lights, **all required methods must be absolutely silent** and print absolutely nothing on the console during their execution. **Any lab solution that prints anything at all on the console during its JUnit test will be rejected and receive a zero mark.** Make sure to comment out all your console output statements you used for debugging before submitting your project.

Once you have completed all the labs that you think you are going to complete before the deadline, you will and must **submit all your completed labs in one swoop** as the entire "209 Labs" project folder that contains all the source files along with the provided JUnit tests and any necessary text or image data files, compressed into a zip file that you then upload into the assignment tab on D2L. As there is no partial credit given for these labs, **please do not include any solutions that do not pass the tests with flying colours into your submission**. To make it easier for the instructor to tally up your working labs, you should make it clear in the project `README` file how many lab problems you are claiming to have successfully solved.

Students should work on to solve these problems independently. Discussion of problems between students is acceptable, as long as this discussion takes place solely at the level of ideas and no actual solution code is passed between the students. This problem collection also has an official subreddit r/ccps209 to facilitate such discussion. Note that this subreddit is intended only for the discussion of these problems, and any course management issues should be handled elsewhere through the appropriate channels.

Lab 0(A): Plain Old Integers

JUnit: P2J1Test.java

The transition labs numbered 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 this author's other course CCPS 109 Computer Science I. Even if you are coming to this second course via some other route, these problems are self-contained and require no specific background knowledge. 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 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. Inside the new class, the first method to write is

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

Python has the integer exponentiation operator `**` conveniently built in the language, whereas Java unfortunately does not offer that. Exponentiation would be basically useless anyway in a language that uses fixed size integers that silently hide the overflows that the integer exponentiation will be producing in spades. (You should also note that in both Python and Java, the **caret** character `^` denotes the **bitwise exclusive or** operation that has nothing to do with integer exponentiation.)

Instead of integer exponentiation, your task is to implement the related operation of **falling power** that is useful in many combinatorial formulas. 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 $8^{\underline{3}}$ is computed as 8 * 7 * 6 = 336. Similarly, the falling power $10^{\underline{5}}$ equals 10 * 9 * 8 * 7 * 6 = 30240. Nothing essential changes here even for a negative base; the falling power $(-4)^{\underline{5}}$ is computed with the same formula as -4 * -5 * -6 * -7 * -8 = -6720. Analogous to ordinary exponentiation, $n^{\underline{0}} = 1$ for any integer n.

This method should return the falling power $n^{\underline{k}}$ where the base n can be any integer, positive or negative or zero, and the exponent k can be any **nonnegative** integer. The JUnit fuzz tests are designed so that your method does not have to worry about potential integer overflow… provided that you perform your arithmetic calculations with the `long` kind of 64-bit integers! If you use the bare 32-bit `int` type anywhere, a silent integer **overflow** will make your method occasionally return incorrect results and fail the JUnit tests, even if that method worked correctly when you tried it with your own small values of n and k.

```
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`. For example, given the array `{5, 2, 3, 10, 2}`, this method would create and return a new array object `{5, 3, 2}` that contains the elements in

the even-numbered positions 0, 2 and 4 of the original array. Make sure that your method works correctly for arrays of both odd and even lengths, and for empty arrays and **singleton** arrays with just one element. The length of the result array that you return must also be exactly right so that there are no extra zeros hanging around 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 sorted 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 show up as

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

when displayed in the standard **matrix form** that is more readable than the more truthful form `{{4,5,6,7,8},{13,12,11,10,9},{14,15,16,17,18},{23,22,21,20,19}}`,in reality a one-dimensional array whose four elements are themselves one-dimensional arrays of integers that serve as rows of this two-dimensional array. In addition to rectangular grids of values, this scheme can represent arbitrary **ragged arrays** whose rows don't need to all be of the same length.

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

Inside an array `arr`, an **inversion** is a pair of two positions `i<j` so that `arr[i]>arr[j]`. In combinatorics, the inversion count inside an array is a rough measure of how much “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 inversions again in this course, if you work through the 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, add the above **JUnit test** class P2J1Test.java inside the same project. 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 must 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. Even better way to handle this situation would be to have the method consists of the one-liner body

```
throw new UnsupportedOperationException();
```

that tells the caller in a controlled fashion that the method does not even pretend to achieve anything at all, but gives up right away. Of course all such method stubs will immediately fail their respective tests, as they properly should. However, their existence allows the JUnit test class to compile and run cleanly so that you can start testing each one method the moment you have properly replaced the above statement in its body with the actual Java code that solves the specified problem. IntelliJ IDEA conveniently allows you to run all tests for that problem, or just one individual test, just by clicking the green Play button next to each individual test in its source code.

Lab 0(B): Dublications

JUnit: P2J2Test.java

Create a new class named `P2J2` inside the very same 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 brand 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. Only 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. For the purposes of this problem, the upper- and lowercase versions of the same character are considered different, so that removing duplicates from `"AaabbBBBCc"` should return the result `"AabBCc"`.

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

Create and return a new string that contains each character of `text` only once regardless of its appearance count there, these characters listed in order that they appear in the original string. For example, given the argument strings `"Kokkarinen"` and `"aaaabbxxAxxaaxa"`, this method should return `"Kokarine"` and `"abxA"`, respectively. Same as in the previous problem, the upper- and lowercase versions of the same character are treated as different characters.

You could 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. But you can be more efficient and use a `HashSet<Character>` to remember which characters you have already seen, and let that data structure help you decide in constant time whether the next character is appended into the result.

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

Some number of chess 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 return the count of how many remaining squares are safe from the rolling wrath of these rampaging rooks, that is, do not contain any rooks in the same row or column.

Probably the best way to do this is to create and fill in two one-dimensional boolean arrays `safeRows` and `safeCols` with `n` elements each. These arrays keep track of which rows and columns of the chessboard are unsafe. Initially all rows and columns are safe as far as we know, since both arrays are initially all `false`. However, to get count of the actual situation in the given

chessboard, loop through all the squares of the chessboard. Whenever you find a rook in some position `(row,col)`, mark that entire `row` and `col` as being unsafe in those respective arrays. After you have done this for all the squares throughout the entire chessboard, loop through the `safeRows` and `safeCols` arrays separately to count how many rows and columns are still safe. Return the product of those two numbers as your final answer.

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

Compute and return *n*:th term in the Recamán's sequence, starting the count from term $a_1 = 1$. See the definition of this curious sequence on the linked Wolfram Mathworld page, or read more from the page "Asymptotics of Recaman's sequence". 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[]` 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. The size `10*n` should be enough, and yet this scheme uses roughly ten bits of heap memory for each number generated. (Storing the encountered numbers into some `HashSet<Integer>` instance instead would burn up memory at least an order of magnitude harder.)

Lab 0(C): Reverse Verse

JUnit: P2J3Test.java

One last batch of transitional problems adapted from the instructor's collection of Python graded labs. Only three methods to write this time, though. The JUnit test for these labs uses the `warandpeace.txt` text file as source of data, so please ensure that this text file has been properly copied into your course labs project folder.

```
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}` (the 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}`, the contents of that array would stay as `{5, 4, 3, 2, 1}` seeing that its each element is a maximal ascending subarray of length one.

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

This nifty little problem is taken from the excellent Wolfram Challenges problem site where you can also 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, even in this day and age the Java `String` type still does not come with a `reverse` method built in. 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 without fuss or muss. Maybe one day the Java strings will come with the `reverse` method, just like the string data types of all sensible programming languages these days. Or at least have that available 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 `"Uncle Sente lives in Russia"`, this method should create and return the string `"Ancli Sunti levis en Resseu"`.

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 character classification methods in the `Character` utility class to determine whether some particular character is in upper or lower case, and convert some character to its upper or lower case version as needed.

Lab 0(D): Lists Of Integers

JUnit: P2J4Test.java

This 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>` data type of Java standard library (remember to put the necessary imports to the top of the source code of your class) whose behaviour and operations are essentially identical to those of ordinary Python lists of integers, except with a way 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 a positive integer, return the first positive integer missing from this list. For example, given a list `[7, 5, 2, 3, 10, 2, 9999999, 4, 6, 3, 1, 9, 2]`, this method should return 8. Given one of the lists `[]`, `[6, 2, 12345678]` or `[42]`, this method should return 1.

Since the **pigeonhole principle** dictates that the first missing positive integer of an n-element list must necessarily be less than or equal to $n+1$, a `boolean` array of that many elements can conveniently be used to keep track of which numbers you have seen two far, and solve this problem efficiently in both time and space with two **consecutive** for-loops, instead of being a Shlemiel who does the same job using two **nested** for-loops. For the purpose of minimizing the method running time, these two constructs are not at all equal!

```
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 have the same frequency in the parameter list, those elements should end up in **ascending order of values**, the same way as we do in ordinary sorting of lists of integers. Note that this method does not return anything, but modifies `items` in place.

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]`. The return type of this method is `void`, because 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. Start by building a local **counter map** of type `Map<Integer,Integer>` used to keep track of how many times each value that you see appears inside the list. Next, define a local class that implements `Comparator<Integer>` and whose `compare` method performs integer order comparisons by consulting this map for the counts of those elements to compute the answer, reverting to ordinary integer order comparison only in the case where these counts 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 of prime factors 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 3,628,800, which equals the product of the first ten positive integers. Remember that when `n` equals either zero or one, its factorial equals one (as there exists exactly one way to arrange the elements of the empty list and the singleton list in a row), and therefore produces an empty list of prime factors.

The factorial `n!` grows exponentially, so when `n` goes up to not just mere eleven but even past that mythical barrier, these factorials would no longer be representable inside the 32 bits used to house the primitive `int` type of Java. Since this method must be able to work for values of `n` that are in the thousands and their factorials consist of tens of thousands of digits, you should not first compute that entire 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. When you have done this for all integers up to `n`, sort the result list in ascending order 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 this time. The Python integer type is **unbounded** so that its magnitude is limited only by your available heap memory. The Python virtual machine silently switches between different efficient representations for small and large integers, so that up here we can concentrate on the actual problem instead of the details of nitty gritty integer arithmetic. When dealing with the primitive integer types of Java, we have to constantly worry about and protect against overflow errors. (This is typically achieved by closing your eyes and hoping really hard that nobody will ever pass your methods integer arguments big enough to cause some intermediate result to overflow.)

The utility class `java.math.BigInteger` allows you to do computations in Java with similarly unlimited integers. This again being Java as originally conceived in the early nineties, the syntax is not quite 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` as if our nation had lost a war or something. (Online explanations why Java freaking still doesn't have **operator overloading** make for a good reading for those times that you feel like you could use a chuckle.) 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 interesting combinatorial properties that also make for more meaningful problems for us to play with. 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 performed from higher indices going down.)

The unique breakdown of `n` 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 the list `[832040, 121393, 46368, 144, 55]`.

Your method must remain efficient even if `n` contains thousands of digits. To achieve this, maintain a `static` 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); }
```

Whenever the last Fibonacci number stored in this list is not big enough for your current needs, extend the list with the next Fibonacci number, computed by adding up the last two known Fibonacci numbers. Even though this is not needed in this problem, keeping the Fibonacci numbers discovered so far in one sorted list would also allow you to quickly determine whether the given integer is some Fibonacci number with `Collections.binarySearch`.

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

Since seven is a 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 7000000000000000000000000000000000000. All sevens must be in one consecutive bunch, followed by all the zeros in another consecutive bunch. This rule keeps the space of such numbers small enough to examine through a brute-force search, since there exist only $n + 1$ such numbers that have n digits.

In the wonderful MIT OpenCourseWare online textbook "*Mathematics for Computer Science*" (PDF link to the 2018 version, for anybody interested in that sort of stuff), one of the examples of **non-constructive proofs** with the pigeonhole principle to establish the necessary existence of at least one mathematical object with the particular quality proved that for any positive integer `n`, there exists some seven-zero integer that is divisible by `n`. This integer made of sevens and zeroes can quickly get gargantuan even for relatively small values of `n`. For example, for `n=12345`, the smallest working seven-zero number that is divisible by `n` consists of whopping 822 copies of the digit seven, followed by a single zero digit tacked to the end.

This method should find and return the smallest seven-zero 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 digit lengths (that is, how many digits the number contains) of the number. For each digit length, the inner `for`-loop iterates through all legal sequences of consecutive sevens followed by a sufficient number of zeros to make that number to have the desired digit length. Keep going up until you find the first seven-zero number exactly divisible by `n`. Since your loops iterate through the seven-zero numbers in ascending order, the number found this way must be the smallest such number that satisfies the problem requirement.

Furthermore, you can utilize an additional theorem proven in the same book to speed up your search. Unless `n` is divisible by either 2 or 5, the smallest seven-zero number that is divisible by `n` is guaranteed to contain only sevens, and no zeros. This realization should 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 that goes through the numbers 7, 77, 777, 7777, 77777… in search of the answer. On the other hand, numbers that contain the powers of two and five as their factors require seven-zero numbers whose tails of zero digits in the end grow along the exponents of these two little prime powers.

Lab 0(F): Two Branching Recursions

JUnit: P2J6Test.java

In most of the mainstream computer science education at university level, teaching of recursion tends to nerf down this mighty blade to cut through exponential possibilities. Recursion is used mainly as a toy to simulate some linear loop to compute factorials or Fibonacci numbers. This gains nothing over the everyday alternative of ordinary for-loops, the way that any competent coder would have automatically done it anyway. This also tends to leave the students dazed and confused about the general usefulness of recursion, since from their by that time glazed-eyed point of view, recursion only ever seems to be used to solve toy problems in a needlessly mathematical and highfaluting 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.** This remarkable ability allows a recursive method to explore a potentially **exponentially large branching tree of possibilities** and **backtrack on failure to the previous choice point**. Even better, the recursion can be trivially modified to either find the first branch with a working fruit hanging from its end, or alternatively collect such fruit from all branches. In this spirit, this last transition lab uses branching recursions to solve two interesting combinatorial problems. So, you handsome maniac you, without any further ado, create a new class `P2J6` to write this lab's two methods into.

```
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 of two cubes $11^3 + 5^3$. This method should return the list of these distinct integers as an object of some subtype of `List<Integer>` that prints out as `[11, 5]`, the elements listed in descending order. If there is no way to break the given integer `n` into a sum of distinct cubes, this method should return the empty list. Note that for the purposes of "gleaming the cube" by pushing your limits to the edge as if this were some energy drink commercial, the **empty sum** and a **singleton sum** are still sums in the same sense that zero and one are integers, as counterintuitive as this might seem. For example, the integers 0 and 8 are representable as sums of distinct cubes as the empty sum and the singleton sum 2^3, respectively.

Some integers can even be broken down into a sum of cubes in different ways. For such cases, this method must return the breakdown that is **lexicographically highest**, that is, starts with the largest possible working value for the first element, and then breaks down the rest of the number into a list of distinct cubes in a similar fashion. For example, when called with `n=1729`, the famous Ramanujan taxicab number, this method **must** return the list `[12, 1]` instead of the list `[10, 9]`, even though $12^3 + 1^3 = 10^3 + 9^3 = 1729$ just as well. This constraint may initially look scary, but you can really just *fuhgettaboutit*, since its enforcement is trivial by arranging the loop inside the recursive method to scan through the current possibilities from highest to lowest. As with such recursions, you should again use a private helper method that accepts additional parameters that were not present in the public method.

(To allow the search to cut off in some situations where continuing the search would be futile, it might also be of interest to know that the sum of cubes of integers from 1 to k can be computed quickly without loops with the formula $k^2(k+1)^2/4$, a polynomial of degree four.)

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

This helper method receives the two extra recursion parameters `c` and `soFar` from the original method. The parameter `c` contains the highest integer that you are still allowed to use. Initially this should equal the largest possible integer whose cube is less than or equal to `n`, easily found with a while-loop. The parameter `soFar` contains the list of numbers that have already been taken into the list of cubes so that they will be available once the recursion has returned.

The recursion has two base cases, one for success and one for failure. If `n==0`, the problem is solved for the **empty sum**, so you can just return `true`. If `c==0`, there are no numbers remaining that you could use, so you simply return `false`. Otherwise, try taking `c` into the sum, remembering also to add `c` to `soFar`, and recursively solve the problem for new parameters `n-c*c*c` and `c-1`. If that attempt was unsuccessful, 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`, its caller can also immediately return `true` without exploring the remaining possibilities. This first success, guaranteed to be the lexicographically first one by the order these alternative solutions are explored, 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 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 three 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 JUnit testing, your method must return this list of strings in alphabetical order.

Following the exact same principle as in the previous problem of adding up distinct cubes, this recursion is also easiest to implement with a private helper method

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

that takes two additional parameters `soFar` and `result`, where `soFar` is the partial string constructed down the path to this point (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 `String` instance method `endsWith` might come handy in this.) The base case for success is the case where `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 character to the `soFar` string before the recursive call, and removing it from the string after that recursive call.

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. Even if this method finds a solution right away, it still has to keep chugging through the entire tree of branching possibilities to glean the successes from the entire search tree.

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

JUnit: P2J7Test.java

On the day of creating this lab, your instructor had to wait one hour in line in cold winter outside Costco during the early stages of the COVID pandemic isolation measures, unsure of what the future would bring. This wait gave him ample time to think up a bunch of new problems (this entire document used to consist of only about twenty problems before the whole world 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** (often called "**deque**") when there is no need for random access into the list, but the queue is accessed only from both ends. (As an alternative, you could also try out the actual `ArrayDeque<E>` class intended to be used as a proper deque.)

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 such parts will never be touched by either rust or moth.)

Most of the time, double-ended queues are used as ordinary "**first in, first out**" (FIFO) queues so that elements always enter the queue from the rear to eventually exit from the front in the exact order that they came in. But as long as it costs us nothing to have the ability to potentially do something, we might as well keep that possibility in our sleeve!

The first problem (no doubt at least subconsciously inspired by the long wait in the Costco line), takes us to the world of the ancients where a bunch of zealots (yes, Lana, *literally*, look it up) had been surrounded by Romans with no escape available through the front or the back door. Preferring mass suicide to capture and torturous death by crucifixion, these men arranged themselves in a circle and drew lots for a random starting location. Counting from that location *k* steps ahead each time, the… man… who… was… *it!* was killed and removed from this grim game of eeny-meeny-miney-moe. The last man standing was expected to fall on his own sword in a swift and gentlemanly fashion to join his fallen brothers. Josephus, our lovable wandering hero who had become entangled with this bad crowd, had other plans to escape. After some quick mental arithmetic, he managed to finagle himself to this last surviving position to tell this tale. Can you do as well as Josephus under this considerable pressure and time limit?

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, counting initially starting from the first man. This method should create another list that contains the men in the order that they were eliminated. 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 allowing the parameter type to be an arbitrary `List<T>`. As explained in the lecture about generics, this is not at all the same thing as defining that parameter type to be `List<Object>` ! A polymorphic method that expects to receive a `List<Object>` argument will not be able to handle an argument of type `List<Person>`, whereas a method that expects a `List<T>` argument can do so easily.

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"]`. Shirts or skirts, none will be spared!

Our second example of patiently waiting in queue 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 the part of computers (in fact, that is what the word "computer" originally meant as somebody's job title), we surely want to optimize every algorithm to reach the answer with the smallest number of computations.

Every ten years, the United States of America is constitutionally obligated to run a nationwide census, so that the seats to the congress (more precisely, 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 closest integer does not work, because this method would not preserve the total number of congressional seats in the entire nation. How can you tell which states are you supposed to round up, and which ones you round down? Paradoxically in situations of this nature, even though integers in general are much easier and more accurate to perform calculations than floating point numbers used to approximate real 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 when arbitrary floating point numbers are allowed in the solution.

Fortunately, the equivalent Huntington-Hill method will solve this integer programming to minimize the overall **quantization error** caused by rounding the values of the solution to lower precision. Each state constitutionally receives one initial seat, regardless of its population. The remaining seats are then distributed one seat at the time. The next seat always goes to the state with the current highest **priority quantity**, as computed with 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 the number of seats s is in the denominator of this fraction whereas the numerator P^2 remains constant, each additional seat always decreases the priority of the state receiving it, so that other states get to the head in this priority queue to receive seats only after California, Texas, Florida and New York have taken their first couple of dibs. (Sit still, Rhode Island, and patiently wait for your turn. You will eventually get that second seat that you are entitled to.)

The Wikipedia page gives a bit different formula for priority quantity with a square root. 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 care about their mutual ordering? This ordering of priorities is unaffected by the application of any **monotonic** function to these quantities, such as **squaring**. However, squaring these irrational roots back to integers allows us to deal with nothing but exact integers, instead of having to juggle around inherently imprecise floating point numbers!

Inside the same class `P2J7` where you wrote 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 counts of seats given to each state. To make results unambiguous for the JUnit fuzz test, whenever two states currently have the same priority quantity, the next seat is given to the state that is **earlier** in the `population` array.

To write this method, first add the class `Fraction` from the class examples project into your labs project, to let you perform your calculations with exact integer fractions **without any possibility of integer overflow or floating point rounding error**. Not even one floating point number, let alone any calculation involving them, should appear inside your method!

You should also note that even though the population of some state fits comfortably inside an `int` without any overflow, the square of that population will not be equally friendly once this population is in the millions, which surely is not an unrealistic case for electoral math! For this reason, these population squaring operations absolutely have to be performed using exact `Fraction` objects of unlimited range, not as primitive `int` values. Just like it is too late to close the barn door after the horse has escaped, it is too late to convert an `int` value into an unlimited `BigInteger` once the overflow has already taken place! For example, instead of trying to be concise, normally a virtue in this place, by writing a statement

```
num = BigInteger.valueOf(population[i] * population[i]);
```

where the `int` multiplication can overflow before conversion to `BigInteger`, split this into

```
num = BigInteger.valueOf(population[i]);
num = num.multiply(population[i]);
```

to force the integer multiplication take place inside the `BigInteger` mechanism where there is enough room for all digits. On that note, everybody stand up in attention, eyes front to receive the following extremely important instruction!

Just because an integer value fits into an `int`, its square might not do so. Just because the intermediate results of your computational formulas are not given explicit symbolic names in your code, it doesn't mean that those intermediate results somehow magically wouldn't exist, nor be subject to the exact same laws of computations as your named data items!

Would your instructor really be so mean as to design the method `testHuntingtonHill` to fuzz 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 equal populations that differ by a measly little 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 (each state identified as an integer as its position in `population` table) is to keep them inside a `PriorityQueue<Integer>` instance with a custom `Comparator<Integer>` object whose `compareTo(Integer a, Integer b)` method renders its verdict 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 `huntingtonHill` method, outside the nested comparator class. To find out which state gets the next seat, simply `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 has been to the fate of the free world and our ability to keep rocking in it, it also has a more mundane application in displaying rounded percentages based on a list of numbers from some real-world data. You know how you sometimes see a disclaimer phrased something like “Due to rounding, displayed percentages may not add up to exactly one hundred” under some table of numbers and their associated rounds percentages? Well, *dude*, just use the correct algorithm to display your rounded percentages, and the need for such disclaimers utterly vanishes! For example, to compute the percentages rounded to one decimal place, simply distribute 1,000 virtual “seats” among your data items. Each of these virtual seats corresponds to one *per mille*, one tenth of a percentage point (yep, those are an actual thing, as every Finn with a driver's license would surely know), to be displayed as the exact percentage share of that data item.

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 lab brings you two more interesting problems dealing with "powerful" integer sequences reaching high about the weight class of those primitive types.

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

Define two positive integers be "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 $t = 10$. Tolerance of $t = 100$ 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 powers to be within one thousandth of a percent of each other, which ought to satisfy even the most ardent six sigma advocate auditing this course from the business school.

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 the test cases given to your method.

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 `(pa,pb)` of the exponents that fulfill the above requirement. Java does not have tuples in the core language, so we sidestep this by using a two-element `int[]` object to hold the result. Note that the return type of this method is `void` so that nothing is actually `return`ed from this method. Instead, the method copies its answer into the two-element array that is handed 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 millions of times, or if only to simulate the behaviour of some Python function that returns a tuple. Instead of this 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, which ought to lighten the load on the JVM garbage collector at least back when the art and science of garbage collection techniques were not at the level where it is today.

```
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 denote unknown but **fixed** integers anyway, whereas the letters *i* and *j* denote integers that **iterate** through position **indices** of loops.) For example, the infinite sequence of octagonal numbers that springs forth from *s* = 8 starts with 1, 8, 21, 40, 65, 96, 133, 176...

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

`n`	`s`	Expected result (as `BigInteger`)
`5`	`3`	`6`
`27`	`4`	`25`
`450`	`9`	`474`
(ten billion)	`42`	`9999861561`
(one googol)	`91`	`100000000000000000000000000000000000000000000000000000041633275351832947889775579470433400300354421242035 6`

This problem is best solved 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)/2` 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 to continue. 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, 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 positions...

Lab 0(I): Squaring Off

JUnit: P2J9Test.java

The two problems in this lab produce results of type `boolean[]`, arrays whose each element is a truth value. Since these individual bits can be packed eight into each memory byte tight as sardines, the storage of these arrays as **bit vectors** will be as compact as it can be, not one bit or byte of storage wasted in taking these truth bombs to the enemy!

Create the class named `P2J9` in your labs project, and there the following two methods that each solve a problem that involves **squares** of positive integers. Given this upper limit `n` as parameter, the result of both methods is a `boolean[]` array of `n` elements that contains the answers for all natural numbers below `n` in one swoop. Compared to computing and returning the individual elements as requested by the caller, solving interdependent subproblems in bulk showers the benefits of Taylorism on us since many things are cheaper by the dozen. Unlike the industrial workers of the past, today's programmers prefer to sip their soft drinks in air-conditioned offices as the mindless machines perform those tiresomely repetitive tasks. Any decision or action done by a machine is always one fewer decision or action for us humans to concentrate our brains on.

```
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 natural 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 numbers 14, 11, and 6 that respectively correspond to subtracting the square of one, two or three from the current number. Subtracting the square of four or higher is not allowed, since this would make the resulting value to be negative. The player who gets to move to zero wins the game, since the opponent can no longer make any moves and therefore loses the match under the **normal play convention**.

Using the standard terminology of this sort of impartial combinatorial games, a state is **winning** or **hot** if there exists at least one move into some cold state, and **losing** or **cold** if no such winning move exists. This rule makes the state $n = 0$ 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 that state leads to a hot state, thereby granting the opponent the guaranteed eventual win. The infinite sequence of cold states of this game begins with 0, 2, 5, 7, 10, 12, 15, 17, 20, 22, 34, 39, ...

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. 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 have already been filled in…

When played under the **misère play convention**, the player who moves to the zero state loses the game. However, the strategy of playing the misère version of the given combinatorial game cannot in general be constructed by merely interchanging the roles of the hot and cold states of the original normal play convention game, illustrating again how reversed stupidity is not intelligence. Generally, it does not pay to be miserly in analyzing the optimal actions of actually being a miser; we eat to live, not live to eat.

Lab 0(J): Similar Measures Of Dissimilarity

JUnit: P2J10Test.java

Consider two **bit vectors**, that is, 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 **hypercube** of all 2^n possible vectors of truth values. If both bit vectors are identical, all these dissimilarity metrics will produce the answer zero. Otherwise the answer will be some positive quantity whose magnitude corresponds to the perceived dissimilarity between these bit vectors. Obviously, swapping the roles of the two bit vectors should not affect their dissimilarity. However, **permuting** the boolean elements of these bit vectors the same way in both vectors should not affect the dissimilarity between those two bit vectors.

In this lab you will have 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} tallied 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 other two counts n_{10} and n_{11} are defined symmetrically. Therefore 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 invoke as their first step. Having written that method, each method below 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 page 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 is to be implemented as a separate method inside the new class `P2J10` that you should create in your 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` of $n = 5$, along with their expected correct answers for each dissimilarity metric. To make the following table a "bit" more compact, the truth value arrays are displayed as bits so that `0` means `false` and `1` means `true`.

`v1`	`v2`	*Matching*	*Jaccard*	*Dice*	*RT*	*RR*	*SS*
`10101`	`10101`	0	0	0	0	0	0
`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 0(K): Suffix Arrays

JUnit: P2J11Test.java

Strings are a stark data structure that allows random access to the linear text stored within, the universal and portable representation of all information. Under the hood, strings are character arrays, but the primitive nature of such arrays is hidden behind a better interface of public methods of the `String` class. Surprising algorithms have been developed over the years to quickly search and analyze both the string itself and the semantic information encoded in it. Even the seemingly simple task of searching for a pattern inside the given text can be optimized in countless ways, let alone various higher level tasks that are performed on these string instruments.

Occasionally the string data structure can be **augmented** with additional data that speeds up operations that would be inefficient if given only the raw and unadorned text string itself. Even though this data in principle adds nothing new to the mix in that its structure is fully determined by the original string itself, having that information available can provide significant speedups as yet another form of **space-time tradeoff**. This lab showcases a simple but powerful suffix array method as a way to preprocess any text so that after the metaphorical cheque for the one-time payment of this preprocessing has cleared, all future searches of arbitrary **patterns** can be executed in time that grows only logarithmically with respect to the length of the text. This makes these pattern searches blazingly fast even when performed on the entire *War and Peace!*

Computed from the given `text` string with n characters, its **suffix array** is an n-element array of integers that contains the position indices 0, …, n-1 to the `text`. Since each position is stored in the suffix array exactly once, the suffix array is automatically some **permutation** of order n. The suffix array lists these n positions sorted in the **lexicographic order** (also called the "dictionary order") of the suffixes that start from each position. Note that the presentation on the linked Wikipedia page on suffix arrays uses one-based position indexing to the string, whereas here we naturally use Java's zero-based indexing to remain consistent to our principles.

For example, the non-empty suffixes of the string `"hello"` are `"hello"`, `"ello"`, `"llo"`, `"lo"` and `"o"`, corresponding to positions ranging from 0 to 4 in the original string. Sorting these positions to the lexicographic order of the corresponding suffixes gives these positions in the order `[1, 0, 2, 3, 4]`, which then becomes the suffix array of the original string. Since all n suffixes of `text` have a different length, the possibility of equality in their lexicographic comparison can never arise. This guarantees the uniqueness of the suffix array for the given `text`.

For reasons of simplicity and convenience, suffix arrays are represented in this lab as instances of some subtype of `List<Integer>`, despite the use of the standard technical term of "suffix array". Create a new class named `P2J11`, and there first the method

```
public static List<Integer> buildSuffixArray(String text)
```

that builds and returns the suffix array for the given `text`. In this lab, this can be done with the naive algorithm, since the structure of our test case strings guarantees that the worst case scenario

of this algorithm, a string whose all characters are the same, is never realized. The easiest way to implement this naive algorithm in Java should be to define yet another custom subtype of `Comparator<Integer>` whose method `compareTo` lexicographically compares the two substrings that start from the positions that it receives as parameters. In this method, first define the local variable `ArrayList<Integer> result`, and fill it up the brim with the integers from 0 to *n*-1. Then, just use the utility method `Collections.sort` to sort this `result` list with your custom `Comparator<Integer>`. This discipline should allow your `buildSuffixArray` method to be reasonably fast even when building the suffix array for the entire *War and Peace*, as performed by the JUnit test class using the `warandpeace.txt` data file.

Once the suffix array has been constructed for the given fixed `text`, it can be used to rapidly find all positions inside `text` where the given `pattern` occurs. These would be precisely the very same positions whose suffixes start with that `pattern`! Having already done the work of sorting these suffixes in lexicographic order in the previous preprocessing step allows us to use a slightly modified **binary search** performed to the suffix array to find the lexicographically first such suffix. Since all suffixes that start with `pattern` must necessarily follow each other as a bunch in the sorted array of such suffixes, looping through all these positions is a straightforward while-loop once the binary search has determined the first such position.

To achieve all that, write the second method

```
public static List<Integer> find(String pattern, String text,
List<Integer> suffix)
```

that creates and returns a list of positions of the original `text` that contain the given `pattern`, using the given `suffix` array to speed up this search. Note that this returned list of positions must be sorted in ascending order of the positions themselves, instead of the lexicographic order of the suffixes of `text` that start at these positions.

Lab 0(L): I Can, Therefore I Must

JUnit: P2J12Test.java

A jovial old grandpa mathematician straight out of central casting often appearing in Numberphile videos, Neil Sloane is behind the Online Encyclopedia of Integer Sequences, a wonderful tool for combinatorial explorations. This lab has you implement two methods to generate elements of two interesting integer sequences whose chaotic behaviour emerges from iteration of a deceptively simple rule. Both sequences are filled in ascending order in a **greedy** fashion, so that each element is always the **smallest** number that avoids creating a **conflict** with the previously generated elements in the sequence. (Also, before you start: `remySigrist` requires **bitwise arithmetic**.)

Note that being defined by mathematicians, these sequences start from the position one instead of the position zero, unlike how sequences work for us budding computer scientists who have to actually get our hands dirty and therefore prefer zero-based indexing. When asked to produce the first n elements of the sequence, these methods should create an array with $n + 1$ elements. The zeroth element that we don't really care about is set to zero, after which the actual fun begins.

```
public static int[] forestFire(int n)
```

Explained in the YouTube video "Amazing Graphs II", the first two elements are both equal to one. Then, each element `a[i]` is the smallest positive integer for which no positive integer offset `j` creates an **arithmetic progression** of three equally spaced elements backwards in the sequence from the position `i`. More formally, the difference `a[i]-a[i-j]` may never equal the difference `a[i-j]-a[i-2*j]`, even if your worst enemy got to pick the position `i` and the offset `j`.

```
public static int[] remySigrist(int n)
```

As explained in the YouTube video "Amazing Graphs III", this sequence devised by Rémy Sigrist assigns to every positive integer a natural number, called the "colour" of that number. After the first colour assignment `a[1]=0`, each following `a[i]` thereafter is the smallest natural number `c` for which the binary representation of any earlier position `j` that has already been assigned the colour `c` has any bits turned on common with the binary representation of the position `i`. This can be checked quickly with the expression `i&j == 0`, where `&` is the **bitwise and** operator of Java.

This problem can be solved with two nested for-loops. The outer loop iterates through the positions of `a`. The inner loop counts upwards through the colours until it finds one that does not create a conflict. However, you should not be a "Shlemiel" who uses a third level inner loop to determine whether the current colour candidate `c` creates a conflict. Instead, you should trade space for time with a second integer array `taken` whose each element `taken[c]` gathers all the bits that are on in any previous position that was assigned the colour `c`. Then, whenever you to assign `a[i]=c`, update this lookup table with the assignment `taken[c] |= i`, where `|` is the **bitwise or** operator of Java, and `|=` is its **assignment shorthand** analogous to `+=` and other such more familiar forms.

Lab 0(M): Tower Blocks

JUnit: TowersTest.java

The following problem is adapted from the problem "Towers" in CSES, the collection and online judge for coding problems. This site is run by Antti Laaksonen whose "Competitive Programmer's Handbook" is an excellent resource about the application of data structures and algorithms for competitive programmers of all ages and sorts, even if they only ever compete with themselves to become slightly better every day. However, this problem does not require any of the techniques explained in that book, or for that matter, in any other classic algorithm textbooks. Instead, this problem is an exercise in coming up with the efficient mechanism to build the solution within the given **constraints** that prevent most of your possible moves and that way gently nudge you towards the optimal solution. (Same as in all other walks of life, fences can often be your good and trusted friends, not mere hindrances that some unseen moron placed there only to annoy you!)

You are given a series of `blocks` as an integer array, for example `{14, 9, 12, 4, 7, 1}`. You must process these blocks in **strict order from left to right**, which means that the order of these blocks in the array actually does matter a lot. These `blocks` are used to build **towers** so that every block must become part of some tower. On its turn to be processed, a block must be placed either on the table to start a new tower, or on top of some **strictly larger** block that is currently on top of some previously created tower, making that tower one block taller. Again, there are **no backsies**; once you have placed a block somewhere, that block cannot later be taken out in style of jenga to be snuck into some more tempting and convenient location. As long as these rules are followed, each tower can end up being of any height, regardless of the heights of the other towers.

Your task is **minimize the total number of towers**. For example, the previous `blocks` can be arranged into two towers, but no fewer. One working arrangement makes the first tower contain the blocks `{14, 12, 7}`, and the second tower the blocks `{9, 4, 1}`. As you can verify, both of these towers contain their blocks in the same order as those blocks appeared in the original array relative to each other. The optimal arrangement of `blocks` into the minimum number towers is not necessarily unique. Your method therefore needs to return only the minimum number of towers, not any actual working arrangement of blocks into towers.

In your labs project, create a new class `Towers`, and there the single method

```
public static int minimizeTowers(int[] blocks)
```

that computes and returns the minimum number of towers achievable with the given `blocks`. Your solution should be sufficiently efficient to complete the pseudorandom fuzz test for thirty thousand randomly created `blocks` of pyramidally increasing lengths in at most a couple of seconds.

Lab 0(N): Substring Parts

JUnit: P2J13Test.java

The two problems of this lab concern Java strings and substrings. Contrary to how things tend to work in other languages such as Python, Java 7 took advantage of the immutability of its `String` type by having the new object returned by the `substring` method share the same underlying character array with the original string. Since the contents of this shared `char[]` will never change, no harm can result in having multiple `String` objects share that same array. Immutability, huzzah! This policy allowed the `substring` method to work in guaranteed constant time needed to allocate the object from the heap and initialize its fields, regardless of the length of the original string and the substring being extracted into a separate object.

On the other hand, the big downside of this policy is that every `String` object in Java must know its `length` and the starting `offset` from the beginning of its character array. This causes every `String` object in the heap to be eight bytes larger than would be necessary, creating an instance of a curious phenomenon for which we could perhaps coin the term **space-space tradeoff**. Furthermore, sharing character arrays between strings and substrings prevent garbage collection of the large character array of the original string that went out of scope, still kept alive by some small substring object that uses only a small part of that big character array.

In your labs project, create a new class named `P2J13`, and in there two `static` methods

```
public static int countDistinctSubstrings(String text)
public static String reverseSubstringsBetweenParentheses(String text)
```

The first method should count how many distinct non-empty substrings exist in the given `text`, and return that count. For example, the string `"lollo"` contains a total of eleven distinct substrings: `"lollo"` of length 5, `"loll"` and `"ollo"` of length 4, `"lol"`, `"oll"` and `"llo"` of length 3, `"lo"`, `"ol"` and `"ll"` of length 2, and `"l"` and `"o"` of length 1.

This problem could be solved in a straightforward "Shlemiel" fashion with two nested for-loops to iterate through all pairs of possible starting points and substring lengths, and using some kind of `Set<String>` to maintain the distinctiveness of the strings so discovered. Even though the constant factors in the running time of this method are small enough to make it fast when most of the substrings are distinct from each other and it become necessary to iterate over all of them anyway, this technique is subject to accidentally quadratic running time, provided that your worst enemy gets to choose the `text`. He can make the `text` consist of *n* repeated copies of the exact same character, the running time becoming painfully noticeable once *n* reaches into the thousands, as it will in the JUnit test for this problem. Such a string contains precisely *n* different distinct substrings, one for each possible length from 1 to *n*, but this algorithm would loop through a quadratic number of combinations to discover these substrings only to immediately throw away most of them!

A more complicated **adversary-proof** solution guaranteed to be invulnerable to accidentally quadratic running time, but pays this guarantee with a higher constant factor in its running time over average-case inputs, stems from the realization that the substrings of length *k* can be constructed from the substrings of length *k*+1, by removing either the first or the last character from each one of these longer substrings. Your method should therefore generate the distinct substrings in the descending order of length. Use two separate instances of `HashSet<String>`, the first to remember the known distinct substrings of the previous length *k*, the second to store the substrings of length *k*-1 that are currently being generated.

Initially, the first set is a singleton that contains only the original `text` itself. In each round of the loop, iterate through the known strings of *k* characters currently in the first set. Add to the second set the two substrings of length *k*-1 constructed by dropping the first and last characters of that string of length *k*. Once you have done this for every string in the first set, add the size of the second set to your current tally of distinct substrings found. You now have the green light to `clear` the first set, and exchange the roles of the two sets for the next round of the loop to generate all the distinct substrings of length *k*-2.

The second method of this lab also concerns substrings, but also illustrates the use and methods of the `StringBuilder` mutable string to efficiently construct a new string piecemeal by repeatedly `append`ing characters to the end, and modifying this string while it is under construction. Given a `text` that consists of letters grouped with possibly nested parentheses, this method should return a new string where the contents between each matching pair of parentheses have been reversed, leaving out the parentheses in this process. For example, given the string `"(ab)c(de)"`, this method should return the string `"baced"`.

Whenever the `text` contains nested parentheses, these substring reversals must be performed inside out. For example, the string `"(a(bc)d)"` should become `"dbca"`. The JUnit tester guarantees that the parentheses inside the `text` given to your method are always **properly nested and balanced**, so your method does not have to concern itself of handling malformed strings whose parenthesis structure is internally inconsistent.

To perform this operation efficiently, use a `StringBuilder` to accumulate the `result`, along with a `Stack<Integer>` instance to keep track of the positions of the left parentheses encountered along the way. Loop through the characters of `text` from left to right, and `append` each character that is neither kind of parenthesis to the `result`. At each open parenthesis, `push` the current length of the `result` buffer into the stack. At each closing parenthesis, `pop` the position of its left counterpart from the stack. Extract and delete the substring of the `result` from that position into a separate substring that you then reverse and `append` back to the `result`.

Wham, bam, thank you, ma'am... except that even after almost three decades of evolution of the Java programming language, its `String` objects still do not have the simple `reverse` method that we take for granted in basically any other programming language used for serious professional work. The canonical way to create a reversed version of some Java string object `str` is by going through the expression `new StringBuilder(str).reverse().toString()`.

In general, Java enjoys so few advantages over Python 3 that we are almost obligated to point them out every time, and the guaranteed constant time operation of the `substring` extraction certainly qualified. This idea could potentially be taken much further especially if the `String` data type were not defined to be `final`, but allowed different implementations to store the immutable text in different ways. Then, text editors and other programs that need immutable strings that are not merely sliced and diced into smaller pieces but also concatenated to create new strings from these pieces, could use the rope data structure where these operations work in guaranteed constant time, as illustrated in the labs 64 and 65 of this document, The price of storing additional references inside each rope node would be well countered by the fact that each piece needs to exist only once in the memory, regardless of inside how many larger concatenated pieces of text that piece has been used as a building block.

Lab 0(O): Chars To The Left Of Me, Integers To The Right...

JUnit: P2J14Test.java

The two methods of this lab continue on the theme of strings, but this time operating at the level of individual characters. Both of these problems were adapted from the ample offerings of LeetCode, one of the premier collections of online coding challenges and automated testers. The JUnit test for this lab also uses the text file `warandpeace.txt` as data source, so make sure to also copy this file in your project directory. Create a new class `P2J14` in your project, and there two `static` methods

```
public static int[] distanceFromCharacter(String text, char c)
public static String pushDominoes(String dominoes)
```

In the first method, you are given a `text` and some character `c` that is guaranteed to occur inside the `text` at least once, possibly more. This method should create and return an integer array the same length as `text` so that each element gives the distance to the nearest occurrence of the character `c` from that position. For example, the array returned for the argument `"hello world"` and the character `'o'` would be `{4, 3, 2, 1, 0, 1, 1, 0, 1, 2, 3}`. If the character `c` had instead been `'e'`, the returned array would be `{1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9}`.

Again, this problem could be solved in a simple "Shlemiel" fashion with two nested loops. The outer for-loop would iterate through the positions of the `result` array. For each such position, the inner while-loop would advance left and right from that position in lockstep until it sees the first occurrence of the character `c`. The grinding inefficiency of such approach can be seen from any `text` where the character `c` occurs as the first character, but nowhere else after that. The pain of the quadratic running time becomes evident once the length of the `text` gets in the thousands.

To solve this problem in linear time, we note that the values of consecutive positions in the `result` array cannot be just every which way but loose, but can differ from each other by at most one. This allows you to use a single for-loop to fill in two temporary arrays `left` and `right`, that will then be combined at the end of this method to produce the final `result`. Loop through the positions of `text` to fill in the `left` array to fill in each element `left[i]` with the distance to the nearest occurrence of the character `c` when you are allowed to go only left, but not right. To update these values in constant time, you should maintain an integer counter variable to keep track of this distance, initialized to `text.length()+1`. At every character other than the `c` that you are looking for, increment this counter by one, whereas at every occurrence of `c`, bring that counter back to zero. The current counter value becomes the value `left[i]` being currently filled in.

Filling the values of `right[i]` happens exactly the same way, except as a mirror image going from right to left. You can either do this in a second loop for clarity, or combine both loops into one for-loop with a bit of simple integer arithmetic for brevity . Once you have filled in both arrays `left`

and `right`, each element of the `result` array is the minimum of the `left` and `right` elements in that position.

(This same technique can also be used to solve a classic algorithm chestnut puzzle usually called something like "Trapping rainwater". The `left` and `right` arrays keep track of the tallest wall seen to that direction from each position, filled in with a similar linear time for-loop that remembers the tallest wall that it has encountered so far. The water pillar at each position is again the minimum of the two values of the `left` and `right` arrays for that position.)

In the second method, you are given a string that represents a line of `dominoes`. Instead of placing these dominoes neatly on the table so that the pips match between consecutive dominoes like a civilized person, we are being more whimsical and place these dominoes standing up to create a chain of falling dominoes, itself a well-known metaphor for various things in life.

In the `dominoes` string given as argument to this method, the character `'.'` denotes a domino standing still, the character `'R'` denotes a domino falling right, and the character `'L'` demotes a domino falling left. The simulation of this domino chain proceeds in discrete time steps. In each time step, every falling domino causes its neighbouring domino to that direction to also become a falling domino, provided that that neighbour was still standing at that moment. If a standing domino is simultaneously pushed right by its left neighbour and pushed left by its right neighbour, these two forces cancel each other out, and that domino remains precariously standing.

This method should compute the end result of the domino chain once the situation has become stable so that nothing changes in the single time step. For example, given the initial state `"R..."`, the result would be `"RRRR"`. Given the initial state `".R...L"`, the result would be `".RR.LL"`, the two chains from left and right meeting in the middle to leave the domino standing.

Lab 0(P): Two Pointers

JUnit: P2J15Test.java

The two methods of this lab demonstrate the classic programming technique known as "**two pointers**" that helps you not be a "Shlemiel" when solving various array problems whose structure happens to conform to the needs of this technique. (In this context, the term "pointer" refers to an integer index that can be used to access an array element in constant time, not literally the low-level pointer variable that stores that actual memory location.) Even better, this bad boy of good coding techniques comes with the built-in advantage of making it "♫ clear and rational to anyone who does not belong to the oppressor class ♫" that its running time is at most linear with respect to the length of the array! Once absorbed into your "array" of coding techniques, the technique of two pointers is a skeleton key that hinges open countless doors that the poor Shlemiel would spend an entire day of mindlessly running back and forth to squeeze open. A multitude of examples of this technique can be found by browsing the corresponding categories at LeetCode and CodeForces.

To search for a solution inside the array, the basic version of the two pointers technique uses two **inclusive** array indices `i` and `j`. These indices together define the start and the end position of the current subarray that is guaranteed to contain the solution, assuming that some solution to the problem exists inside the array in the first place. Such a bold **invariant** claim is trivially true at the start of the method, since the index `i` is initialized to the first position, and the index `j` is initialized to the last position of that array, together spanning the entire array outside which nothing exists as far as this method is concerned. From there, a while-loop chugs on until either a solution is found in the subarray defined by the current values of `i` and `j`, or these two indices go past each other like strangers in the night to establish that no solution exists. Each round of this while-loop is designed to move at least one of these indices towards to the other one, but in a disciplined manner that **maintains the invariant** by guaranteed that neither index can accidentally skip over a position that is needed for the solution.

The two pointers technique can be used for problems where this decision of which index to advance and how much can always be made based on the local information about the elements at positions `i` and `j`, possibly with some other state information that the loop maintains along these indices to remember some relevant aspects of the past that may affect the future decisions that lead to the final answer that we can confidently give to Regis.

Before looking at the two problems needed to solve for this lab, we demonstrate this technique with the classic example of **two summers**. From an array of integers guaranteed to be sorted in strictly ascending order, find two distinct elements a and b that add up to the given goal value $x = a + b$. Again, Shlemiel would brute force through this problem this by ignoring all the information available in that the array is guaranteed to be sorted, and use two nested for-loops to crawl through the $n(n\text{-}1)/2$ possible pairs of elements to find two elements that add up to x.

A better way to solve the two summers problem comes from the realization that when `i` and `j` define a subarray that contains the solution whenever one exists, comparing the sum `a[i]+a[j]` to the goal value x will always either find a solution, or give us an ironclad guarantee that moving

one of these indices one step towards to the other cannot possibly skip over the solution. For example, if the sum of these two elements is less than *x*, this means that even the largest available element `a[j]` was not large enough to reach the goal when paired with the smallest available element `a[i]`. This smallest element `a[i]` therefore cannot be part of any element pair that would constitute a solution, so skipping over this nail cannot possibly cost us the kingdom. Symmetric reasoning allows us to safely skip over element `a[j]` whenever this sum is greater than *x*. The complete method based on this idea should be close to the following:

```
public static boolean twoSummers(int[] a, int x) {
    int i = 0, j = a.length - 1;
    while(i < j) {
        int s = a[i] + a[j];
        if(s == x) { return true; } // Got it!
        if(s < x) { i++; }  // a[i] is too small to work
        else { j--; }       // a[j] is too big to work
    }
    return false;
}
```

For an *n*-element array, these two indices must come together after at most *n* - 1 rounds, forcing the loop to terminate after at most that many rounds have elapsed. Since the work done in the body of the loop takes a constant time, the running time of the entire loop is at most linear with respect to the length of the array `a`.

Enough talk, on to action. Create a class `P2J15` in your labs project, and there two `static` methods

```
public static int[] findClosestElements(int[] a, int x, int k)
public static int countSubarraysWithSum(int[] a, int sum)
```

The first method is given an array of distinct integers guaranteed to be sorted in ascending order, and to contain the element `x` somewhere inside it. This method should first find the location of the element `x` inside the array using binary search, the ur-algorithm of the two pointers technique where in each round, one of the indices can safely jump halfway towards the other index without the risk of jumping over the element `x` being searched for. However, be sure to implement your binary search correctly by realizing that the two index jumps are **not** symmetric! This sidesteps both of the two classic possible bugs of this algorithm; that one index jumps one step too far to potentially miss the element `x`, or that in the end both indices stay forever in adjacent positions because the computed size of the jump becomes zero.

As a sidebar, we all intuitively understand how sweet it is to keep things such as library books or house numbers along a street in sorted order, for fast retrieval. Yet, few people seem to be able to coherently explain why this be so. The reason becomes evident by comparing the behaviour of the binary search over a sorted array to that of the **linear search** over an unsorted array. The state of being sorted effectively boosts the power of the order comparison operator between the array elements to nearly divine unlimited power. Comparing an element in an unsorted array gives you

information only about that element and nothing else, whereas comparing an element in a sorted array gives you useful information about not merely that one element, but about all the elements that come before or after it!

Once you have found the position where the element `x` resides, your second loop should make these indices proceed left and right from that position, to collect the total of `k` elements whose values are closest to the element `x`, as measured by the absolute value of the difference of that element value and the element `x`. If two elements are equally close to the element `x`, one to the left of `x` and the other to the right (such as elements 0 and 10 when `x` equals 5), use the smaller one if you can take only one element in the result. For example, the four closest elements to the element 10 inside the array `{1, 2, 8, 10, 11, 13, 22}` would be `{8, 10, 11, 13}`. Note that your method must return these elements in sorted order.

(It is actually possible to modify the condition inside the binary search to directly find the first element to include to the result, instead of finding the element `x`, to simplify the second loop into a trivial copying of the *k* elements from that position. Students who are sufficiently clever to achieve this feat can consider themselves to have been given a gold star and a pat on the head for being such good coders. Small chances to optimize things can be hiding in surprising places under the surface.)

The second method receives an array `a` that consists of **positive** integers that have not been not sorted in any particular order. The method must count of how many contiguous subarrays that array contains so that their elements add up exactly to the given nonzero `sum`. For example, the array `{6,3,1,2,3}` contains three such subarrays whose elements add up to six; the subarrays `{6}`, `{3,1,2}` and `{1,2,3}`. Note how these subarrays can be partially overlapping, or even singletons whose element equals `sum`.

The Shlemiel way to brute force this problem would be to use three nested for-loops; the outermost level looping through all possible starting positions of the subarray, the second level looping through all possible lengths of that subarray, and to top off this idiocy, the innermost third level adding up the elements in that subarray. The better technique uses two pointers, but this time as a variation where both indices `i` and `j` start at the beginning, and advance towards the end so that the **forward scout** index `j` is always ahead of the **rear guard** index `i` trailing somewhere behind it, this rear guard never stepping past the forward scout. Once the index `j` steps past the end of the array, everything that is possible to be found has been found, so the method can terminate.

If the sum of the elements of the subarray from `i` to `j`, inclusive, is greater than `sum`, you can safely advance the rear guard index `i` (or `j`, when both indices are equal and thus define a singleton subarray) by one step to make the subarray (and therefore also the sum of its elements) smaller. When the sum of the elements of that subarray is less than `sum`, you can safely advance the forward scout index `j` by one step to make the subarray larger. When the sum of the elements of the subarray is equal to `sum`, increment your tally of such subarrays by one, and advance `j`.

To eliminate also the innermost loop of the Shlemiel solution, note how each subarray that you examine is always the same as the subarray that you examined in the previous round, except with either one element added to the end (when advancing `j`), or one element removed from the front

(when advancing `i`). You should therefore maintain the sum of elements in the current subarray in a local variable, and update that sum in constant time whenever either index advances; either by adding the element that the forward scout index `j` enters, or subtracting the element that the rear guard index `i` leaves behind. This allows your method to pass the mass fuzz test of the JUnit test class within at most the required second or two, instead of taking an unacceptable long time to complete the fuzz test.

Lab 0(Q): Aboutturn Is Playfair

JUnit: PlayfairCipherTest.java

With the exponential progress of computing technology, the Playfair cipher has become adorably inadequate for today's needs for encrypted communications. We can still appreciate its emergent complexity from the rules that are simple enough for the average person to use with nothing but pen and paper, given the secret **passphrase**. We could now pretend to be time travellers visiting the stressful and chaotic trenches of the Great War, or the vaguely steampunk parlour of some gentleman detective leisurely cracking the intercepted communications between the enigmatic Professor Moriarty and his colourful cast of henchmen known under the collective sobriquet "The Crime Chums". Systematically going through the combinations in the same cozy spirit as we would solve a Sudoku today, even as there is no shortage of messenger boys and other street urchins ready, willing and able to brute force that same task for a tuppence, our good detective slowly drifts off into a dream of Her Majesty's finest scientists one day building a steam-powered machine that would reckon these mechanistic calculations for him...

Meanwhile up here in the reinvigorated cyberpunk genre that has spilled over to our actual reality, the Playfair cipher has practical value as an educational exercise of two-dimensional array operations. In our variation of this cipher, we shall treat the letter J as being the exact same letter as the letter I, so that there are only 25 letters in the English language. The Playfair cipher uses a 5-by-5 grid of letters whose cells are filled with distinct letters in an order determined by the secret passphrase known to both parties who wish to exchange secret messages encrypted with this scheme. The grid is filled one row at the time while reading through the passphrase, skipping over the spaces, punctuation and every other character outside the 25 letters of our reduced English.

Each letter is placed in the current cell in uppercase at its first occurrence in the plaintext. The cursor is advanced to the next column, starting from the beginning of the next row once the current row has been completely filled in. At the end of the passphrase, the leftover letters that did not appear in the secret passphrase are filled in the rest of the grid in alphabetical order. For example, the passphrase "There will always be suffering!" would give us the following grid:

T	H	E	R	W
I	L	A	Y	S
B	U	F	N	G
C	D	K	M	O
P	Q	V	X	Z

The Playfair cipher assumes that the plaintext message contains only uppercase letters, such as `"WEWILLATTACKATDAWN"`. The plaintext is partitioned into pairs of letters, and each pair is encoded separately into a pair of ciphertext letters. However, when both letters in the pair would be the same letter, an extra letter `X` is inserted between them. (This being the stuffy nineteenth century,

the original plaintext never contains two consecutive occurrences of the letter `X`.) For example, the message `"JOLLY"` would be encrypted as if it had been `"IOLXLY"` all along, again treating the letter `J` as if it really were the letter `I`. If the plaintext message consists of an odd number of letters, an additional `X` is tacked to the end to dance with the last letter that would otherwise be left without a partner. For this reason, the plaintext message also cannot end with the letter `X`.

Thanks to this preprocessing, every pair to be encoded will consist of two distinct letters. To encode a pair of such letters located in the same row of the grid, use their neighbours one step to the immediate right, with the end of the row looping cyclically back to the start of that row. For example, using the above grid, the pair `SL` would be encoded as `IA`, and the pair `BU` would be encoded as `UF`. The rule of encoding a pair of letters located in the same column is simply the transposed version of the rule to encode two letters in the same row; each letter is replaced by its neighbour one step immediately below. For example, the pair `DQ` would be encoded as `QH`, and the pair `GW` would be encoded as `OS`.

The last possible situation is where the two letters of the pair are located in different rows and columns, such as `BV`. In this case, these letters are replaced by the two letters that complete the rectangle in the grid, using the original two letters as two opposite corners of that rectangle. To make the decryption unambiguous, we have to make a choice between two symmetric and equally correct ways to do this, and choose to encode each letter with the letter in the corner that is in the same **row** as the letter being encoded. For example, the pair `BV` would be encoded as `FP`, and the pair `GX` would be encoded as `NZ`.

That's it! That is the entire scheme that could be taught in fifteen minutes to any schoolboy and his little chums, and is now ready to be translated into Java code. In your project folder, create a new class named `PlayfairCipher`, and there the method

```
public static char[][] constructPlayfairTable(String passPhrase)
```

to create and return the 5-by-5 character array Playfair table that the JUnit test class will pass on as the `grid` to the two methods

```
public static String encryptPlayfair(String plaintext, char[][] grid)
public static String decryptPlayfair(String ciphertext, char[][] grid)
```

for encrypting and decrypting the **plaintext** and **ciphertext** messages. In spirit of the age where the Playfair cipher originated, our JUnit tester will encrypt the entire `warandpeace.txt` text file with that cipher. (Put that in your pipe and smoke it, Russian formalists.)

The decoding of the ciphertext back to the original plaintext is almost the same algorithm as the one used to encode the original plaintext into that ciphertext. In fact, you can extract the common parts of these two methods into a `private` utility method of its own, making the bodies of these two methods simple one-liners. In addition to the `text` and the `grid`, this utility method is given an integer `offset` that tells it how far to look at the neighbour in the same row or the column. For the

encoding, the `offset` is set to +1, whereas for the decoding, the `offset` is set to -1. The rest of the shared algorithm is exactly the same for both encrypting and decrypting.

(If you want to be really clever and have the remainder operator `%` work correctly for both cases to keep you wrapped inside the grid, the `offset` for decoding could be `+4` as well! Java uses the mathematically correct version of the remainder operator so that `-1%5` correctly equals `-1`, even though this result unfortunately falls outside the legal indices to the table. Python uses the mathematically incorrect version of the remainder operator that happens to work correctly for purposes of wrapping indices inside sequences. For Pythonistas, the expression `-1%5` equals `4`. The reader is invited to ponder this sort of "worse is better" phenomenon that is occasionally observed in other fields of human enterprise and endeavour.)

Despite its simplicity, Playfair cipher still illustrates some important ideas that are relevant even for modern cryptography. For example, the Kerckhoff's principle that states that all security of the system should rest on the secret key, in that even if every detail of the algorithm were leaked to the enemy, that knowledge would not help them decipher any of our ciphertext messages, seeing that their security depends only on the secret key without which the algorithm itself is useless. Systems that violate this principle and rely on "security by obscurity" can always be permanently broken with the ancient technique of "rubber hose cryptanalysis", and will not survive for very long in today's interconnected world. In a feat of usability design well before anybody bothered to think about such concerns of the servant class, the Playfair cipher is also insensitive to whitespace and capitalization. This greatly simplifies the communication, memorization and application of secret passphrases for us mere humans.

Lab 0(R): Frogs On A Box

JUnit: FrogCrossingTest.java

This problem is adapted from a problem in CodeForces, the premier Russian competitive coding problem collection and online judge. Even if you never participate in competitive coding contests, we can still partake in the most important contest of them all, trying to beat the yesterday's version of yourself. You never know when that extra coding strength and knowledge of little techniques will come handy in the future.

Inside a version of the classic video game Frogger updated for the needs of our course, an army of frogs is standing at the edge of a canyon, aiming to escape to the other side to avoid the fate of being slowly boiled in their tanks. The distance between the canyon edges is far too long for even the mightiest frog to levitate its way across. The good news is that a row of boxes has been magically suspended to float in the mid-air between the edges of the canyon. The bad news is that each of these boxes can be used only once, since each box vanishes the moment that some frog propels itself off it. Given the *x*-coordinate positions of the boxes and the jumping strength of each frog, how many frogs are able to cross the canyon successfully?

In your labs project, create a new class named `FrogCrossing`, and there the method

```
public static int maximumFrogs(int[] strength, int[] boxes)
```

The first parameter `strength` is an array whose length equals the number of frogs in the army. The element `strength[i]` gives the maximum distance between two boxes that the frog `i` can clear with a single hop. The strength array is guaranteed to be sorted in descending order so that the strongest frogs take their turn first, the way Nature and Nature's God dictated it to be. The second parameter `boxes` gives the horizontal coordinates of the boxes floating in the canyon, listed in ascending order. The first and the last element of this array are the fixed positions of the edges of the canyon that do not vanish when used by some frog.

For example, if `strength` equals `{7, 5, 4}` and `boxes` equals `{0, 3, 4, 6, 8, 10, 11}`, it is possible for the strongest two frogs to cross the canyon from the edge at $x = 0$ to the opposite edge at $x = 11$, with several different ways to achieve this. However, no matter how you try to finagle it, all three frogs will not be able to cross the canyon, as can be established with the following argument. The two weakest frogs will use up the boxes at $x = 3$ and $x = 4$, and then the boxes at $x = 6$ and $x = 8$. Since this uses up all the boxes between $x = 0$ and $x = 10$, the first frog can now only imitate Wiley E. Coyote. Having the strongest frog cross first changes nothing, since he will use up at least one box before the box at $x = 10$ that was necessary for the other two frogs to cross the canyon successfully.

This problem is best solved recursively with the aid of the aid of the `private` utility method that determines whether it is possible for the *k* strongest frogs to cross the canyon. The public method `maximumFrogs` should consist of a single loop that counts the values of *k* from the size of the army down to zero, and return the largest value of *k* for which the recursive method given below returns

`true` to indicate a successful crossing. If there is no solution using the *k* strongest frogs, abandon the weakest of those frogs as the Cold Equations of Nature dictate, and try to find the solution for the *k* - 1 strongest frogs. (This solution illustrates the general technique known as **iterative deepening** that often comes handy in recursive optimization problems of this nature.)

```
private static boolean canCross(int[] strength, int[] boxes, int[]
atBox, boolean[] used)
```

The first two parameters `strength` and `boxes` are the same as in the previous method, and their contents do not change during this recursion. The *k*-element array `atBox` gives the index of the box that each frog is currently standing on. (You should use the indices of the boxes directly instead of their *x*-coordinate values, to make the lookup of boxes and their successors faster.) At the top-level call, the values of the `atBox` array are all zeros, since every frog starts at the edge of the canyon. The fourth parameter is a truth valued array to keep track of which boxes have already been `used` by some frog so that other frogs can't hop on them any more.

The base case of this recursion is when every element of `atBox` equals `boxes.length-1`, for all *k* frogs having successfully crossed the canyon. Otherwise, find the frog currently on the box with the lowest *x*-coordinate. If it is possible for that frog to jump directly to the opposite edge, make it do so and update the `atBox` array accordingly, and try to solve the rest of the problem recursively.

If the leftmost frog cannot reach the opposite edge directly, use a while-loop to iterate through the unused boxes that it can reach with a single jump. For each such box, see what happens when making the frog jump on that box and update the `atBox` and `used` arrays accordingly, and try to solve the problem recursively for the new situation of frogs and boxes. If that recursive call returns `true`, your loop can also stop and return `true`, since we are looking for any one working solution. Otherwise, continue looping over the rest of the reachable boxes to if the leftmost frog jumping there would produce a more favourable result. Once you have tried jumping on every reachable box but the recursive calls from those jumps led only to failure, your method can also stop and return `false`. Since the hop sequences of the crossing frogs never intersect each other in their visited the floating boxes and can therefore be arbitrarily interleaved and still remain successful, there is no point looping over the other frogs in search for a solution. If some solution existed starting with some frog other than the current leftmost one, that solution would have been found by starting with the leftmost frog.

We shall leave it as an exercise for the reader to note other hard and thus useful constraints that can be used speed up this basic search. Your mind will answer most questions if you learn to relax and wait for the answer. In this kind of branching recursive searches it is hugely important to detect inevitable failures as soon as possible. In that frozen moment when everyone sees what is on the end of every fork, you should turn back and return to the previous level of recursion to try out some earlier alternative instead.

Lab 0(S): Hamming Center

JUnit: HammingCenterTest.java

The **Hamming distance** between two arrays of equal lengths is defined as the number of positions where those two arrays differ. The Hamming distance is easily computed with a single for-loop traversing through the positions of both arrays in lockstep, incrementing an integer counter whenever the two elements in the same position disagree. The Hamming distance is a proper distance metric, since every array is trivially at zero distance from itself, the distances back and forth between two arrays are symmetric, and these distances satisfy the triangle inequality.

For a collection of `boolean` arrays, we define its **Hamming center** to be a `boolean` array whose Hamming distance from all the arrays in that list is at most *r*, where *r* is the smallest radius for which some `boolean` array exists that satisfies this requirement. The Hamming center is still not necessarily unique; for example, both `01` and `10` can serve as Hamming centers for the arrays `00` and `11` with the radius of one. (We conventionally write out `boolean` arrays in a compact form of zeros and ones, since it's easier to justify and visualize the five truth values from `01101` than from `[false, true, true, false, true]`.)

This lab uses recursion to construct the **lexicographically lowest** Hamming center for the given array of `boolean` arrays and the desired radius *r*. In your labs project, create a new class named `HammingCenter`, and there the method

```
public static boolean[] findHammingCenter(boolean[][] bits, int r)
```

The first parameter `bits` is a two-dimensional array whose each row is an array of `boolean` values, representing the `boolean` arrays whose Hamming center we wish to discover. The caller guarantees that each row of this array has the exact same length, so that the concept of Hamming distance is meaningful in the first place. This method should construct and return the lexicographically lowest Hamming center whose distance from all the rows of the `bits` array is at most `r`. If no Hamming center exists for the given `bits` array and the radius `r`, this method should return `null` to inform the caller that they need to try a larger value of `r` next.

This method should first set up the necessary array objects `distance` and `center`, passed down in the top-level call of the following recursive utility method that does all the heavy lifting:

```
private static boolean findCenter(boolean[][] bits, int[] distance,
boolean[] center, int k, int r)
```

The parameters `bits` and `r` play the exact same roles as they did in the first method. The recursion should not mutate the `bits` array in any way as it fills in the Hamming center to the parameter array `center`. The parameter `k` indicates which position of the `center` array is being filled at this level of recursion. Since we are filling the `center` strictly from left to right, `k` will be initialized to zero in the top-level call to this recursive method.

The new parameter array `distance` maintains a counter for every row of `bits`. Each counter `distance[i]` keeps track of how many of the `k` bits already assigned to the `center` differ from the corresponding bits in the *i*:th row of `bits`. As soon as any one of these counters becomes greater than `r`, the current branch of the recursive search for the Hamming center can safely be deemed an inevitable failure. It would be pointless to examine all possible ways to complete the `center` array from the current prefix already known to be hopeless, so the method can just terminate and return `false`.

The successful base case for the search for the Hamming center is when `k==center.length` so that the entire `center` array has been filled in. The method can now return `true` to make the previous levels aware of the good news and pass it on to their own callers.

Outside these positive and negative base cases, this method should loop through the truth values `false` and `true`, assigning each truth value to `center[k]` on its turn. It is important that you try out these truth values in this exact order, `false` first and then `true` second, to ensure that the `center` that gets completed first will be the lexicographically lowest of all possible centers. After assigning the truth value to `center[k]`, loop through the rows of the `bits` array to **update** all the `distance` values for the rows whose `k`:th bit differs from `center[k]`, and try to recursively fill the rest of the `center` array from position `k+1`. If this recursive call returns `true`, the current level of recursion should also immediately return `true`, without erasing all the good work that is has achieved in the `center` array. Otherwise, **downdate** the values in the `distance` array back to the the previous values that they held before the recursive call. Once you have tried both truth values `false` and `true` to fill in `center[k]` without success, return `false` to inform the previous level that the current prefix of the `center` cannot be extended to a working Hamming center.

Finding the Hamming center is a surprisingly difficult problem, far deeper than it may first appear on the surface. In fact, this problem is provably NP-complete, as it belongs to the set of famous decision and optimization problems that, in some visceral sense of cosmic justice and order, all feel like there ought to exist some simple algorithm to solve them, and yet every known algorithm to solve these problems will spend an exponential time in the worst case to grind out the solution! These NP-complete problems share the curious property that their verification is much easier compared to solving them; if some hinky dude in a dark alley offers to sell you a solution, it is "no problemo" for you to test quickly whether the solution offered to you is any good. However, coming up with a working solution all by yourself is a Herculean task of finding a small needle in an exponentially large haystack, with no hope of a magnet to help you prune this search.

In an even more annoying fashion rarely seen since the days of Tantalus, all these NP-complete problems turn out to be the same problem wearing different disguises, and the ability to solve any one of them efficiently would directly translate into the ability to solve all of them efficiently! Since this ability would basically be the equivalent of Christmas coming every day, only an incurable optimist would wait for a modern Prometheus to ever bring down such an algorithm to us mortals and effectively make us, if not actual primordial gods, at least some obscure minor deities hanging around the Olympian lounges hoping for our turn to get noticed. (Good news is that as long as Moore's law will keep on keeping on, and all computing devices become connected into one giant

Computer, the ability to program this Computer will make its wielder essentially equivalent to Loki in the Marvel Cinematic Universe.)

This same asymmetry between the ease of verification and the difficulty of actually constructing a working solution shows up in many other facets of our reality. For example, consider how much easier it is to enjoy a good book or a piece of music that somebody else has already created, compared to how difficult it is to author a good book or a good song by yourself! Bridging the enormous and treacherous ocean that separates the balmy beaches of "P" of verifying that the offered solution is good, from the arctic shores of "NP" of actually being able to construct a working solution, is by far the most important open problem in theoretical computer science, and will surely remain so for a long time.

Lab 0(T): Clique Mentality

JUnit: CliqueTest.java

Consecutive integers from 0 to n-1 are used to represent some kind of **entities** in whatever **problem domain** we are currently interested in. Translated into integers, these entities are called **nodes**. Each node is not an island, but can be **pairwise connected** to any number of other nodes using symmetric pairwise connections called **edges**. Between any two entities, there either exists such a symmetric connection edge, or there does not exist such an edge. Once the essence of these entities and their relationships in the original problem domain has been boiled down to an abstract graph made of these nodes and edges and nothing else, whatever vibrations our mouths happen to make the air move to tell each other tales about about these entities and their pairwise connections become irrelevant. The mechanistic algorithms that operate on these abstract graphs to compute their various properties with no concern to the original problem domain never need to refer to those words anyway. Immaterial words do not magically affect the operation of the mindless machine that executes such algorithms with no concern to any big picture or such grand purpose.

A myriad of graph algorithms have been devised to efficiently compute all sorts of interesting and useful properties on graphs. An entire course on combinatorial algorithms could easily be filled with nothing but study of all sorts of interesting and clever graph algorithms! These properties of the abstract graph will then directly translate to the original problem domain to reveal the corresponding interesting properties of the original entities and their connections.

This lab is designed to give you a first taste of graph computations by having you construct the largest clique inside the given graph. A clique is a subset of nodes in which any two distinct nodes are connected to each other with an edge. In this lab, you also again get to practice an extremely useful style of branching recursion that not only achieves the goal in this particular problem, but generalizes to countless other similar problems that you might encounter in the future. The ability to systematically iterate over all possible subsets of the given set of elements is the heart of solving many combinatorial problems on graphs and other sets of things.

Instead of using an actual `Set<Integer>` implementation, we shall simply use an integer array of length m to list the elements of the subset of m elements. Since the order in which these elements are listed inside the subset makes no difference to the subset itself, so that `{1, 3, 5, 8}` and `{8, 3, 1, 5}` are two possible representations for the exact same subset, our recursion might as well fill in this array in ascending sorted order. This recursion will systematically visit all subsets for the given m, starting from the **lexicographically lowest** subset `{0, 1, 2, 3}`, and finishing with the **lexicographically highest** subset `{7, 8, 9, 10}`, for the purposes of this example now assuming that n = 11 and m = 4.

However, since we are only looking for those rare particular subsets that form a clique, the recursive construction of the current subset can **backtrack** as soon as some node is not connected to one of the nodes chosen into the subset in the previous levels of recursion, instead of keeping on the futile effort to iterate through all possible ways to complete the prefix into a complete subset only to always realize at the end that the constructed subset did not end up being a clique. In general, it is

always better to turn back sooner than later once you realize that the current approach is ultimately futile and doomed to fail no matter how you continue.

In your lab project, create a new class `Clique`, and there the static field

```
private static int[] bestSoFar;
```

to keep track of the best solution found so far in a form that can be seen by all levels of recursion, and the static method

```
private static int[] findFirstClique(boolean[][] adjacencyMatrix)
```

The parameter `adjacencyMatrix` is guaranteed to be an *n*-by-*n* grid of `boolean` truth values so that the element in the position `[i][j]` is `true` if and only if the nodes `i` and `j` are connected with an edge. Since the graph from which we search a clique in is guaranteed to be symmetric, the truth values in positions `[i][j]` and `[j][i]` will always be equal to each other. The adjacency matrix is a compact way to represent a graph whose nodes are known to be consecutive integers, needing only one bit of memory to encode the existence of the edge between every pair of elements, for a guaranteed constant time query to find out this fact.

The method `findFirstClique` should start by initializing `bestSoFar` to a new 0-element array, and then create an `n`-element array `clique` to be passed to the recursive utility method

```
public static boolean findFirstClique(boolean[][] adjacencyMatrix,
int[] clique, int k)
```

This recursive utility method fills in the `clique` array from left to right. The recursion depth parameter `k`, initially 0 at the top-level call, indicates which position of the `clique` array is being filled in the current level of recursion. The method assumes that the first `k` positions from `0` to `k-1` of the `clique` array have already been filled with some `k`-element clique whose nodes are listed in the ascending sorted order, and tries to systematically extend this clique into a larger clique.

The recursive method should start by checking whether `k > bestSoFar.length`, meaning that recursion has constructed a clique that is larger than the best clique found so far. In this case, the method should copy the `k`-element prefix of the current `clique` to become the new `bestSoFar`. With the aid of `Arrays` utility class, this feat is achieved with a one-liner assignment

```
bestSoFar = Arrays.copyOfRange(clique, 0, k);
```

After this check, the method should enter a for-loop to iterate through the possible nodes that can be assigned to `clique[k]` to extend the current clique, even when the current clique is the best one that we have found so far. (As in life in general, don't let the good become the enemy of the perfect.) Since the nodes will be assigned to `clique` in ascending sorted order, this for-loop can start counting through nodes up from the previous node `clique[k-1]+1`, or from zero when `k==0` so that there is no previous node assigned.

Since we are aiming to complete a clique that is strictly larger than the largest clique found so far, this loop should only count up to `clique.length-(bestSoFar.length-k+1)`, instead of all the way up to `clique.length-1`. As we let the best solution prune the search for later solutions, this upper bound might even end up being strictly less than `clique[k-1]+1`, so that this for-loop will correctly examine no nodes at all! The same will also happen when `k==clique.length`, so this recursive method does not actually need to first check that the position index `k` is still within the bounds of the `clique` array. This shaves off one redundant comparison for every recursive call executed during the search. A small optimization, yes, but still quite elegant, as making some code not exist at all is the ultimate elegance in coding. (In general, non-existence tends to be a highly underrated property of things.)

For each node in the range of potential nodes to be assigned to `clique[k]`, the body of the for-loop should first ensure that that node is connected to all of the `k` nodes previously chosen into the current `clique`. Once the current node has passed this hurdle, you can assign it to `clique[k]`, and try to complete the rest of the `clique` with a recursive call using the depth `k+1`. Once this for-loop has tried and tested all the relevant nodes in its range in this manner, the method has nothing more left to do and can return. Once the top-level call of the recursive utility method has returned to the first method, that method can simply return the `bestSoFar` array as its final answer.

Lab 1: Polynomial I: Basics

(Arooga! Arooga! Attention, young citizens of the future! 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, or the labs 64 and 65 to implement the `Rope` data structure for efficient string operations, or the labs 68 and 69 to implement an `IntervalSet` data structure for efficient set operations on sets whose elements tend to be contiguous. 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>`, the 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 univariate polynomials with integer coefficients, along with 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 comprehensively concise online resource for undergraduate algebra and calculus that your instructor has ever traipsed upon.

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 to make it worth taking the fishing trip even if you don't actually happen to catch any! Even though those advantages might not yet be fully evident, that should not be any reason to stop you from benefiting from them. (This is similar to how you still benefit from a refreshing glass of cool water on a hot day just the same, even if your knowledge would not stretch to give an entire lecture of the biochemical effects that this H_20 has 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}` to begin with, without those pesky and redundant higher order zeros causing confusion.

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 test 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, collisions will happen *especially* from perfect randomness, unlike the finger-quotes “randomness” haphazardly generated by humans whose brains evolved to be much superior in the task of recognizing and generating patterns rather than 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 regardless of the magnitude of that constant.

`public int getCoefficient(int k)`

Returns the coefficient for the term of order *k*. For example, the term of order 3 in the previous polynomial equals 5, whereas 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 for such 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 polynomial with coefficients `{42,-7,0,5}`, this method should return `68`.

Your method does not have to worry about potential integer overflows, since the automated test class has been designed to allow the final and intermediate results of this computation inside the range of the primitive data type `long`. Motivated students can also perform this evaluation with the Horner's rule that is both faster and more numerically stable than the naive evaluation. However, even if you use the evaluation formula where each term is computed separately, make sure that you are not some "Shlemiel" who computes every term such as x^{10} from scratch with a loop, even though x^9 from the previous round is right there and needs only one more multiplication to give us x^{10}.

Unless you can understand and explain without any hand waving all the complexities involved with floating point numbers and their arithmetic operations such as `Math.pow`, as a rule floating point numbers should be avoided in all applications that require their results to be identical and repeatable in all computing environments, the way that all integer arithmetic is guaranteed to be in Java. Control the matter, and you control the mind. After the great reset button of this world has been pressed down and kept there, the following Year Zero will be the Current Year forever. All bourgeois ideas of "positive zero" and "negative zero", and especially the ideas of "infinities" and that despicable traitor Emmanuel NotANumberstein who pollutes every computation that he infiltrates with his crimethink, will be eradicated not just from the dictionaries in our programming languages, but also from our minds.

Nothing exists except through mechanistic consciousness, as it is impossible to see reality except by looking through the eyes of the Compiler. In the dawning new age of enlightenment and borderless fluidity, every properly educated young citizen must tirelessly `/root` out and report any and all instances of such crimethink to proper authorities who are ideologically trained and equipped to permanently `#cancel` any toxic and reactionary ideas of numeric inequality sputtered from the keyboards of wreckers and data hoarders. If you want a vision of the future, imagine a boot loader stamping on human face… forever.

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 `Polynomial` type is intentionally 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 $-5x^{10} + 7$, both having the same degree of 10, produces the result $-x^2 + 3x + 7$ that has a degree of only 2. As the JUnit test class generates random polynomials and adds them together, occasionally some polynomial will be added to its own negation, producing the zero polynomial as the result. Your code must handle this important edge case correctly.

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

Creates and returns a brand new `Polynomial` object that represents the result of polynomial multiplication of `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.

Multiplication of two polynomials can cancel out some internal terms whose coefficients were originally nonzero in both polynomials. For example, multiplying x^2+3 with x^2-3 gives the result x^4-9 whose second order term ends up with a zero coefficient and vanishes from the result. However, unlike when adding two polynomials, the highest term can never get cancelled out this way because the product of two nonzero integers can never be zero. You therefore know the degree of the entire result right away once you know the degrees of the original polynomials.

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 only 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` dynamic type 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 universal and common enough to make it silly to force everyone to keep greasing this same old squeaky wheel, you can nowadays use the method `Arrays.hashCode` to compute a hash value for your coefficients, good enough for the government work. If the rest of your code is any good, kid, using this method instead of rolling your own hash function will scarcely make the whole thing any worse. (You connect the dots, you pick up the pieces.)

```
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 in student code at this point is to loop through the coefficients from lowest to highest, instead of looping through them down starting from the highest. This bug is particularly tricksy to pinpoint, 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 could the student know that if they missed it in the casual reading of the specification? (“Not fair, Not fair! It isn't fair, my precious, is it, to ask us what it's got in its nassty little arrayssess?”)

Lab 5: It's All In Your Head

Historically, **graphical user interface (GUI) programming** was the first **killer app** that propelled the entire oriented programming paradigm into the mainstream after its languish of two decades as purely theoretical academic research. (Most things in computer science are at least a decade or two older than most people might casually assume; this author suspects this same principle to silently hold behind the scenes in all walks of life, not merely in 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, thus becoming a living and breathing example of both Greenspun's Tenth Rule and the Blub Paradox.

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 most likely copy-paste a bunch of boilerplate code from the example class `ShapePanel` into this class.

Your class should contain a `private` field `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 use `implement MouseListener` instead of saying `extends MouseAdapter`, you need to also provide the empty implementations for all the other methods in that interface, even if those methods end up doing nothing at all. The `MouseAdapter` utility superclass provides these empty implementations for its subclasses free of charge, saving you some inconvenient typing while making the code a bit more readable. In our more enlightened age, all these methods could have been `default` methods defined in the interface itself with do-nothing empty bodies, but of course the design had to use the mechanisms that were available in the language at the time.

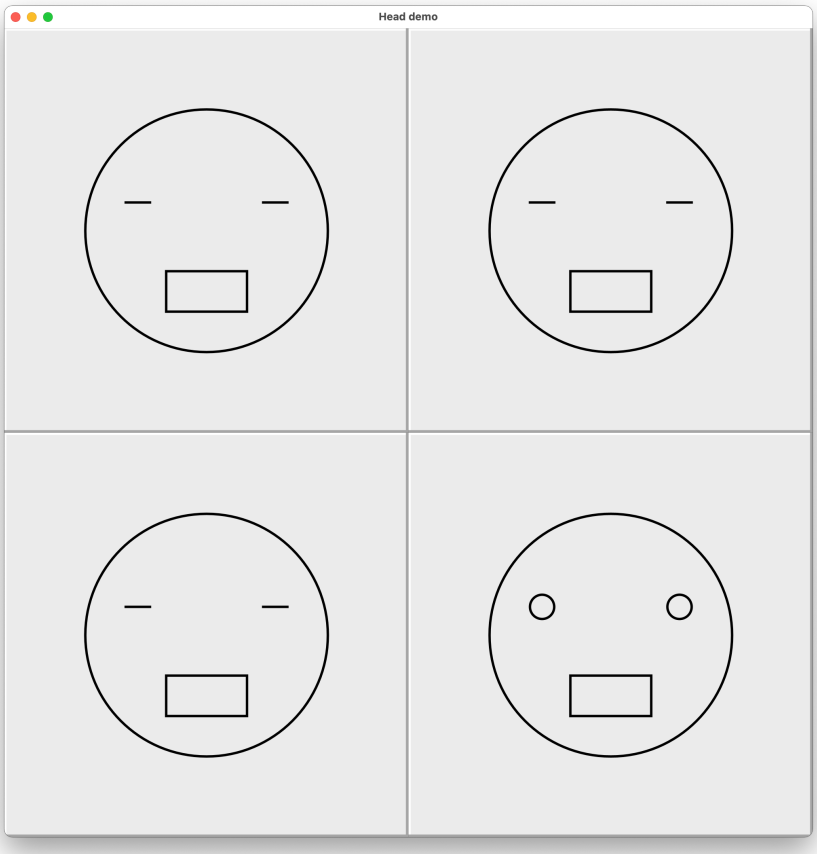
```
@Override public void paintComponent(Graphics g)
```

Renders some kind of abstract cartoon image of a simple human head on the surface of this component. If `mouseInside` equals `true`, the eyes of this head should be drawn to look open, whereas if `mouseInside` equals `false`, the eyes should be drawn closed as if sleeping. However, this is not an art class so this head does not need to look fancy and all coiffed up; a couple of simple

rectangles and ellipses suffice. Of course, students with a more artistic bent and interest might want to check out more complicated shapes from the package `java.awt.geom` to render a prettier image, or perhaps even using `Image` objects whose contents are read from some semantically appropriate GIF or JPEG file.

(Back in this author's salad days, GIF and JPEG were the only image formats that the average end user ever saw on their mighty screens of 640-by-400 pixels. You kids have your new fancy kinds of formats these days transmitted over network connections that are faster than a greased lightning and serve you high-definition video streamed in real time, while back in our youth, by Jove, an honest fellow had to grind his teeth through minutes of modem screech just to hope to catch an eyeful of some nice lady's bare ankle!)

To admire your interactive Swing component, 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, as in the screenshot below. Then you can wiggle your mouse cursor from top of one component onto another to watch the eyes open and close with your mouse movements.



Lab 6: Text Files I: Word Count

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". Who knows what kind of interesting correlations would a small army of future Davids find, armed with the slingshots of modern data analytics tools amply available for anyone who wants to use them to turn on the lights to yet another dark room in the Grand Hotel of Reality, usually revealing somebody who has been sticking his hand into somebody's pockets under the cover of darkness?

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 three consecutive **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 remain unchanged in all future subclasses of 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 at this same school 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 tool based on the following specification of its expected behaviour.

The existence of `wc` in all Unix systems by itself proves its usefulness as it solves an 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 should definitely happen to you during this process.

Create a class `WordCount extends FileProcessor<List<Integer>>`. This class should define three integer instance 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 a "loop and a half" situation where the first character has to be handled separately because it has no predecessor, we can all just pretend that the first character in `line` was preceded by some 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 can even name, or even see any need for their existence. Since that mythical cadre of Top Men in their horn-rimmed glasses and penchant for sticking tail fins on everything already did all the grunt work in tabulating all the properties of characters that are acknowledged to exist, 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 pummelled by user code for decades to various ends and purposes, all the edge case and corner case bugs that were lurking inside these methods have surely been brought to sunlight 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, I will similarly "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, as the event listeners pass along the player's 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 we want to be home 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 first sets the preferred size of this component to be `width`-by-`height`, as must be done for every new Swing component subtype. Then, 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 dream about launching a separate thread for each particle without waking up to profusely apologize about this transgression. 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)
```

Render the visual image `this` component by looping through particles, rendering each individual particle as a 3-by-3 pixel rectangle to its current *x*- and *y*-coordinates. As with the other labs whose result is a graphical component, students are free to try out variations to the artist detail, as long as the overall theme and spirit of the modified component remain as specified in this document.

```
public void terminate()
```

Sets the field `running` in `this` component to be `false`, thus 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 lecture example class `SpaceFiller` as a model of how to achieve 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 actually disposing that `JFrame` itself by calling its method `dispose`.

Try out the effect of different values between 0.0 and 10.0 of `BUZZY` to the motion of your particles. When `BUZZY` is small, the motion of each particle tends to maintain its current direction, whereas larger values of `BUZZY` shift their 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.

This sort of **random walk** simulation also leads to another interesting and educational thought problem for the more mathematically and statistically minded students to possibly ponder. When you set `BUZZY` to be near zero, you can see the particles soon leave the viewing peephole in their journey towards infinity in different directions. However, as `BUZZY` increases and the overall motion of each particle starts approaching the Brownian motion, the time that each particle spends inside the peephole increases. For different values of `BUZZY` ranging from 0 to 1, how long would you think you have to wait on average for the entire peephole to become devoid of particles that it was initially seeded with? What do you think this escape time curve would look like if you tried to plot it as a function of `BUZZY`?

Lab 8: Text Files II: Tail

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 protected void startFile()
```

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

```
@Override protected 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 help you to avoid being a "Shlemiel", you should 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 constant-time updating operations at both ends would be far more appropriate.

```
@Override protected 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. For example, the tail of the last ten lines of a file that consists of five lines would be just those five lines.

Lab 9: Lissajous Curves

In this lab you are going to write a Swing component that displays a Lissajous curve, a famous parametric curve whose simple periodic motion produces interesting self-intersecting visuals. The Swing component should allow the user to control the parameters *a*, *b* and *delta* that control the shape of the Lissajous curve by entering their values into three `JTextField` components placed inside this component. This lab is therefore an exercise in not just rendering curves and other shapes on a component, but also in positioning 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. Create a new 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 to values 6, 5 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 single instance of this class can simultaneously listen to all three text fields, since the reaction to the event is identical for all three sources of these 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 counter `double t` goes through the values ranging from `0` to `(a/gcd(a,b))*b` in suitably small increments, where `gcd` is the **greatest common divisor** of two positive integers. In the body of the loop, compute the point coordinates `x` and `y` with

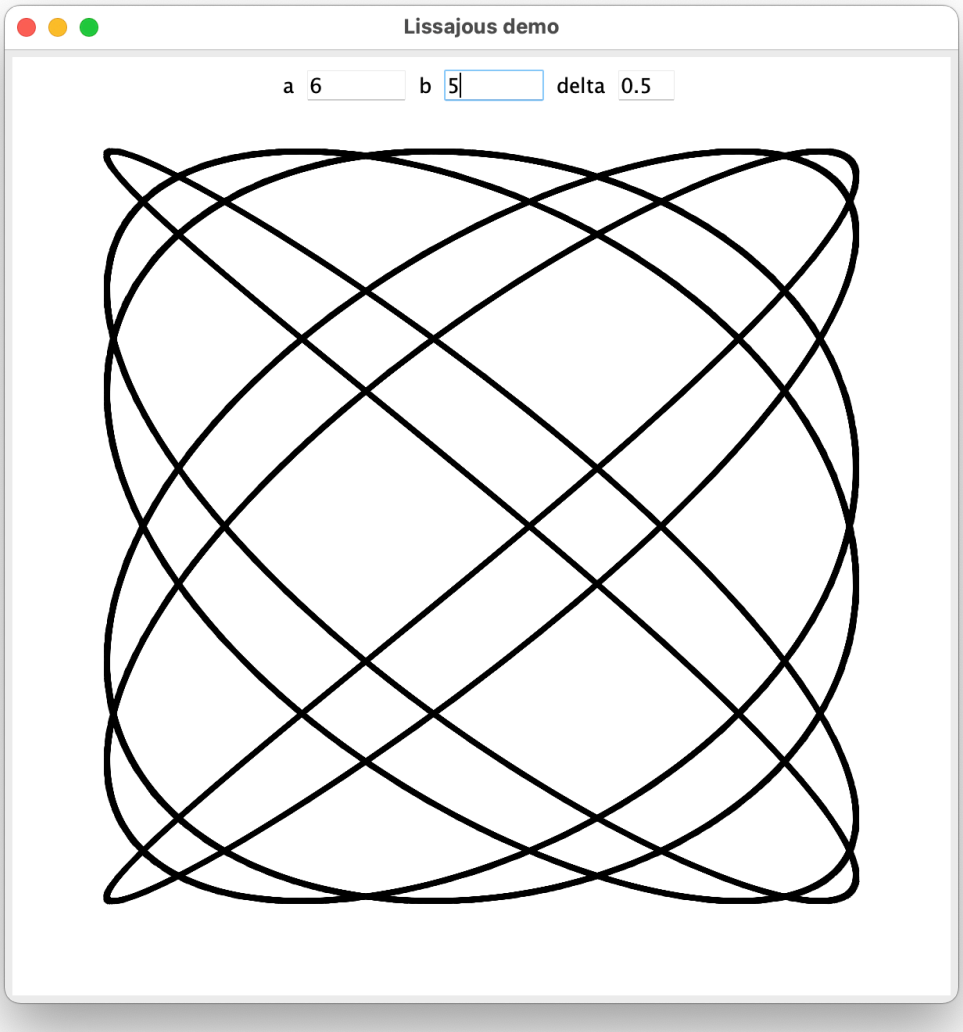
```
x = size/2 + 2*size/5 * Math.sin(a*t*Math.PI + delta);
y = size/2 + 2*size/5 * Math.sin(b*t*Math.PI);
```

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 transported inside some 1970's dystopian science fiction movie and trapped inside the lair of the mad professor Ilkkarius, create a separate class `LissajousMain` whose `main` method creates a `JFrame` that contains your `Lissajous` component. The end result with the previous initial values for *a*, *b*, and *delta* should resemble the following screenshot. You should verify that your code is correct by trying out other values for *a*, *b* and *delta* shown on the Wikipedia page "Lissajous curve" to verify that the displayed curves has the same shape as the example images over there.

Parameters *a* and *b* represent the frequencies of horizontal and vertical motions, so their greatest common divisor greatly affects the period of the resulting shape. This period is longest when *a* and *b* are **relatively prime** so that $gcd(a, b) = 1$. When *a* and *b* are equal, the resulting shape will simply be a circle whose elongation depends on the value of the *delta* offset between the periods of

horizontal and vertical motions. Using the value of *delta* of $\pi/2 \approx 1.5707$ makes the phases of *x*- and *y*- coordinates to be maximally apart (effectively turning the sine function of the first formula into cosine), whereas *delta* of 0 makes these two coordinates act in lockstep for a less exciting curve.

Motivated students can take on as an extra challenge to make the displayed image look smoother by eliminating some unpleasant visual jaggies, and possibly make this more artistic by using more complex curves and graphical shapes rendered to the positions of these "points going for a walk" around the two-dimensional plane. For example, instead of subdividing the entire curve into small line segments, you could subdivide it into more suave cubic curves whose consecutive pieces are guaranteed to connect together seamlessly at their shared end points, since the shared additional control points guarantee the tangent lines to match perfectly at these shared end points.



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 8 computation streams and their core operations!**

In this lab, we shall also briefly check out the **Java NIO framework** for improved operations on files than those that were offered in the old package `java.io` and the class `File` therein. Your methods receive some `Path` as an argument, guaranteed to refer to some text file in your file system whose contents your methods will then process with computation streams. In the JUnit test, this will again be the example text corpus of `warandpeace.txt`. Create new class named `StreamExercises`, inside which you write the following two `static` methods.

```
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 stream transformation method `filter` to keep only those whose `length` is greater or equal to the threshold value `thres`, and in the end, return the 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 aid of the method `split` in `String` with the word separator regex `"[^a-z]+"`. Then in the stages of the computation stream that follow, discard all empty words with another `filter`, 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). To wrap up this computation, `collect` it into a `List<String>` as the final answer.

As this new century shapes up to be a absurdist reboot of its predecessor during the worldwide pandemic, can rediscovery of principles of futurism be far away? After all, the original futurists certainly were the original "tech bros" of their time! At some point computer users will also notice that despite all the advances in computer processing power, their computers work subjectively no faster than they did two decades ago. What Andy Grove giveth, Bill Gates still taketh away…

Lab 11: And You Will Find Me, Prime After Prime

JUnit: PrimesTest.java

"There are only two hard things in Computer Science: cache invalidation and naming things." — Phil Karlton
"*All programming is an exercise in caching.*" — Terje Mathisen
"*Good programming turns caching into cha-ching!*" — Ilkka's corollary

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. It is sufficient to use the plain old **trial division** algorithm for this purpose. 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 any. To optimize this further, you realize that it is enough to test for divisibility of `n` only by primes up to the square root of `n`.

```
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 occasionally 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 to speed up our programs.

This class should maintain a private instance of `ArrayList<Integer>` in which you store the sequence of the prime numbers that you have already discovered, and **lazily expand** this list by generating and appending new primes to its end. Knowing that this list of all prime numbers up to

the current upper limit is sorted not only allows you to quickly look up and return the `k`:th prime number in the method `kthPrime`, but also to quickly iterate through all prime numbers up to the square root of `n` inside the method `isPrime` to look for potential divisors.

Furthermore, `Collections.binarySearch` can be used on the sorted list of these so far known prime numbers to determine almost instantly whether some given positive integer is a prime number. 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 probably be a good idea to have a `private` helper method `expandPrimes` that finds and appends new prime numbers to this list as needed by the `isPrime` and `kthPrime` methods.

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

Lab 12: The Second Hand Unwinds

JUnit: PrimeGensTest.java

In Python 3, all kinds of lazy sequences are easy to produce with **generators**, special functions that `yield` their results one element at the time and then 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 offer generators (at least not around the version that this author is familiar with), 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 collected on the Wikipedia page listing these special subtypes of prime numbers. Create a new utility wrapper 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 prime numbers** that are not explicitly stored anywhere, but are generated **lazily** whenever the user code request the `next` such prime number.

Each of these three classes should define the basic methods required by `Iterator<Integer>`. The method `public boolean hasNext()` should always return `true`, since for our purposes, all these sequences are infinite. (It is currently unknown whether the twin primes sequence is truly infinite, but there are enough twin primes for us to not run out during the JUnit tests.) The method `public Integer next()` generates the next element of that sequence.

First, to help you get started with the required three nested classes, below is a complete model implementation of two such example classes. The first one produces all palindromic prime numbers, and the second produces all **composite** numbers, that is, those that are not primes. Your three classes will have the same structure as these, but use different logic inside the method `next` to find the next prime number with the required property. (Note also the tactical use of both **pre-** and **postfix** forms of the ++ operator in `Composites`.)

```
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;
      }
    }
  }
```

```
}

public static class Composites implements Iterator<Integer> {
  private int curr = 4, k = 2, nextPrime = Primes.kthPrime(2);
  public boolean hasNext() { return true; }
  public Integer next() {
    if(curr == nextPrime) {
      nextPrime = Primes.kthPrime(++k);
      curr++;
    }
    return curr++;
  }
}
```

Having carefully studied both classes 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 for you to compare to the expected initial sequences given below. Once the first twenty or thereabouts special primes produced by your iterator match the expected sequence, it is nearly certain that the rest will also follow suit, same as even though the race is not always to the swift nor the battle to the strong, that is still the way to bet. (At some point incredible luck becomes indistinguishable from incredible skill, if only to mangle Arthur C. Clarke's famous observation about science and magic.)

`public static class TwinPrimes implements Iterator<Integer>`

Generates all twin primes; 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 can be expressed in the form `2*p+1` where `p` is also a prime number. (The smaller prime `p` that establishes the "safety" of the prime `2*p+1` in certain technical sense is then called a "Sophie Germain prime"). 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, ...

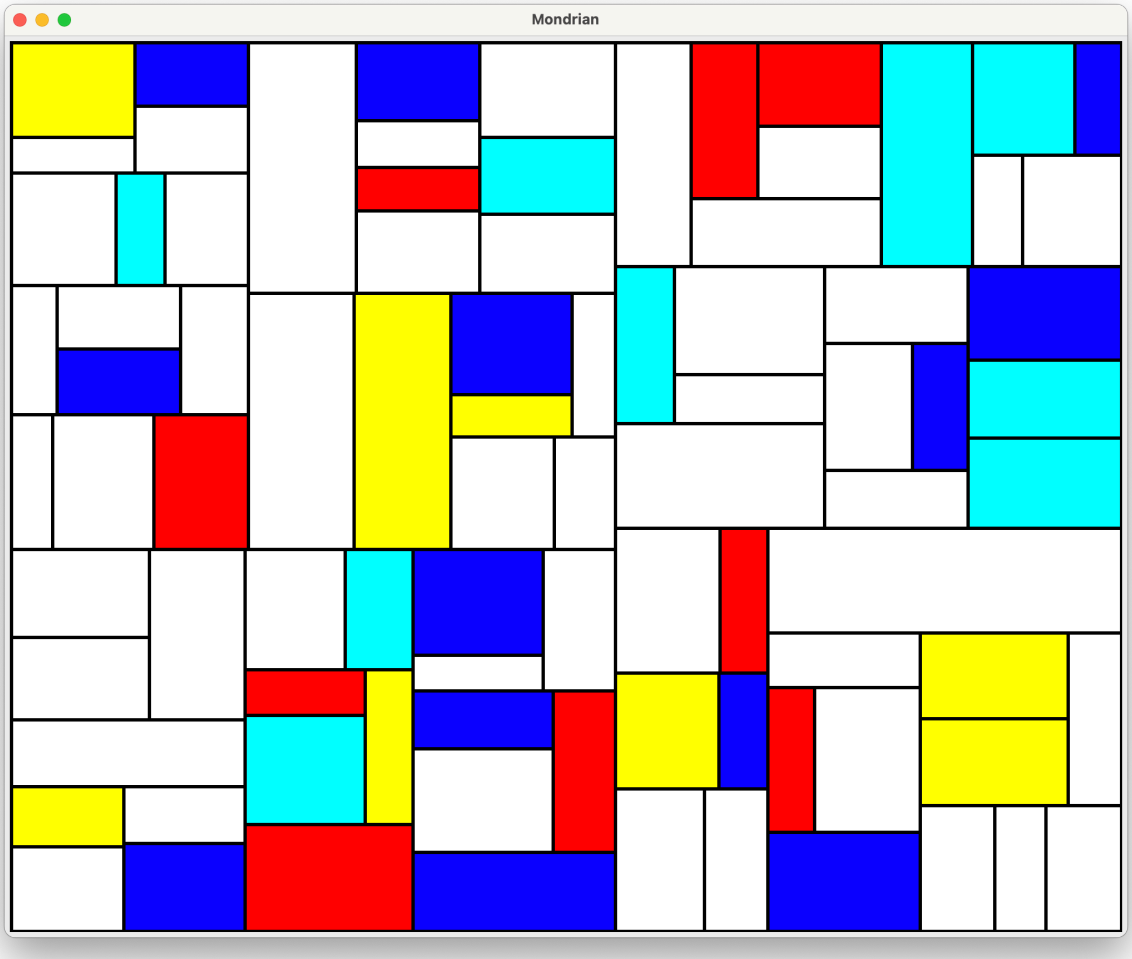
`public static class StrongPrimes implements Iterator<Integer>`

Generates all strong primes, primes `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 classic problem has resurfaced in Nifty Assignments, perhaps the time has come to revisit its timeless message. Especially since during these years in between, your instructor also managed to get himself hopelessly 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)`

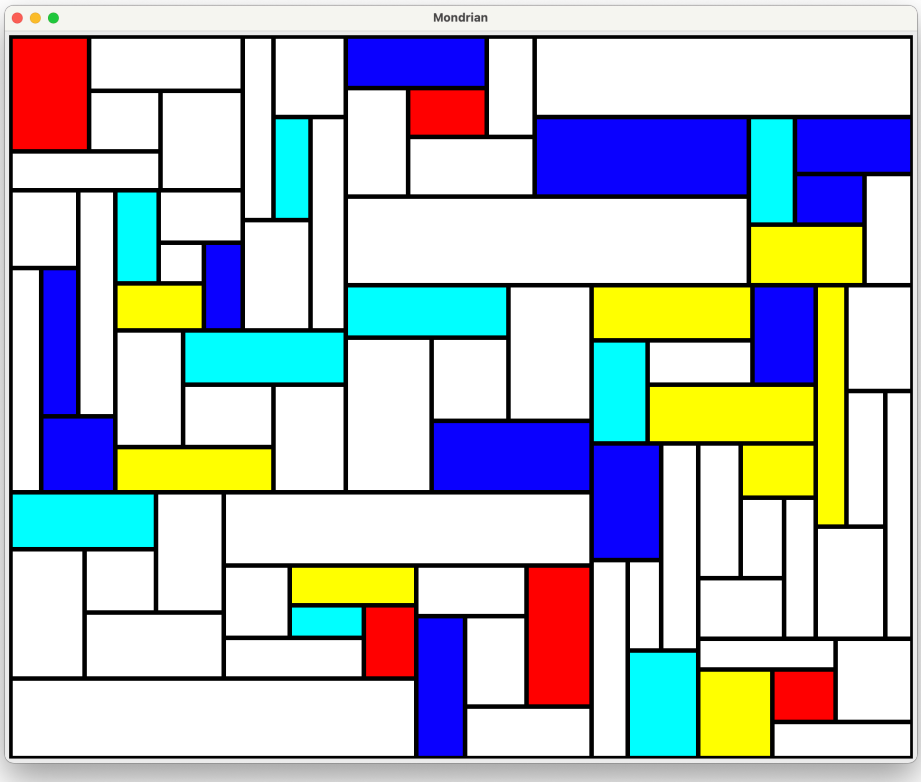
Recursively subdivides the given rectangle whose top left corner is in coordinates `(tx, ty)` and whose width and height are `(w, h)`. The base case of the recursion is when either `w` or `h` is less than `CUTOFF`, in which case this rectangle is drawn on the `Graphics2D` object provided. (The top-level 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, same as when somebody writes cursive by hand and has to squeeze the words in more narrowly at the end of the line to stay within the margins. In all walks of life, that elusive sweet spot at the middle way between the extremes of predictably solid and boring order opposite the screeching cacophony of randomness will usually be perceived by actual humans as the most aesthetic. (Reaching that sweet spot can be quite a tightrope act in practice.)

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.

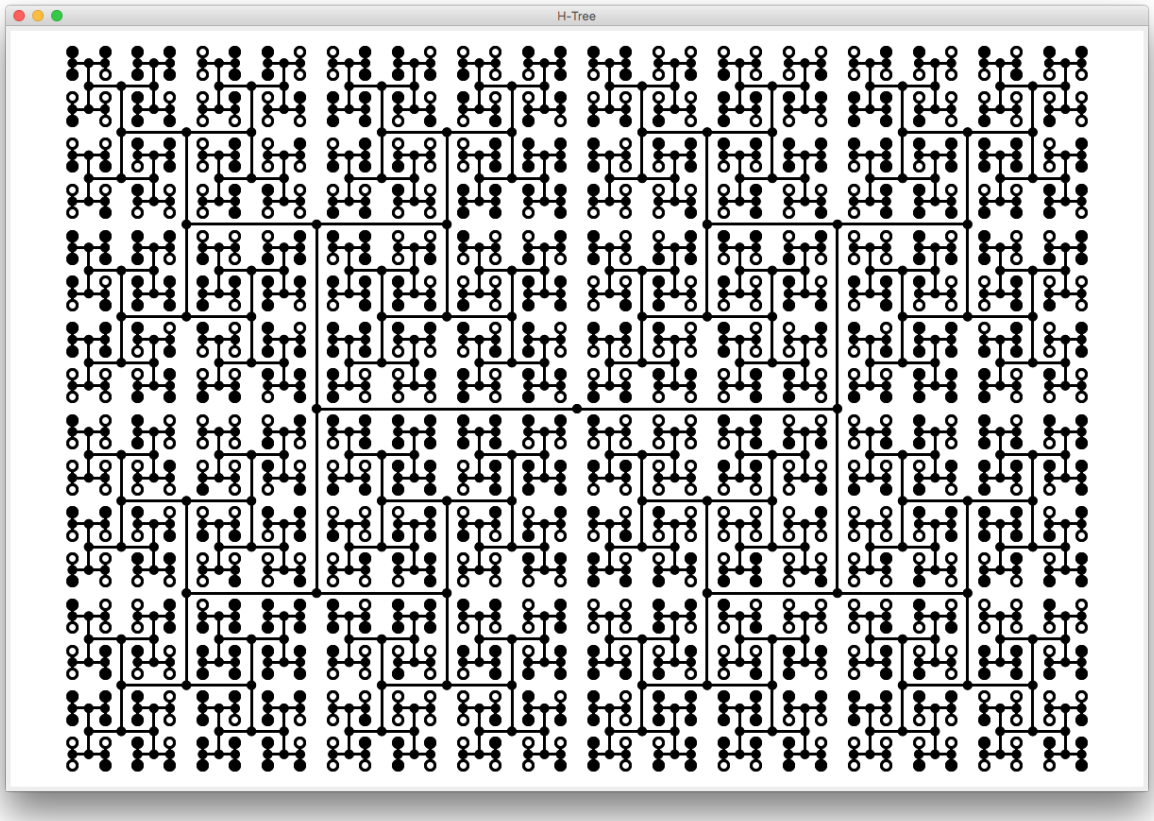
This random subdivision is not yet aesthetically the best possible due to a massive bias in the positioning of the subdivision walls inside the rectangle. Each subdivision will always contain exactly one straight **fault line** directly through that entire rectangle, which then continues to recursively 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 the fault lines. (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.)

Below, a more complex subdivision rule into five spiral-like pieces is mathematically guaranteed to eliminate such jarring fault lines, to make the resulting subdivision structure more organic and pleasant to the eye. More complex subdivision rules would converge towards random results that seem perfectly natural in the sense of being completely unforced and not in any way artificial, that way achieving the maximum artistic value reachable by a mindless machine in this domain. No law of nature or man will also stand athwart history yelling stop even if you use multiple different rules, one of which of which randomly chosen at each recursive subdivision call. No opponent, not even the Old Nick himself, can systematically defeat you, when you don't even know yourself what you path you will be taking in the future! We should also mention the completely different approach of Gilbert tessellations that produce even gnarlier tessellations, with the cost of some of these tiles becoming unpleasant thin shards.



Lab 14: H-Tree Fractal

In spirit of the recursive Mondrian subdivision from the previous lab, here is another recursive fractal problem of rendering the H Tree fractal on a Swing GUI 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 again adjust the style to be more aesthetic, provided that you maintain the basic nature and 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 first 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 same guarantee is always in effect for all objects created into the heap memory are of the Java Virtual Machine.

```
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.

Thanks to the magic of subdivision fractals, 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 from this exercise a funky case of “fractal fever”, a once fashionable ailment among the young but assumed to have gone the way of mange, grunge, the itch, the twitch, the thrush, the scoff and the rot since the early nineties, at least outside California (the epicentre of the original epidemic), might 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 head left, and the second call should head 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 in the recursive Mondrian subdivision, once you get this program to work, feel free to experiment to make the result look more pleasant. 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 some a flesh-and-blood human had drawn 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 in 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 the absence of that number. 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 the 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 that never appears inside the Zeckendorf representation, for a roughly 25% loss of storage efficiency. However, this limitation suddenly turns into a major advantage if you want to encode not just one integer, but an entire sequence of arbitrarily large `BigInteger` values so that we can still tell apart the individual elements. When a sequence of integers is encoded using the ordinary binary encoding for each individual element, this encoding must somehow be augmented to let the reader know where each number ends and the next number 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, in spirit of the **UTF-8 character encoding** discussed in the I/O streams lecture.

Using Zeckendorf representation, determining where the encoding of the current number ends for the encoding of the next one to begin is automatic. The otherwise unused bit pattern 11 acts as an artificial **separator mark** that tells the decoder that the representation of the current number ends there, so 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 forbidden second 1 acts as a virtual “comma” that separates that number from the following on. Even better, this encoding is **self-synchronizing** and thus more **robust**; the reader can recover from any incorrect bits the moment it arrives at the next separator mark, from where the rest of the sequence will be faithfully restored. (The UTF-8 encoding for Unicode characters famously also shares this convenient guarantee, over there enforced by the high-order bits of every payload byte.)

We next realize that the convention of writing digits in descending order of positions is but a societal agreement, so we all could have just as well agreed to write our bits, zits and other kinds of digits in the 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. Fibonacci coding of the given 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 virtual “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 even 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 "`1011010000011``10100001011`" (coloured here for extra readability), 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 reuse 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 just for convenience. 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.

```
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 test turns out an especially trivial task 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 value! Of course, to debug your code that is not yet passing this passive-aggressive mass test, you can create yourself specific desired test cases with the above online Zeckendorf calculator.

To gain even stronger appreciation about the general awesomeness of the Zeckendorf encoding, discrete math and number theory enthusiasts can check out David Eppstein's discussion about efficient integer arithmetic performed directly on the Zeckendorf representation.

Lab 16: Egyptian Fractions

JUnit: EgyptianFractionTest.java

As explained on the Wikipedia page "Egyptian fraction", ancient Egyptians had a curious way to represent arbitrary positive integer fractions a/b by breaking them down into sums of distinct **unit fractions** of the form $1/n$. Such breakdown could even be done in infinitely many different ways for any rational number, depending on what additional obligations 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 those fractions $a/b < 1$ that satisfy $a < b$ to allow 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 rational numbers needed here. Instead of reinventing that rusty old wheel all by yourself, you get to have more time to not merely walk but also do your arithmetic like an ancient Egyptian!

Create a new class called `EgyptianFractions` into your 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** originally proven to always 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 $1/n$ is less than 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.

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 there, restricting the breakdown to use only distinct unit fractions can cause quite a blowup in terms for even some seemingly simple fractions. As is typical with greedy algorithms, the result is not necessarily the simplest or the most balanced possible subdivision into unit fractions, since the last term can be a pretty thin sliver compared to the greedily chosen “thicc” terms that precede it.

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 from its slumber!) 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 split into in two terms $(a\text{-}1)/b + 1/b$. Repeating this step allows us to "carve" unit fractions off the bulk of the remaining fraction one at the time, as if carving delicious slices off the 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 $(a\text{-}1)/b$ until the entire number has been carved out so that only a zero remains. Often, a-1 and b have common factors to further simplify this task.

Comparing the breakdowns for some randomly chosen simple fractions, the greedy algorithm seems to typically produce fewer terms, but not always. For example, 5/91 breaks down greedily into unit fractions `[19, 433, 249553, 93414800161, 17452649778145716451681]`, whereas splitting produces a far more balanced subdivision `[23, 91, 2093]` for the same quantity. As so often in all walks of life, we have ample time to regret in leisure our greedy initial choices as they slowly take us towards their inevitable ends. On the entire way down there we can only dream of what could have been, had we initially stayed our hand and not shot that bloody albatross, but followed the dove and taken the road less traveled instead. (Making your choices inside a computer program instead of directly in the physical reality does not affect the poetic justice of the tragic consequences of such choices. As above, so below.)

To maintain the discipline of using each denominator at most once, we need to somehow handle the situation where the new slice $1/b$ is already in the 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, two equal unit fractions 1/10 and 1/10 can be replaced by a single 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 depending on the terms waiting inside the `buffer` and the needs of the rest of the number $(a\text{-}1)/b$ 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" although this famous result is actually quite not as trivial as it might initially seem, and certainly is not true just because of some hand waving in the likes of “But, like, it’s the *primes*, so how could it even be otherwise, *maaaan*, so much confusion here sweaty that I can't even!” 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 broken down an infinite number of different representations as sums of distinct 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 timeline of cyberpunk future so bright that we need to wear shades, anybody interested to learn more about this topic can start on the page "Egyptian Fractions" by that David Eppstein guy again. (“Once is happenstance, twice is coincidence…”) Most of the external links emanating from the page are as dead as the ancient Egyptian culture that they echoed, but at least "Algorithms for Egyptian Fractions" fortunately is still alive at the time of this writing. (On the page "Conflict resolution methods", Eppstein calls the algorithm “pairing” that we call “splitting”, and the algorithm called “splitting” on that page is then something else altogether.)

Lab 17: Diamond Sequence

JUnit: DiamondSequenceTest.java

The `Set<E>` implementations provided in the Java Collection Framework are flexible, but not the 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 dynamically adding new elements but disallow dynamic removals are called **monotonic**. Sometimes such structures can be implemented more efficiently than the full versions that allow `remove`. (Since each element removal decreases the **entropy** of that data structure, the `remove` operation must involve physical work that makes it inherently more complex than the entropy-increasing `add` and the nominally entropy-neutral `contains`.)

Further optimizations are possible when members of the set are known to be **natural numbers**. This turns out to be a rather common use case, given that everything inside a not just the physical computer but in the entire physical reality 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, and the growth of the sequence is inherently so modest that new heights are reached slowly and the sequence must therefore fill in the gaps that it leaves behind. For example, the Recamán's sequence that we met back in Lab 0(B). Membership information can in such cases be packed maximally tight 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 absolute position the relative zero index represents, and implement the methods `add` and `contains` to operate relative to this index.

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. The public method `void add(int n)`, well, adds `n` into the set. (There is no evident way to say that in any less convoluted fashion, and yet leaving in that phrasing somehow feels like giving up as a technical writer.) The public method `boolean contains(int n)` determines whether `n` is a member. No shrinking allowed, so no `remove` method exists in the 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 in the book "Mathematical Diamonds" by Ross Honsberger. Since that book deals with mathematics instead of computer science, it uses the conventional numbering of sequence positions starting from one, not from zero the way it always does for us propeller beanie hat code monkeys and web designers. The first element in the position $k = 1$ equals one. Each

position $k > 1$ contains the smallest positive integer that so far has not appeared anywhere in the sequence and makes the sum of the first k elements of the sequence to be an 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 is 2, since 1 + 3 + 2 = 6, and the deuce is the smallest of all unseen numbers that work. The sequence starts its eternal journey 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 mathematical depth or sophistication from this author's part), as a virtual iterator. Inside your labs project, create the new class

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

with the instance methods required in the interface `Iterator`:

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

Since this sequence is infinite, the method `hasNext` always returns `true` without further ado. 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` 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 for a moment try to wrap our heads around this puzzling fact that somehow follows from the construction of this sequence. (In discrete math terms, when this sequence is viewed as an infinite **permutation** of natural numbers to themselves, all cycles of this permutation have the same length of two!) The method `testSelfReferentiality` in the JUnit test class verifies this bold claim 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 actual seconds of "wall clock time", but is merely a number that can be positive, negative or zero, no different from the numbers used to denote the spatial dimensions. 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 leading there.

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, do not fail to appreciate the isomorphism between the concept troikas `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 done correctly and without pixellation for all possible image sizes, from a small icon up to a screen splash.

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. 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 an unnaturally 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 how the last frame is not `t=1`, since this would be an exact duplicate of the first frame `t=0` and show up as a small but annoying duplicate frame glitch in the animation. Since animated GIFs tend to use more space than necessary (the old format does not exploit any of the myriad statistical correlations between consecutive frames during the compression, but encodes each frame from scratch as a separate image), these days we convert any such animated GIF into various other formats to boil off about half of its bits. You kids these days have it so easy...

To make your periodic animation smooth so that there is no visible discontinuity at the times that are whole integers, 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...)

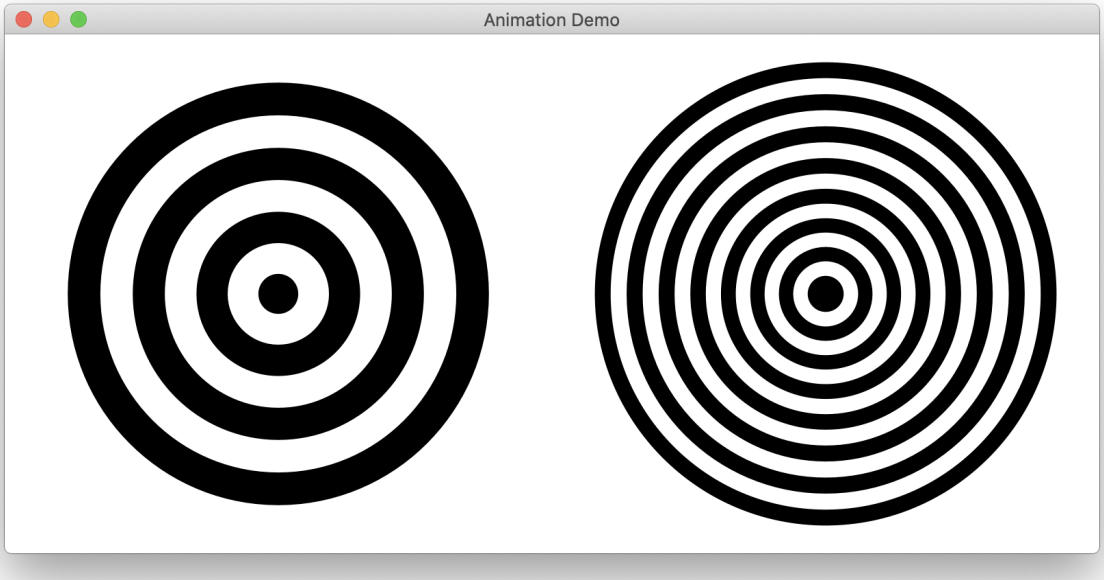
Calculations that consist of such periodic motions will greatly benefit from the existence of periodic functions, which most often are the basic trigonometric functions of sine and cosine. Since the purpose of this lab is not give you unpleasant flashbacks to the battlefield of trig back in the jungle of 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 as `default` method in `AnimationFrame` so that any class that implements `AnimationFrame` by default inherits these implementations. You can use these functions to calculate 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 the center point `cx`, simply compute it with the formula `x=circleX(cx, r, 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 of one.

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 for flexibility) 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 these shapes in their proper places on the component, 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 have also been tactically chosen to vanish to 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 source code, during the execution of compiled code all these variables contain only the numerical results produced by evaluating some of those expressions. Numerical results usually suffice for our needs (you only have to "show your work" in school exams, whereas the rest of reality acting as a grumpy boss is happy with just the final result and just enough paperwork to cover their arse if you catch a lawsuit), 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 the expression `0.1+0.2`, or comparing the results of `(0.1+0.2)+0.3` and `0.1+(0.2+0.3)`.

We are forced to define new data types as classes to represent some restricted forms of symbolic expressions as exact values. We already saw the `Fraction` type to keep rational numbers such as 1/3 in the exact form, instead of having to round them to 0.3333333333333333 so that this fundamentally incorrect result could be assigned to some variable to allow the rest of the computation to proceed. Some languages come with useful data types for **fractions** and other limited symbolic expressions out fo the box. Stratospherically high level languages such as **Wolfram Mathematica** even keep every expression in its symbolic form and operate on those forms, so that only the final result needs to be converted to its 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 once they are reduced to numbers. This despite the fact that inside the computer there exist 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 squarefree if n is not divisible by c^2 for any integer $c < n$. Some examples of such numbers that we would like 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 the two fours to be extracted from under the square root for a more elegant representation $28\sqrt{3}$ for the same irrational quantity.

(A relevant fact about integers at this point 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 integer under the root symbol to be a **square number**. Otherwise, that square root must be an **irrational** number, and an **algebraic** one at that, being a root of some quadratic equation.)

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 allow division as the fourth arithmetic operator while still keeping this system of number closed, we would have to extend the integers to rational numbers, perhaps with some sort of more sophisticated `Fraction` type that is able to handle the numerators and denominators as **symbolic expressions** 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 labs project. 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. (Two terms $a\sqrt{b} + c\sqrt{b}$ can always be combined into $(a + c)\sqrt{b}$ 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. Declare 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` instance the same way as Python does for its equivalent `dict` data structure. If only the rest of the Java language would also play ball here...)

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 stop that buck right there at your desk, so that the rest of your methods may assume that every term has a nonzero coefficient. (Same as those of the President, every word of the constructor weighs a ton.)

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 the biggie. The first `public` method for you to implement when creating a new data type is surely `toString()`. Most of the time that task is straightforward, so we might as well do all that 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 can claim without being run out of town tarred and feathered on a rail is being a veteran reader of the “Humor in Uniform" column in *Reader's Digest*.)

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

Because the automated JUnit tests record the results from this string representation, the string returned for each instance of `Distance` has to be unambiguous. The five rules that **you must follow in this lab** to guarantee that the outcome is both unambiguous and readable are:

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 these terms in ascending order.)
2. If the coefficient of some term is 1, that coefficient 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 now been field tested over thirty arduous years, is there any reason not to go with the tested and true? The following table shows a list of illustrative examples of the expected way to transcribe a `Distance` object, given the `coeff` map given to its constructor.

`coeff`	Expected result of calling `toString()`
`{}`	`"0"`
`{42=0,99=0,123=0}`	`"0"`
`{1=1}`	`"1"`
`{1=-42}`	`"-42"`
`{5=7}`	`"7Sqrt[5]"`
`{7=5}`	`"5Sqrt[7]"`
`{7=4}`	`"14"`
`{5=1,10=-1,15=1,21=-3}`	`"Sqrt[5] - Sqrt[10] + Sqrt[15] - 3Sqrt[21]"`
`{5=1,10=-1,15=1,20=-1}`	`"-Sqrt[5] - Sqrt[10] + Sqrt[15]"`
`{3=-5,7=11,6=-17}`	`"-5Sqrt[3] - 17Sqrt[6] + 11Sqrt[7]"`
`{10=-1,43=99}`	`"-Sqrt[10] + 99Sqrt[43]"`

Of course, your constructor of `Distance` will make sure to extract all possible squares out of these coefficients, rather than passing that buck to every future user of this class. (To add a touch of insult to that type of injury, remember to proudly proclaim "YAGNI!" before patiently explaining how designing your class to do the bare minimum just to pass the tests makes your class "simple", "common sense", and most importantly, "lightweight".)

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. Time has now come to perform computations on such numbers using 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.

Writing the methods of this lab, you are allowed to assume that the JUnit test class gives your methods 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 the `Math` utility class are guaranteed to throw an `ArithmeticException` whenever they detect an overflow. The absence of the said exception then lets you sleep sound in the knowledge that no overflow has occurred.

```
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 `add` and `subtract` will then 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 implemented easiest using an empty `TreeMap<Integer,Integer>` instance that will contain the result. That way, 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 b and d. Then, either create a new entry for bd inside your result map, or update its existing entry by adding the coefficient ac to its value, also correctly handling the edge case where that existing value is already equal to the negation of ac…

Lab 21: Symbolic Distances III: Comparisons

JUnit: DistanceTestThree.java

The previous two labs bestowed upon us the ability to perform arithmetic on `Distance` objects and print out the results in a human-readable format. The time has now arrived to make these `Distance` objects behave properly so that the existing classes in Java Collection Framework and elsewhere work properly with this new data type. In Python, this is achieved simply by defining some **magic methods** that correspond to the operators of the language. The following two 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 each numerical quantity representable within this system will have exactly one such representation. If two `Distance` objects have different `coeff` maps, the quantities that these objects represent are known to be different as surely.

```
@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 reliably compare any two arbitrary `Distance` objects in this 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 that internally uses base ten instead of base two to represent the decimal numbers, 0.1 + 0.2 actually equals 0.3 and not 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 exact! Whenever some `BigDecimal` operation produces an approximate result, the **rounding mode** determines how the value of the last included digit is determined 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. It is hard to know *a priori* how much precision the `MathContext` needs to have for this comparison to be accurate enough to settle the issue of who is truly the mightiest. Otherwise, two sums whose values differ by some microscopic amount will seem equal when viewed through the numerical equivalent of Coke bottle lenses.

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, from example 15. And so on. Simple order comparison between two sums of square roots of integers is computationally a far more difficult problem than most people would expect it to be! However, for the test cases emitted by our JUnit test class, this scheme to increase precision will eventually become sufficiently surgical to be able to tell apart close sums separated only by some subatomic margin that makes even obsidian razors seem dull by comparison.

Those interested to see more `BigDecimal` computations in action can check out our `Mandelbrot` example. This **fractal renderer** performs its computations with the `BigComplex` type defined to perform addition and multiplication of **complex numbers** to arbitrary desired precision. Akin to that old joke of how it is easy to sculpt a statue of an elephant ("Start with any old block of stone, and chip away every piece that doesn't look like it is some 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 in 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 strictly inside the Mandelbrot set, although one downside is that the computation never terminates as the border pixels are iterated forever in a vain hope of escape.

Lab 22: Truchet Tiles

For testing: TruchetMain.java

As explained in the Wikipedia page "Truchet Tiles", tiling the two-dimensional grid using four rotationally symmetric black-and-white tiles can produce surprising "op-art" geometric patterns. The four types of 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 anti-diagonal, and the parameter `parity` determines which side is black and which side is white. In your labs project, create a new class

```
public class Truchet extends JPanel
```

to make your class to be a full-fledged Swing component out of the box. Each instance of this class should then display the Truchet tiling defined by its 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>` to encapsulate a function that takes two `Integer` parameters and returns a `boolean` result, provide your renderer the parity and diagonal of each tile based on its coordinates `(x,y)`. This constructor should store its arguments into object fields, and set the preferred size of this component to `new Dimension(s*w,s*h)`.

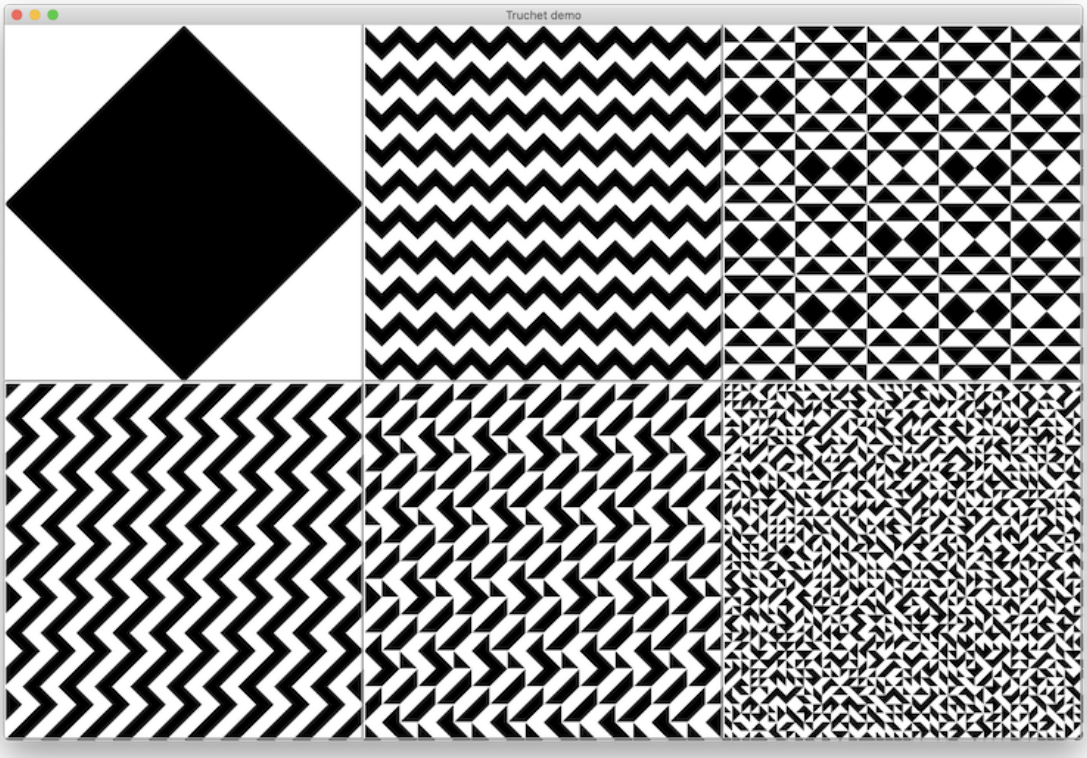
```
public void paintComponent(Graphics g)
```

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 assign the these colours based on the parity and diagonal direction of the tile. However, to allow both the student and the grader to agree at a glance 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 test that do their work even if we close our eyes and pretend that we are no longer in the room, in this problem our 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 right on the **wetware** that JUnit or any 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, similar to the vase illusion, as told in the voice of Alan Watts), 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. Even randomness is more pleasant after adding nonrandom things to balance it out so that the human visual system has something to latch on to interpret. (The TruchetMain.java class also has a method that uses Truchet tiling to showcase the use of basic punctuation characters of *War and Peace*, as inspired by the post "Punctuation in Novels".)



Once you get this component to work, you can also try out more artistic variations such as **Smith tiles** that use quarter circles instead of triangles to create a curved shape. Or create a **labyrinth** by using `diag` to draw a black diagonal line to separate the two white halves ignoring `parity` completely, in spirit of the classic Commodore 64 Basic one-liner. You could even subdivide some tiles into smaller tiles for a fractal subdivision effect, or explore the properties of random Truchet tilings. Only your imagination shall be the ultimate ceiling of your art, since in this course, the only law that we acknowledge and allow bind our hands 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, the unconventional mathematical genius who passed away during the COVID pandemic (xkcd tribute) one week before this lab was originally written. Game of Life is used in many introductory programming courses, including this one, either as an example or as a programming project for a good reason. This deceptively simple setup offers ample educational value not just for the first coding course, but far beyond that all the way up to the ultimate limits of computability.

However, just to be different from this well-beaten path, we shall instead implement a somewhat similar Ulam-Warburton automaton that produces its own distinctive fractal patterns of more crystalline nature. Besides, that Stanislaw Ulam fella was also a pretty important figure in applied mathematics and computing in his time, including the Manhattan Project. His life and creations deserve at least some attention up here in this cyberpunk science fiction future timeline that we have suddenly found ourselves living in even though 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 not just its standard staple of video phones, but also real time face substitution, subtitles and translation to another language elicit only a bored yawn. (Any readers who have a better explanation for this mass amnesia than the meek "a wizard did it" 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" representation allows the individual rows to be **ragged** so that don't all necessarily have to 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 compute the target position in the one-dimensional array. Compared to the redundancy of storing every non-diagonal element twice, this trick stores these elements into almost half of that memory. Cutting your memory use in half is nothing to sneeze at even if the program ran to completion even under a more wasteful memory allocation scheme, since it allows you to solve instances twice the size, and that way have hopefully at least 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 eight cells in its 3-by-3 Moore neighbourhood (fewer at 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 its 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 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 be redundant to require 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 JUnit tests 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 again try something new in this lab, instead of animating the evolution of the automaton states using some form of Java concurrency, this 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 forward or backwards to any particular moment of time to see the board displayed in the `paintComponent` method in a manner that resembles the screenshot below.

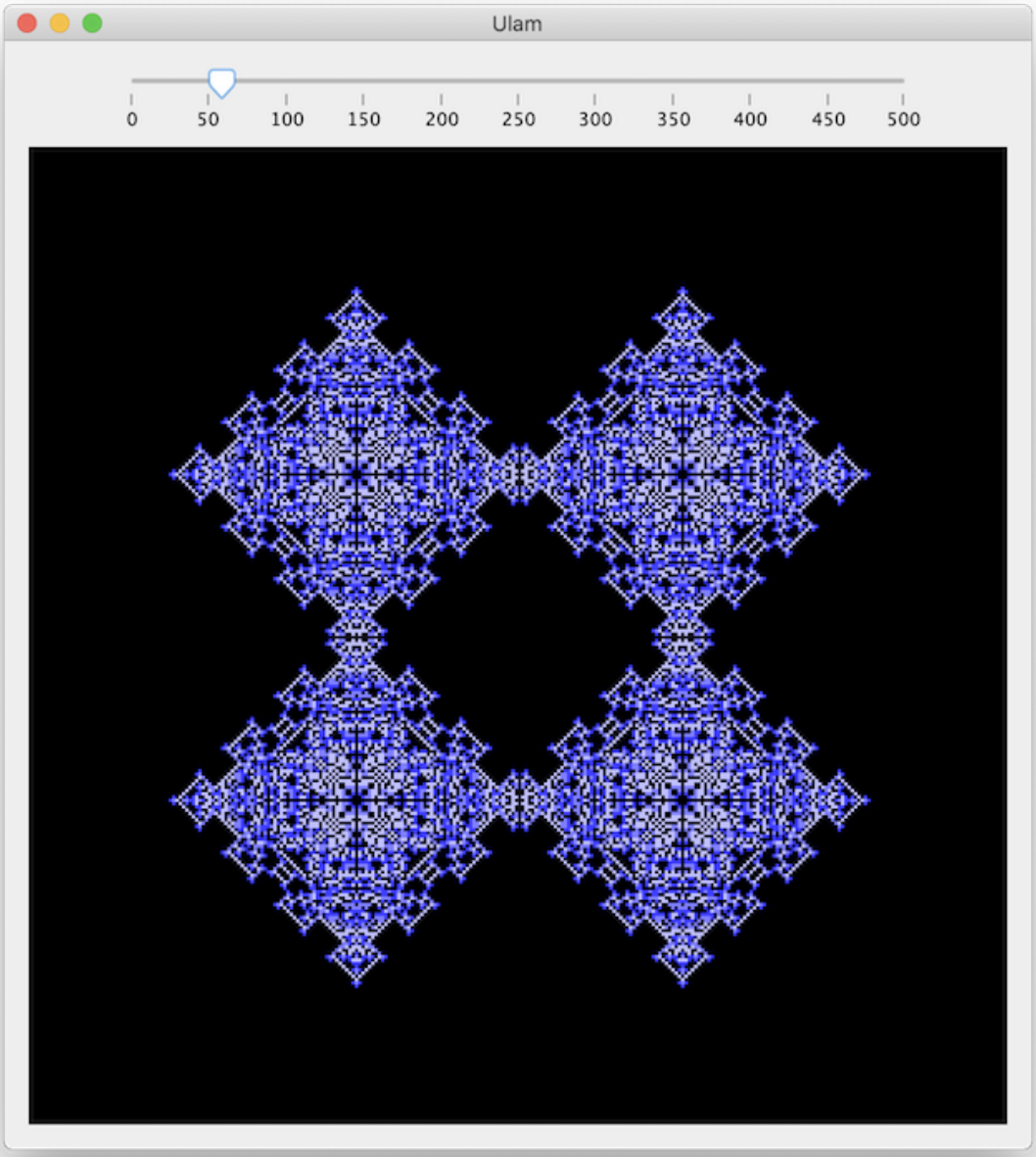
This display of a 300-by-300 board renders each cell as a 2-by-2 pixel box to make the result easier to see, but you can adjust this if you want to explore the evolution of the crystalline structure in larger game boards. You can also try seeding the automaton in a non-symmetric fashion to break the symmetry in the evolution of the automaton. The colour scheme used in the screenshot uses the colours listed in the following to render a cell based on its state at times *t* and *t* - 1.

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 surely 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 into. 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 the cipher-text 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**. So apologies for the inconsistent terminology used here. The first version of this problem was talking about "coins" until the author realized that in the literature of this field, the systems described below are known as chip firing systems. As long as the names we utter from our virtual mouths are sufficiently memorable and unique to keep their referents apart, it's all to-*may*-to, to-*mah*-to anyway.

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. For each time step *t*, the following operation is performed for every position of the array, producing the chip distribution for the next time step *t* + 1. The logic of your method must be designed so that its result is equivalent to performing this operation in logical unison for every position in an instantaneous **atomic** step so that all positions fire their chips in unison, as if they were saluting the most powerful *maharaja* that the brave forces of our beloved Queen ever had the honour to encounter!

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 inside the system, it just moves them around the positions. The **states** of this chip-firing system therefore correspond to the possible ways this fixed chip stack can be distributed among the positions. The **transitions** lead from each state to the next state that results from the previous operation. To allow you to compute these transitions, create a new class `CoinMove`, and there 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 lists `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 buzzes in this 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 a classic in computer science that deserves way more fame than it currently enjoys.

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, shake hands and say their equivalent of “jolly good, old chap” to each other, knowing that the current state must lie inside the cycle. The hare can now stay in place and take a well-deserved rest, while the tortoise makes another trip around the same cycle, counting the steps needed to return to hare.

As revealed by the JUnit fuzz test, the period of this cycle greatly depends on the structure of obligations that the positions have for each other. For most configurations 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, even well over one million for some of the initial states and obligation lists randomly generated in the JUnit test. Same 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…

The famous space-time tradeoff of computer programming is usually performed to the other direction than in our case here. Normally we make the program run faster by granting it more memory to remember the relevant stuff that it has seen and done. The exact opposite technique used by the Tortoise and Hare is used in situations where remembering all those relevant things would require way more memory than we can possibly have available in any physically realizable computation device.

In addition to being used to model serious stuff during the billable hours while wearing a tie, chip firing systems can also be used to create computational art by having each pixel correspond to some position at the given time, and colour that pixel according to the number of chips present in there at that time. Anything that we can do to bring art and science closer together is by definition progress. Chip-firing systems are closely related to the Abelian sandpile model, presented in an entertaining fashion in the Numberphile video "Sandpiles".

Lab 25: Manhattan Skyline

JUnit: ManhattanTest.java

Manhattan skyline problem is a well-known exercise in **computational geometry**, with room for many educational and interesting variations. Its setup makes for a good illustration of the **sweep line algorithm** used to efficiently solve this problem. 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 located far enough 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 and e that always satisfy $s < e$, and the **height** of h. These projected silhouettes can overlap each other in part or whole even as the original three-dimensional buildings are physically disjoint. Create a new class `Manhattan` in your labs project, and the method

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

The three arrays `s`, `e` and `h` (guaranteed to all have the same length n and consist of positive integer values only) contain the start coordinate, the end coordinate and the height for each individual building numbered 0, ..., n-1, respectively. These 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 overlap at that x-coordinate.

Conceptually, a sweep line algorithm "sweeps" through the horizon from left to right, noting all the points along the way where some "interesting" event takes place, in the sense that event potentially affects the result computed by the algorithm. The algorithm should start by creating a list of these interesting events, and sort these events in order of their x-coordinates. In the Manhattan skyline problem, the interesting events are "building b enters the view" and "building b exits the view" as your virtual "eye" sweeps the horizon from 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`. (We need to do this encoding of events into integers in this tricky way so that each even is encoded to something nonzero, because there is no way to tell apart zero and minus zero!) Add these 2n event encodings into `events`, and sort them with a custom `Comparator<Integer>` local strategy class whose `compareTo` method compares the events `a` and `b` by first decoding them into starting times (taken from array `s`) or ending times (taken from array `e`) that are then compared to render the final verdict.

The events sorted in this manner are ready to be processed in the order that they occur sweeping from left to right, 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. (Sometimes several events take place a the same *x*-coordinate, but since the distance between those events is zero, all such redundant terms from slivers of zero area between the literal and metaphorical walls that separate us from each other vanish from this classic summation, as if sniped to extinction by Tom Berenger.) 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, this method can return the accumulated total.

s	e	h	Expected
{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 examples discussed during the lectures featured decorators for `Animal` and `Comparator` class hierarchies. The decorator pattern works the same way for any other class hierarchy, except that the more methods the public interface of some class contains, the more annoying drudgery decorating that class hierarchy becomes. The `Writer` hierarchy to transmit text data composed of Unicode characters allows for a more convenient writing of decorators. Only three methods need 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, unless the template method is declared to abstract in the superclass.

In your labs project, create the class

```
public class FilterWriter extends Writer
```

with the 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. Characters rejected by this predicate are ignored in the spirit of the followers of a high school queen bee mean girl, all together pretending that the objects of the scorn of Her Majesty do not actually even exist.

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 a vowel, and every consonant that immediately follows a consonant, producing results like following paragraph:

```
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.
```

At least to this linguistic layman, the produced text seems to have a curious pidgin rhythm and flavour of its own. This leads us to further wonder whether simple variations of this operation, possibly aided with some vowel and consonant permutation substitutions, could be mechanistically applied to our natural languages to create a completely made-up and yet internally coherent **pig latin** for the exciting universe of your future series of young adult fantasy novels. The algorithm demonstrated in the `DissociatedPress` lecture example can also be used to achieve such pseudo-linguistic ends. The algorithm could even effortlessly mash together several languages to obfuscate the flavours 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`.

We are accustomed to adding up numbers, but we can also "add" images to each other with concatenation, similarly to the way how strings are "added" by concatenation. Since pixel raster images are two-dimensional, we can stack them up not just horizontally but also vertically, provided that the dimensions of these images are compatible in that dimension. In your labs project, create a new class `ImageAlgebra`, and in 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 accept any number of arguments of type `Image`, including zero. The **horizontal stacking** method `hstack` (the method names were chosen to be the same as they are 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. To implement this method the easiest, just draw the individual images one by one to an appropriate position of the resulting image.) The **vertical stacking** method `vstack` works exactly the same but with the roles of width and height interchanged.

We can immediately put both of these 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 internally represent colours as **RGB triples** of red, green and blue so that each component can get values ranging 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 presenting to the world, whereas turning all three knobs to maximum gives you its brightest possible white colour available on your display device.

Inside Java, objects of `java.awt.Color` represent individual RGB colours as objects. However, when dealing with images whose every pixel could potentially have a different colour, we would prefer not fill the heap memory with actual objects. It seems very wasteful to spend twelve bytes of heap management bookkeeping space for mere three bytes of actual payload per pixel! Instead of acting all purist and insisting that "everything must be an object", four bytes of colour information can be crammed into a primitive `int` value whose four bytes represent the ARGB components of that colour.

In addition to the three colour components, this fourth “alpha channel” tagging along for free inside a four byte `int` has no fixed interpretation on top of its cool secret agent name. In practice, all image processing operations of Java treat the alpha channel as the **opacity** value of that colour, with 255 for a completely opaque colour and 0 for a fully transparent colour. When a pixel is rendered with a colour that is partially opaque, the resulting colour of that pixel is the corresponding partial 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. Some might be a bit tricky; for example, how exactly would you represent the colour brown? (Think about it for a while.) 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 while we ask what it can do for us today? 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**. When done in the HSB colour space, colour calculations and especially colour blending produce more aesthetic results.

To extract the HSB colour components of the given 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 utility method `HSBtoRGB` in the `Color` class. Of course the inverse method `RGBtoHSB` also already exists in the `Color` class, but that one expects to receive the red, green and blue values as three separate parameters, instead of being packed into a single `int` the way that the above method does the conversion.

To play around with all these pretty colours, create a class `Shades` in your 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 same components of the pixels (x, y) in the two source images. The weight used to mix these components is not constant, but determined for each pixel (x, y) separately 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 2.05 or thereabouts does the job pretty much everywhere where we would normally use an actual circle, and yet it is subconsciously perceived to be more organic and pleasant. The main problem of perfect circles is that they are literally too darn perfect to be perceived as natural and organic things. Few perfectly round things seem to exist in nature, at least in the resolution that our eyes can discern; immediate examples that spring to mind might be the Sun and the Moon in the heavens, and the iris and the pupil in the human eye. (It is probably not coincidental that humanity has historically ascribed preternatural properties to these objects.)

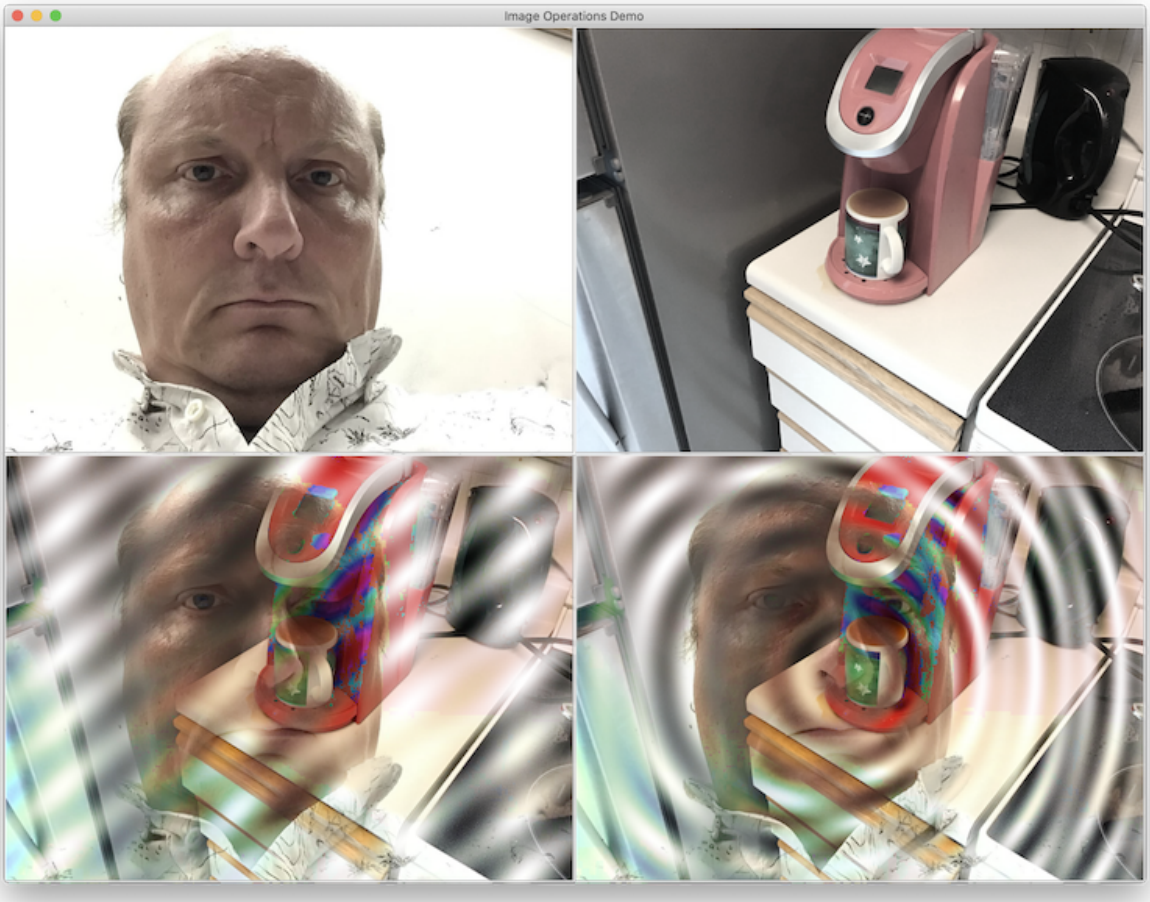


Image Operations Demo

Lab 29: In Timely Manner

JUnit: TimeProblemsTest.java

Computations on times and dates are tremendously 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 such happy-go-lucky ideas of how time works. Everyone who has come this far surely knows that during the year 2020 when this problem was originally 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 ensnare those patrolling this enormous jungle, nervously scanning for traps laid by Time Cong as millions of unblinking eyes stare at them from behind every leaf of grass?

To see the need for a 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 and corner cases to trip the unwary programmers who have to deal with these issues without any hand waving! Normies have the luxury of closing their eyes and hand-waving at such problems until they go away the same way as all problems eventually will, but computer programmers do not have such luxury.

Before Java 8, the standard library classes `Date` and `Calendar` encapsulated such computations. However, these classes were defective in too many ways to count. They would be more appropriate to be fed on punch cards into the maws of the sixties vault-sized Bat-Computer to predict the next crime spree of Calendar Man, rather than dealing with the real world in this millennium. The class `Date` is now officially **deprecated**, and the utility class `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 actually is well worth studying for its clean and solid object oriented design standpoint from all perspectives, once you have learned how to use its classes and methods in general. The design of this package is so clean that doctors could use it as sanitizer.

The classes `LocalDate`, `LocalTime` and `LocalDateTime` represent dates and times without any time zone information. (Pythonistas may note how 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** to bestow upon us a multitude of blessings 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, sort of, be a "mutator" for an immutable type) is always named `with`.

The first observation that jumps out from 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 originally written.

Create a new class `TimeProblems` into your 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 translate your ideas from there to actual code, with each individual step 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 not quite as cognizant of the Pigeonhole Principle compared to us moderns, and thus easier to dazzle with numerological coincidences. Such beliefs seem to be slowly fading into distant memories in our culture, although in relatively recent times, the notion of some numerical dates being inherently unlucky spawned an entire series of popular docudramas to warn the movie-going youth of the twin moral dangers of premarital sex and getting intoxicated.

The next problem was inspired by the "Gigasecond challenge" problem at CodeWars, another good online programming problem whetstone for journeymen to sharpen their coding and algorithmic thinking skills against. (While others were partying, you studied the code.) 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. (The class `Period` is similar in both spirit and usage, but with a less precise granularity of days instead of nanoseconds.)

To ensure proper understanding of the difference between the concepts represented by `Duration` and `LocalDateTime`, do you see how adding or subtracting a `Duration` to a `LocalDateTime` or another `Duration` is a perfectly sensible thing to do, whereas trying to add up two `LocalDateTime` objects would be pure codswallop? Even more contrary to intuition, even though dates cannot be **added** to each other (well, at least not in any way that would have the result meaningfully relate to any aspect of reality that is relevant to our future decisions) the **difference** between two `LocalDateTime` instances is a legitimate thing of type `Duration`!

```
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, just like we all do in our highly local everyday lives. This 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 are made 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.)

This pale blue marble whose spinning surface we live on against the dark eternity of cosmos is divided into **time zones** so that the same instant of time corresponds to a different time and date depending on your current location on Earth. We ignore the theory of relativity and any philosophical issues about what it means for two instants of time to be the finger-quotes "same" time instant, with or without any the theory of relativity or any "quantum" mumbo-jumbo sprinkled in. Time zones aren't quite as straightforward as most people assume; for starters, there exist 38 time zones, not just the main 24, and their clocks are not always some integer multiple of hours apart from each other. (Time zones also aren't easy to get rid of, nor made continuous.)

Instances of `ZoneId` represent the recognized time zones, and are identified by standardized strings such as `"America/Toronto"` or `"Pacific/Marquesas"`. However, before you optimistically extrapolate the general form of these strings from a sample of mere *n*=2, you should note that `"Eire"`, `"Navajo"`, `"US/Hawaii"` and `"America/North_Dakota/New_Salem"` are equally 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 that consist of multiple separate words, so that you don't get confused when there is no time zone of `"America/Los Angeles"`. Fortunately, all this preparatory work has already been done in the `ZoneId` and other classes in the package so that you don't need 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 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 the use of `withZoneSameInstant` method as the "mutator" for immutable objects to access the same instant 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 with 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 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 pure black or pure white) can be compactly stored as a two-dimensional array of `boolean` truth values. This representation allows reading and mutating 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 general bookkeeping of the two-dimensional array structure), the available memory sets a hard 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 “Not enough short bit patterns exist to uniquely encode not only themselves but also a whole lotta more of longer bit patterns”, establishes that no general method can possibly compress data made up of arbitrary bits in a way that would be guaranteed to 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 the individual bit values. Compressing that data will then erase such dependencies, as otherwise the compressed data could be run through another compression algorithm to squeeze it even tighter. Lather, rinse and repeat, until the data has been compressed to next to nothing. 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 black and white 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 whose side is some power of two. If this square is all white or all black, the object is a **leaf node** of this quadtree structure. For squares that contain both black and white pixels, that square is subdivided into four smaller squares that are recursively converted into region quadtree objects. These subtrees become children of the quadtree object that represents the original square. The recursive subdivision will necessarily end at a single pixel, but hopefully much sooner in areas of uniform colour inside the image. (At this point, readers may also note how "The Most Intolerant Wins", as observed by Nassim Nicholas Taleb.)

To do this lab, you should first copy the interface `QuadTree` and its two concrete implementations `WhiteQuad` and `BlackQuad` into your labs project folder. The interface `Quadtree` defines the functionality of a region quadtree as two public 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 $2^{\texttt{scale}}$. For example, the area of a quadtree that is all black in the scale of 3 would equal 8 * 8 = 64. For coding this method, note that Java does not have integer exponentiation operator baked into the language akin to `**` operator in Python. However, this particular case where the base of the exponentiation equals two can be easily computed with the **bit shift** operator `<<` so that the expression `1L<<scale` computes the quantity $2^{\texttt{scale}}$. Note that the 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 bits store the sign in the **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 the 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 consider singleton to be an object-oriented design **anti-pattern**. Vigorous debate continues.)

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

Instances of this class represent region quadtrees that are not of uniform but contain both white and black pixels, so they could not be represented as instances of either `WhiteQuad` or `BlackQuad`. In the spirit of this course, the class `QuadNode` is designed to be **immutable** so that you get to practice **caching** of results the first time that they are computed, so that any future calls of those methods can simply look up the cached result. This is yet another advantage that immutable types enjoy over mutable types. The class `QuadNode` should have 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 a 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 as in the earlier example implementations `WhiteQuad` and `BlackQuad`, the class `QuadTree` should also have a private constructor to disallow 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 stored 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 freely and safely shared between any number of users!

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

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 an instance of arbitrary subtype of `QuadTree` depending on the needs of the current situation. Since this constructor is intended to accept a small array arguments, we might as well make the corresponding parameter `children` to be a **vararg** so that the array elements can simply be given as separate arguments to this method, instead of forcing the caller to explicitly create a small array object first. This factory method should recognize special situations of these four children being either all uniform white or all uniform black, and return the correct singleton `WhiteQuad.get()` or `BlackQuad.get()` that is appropriate for the current situation. Otherwise, this method creates and returns a new `QuadNode` object with the given `children`.

Knowing that the child nodes cannot all be the same uniform colour, the method `isOneColour` becomes trivial to implement in the `QuadNode` class, to always return `false`:

```
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 for 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, using a `scale` one less than the `scale` in the current level of recursion for each of these recursive calls. 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? Assuming that `lastScale<scale`, you can simply adjust the previously computed `area` for the needs of the new `scale`. (Even in the opposite case where `scale<lastScale`, there are some moves you might be able to pull off before accepting the harsh reality of recomputing the area from scratch.)

Lab 32: Triplefree Sequences

JUnit: TripleFreeTest.java

A sorted list of positive integers is **triplefree** if it does not contain an **arithmetic progression** of length three or longer, that is, does not contain three elements `a<b<c` so that the differences `c-b` and `b-a` are equal. One more way to express this is to require that for any two elements `a<b` in the list, the quantity `a+2*(b-a)` is also not a member of that same list. For example, the three lists `[2, 3, 5, 6]`, `[1, 2, 7, 8, 10, 11]` and `[1, 2, 6, 9, 13, 18, 19, 21, 22]` is triplefree. Try as you may, you won't find three elements that together form a three-step arithmetic progression inside any of them! For example, since the last list contains both elements 18 and 21, it cannot contain either 15 or 24 that would complete such a triple.

Create a class `TripleFree` in your 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 from highest down 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 still return the second list for `n=6`.

`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. 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* want 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, more famous combinatorial problem of constructing the shortest possible Golomb ruler hardens these constraints so that each possible difference between two element values must occur exactly once in the entire array. Constructing such rulers for large *n* is not an entirely trivial task.

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. When the element type of an array object is a primitive, these elements are stored in consecutive bytes with no additional bookkeeping.

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, leaving less wasted space between them than inside a standard shipment of pre-assembled Ikea furniture.

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 available for each program was measured in kilobytes. Using these bits more productively lets our programs to solve bigger instances of the same problem within the same confines of heap memory.

In this lab, we shall design and implement a general purpose class whose instances behave as simulated `<DrEvil>`"arrays"`</DrEvil>` 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 between those bits of information than inside 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! In addition to the **bitwise arithmetic** operators to encode and decode the given integer into individual bits, the other new operation that you need in this lab is the **bitwise left shift** operator `1<<k` to quickly compute 2^k with only integers without any 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 greater or equal to `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.

You need to be careful with how you actually check for the condition (b) in your constructor. The seemingly obvious condition `n*k>Integer.MAX_VALUE` is always `false`, regardless of the `int` variables `n` and `k`. No result of integer arithmetic that has been truncated to fit inside an `int` can possibly be larger than `Integer.MAX_VALUE`!

To read and write the value of an element in the simulated sardine array, this class should provide the corresponding 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. These methods must also throw an `ArrayIndexOutOfBounds` exception if `idx` is outside the range of legal indices ranging from zero to `n-1`, and throw an `IllegalArgumentException` if the element value `v` is not representable in *k* bits. Again, the test method `testExceptions` in the JUnit test class `SardinesTest` expects you to throw these exceptions correctly to pass that test.

One last pitfall in the implementation of this class hides inside the corner case `k==31` for this problem. Using that value for `k`, the expression `1<<k` will fill in the **sign bit** in the highest position of that number, which makes that number equal `Integer.MIN_VALUE`, the smallest negative integer value that fits inside an `int`. You therefore need to be a "bit" more careful when checking whether the argument `v` of the `set` method fits in or whether you should throw an exception. Otherwise the JUnit test method `massTestMillionAndThirtyOne` will fail and reveal the final mishandling of this particular situation.

Lab 34: Preferential Voting

JUnit: RunoffVotingTest.java

For elections to determine exactly one winner from a group of C competing candidates vying for the job, Anglosphere nations tend to use the "first-past-the-post" system where each voter casts a ballot for exactly one candidate. The candidate with the **plurality** of votes wins the seat, 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. For example, the possibility of "spoiler candidates" whose existence induces **tactical voting**; some voters who know their favourite candidate is unpopular in general will cast their ballots against their true preference to help boost the lesser of remaining evils.

This lab examines two voting systems where each voter, instead of choosing just one candidate to vote for, ranks all C candidates in linear **preference ordering**. For the purposes of this problem, each `ballot` is a one-dimensional integer array that contains each of the numbers 0, ..., C-1 exactly once. For example, assume C=4 with the candidates numbered 0, 1, 2 and 3. The ballot `{3,1,0,2}` would then indicate that voter wanting to see the candidate 3 win, but if that candidate cannot win, that voter would rather see the candidate 1 win, and so on.

Both methods of this lab receive the stack of `ballots` as a two-dimensional array of integers whose rows represent the individual ballots. Now that each ballot contains more information than merely identifying that voter's most favoured candidate, voting systems can be generalized to use all this information to determine the fair winner… with or without using the **Benford's law** to pry out any irregularities to shine the light to some sneaky bastard stuffing the ballot box in dead of night.

Create a class `RunoffVoting` in your 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. To guarantee the uniqueness of expected results for our JUnit fuzz tests, we replace the coin flips with the rule that **all ties that occur during the calculation must be resolved on spot in favour of the higher-numbered candidate**.

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

Resolve the winner of the election using the pairwise Condorcet method. In our variation to guarantee a winner within this method without resorting to additional tie breaker mechanisms, each candidate engages in a pairwise match against each other candidate. Each pairwise match between two candidates Winston Noble and Hubert Hoag tallies how many ballots rank Winston above Hubert, versus how many ballots rank Hubert above Winston. During the pairwise match between Winston and Hubert, all the other candidates and their relative rankings to Winston and Hubert are temporarily ignored. The winner of each pairwise match earns one "match point", and the candidate with most match points after all pairwise matches have been completed wins the entire election. Note that whether Winston expectedly beats Hubert in their pairwise match by one vote or by millions of votes makes no difference; Winston gets exactly one match point either way, no matter how comfortable his margin of victory.

An election with C candidates should therefore end up with exactly $C(C\text{-}1)/2$ match points, regardless of the number of ballots cast. You should debug your method by `assert`ing this fact after the fact. To pass the JUnit test in a reasonable time, make sure that you are not being a "Shlemiel" who combs through each ballot and each candidate more times than is actually needed.

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

Resolve the winner of the election with instant runoff, a more complex but also more fair and just manner of choosing the winner. Initially, each candidate is awarded all the ballots that placed him as the first choice. Then, as long as no candidate holds **more than half** of the ballots that would give that candidate an immediate win (even though his eyes can already see the promised land, having exactly half is not enough to start the ticker tape parade), the candidate who currently holds the fewest ballots is eliminated. Again, in case of two or more candidates having the same number of fewest ballots, be sure the eliminate the lowest-numbered such candidate. Ballots held by the eliminated candidate are distributed between the candidates who are still in the race, so that each ballot goes to the next candidate down the preference list stated in that ballot who has not yet been eliminated from the race.

Once some candidate has been eliminated, he is out for good, and no future redistribution of ballots can resurrect him back to this process. This spells doom for any "apple pie" candidate, where this term is not a reference to nostalgic Americana out of some Norman Rockwell painting, but for being everybody's second choice but nobody's favourite. In other words, a neutral candidate who nobody hates but nobody feels any jingoistic enthusiasm for either. That candidate will therefore be eliminated in the very first round, even though in the grand scheme of things, that candidate could have been the optimal global choice in the hypothetical situation of extremely fragmented partisan jingoism where every voter loves their own Dear Leader and fervently hates every other candidate who is not Mr. Apple Pie...

The critical step to the performance of this method is the redistribution of ballots from the candidate being eliminated in the current round. 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 along the same parts of the fence that he has already painted many times 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 purpose) so that you can quickly loop through the ballots of the candidate being eliminated without having to loop through all ballots of the entire election to find out which candidate each ballot currently belongs to. Note that each ballot can be identified as an integer that gives its row position in the `ballots` array.

Again, be careful to ensure that your method resolves all ties in favour of the higher-numbered candidate. Also, treat any candidate who is not ranked as the first choice in any of the ballots as having already been eliminated in this computation even before even the first elimination round, so that that candidate doesn't come alive in any later later elimination round.

Lab 35: Multiple Winner Election

JUnit: MultipleWinnerElectionTest.java

Anglosphere countries tend to use voting systems where each district elects exactly one representative, typically done with the blunt first-past-the-post method to determine the sole winner of each district. Some 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** for each party, as described below. You might therefore want to implement this entire voting algorithm as one `private` method that the following three `public` methods pass the buck to.

```
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` gives the number of votes received by each party. The `seats` are given out one at the time same as in the Huntington-Hill method back in the Lab 0(G) so that each seat goes to the party that currently has the highest priority. To make the results sufficiently unambiguous for the JUnit fuzz tests, all ties should be resolved in favour of the party whose number is higher, same as in the earlier "Preferential Voting" lab. The priority quantity of the party that received v votes and has so far been given s seats 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 again perform all these calculations with exact integer arithmetic using our `Fraction` data type, 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 how adjusting the divisor sequence in the priority quantity calculation can cause this process to favour either larger or smaller parties, the Webster method being most favourable and the Imperiali method the least favourable for small parties.

(As a final humorous aside on the general theme, what should we think of a method where the priority quantity is computed with the formula $v/2^{s+1}$ dealing the seats out logarithmically so that doubling your number of votes gives you one additional seat? Let those wonks at FiveThirtyEight wrap their pointy eggheads around that conundrum!)

Lab 36: Rational Roots

JUnit: RationalRootsTest.java

Everyone who has come this far surely heard of the quadratic formula back in high school. Since that already brings up nasty square roots with irrational and even complex results, we shall leave that problem for higher level languages such as *Mathematica* that can properly handle algebraic computations on symbolic expressions. Instead, this lab makes you implement a handy method based on the rational root theorem for polynomials that is honest-to-God guaranteed to find *all* rational roots of rational polynomials of *arbitrary* degrees, not merely for some quadratics. (Put that in your pipe and smoke it, Abel!) The method itself is clearly explained on the page "Finding zeroes of polynomials" of the course "Algebra" in "Paul's Online Notes", the most concise and still the best among the undergraduate calculus online courses that 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}` where the coefficients are read from the lowest order term to the highest. Terms that are missing from the polynomial (in this example, the exponents 1, 4 and 5) are not actually missing from the array, they just have a coefficient of zero.

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 gather to the result precisely those points that make the polynomial 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, so not much choice there) 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 to try out; two for c, eight for b, and for each such pair, try both possibilities b/c and $-b/c$ for the sign of that root.

The operation to evaluate the given polynomial at given rational point 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, again please don't be a Shlemiel and evaluate each term separately, but you should rather realize that you can get the next term x^{k+1} with just one more multiplication from the previous term x^k. (The best way to evaluate an arbitrary

polynomial at the particular point *x* is the Horner's rule, shaving down the action into one measly multiplication and one addition per term.)

```
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 even in the case of a double root. 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 a breeze 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 small screenshot below copied from the session used to create the automated test class shows the idea of building up symbolic polynomials that have the rational roots that we want them to have:

```
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. This author finds it strange that this wonderful system is not taught to everybody during the first week of the intro calculus class. Plus, even though even many active 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 mathematics, especially when integrated with the Wolfram Alpha knowledge base. As high-level languages go, the experience of 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 start with 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

When standing on any integer a, you can step into either $10a$ or $10a+4$; also whenever a is even, you can additionally 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 back when that kind of thing was still a thing that people occasionally did. As this instructor learned from "In Polya's Footsteps", collection of entertaining expositions on mathematical topics by the late great Ross Honsberger, a crafty young fella could get from $a = 4$ to any other positive integer of their choice by using only these three basic moves.

This lab examines a mathematically less sophisticated but still universally effective **systematic search** method of breadth-first search. This search algorithm is guaranteed to discover the shortest path of steps from the number 4 into any given n. Implementing this search algorithm for this problem teaches you the general idea of how to solve literally any problem of this nature with the systematic approach of generating all shortest paths to every number in such breadth-first manner, terminating as soon as once the shortest path to the goal has been discovered.

Create a class `TenFour` into your labs project, and in that class write 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 integer `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`. To implement 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. For a refreshing change, this `frontier` list should be an instance of our red-headed stepchild `LinkedList<Integer>` instead of the golden child and usual go-to guy of `ArrayList<Integer>`, because to implement a **first-in-first-out queue** this algorithm needs, we surely want to add elements at one end of the list, and remove them from the other end. Depending on which way you slice it to choose your poison, either the addition or the removal operation for `ArrayList<Integer>` will necessarily take a linear time to restore the element structure to be consecutive without gaps, whereas a `LinkedList<Integer>` guarantees both of these operations to consume only a 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(not just their lengths) are unique to 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`.

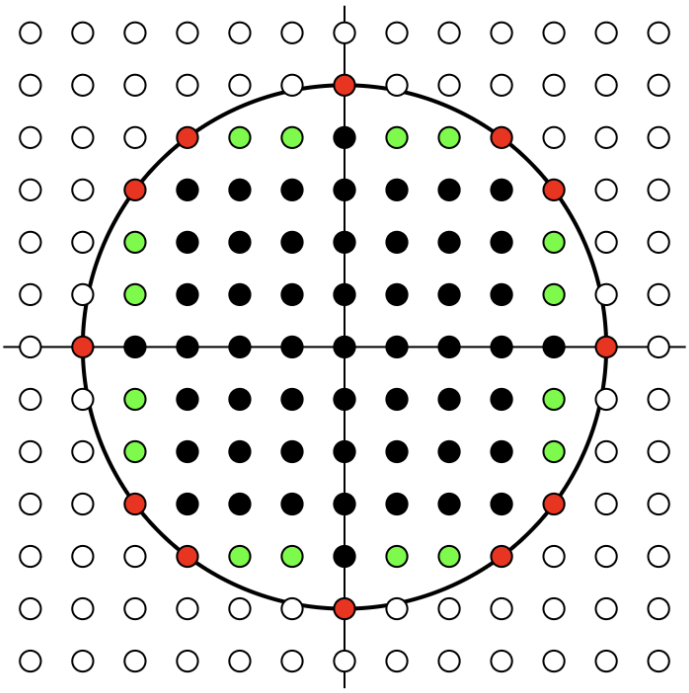
Keep going until you discover the goal number `n` during the expansion of some other number `v`. At that point you can construct the path from `n` back to 4 by following the `parent` information that you stored along the way, and then reverse that path from `n` to 4 to be returned as your final answer 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 that stayed within that limit) 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 $x^2 + y^2 \le r^2$. The points on the boundary **circle** make this an equality instead of inequality. As we were taught back in school, the area of this disk is given by the famous formula πr^2. Note also the correct terminology in this context in that a *circle* is the one-dimensional boundary curve around a two-dimensional *disk*. (Equivalent concepts in three dimensions are the boundary *sphere* around the solid three-dimensional *ball*.) So, next time some weisenheimer challenges you to recite the formula for the area of a circle from memory, you can remind him that being a one-dimensional curve, the area of the circle around the disk is zero...



The Gauss circle problem asks for the exact number of 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 the disk but neither on edge or border are **proper internal points**.

As can be tallied from the above image, the disk with $r = 5$ contains 12 edge points (red) that correspond exactly to Pythagorean triples of integers (x, y, r) whose hypotenuse equals r. That same disk also contains 16 border points (green) and 53 proper internal points (black).

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

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

Were this course given in Python, this function would surely return a three-tuple of integer values. Since Java does not have tuples in the language, this method does not return anything, but instead 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 when the value of `r` fits inside an `int`, its square `r*r` might not be quite as friendly.)

The following table lists a couple of values for *r* along with their expected numbers of internal, border and edge points. Due to symmetry, the edge point count will always be 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 along with *r*, but the exact 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 still 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. The relative area of these trimmings compared to the total area converges to zero as *r* increases.

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 iterate through all possible combinations of *x* and *y* both 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 even for pretty 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. Manipulating and evaluating symbolic formulas is always more efficient than grinding low level arithmetic like some machine.

Place your finger on the topmost edge point (0, 5) of the previous picture 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 you have no other choice. This technique is guaranteed to work for 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 $y = 0$ is its own symmetric row, unlike the other rows with $y > 0$ that are symmetric with the row $-y$ mirrored along the x-axis.

This problem would become even trickier if the coordinates of the center point (c_x, c_y) could be arbitrary real values, instead of being permanently nailed to the origin. An interesting theorem says that for any positive integer n, there must exist 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 to give us the answer on the spot without having to crawl through the rows of points one at the time. However, this problem still stubbornly remains unsolved, despite all the effort to come up with such formula! 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; that is, how many coordinate pairs (x, y) inside the disk are **relatively prime** to each other so that they satisfy the equation $gcd(x, y) = 1$.

Lab 39: Hot Potato

JUnit: HotPotatoTest.java

A group of *n* people numbered from zero to *n*-1 stand around a circle. Each person has memorized their personal "enemies list" of other people in the circle. However, this enemy relation is not symmetric so that Moe can be an enemy of Joe even when Joe is not an enemy of Moe. (You may not be interested in war, but war is still interested in you; the relationship between predator and prey is inherently asymmetric and does not go away by merely closing your eyes and wishing so.) For the purposes of this problem, everyone has at least one enemy inside the room and, unlike in the real world, nobody is messed up enough in the head to be their own enemy. During this process, no new enemies are created or friendships forged in the room.

We may also imagine that these events take place in some very official room with a high vaulted ceiling, and that these men are dressed in some sort of fancy robes or togas to befit their position. Either way, all these people are assumed to be some sort of perfect Machiavellian schemers who hatch all their plans in secret, but do not hold any grudges or desire for payback for any past acts of trespass against them and theirs, unless the revenge actually benefits them in the future. (There is no love to be found in this room, so preference between being hated rather than loved is not an issue.) They have properly internalized the ideal Markovian view that once the present state of the world is sufficiently known as to determine all our future decisions, the past does not exist in the relevant sense that it could afflict the outcomes of our actions. The only thing that matters at the present moment is to accept that present moment as it is, and choosing your actions at that eternal infinite now to maximize your future gains over all the future present moments.

Students who are interested in this general theme should also check out the popular writings of Nassim Nicholas Taleb in their spare time. To sum up its relevant ideas while standing on one foot; you should only care to listen to advice from those who have demonstrated that they understand the concept of ergodicity (if you have to ask what that means, you definitely don't understand it) and can explain its effect on their decision making, and that they have a "skin in the game"... and significantly more of risk of burning to crisp than merely getting to eat delicious potato skins!

At time $t = 0$, the person number 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 to catch. That person will then similarly 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 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 the 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 contains 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 without any rounding errors, which is why 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 the person 0 is initially holding the hot potato.

Probability $P(t+1, p)$ can be computed from previous probabilities at time t by first finding all people p' who consider the person p an enemy. (Since the enemy relationships never change during the execution of this method, it would be smart to precompute these at the start of this method, so that you can just look them up whenever you need them.) For each such person p', compute the quantity $P(t, p') / m$, 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 it to the person p." The desired $P(t+1, p)$ can then be computed by adding up all such terms $P(t, p') / m$, since those are the only possible ways for p to receive the potato at time $t+1$. Fortunately these are all mutually exclusive, so we can simply add up their individual probabilities to find the desired probability.

The following table 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 the previous distribution at the time $t = 2$. This second 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. (Long-term behaviour of any such Markov system can be solved symbolically with linear algebra arbitrarily far to the future.)

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

All magic tricks are trivial once you get to watch them from the back stage (even better, in the rehearsal room), as opposed to seeing them unprepared through the fourth wall in the actual audience. The seam carving algorithm for **context-aware image resizing** is deservedly famous for the almost magical outcome that it achieves relative to its own far smaller complexity. Modern image processing algorithms employ arduously trained **deep neural networks** that implicitly identify the **semantic content** of the image. The seam carving algorithm is significantly less lofty, as it employs only good old fashioned arithmetic and algorithmic tabulation of intermediate results to determine which pixels to eliminate. Seam carving performs its magic with simple local calculations with no higher semantic knowledge of the objects that appear in the image, nor any ordering of importance between them. Everything is done at individual pixels and their local gradients.

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. For simplicity, 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 whose width has decreased by one pixel. The resizing can go on arbitrarily deep by carving these individual slices of bologna out of the image one at the time.

The finer 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. Here in the Wossamotta U. whose faux marble columns tend to be covered with lichen, graffiti and vomit instead of ivy, our practical can-do attitude ought to propel us to do at least as well as those perfumed princes in their secret handshake clubs. Have all those classic college comedies really taught us nothing, my fellow band of lovable underdogs itching to take on "The Man"? Come on, the movie trailer practically writes itself! Create a class `SeamCarving`, and there the method

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

to create and return the seam carved version of the `image` horizontally scaled to the desired `width`. To allow you to concentrate on the actual seam carving algorithm instead of the gritty details of the bitwise arithmetic needed in 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" that pixel contains.

```
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 with repeated carvings can be seen in the screenshot below.

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

Lab 41: Geometry I: Segments

JUnit: CompGeomTestOne.java

"*Whoever says X, must also say Y.*" — V. I. Ilkkanov
"*Whoever says 'ex', must also say 'why?'* " — its modern relationship corollary, apparently

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 analyzed in symbolic forms within the homoiconic language itself. No, this term does not refer to Liza Minnelli in this context. "Homo" means "same", and "icon" means representation, referring to how the structures of the a homoiconic language itself are also data that can be processed in that same language.

Even worse, algebraic calculations of even seemingly simple geometric shapes tend to be chock full of square roots and other irrational quantities, especially once trigonometric formulas for arbitrary angles enter the fracas. Even the familiar junior high school mathematics notion of how a "line" is defined by its "slope" and "offset" goes what-sorcery-is-this *kaput* the moment it encounters the first vertical line segment, since computing the slope of that vertical line involves a division by zero. It is our duty as responsible educators to save our students from the claws of such *comprachicos.* In the next four labs, students are invited to enter Leopold Kronecker's paradise of well-behaved integers and everything good and plenty that springs forth from them, before that old wicked serpent of division sneaks its forked tongue inside this big happy tent.

Geometric computations cannot be avoided in some applications, most exciting of which are surely games played in a simulated microworld in some finite box inside the infinite 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 must be such lattice points. Since this lattice can be arbitrarily large, we can get as much precision as we want simply by scaling the world to be large enough to make every point inside it to be a lattice point, with no need for fractional parts within the pixel raster of the game display.

Instead of using 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 so wouldn't be able to explain standing on one foot exactly what makes these functions tick.

Create a class `CompGeom` in your labs project. You will be writing all methods from these computational geometry labs inside this same class. Each lab will have its own JUnit test class that depends only on the methods written up to that point. To begin, copy-paste the following implementations of the methods `cross` and `ccw` inside your class:

```
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 to keep the numbers small enough so that your code does not need to worry about the effects of integer overflows.) 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, nor care about the inaccuracies of floating point division.

Analogous to the bumper sticker version of Liskov Substitution Principle that your instructor keeps parroting every chance he gets during the lectures, you should also follow a similar bumper sticker maxim for computational geometry: **Should any one of the forbidden words "angle", "slope" or "root" come out in your description of how your method solves that computational geometry problem, 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). (Any point (x, y) on the two-dimensional plane can be thought of as a **vector** from the origin (0, 0) to that point (x, y).) The cross product of two vectors is actually a three-dimensional operation that produces a three-dimensional vector result, not merely a single scalar, but since any two-dimensional vector (x, y) can be considered a degenerate three-dimensional vector $(x, y, 0)$, multiplications by these zeroes cancel out most terms. Only the z-component of the result vector 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 cuts through seemingly complex geometry problems faster than a chainsaw 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 tend to have trouble dealing with, especially if implemented using 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 problems of points and line segments often begin with the order to “assume the general position" where no three points are collinear and no four on the same circle...

And that's it! That’s all she wrote and said! 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 turn defined by the three tactically chosen 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”, as the letter *s* denotes the 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 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 $O(n^3)$ from

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.

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 here, but the point (or more accurately in this case, the lack of `Point`) remains as valid.

Once again: no angles, no slopes, no trig, no nothing that could ever produce anything that is not an exact integer, since we can 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 applied to y-coordinates of these endpoints. In practice, most of the time we will receive the negative answer from quick rejection tests, since most pairs of line segments on the plane are mutually distant enough for this quick rejection test.

The method moves to the next stage 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. The same test must be repeated with the roles of two segments interchanged, since the segments must cross each other both ways to intersect.

This logic will take care of the intersection test between two arbitrary line segments, even in the degenerate case where one of these segments is a single point. To find all intersections within a set of n such line segments, of course you can loop through all $n(n - 1)/2$ pairs of segments, which would become a prohibitively inefficient "Shlemiel" algorithm as n increases. Assuming that the segments have been spread around reasonably evenly, a sweep line algorithm similar to the one that took apart the earlier Manhattan skyline problem will do this job in expected linear time. This algorithm would be slow only for pathological special cases such as these line segments being stacked vertically in a slab around the same x-coordinate.

Lab 42: Geometry II: Polygons

JUnit: CompGeomTestTwo.java

The previous lab of computational geometry introduced the `cross` product and `ccw` operations as the fundamental building blocks of our geometric algorithms. We now move on from sets of *n* points to ordered lists of *n* points that define an *n*-sided polygon whose edges are the line segments connecting the consecutive points in this list. The line segment from the last point in the list back to the first point closes the polygon. Depending on the needs of the current situation, we can think of the polygon being made up of "edge segments" or being made up of "corner points", but that is just a different "point" of view (surely it is not a coincidence that our everyday language is 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 the purposes of this lab, 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 no holes are left in the polygon boundary, but no edge segment intersections exist other than the *n* trivial 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 defined. 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 whether the collinearity avoidance rule is in effect.)

Furthermore, performing any **cyclic shift** to these corner points changes nothing, but the shifted points still represent the exact same simple polygon as a geometric object on the plane. The same would hold also for reversing the list of corner points, although 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 before, such as with `Fraction` class.)

Only the sign of the result of `ccw` has any effect on our decisions, and the magnitude of the result is irrelevant. Akin to how one-dimensional sorting algorithms only compare elements for order and never use arithmetic to find out the absolute difference between elements, **topological** algorithms whose results are unaffected by arbitrary scaling, shifting and rotating of these points, as a rule of thumb should never need the actual magnitude of the result for anything. However, this magnitude does still have a meaning that can be put to good use in another important problem, namely 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 whose corners lie at integer lattice points can only be some integer or some integer plus one half**, regardless of how these three points lie on the plane. The result is always this neat regardless of whatever square roots their side lengths, and whatever irrational numbers their angles happen to be.

Since any simple polygon can be broken down into disjoint triangles sharing edges and corner points, the same immediately follows for areas of arbitrary simple polygons. (As with many other theorems of computational geometry that feel to our intuition as if they were "🎶 so very clear and rational to anyone who does not belong to the oppressor class 🎶", this result is a bit more, ahem, *complex* than that.) **Triangulation** would then immediately provide us a way to compute the area of any simple polygon. Efficient algorithms have been developed to rapidly triangulate any polygon into a partition of disjoint triangles. (These algorithms have to tread carefully if the polygon to be triangulated is not convex; the greedy approach of just taking three consecutive corner points and clipping that triangle on its own, then recursively triangulating the rest will fail for such polygons.) However, we don't actually need to perform any triangulation to merely compute the area of a simple polygon, since we can compute this area in linear time just from the list of edges using the shoelace formula, courtesy of the awesome mind of Carl Friedrich Gauss himself!

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

Since the area of a lattice polygon can only ever be an integer or an integer plus one half, this method should return the area of that polygon multiplied by two, so that the result will always be an integer. The caller can always divide this result by two for the real area, but this way we can return 30 when the actual area is 15, and return 31 when the actual area is 15 1/2, so that the caller will be able to tell those two 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 typically don't have an exact representation in floating point anyway. You should still always be careful with potential integer overflows.)

Choose any reference point from the plane. It doesn't even matter which point you choose, or whether that point lies inside or outside the polygon. Given such freedom, we might as well choose the origin (0, 0) as this reference point to simplify the calculations, even though any other point

would have worked just as well. Initialize the integer `area` variable to zero. Loop through the consecutive edge segment 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, that's all the fat lady sang.

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 fractal 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. Only the area inside the polygon will not get cancelled out that way, so the final result will be exactly equal to the area of that polygon.

Lab 43: Geometry III: Points In Polygons

JUnit: CompGeomTestThree.java

Intersection and **collision** tests (the former for **static** worlds, the latter for **dynamic** worlds) are important in many games that take place on a two-dimensional plane. This lab gets the ball rolling with the simplest possible such intersection test to determine whether the given point lies inside a particular polygon. More advanced algorithms exist to perform efficient intersections and collision tests between arbitrary polygons, disks and other complex geometric objects. To speed up the intersection test between two complex objects, a **quick rejection** test that first checks whether some simple **bounding boxes** around those two objects intersect will provide the negative answer in the vast majority cases, without the need to actually execute the expensive intersection test between the actual objects themselves.

A simple polygon is **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 when the polygons involved are known to be convex. Every **triangle** is automatically convex as it came into this world (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 in computer graphics.

During the traversal of the boundary of a convex polygon, every turn must necessarily have the same handedness. 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 our 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 very 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 further guaranteed to be listed in counterclockwise direction. Instead of returning a mere yes/no answer as a `boolean`, it is 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 ranging from zero to three** so that the result zero means that the point (x, y) lies outside the polygon. Result one means that the point (x, y) is one of the corner points. Result two means that the point (x, y) lies on some edge segment but is not a corner. Finally, the result three means that (x, y) lies properly inside the polygon.

The caverns and tendrils of an arbitrary polygon can sprout into fractal-like complexities, so the question of point containment in a non-convex polygon is not quite as trivial as it is for their convex brothers. The first idea, same way as in the earlier problem of computing the area of a polygon, might be to subdivide the polygon into disjoint triangles, and then test for point containment within these triangles with the previous method. If some triangulation of that polygon is already available,

there certainly is no harm in using it, especially if those triangles have been arranged and sorted into horizontal **slabs** so that we can first use **binary search** to determine the slab where the point lies according to its *y*-coordinate, and then another binary search to determine the triangle that the point lies according to its *x*-coordinate. Otherwise solving one instance of this problem via triangulation would be the textbook definition of overpaying for overkill, since a far simpler algorithm already solves this problem for arbitrary polygons in linear time!

```
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 only one, and a line has none but is infinite to both directions. Most people erroneously say "line" when they `<fedora>`*ackshually*`</fedora>` mean to say "line segment", but this is a long-lost linguistic hill for anyone other than the most valiant and euphoric gentlesir to bravely die on.) Since we get to choose this direction freely, we might as well choose this direction to be one of the main axis directions with one coordinate being zero, to simplify the formulas involved. Any one of the four compass directions would result in a symmetric method, but since we have to pick one of these four, 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 as a trivial special case of parameterized curves, but even the simplest of those tend to soon enough end up requiring non-integer arithmetic), we realize that it is enough to look at the line segment from point (x, y) to the right just far enough so that its endpoint is guaranteed to lie outside the polygon. Any *x*-coordinate larger than the maximum *x*-coordinate among the corner points of the polygon will do just fine. Since the end position $(\max x_i + 1, y)$ of our rightward ray (there is another good name for a pompadour dad rock band, for any aging hipsters with missed musical aspirations 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 proves 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 frustrating as trying to get to the outside of a Klein bottle, but the nonzero winding rule handles such situations without confusion.

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. To pass the JUnit test, **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 rightward 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 corner (heh) case lurking in the grass waiting 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 determine, and let the polygon have two edge segments that connect the three consecutive corner points (5, 1), (7, 3) and (7, 5). The rightward ray emanating 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 at 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 often good to step back for a moment for a more calm and reasoned view. Cowboy coders (not a compliment) would add tests for failing test 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 happens to solve 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 needs to be more general so that it works under arbitrarily mirroring, rotation, translation or any other transformation of original points, as long as that transformation does not affect the topological properties of the original set of points. We want to be correct in general, not just *ad hoc*.

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, **treat the y-coordinate of the segment endpoint that is equal to y as if it were an infinitesimal distance lower than y for the purposes of this quick rejection test.** You

don’t actually need infinitesimals to exist in your number system, you just need to behave as if it did. (This idea also generalizes for many other problematic situations in programming.)

Using the previous numbers, the edge segment from (5, 1) to (7, 3) will now be 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 did intersect the rightward ray at the corner point (7, 3). However, the next edge segment from (7, 3) to (7, 5) is not quick-rejected, as the point (7, 3) is considered to lie below the ray whereas the other endpoint (7, 5) clearly 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 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 between those points. To conclude our discussion of basic computational geometry, we will tackle the problem of building a guaranteed simple polygon out of the list of points given in arbitrary order. In general, exponentially many different simple polygons could be constructed from the given set of distinct points. However, as always when we are granted the luxury of such choice, we might as well try to build a polygon that has some kind of useful properties for the rest of our program to enjoy!

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 nothing about the existence of other triangles that share common corner points, 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 particular triangle. Another triangle encoded as the array `{42,43,77}` shares the corner point with index 42 with the previous triangle. Adjusting the coordinates of the point number 42 will then automatically adjust every triangle that shares the same corner point, without us needing to explicitly search for all such triangles and painstakingly update them all! (In some problems, additional computation may be required to maintain additional constraints that prevent the mesh from becoming degenerate in some form.)

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 integers 0, ..., *n* - 1. Next, define a local class for a `Comparator<Integer>` strategy objects to perform the comparisons needed in sorting the array of indices.

```
class CCWComp 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 CCWComp());
```

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... nope, we were never supposed to use that bad word that Old Nick himself conjured up to lead us astray from our

path of righteousness! Try again. Points end up sorted in counterclockwise order relative to the reference point in the lower left, explaining the name `sortCCW` given to 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 there exists at least one point inside it so that standing on that point, you could "see" every other point inside that room by turning around in place. The so-called "star" shape of such a polygon might not be something we consider to be even a cartoon blammo star. All these names are made up anyway to bolster our intuitive feel 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 a further special case star-shaped polygons whose every point would work 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 in literature as Graham scan. 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 straightforward two-dimensional generalizations of the classic quicksort and mergesort algorithms originally designed for efficient one-dimensional comparison sorting!

(For a reduction to the other direction, 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. The existence of this reduction, even if never actually used in practice, establishes that the convex hull cannot be constructed essentially faster within the same model of computation than the time required to sort an array of one-dimensional such numbers.)

We implement specifically the Graham scan algorithm in this lab basically because the previous `sortCCW` method already did half of its work. Besides, this lets us finally use a **last-in-first-out stack** in the implementation of this algorithm, which turns this lab into an even more useful exercise in programming in general. 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 encountered during that period.

```
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`, whose value equals 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.

The second rule must be implemented as 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 is guaranteed to work in linear time despite the fact that it consists of two levels of nested loops, since the answer to this question generalizes to the study of such algorithms in general.) Once the outer for-loop has finished processing all corner points of the fan polygon, the stack contains precisely the points that lie on the convex hull. DUCY?

Each of the four labs of computational geometry has a separate JUnit test class to test only the methods of that particular lab. However, the `verifyPicksTheorem` method will act as the Final Boss to challenge the students who complete all four geometry labs in this fourth JUnit test class, these labs uniting to battle the Final Boss like those big robots in that big space robot show. This pseudorandom fuzz test repeatedly creates a set of random points on the plane, and uses Graham scan to compute their convex hull polygon. The area of this polygon is then computed with both the shoelace method and a brute-force 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. As we observe both of these methods for area calculation return the same area for a large number of such randomly generated convex hull polygons, our faith on the general 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 can put their finger on every item, and ask for each one why that be so.

Lab 45: Permutations I: Algebraic Operations

JUnit: PermutationsTestOne.java

The next troika of problem specifications covers the computational operations of permutations, integer arrays of *n* elements that contain each natural number from 0 to *n*-1 exactly once. The required methods in these three labs may assume that the arguments given to them by the JUnit fuzz tests satisfy this definition. Since we are computer programmers, we shall once again follow the secret pact of our ancient brotherhood of extreme finitists and start counting the elements and positions from zero, whereas real mathematicians prefer to define permutations using numbers 1, ..., *n*, which you 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 `<DrEvil>`"function"`</DrEvil>` that can be "applied" to any array of *n* elements. Since permutations themselves are also such arrays of *n* elements, permutations can be applied to existing permutations to create new permutations, which should make Keanu say "whoa" once again. 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 for which `perm[4]=9`, applying it to an array `arr` where `arr[4]=="Bob"` will see `"Bob"` in the position 9 in the result array.

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. 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 directly applied to its special case of permutations.

Enough theory, let's see some action. Create a class `Permutations` in your 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 visiting the intermediate position 4 in between.

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

Given the permutation `perm`, construct its **inverse permutation** that, if chained with `perm`, would produce 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 chaining 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 (since the element values never change, permutations play nicer than integer matrices so there can be no possibility of overflow), you should use the technique of exponentiation by squaring to compute the result much faster 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 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, especially a course on **abstract algebra**, can note how 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, the identity permutation acting as the unit element of these operations. Powerful algebraic truths and useful algorithms such as **exponentiation by squaring** spring forth from this realization. (Besides, permutation groups are themselves a big deal 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 implicitly contains all possible 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 its disjoint **cycles**. To get an idea of how these cycles work, you can first watch the YouTube video "Cycle Notation of Permutations" at Socratica.

To find the cycle of `perm` that the position `i` is part of, start a loop from position `j=i`. Keep moving from the current position `j` into the position `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 together as if piled on one of those tandem bikes that is so large that it automatically becomes comical. In the extreme case at the other end of this lattice poset, 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 the list of its cycles.

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

In fortunate situations like this where we have both a function and its inverse available as black boxes, an automated fuzz test is trivial to implement. Repeatedly create a permutation (either systematically or randomly), convert it to cycle form, translate that cycle form back to the explicit permutation that it represents, and verify that this permutation is equal to the original permutation. Such a fuzz test can be left running overnight to strengthen our belief in the correctness of the methods when we see no discrepancies with our morning coffee. Trying out a million random permutations at different scales should realistically at some point hit and reveal all possible edge cases and corner cases that this method could 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 its cycle form, since we could merely 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.

Parity of the permutation is another important algebraic property that can be easily computed from its cycle representation. This parity tells us something about how many **pairwise swaps** of two distinct elements are necessary to actually act out the permutation in practice. These swaps can then be organized in a host of different ways, but whether an odd or an even total number of swaps is needed to reach the goal is an inherent fixed quantity for that permutation. (Try it for yourself, if you don't believe this. You will get inevitably 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, you only have to 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.

The somewhat “odd" terminology where minus one stands for odd and plus one stands for even, despite the fact that both of these are odd numbers, comes from the 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, explaining the terminology of parity used here. In practice, this expression should always be evaluated with `n%2==0?+1:-1` in guaranteed constant time independent of the sign and the magnitude of n.

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

Since the canonical cycle notation is more readable than plain arrays, 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 stringify these permutations without any commas. Parentheses separate cycles from each other, even though these parentheses are actually redundant and could be removed to make this representation even more compact. The parentheses make the expression easier for humans to digest in a 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 with how *n* distinct elements can be arranged into exactly *n*! permutations, the product of all positive integers up to *n*. For example, 5! = 1 * 2 * 3 * 4 * 5 = 120. Well-known techniques exist to generate all these permutations one at the time in various orders. If you would first like to fool around with some classic techniques to systematically generate all permutations, check out the page "Generate Permutations" at Combinatorial Object Server.

In this lab, we look at an interesting one-to-one conversion between permutations to integers using Lehmer codes. To explain how Lehmer codes work, we first need to think a little bit about how integers are typically represented as sequences of digits. In everyday **positional number system of base ten**, 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 we don't say "ten hundred", since the latter carries to the next column to become "one thousand". (Yet in English, we still do say "eleven hundred"...) Humanity has historically agreed to use the base ten since our hands literally have ten digits (get it now?), so we tend to assume ten to be the only possible base by some law of Nature and Nature's God. But the same principle works just as well for any other base, most importantly for base two for **binary numbers**.

This base does not even need to stay the same for every position! **Mixed-radix representations** allow using different bases in different positions. We all do this routinely without even realizing it every time we talk about times and dates 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 must say "one hour and fifteen minutes" instead of "seventy-five minutes" so that values that grow too big for their britches get carried over to the next column.

Since exactly *n*! permutations can be constructed from *n* elements, the process of encoding each permutation into an integer should exploit this to the maximum effect. This can be achieved with the factorial number system whose positions denote the consecutive factorials 1, 2, 6, 24, 120, 720, ... (The linked Wikipedia page starts the factorials from zero, but this is redundant due to duplication of 0! = 1! = 1.) The largest digit allowed for the position that corresponds to *k*! is precisely *k*, since if you have the term *k*! repeated (*k* + 1) times in the sum, you actually already have the next higher factorial (*k* + 1)*k*! = (*k* + 1)! in your hands without realizing it. These factorial positions carry their spillage over to the next position, analogous to the positions in the ordinary base ten positional number system.

The following two methods implement the conversion between long integers and their factorial number system ("**factoradic**") representations. Since they are each other's inverses, the JUnit fuzz test repeatedly generates random `long` keys to verify 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 the position that corresponds to k! may be used at most k times. Note that the `toFactoradic` method does not return anything, but writes the answer into the array named `coeff` provided by the caller. (The caller is responsible for this array object having sufficient length to contain the answer.) This 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,`

To see how we can use this representation to encode permutations of n elements uniquely into integers from zero to n! - 1, we need the concept of an **inversion** inside a permutation. In your later courses on data structures and algorithms, this same concept will pop up again when analyzing the running time of the **insertion sort** algorithm. A **right inversion** inside the permutation `perm` consists of 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 n-element array in descending order has the largest possible number of $n(n-1)/2$ inversions, for the "n choose two" ways to choose any two positions among n positions without replacement when the order of the chosen elements doesn't make a difference.

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 later positions $j > i$ are smaller 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. 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 is always one element shorter than the original permutation that begat it. For example, the permutation `{5,6,2,3,1,0,4,7}` contains 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 somewhat disconcerting to realize that after almost three decades of existence, the `Arrays` utility class in the Java standard library still does not offer us a utility 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. Clever techniques iterate through *n*! permutations in lexicographic and other useful orders, spitting these permutations out one 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 recursive **backtracking** 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. (That one is another nifty little OOP trick that comes handy whenever some internal implementation details need to be exposed to the cold and inhospitable winds of the outside world for efficiency reasons.)

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 algorithm 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: The Curse Of The Clumpino

JUnit: ClumpsTest.java

All serious programming languages have the "A-list" data structures such as sets and lists lounging in the language itself or at the very least in its standard library, since these data structures are so immensely useful in solving problems universally through all possible problem domains. However, not every data structure ever invented passes the muster of practical applicability to earn its entry past the velvet rope into this exclusive club. In this lab, we will implement the disjoint set data structure, one of the lesser known "C-list" data structures who never got his big break to the major leagues and eventually drifted back to working at his old man's used car dealership in his Rockwellesque home town, occasionally crying over beers how he could have been a *contendah* when he was given the chance to speed up this more famous algorithm who *totally promised* to get him to the B-list as his plus one. The disjoint set data structure is worth appreciating on its own, even if we never went on to actually implement any of the classic algorithms that gain from the help of this plucky little hometown champ.

Create a new class `Clumps` into your 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 nothing but good old 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 produce new bigger clumps, as if they were made of clay. Before looking at the implementation details, here are the public methods that define what this data structure will do for its user code.

```
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 part of 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 two 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 count. (Code inside this method will not actually be counting anything, but will simply look up the value from the `size` table that the other methods continuously update as they mutate the data structure.)

These clumps can only ever grow larger when they `meld`, since no clump is ever broken apart into smaller clumps. Eventually all numbers belong to the same clump, after which point nothing can ever 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**, sort of primus inter pares that can be any one of the numbers currently in that clump, as long as one of the numbers shoulders the responsibility of acting as the representative. Which one of the numbers in that club assumes this role depends on the precise order of the `meld` operations that led the entire structure to its 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 integer `a` currently belongs to, continue looking for the representative in `parent[a]` the same way, until you arrive at the representative of the clump that is 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 follows the parents all the way up to find the representative of the clump that contains the integer `a`. After this representative has been found, this method should perform **path compression** as explained on the Wikipedia article about the disjoint set data structure, since this cleanup operation costs us essentially nothing while it will massively speed up future queries for numbers

whose parent chain ever leads to the representative through this same waypoint number. To perform this path compression, simply follow the parents from `a` to its representative `r` the exact same way as before, and reassign `parent[x]=r` for every integer `x` along this path.

The method `sameClump` should at this point be a one-liner that merely checks that its parameters `a` and `b` have the 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 `a` and `b` are already in the same clump, and return `false` to indicate having done nothing to nobody 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 seemingly “constant time” is not a constant for adult realsies, but a constant plus a certain weird little function of “inverse Ackermann" that is difficult to explain at this point of your computer science studies. This function depends on *n* and will grow without bound as *n* increases, but grows so excruciatingly slowly to make 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 an infinitesimally minuscule rounding error in the constant running time can never give you any hassle in any possible counterexample, not just in this real world of ours but in any imaginable one.

Lab 49: No Bits Lost

JUnit: BitwiseStuffTest.java

Dealing with bits and bytes directly inside Java is the closest that we ever get to the true essence of the underlying machine running our code. We prefer to think about execution and the general meaning of a program using higher level languages and abstractions, so this lab shall be considered both character and morale 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. (The same principle of learning something once and then getting to use it everywhere also applies to **regular expressions**, equally indispensable in their own problem domain of text pattern matching.)

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 from this entire problem collection, 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 where you study this, many computational problems have this kind of curious nature in that we cannot hurry up to get the result faster than helplessly waiting for the system eventually produce the result for us. Such problems are not limited to those that we capture inside our computing, looking in at them through the eyes standing outside as into the cages of some rare creature in a zoo.

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. However, this is lab is important not just for general knowledge of the reality underneath, but also for the many practical optimizations of time and space that this instrument allows in the hands of those who know how to insert it deep into the computer's memory. We should occasionally get our hands a little dirty in all walks of life, instead of always waiting for the tape to run by passing the buck to some public interface where some kind of magical elves do the work for us. Even if you got the spirit, don't lose the feeling.

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" even before humanity itself entered the cosmic game. The integer object that we normally write out in base ten as `42`, is still the exact same integer written out in binary as `0b00101010` to make it clear that 42 = 32 + 8 + 2, as you can see from how the expression `42==0b00101010` is `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 sort 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 are on to something with the base 12 of the **imperial system**, especially since the digits in that base 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` along with its more juvenile variations such as `0x0B00B1E5`. The prefix `0x` tells the compiler that the digits following it should be interpreted as hexadecimal even if they happen to all lie in the range `0–9`, so there can never be ambiguity of which representation of integers is used in the source code. The 64 bits inside each `long` value can similarly be encoded into sixteen snappy hexadecimal digits.

Again, all these integer literals could just as well have been written in base ten, since the integer itself is the same regardless of which one of its infinite names we happen to be summoning it this time, analogous to how "Clark Kent" and "Superman" are two separate names for the same actual object. Hexadecimal notation makes it clear to the reader that the integers are supposed to be semantically thought of as bit patterns with no direct interaction taking place between bits in different positions, instead of ordinary integers whose semantic purpose we normally associate with various counting and arithmetic operations.

Bitwise arithmetic operators `&`, `|` and `~` correspond to the logical operators `&&`, `||` and `!`, with the difference that the former operate on integers instead of `boolean` truth values. These operations perform the operations separately for each bit position, which allows to achieve some poor man's parallelism by computing the results of up to 32 logical operations in parallel this way, instead of having to process them one at the time with a for-loop.

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. 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 vanish. The **right shift** operator also is similarly not cyclic, but unlike the left shift, offers two separate versions >> and >>> with the difference that >> maintains the value of the highest order bit (since that important bit denotes the **sign** in **two's complement** signed integer representation) whereas the operator >>> 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 their aggregations do not come with any inherent semantics, since those same 32 bits inside a computer finger-quotes "mean" whatever you want them to mean. If you treat those bytes as an unsigned integer, then that is what they are, whereas if you treat those exact same bytes as a signed integer, that is what they are. By themselves, bits and bytes are nothing but zeros and ones without any inherent meaning. (Made of ephemeral electric currents, even "zero" and "one" are words made up to bolster our intuition, and could as well have been "true" and "false", "black" and "white", or "foo" and "bar".)

You can use the left shift as a shorthand for multiplication by some power of two 2^k, but the results can be a "bit" surprising if that highest sign bit of the result differs from the sign bit of the original number. This move allows us to create masks for a 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. (This analogy is apt in more levels than merely the surface one.)

Enough theory. Note that all the methods below have intentionally been specified to operate on 64-bit `long` parameters instead of 32-bit `int` values, since we now live in the science fiction age of 64 bits anyway, perhaps equipped with tail fins for additional "retro" appeal. (Four, sixteen windows, ten in a row. Some art major who has stumbled this far to this bug-infested dried-out forest when they originally just wanted to dip their toes in a computing course could perhaps turn this into one of those "Virgin vs. Chad" memes, setting that at the time of this writing those are arguably the 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 find useful writing the following methods in the class `BitwiseStuff` in your 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 in) 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 represents and describes 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. The four bits inside each nybble should remain as they were in the original. For example, when called with the `long` value whose hexadecimal representation is `0x63e821ceea5afcc6`, the result would equal `0x6ccfa5aeec128e36`, the original hexadecimal representation read backwards.

```
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 `long` value with hexadecimal representation `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 be handy in some combinatorial problems. These gritty operations are assigned to play in the alternative low-rent venues as they won’t get past the velvet rope of the “A-list lounge” of the language to hang out with superstars such as `+` and `*` so that, like Madonna or Cher in our world, they would be universally known for the character tokens that represent them. (Let alone being so important that, akin to The Artist Formerly Known as Prince, our rulers would add entirely new symbols to the Unicode character set specially for them!) Instead, they have been implemented as static methods `rotateLeft` and `rotateRight` in the utility 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 several values of `k` are equally good for this, return the smallest such `k`.

Applying the same ideas that were needed to implement the previous four operations, many other bitwise operations can also 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 at this low and dark realm of spirits. Students who are interested in these sorts 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 combinatorial and numerical techniques from the bit level up.

To bring this lab specification to its 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 the strict **left-to-right evaluation** and the resulting **short circuit** behaviour for the logical operators `&&` and `||`.

The evaluation order of Java subexpressions is otherwise generally left for the compiler to decide, and is not guaranteed to take place same as reading the source code from left to right. For example, the expression `rng.nextInt(42)*rng.nextInt(99)` is not guaranteed to produce the same result in every environment even under 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 abominations such as `a=++a+a---a+++--a;` whose runtime behaviour the language can only leave undefined, despite the fact that the expression on the right hand side has a unique parse tree to decipher its operations. To ensure your intended evaluation order, such expressions must be taken apart into multiple statements separated by semicolons that establish the strict sequential order in which these subexpressions are computed.

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 some evaluation order of subexpressions 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 the Current Year version of that compiler that will then emit some different evaluation order that *never* makes any difference to anything… at least until that initially adamant “never” once again turns out to be “hardly ever” in this creaky old Ship of Fools.

Lab 50: The Weight Of Power

JUnit: PowerIndexTest.java

Consider a weighted voting system such as a corporate shareholder meeting where participants wield 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 could also be something other than one half in special situations, such as the rules of some organization requiring at least two thirds of the votes for some particularly fundamental measure to pass. Sometimes even an unanimous decision is required such, as in some jury verdicts. (As many dramatic movies and special episodes of beloved television shows have taught us, being a one-man majority can be a nefarious position!)

In a weighted voting system, the actual power of each voter is not necessarily linearly proportional to the weight of that voter. For example, consider weights `[9,9,7,3,1,1]` with the quota of 16 votes required to pass the motion. There are really only two relevant situations to consider in this setup. Whenever the first two voters agree with each other, that already determines the final outcome regardless of the votes of the four other guys. Whenever these first two voters disagree with each other, 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 last three voters are **dummies** who might as well stay home, since the outcome tallied with their votes included can never differ from the outcome tallied without them. (The term "dummy" is not used here in the sense of being some sort of imbecile, but a mute and inanimate doll that cannot affect anything in the external reality with its choices and actions.)

To see even more clearly how the actual 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! Machiavellian operators in both fiction and real life throughout history have strived to find such 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 power index; all **subsets** for Banzhaf, and all **permutations** for Shapley-Shubik. Besides, the last names of these guys just sound so adorable, as if these gentlemen were rubbery characters in a *Mad Magazine* cartoon by Don Martin that depicts these algorithms as delightfully wacky machines with gears and whistles that sounds like "Fwababa dabada fwababa dabada" and "Shtoink!" as the ballots chug through.

Both algorithms operate with the concept of **coalition**, which here is a five-dollar word for what we normally would just 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 who 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 are both critical for this winning coalition, since removing either one of them makes the total fall below the quota. Voter number 5 with his single vote is not critical for the win, since the coalition `{0,2}` would still be winning even without his vote. That guy only gets the satisfaction of having rooted for the winning team this time… typically feeling all the more triumphant about this the less important his actual vote was for achieving this outcome, a tragicomical psychological phenomenon often seen in fans of movies, sports and politics. The human condition is as mysterious as it is tragic.

The Banzhaf power index for an individual voter is defined as **the number of winning coalitions in which that voter is critical**. These numbers are usually **normalized** by dividing them by the total number of winning coalitions so that they add up to one, but in this problem we will keep these counts as integers for simplicity. The simplest way to compute the Banzhaf power index is to recursively generate all possible subsets of voters. Whenever you find a winning coalition, increment the counter `out[m]` by one for its every critical member `m`. Recursively generate the coalitions that the current winning coalition is a subset of, since everyone will get points for all winning coalitions whose critical member they are, not just for the minimal winning coalitions. However, if none of the members of the winning coalition was critical for the win, you can save plenty of time by not recursing through the supersets of such redundant winning coalitions.

Create the class `PowerIndex` in your 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 best extracted into a 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`. (We could always compute this value by adding up the votes in the current `coalition`, but this way we have it available at each level in one constant time lookup, much better than looping through the array each time.) Parameter `n` gives the length of the prefix of `weight` array whose subsets this recursion should visit from the current level downwards.

The recursion to systematically iterate through all 2^n possible subsets of given set of n elements is essentially the same as the recursion seen in the **subset sum** problem. The base case of this recursion is when $n = 0$ and there is nothing to do. At each level in this recursion, if the current `coalition` is winning so that its votes together add up to at least to `quota`, every critical voter of the `coalition` without whom the quota would not have been reached by the rest of this coalition

gets its corresponding counter in the `out` array incremented by one. You need to check for criticality of each member of the winning coalition separately inside the loop where you hand out these points, since the winning coalition constructed so far is not necessarily minimal, but can theoretically contain any number of non-critical little guys.

In non-base cases with $n > 0$, regardless of whether the current `coalition` is winning, this recursion should branch in two different directions; one for taking the voter number `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, this method has nothing left to do but return to the previous level. Many legitimate recursions have this kind of nature in that the recursive method itself does not return anything, but some data structure shared between every level of recursion simultaneously gets incrementally updated during the systematic exploration of the branches of this implicit recursion tree. Once the recursion has systematically examined all possibilities, the final state of that shared data structure is our final answer given to Regis.

The Shapley-Shubik power index is computed by summing over the $n!$ possible **permutations** of these n voters, instead of summing over their 2^n possible **subsets**. 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 the `quota` is the **pivotal voter** of that permutation. The Shapley-Shubik power index of that voter is simply the number of times that voter is pivotal over all $n!$ 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.

```
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 the pivotal voters.

```
private static void shapleyShubik(int quota, int[] weight, int[] out,
int level, int last, boolean[] taken, int sum)
```

The parameter `level` keeps track of the current recursion level. It is initially zero in the top level call, and gets incremented in each recursive call. Knowledge of the current level of recursion tells us how far the prefix of the current permutation has been generated. Unlike 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` is greater than or equal to `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 incomplete but winning prefix into a complete permutation. This count is simply the factorial of how many unassigned voters still remain given by `weight.length-level`. You might want to **precompute** all factorials up to 17! that fit inside an `int` without overflowing into a `private static` array for quick lookup in 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 one half the sum of weights, **plus one**. The first row is the numerical example taken from the Wikipedia page for the Banzhaf method. The third row, constructed from the previous row simply by adding the big guy one more vote, represents a **dictatorship** where one of the voters outweighs everyone else together, and both metrics correctly assign all power to this dictator. The fourth row shows how equalizing the weights of every voter gives each voter the exact same voting power, as is to be expected when nothing external breaks the symmetry between these voters. Note how the power indices for *n* voters must always add up to *n*! in the Shapley-Shubik method, since every permutation of these *n* voters is examined.

`weight`	Banzhaf	Shapley-Shubik
`[4, 3, 2, 1]`	`[5, 3, 3, 1]`	`[10, 6, 6, 2]`
`[1, 2, 5, 8]`	`[1, 1, 1, 7]`	`[2, 2, 2, 18]`
`[1, 2, 5, 9]`	`[0, 0, 0, 8]`	`[0, 0, 0, 24]`
`[42, 42, 42, 42]`	`[3, 3, 3, 3]`	`[6, 6, 6, 6]`
`[2, 13, 17, 26, 30]`	`[2, 2, 6, 6, 10]`	`[8, 8, 28, 28, 48]`
`[10, 24, 24, 32, 40, 42]`	`[2, 10, 10, 10, 14, 14]`	`[24, 120, 120, 120, 168, 168]`
`[3, 14, 24, 26, 27, 64, 67, 70]`	`[2, 10, 14, 14, 14, 58, 58, 62]`	`[288, 1824, 2592, 2592, 2592, 9888, 9888, 10656]`

No matter how many orders of polynomials you manage to shave off from the exponential running time of your search as if carving ever thinner slices off an exponentially expanding turkey, you still always have an exponential search left in your hands. Assuming that you previously had enough patience to solve the problem for *n* voters before this optimization, you could probably solve that same problem for *n*+2 voters after that optimization. Meh. Inch-wormy increases rarely arouse any great enthusiasm for the future of the algorithm. Problems with a large *n* are best solved with randomized **Monte Carlo sampling** to evaluate each term only up to precision that is enough for

the purpose that we are grinding these power indices for. That's the cost of doing business in all problems that are inherently exponential. Which, to a first approximation, is basically all of them.

Exponential haystacks can be a bitch to find even one measly little needle from, let alone carefully combing through every possible straw of hay inside that stack like some modern day Rumpelstiltskin. 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 for this to pay off for the total running time. (Some examples on the web that perform these computations by hand 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 necessarily educational just for math and computer science, but for understanding all of life and social reality in general. For example, see the essay "The Inner Ring" by C. S. Lewis that uses no math, yet still makes important observations about this general topic. (Not everyone is doomed to talk nonsense even when they refuse to do arithmetic, be that in our real world or in Narnia.)

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 after we realized 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 for faster algorithms for primality testing by harnessing the predictably 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 even know yourself what you are going to be doing, no malicious **adversary** can exploit what you are doing by setting up the problem to seem as if your choices are reasonable at each turn, and yet in the aggregate, these choices lead you to the worst possible outcome!

In this lab, you will implement the Fermat primality test, mathematically surely the simplest one among 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 mod 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 will give very different results. For example, `-7%3` equals `-1`, whereas the quantity `-7 mod 3` would equal `2`, if the Java language actually had the `mod` operator.

The massive advantage of computing your all integer arithmetic modulo some tactically chosen m is that the result of every intermediate arithmetic operation will always be less than m. This allows humongous powers of integers to be computed without using 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 m, be it prime or composite. 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}. This multiplicative inverse satisfies the equation $(aa^{-1}) \bmod m = 1$ and therefore allows also division to be performed inside the field. This multiplicative inverse can be efficiently computed using the extended Euclidean algorithm without any divisions that tend to cause problems with integers.

To make this method ridiculously 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 permutation labs. When the exponent $b = 2k$ is even, use $a^{2k} = (a^k)^2$ to chop that exponent in half, and in the cases where the exponent $b = 2k+1$ is odd, use the subdivision $a^{2k+1} = a(a^k)^2$ to effect the same outcome. Depending on how your iteration is organized to run either as a while-loop or the equivalent **tail recursion**, equivalent variations of the subdivision formulas $a^{2k} = (a^2)^k$ and $a^{2k+1} = a(a^2)^k$ can be used.

Unlike Python with 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 anyway, the same 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 any 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 “papers check” inspired by this author watching one too many Big Herc and Wes Watson videos on YouTube discussing the hard reality of prison life, to help and motivate him to stand up and “do his own time” during this pandemic lockdown. Once the lockdown is over, it's time to get some coding time in!

Imagine the number *n* as a criminal who caught a case that he couldn't beat, so he has been sent up the river for a telephone number sentence. Having settled in, he cheerily greets his fellow crime lads, going on to proclaim to be in good standing in the cold and hard streets of the criminal underworld in that he has never ratted on his fellow criminals, nor has he ever committed certain types of crimes that the other men of conviction would not look too kindly upon. Any failure in this paper check when demanded (as the late great hardboiled crime author Andrew Vachss noted in one of his *Burke* novels, no matter how many years in 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 Punk City and never show his mug to the righteous cons 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 and ready to go at a moment's notice, knowing that his name has already

been carved deep into the cold hard streets that raised him. No matter which particular underling number *a* the shot caller of that particular modular gang will randomly send to do a heart and papers check, this convict keeps his head up high with certainty that the equation $a^{n-1} \bmod n = 1$ will stand up for him under any amount of pressure and interrogation under hot lights that the screws and 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 kind of bullshit charges they might fall behind.

On the other hand, any composite number is certain to eventually crack under sustained interrogation, and will sing like a little bird about his accomplices. It is definitely in the interest of the criminal fraternity to quickly reveal such snakes waiting in the grass. Those punks will soon realize that their bad life choices 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 such checks coming randomly at them from every direction with the knowledge that should they fail any of these tests at any time, every righteous 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. Your sentence is now fully served and you got your walking papers, so there is nothing else to do but to walk out of the prison of your own making. Keep reading on further only if you are actually interested in this sort of stuff.

Safely back in The World again, we can try a large number of randomly chosen integers *a* to challenge the primality of integer *n* for a **randomized algorithm** for primality testing. Fermat's criterion is merely the simplest of such properties of primes that have been discovered over time. Analogous to how barbed wire, guard towers and slightly cartoonish 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 is 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 to be accepted 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 the famous Cromwell's rule, we still ought to ascribe a small probability of erroneous bits occurring in the hardware executing the algorithm. (And what about the possibility of an error occurring in the underlying reality that is “executing” the physical computer 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 algorithm is in `BigInteger` as a `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 this method. (It is unclear to this author what `rnd` exactly stands for in this part of the Java standard library, since `rng` is more standard and clearly means "random number generator". Since the formal name of the parameter does not affect the caller anyway, this change would not break any existing Java code.)

This 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. This is trivial to do for the set of n-bit integers 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 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 an approach that resembles a headless chicken running around holding a scattergun 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 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 an ally in your corner, if you let it! Analogous to how we use fire properly down here in the physical world always as a servant but never a master, you should not let randomness wildly burn all over everything inside your program (or even worse, write your entire program randomly), but strictly limit its effects to rage inside some small, well insulated part of your program while keeping the rest of the program

predictably deterministic. This lets you have the best of the both worlds as the unpredictably raging fire of randomness takes apart the things that you need taken apart, no matter how tricky and convoluted your Adversary has tried to set up these things to hinder your progress.

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 for the gesture of such sorcery taking place, the sheer effectiveness of this random prime number generation algorithm might well be the closest thing to magic that we shall ever see in this course.

Lab 52: Linus Sequence

JUnit: LinusSequenceTest.java

This lab returns to the the theme of interesting integer sequences where iterating some simple rule makes the sequence bounce around like a headless chicken. The apparent incongruence between the algorithmic simplicity of the rule, versus the almost lifelike complexity of the outcome that emerges from the repeated application of that rule, is sometimes perceived as adorkably comical. It is as if "The Little Sequence That Could", delightfully portrayed in our imaginations by Zooey Deschanel or some other lovely *ingénue* of equal skill, aspired to become something higher than it has any right to be. Naturally we tend to root for such lovable underdogs.

In this spirit, the definition of the Linus sequence tries its best to be as unpredictable as possible, and yet this very attempt makes it easily predictable! This paradox combines elements of both the zen koan of the tip of the tail of the ox inevitably getting caught in the gate, and the sociological phenomenon of how every "hipster" (or other "rebel" whose cookie-cutter opinions would not raise an eyebrow in the HR department of any Fortune 500 company in the Current Year) trying hard to be all unique and mysterious will end up being a nonconformist the exact same way.

Before embarking on coding, we should allow the seemingly 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 towards wherever its by now well-worn tracks carry it. 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 any use of dice or other physical aids. Somebody whose "random" sequence starts with four equal elements is quite a unicorn among us Earthmen, even though there ought to exist nearly one billion people on Earth doing exactly that, if every human currently alive were somehow asked to mentally generate four random bits! Being in this sense very different from Vulcans, humans are in general notoriously bad at both emitting and detecting randomness. Fortunately, algorithmic techniques have been developed to combine and convert biased sources of random bits, either man or machine, into 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 (at least around your third year you should be taking a computer science course where this convention is still used, so that it is not "all Greek to you"), the **maximum repeated suffix** of the sequence s is its longest possible suffix β that allows the breakdown of sequence s into three component substrings $s = \alpha\beta\beta$ where α is the prefix of s before the duplicated suffix β. The maximum repeated suffix exists and is well-defined for any finite string s, since the empty string always works as the suffix β to force $s = \alpha$. For example, the maximum 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 labs project, and there the method

```
public static int maximalRepeatedSuffix(boolean[] bits, int n)
```

that computes and returns the length of the maximum repeated suffix of the sequence that consists of the first `n` elements of the boolean array `bits`. As 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 maximum repeated suffixes. We might as well use this 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 always called `1` and `2`.

`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 always either `1` and `2`, symbol that minimizes the length of the maximum repeated suffix of the corresponding extended sequence θ`1` and θ`2`. The maximum repeated suffixes of these extended extended will always have maximum repeated suffixes of 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 truth values so that `false` stands for `1`, and `true` stands for `2`. The first element of the sequence is in the position zero, set to `true`. (Absolute position indices do not matter in this problem anyway, as the rule talks about suffixes of the sequence generated so far and therefore takes the agnostic position in the schism between the zero- and one-based sequence indexing.)

The emergent behaviour of the Linus sequence is largely unknown, and chock full of mysteries far above this humble instructor's pay grade. Appropriately enough, your instructor first learned about this tricky sequence when browsing the Futility Closet, an excellent blog at the intersection of recreational mathematics and historical curiosities. The category "Science & Math" is probably the most relevant for the aims and the spirit of this course.

Lab 53: Tchoukaillon

JUnit: TchoukaillonTest.java

Variations of Mancala have been played around the world, but are relatively unknown in the West. The simple structure of such games that gives birth to a rich complexity of emergent situations, would make for good exercises on integer lists and arrays, so that the legal moves of the game are implemented as methods that operate on 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, then covers the **minimax algorithm** to find the best move that thwarts the opponent's potential responses.

Until we have that algorithmic machinery available, we have to content ourselves with Tchoukaillon (bless you!), one of the simplest **solitaire** puzzle versions of this game. Conveniently for our coding purposes, especially compared to the more convoluted boards used in many two-player variations of Mancala (even ignoring the symbolic decorations and other chrome), the Tchoukaillon board is a simple one-dimensional row whose pits 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 *k* that contains exactly *k* pebbles in it; for example, the position number five that currently contains five men, no more or less. If no positions that fulfill this requirement exist on the board, it's game over, man, game 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 now. 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` in your 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 shared in Java, to correspond to the new reality after this move. When called for `{3,1,2,0,4,5}` and `k=4`, the contents of that same list object 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. It is possible for the parameter `k` to lie 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 indicates whether the arguments `board` and `k` represent an undo of some legitimate move within the rules of Tchoukaillon. The move is legal when the position `k` on `board` contains zero seed, and every position lower than `k` contains at least one seed (including the position zero, that for the purposes of the JUnit test, must contain exactly 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 the position `{3,1,0,0,4}` would be a no-op, since the position 2 doesn't have a seed that it can give back for the undo.

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 that resides eternally in the Platonic paradise that is fundamentally out of reach of our mortal minds. However, that vile serpent Beelzebub will now slither in and legitimately point out the giant downside of immutable types; iterating through a large number of separate values using an immutable type necessitates the creation of a separate object for every such value. Furthermore, such **functional programming** approach 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 (in both senses of that word) sequence of values being iterated over.

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 for these 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 at least the best result possible, if not the hoped-for best possible result.

When it comes to the puzzle being fully winnable, Tchoukaillon is a rare bird among all such flightless waterfowl in that for every positive integer *n*, there exists **exactly one** possible way to place *n* men on the game board so that it is possible to bring all of them safely home using the 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 hopelessly stranded behind the enemy lines, no matter how you finagle the individual moves in the interim.

Since each move will always decrease the number of men outside safety by exactly one, and these men must be arranged in the game board 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, never by the **sunk reward** of men who already reached it… and besides, of what import are brief and nameless lives to Galactus anyway?

Lab 54: All Hail The Mighty Xor

JUnit: SigVectorTest.java

The **caret** operator `a^b` has nothing to do with exponentiation in either Python or Java, but serves as the **bitwise exclusive or** between two integers `a` and `b` when these integers are thought as **bit patterns** instead of ordinary integers that could be added, multiplied and order-compared. Unlike the familiar operators **and** and the **inclusive or** that tend to erase information as they go, this delightfully symmetric and most importantly **reversible** bitwise operator can be thought to take a bit pattern and a **stencil mask** that determines which bits are **flipped** to the opposite values. This intuition of stencils makes it easy to see why applying the same mask twice on the same bit pattern produces the original pattern. Generalizing this idea explains why the successive operations of the same mask cancel each other out even if other masks are used to flip the same bits in the interim. This seemingly magical property of exclusive or to perform tricks that apparently require memory but without actually needing any memory has several applications in computer science.

For example, it is occasionally convenient to compute and keep track of some sort of **checksum** of a large data structure or a data clump to speed up equality comparisons. If the checksums of two large objects are different, their payload data is also known to be somehow different without further ado. Especially when this data is **immutable**, the checksum can be cached alongside the actual data in a separate field. For example, the Java `String` class lazily computes the **hash code** of each object the first time that it is actually needed for something, and caches this code so that it can looked up in constant time in every future occasion. After paying the cost of each `String` object having another integer field, practically every future negative equality comparison will be correctly resolved by a single comparison of these cached hash codes.

For **mutable** data, updating the checksum on the fly at every mutation could be inconvenient. It seems especially wrongheaded to diligently perform some linear-time checksum recalculation after every little constant-time update! To avoid such silliness, this lab explains and implements Zobrist hashing that uses the exclusive or operator to a bunch of random integers to generate scrambly and chaotic checksums that can be updated with simple constant-time operations at every update of the data! This ingenious technique is usually presented on chess and other exciting game boards made of two-dimensional arrays. For simplicity, we shall restrain ourselves to compute these Zobrist checksums for one-dimensional **bit vectors**. Higher-dimensional data, or data of larger domains than mere binary, can (and *will*) always be encoded into bit vectors anyway.

To kill two birds with one stone, we shall also finally explore the Java language mechanism to allow **cloning** the instances of some class. In fact, the automated fuzz test will be cloning your objects as if were the mid-to-late nineties again with the cloning craze all over the media in both news and entertainment! Create a new class with the following signature and the three data fields

```
public class SigVector implements Cloneable {
    private boolean[] bits;
    private long[] keys;
    private long sig;
```

The objects of this class will serve as bit vectors that maintain a running checksum of type `long` to describe their contents. The actual `bits` are stored in the eponymous array field, and the field `sig` contains the current checksum. Your class should have the constructor

```
public SigVector(int n, long[] keys) {
    this.bits = new boolean[n];
    this.keys = keys;
}
```

to initialize these fields to create a bit vector of length `n` for which we maintain a checksum.

Zobrist hashing uses a table of random integer `keys`. Since these random `keys` are given to this class from the outside, and the correct operation of its methods is never supposed to mutate these keys, the same `keys` can be shared between all the instances of this class, instead of feeding the expensive habit to create a separate defensive copy for every individual instance jonesing one.

To access the individual bits and the current signature, the class should have the methods

```
public long set(int i, boolean b) throws IllegalArgumentException
public boolean get(int i) throws IllegalArgumentException
public long getSignature()
```

The first two methods are used to `set` and `get` the value of the bit `i` of this bit vector. The `set` method should return the checksum after setting the bit `i`. Furthermore, whenever the bit `i` flips to a new value, this method should update the current checksum with the assignment

```
sig = sig^keys[i];
```

or more briefly using the notation analogous to += and such operators

```
sig ^= keys[i];
```

to mix into the checksum the random key that corresponds to the bit `i`. It is now easy to see that for any sequence of bit flipping operations, flipping any bit an even number of times does not affect the final checksum, regardless of the other bit flipping operations. Especially if all bits are turned on and off in an arbitrary order, the resulting checksum ends being the same goose egg as it started as.

With the checksum mechanism available, the following methods should now be embarrassingly easy to make both effective and highly efficient. For the `hashcode` method, you can simply extract any 32 bits of the `sig`, usually the bottom 32. The `equals` method should first compare the checksums, and only if they are equal, compare the actual contents of the bit vectors.

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

(In fact, for the bit vectors generated during the run of the JUnit test, the checksum inequality is always sufficient to infer content inequality! Just don't rely on this happening in general. Even stars occasionally collide with each other in the vast emptiness of space.)

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

As so often in the real world, the `toString` method can return pretty much anything that aids your debugging purposes instead of hindering them. Finally, implement the important method

```
public Object clone()
```

This method should first call `super.clone` to duplicate the actual bitwise identical clone object, since this method has an **implicit contract** of never using the operator `new` for this purpose. Since the superclass version of `clone` inherited from `Object` performs a **shallow copy**, you need to perform the required **deep copy** by yourself and `clone` those freaking `bits` so that these bit vectors can later be safely mutated without silently affecting any other bit vectors that were supposed to be separate entities.

(Those among us who truly are of pure heart must also actually `clone` these `bits` instead of resorting to `Arrays.copyOf` or similar. Such methods will certainly be using the operator `new` behind the scenes, causing us to negligently violate the sacred implicit contract passed from the subclass to the superclass without saying a word in the class itself.)

Lab 55: Accumulated Wisdom

JUnit: AccumulationTest.java

Numerical arrays allow constant-time **random access** to their individual elements, regardless of their positions. The situation is not equally serene for **summation queries** performed over arbitrary subarrays, seeing that those require looping through that subarray to add up its elements one by one. However, with the aid of an **accumulation array** constructed from the original array, arbitrary summation queries over contiguous subarrays can be performed in guaranteed constant time, regardless of the size and position of that subarray!

In fact, since the elements of the original array can be accessed as trivial summation queries for **singleton** subarrays, you can potentially save a bushel of memory by throwing out the original array once you have constructed its accumulation array. The accumulation array contains all the information available in the original array, but in a form more suitable for subarray summations. The small price to pay for this convenience is having to use more bits to store the individual array elements.

An accumulation array has the same dimensions as the original array, but each element equals the **sum of all elements up that particular position**. For example, the accumulation array for the original array `a={4, -2, 3, 1}` would be `acc={4, 2, 5, 6}`. To construct this accumulation array in linear time, we simply need to realize that each element `acc[i]` can be computed with the expression `acc[i-1]+a[i]`, utilizing the fact that the sum of the elements up to the position uses the same terms as the accumulation up to the previous position, with only one more original element included. The initial element is set up with `acc[0]=a[0]` as a base case for the formula.

In your labs project, create a new class `Accumulation`, and in there two `static` methods

```
public static int[] accumulate1D(int[] items)
public static int subarraySum(int[] accum, int start, int end)
```

The first method creates and returns the accumulation array for `items`. The second method uses the given `accum`ulation array to quickly add up the elements of the subarray of `items` from `start` to `end`. (Yes, despite the fact that this method never even sees the original `items` array object!) Also note that these parameters follow the Pythonic convention where the `end` position is treated as **exclusive**, whereas the `start` position is **inclusive**. Therefore when both of these parameters are equal, they span an **empty subarray** whose sum is zero.

(After completing the accumulation logic for one-dimensional data, the reader might take a breather to philosophize about the usefulness of computing a second-order accumulation array whose elements are the accumulated sums of the accumulations of the original elements. Elegant facts await us there.)

Next, we shall generalize this technique to handle **grids**, two-dimensional arrays whose every row has the exact same number of columns, so that no row is **ragged** and leave gaps or protrusions in

the general rectangular shape of that array. Each element `acc[row][col]` of this accumulation array now equals the sum of elements in the rectangular subarray from the **origin** position `[0][0]` up to and including the position `[row][col]`. Note how the previous one-dimensional accumulation is merely a special case of this two-dimensional accumulation done to a two-dimensional array that contains only one row and is therefore effectively one-dimensional. The corresponding methods for you to write are this time

```
public static int[][] accumulate2D(int[][] grid)
public static int subrectangleSum(int[][] accum, int row, int col, int h, int w)
```

In the second method, the parameters `row` and `col` determine the top left corner of the rectangular subarray of the `grid` whose elements are being summed. Parameters `h` and `w` give the height and width of this subarray, measured in rows and columns. For example, if both parameters `h` and `w` were to equal one, this sum would equal the singleton element `grid[row][col]`.

Analogous to the one-dimensional case, the computation of each element `acc[row][col]` should not require brute force looping through the entire subarray. Instead, each `acc[row][col]` should be computed by adding and subtracting a fixed-size handful of previously computed values from the two arrays `acc` and `grid` in a more sophisticated manner that, as a nifty teachable moment for us, showcases the inclusion-exclusion principle sometimes seen in discrete math calculations!

(This is one of those problems where you should literally draw a picture to help you maintain all those parameters and their meanings straight in your head.)

After completing these two methods, instead of mindlessly repeating this approach once more with a feeling for three-dimensional **cuboids**, we shall instead finish this lab in style by solving a classic coding job interview chestnut. Like all the best problems, this one is simple enough to state in one sentence: Given a two-dimensional `grid` of `boolean` truth values, return the side length of the **maximal square** inside it whose every element is `true`.

```
public static int largestTrueSquare(boolean[][] grid)
```

Every "Shlemiel" would solve this problem by throwing at it some horrendous amount of brute force, in here **four** levels of nested loops. The outer two loops iterate the positions of `grid` that could potentially be the top left (for a symmetric approach, the bottom right) corner of the largest `true` square. For each such potential corner, the two inner loops gauge the size of the `true` square that emanates from that corner. For arrays that have been randomly filled with truth values taken from random coin flips, the average running time of this technique would be pleasantly low. However, the worst case would absolutely kill us. Simply imagine what would happen to the running time if every element of the original array were `true`!

The attempt to make the method more clever so that it knows to terminate in this special case can always be thwarted by adding enough `false` values to tactically chosen positions to coax out the **quartic** worst case behaviour. You have to implement the method first, and only after that, the

adversary is made to choose the arguments given to your method. Since this nasty little imp who we already know and hate from **Murphy's law** will *always* make *everything* turn out the worst possible way for your algorithm, the average running time over all possible arguments is rendered meaningless. We cry out for a more sophisticated approach to defeat the adversary by solving the problem with only two nested loops whose running time is **oblivious** to the exact values of the array elements, and deterministically depends only on the size of the `grid` itself! The adversary can then fill in this `grid` literally any which way he wants, and still has no chance to gum up the works by making your algorithm run back and forth in some maximally inefficient and silly manner.

Even though the following approach technically requires some techniques of **dynamic programming**, taught in a course normally taken during the second or third year of a typical computer science undergraduate program, this author hopes that completing the previous methods has psychologically primed the student for the flash of insight needed for the correct approach. Start by creating a two-dimensional integer array `int[][] s` of the exact same size as the original truth value `grid`. Fill up this integer array so that each element `s[row][col]` gives the size of the largest `true` square whose bottom right corner lies in that position. Once this array has been completely filled so that each element is correct , its maximum value is returned as the final answer.

The outer two loops are exactly the same as they were in Shlemiel's solution. However, how might the three neighbouring values `s[row-1][col]`, `s[row][col-1]`, and `s[row-1][col-1]` adjacent to the north, northwest and west to the current position be used to compute the correct value of the current position `s[row][col]` without **any** inner looping, assuming that your two nested for-loops have already correctly filled the values of these three neighbouring elements? Once you see this, that's all she wrote about this problem.

The following optimization is not required for your code to pass the JUnit test in reasonable time, but observant viewers may still note that to save another big bushel of memory, the array `s` does not actually have to be the same size as the original `grid`. This array can simply act as a **sliding window** to keep track of only two consecutive rows; the **current row** that is presently being filled in the inner loop, and its **previous row** whose values are needed for correctly filling the current row. Once the current row has been filled, its previous row will never again be needed, assuming that you keep track of the overall maximum value that has been seen in all previously generated rows, even those already memory-holed.) The current row becomes the new previous row, from which the next current row (recycling the space of the now redundant previous row) is computed.

One final optimization hides inside the previous optimization. There is no need to explicitly copy the elements of the current row to the previous row every time the sliding window is moved one notch downward. Once we realize that those things that walk, look and quack like two-dimensional Java arrays are under the hood one-dimensional arrays whose elements themselves are one-dimensional arrays, swapping these two rows in constant time achieves the desired effect.

One final thing that you might want to notice is that this very technique can be used to solve any well-formed recursive equation whose subproblem parameter domains are contiguous subsets of integers. You can even fill in the subproblem solutions in literally any order (yes, Lana, *literally*), as long as your filling order guarantees that when at some particular position, the positions that store

the recursive subproblem solutions needed to fill in the current position have already been computed. Recursive calls to solve these subproblems can be replaced by trivial array lookups!

A ravishing beauty of a technique, **dynamic programming** is truly a skeleton key to unlock a thousand doors that Hercules himself could not pry open with sheer brute force. The hardest part of all such problems is always coming up with the recursive equation to describe the system. Once you have that equation written down, you should always draw the picture first (even if only as a mental image) to make the actual coding stage as mechanistic and predictable as any kabuki act.

Lab 56: `find()` Me A Find, `catch` Me A Catch

JUnit: MatchmakerTest.java

In the famous stable marriage problem in **combinatorics**, a group of *n* boys is to be paired one-to-one with a group *n* girls. Each boy and each girl has privately **ranked** all the members of the opposite sex in some total preference ordering. Everyone is honest about their true preferences without shame, not trying to go after some hot babe just because that is what the mass media told them to do while every other guy also tries to do the same.

An arrangement that pairs each boy to exactly one girl is called a **matching**. When both boys and girls are numbered from 0 to *n*-1, such a matching can be expressed as a *n*-element array that describes which girl each particular boy has been paired with. Since this array must be a **permutation** of *n* elements, its inverse permutation describes which boy each girl has been paired with. This problem is completely symmetric between boys and girls, but is conventionally phrased in line with traditional gender roles so that the boys take turns to individually approach girls.

Suppose that the matching contains two pairings Chad-Becky and Norbert-Stacy so that **both** Chad and Stacy would prefer to break off their current pairings to elope together, leaving Norbert and Becky in the dust to settle for each other. The existence of even one such unstable pair makes that entire matching **unstable**. Note that merely having Chad desire Stacy (or just Stacy desire Chad) over his current partner does not make that pairing unstable, unless Stacy (Chad) is equally willing to break away from xir current pairing.

Regardless of how each boy and girl privately ranks the members of the opposite sex, at least one **stable matching** with no unstable pairings will always be exist. In fact, depending on how these preferences mash and conflict with each other, more than one stable matching may well exist, even exponentially many! For example, consider the situation for $n = 2$ with two boys Chad and Norbert and two girls Stacy and Becky. Chad prefers Stacy over Becky, whereas Norbert prefers Becky over Stacy. So far, no conflict. Unfortunately, the gals have opposite preferences so that Becky prefers Chad and Stacy prefers Norbert, setting up the scene for some raucous nineties teen comedy. Both possible matchings in this situation are stable, as the reader can verify. However, these two matchings differ drastically in their **fairness** for whose preferences are satisfied. One of these matchings gives each boy his preferred girl whereas both girls have to settle for her lower-ranked boy, whereas the other matching gives each girl her preferred boy by making the boys settle for their respective lower-ranked girls.

The famous Gale-Shapley algorithm can rapidly construct the matching that is optimal for boys and pessimal for girls. In the guaranteed result, each boy skips to the loo with his highest-ranking "best gal" that he has any chance to pair with at all in any possible stable matching. Conversely, each girl will be paired with her lowest-ranking boy that she could pair with without making the matching to be unstable. Since the roles of boys and girls are perfectly symmetric, this algorithm could be run as its mirror image version to favour girls instead of boys. In our more enlightened times since the swinging sixties back when this algorithm was first developed, it also serves as an example of **algorithmic bias** that systematically oppresses a particular subgroup of participants. (As in so

many other walks of life, even those instinctively sense that the whole system is somehow rigged against them still only perceive the surface level of how deep this rigging *really* goes.)

The Gale-Shapley algorithm makes for an excellent exercise of handling two-dimensional integer arrays in this course. To visualize this algorithm, imagine these boys and girls standing near the opposite walls of some high school gym, everybody dressed and made up as if they were cast in the summer stock revival of *Grease*. After setting up the necessary arrays to keep track at the current state of this high school dance, the algorithm itself is just while-loop that keeps running until every boy have been matched to some girl.

On his turn, each boy approaches his highest-ranked girl that he has not approached before during this process. Since the Bro Code is still decades away from being developed, the boy must approach that particular girl even if she has already been provisionally paired with someone else. If Stacy is still unpaired at the time when Norbert approaches her, they are provisionally paired with each other for the time being. However, if Stacy has already been paired with someone else, say, Melvin, the one of the two boys currently vying for Stacy's hand will get sent slinking back to the opposite wall with his shoulders drooped while the other one stays to twist his limbs in victory is decided by Stacy's preference between these two suitors.

Whichever boy was sent back will get his turn again in the future, but may not approach Stacy again (as any repeated approach would be futile anyway), so on his next turn, he will be approaching the girl next down the line in his preference rankings. Since the preference ranking of each boy contains exactly n girls, and each boy gets to approach each girl at most once, every boy will necessarily become paired with some girl after at most n^2 total approaches, usually substantially fewer.

In your labs project, create a class `Matchmaker`, and there the method

```
public static int[] galeShapley(int[][] boysPref, int[][] girlsPref)
```

that receives the preference rankings of both boys and girls as n-by-n integer arrays so that the i:th row of each array contains the preference ranking list of the i:th boy or girl, respectively. This method should create and return a one-dimensional integer array `boysPair` whose each element `boysPair[boy]` shows which girl that particular `boy` has been paired with. During the execution, you will also need two other one-dimensional integer arrays `girlsPair` and `boysPos`. The former keeps track of which boy each girl has been paired up with, which is necessary for deciding whether an existing pair should be split up. The second array keeps track of how far down the preference ranking each boy has descended during the algorithm execution.

Curiously enough, the unique end result of this algorithm does not depend whatsoever on the order in which the individual boys get to approach these girls. Confident of the final outcome that is set in stone by the iron hand of mathematical necessity from the preference rankings of these individual boys and girls, Chad can be a suave gentleman and nonchalantly comb his hair in an imitation of Kookie Byrnes while he lets Norbert to first approach Stacy, coyly waiting near the other wall presumably wearing some kind of pencil skirt and bobby socks with visions of of hunky Chad to sweep her away from chunky Norbert.

The bottleneck for the execution time of this algorithm is quickly finding some unpaired `boy`, again literally any such `boy`. Details of how to best implement such **nondeterministic** just-pick-any-one-I-don't-care-which-one selection to run efficiently are left for the reader; several good ways exist to do so. Furthermore, since this algorithm needs to be able to quickly determine whether Stacy prefers Norbert or Melvin; some preprocessing of the `girlsPref` array might help Stacy make up her mind faster during the algorithm main loop. The girl that the `boy` will approach in his current turn is then given by the expression

```
boysPref[boy][boysPos[boy]++]
```

that **post-increments** the `boysPos` counter for that particular `boy` so that the next time when his turn to approach a girl comes, he will approach the next girl down the line along his preference ordering. Since the pre- and post-increment operators are a highly convenient feature of Java that Python doesn't have at all, we might as well smoke 'em since we got 'em, in this rare occurrence of the Java syntax being superior to that of Python for the everyday programmer.

As discussed earlier, the result produced by the Gale-Shapley algorithm is optimal for the boys and pessimal for the girls. Advanced algorithms for the stable marriages and other matching problems attempt to make such unfairness a bit less lopsided.

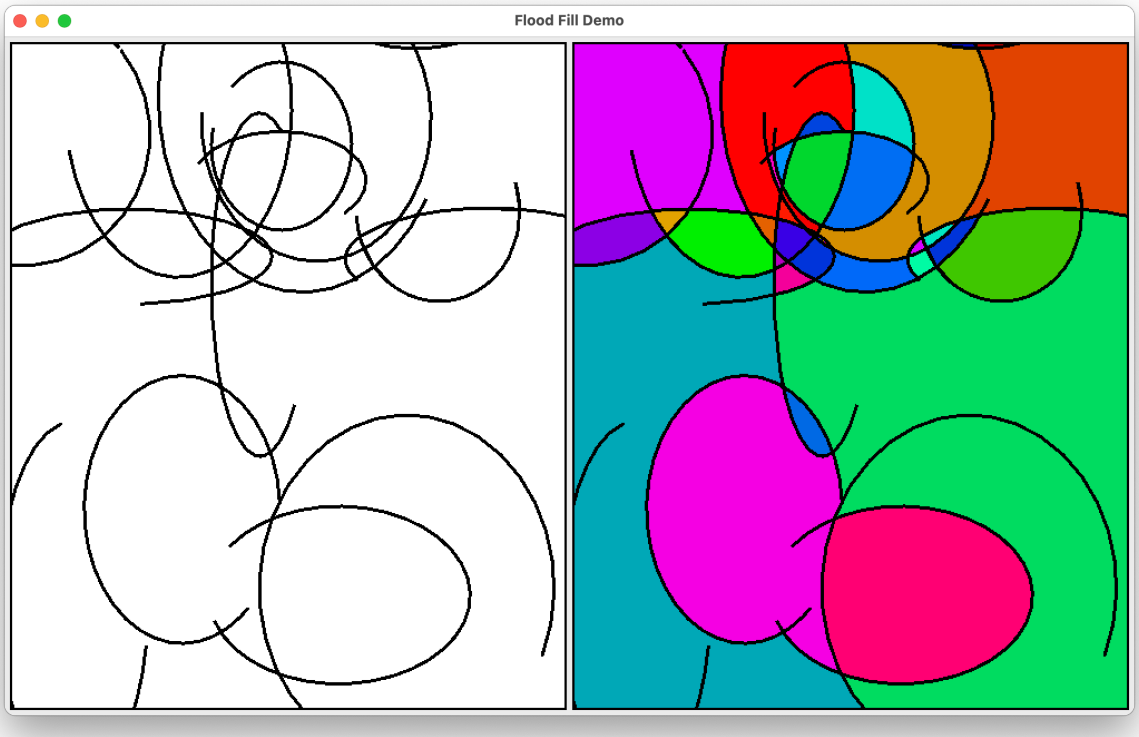
In the more general **bipartite matching** problem, *n* boys are to be paired of with *m* girls, where *n* and *m* are not necessarily equal. The terms "boys" and "girls" are mere placeholder names that can stand for anything in the **bipartite graph** whose nodes they comprise; for example, **hospitals and residents** in the most famous real-world practical application of the stable marriages problem. This problem setting can be further generalized so that instead of being gathered all together in the gym and everybody seeing everybody else, each boy knows only some subset of girls, and each boy knows only some subset of girls. Hall's Marriage Theorem, a famous result of graph theory, shows that some matching exists in the first place if and only if every subset of boys together knows at least as many girls as there are boys in that subset, and vice versa. (As usual, one of those ifs is trivial while the other one is world-changing.)

Any societal implications of this phenomenon in the real world are left was an exercise for the reader. In spirit of these modern times, this problem can be further generalized for the somewhat euphemistically named stable roommate problem where the participants are not divided into binary sexes but anybody can be paired with anybody. Unfortunately, no algorithm can now possibly guarantee finding some stable matching in all situations, for the simple reason that some preference ranking combinations just do not allow *any* stable matching. The simplest example of such situation is when Andy prefers Bob who prefers Carl who prefers Andy to create a **non-transitive cycle**, and all three men equally agree on their dislike of Dave whose preferences don't really matter here. (This is somewhat analogous to Arrow's Impossibility Theorem for inconsistent group preferences even when the preferences of each individual are perfectly coherent.)

Lab 57: Flood Fill

JUnit: FloodFillTest.java

Given a pixel image that consists of a solid background colour (in the example on the left below, white) that has been divided into disjoint areas using lines, curves and other shapes drawn with other colours, flood fill is a classic technique to paint over all pixels inside the same contiguous area. The flood fill algorithm carves its way through precisely the pixels that belong to the same disjoint area, regardless of the complexity of the shape of that area and the curvature of its boundaries. The area to be painted is defined by any given **seed** pixel (x, y) that is known to be inside that area. See the screenshot from the `main` method of the automated test below for illustration for the original image or random arcs on the left to create a maze, and a whole bunch of flood fill operations separately applied to its white areas, using a randomly chosen colour for each such area.



The basic flood fill algorithm is explained well enough in pseudocode in the linked Wikipedia page, from where your task is to translate this given algorithm to Java. For once, you don't need to come up with the algorithm yourself from the specification for what it is supposed to achieve! However, your flood fill method certainly should not be recursive, since the recursive flood fill that snakes through the white area in the maximally convoluted fashion will almost certainly require a large recursion depth and run out of the allocated stack space for recursion bookkeeping. Solutions that work for small arrays may be prohibitively expensive for large arrays whenever the growth rate of the algorithm running time is not linear. Instead of once again heading over to Stack Overflow to ask

how to increase the recursion limit inside your Java Virtual Machine, your method should be non-recursive from the get-go, and instead use some kind of **queue** of pixels waiting to be processed.

In your project folder, create a new class `FloodFill`, and there the method

```
public static void floodFill(BufferedImage img, int x, int y, Color c)
```

Given an instance of `BufferedImage` and the pixel coordinates `x` and `y` of the pixel inside the current area, this method should paint over that entire area, and not one pixel more and no less, with the new colour `c`. Instead of the more general `Image`, this method expects its more specific subtype `BufferedImage` that allows additional methods `getRGB` and `setRGB` to read and write the colours of its individual pixels. Note that these two methods expect the colours to be represented as primitive `int` values, whose four bytes encode the ARGB components of the actual colour, one byte per colour component. However, this representation should really not be a problem, since the Java `Color` objects come with the method `getRGB` to extract this compact `int` representation from that object. You don't need to perform any bitwise arithmetic to convert the colours to bytes and vice versa.

The original colour of the seed pixel (*x*, *y*) is treated as the background colour that the flood fill should paint over, whereas every other colour found in the image is considered to be a wall that prevents the current fill colour from spreading over the entire image. Note also that the return type of this method is `void`, so that this method should modify the contents of the original `img` object shared by the caller and this method, instead of creating a brand new image to represent the result while keeping the original `img` intact.

Since Java does not have a `Pair<Integer,Integer>` data type (let alone the **tuples** that we know and love in Python) whose instances could be pushed and popped into queues, the easiest way to operate here might probably be to use two separated instances of `ArrayDeque<Integer>` operating in lockstep, so that one of these queues stores and *x*-coordinate and the other stores the *y*-coordinate of the pixel. When pushing the pixel (*x*, *y*) into the virtual queue, push the *x*-coordinate into one of these queues, and the *y*-coordinate into the other. Symmetrically, to pop out the next pixel to be processed, pop its *x*- and *y*-coordinates separately from these queues. Parallel arrays are not pretty, but get the job done in languages whose type system is stuck in the eighties.

After initializing your necessary variables and the queue data structure, the flood fill algorithm is yet another application of the classic **breadth-first search**, already seen a couple of times before in this problem collection. Initially, the **frontier queue** contains only the seed pixel (*x*, *y*). The algorithm main loop keeps going as long as the frontier queue is nonempty. In each round of this main loop, pop out the next pixel to be processed from the front of the queue. If this pixel at the current moment has some other colour than the background colour, just ignore it and go to the next round without passing Go in between. Otherwise, colour that pixel with the new colour `c`, and push its **four** neighbours to the main compass directions into the queue. The frontier queue will become empty precisely when all the background pixels reachable from the seed pixel have been processed and painted with the new colour.

To simplify and streamline the translation of the algorithm from the Wikipedia pseudocode presentation to actual executable Java, it might be convenient to first define the utility method

```
public static void inside(BufferedImage img, int x, int y, int bgCol)
```

that checks whether the pixel (x, y) lies inside the dimensions of `img` and that its colour is still equal to the given background colour `bgCol`. Note the methods `getWidth` and `getHeight` of the `BufferedImage` class that give you the exact pixel dimensions of that image.

The `main` method of the automated test tries out your flood fill method to all of the areas of the pseudorandom image of black curves and lines seen in the above screenshot. Since your method should produce that exact same result when applied to the pseudorandom test image, you can use this `main` method to verify by eyeballing that your flood fill algorithm works as intended. The actual `testFloodFill` method does not display the thirty pseudorandom images of various dimensions that it creates and paints over using your flood fill method. Instead, the test method computes the checksum from all the pixels of each of these pseudorandom test images, to ensure that your flood fill algorithm fills in **precisely** the same pixels with the same colours into every such test image as this author's private model solution would.

Once they get the basic flood fill method working, interested students can also try out implementing the more advanced and memory-efficient **scanline fill** techniques also described in the Wikipedia page. Even though we are most familiar with the result of flood fill from the now meme-worthy MS-PAINT images, this algorithm allows many optimizations for both time and space that, back in the day, allowed it to run even in an eighties 8-bit home computer such as Commodore 64 and Sinclair ZX Spectrum, both of which this author still has both fond and not so fond memories of.

Despite being adorably primitive from our point of view up here in the cyberpunk future, the Basic programming language of the ZX Spectrum even had the flood fill operation available as a keyword in the language, along with the more elementary graphical primitives such as drawing a line segment or a circle. However, the mighty one megahertz processing power of the Z80 processor could only execute this operation so slowly over the 256-by-192 screen resolution (and even that split into 8-by-8 regions with only two colours used per region) that you could watch the algorithm paint the pixels, almost as if the whole thing were some animated YouTube demo of the algorithm.

(To see how far humanity has gradually advanced in this respect, and what the exponential growth of processing power means in practice, you can try out this author's hand-rolled and non-optimized Mandelbrot.java demo that not only flood fills the Mandelbrot fractal image with twenty times as many pixels, but determines the positions of the walls and the colour of each pixel on the fly from a complicated iterative process of high-precision complex number arithmetic!)

Lab 58: Block Cellular Automata

JUnit: BlockCellularAutomatonTest.java

Various cellular automata, the most famous of which surely are the variations of Conway's Game Of Life, produce a surprisingly chaotic emergent complexity of interesting patterns from the local interactions of applying simple rules uniformly over a grid of truth-valued cells. We have already seen Conway's Game of Life as one of the example programs `GameOfLife.java` in this course material, and the Ulam-Warburton crystals as an earlier lab problem in this very document.

Block cellular automata are an interesting generalization of the basic idea of cellular automata. In a block cellular automaton, the local rules are not applied directly to the individual cells, but to some larger neighbourhood, in the simplest nontrivial case a 2-by-2 square that the grid has been subdivided into. For brevity, each such 2-by-2 square is called a **Margolus square**. Since the width and the height of the entire grid are assumed to be even, this subdivision neatly places every cell into exactly one Margolus square. Since each `boolean` cell inside the same Margolus square can be in two possible states (as usual, these states are again colourfully called "dead" for `false` and "alive" for `true`), that Margolus square can be in one of its $2^4 = 16$ possible configurations.

Same way as in all other cellular automata that we have seen so far, a block cellular automaton executes forward in discrete time steps. In each time step, the hardcoded rule is used to separately convert each Margolus square to its next configuration. So that these squares could interact with each other instead of being small isolated islands, the origin location from where the grid is subdivided into these squares is placed at position (0, 0) at the even time steps, and at the position (1, 1) at the odd time steps. So that the cells on the edges of the grid will also get properly treated in both odd and even time steps, this grid is treated as **toroidal**, meaning that its east and west edge are treated as to have been logically wrapped together. The wrapping of its north and south edges is analogous. For a cell located at the right edge of the grid, its immediate right neighbour lies on that same row at the left edge, and vice versa.

The four truth values inside the Margolus square are treated as four bits of the binary representation of an integer, so that the northwest cell adds 1, the northeast cell adds 2, the southwest cell adds 4, and the southeast cell adds 8 to the value of the square. For example, the northwest and southeast cells of some Margolus square are currently alive, and the other two cells currently dead, that square as a whole will have the numerical value of 1 + 8 = 9. Some other Margolus square whose all cells are dead would have the numerical value of zero, whereas a third square whose all cells are alive would have the maximum numerical value of 1 + 2 + 4 + 8 = 15.

Since the chosen rule for the block cellular automaton turns the configuration of each Margolus square into another configuration of that same square, the entire `rule` can be compactly expressed as an array of 16 integers. Once you have computed the numerical `value` of some Margolus square, the numerical value that square will simply equal `rule[value]` in the next time step. You can then convert this numerical value back into the individual dead and live cells inside that Margolus square by expressing that value in binary and making precisely those cells be alive in the next time step for which the bits for the corresponding powers 1, 2, 4 and 8 are turned on in that number.

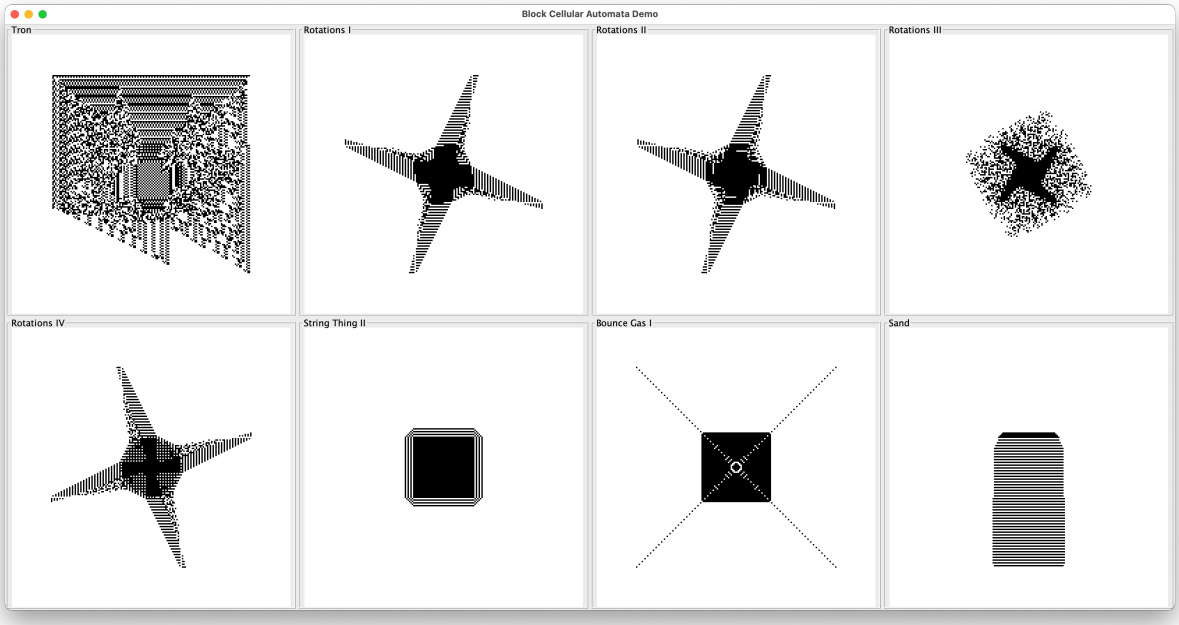
In your labs project, create a new class `BlockCellularAutomata`, and there the method

```
public static int[] margolusNextState(boolean[][] current, boolean[][]
next, int[] rule, int off)
```

that is given the `current` state of the grid of cells, from which this method computes the `next` state of grid after using the given `rule`. The parameter `off` indicates whether the current time step is odd or even, so that this parameter equals 0 for even time steps, and 1 for the odd ones.

Note that this method does not create and return the new `next` array object, but the caller must give this method as a parameter the array object `next` to be filled in. In the automated JUnit test, this `next` array is guaranteed to have the exact same dimensions as the `current` array from which the next state of the block cellular automaton is computed. This handy scheme eases the burden of the Java garbage collector when the execution thread of the `main` method of the JUnit tester (provided by the instructor here, you don't need to worry about it) does not constantly have to be creating new grid arrays only to then throw these perfectly good brand new array objects into garbage at the very next time step! Instead, this animation logic can simply keep recycling the same old array objects back and forth to keep track of the current state and the next state of the automaton, interchanging the roles of the two array objects at every time in this dance.

The JUnit test class for this lab also contains a `main` method that launches eight separate instances of a block cellular automaton, each instance using eight aesthetically behaving rules copied from the page "Reversible Cellular Automata". When run with your method, the initial state of a solid block of live cells as the center of each grid should evolve towards the screenshot below.



Students interested in GUI programming with Swing should again note in the `main` method that out of the box, the `JFrame` instances that represent the underlying GUI windows inside Java and Swing do not obey the close button and go gentle into that good night. Instead, they will rage, rage against the machine by ignoring the order to ride into the valley of unreachable objects whose role is only to do or die without making a reply. Normally we use the method `setDefaultCloseOperation` to tell a `JFrame` instance to become more a bit more respectful of the chain of command. However, we cannot do so in this example, since launching a new `Timer` instance creates a new execution thread that we must ensure will also cease to run when the user closes the window. Otherwise the Java Virtual Machine process would not be able to terminate, since it has to keep the timer ticking like a forgotten metronome that nobody is listening to any longer!

As an interesting aside, all instances of the Swing `Timer` are managed by the same execution thread that internally keeps all active timers in a **priority queue** data structure (presumably implemented as a **binary heap**) to quickly find the active timer that currently has the least amount of time left until its next moment to step in the limelight. This common thread that manages all timers will repeatedly `sleep` until that time to shine comes up for the first timer waiting in this priority queue. It then executes the action associated with that timer, and pushes that timer back to the priority queue where the binary heap logic quickly moves it to its correct place in the priority queue. Unlike actual execution threads whose creation is expensive, you could easily launch thousands of timers to simultaneously tick in background in arbitrary speeds! How wonderful it is that things that `sleep` quietly so that the processor does not need to be constantly checking up on them do not consume even one processor cycle by their mere existence in the memory!

In addition to the eight example rules illustrated in the test class, this mechanism allows so many possible rules even for 2-by-2 squares and `boolean` cells that humanity will surely never explore the possibilities of but a tiny portion of these $16^{16} = 2^{64}$ possible rules. Even if the `rule` array is restricted to be some a permutation from 0 to 15, this still leaves us 16! = 20,922,789,888,000 possible rules to play with. All such bijective permutation rules are guaranteed to be **reversible**; such an automaton can always be run backwards in time from any arbitrary future state back to the starting state and even past that merely by inverting that permutation to use as a the new rule! (Such an automaton cannot have any Gardens of Eden.)

Restricting these rules even further to those that are symmetric to all four compass directions will also drastically lower this count. Furthermore, the configuration where all cells are dead is usually restricted to stay dead (that is, a legal rule must map the Margolus squares of value zero to the same squares of value zero), as is the case with all of the rules used here. This still leaves us 15! possible rules to explore. In absence of this natural restriction of nothing ever coming out of nothing, the entire automaton grid would immediately fill with nonzero stuff to create an annoying flashing, as is the case with the famous Critters rule.

The block cellular automaton mechanism also trivially generalizes to larger k-neighbourhoods, of which Margolus is a trivial special case of $k = 2$ (the neighbourhood shift offsets then must also use the same period of k), and a larger number of possible states for the individual cells (usually then called the **colour** of that cell), with the number of possible rules blowing up exponentially into unimaginable numbers. Such automata can create generative and computational art. See the page "6-Colour Block Cellular Automata" for some illustrative examples of this kind of tomfoolery.

Lab 59: Hitomezashi Golygons

JUnit: BlockCellularAutomatonTest.java

This problem was inspired by the Numberphile video "Hitomezashi Stitch Patterns" that the reader might want to first check out to get a quick idea what these patterns are all about. (You should also note the clever mental technique used in this video to create a random bit pattern!) The blog post "MathArtChallenge Day 14: Hitomezashi stitching" at Arbitrarily Close also succinctly explains the mathematics behind these patterns. Such patterns are used in traditional Japanese stitching to create regular designs that are both useful and pleasant to look at. However, sprinkling just a pinch of randomness on this surprisingly simple process reveals an emergent complexity that was waiting underneath the surface this whole time for its chance to blossom!

A **hitomezashi** pattern is drawn on a rectangular area that has been subdivided into a grid of squares. In the space that separates each row of squares, a horizontal stitching line is drawn so that a line segment is rendered visible only under every other square, as if creating a stitch on a fabric with a needle and a thread. Each row is initially assigned a truth value to decide whether the squares in the odd positions or in the even positions get a visible line segment under them. This same process is then repeated for all the vertical columns of squares, to complete the pattern.

If the rows and columns are assigned truth values in some simple repeated patterns, the entire two-dimensional pattern will end up being uniformly repeated with the same period. However, a random assignment of truth values to each row and column produces a chaotic fractal pattern of golygon shapes whose every edge has length of exactly one. As long as there is room inside the shape, they properly contain smaller such fractal shapes tucked inside them, with none of these inner shapes ever abutting any of its brothers or the larger shape that immediately surrounds it.

This lab is another exercise of how to create and draw on `BufferedImage` objects to create two-dimensional pixel raster images. Create a new class named `Hitomezashi` in your labs project folder, and in that class the following `static` method

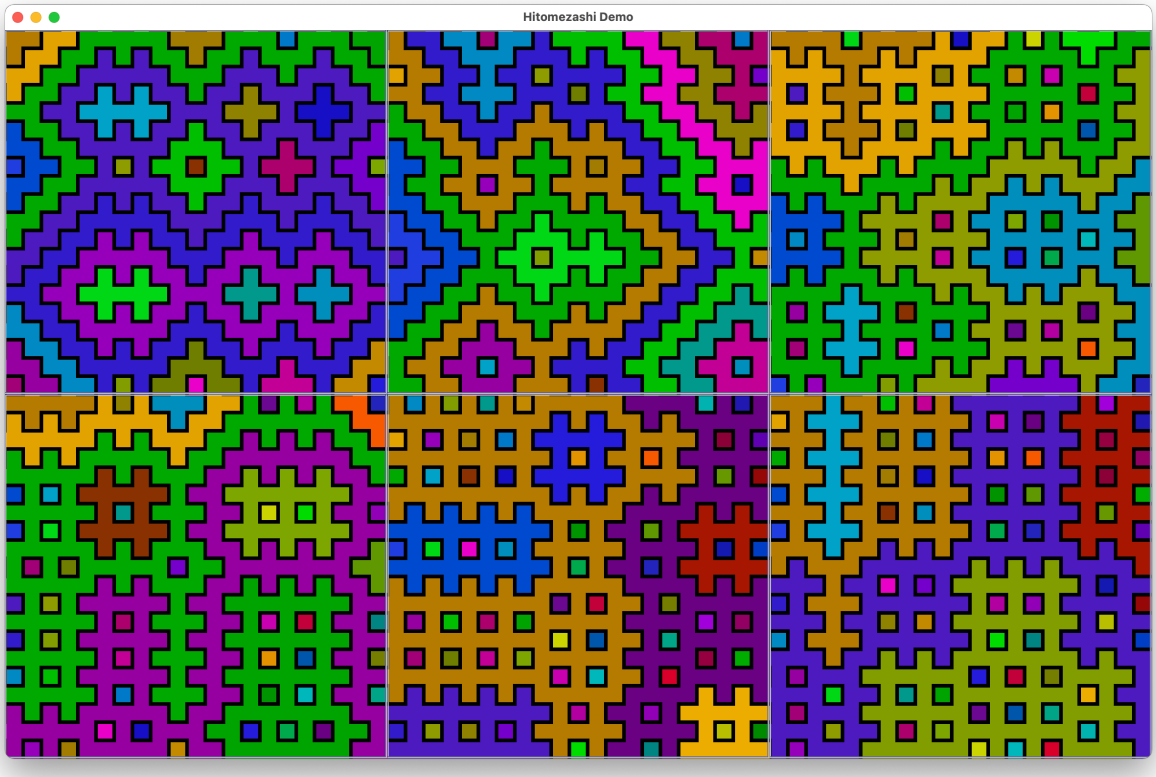
```
public static BufferedImage createPattern(int width, int height, int square, boolean[] horizontal, boolean[] vertical)
```

The first two parameters `width` and `height` define the dimensions of the pattern, as measured in squares, and the parameter `square` gives the size of each individual square in pixels. For example, if both `width` and `height` equal 25, and `square` equals 20, the pattern created by this method would be 25*20 = 500 pixels wide and tall. The bit vectors `horizontal` and `vertical` determine the parity of squares that get a visible line segment, given separately for both rows and columns.

From these parameters, this method should create a new `BufferedImage` instance producing that pattern as the line segments rendered in black against a white background. The instructor provides a main class `HitomezashiMain.java` that calls this method with six different random horizontal and vertical bit vectors, and displays the gallery of returned images inside a Swing `JFrame` window. The bit vectors that produce the six patterns of the screenshot below are created by making each bit

to be equal to the previous bit with a probability that ranges from 30% to 80%, with a ten point increment between the successive patterns. Note that to ensure that your code treats both width and height parameters as it should, each of these six shapes is 21 squares wide and 20 squares tall, to catch any accidental misuse of width as height, or vice versa.

Colouring the golygons is not required to complete this lab. You can render only the black line segments that form the outline of that shape**.** Even so, make sure that your golygons end up having the exact same outline as those given in the example screenshot from the main class shown below. However, if you have already completed the Flood Fill exercise earlier in this document, you might as well use its `floodFill` method to paint the contiguous golygons with random colours to make these golygons pop out more powerfully from their statistically similar surroundings.



At least to the untrained eye of this author, these patterns evoke nostalgic visuals of Native American artwork, at least if the colour knob used to fill in these golygons were adjusted towards more earthy and muted tones. Readers who have completed the Flood Fill project can experiment further with such tones. Furthermore, as explained in the above Numberphile video, this technique can be applied to other grids such as **triangular** grids, not just to the familiar grid of integers.

This technique can be further generalized by using some more complex pattern to fill in the line segments between the rows and columns, instead of the regular on-off metronome stitch. If such stitches are rendered randomly using line segments aligned to the four intercardinal compass directions, the result is a maze generator made famous in the classic Commodore 64 Basic one-liner 10 Print, arguably the spiritual great granddaddy of later such generative art.

Lab 60: Bit Deque

JUnit: BitDequeTest.java

Same as other imperative programming languages, Java guarantees that the elements in an array are stored in consecutive memory locations. Assuming that the arrays are **homogeneous** in that each element takes the same number of bytes to store, such arrays enjoy the advantage of being **random access data structures**. This conceptual misfire of terminology has nothing to do with randomness, and would be much more accurately called *arbitrary* access data structure. In a random access structure, as opposed to a sequential structure that must be accessed in order from beginning to end, the element in any position can be accessed for both reading and writing in the same constant time, regardless of the earlier operations performed on that array.

(Okay, technically the above isn't actually true for truly huge arrays, due to **memory paging** and **processor caching** effects. However, this is about as relevant for the discussion here as planning your next trip to supermarket using the mental model of moving on a spherical surface: globally true, yet locally utterly irrelevant.)

The downside of such rigid memory order is that every array object must maintain the same length during its entire lifetime. This makes arrays inconvenient for storing dynamic data structures where the number of elements can increase and decrease during execution. At this point, old books and other such material on data structures used to recommend the use of **linked lists** to represent such dynamic structures. Each element stored in a linked list is kept in a separate **node** object allocated from the heap, requiring all the associated bookkeeping needed for the heap management and garbage collection to take place behind the scenes.

Furthermore, each node object still has to use an additional 8 or 16 bytes to keep track the memory address of its successor and predecessor. Spending all this extra memory in addition to the **payload** of the data elements allows adding and removing an element from a known position in guaranteed constant time, the upside of the flexible chainlike structure of linked lists whose nodes and chains of nodes can be extracted and inserted in arbitrary positions in constant time. Arrays can contain only the payload data bytes, and don't need anything else to keep them in order, since the memory hardware already does that job for them for free.

However, sometimes we know for a fact that elements will always be added and removed at the end of the current structure. Consider the `StringBuilder` class of the Java standard library, where new characters and strings are always appended to the end of the string that is currently being built piecemeal. A linked list would be overpaying for overkill in such situations. A superior technique uses an ordinary array to store the payload elements, with a separate integer data field (or two) on the side to keep track how much of this array is actually being used to store the current payload cargo. The positions that do not currently contain any payload elements are **slack** available for future additions elements to the structure. The element values in the slack area of the array can be anything, since they will never be read out as actual elements.

The standard terminology is to define the **capacity** of the structure to equal the actual `length` of the underlying array, and define the **size** of the structure to equal the number of actual payload

elements stored inside the array. This terminology can be seen in the public interface of the good old collection workhorse `java.util.ArrayList<E>`, for example. Subtracting these two quantities therefore gives us the amount of slack space currently available. As long as slack space exists, adding new elements to the structure is a simple constant time operation, regardless of the size and the capacity of the structure. Once the slack runs out at the time when a new element is being added, a new array twice the size the previously used array is created to copy the existing elements into. The structure then says *sayonara* to the old array as thanks for its completed service, and continues to use the new bigger array from that point onwards, as if that new array had been the original array all along, but now with plenty of slack space now available for future additions. (The convenience methods `copyOf` and `copyOfRange` of the `java.util.Arrays` utility class make this simulated resizing practically a one-liner.)

Primitive arrays are often used to implement various forms of **queues**, a generic name for a certain type of data structure of which the "queue" in the everyday sense of this word is but one possible special case. In the terminology of data structures and algorithms, a queue is something that you can at any time **push** elements into one by one, and can any time **pop** to ask for one element to come out. The actual subtype of the queue determines the element that comes out at the pop operation. A **first-in-first-out queue (FIFO)** behaves like a queue in the everyday sense of this word, popping out the element that has waited in line inside the queue the longest. A **last-in-first-out queue (LIFO)** behaves as a **stack** so that the element popped out is always the one who has waited the least time inside the queue. Both FIFO and LIFO queues have useful applications in many algorithms that we have already seen. For example, the **breadth-first search** algorithm used to solve the "Big Ten-Four" and "Flood Fill" exercises earlier in this document uses a FIFO queue to store the frontier of elements that are waiting to be processed.

Of course, the standard library of not just Java but any programming language that we could ever seriously consider for real work in this millennium offers both these queue structures, and more. Nobody should ever have to implement either one from scratch any more than, say, they would implement their sorting algorithm. (What separates computer science from most other fields is that no problem ever *needs* to be solved twice.) However, in the same spirit as the `NatSet` example class used to store a monotonic set of natural numbers with one bit per element, we realize that these generic queue classes that are able to store any type of elements will be horrendously inefficient with respect to the memory use when used to store primitive data types. You don't have be Uncle Scrooge to viscerally shudder at the idea of wrapping each payload item, itself only a couple of bytes, inside some object that, even in the best scenario, takes at least 24 bytes in the object heap!

In this lab, we implement an array-based a **double-ended queue** data structure to store `boolean` values, for as tight use of memory as this can possibly get. A double-ended queue, or just **deque** from now on, is a linear queue that allows elements to be pushed and popped from either end at any time, for four different operations `pushFront`, `pushBack`, `popFront` and `popBack`. If you have a deque available, you can always voluntarily restrain yourself to use only the operations of the LIFO stack and FIFO queues, making these structures special cases of a deque. The added flexibility of the deque can come handy in special situations... provided that we first figure out a way to implement all these deque operations to work in constant time, at least when **amortized** over a long series of such operations. If each deque operation needs a linear time internal reorganization after every operation, its user code might as well just keep all the data in its own arrays anyway.

In your project folder, create a new class named `BitDeque`. This class will have the following six public methods for you to fill in once we get the preliminaries out of the way. The first four methods implement the deque operations, and the last two allow the user code the query the current size and the current capacity of the queue.

```
public void pushFront(boolean v)
public void pushBack(boolean v)
public boolean popFront()
public boolean popBack()
public int getSize()
public int getCapacity()
```

To store the elements currently inside the deque, this class should first define the data fields

```
private boolean data[];
private int startIdx;
private int endIdx;
```

The elements are kept in the bit vector, here named `data`, that will have slack in both ends to allow pushing and popping elements from both ends in constant time. The position indices `startIdx` and `endIdx` delimit the payload area of stored elements in the middle. Conventionally, the position `startIdx` is treated as being **inclusive**, whereas the position `endIdx` is treated as being **exclusive**. For example, let `data.length==20` so that the array can store twenty elements. Supposing that `startIdx` equals `5` and `endIdx` equals `13`, the deque currently contains exactly eight payload elements, as given by the elegant formula `endIdx-startIdx`. (If the end index were also treated as inclusive "because *muh symmetry*" or some other sentimental *ad hoc* rationalization, you would always have to remember to add one to the result given by this formula. Other benefits of using an exclusive end index will soon follow.)

Once you understand how this structure works by packing everything inside a simple `boolean[]` bit vector, all four deque operations can be implemented with just a couple of lines of code, with delightful symmetry to boot! For example, the `pushFront` method will first decrement `startIdx`, and place the new element into the new `startIdx` position. Because the end index is treated as exclusive, the `pushBack` method will first place the new element in position `endIdx` that is known to be the first position of the slack after the payload, and increment `endIdx` after adding the element. (Knowing the distinction between the prefix and postfix forms of the operators ++ and −− will now pay dividends.) The methods `popFront` and `popBack` operate symmetrically to undo the effect of the corresponding `push` and, analogous to the two pushing methods, therefore differ from each other by performing the element insertion and updating the index in the opposite order.

The class should also define the following constructor to initialize the deque. Since there is no point being miserly with individual bits, the capacity of the `data` array will be kept at minimum of 512 bits during the lifetime of the deque, regardless of how many payload elements actually happen to

be stored in the deque. The payload area in the middle is initially empty, but thanks to the decision to use the `endIdx` as exclusive, recognizing this situation and handling it correctly is no problem, since the deque is empty if and only if both indices `startIdx` and `endIdx` are equal.

```
private static final int MIN_CAPACITY = 512;
public BitDeque() {
    data = new boolean[MIN_CAPACITY];
    startIdx = endIdx = MIN_CAPACITY / 2;
}
```

As long as there is slack available in the direction of the push, everything is straightforward. But what happens when this slack inevitably runs out after enough additions? To get out of this predicament, your class should define a private utility method

```
private void expand()
```

that both methods `pushFront` and `pushBack` call when there is no slack available in the desired direction. This method creates a new bit vector **twice the size** of the current data array. After the existing payload elements have been copied to the center of this new vector, the previous `data` array is discarded in the loving arms of garbage collection. The `startIdx` and `endIdx` indices are updated to reflect the correct location of the payload elements in the new larger array, after which there will plenty of slack available in both ends for future deque operations.

Note that the automated JUnit test class will enforce the fact that the threshold and logic of calling `expand` is symmetric in both directions, and your code will not pass this test unless it does this.

For the common use case where the deque is used as a FIFO queue where the payload area is always moving in one direction, there will be plenty of slack available in the other end of the array, all seemingly sitting there doing nothing but twiddling its virtual thumbs. Instead of wasting perfectly good memory by allocating a new bigger array and then continuing to essentially use only one quarter of its space, you can implement two private utility methods

```
private void shiftRight()
private void shiftLeft()
```

that move all the payload elements so that they are again centered in the array. you should use the method `System.arraycopy` to perform this move efficiently in bulk, instead of blithely using a for-loop through the relevant positions and copying their elements inefficiently one at the time. This method has been implemented `native`ly inside the Java Virtual Machine, and will therefore be significantly faster than any for-loop that we could ever write in the higher level of abstraction of Java. Remember to also update the `startIdx` and `endIdx` indices yourself, though, since no method can magically update your fields and local variables for you.

(You may also note how the method `System.arraycopy` shows its age all the way to the internal development versions of Java in that its name does not follow the standard naming convention and

be named `arrayCopy`. This method has been grandfathered in same as `Color.Black` and other named constants for common colours that should have originally been named `Color.BLACK` in accordance to the all-uppercase naming convention of Java named constants.)

An alternative solution to using the previous two methods, although somewhat more complicated to code, is to treat the `data` array as being **circular**, so that either index that goes past the edge of the array will appear at the other edge. This scheme is more efficient in its memory use in that the space available on one side is available for pushing elements from the other side. Data array expansion would be triggered by a push operation that would make both `startIdx` and `endIdx` become equal, a situation that was supposed to mean that the entire deque is empty. The expansion code will also become somewhat more complicated. Students who implement their deque this way need to be mindful of edge cases (in this case quite literal) where these indices are pointing to the first and the last position of the array, which can happen either way during the use of the deque. Even outside this edge case, the contents of the deque are generally split into two contiguous subarrays, and the expansion must correctly copy correctly in the newly allocated larger array.

The last situation that you unfortunately have to consider in this lab is when a large number of elements is first pushed in the deque, after which then almost all these elements are popped out. It seems very inefficient to end up using a giant array to store a handful of elements, most of this giant array ending up as slack that may never be filled in again. Even if we knew for certain that it will be filled in later, all those megabytes might as well be put in some more productive use before that happens!) (A megabyte here and a megabyte there, and pretty soon you are talking about real memory.) An immediate idea would be to define a utility method for contracting the deque as

```
private void contract()
```

Your `popFront` and `popBack` methods must call this method whenever the size of the payload block becomes too small compared to the overall capacity of the `data` array. The automated tester will enforce that unless that capacity of the deque is at its smallest possible value of 512 bits, the capacity of the deque **may not be more than four times the current number of elements** in that deque. Again, be mindful of edge cases when implementing the `contract` method. Symmetric to the previous `expand` method to be able to undo its effect, `contract` should allocate a new array half the size of the original `data`, copy the existing elements around the center of this smaller array, and discard the previous `data` array hoping that the bits and bytes that comprise it will hopefully be put in better uses. However, realize that intuition of when to trigger the contraction can now easily lead you astray, especially if you again vigorously hand wave something about symmetry!

The threshold for triggering a contraction operation absolutely positively **cannot** be that the `data` array is only half full, even when the automated tester does not enforce this. Following the general principle of hysteresis, this threshold must be that the array is **one quarter full**. Otherwise, a clever adversary could engineer a situation where immediately after an expansion has taken place, precisely two elements are popped out to trigger a contraction, which is then followed by two elements being pushed in to trigger an expansion. The malicious adversary could repeat such a back-and-forth cycle any number of times, and the push and pop operations would no longer be guaranteed to work in amortized constant time, making the surrounding system that relies on the efficient operation of this deque to slow down to a glacial crawl.

Lab 61: Save The Date

JUnit: CalendarQueueTest.java

Continuing on the general theme of queues from the previous problem, this problem looks at **priority queues** where the universe of elements that can potentially get pushed in the queue must be **comparable** with some total order that, for any given two elements, can unambiguously determine which one of these elements is smaller. This ordering determines a **priority** for these elements, so that the pop operation always brings out the element with the lowest priority among those that are currently in the queue.

The class `PriorityQueue<E>` in the Java standard library uses the classic **binary heap** to maintain the priority ordering behind the scenes. This class normally compares the elements by their built-in comparison `compareTo`, but it can also be given some `Comparator<E>` strategy object to perform the order comparison according to any desired comparison criteria to determine the priorities of these objects. However, same as how bees make honey and beavers build dams all by instinct, as a species humanity has come up with a multitude of data structures that implement a priority queue with efficient operations even for millions of elements. These data structures allow the basic push and pop operations to run significantly faster compared to keeping all elements in a messy unsorted array, and always looking up the smallest element with a linear loop every time the next element is popped out. (Some of these advance priority queue data structures allow additional operations such as **melding** two priority queues together into a single priority queue, or **updating the priority** of some element that is currently somewhere in the queue, again significantly faster compared to just keeping the elements in an unsorted array.)

This lab looks at a classic but now mostly forgotten data structure of calendar queue, another has-been from a simpler age that might still have some vinegar in its veins to save the day in situations that makes the more general structures grind down to a crawl. Consider a typical algorithmic application of any priority queue in an **event simulation**, where each **event** consists of a time when that event will take place in the future, plus some payload information associated with that event to describe how that event affects the simulation. The event simulator algorithm itself is a simple while-loop that repeatedly pops out the earliest event from the priority queue, and executes the effect of that event in the world being simulated. Each event can and will produce new events to take place in the future, and these events are added in the priority queue during the execution of that event that begat them. In this technique, the event simulation algorithm can always instantly jump to the next interesting event to happen in the future, instead of patiently waiting for something interesting to happen while advancing the simulation only one time unit at the time.

If the world being simulated follows the usual laws of **causality** in that events taking place at the present time cannot affect the past by creating new events that do take place in the past (thus potentially creating ill-defined situations in the simulation such as whatever would be the equivalent of the famous grandfather paradox in the world being simulated), the event popped from the head of the priority queue can only cause later events to be pushed into the queue. Furthermore, in most simulations, most events will cause new events to take place in the very near future. Under these two reasonable assumptions about how events beget other events to follow them, a calendar queue can perform the basic push and pop operations of a priority queue in expected constant-time

that is almost a cinch. The associated algorithms are also among the simplest to be found among the of the priority queue data structures linked above.

Analogous to an ordinary desk calendar, the calendar queue assumes that time proceeds in discrete **years** that have been divided into **days**, the smallest unit of granularity available for scheduling these events. To get started with this lab, add the instructor's class `CalendarEvent` in your labs project. **Do not modify the contents of this class in any way while completing this lab**. Instead, just read through this simple **data class** to see how its objects represent events that take place in a given year and day, with the payload information of each event given as a `message` string, and the `year` and `day` given as integers. Unlike us glorified apes in trousers who have trouble remembering anything symbolic better than a toaster could, the computer does not need the convenience division of years into months, but each year will consist of exactly `DAYS_IN_YEAR` days, with no leap years existing. Days are numbered starting from zero, and count up to `DAYS_IN_YEAR-1`, after which the year increases by one as the day rolls back to zero to begin a new year.

For the purposes of automated testing of this lab, this named constant `DAYS_IN_YEAR` has been defined to equal ten inside the `CalendarEvent` class. (We can pretend that this exercise takes place on one of those nanoplanets populated by various colourful characters in *The Little Prince*.) However, changing this named constant to 365, or any other positive integer for that matter depending on which planet we live on, should not change anything in the logic of your solution code. Never hardcode any **magic numbers** directly in your code, but always refer to such quantities using their named constant!

Internally, a calendar queue consists of an array that has a slot for each of the `DAYS_IN_YEAR` possible days that exist in our time keeping. From each such slot hangs a **singly-linked list** of events that fall on that particular day. The events of each list of all such same-day events are kept sorted by the year of that those events will take place. For example, if the queue contains three events that all take place on day 7, but in different years 2525, 3535 and 4545 as in the famous song, these three events are guaranteed to be stored inside the list so that the event of the year 2525 is first on in this list, the event of the year 3535 comes right after that, and the event of the highest year 4545 is the last. With all events distributed over these linked lists in this manner, finding the earliest event that takes place in any given particular day is quick and easy, regardless of the total number of events stored in the calendar queue. (Readers familiar with the internal operation of **chaining hash tables** may recognize the same basic technique that get productively applied in this different context.)

As for representing these chains, the Java Collection Framework offers the `LinkedList<E>` data structure for linked list operations. However, because of the utterly idiotic design decision of pre-generics Java to make arrays covariant that forces every Java array object to know its full element type at the runtime, the implementation of Java generics using **type erasure** that throws out all this runtime information after compilation will by definition make it impossible to use arrays whose element type is any generic class, even if the type of that generic class has been fully instantiated! However, this lab will be an exercise in implementing the basic linked list operations from scratch, as has historically usually been done in a second programming courses such as this one. We all have to eat a little bit of dirt before we die, even if we get to live on higher Platonic planes in our coding.

In your labs project, create a new class `CalendarQueue`. Inside this class, copy-paste the following nested utility class whose objects represent individual **nodes** of a **singly linked list**. Each such node contains the payload of the `event` that is stored in that node, and a reference `next` that, just like it says on the tin, points to the next node in the linked chain.

```
private static class CalendarNode {
    public CalendarEvent event;
    public CalendarNode next;
    public CalendarNode(CalendarEvent event, CalendarNode next) {
        this.event = event;
        this.next = next;
    }
}
```

In the last node object currently stored in a chain, the reference `next` equals `null`, to indicate in a controlled fashion the absence of a successor node object. To implement all the operations of a calendar queue, we only need the operations of a singly linked list, so these node objects don't need to waste memory for another reference `prev` that points to the predecessor node in that chain.

Note also that this nested class is not an **inner class**, seeing that it has been defined to be `static`. Unlike the non-static inner classes whose objects cannot exist on their own but must always exist in the context of some mothership of an outer class object (this allows the inner class objects to access all public or private members of this mothership object directly exactly as if these members were their own, without even any special syntax), objects from static nested classes don't need anything from the outer class and may therefore exist on their own, completely independent of whatever outer class objects have come into existence so far. The advantage of using static nested classes whenever feasible is using eight bytes of heap storage less per object, due to the absence of the hidden reference to the mothership object that must be stored separately inside every inner class object, as the inner class objects can have different motherships.

(Students who have been paying attention during this instructor's lectures might at this point object that this nested class violates the proper principles of **encapsulation** by defining its two members `event` and `next` to be `public`! Hold your horses; no violation of any noble engineering principles is actually taking place here, since this nested class has itself been defined `private` inside the outer class. No information about implementation details leaks out to the outside world any more than would leak from using local variables inside some method.)

After this nested class, define the following fields inside the class `CalendarQueue`:

```
private CalendarNode[] head;
private int size;
private int currentYear, currentDay;
```

The `head` array contains a total of `DAYS_IN_YEAR` elements, each element the reference to the first node of the linked list hanging from that slot. For days that have no events scheduled to happen

during that day, this slot is initially `null`. The field `size` contains the current number of events stored in this calendar queue, updated inside every `push` and `pop` for quick constant time lookup in the `getSize` method. The calendar queue object must also keep track of the current year and the current day of the simulation, here stored in two fields `currentYear` and `currentDay`. As the last piece of code that this specification will gift you before unleashing you to complete the rest yourself, here is the constructor of `CalendarQueue` to properly initialize these fields.

```
public CalendarQueue(int currentYear, int currentDay) {
    head = new CalendarNode[CalendarEvent.DAYS_IN_YEAR];
    this.currentYear = currentYear;
    this.currentDay = currentDay;
}
```

(The field `size` is already zero to begin with, and thus needs no initialization.)

The three `public` methods left for you to implement are

```
public int getSize()
public void push(CalendarEvent event)
public CalendarEvent pop()
```

The automated JUnit test for this problem guarantees that it will never `push` in any events that take place earlier that the current year and day of the calendar queue, and will never try to `pop` out an event from a queue that is empty. Nobody can be reasonably expected to achieve anything that is logically impossible, after all.

The method `push` should first access the day of its argument `event` to decide which slot the new `CalendarNode` object that contains that event as a payload will be linked into, and from that chain, find the position where this new node should be linked to. There are essentially three possible cases to get right. First, if the linked list hanging from that slot is empty, or if the event stored in the first node of the chain takes place later than the event currently being pushed in, the node that stores the new event becomes the first node in the chain. (Your code must correctly break the ties between two events with the same year and day by the message component. This happens automatically if you use the `compare` method of `CalendarEvent` to perform this comparison.) This is easily achieved by assigning the reference in the proper `head` slot to point to this new node, whose `next` reference in turn made to point to the node that was previously the first node hanging in that chain. That previous first node will therefore now become the second banana, properly positioned to become the successor node of the new node.

(Note how the same unconditional logic for assigning the `next` reference works for both cases. It is not okay to try to follow a `null` reference to get to an object, but merely assigning a `null` value to some reference is perfectly hunky dory!)

Otherwise, if the chain hanging from that day already contains at least one node, loop through the nodes of this chain by repeatedly following their `next` references to step to the next node in the

chain. Keep going until you find the **predecessor node** whose new successor the new node should be to maintain the sorted order of the events stored in that chain. Reassign the `next` reference of this found predecessor node to point to the new node, and assign the `next` reference of the new node to point to the earlier successor of that predecessor node. (Again, the same unconditional logic works even when the predecessor is the last node of the chain.)

The method `pop` should first look at the slot of the `currentDay`. If the year of the event stored in the first node of that chain equals `currentYear`, remove that node from the chain using a symmetric approach to the one used to insert a node, and return the event stored in that node as the answer. Otherwise, keep incrementing `currentDay`, rolling back to zero and incrementing the `currentYear` every time you go past the end of the array, until you eventually must come to a slot that satisfies the previous condition for its first node to allow you to terminate. Assuming a reasonable distribution of event times in the near future, this approach will usually return the next event faster than you could say boo to a goose, regardless of the number of events currently stored inside that priority queue.

To admire the simple elegance of the calendar queue approach, just realize that this search algorithm will only look at the first node of any slot! Even if a billion other future events that take place that same day in some far future year (perhaps something funny has already been destined to take place at The Restaurant at the End of the Universe) are hanging from every chain, these unseen nodes can not affect the speed of searching for the next event. More remarkably, these unseen nodes will also not affect the speed of updating the priority queue data structure after extracting that first event! This is something truly unusual among not just priority queues but all data structures that this author has ever seen. Typically, even if that first element can be found and extracted in constant time, the time needed to update the rest of the structure to reflect the new state is usually at least **logarithmic** with respect to the size of the data structure.

Lab 62: Worley Noise

For testing: WorleyNoiseMain.java

The advantage of using pseudorandom number generators instead of true randomness is that any algorithm that uses the same RNG initialized with the same seed value is guaranteed to produce the same results every time. Since the golden days of Elite, many computer games have utilized this approach by using procedural generation to build the levels of the game. Even confined to use 64 kilobytes of memory for literally everything in that game, such games can potentially pack a truly enormous game worlds into such tiny memory space. Only the level generation algorithm needs to be stored, and its RNG is each time seeded with, say, the level number or the coordinates of the level that is being generated for the player to enter. Of course, such levels can't remember any changes that the player caused in them, but are generated afresh each time the player returns to them.

In general, using raw randomness to decide what goes where in each level is usually not the best. for an aesthetically optimal outcome. Imagine playing a game of *Civilization* where the type of each tile has been randomly chosen among all possible types of tiles in that game! As amusing as playing in this kind of surrealist patchwork battleworld might be, even if only as an experiment, we normally want to place our thumb on the scale to produce structures whose complexity falls somewhere around the sweet spot in the continuum between the two end points of the rigorous uniform determinism that allows no variety, and the boiling random noise that allows no permanent meaning.

If the randomly generated structure is supposed to resemble something familiar to us that has an organic existence in our physical reality, such as an island, patterns on granite or other type of stone, or biological cells, finding the right balance between chaos and order might at first seem like an impossible task. Fortunately, many techniques to create procedural noise have been discovered, so that tweaking the parameters of the noise generation and the palette user to render the result can produce surprisingly natural looking patterns. Even better, the algorithm can produce the result image in any desired resolution, without any pixellation. In this lab we implement one famous procedural noise generation algorithm known as the Worley noise.

To understand how Worley noise works, we should first look at a classic computational geometry mainstay of a Voronoi diagram. For the given set of points *P* that can be arbitrary (but have typically been placed pseudorandomly on the image to be generated), the Voronoi diagram divides the image into disjoint convex polygons so that each polygon contains precisely the points that share the same nearest point in *P*. Once the Voronoi diagram for the given set of points has been constructed, many other important computational geometry properties can be conveniently computed from it. However, colouring all the pixels of the same polygon using the same flat uniform colour does not create a very aesthetically pleasing image, which is our goal and purpose in this lab.

Worley noise takes the same approach and runs with it by considering not just the point in *P* that is closest to the pixel being coloured, but potentially all points in *P*. The colour of each pixel is individually computed from the weighted sum of its sorted distances to all points in *P*, although in practice, most of these weights are set to zero. For example, the colour of a pixel might be

determined by its distances to its second, fifth and seventh nearest points in *P*, and the rest of the points do not matter for colourization of that particular pixel.

Since weighted distances can become arbitrarily large and we want to normalize the colours to be between zero and one, we first need some function to map any nonnegative real number distance into the real line interval [0,1], preferably in some efficiently computable but generously aesthetic fashion. Even at the slight risk of giving the readers painful flashbacks to their high school trigonometry classes, we recall that the trigonometric functions *sin* and *cos* not only oscillate between values -1 and 1, but are smoother than the proverbial baby's bottom in all their differentials. A slight adjustment to also consider the desired frequency gives us the utility function

```
private static double wave(double x, double a, double f) {
    return (1 + Math.sin(Math.pow(Math.abs(x), 0.95) / f)) / 2 * a;
}
```

where the parameter a is the **amplitude** and the parameter f is the **frequency** of the wave that maps the given real number x into the interval $[0, a]$. To make this transformation a bit less boring, the parameter x is first raised to some fractional power before the actual transformation, to make the frequency changes slow down as x increases. (After completing this lab, the student may wish to change the exponent from its present value of 0.95 to something larger or smaller to observe the effect that this exponent has to the shape of the noise.)

We next need to refresh our high school algebra for the formulas of computing the distance between two points (x_0, y_0) and (x_1, y_1) on the two-dimensional plane. The **Euclidean distance** is derived from the Pythagorean theorem, but **Manhattan distance** and **chessboard distance** could just as well be used as alternative (but mathematically perfectly sound) distance metrics, from the set of infinitely many other functions that could serve in such role.

```
private static double euclidean(double x0, double y0, double x1,
double y1) {
    return Math.sqrt((x1-x0)*(x1-x0) + (y1-y0)*(y1-y0));
}

private static double manhattan(double x0, double y0, double x1,
double y1) {
    return Math.abs(x1-x0) + Math.abs(y1-y0);
}

private static double chessboard(double x0, double y0, double x1,
double y1) {
    return Math.max(Math.abs(x1-x0), Math.abs(y1-y0));
}
```

These preliminaries out of the way, we now have everything that is needed to generate some Worley noise. In your project folder, create a new class `WorleyNoise`, and in there the following method

that probably takes the largest number of parameters among all the methods seen somewhere in this specification.

```
public static void compute(double[][] out, double[] xs, double[] ys,
double[] weight, double a, double f, String distance) {
```

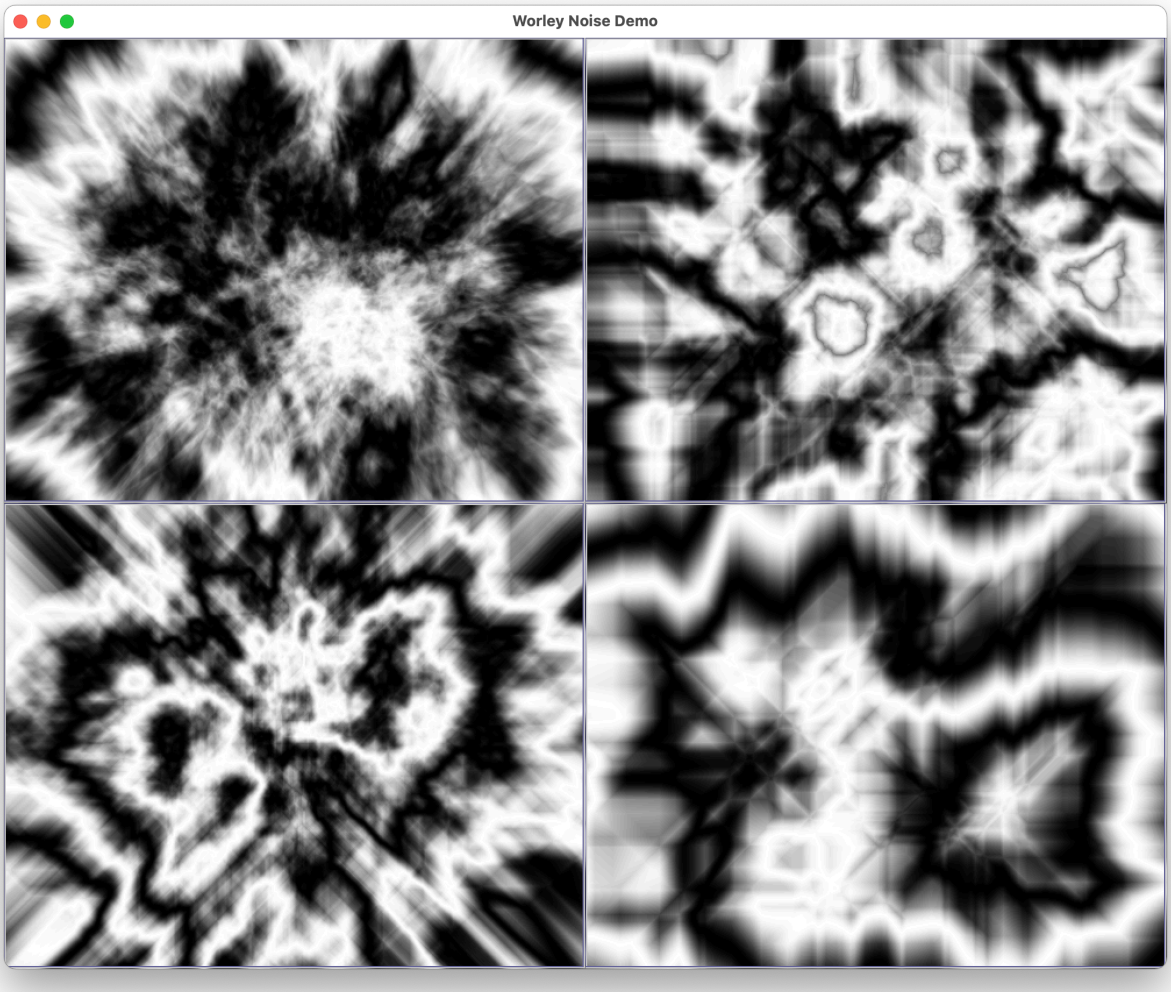
Note again that for efficiency, this method does not create a new array to be returned as the answer each time, but fills in the computed result to the given array object argument `out`. The raw numerical values computed into this `out` array will later be turned into a `BufferedImage` that has the exact same dimensions as this array. The caller is responsible for ensuring that the dimensions of the array object passed as the `out` argument are large enough to contain the result. As the fundamental principles of the contract law dictate, your method can fail any which way it wants, should the caller not hold on to their end of the contract. This is not the time to be defensive.

The arrays `xs` and `ys` contain the *x*- and *y*-coordinates of the points *P* from which the value of each element of `out` is calculated. Again, the caller is responsible for ensuring that both arrays have enough elements to list all the points in *P*. Your method should loop through all the positions of the `out` array and fill in the value of each `out[y][x]` that corresponds to the point (*x*, *y*) on the plane in the following manner. (The swap of coordinate roles is done here because Java arrays are stored in row-major order.) First, calculate the array of distances from the point (*x*, *y*) to all points of *P*. The value of the `distance` parameter determines which distance metric you should use for this calculation, for the three possible **option constant** values of `"euclidean"`, `"manhattan"` and `"chessboard"` that the automated tester will ever give it.

Next, sort this array of distances in ascending order, without caring which distance corresponds to which point in *P*, since we need only these distances themselves to compute the value of the element `out[y][x]` currently being filled in. Then, calculate the weighted sum of these sorted distances so that the *i*:th distance in this sorted order is weighted by `weight[i]`. Finally, transform that weighted sum of distances into a value in the interval [0, *a*] using the previous `wave` function with the two remaining parameters `a` and `f` finally coming in to play their part at this point.

Note that the `weight` array can be shorter than the number of points in *P* given in the arrays `xs` and `ys`, in which case the weights past the end of the `weight` array should simply be considered to be zeros. For example, should the `weight` array happen to equal `{1, 0, -0.5, 0.5}`, this would mean that the distance of the closest point is given the weight 1, the distance to the third closest point is given the negative weight of -0.5, and the distance to the fourth closest point is given weight 0.5. Other points in this ranked order are ignored in this calculation.

This is it for this lab: the `main` method of the WorleyNoiseMain.java test class takes care of converting these noise values into grayscale pixel values to be proudly displayed as images inside a new `JFrame` instance. When run, your code should produce the result visually identical to the screenshot below. The top left image uses Euclidean distance applied to hundred points and fifty randomly chosen weights for a chaotic explosive result. The top right image uses Manhattan distance and fewer points. The bottom left image uses chessboard distance with the same number of points, and the bottom right image goes back to Euclidean distance but with only twenty points.



In practice, the points of *P* are not thrown on the image the way that a truly blind man throws darts, but the image is first subdivided into some smaller grid, and each cell of this grid will receive the same number of points, typically just one point per cell. Such a horrendous zero-variance deviation from the way that true randomness would be, with probability almost close to one, guaranteed to create clusters and clumps somewhere, ensures that the pixels near the edges of the image don't all essentially behave the same way as the pixels in their local neighbourhood, but the noise remains pleasantly detailed and varied even near the edges. (As discussed in the earlier Linus Sequence problem, the easiest way to detect most of human-generated fake randomness is in its notable lack of such clustering. If the image were divided five-by-five into 25 grid cells, many people would automatically place one point in each cell when asked to simulate the results of throwing darts randomly, and imagine themselves especially devious in doing so!)

To create even more natural results, multiple images generated with either Worley noise or other suitable types of noise, for example Perlin noise, can be combined in layers to produce **fractal noise** to further smooth out the result to make it seem more natural without merely just making it blurry

by losing all the interesting little detail. To achieve this outcome, these layers must be computed with different amplitudes and frequencies so that the higher the amplitude, the lower the frequency and the number of points in P used to generate that particular layer. The high-amplitude layer therefore defines the main features of the landscape, on which the lower-amplitude layers then build on detail of smaller amplitude but higher frequency without distorting the general main features too much.

If the amplitude and frequency of each layer are chosen accurately, zooming in this landscape brings forth the fractal nature of the result in that the result looks similar in every scale, the way that all fractals are by convention expected to do. Of course, same as in the real world, zooming in deep enough will eventually reach the final layer, after which the microstructure will be very different from the way that the whole thing looks like at the macroscopic level. As above, so below; repeated subdividing the coastline of England can make it to be as long as you want it to be, but eventually you will reach the subatomic level and the whole approach breaks down.

Up here at the practical level of things, suitable choice of the noise parameters and the colour palette can make this fractal noise visually indistinguishable from actual photos of a real mountain landscape or some other organic entity simulated with this noise. However, the parameter space has enough degrees of freedom to be able to simulate not just alien landscapes, but biological textures and structures.

Lab 63: Recursive Subdivision Of Implicit Shapes

For testing: ImplicitShapeRendererMain.java

The earlier lab about Lissajous curves demonstrated the important technique to render one-dimensional **parametric curves** on the plane. A parametric curve is defined by two functions $x(t)$ and $y(t)$ of one variable t so that the curve consists of all points $(x(t), y(t))$ produced by taking the parameter t for a walk through some range of values, typically from 0 to 1. To render such a continuous curve into a discrete pixel raster, the curve is subdivided into smaller pieces to be rendered in a piecemeal fashion typically as quadratic or cubic segments. Each piece should smoothly connect with the previous and next piece around it without any visible discontinuities in either the curve or its tangents. Viewed from a distance, the pixel representation of that continuous curve will be perceived as actually being that continuous and smooth curve, of course within the limitations of the granularity of the pixel raster itself.

In this same spirit, implicit functions define two-dimensional **implicit shapes** on the plane. One function f receives as its arguments both the x- and y-coordinates of an arbitrary point (x, y), and returns a numerical result $f(x, y)$ whose sign indicates whether that point lies inside the shape (negative), on the shape boundary (zero), or outside the shape (positive). For example, the implicit function of the unit disk centered at the origin is $x^2 + y^2 - 1$, which the reader can surely verify is negative for all points (x, y) inside the unit disk, zero for all points on the circle wrapped around that disk, and positive for all points outside the disk. (The implicit formula for the general case of disk centered at the point (x_c, y_c) with the radius of r would be $(x - x_c)^2 + (y - y_c)^2 - r^2$, of course.)

To render a continuous two-dimensional shape into a discrete pixel raster, some subdivision scheme is again used to sample individual points on the plane, and the information gleamed from those points is used to render something reasonable to depict the infinite continuum of points that could not get get sampled. The technique of marching squares subdivides the plane into a regular grid of boxes of uniform size. The contents of each box are filled with triangular shapes of foreground colours depending on which of the four corner points of that box lie inside and which lie outside the implicit shape. Ambiguous ties that occur around **saddle points** are resolved by letting the center point of the box decide which direction is more dominant.

At most five points sampled from the infinite continuum inside the box hopefully reveal the entire continuum with reasonable accuracy. Same as with any numerical method based on finite sampling, the implicit function given as a black box must be assumed to be sufficiently well-behaved so that the global and local structure of the shape can be reasonably estimated from such point samples. Of course, a crafty adversary who knows our sampling scheme could always design this black-box to be a fractal made of holes, islands and tendrils to make the result of any such sampling scheme to be arbitrarily far off from reality. No algorithm based on finite sampling of a black box can possibly hope to fight back against this... at least unless we resort to randomness to prevent the adversary from predicting which points we are going to be sampling!

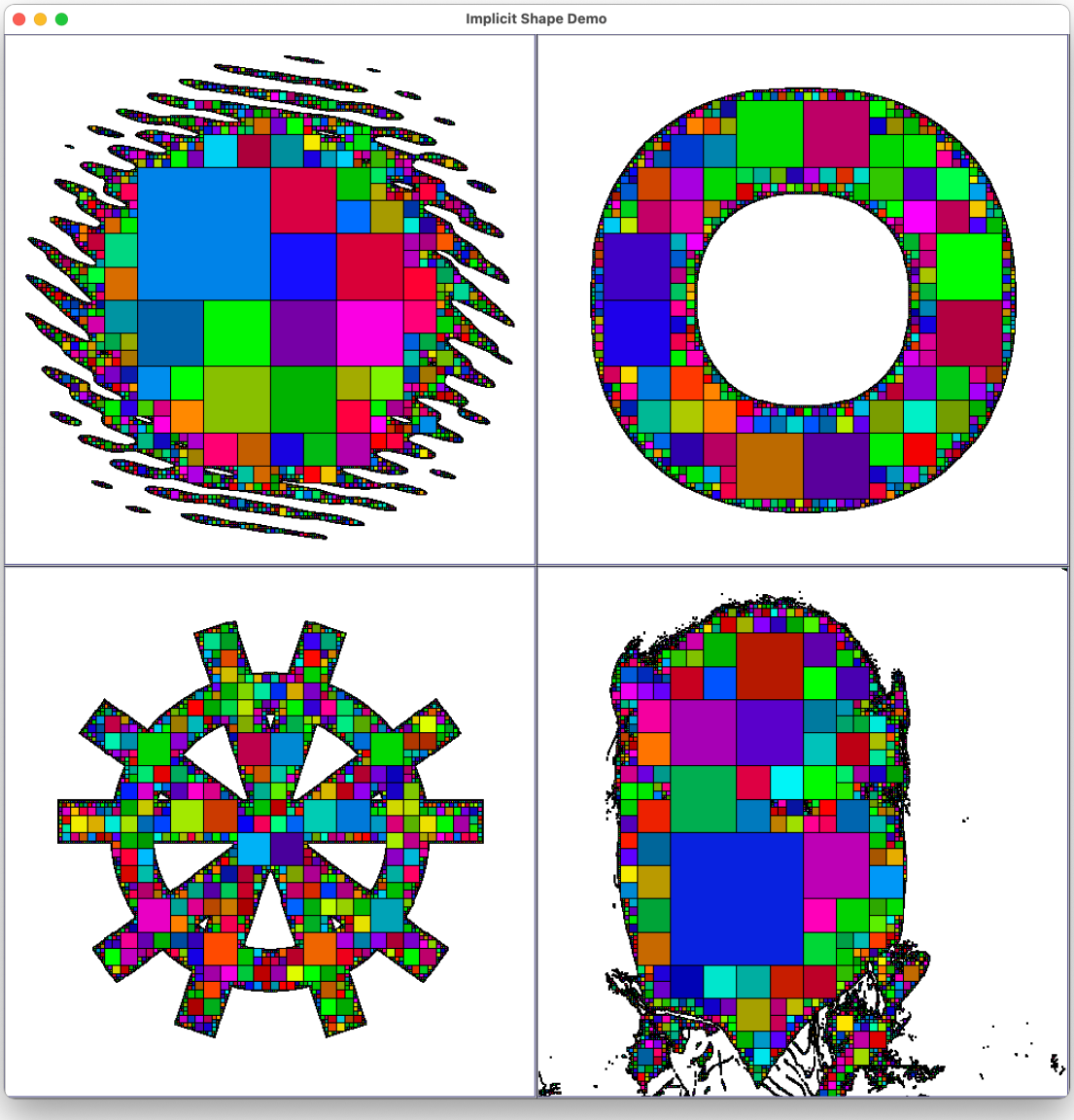
This lab concerns another rendering technique that also subdivides the implicit shape into rectangles. However, unlike for the marching squares, our subdivision will be performed recursively so that the depth of this subdivision is not uniform across the plane, but automagically becomes

more fine-tuned near the edges of the shape where such precision is needed. In your project folder, create a new class named `ImplicitShapeRenderer`, and there the recursive method

```
public static int render(DoubleBinaryOperator f, double xStart, double xEnd, double yStart, double yEnd, Graphics2D g)
```

The first parameter is the implicit function *f* that determines the shape to be rendered with recursive subdivision. Since functions are not **first-class objects** in Java so that they could be passed back and forth in method calls, the function is wrapped inside the strategy object `DoubleBinaryOperator` defined in the package `java.util.function`, and this object is then passed as an argument to this method.

The next four parameters define the rectangular box whose contents being rendered in this method call into the `Graphics2D` object `g` given as the last parameter. This box is an axis-aligned rectangle that covers all points whose *x*-coordinate lies anywhere between the values `xStart` and `xEnd`, inclusive, and whose y-coordinate lies anywhere between the values `yStart` and `yEnd`. The width and the height of this box are given by formulas `xEnd-xStart` and `yEnd-yStart`.



The caller is assumed to have already set up the coordinate system along with the shape of the pen in the `Graphics2D` object on which this method is supposed to render the image, so this method doesn't need to worry about setting up those. However, the method needs to choose random colour randomly for each box that it renders to create the effect seen above in style of the psychedelic sixties. To create a random colour, it is bad form to just choose a random red, green and blue component for the RGB colour, as the palette created this way will usually end up looking like something that we would expect to see hanging from the rack inside a Goodwill store. A better technique, as illustrated inside the example program `Sliders`, first chooses a random **hue** from the visible spectrum, and then adjusts it with the random **saturation** and **brightness** so that the latter two components are kept in reasonable limits. Fortunately, the `Color` class in the AWT package knows how to perform the necessary conversions between the RGB and HSB colour spaces.

As you can see from the above screenshot, we aim to subdivide the implicit shape by making each box as large a rectangle as we can, and have the subdivision recurse deeper to produce smaller boxes only near the edges of the shape. This approach very much resembles the idea that we have already seen in the earlier lab where a binary grid of pure black and white cells was subdivided into a **region quadtree** whose branches were on different depths depending on the uniformity of the shape represented by that branch.

At the top level call of each of the four example images in the screenshot, the box has dimensions of 500 by 500 pixels, with the origin set at the top left corner. The shape depicted on top left is a circle where the x- and y-coordinates have been perturbed with some trigonometry before passing them to the circle formula. This example illustrates one big advantage of implicit shapes in that they can be easily **geometrically transformed** by *any* invertible transformation (not merely by the affine transformations of **translation**, **rotation** and **scaling**, for when the shape must consist of a **triangle mesh**) by first applying the inverse transformation to the point (x, y) whose membership inside the implicit shape is to be determined.

The shape depicted in the top right image is an **annulus** created by subtracting a small superellipse from inside a larger one, both superellipses using the exponent of 2.3. This example showcases another nifty advantage of implicit shapes in that their **constructive geometry** operations of **union**, **intersection** and **difference** (either symmetric or not) become trivial by combining the results of two implicit functions by appropriate logical operators. The resulting shape will appear seamless when viewed by the subdivision algorithm that treats the combined shape as a black box.

The bottom left shape illustrates the use of the `Area` decorator that can be applied to arbitrary instances of `Shape` to give them super powers of not just constructive geometry, but arbitrary affine transformations. The function to determine whether the point (x, y) lies inside the shape then translates into a one-liner that asks the `Area` object the very same thing, and converting the received `boolean` answer into a signed `int`. The bottom right shape uses an implicit function that determines the membership of point (x, y) in the area from the colour of that pixel in the underlying image, here again the example image ilkka.jpg, this colourful subdivision transforming your instructor's ugly mug into something more enjoyable.

To achieve this fabulous effect, the integer return value of the previous method should always be one of the three possibilities -1, 0 or +1. The return value of -1 means that the entire box lies inside the implicit shape without any holes anywhere inside it, at least to the extent that our finite sampling of the infinite continuum would be able to detect. (The Adversary is assumed to be a gentleman and play nicely along with this assumption.) Symmetrically, the return value of +1 means that the entire box lies outside the implicit shape. The return value of 0 conveys the message that some parts of the box are inside the implicit shape and some are outside it.

The base cases of this recursion are when either the width or the height of the box is less than 1.0, the size of a single pixel. In this base case, this method should pass the coordinates of the center point inside this box to the function `f`, and return -1 if that center point lies inside the shape, and +1 if it does not. (In this finite sampling of the continuum, we treat the vanishingly small subset of points that lie precisely on the edge of the shape as if they were inside the shape.) Note that nothing gets rendered on `g` at these base cases, not even if the center point is inside the shape! After all, we cannot know whether this small base case box will eventually become a part of some larger box whose points also all lie inside the implicit shape.

This method should subdivide boxes that are too big to qualify as a base case into four equal-sized quadrants, and recursively call itself for each of these subdivided quadrants. The four answers received from the NW, NE, SE and SW quadrants determine whether the method will perform any rendering at this level of recursion, and what answer it will return to the caller at the previous level. If all four quadrants returned the same answer that is either -1 or +1, this method should also return that same answer without rendering anything. This is exactly what happened in the base case, since the entire box lies fully inside (-1) or fully outside (+1) the shape.

However, if any of the four recursive calls returns a zero, or if any two of the four quadrants return a different answer, this method should itself return zero, and before that render a filled rectangle over precisely those quadrants whose recursive call returned the answer -1. First `fill` the rectangle with a randomly chosen colour as explained earlier, then `draw` a black outline around that rectangle. Pay attention to calculating the arguments given to the constructor of the `Rectangle2D.Double` object, seeing that this constructor expects to be given the coordinates of the top left corner along with the width and the height of the rectangle, whereas the parameters of the method that you are writing here essentially define the coordinates of the top left and the bottom right corners of the box. Some care is needed in calculating the arguments for the rectangle constructor from the coordinates of the box corners.

Nothing is rendered for the quadrants whose recursive call returned the answer +1, since those quadrants told us that their contents lie fully outside the shape currently being rendered. Most importantly, nothing is also rendered for the quadrants whose recursive calls returned the answer zero, since that answer indicates that that recursive call already properly rendered the contents of such quadrants! Anything that we could possibly render on top of the work done by that recursive call could at best only be superfluous, and at worst mess up the perfectly good rendering painstakingly achieved by those recursive calls.

Lab 64: Learning The Ropes I: Composition

JUnit: RopeTestOne.java

The `String` data type in Java is both powerful and useful, especially due to its guarantee of **immutability**. Since the `String` type is basically a facelift over an ordinary character array stored privately inside the `String` object, accessing the individual characters with the `charAt` method based on their position happens in guaranteed constant time. However, the operations of string **concatenation** and **substring slicing** will always take linear time with respect to the length of the strings, since both operations require creating an entirely new `String` object to represent the result. The rigidity of the underlying character array necessitates the creation of an entirely new character array to store the individual characters of the result.

However, raw character strings are not the only game in town to represent immutable sequences of characters that we would like to concatenate and slice pieces out of. This lab and its followup introduce a more flexible representation of character sequences with a data structure known as a rope. Just like the ordinary `String` type, each instance of `Rope` represents a linear sequence of Unicode characters. For example, we could start with two ropes that represent the character sequences `"hello"` and `"world"`. The truly astonishing property of the rope data structure is that any two instances of `Rope` can be concatenated into a new rope `"helloworld"` so that the time and memory use of this operation is guaranteed to be constant, regardless of the length of the two ropes being concatenated! Even more amazingly, the exact same guarantee of constant time and memory use holds for the operation of slicing out a subrope from any existing rope.

Since there is no free lunch, obviously there has to be a hidden catch somewhere; otherwise, why should the ordinary `String` data type even exist, if ropes could just waltz in any time to sing how anything strings can do, they can do better? The disadvantage of ropes is that accessing the individual characters by their position with the `charAt` method can no longer take place in a guaranteed constant time, but the time required will be **logarithmic** with respect to the `length` of the rope. Even that assumes an industrial strength implementation of this data structure that can do proper **rebalancing** operations behind the scenes whenever the entire structure starts getting too lopsided. In this lab, we implement a slightly simplified version of ropes that do not perform any complicated rebalancing, since the student will surely agree that even the toy version presented here is already amazing not just in its algorithmic elegance, but as an example of good object oriented design that we are supposed to be learning in this very course!

Unlike a string object that represents the text `"helloworld"`, a rope structure is not necessarily a single object, but a tree structure made of several rope **nodes**. All these nodes are instances of subtypes of the abstract superclass `Rope` that defines the public functionality common to all ropes. Start by adding the following implementation of that superclass in your labs project.

```
abstract public class Rope {
    abstract public char charAt(int index);
    private int len = -1;
    public final int length() {
```

```
        if(len == -1) {
            // Call the template method to initialize len.
            len = computeLength();
        }
        // Either way, we can now return the cached value.
        return len;
    }

    // Subclasses must override the actual length computation.
    abstract protected int computeLength();
}
```

Each concrete subclass of `Rope` must then implement the public method `charAt` to return the character at the given position `index`, doing this in some appropriate fashion that depends on the way that the necessary data is stored inside that subclass. If the given `index` is out of bounds for the sequence of Unicode characters represented by the rope node, this method must throw an `IndexOutOfBoundsException` instead of returning anything. Note that since this exception type is **unchecked**, it does not show in the signature of the `charAt` method.

Each instance of any subtype of `Rope` will also be able to tell you its `length`. However, since all instances of `Rope` are designed to be **immutable**, their length cannot possibly change after object creation, so it would be redundant to compute this same length over and over again each time the user code requests to know it. Therefore, the method `length` has been implemented as a `final` method in the `Rope` superclass to perform **caching** of the computed length into the `private` data field `len`, in a suave application of **trading space for time**. Since no rope can possibly have a negative length, this field is initially `-1` to indicate that its true value has not been computed yet.

The first time someone requests to know the length of this rope, the method `length` calls the `protected` **template method** named `computeLength` that each concrete subclass must implement to correctly return the length of that particular rope instance. This template method design pattern allows both the `Rope` superclass and all the user code to be blissfully ignorant of the gory details of how the sausage of computing the length of each instance is actually made in the smoke-filled rooms hidden behind the encapsulation interface.

The first concrete subclass of `Rope` represents a simple rope that has been constructed from the given instance of `String`. Create the class `StringRope` in your lab project, with the signature

```
public class StringRope extends Rope
```

This class should have a public constructor that receives a `String` parameter that it simply stores into some `private` data field. Both methods `charAt` and `length` can then simply call these very same methods of that privately stored string, and return whatever those calls returned.

Same as the other subclasses that we extend from the abstract superclass `Rope` in this lab, the subclass `StringRope` must also override the method `toString` that allows converting the contents of that rope instance back into a Java string used by the rest of the user code program. Inside this class, this method can simply return that privately stored string.

Instances of the second concrete subclass written in this lab represent ropes that have been constructed by concatenating two existing ropes. In your lab project, create the class

```
public class ConcatRope extends Rope
```

whose polymorphic constructor receives two parameters `left` and `right`, both instances of the type `Rope`. Same as what `StringRope` did for the parameter `String` that it was given, the constructor of `ConcatRope` should store these parameters into private data fields with the same names.

The method `computeLength` should call the method `length` of its `left` and `right` pieces and return the sum of these lengths. (Note that you need to specifically call the method `length`, not the method `computeLength`, to enjoy the benefits of caching the length for these pieces!) The method `charAt` should decide which piece it recursively consults for the character in the given position `index`, based on whether that `index` falls in the `left` piece or in the `right` piece. This decision is easily made by comparing the `index` to the `length()` of the `left` piece. Continue from the left piece if the `index` falls in there. Otherwise continue from the right piece, but don't forget to first subtract the length of the `left` piece from the `index` that you pass on!

Instances of the third concrete subclass in this lab represent ropes that have been constructed by slicing off a subrope from an existing rope. In your lab project, create the class

```
public class SubRope extends Rope
```

The constructor of this class should expect three parameters, the `original` rope being sliced, and the `start` and `end` indices that define the `start` and `end` position of the subrope inside the original. To be consistent with the behaviour of the `substring` method in Java `String` class that this operation mimics, the `start` index is defined to be inclusive, but the `end` index is guaranteed to be exclusive. Also, again in spirit of the `substring` method in `String`, the constructor of this method must immediately throw an `IndexOutOfBoundsException` if the `start` index is negative or greater than the `end` index (these indices can be equal, to denote slicing off the empty substring), or where the `end` index is greater than the length of the `original` rope. (The corner case where both `start` and `end` equal the `length` of the string returns an empty rope.)

The method `charAt` can now consult the same method of the `original` rope, but must remember to shift the `index` to the right by adding the `start` value to it. The `length` method simply returns the value of `end-start`. Without access to the actual type of the original (which the method should not care one whit about anyway, following to the bumper sticker version of the **Liskov**

substitution principle), the `toString` method has no choice but create a new `StringBuilder` and append in there all its individual characters extracted with the `charAt` method.

The JUnit test class RopeTestOne.java uses these three subtypes to build up and slice out ropes of various lengths and contents, including one giant instance of rope that consists of over one billion characters, far too many to be represented as an ordinary `String`.

One major criticism of this design still lingers in that the user code must be aware of all of the subtypes `StringRope`, `ConcatRope` and `SubRope`, since the user code operations on ropes involve explicit creation of instances of these subtypes with the operator new. However, this is easily remedied by defining appropriate methods in the superclass `Rope` to achieve these desired effects. For example, the operation of concatenating two existing ropes could be defined as a one-liner method

```
public final Rope concatenate(Rope other) {
   return new ConcatRope(this, other);
}
```

to allow the user code then to just say

```
Rope result = r1.concatenate(r2);
```

to concatenate their two existing `Rope` instances `r1` and `r2`, without knowing or caring about the name of the subclass that represents this concatenation. Such a method could then potentially also internally involve other optimizations. For example, the subrope extraction utility method could always return a reference to one and the same `Rope.EMPTY` object, instead of always redundantly creating a new object when extracting an empty subrope with `start == end`.

Since the rope data structure is designed to be immutable, the operation of deleting a subrope from an existing rope would not actually modify that existing rope, but again create a new instance of `Rope` to represent the result. We don't even need to define a new subclass of `Rope` to represent instances created in this fashion, since even this operation can be made to work in guaranteed constant time and memory once we realize that it is merely **syntactic sugar** for concatenating two subropes tactically sliced out of the original rope, implemented in `Rope` as something like

```
public final Rope delete(int start, int end) {
    Rope left = new SubRope(this, 0, start);
    Rope right = new SubRope(this, end, original.length());
    return new ConcatRope(left, right);
}
```

Lab 65: Learning The Ropes II: Comparisons

JUnit: RopeTestTwo.java

After completing the basic rope operations for character sequence in the previous lab, time has come to implement the core methods that are necessary and sufficient for the `Rope` data type to stand proud and tall as a first class citizen among all the other classes in the Java language. That is, the method `equals` for equality comparisons, the method `hashCode` for calculating an integer index needed to store arbitrary ropes inside a `HashSet<Rope>` data structure for fast lookup and retrieval, and the method `compareTo` to order compare two `Rope` objects in lexicographic order.

We shall start with hash codes, since they can later be used to speed up the equality comparisons. Since the instances of `Rope` are designed to be immutable, their hash codes can never change. We can therefore cache the hash codes the exact same way as was done with the lengths. Inside the abstract superclass `Rope` that was given to you in the last lab, define a new data field

```
private int hash = -1;
```

that will cache the computed value of the hash code the first time that someone requests to see it, available in constant time for all future hash code queries after that. The special value `-1` again denotes that the actual hash code has not yet been computed. Then, write the method

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

that computes and caches the hash code whenever `hash == -1`, and returns the value of the cached `hash`. The simplest way to turn a character sequence into a reasonably chaotic hash code so that every character will affect the computed hash code is to use a for-loop to iterate through the characters of the sequence from beginning to end. For each character `c` encountered along the way, update the `hash` with the assignment

```
hash = 33 * hash + (int)c;
```

After the loop has terminated, return the accumulated hash, although you also need to ensure that in the rare case where the `hash` ends up being equal to the special value `-1`, the method returns some other fixed value instead. (Just pick any value, for example 17 or 42, since any fixed value is as good as any other value.) Alternatively, if you dislike negative numbers and want the hash code always be nonnegative, if only to make the job a bit easier for those guys who thanklessly implement the data structure to use these hash codes, the fast and correct way to do so is to use **bitwise arithmetic** to chop off all but the lowest 31 bits of the signed integer with the expression

```
hash = hash & 0x07FFFFFF;
```

After implementing `hashCode`, the next job in our TODO list is to allow lexicographic ordering comparisons between arbitrary ropes. Modify the signature line of the entire class to say

```
public class Rope implements Comparable<Rope>
```

to indicate that the order comparison is a meaningful operation between any two instances of this type. Implementing the interface `Comparable<E>` then requires you to implement one method

```
@Override public int compareTo(Rope other)
```

whose return value should be `-1` if `this` rope is lexicographically smaller than the `other` rope, `+1` if this rope is lexicographically greater than the other rope, and `0` when both ropes are equal. To perform this order comparison, loop through the positions of `this` and `other` rope in lockstep from beginning to end, comparing the two corresponding characters in each position. As soon as those two characters are in any way different, or if the loop runs past the end of either rope, you have your answer and don't have to even glance at the rest of the characters in either rope. Only if you run past the end of both ropes simultaneously, you may conclude that both ropes are equal, so the method will return `0`.

We leave the equality comparison method to be the last one implemented, since its implementation greatly benefits from the existence of the `hashCode` and `compareTo` methods. Again, remember that the method

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

must accept any `Object` whatsoever as its argument, since in the Java type system, equality comparison is a perfectly meaningful concept between any two objects whatsoever. (Even if the answer to a question happens to be some variation of "No way, José", that fact by itself doesn't render the question itself meaningless or malformed!) After using `instanceof` to verify that the other instance is also some kind of a `Rope`, create a new properly typed reference to the argument object with a **downcast**, to allow you to access the `Rope` functionality of that `other` object.

Since the inequality of the hash codes entails the inequality of the objects themselves, this method should start by first checking if both `this` and `other` ropes have their cached `hash` values set to something other than `-1`. If that is indeed the case, and those cached hash values are different, you can smoothly return `false` without further ado. Otherwise, call the previous `compareTo` method to find out how `this` and `other` rope relate to each other in lexicographic order, and return `true` if and only if these ropes are equal in that sense, as this can only happen when the ropes themselves represent equal sequences of characters. It is theoretically possible for two ropes to have the same hash code despite representing distinct character sequences. As unlikely as this possibility may be, your code still needs to be prepared for that possibility.

Lab 66: Slater-Vélez Sequence

JUnit: SlaterVelezTest.java

The lab problem 17, "Diamond Sequence", as well as the two sequences implemented in the problem 0(L), "I Can, Therefore I Must", were based on the idea of always continuing the given positive integer sequence greedily with the smallest unused positive integer that did not create a violation of the sequence-defining rule with the values in the sequence generated so far. This lab continues the same theme with generating the elements of the remarkable Slater-Vélez sequence, this author becoming aware of this sequence from the Code Golf problem that the link leads to.

Each element of the the Slater-Vélez sequence is the smallest positive integer *n* that has not appeared so far in the sequence, under the constraint that the absolute difference of *n* and its immediate predecessor in the previous position of the sequence also has not previously appeared as the absolute difference of any two consecutive elements in the sequence. This definition makes the sequence unique, and the reader can verify that the Slater-Vélez sequence starts as

1, 2, 4, 7, 3, 8, 14, 5, 12, 20, 6, 16, 27, 9, 21, 34, 10, 25, ...

with the corresponding sequence of absolute differences between consecutive elements being

1, 2, 3, 4, 5, 6, 9, 7, 8, 14, 10, 11, 18, 12, 13, 24, 15, ...

It follows trivially from the rule used in the construction of this sequence that no element can ever repeat. However, the same way as is guaranteed to happen in the diamond sequence, this sequence doesn't leave any holes behind in its journey towards the omega, but will eventually visit every positive integer, as proven by those two jolly *caballeros* Slater and Vélez back in 1979. This sequence is therefore an infinite permutation of the positive integers, although its cycles are not quite as tight as those in the diamond sequence where every cycle had the exact same length of two.

As was done in the diamond sequence lab, we implement this sequence as a **virtual iterator** that computes the next element of the sequence lazily on command. Unlike Python, Java doesn't have lazy sequences and generators built in the language itself. The iterator framework then has to serve as the next best way to achieve the bliss of enlightened laziness. In your lab project, create the class

```
public class SlaterVelez implements Iterator<Long>
```

Since the sequence is infinite, the method `hasNext` should always unconditionally return `true`. We will tacitly assume that the sequence won't be generated far enough for our values to get their heads chopped off by spilling outside the Procrustean bed of 64-bit `long` integers.

Since the sequence generator needs to keep track of the numbers and their absolute differences that it has already seen, we can again follow the lead of the diamond sequence problem and use two instances of `NatSet` utility class to achieve this. Inside your class, define the data fields

```
private NatSet seen = new NatSet();
```

```
private NatSet diffs = new NatSet();
private long previous = 0;
```

The two instances of the `NatSet` utility class, named above `seen` and `diffs`, keep an eye on the positive integers and their absolute differences that have already been seen in the sequence generated so far. Furthermore, the class needs to remember the `previous` number in the sequence, which we might as well initially set to zero to indicate that nothing has been generated. To complete the obligations of implementing the interface `Iterator<Long>`, write the method

```
public Long next()
```

to virtually extend the sequence by one more element that becomes the new `previous` number and gets returned from this method. This method should also update the `seen` and `diffs` sets to correspond to the new sequence that is now one element longer than it was before this method call.

Since the `NatSet` instances keep track of natural numbers that include a zero, unlike the positive integers that start from one, make sure to add the zero as a member to both `seen` and `diffs` sets in the constructor of this class, before you start generating the actual values. Furthermore, to speed up the generation of the `next` element of the sequence, you should note that the `NatSet` utility class comes with the convenience method

```
public long allTrueUpTo()
```

that returns the smallest natural number that is not a member of this set. By getting this invaluable information about the numbers that you have already `seen` as part of this sequence, you know right away where to start looking for the next number in the sequence, instead of always starting to look for the next element from counting up from one. By cleverly using also the information about the `previous` value and the smallest difference that is not yet a member of the `diffs` set, you can sometimes jump even higher for your starting point to look for the next element, without any risk of jumping over the smallest positive integer that would have satisfied the rule of this sequence. With these optimizations, your code should be able to produce the first half a million elements of the Slater-Vélez sequence in a couple of seconds.

Lab 67: Lemire Dynamic Frequency Sampling

JUnit: LemireFrequencySamplingTest.java

To sample a random element fairly from the given array `arr` so that every element has an equal chance of being chosen, the expression `rng(nextInt(arr.length))` can be used to get a random position to go get the actual element, where `rng` is your random number generator. However, sometimes we don't want all elements to get chosen with the same uniform probability, but we want these random choices to be performed according to the given **frequency array** of the exact same length as the array `arr`.

Each element of the frequency array gives the probabilistic **frequency** for the corresponding original element to be chosen in the long run. For example, if the original array `arr` has five elements, the frequency array `{3, 4, 0, 1, 4}` would indicate that the element in the position 0 would get sampled roughly three times as often as the element in the position 3, and roughly 3/4 as often as the element in the position 1 or in the position 4. As you can see, some frequencies can also be zero, so that those elements should never be sampled.

To sample the next element from the given frequencies, each element must have the probability *f* / *s* to get sampled, where *f* is the frequency of that particular element and *s* is the sum of all frequencies in the frequency table. For example, for the above example frequencies, the probability for the element in the position zero to get sampled would be 3 / (3 + 4 + 0 + 1 + 4) = 3/12 = 1/4.

Note that the choice of each random element must be done independently of the history of the past selections, with no attempt to "correct" the frequency of some element that so far has not appeared as much as its desired frequency would predict. In the short run, all random walks exhibit variance and clustering, so that the frequencies of the sampled elements will basically never be exactly equal to what the frequency table says that they would ideally be. In the long run, the random sampling algorithm should see the proportion of each sampled element converge to its given frequency divided *s*, in line with the frequentist interpretation of probabilities.

The simple algorithm that samples a random element according to the given frequencies starts by creating a random number *r* between 0 and *s*-1. It then adds up element frequencies from the beginning until *r* becomes strictly smaller than the sum of the frequencies up to that position. The element in the position where this happens is then returned as the answer. Written in Java, the method might look something like the following. (note the zero-based counting used in both `total` and `pos`.)

```
public int frequencyChoice(int[] arr, int[] freq, Random rng) {
    int s = 0, total = 0, pos = 0;
    for(int f: freq) { s += f; }
    int r = rng.nextInt(s);
    while(total <= r) { total += freq[pos++]; }
    return arr[pos - 1];
}
```

This algorithm is fine for sampling a single element from a handful of elements. However, if a large number of elements are to be randomly sampled one at the time, the first obvious optimization is to compute the sum of frequencies `s` only once, and pass this quantity to this method as an additional parameter instead of redundantly recomputing it every time in the first for-loop. If the array `arr` is large, the next opportunity to optimize is to precompute the **accumulation array** for these frequencies, as seen back in the Lab 55, "Accumulated wisdom". From this accumulation array, the **binary search** algorithm finds the position `pos` where the sum of elements first becomes larger than `r` quickly in logarithmic time, instead of using a linear time for-loop to find that position.

The previous discussion assumed that we are performing **sampling with replacement**, so that sampling a random element does not affect the frequencies used in future sampling. However, this problem becomes more interesting when adapted to **sampling without replacement**, so that each time the element in the position `i` gets randomly sampled, its associated frequency `freq[i]` should decrease by one! It is not immediately obvious how to perform such sampling efficiently. However, in this lab we shall implement a slight variation of the ingenious technique to achieve this goal, as explained by Daniel Lemire in his blog post "Sampling efficiently from groups".

In your labs project, create a new class named `LemireFrequencySampling`, in which you will be writing the following two static methods.

```
public static int[][] buildSamplingTable(int[] freq)
public static int updateSamplingTable(int[][] table, int r)
```

Given an arbitrary frequency array `freq`, the first method will build and return a two-dimensional sampling array. The topmost (zeroth) row of this sampling array has the same elements as the original frequency array `freq`. After this zeroth row, the length of each following row is always exactly one half of the length of the row immediately above it. The *i*:th element of each row equals the sum of the two elements in the positions 2*i* and 2*i*+1 in the row immediately above it. For simplicity, this method may assume that the length of frequency array is an exact power of two. (Any frequency array of some other length can always be **padded** with zeroes until its length becomes some exact power of two, without affecting the probability of any element being sampled.)

For example, given the frequency array `{4, 0, 1, 5, 2, 2, 3, 4}`, the resulting two-dimensional sampling array would have four rows. Row 0 would have eight elements, and equal that same frequency array. The length of the next row 1 would be 4, and its elements would equal `{4, 6, 4, 7}`, the sums of pairs of elements in the previous row. The length of the next row 1 would be 2, and its elements would be `{10, 11}`, since 4+6 = 10 and 4 + 7 = 11. The bottom row 3 would have just one element `{21}`, always equal the sum of the original frequencies.

The technique introduced in professor Lemire's blog post cleverly performs these accumulations **in place** in the original frequency array. However, the resulting compact representation of data also makes the algorithm that performs the actual random sampling without replacement slightly more

tricky. To nimbly sidestep such pitfalls, we choose to **trade space for clarity**, a tacit tradeoff similar to its more famous cousin of **trading space for time**.

The second method is given as argument the very sampling `table` constructed in the previous method, and a nonnegative integer `r` that serves the same purpose as the local variable `r` in the earlier linear time algorithm for frequency sampling. This method should find and return the position in the current frequency array (that is, the current zeroth row of the given sampling `table`) where the sum of frequencies up to that position becomes strictly greater than *r*. Before returning that position to the caller, the frequency of that position should also decrease by one, since the randomly sampled element is not replaced to the population.

Since our automated JUnit tester will repeatedly try out this method with frequency tables of over a hundred thousand elements, updating the entire sampling table to correspond to the new situation must be blazingly fast for such tests to finish in an acceptable total time. Fortunately, the ingenious construction of the sampling table allows us to achieve this feat in **guaranteed logarithmic time** with respect to the length of the frequency table for each individual sample. The sampling algorithm is simplest to implement as a while-loop that starts at the bottom row of the sampling table, and keeps climbing up until it has reached the zeroth row. During this climb, the body of the loop will update exactly one element of the current row, and look at exactly one element in the previous row to decide which direction to continue. The total work done by this loop is therefore guaranteed to be linear with respect to the number of rows in the sampling table. This quantity is in turn logarithmic with respect to the number of elements in the original array and its frequency table.

The while-loop of this method should keep track of the `row` and `col` position that its traversal up the sampling table is currently at, starting at the lone element at the bottom row. The body of this loop should first decrement the element at the current `row` and `col`, and then look at the element `e` at the position `[row-1][2*col]` to decide which direction to continue. If `r<e`, continue left to the column `2*col` in the `row-1`, keeping `r` as it were. Otherwise, continue right to the column `2*col+1` in the `row-1`, and also decrease `r` by `e`. Once you reach the zeroth row of the sampling table, decrement the frequency stored in the column that you ended up at, and return the index of that column (not its value!) to the caller.

Lab 68: Interval Sets I: Dynamic Set Operations

JUnit: IntervalSetTest.java

Bit vectors can be used to store a set of natural numbers in a compact fashion, needing only one bit of memory for every element of the appropriate type and range that could potentially become a made member of this set. However, this representation might still use space very inefficiently for **sparse** sets where most of the bits end up being `false`, so that only some dust of `true` bits is scattered throughout this astronomical emptiness. In situations where both the members and the nonmembers of the set tend to form contiguous **clusters**, a vastly more efficient representation for such sets can be achieved with ideas adapted from the technique of **run-length encoding** by storing not merely the individual elements, but entire consecutive **intervals** of contiguous elements that are members of the set.

This lab has you implement a rudimentary version of this kind of approach to represent such clustered sets of natural numbers in a highly compact fashion so that their basic **dynamic set** operations `add`, `contains` and `remove` end up being as efficient as the student will hopefully end up becoming effervescent while writing this code. This lab also serves an exercise of using the `LinkedList<E>` subtype from the Java Collection Framework along with its **list iterators** that not only allow the **bidirectional traversal** back and forth through the elements of the list one step at the time, but also constant time modifications with the operations `add` and `remove` performed at the current location of the iterator, as opposed to the eponymous list methods that operate on that list as a whole.

Compared to the more commonly used `ArrayList<E>` collections, linked lists are appropriate for situations where we require constant time **random access** to the elements in arbitrary positions, but always traverse the elements sequentially from left to right, with the option to occasionally update the contents of the list in constant time at the current position of the iterator traversal. For the `ArrayList<E>` collections, the rigidity of the underlying array makes such local modifications inefficient, since the elements that follow the current the position all have to be moved one step forward or backward after the modification to either make room for the new element that was added, or fill in the gap left behind by the element that was removed.

The best way to understand the behaviour of a `ListIterator<E>` instance acquired from the underlying `LinkedList<E>` object is to think of it as a **cursor** that points not at an element directly, but at the invisible space that separates each two consecutive elements of the list. For example, the underlying linked list of integers might contain the following elements, with the current location of the iterator cursor shown as a vertical bar between the elements 42 and 333:

```
17   99   42   |   333   81   4   87
```

Calling the method `next()` for the iterator makes it skip over the element that immediately follows it, and return the value of the element that was skipped over. The method `hasNext()` should generally be called before this move to determine whether the cursor has traversed past the end of

the list, to avoid throwing the `NoSuchElementException` for trying to skip over the nonexistent element past the end of the list.

For example, calling `next()` in the situation depicted above would return 333, with the new cursor position inside the list becoming

```
17    99    42    333   |     81    4     87
```

Since each **node** of the underlying linked list contains references to both its successor and predecessor nodes, the `ListIterator<E>` can support bidirectional traversal. The methods `previous` and `hasPrevious` operate symmetrically to the methods `next` and `hasNext`.

The method `remove` of the `ListIterator<E>` cursor removes the element that the cursor stepped over during its most recent `next` or `previous` operator. The iterator cursor remembers only the most recent element stepped over. Therefore, calling the `remove` method twice without calling either of the methods `next` or `previous` in between these two calls, the second call to `remove` will fail and throw an `IllegalStateException` to announce that the iterator object was not in a proper state to perform the requested operation.

For example, calling `remove` after stepping over the element 333 with the previous call to the method `next` would create the following situation, with the cursor position unchanged:

```
17    99    42    |     81    4     87
```

The method `add` of the iterator cursor adds a new element before the cursor position. Unlike `remove`, this method can be called any number of times regardless of whatever calls to `next` and `previous` have taken place in the interim. For example, calling the method `add(38)` for the iterator cursor would produce the new situation

```
17    99    42    38    |     81    4     87
```

Make sure to appreciate what a different thing it is to call the method `add` for the iterator to add the new element to the current position of the iterator cursor, as opposed to calling the method `add` in the list object as a whole to add the new element at the end of the list, regardless of any iterator cursors that might exist at that moment.

These preliminaries give us enough to begin implementing the set of intervals. In your labs project, create a new class `IntervalSet` whose instances represent sets of natural numbers stored as **intervals** of consecutive integers that are members of that set. For example, an `IntervalSet` instance that will print out as `"[4–10, 15, 20–27]"` would contain all integers from 4 to 10, the integer 15, all integers from 20 to 27, and no other integers other than these. Each interval is considered to be inclusive at both ends, and the **singleton** intervals whose `start` and `end` values are equal are printed as the starting value only, without the separator dash and the end value.

You should start by first defining the following **nested class** inside the `IntervalSet` class to represent an individual interval:

```
private static class Interval {
```

This class should have two public integer data fields `start` and `end`, and a constructor to initialize these fields from its parameters. Note that we are not designing this nested private class to be immutable, but allow the methods of the surrounding `IntervalSet` to modify the `start` and `end` values of any given `Interval`. You should also override the `toString` method of the `Interval` class to return the string of the form `"start-end"` whenever `end>start`, and just `"start"` when these two values are equal.

Each instance of `Interval` represents the fact that the set contains all elements from `start` to `end`, inclusive. Each `IntervalSet` is therefore a collection of disjoint intervals stored in a linked list. Each `IntervalSet` instance should therefore contain the field

```
private LinkedList<Interval> intervals = new LinkedList<>();
```

in which the intervals that form the interval set are kept in strictly increasing order. Furthermore, we will ensure during the execution that all the intervals are **maximal** so that no two of them could be combined into one larger interval that contains both of the original intervals. For example, instead of storing two intervals `10-20` and `21-30`, the list should only contain one interval `10-30` with the same information in a more compact form.

To help debugging the dynamic set operations that you will be writing in this class, you should first override the `toString` method for the `IntervalSet` class to return a string that contains square brackets as its first and last characters, followed by the individual intervals as produced by the `toString` method of the `Interval` class, consecutive intervals separated by exactly one comma and one whitespace character. Make sure to also not have any silly trailing commas or whitespace in the end. Once you have completed this method, you can start filling in the bodies of the required dynamic set methods

```
public void add(int start, int end)
public void add(start)
public void contains(int start, int end)
public void contains(int start)
public void remove(int start, int end)
public void remove(int start)
```

Since the operations on **singleton** intervals whose `start` and `end` are equal would presumably be common in the user code of this data structure, we will provide the convenience versions of the dynamic set methods that take only one parameter each. All three convenience methods should be simple one-liners that call the more general two-parameter version of that same method by duplicating their parameter. The student should read through the explicit test cases of our JUnit test

class to get an idea how these methods are used, and what their expected results are expected to be in the provided test situations.

The method `add` should first create a new `Interval` object for its given `start` and `end`, and ask the list of `intervals` for a list iterator to traverse through the existing `intervals` in the list. For each existing interval, check whether it could be **absorbed** into the new interval that is being added to the set. To absorb an existing interval, `remove` it from the list, and update the `start` and `end` values of the new interval accordingly to make the new interval fully contain the absorbed one. For example, the interval `10–20` currently being added to the set could absorb an existing interval `5–15` and become the interval `5–20`, so the now redundant interval `5–15` can be removed from the list. Note that an interval can also absorb another interval if either one of these intervals `starts` at the value that is one larger than the `end` value of the other interval. For example, the new interval `4–9` can absorb an existing interval `10–15`, and become a single interval `4–15`.

As the traversal through the existing `intervals` removes some of the existing intervals that get absorbed into the new interval that is currently being added to the set, one of the two things must eventually happen. Once some existing interval reached in the traversal starts after the end of the new interval being added, the new interval is simply added at that position, and the method can terminate. Should the traversal of the list reach the end of the list before such an existing interval is found, the new interval is tacked on at the end of the list.

Thanks to our design decision to store the intervals in the maximal fashion without redundancy, the logic of the method `contains` to determine whether all numbers from `start` to `end` are currently members of the set is straightforward. This method only needs to iterate through the list of existing intervals to find an interval that fully contains the entire interval being searched for. If no such interval exists, the method can return `false`, safe in the knowledge that at least one of the numbers from `start` to `end` is not a member of this set.

As usual with dynamic sets, the logic of the `remove` method is the most complicated of the three dynamic set operations. (Removing an element from any data structure decreases the **entropy** of that data structure, so the operation must perform work against the grain of the natural flow of the universe.) The `remove` method should iterate through the existing `intervals` in the list, and update the list and the intervals stored therein depending on the way that the existing interval overlaps the interval of the given values being removed. To illustrate the different possible cases and the treatment that they require, let us assume for simplicity that we are currently removing the particular interval `10–20`, to make it easier to visualize what ought to be done for the given existing interval in the list.

- If the existing interval is fully inside the interval being removed, such as `10–14`, `15–18` or `13–20`, simply remove that existing interval from the list.
- If the existing interval contains the interval being removed so that their either endpoint coincides, such as `10–30` or `5–20`, update the opposite endpoint of the existing interval, and terminate.
- If the existing interval properly contains the interval being removed so that the endpoints do not coincide, such as `7–30`, modify that existing interval to end one step before the removed

interval begins, and add a new interval for the values that follow the interval being removed. For example, removing `10–20` would split the existing interval `7–30` into two separate intervals `7–9` and `21–30`. Again, you can terminate the method after this operation.

- If the interval being removed partially overlaps the existing interval from the left, such as with the existing interval `15–40`, update the `start` value of that existing interval to one past the `end` of the interval being removed. For example, removing `10–20` from the existing interval `15–40` makes that interval to be `21–40`.
- Symmetrically to the previous item, if the interval being removed partially overlaps the existing interval from the right, such as with the existing interval `0–14`, update the `end` value of that existing interval to one less the `start` of the interval being removed. For example, removing `10–20` from the existing interval `0–14` makes that interval to be `0–9`.

The mass test of these three operations creates a large number of pseudorandom intervals to be added to and removed from the current `IntervalSet`. You can uncomment the lines that print out the current contents of the set in the mass test method to see the results that your methods produce for all these pseudorandom additions and removals.

Lab 69: Interval Sets II: Iteration

JUnit: IntervalIteratorTest.java

The three dynamic set operations `add`, `contains` and `remove` by themselves are not enough for the common use case of **iterating** through the elements of that dynamic set so that each element gets visited exactly once through this traversal. To follow the lead of the classes in Java Collection Framework, we shall adapt the `IntervalSet` class from the previous lab to also allow this. Since it must be possible for multiple iterators to simultaneously exist and point to different locations inside the same collection, the collection object itself cannot act as its own iterator. The iterator objects have be constructed from a separate class that has sufficient access to the `private` internal details of the collection that it iterates through. An **inner class** is just what the doctor ordered for this purpose. Therefore, inside your `IntervalSet` class, define the inner class

```
private class IntervalIterator implements Iterator<Integer>
```

whose instances act as independent iterators on the particular `IntervalSet` instance from which they were originally acquired. Since we define `IntervalIterator` to be an inner class, instead of a `static` nested class, its instances cannot exist on their own independently of whatever other objects may exist that the moment, but these inner class objects can only exist in the context of some mothership object of the outer class `IntervalSet`. Making the class `IntervalIterator` to be specifically an inner class allows its methods direct access all members of the outer class, even the `private` ones. This doesn't even require any special syntax from these inner class methods, but this access happens exactly the same way as these methods access their own members.

Your `IntervalIterator` objects should themselves contain a data field that is a reference of type `ListIterator<Interval>`, to keep track of which `Interval` in the list of `intervals` in the outer class object their traversal is currently at.Another `int` data field named `nextValue` will then keep track of the particular integer value inside the current `Interval` the iterator's traversal is currently at. After initializing these fields in the default constructor of `IntervalIterator`, you have everything you need to implement the two required methods of the `Iterator` interface

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

Even though we could make the iteration more general by defining its type to be a `ListIterator` with the symmetric `hasPrev` and `prev` methods, we will keep this lab simple by allowing iteration only to the forward direction. The method `hasNext` should return `true` if the previous `ListIterator<Interval>` reference is non-null, which means that at least one value remains to be visited during the current iteration, knowing that we never store empty intervals into the `intervals` list. The method `next` will return the current value of `nextValue` inside that current `Interval` and meanwhile also increment `nextValue`, telling the list iterator to step to the next `Interval` in the outer `intervals` list whenever the value of `nextValue` exceeds the `end` value of the current `Interval`.

To keep things simple, we also choose not to allow mutation of the `IntervalSet` directly through the iterator methods `add` and `remove`, so these two methods don't have to be implemented in the inner class. Once you have completed this inner class, you can add the following instance method to your `IntervalSet` class

```
public Iterator<Integer> iterator() { return new IntervalIterator(); }
```

to allow the outside user code to acquire iterator instances to the particular `IntervalSet`, and finally change the signature of your `IntervalSet` class to the form

```
public class IntervalSet implements Iterable<Integer> {
```

to proudly tell the whole world the good news that it has now become possible to iterate through the instances of your `IntervalSet`! Even the **for-each loop** will now work for the `IntervalSet` instances out of the box, just like it does for all those snobby array and collection types who landed on Plymouth Rock back then the Java language and its standard library were first founded.

The iterators in the Java Collection Framework and elsewhere tend to display great variation on how tolerant they are towards their underlying collection being mutated during the iteration. In one extreme, some collection subtypes have been hardened to tolerate even **concurrent** access and mutation from multiple execution threads, without interfering with the iteration over the contents of that set at the time the iterator was created. However, in this lab we are going to make the `IntervalSet` iterators to be of the other extreme by being **fail-fast**; any mutation of the collection while it is being iterated makes any active iterators immediately fail the next time that their methods `hasNext` or `next` get called.

This design decision not only makes our lives simpler, but also allows us to demonstrate the useful programming technique of **timestamping** to allow our iterators to immediately detect any mutations done one the underlying collection, without these iterators actually having to actually look at the elements of the underlying collection to determine whether some stealthy mutation has occurred since the that iterator was last accessed. In the outer class `IntervalSet`, define a field

```
private long timestamp = Long.MIN_VALUE;
```

and modify the methods `add` and `remove` of the `IntervalSet` class to increment this timestamp every time that they are called. The inner class `IntervalSetIterator` should then also have a new data field

```
private long initialTimestamp;
```

initialized in the constructor of `IntervalSetIterator` to the current value of the outer class `timestamp` field. The iterator's methods `hasNext` and `next` should then first check that the initial timestamp stored inside that iterator object still equals the timestamp of the outer class, and

if this is not the case, throw a `ConcurrentModificationException` instead of performing the requested operation.

Since the timestamp needs to be stored only once per each `IntervalSet` instance and once per iterator object, the use case of millions of iterators being vanishingly unlikely in practice, we might as well use the eight-byte `long` integer type to keep track of the timestamp values to ensure that they can't realistically overflow during the use of this data type. ("Four billion mutations on a set should be enough for anybody" just somehow sound like famous last words to me, unlike the same sentence starting with "Sixteen billion billion mutations..." said in the voice of Carl Sagan.)

Lab 70: Cup Tournament Survival

JUnit: CupSurvivalTest.java

Consider a sportsball cup tournament between n teams numbered from 0 to n-1, where n is guaranteed to be some power of two. Each round, pairs of teams play a match that determines which team advances to the next round. The number of teams is halved in each round, until the final match determines the winner of the entire tournament. For example, if n = 8 so that the teams are numbered from 0 to 7, the first round sees the matches between teams 0-1, 2-3, 4-5 and 6-7. The method that you write in this lab will compute the exact probability for each team to win the tournament, using an iterative process described below to build up these probabilities bottom up, using the `Fraction` type to exactly represent the intermediate and final probabilities.

In your lab project, create a new class named `CupSurvival`, and in there the static method

```
public static Fraction[] computeSurvival(Fraction[][] winProb)
```

that returns an array of n fractions that together add up to one, giving the winning probabilities for each individual team. The parameter `winProb` is an n-by-n array of fractions so that each element `winProb[i][j]` is the exact probability that the team `i` wins the game playing against the team `j`. (Perhaps these probabilities were computed by the famous sportsball handicapper "Ilk the Finn".) Since there is no crying or ties in sportsball, each match is guaranteed to see exactly one of the participants advancing to the next round. The two symmetric probabilities `winProb[i][j]` and `winProb[j][i]` always add up to exactly one.

Instead of estimating the winning probability for each team in **frequentist** fashion by simulating the entire tournament a large number of times and estimating the winning chances of each team as the percentage of these simulated tournaments that the team ended up being the top dog, we will rather compute these probabilities as exact fractions from the bottom up without any randomness. The summations in the **marginalization** formulas for these probability calculations can spread out to be pretty hairy, but that is why we have bred our machines to perform these calculations, instead of expecting us slow hairless apes to grind through these formulas by hand.

For the bookkeeping needed to compute the winning probabilities from the bottom up, the method should define as a local variable a two-dimensional array

```
Fraction[][] survival = new Fraction[k][n];
```

where k is the number of levels in the tournament cup tree. For example, k = 3 when n = 8. Each quantity `survival[r][i]` will give the probability that the team i will win the match against its opponent in round r. Once the method has filled in this table, the method can simply return the entire row k - 1 of this array as the final answer, since the probabilities of that row are precisely the probabilities for each team to become the winner of the entire tournament.

This method should start by filling all elements of the bottom row `survival[0]` with values equal to one. Before you start coding this method, you might want to define two named constants

```
private static final Fraction ZERO = new Fraction(0, 1);
private static final Fraction ONE = new Fraction(1,1);
```

inside your class to make your method code easier to read.

After initializing the bottom row with ones, each probability `survival[r][i]` in the higher rows can be computed with a recursive formula that involves the probabilities of the previous row `r-1` and the winning probabilities for the individual matches between the given two teams given as a parameter to this method. However, to prevent this recursive computation from spending an exponential time by redundantly recomputing the same entries in the lower rows over and over again, your method should fill in the `survival` array in **dynamic programming** style using three nested for-loops, with no actual recursion used anywhere! The outer loop will count through the rows from 1 up to *k* - 1. For each row, the inner loop will count from 0 up to *n* - 1 to fill in the survival probabilities for the teams during that round. This computation will require another loop inside its body, which makes the entire algorithm consist of three levels of nested loops. (Hey, it's action, isn't it?)

The probability for the team `i` to emerge as the winner in its match in the round `r` is the product of two separate probabilities, representing the **parlay** of two separate things that both must happen for the team `i` win its match in the round `r`. First, the team `i` must reach the round `r` to get to play there for a chance to advance; that is, the team `i` must emerge as the winner of its match in the previous round `r-1`. Conveniently for us, this very value was already filled in the `survival` table during the previous round of the outer loop! Having crossed the tall hurdle of showing up, the team `i` must win its match against the team that becomes its opponent in the round `r`. Marginalization of the combined probability into the sum of its disjoint elementary possibilities expressed the desired probability as the sum of simple terms of the form `survival[r-1][j]*winProb[i][j]`, where the innermost loop variable `j` counts through all the teams that could end up becoming the opponent for team `i` during the round `r`.

The only remaining problem is to determine which teams this innermost loop must go through to cover precisely the teams that can meet the team `i` in round `r`. We start by noting that if the team `i` starts at the position `i` in the cup tournament tree and wins its first match, it will be in position `i/2` in the next round, where the division operator `/` is the **truncating integer division**. Continuing this reasoning, team `i` will reach the position $i/2^k$ of that level in the cup tournament tree after winning `k` matches in a row. Even though Java does not have integer exponentiation function in the language, powers of two can be handily computed with **bit shifts**, since the expression `(1 << k)` always evaluates to the desired power 2^k.

To determine the potential opponents of team `i` in the round `r`, first compute the position `p` where the team `i` would end up in the round `r-1`, were that team to win all its matches to reach that far. If that position `p` is odd, the opponent would play in position `p-1` in that round; and if the position `p` is even, the opponent would play in position `p+1`. To traverse down the cup tournament tree back to

the original teams that could end up in that opponent's position in the round `r-1`, simply multiply the opponent's position `p` by the factor 2^{r-1} to find the first team in the bottom row that could become the opponent for the team `i` in that round. For round `r`, there are exactly 2^{r-1} such teams, and since they are all consecutive in the original bottom row, it is easy to loop over them. For example, the potential opponents of whichever of the teams 0 and 1 wins the first match are the teams 2 and 3 in round two, and in round three, the teams 4, 5, 6 and 7.

(Readers who will next take the course on data structures and algorithms after this course may then note the similarity between the previous formulas of moving up and down the cup tournament tree and the formulas used to move up and down the positions of the binary heap data structure to implement an efficient priority queue.)

Even when the original winning probabilities are simple fractions such as one half or one third, the summation formulas over all possible opponents for round `r` will quickly produce complicated fractions that have dozens of digits in their numerators and denominators. To make these results exactly reproducible over all computer systems of past, present and future so that we can write a meaningful JUnit test for them, we cannot compute these results in floating point.

Lab 71: The Gamma And The Omega

JUnit: EliasCodingTest.java

Positive integers are simple enough to encode to be processed by mindless machines as **binary numbers** using sequences of zeros and ones. However, encoding not merely a single integer but an arbitrary sequence of integers requires some way to recognize where the binary representation of each individual integer ends and the next one begins. For example, would some given sequence of 32 bits encode a single 32-bit integer, two 16-bit integers, one 23-bit integer followed by a 9-bit integer, or any other possible way to partition those bits into separate pieces? This question has to be resolved without any hand waving such as using additional separator symbols between the binary numbers to make the boundaries between the payload bits stand out.

Of course, we can always agree that each number is stored in some fixed number of bits, typically 16 or 32, or these days actually 64. (As an aside, whenever you need to make up a **magic number** to limit something in computing, make that limit some power of two, or some power of two minus one, and suddenly nobody will question even the existence of your made-up limit, but will imagine the implementation artifact that created this limit to be some kind of immutable law of the universe.) However, not only would such policy set a hard upper bound to the magnitude of the integers that could be encoded in this system, but also end up transporting air by stretching all those itty bitty numbers to fill up this Procrustean bed by padding them from the front with redundant zero bits that convey no additional information.

We already saw one way to solve this problem back in lab 15 with the Zeckendorf encoding. Break down the given positive integer as the unique sum of nonconsecutive Fibonacci numbers, and then use the bits to indicate which Fibonacci numbers appear in this sum. The impossible bit sequence `11` will then act as a separator between the encodings of two adjacent integers in the sequence. In the Zeckendorf encoding, the separator pattern `11` is placed at the end of the encoding of each integer, akin to the period that ends each sentence in English. In this lab, we take a look at two alternative ways to solve this same problem by prefixing each integer with an unambiguous indicator of its bit length. We start with the simpler but less efficient technique of Elias gamma coding, and follow with its more advanced and tight cousin of Elias omega coding.

To encode some positive integer x whose binary representation consists of $N + 1$ bits (that is, N is the highest power of two that fits inside x), the Elias gamma coding prefixes the length N in **unary** by using exactly N zeroes, followed by the $N + 1$ bits of the integer x itself. For example, consider the integer 13, whose binary representation is $8 + 4 + 1 = 1101_2$, which makes $N + 1 = 4$. This number would be gamma encoded as the bit sequence `0001101`. Note that since the highest bit of any positive integer is known to be 1, the `1` in the middle cleverly plays a dual role as both the terminator of the length prefix of unary zeros, and the highest order bit of the x itself!

In your labs project, create a new class named `EliasCoding`, and there the following two methods to encode and decode a list of `BigInteger` values into a seamless sequence of zeros and ones using the Elias gamma coding. Of course, an adult realsies implementation of these method would emit actual bits and bytes instead of Unicode characters, for maximal compression. However, this

lab exercise should be considered to be more of a **proof of concept** for learning and demonstration purposes, so the resulting sequences of bits are expressed as strings that can contain only the characters `0` and `1` to simulate the bit sequences that would actually be emitted.

```
public static String encodeEliasGamma(List<BigInteger> items)
public static List<BigInteger> decodeEliasGamma(String bits)
```

In spirit of "design by contract", the encoding method can assume that all `items` are positive, and the decoding method may assume that `bits` is a complete and legal Elias gamma coding of some list of `BigInteger` values. Neither method has to be able to handle and recover from errors in situations where this is not the case. Being the party that is in complete control of the material facts that affect the preconditions, the caller always bears the fault of the consequences for passing the method argument values that violate the preconditions of that method.

Instead of performing all the bitwise computations by yourself, it pays to remember that the people who created the Java standard library did not choose the classes and methods included in this club as some sort of dadaist abstract performance art event, but to serve programmers by giving them operations that were not important enough to gain entry to the inner circle of the language itself, but are still common enough to show up in all problem domains that it would just be wrong to expect every programmer to roll their own versions of those operations.

Therefore, before coding the above two methods, you should take a moment to appreciate how the constructor of the `BigInteger` class has been overloaded to accept the **radix** as the second parameter, so you can make the parameter string to be in binary instead of base ten just by using the radix of two. Similarly, the method `toString` has been overloaded to accept the desired radix of its result as the second parameter. Finally, the method `bitLength` returns the length of the binary encoding of the positive `BigInteger`, without actually constructing that string representation itself. Subtracting one from the result given by the `bitLength` method therefore tells you the highest power of two that is less than or equal to that `BigInteger`.

The Elias gamma coding is simple enough to understand and implement. However, a moment's meditation reveals its inefficiency in that it almost doubles the number of bits needed to encode each individual integer! We can do *far* better than this. Skipping over the intermediate scheme of the Elias delta coding to get straight to the more advanced **Elias omega coding**, the bit length of each integer is not prepended to its binary representation in such a wasteful splurge, but is encoded recursively using the very same Elias omega coding!

This clever idea has an obvious chicken-and-egg problem of how we can possibly know whether the integer that was currently decoded from the sequence of bits is actually a member of the integer sequence being encoded, or denotes the length of the integer whose encoding follows it in the bit sequence. Since the highest order bit of any integer is known to be 1, this highest order bit can again play a dual role as both the highest order bit of that number, and the **marker** that indicates whether you need to look at the continuation of the bit sequence after decoding the current integer. Once this highest bit is zero, the decoding of the preceding integer is complete, and you can start decoding the next integer in the sequence after that zero bit.

```
public static String encodeEliasOmega(List<BigInteger> items)
public static List<BigInteger> decodeEliasOmega(String bits)
```

The computational steps needed to encode and decode any positive integer in the Elias omega coding are given on the Wikipedia page, and we shall leave the task of translating them to the higher-level syntactic structures of Java as an exercise for the reader. We shall still note that instead of using recursion, this method is more stylishly written as a while-loop where the integer N decreases in each round of prepending the length of the current number to the encoding. The loop terminates when $N = 1$, meaning that the length of the next block of bits plus one can be expressed in two bits.

Anyway, it's not like this recursion could realistically end up being very deep. As the bit length of an integer grows only **logarithmically** relative to the absolute magnitude of that integer, the extra zeros and ones needed to encode the length of an integer will be but a rounding error of a rounding error compared to that integer itself. The Elias omega coding becomes even more efficient as the numbers in the sequence grow large. For example, to quote the example given on the Wikipedia page, the Elias omega coding of 10^{100}, one **googol**, requires only 15 additional bits for the length header before the 333 bits that encode the actual one googol in binary, plus that extra zero bit tacked to the end to indicate that one googol was the integer that was actually encoded, instead of giving the decoder the bit length of the actual integer that follows it. (Such an unimaginably huge integer would not even fit inside our physical universe, even if we could somehow carve each one of its individual bits into one elementary particle!)

As the encoded integers grow even more gigantic, the extra bits for the recursive length header vanish into utter insignificance. For example, according to the same Wikipedia page, encoding the behemoth 10^{10000} requires only 22 additional header bits for its length (and the zero marker bit in the end) in addition to the 33,243 bits needed to encode the hundredth power of one googol. We will gladly pay this chump change to gain the ability to always determine without any ambiguity or error where one integer ends and the next one in the sequence begins.

Lab 72: Lazy Generation Of Permutations

JUnit: PermutationGenerationTest.java

Properties of individual permutations were already thoroughly examined back in labs 45 to 47, in which we implemented methods for the algebraic operations, cyclic forms and Lehmer coding of permutations. This lab examines the extremely important related combinatorial problem of systematically generating all permutations of the given length *n*, one permutation at the time, so that each one of the *n*! possible permutations is visited exactly once during this process.

Algorithms to systematically generate permutations are well known classics in computer science and combinatorics. However, most of the online material that discusses these algorithms such as the Steinhaus-Johnson-Trotter plain changes algorithm and Heap's algorithm tends to present these algorithms in a recursive form, for simplicity of understanding. However, this recursive form is not practical for generating permutations or other combinatorial sequences that consist of an exponential number of elements. Instead of wasting memory by first collecting all permutations into a list for the user code to then loop over, we would like the permutation generator to produce these permutations in a **lazy** fashion one permutation at the time.

The permutation generator should keep track of only the current permutation plus possibly some **auxiliary data** whose exact nature depends on that permutation generator algorithm. This information should be enough for the generator to swap the correct elements inside the current permutation to produce the next permutation that follows the current permutation in the sequence of permutations being generated, making only local decisions based on this information without using any recursion. This way the outside user code can at any time tell the generator when it is done with the current permutation and needs to see the next one, that is then filled in the same permutation array that previously held the current permutation. Instead of having the first construct the entire sequence of permutations into a list, which would soon become prohibitive when iterating through billions of permutations, the generator needs memory only for the current permutation and the auxiliary data.

To allow different algorithms for permutation generation to exist within the same class hierarchy, we make each of these algorithms a subclass of an abstract superclass `PermutationGenerator`, as provided by your instructor here. You should not modify this class in any way whatsoever, but implement the following two algorithms as concrete subclasses of this abstract superclass. Despite being abstract, the superclass defines a constructor for the use code to give it an integer array that both the user code and the permutation generator share to represent the current permutation of the sequence. The user code is responsible for not mutating the contents of this shared array object between computing the next permutation from the current one.

Most importantly, this abstract superclass defines the **template method**

```
protected abstract boolean next(int pos, int[] perm)
```

that the `public` method `next` in the abstract superclass calls to ask the algorithm implemented in the concrete subclass to rearrange the elements of the current `perm`utation to turn that array into

the next permutation in the sequence. This method also receives the position of the current permutation in the sequence as the parameter `pos`, since the position information is needed in some permutation generation algorithms to decide what to do next. This method should return `true` is a new permutation was produced, and return `false` when the current permutation is the last one in the generated sequence so that no further permutations exist after it.

We start with the classic algorithm that produces the permutations in lexicographic order, by each time first swapping two tactically chosen elements and then reversing the suffix of the array after the second swapped position. These tactical steps are explained in the section "Generation in lexicographic order" of the Wikipedia page "Permutation", but we can illustrate them here with an example. Let us assume that the current permutation is `{1, 5, 0, 3, 4, 2}`. whose successor permutation in the sequence we are trying to determine.

Counting down from the second last position before the end of the array, find the first position `k` in `perm` that satisfies `perm[k]<perm[k+1]`. If no such position exists, this permutation is the lexicographically largest permutation that ends the sequence, and the method `next` should return `false`. In this example permutation, this is the position marked with green background. Then, again counting down from the last position of the array, find the first position `l>k` that satisfies `perm[k]<perm[l]`. In this example permutation, this is the position marked with orange background. Swap the two elements in positions `k` and `l`, after which you reverse the subarray from position `k+1` all the way to the end. In our example permutation, swapping the elements 3 and 4 changes the array into `{1, 5, 0, 4, 3, 2}`, and reversing the subarray from position k+1 to the end turns this into `{1, 5, 0, 4, 2, 3}`, the next higher permutation that follows the previous permutation in the lexicographic order.

To implement this algorithm, create a class named `LexicographicPermutation` in your lab project that extends the class `PermutationGenerator`. To allow creation of objects of this concrete subclass, it also needs to have the constructor

```
public LexicographicPermutation(int[] perm) { super(perm); }
```

that passes the argument array `perm` on to the superclass constructor.

Zaks' algorithm of **prefix reversals**, as discovered in 1984 by Shmuel Zaks, is not as well known as its more famous cousins such as the plain changes algorithm or Heap's algorithm. However, Zaks' algorithm is interesting in that it always performs exactly one prefix reversal to the current permutation to produce the next permutation that follows it in the generated sequence! A precomputed scheme is used to quickly calculate the length of the prefix that needs to be reversed to turn the current permutation into its successor, generating the entire sequence of $n!$ different permutations almost as if by magic.

Create another subclass of `PermutationGenerator` named `ZaksPermutation`, with a similar constructor as the one seen in the class `LexicographicPermutation`. Before implementing the template method `next` in the `ZaksPermutation` subclass, you need to first write the following

`static` utility function used to determine the length of the prefix to be reversed in the current permutation, based on the position of that permutation in the generated sequence:

```
public static int zaksSequenceElement(int n, int pos)
```

This utility method is given the length of the permutation `n`, and the position `pos` of the current permutation in the permutation sequence, the position counting starting from zero. This method should compute and return the value in the position *pos* of the prefix length sequence S_n. Each sequence S_n is defined to contain exactly $n! - 1$ positive integers. As the base case of this recursive definition, the sequence S_2 contains just one element whose value is 2. Reversing the prefix of length two is required to turn the first permutation `{0, 1}` of length two into its successor permutation `{1, 0}` that ends that sequence.

After this base case, each sequence S_n is a concatenation of exactly n copies of the sequence S_{n-1}, with the number n tacked in as a separator between these concatenated subsequences. For example, the sequence S_3 consists of three copies of the singleton sequence S_2 with the new element 3 between each S_2, giving us the five-element sequence 2, 3, 2, 3, 2. The sequence S_4 consists of four copies of the sequence S_3 with the new element 4 inserted as a separator between them, which makes this sequence to be 2, 3, 2, 3, 2, 4, 2, 3, 2, 3, 2, 4, 2, 3, 2, 3, 2, 4, 2, 3, 2, 3, 2. Note how the length of S_3 equals $3! - 1 = 5$, and the length of S_4 equals $4! - 1 = 23$.

To make the method run efficiently even when `n` gets larger, it definitely should not attempt to construct the sequence S_n explicitly. The element in the requested position should be computed using only integer arithmetic whose details are left for the reader to discover. It would probably be useful to precompute the factorials of integers from 2 to 13 to some `static` array for a quick lookup, seeing that these factorials are needed in every turn during this computation. Once this utility method is working as confirmed by the JUnit tests, you can finally implement the method

```
protected abstract boolean next(int pos, int[] perm)
```

in this subclass to compute `zaksSequenceElement(perm.length, pos)`, and use that result to decide the length of the prefix of `perm` that needs to be reversed to take us to the next permutation. Before doing this, this method should first determine whether `pos` plus one is greater or equal to the factorial of `perm.length`, which happens when the entire sequence of permutations has been generated and the method should just return `false`.

Zaks' permutation generator will work just as well if converted to reverse the suffix of the array each time, instead of reversing the prefix. This sequence will then go through the permutations in a different order, but finish with the same lexicographically largest permutation as the prefix version.

Interested readers can check out Combinatorial Object Server, an interactive website to produce colourful visualizations for various algorithms to generate permutations and other combinatorial objects. The article "A Unified Framework to Discover Permutation Generation Algorithms" introduces a general template of a permutation generator algorithm that gives all these classic permutation generators as special cases.

Lab 73: Newton-Raphson Fractals

For testing: NewtonRaphsonMain.java

This problem was inspired by the video "Newton's Fractal" by the popular mathematics channel 3Blue1Brown, and is for many reasons suitable for at least the subset of computer science students who are interested in properties of both the complex numbers and their functions. Despite that fact that neither man could have possibly anticipated them, these so-called Newton fractals naturally emerge out of the classic Newton-Raphson method, now several centuries old famous iterative numerical approximation method to find a root for any real-valued function that is differentiable everywhere and is in certain sense sufficiently well-behaved, the tacit assumption that underlies all numerical methods. The choice of the initial guess to start this iteration at determines not just how quickly this method converges to a root, but also which root the method converges whenever function has more than one of those.

As explained in the video, Newton's method works just as well for functions of complex variables, needing nothing more but a simple substitution of a complex variable *z* in place of the real variable *x*. However, when the iteration of some function is not confined to the narrow straitjacket of the real line, but takes place on the two-dimensional complex plane that allows things go around other things in infinitely many squiggly ways without bumping heads with each other all the time, the complex plane gets partitioned into disjoint areas depending on which root the iteration converges to when started with a guess from that area.

Even for simple polynomials, these areas are not some boring Voronoi diagrams, but fractals with infinitely complex details on the boundaries between these areas. Once again illustrating the principle of how the number three is fundamentally different from the number two in computer science, a moment's thought reveals that between any two points in two different areas, there must exist a point between them where the invisible forces pulling towards those two roots are in perfect balance so that the iteration has no choice but to lead into some third root. Since the same reasoning can be continued exactly the same using the third point as one of these two endpoints, this detail will keep on keeping on like the Energizer bunny, no matter how deep you zoom into it!

Since Java does not have a `Complex` number type in the language, our first task for this lab is to define such a type. In your labs project, create a new class named `Complex`, whose instances represent immutable complex numbers made up of `real` and `imaginary` parts, each of which is, unusually for these labs, a `double` precision floating point number. We will not be zooming in deep enough into the resulting image for the limited precision of the `double` type to become an issue, and besides, no matter what finite precision you use, every image of a fractal of this nature is doomed to inexact anyway due to the inaccuracies of floating point iteration. (On the other hand, rendering fractals with some woefully insufficient floating point precision can create intentionally dramatic displays of the glitch art aesthetic.)

This class should have a constructor to initialize these fields from its arguments, the **accessor** methods `getRe` and `getIm` to read their values, and a `toString` method to produce a human-readable description of that complex number to aid you in debugging the following methods for

arithmetic, analogous to the four methods for the arithmetic operations that were seen in our lecture example class `Fraction`:

```
public Complex add(Complex other)
public Complex subtract(Complex other)
public Complex multiply(Complex other)
public Complex divide(Complex other)
public double abs()
```

Consult any introductory online or printed material on complex numbers for the formulas to perform arithmetic on complex numbers (especially the division), and how to calculate the **absolute value** of a complex number. (Instead of the computing absolute value itself, it might actually be better to define the function to return its square, to avoid the expensive square root calculation that usually isn't even needed for the use cases of this absolute value.)

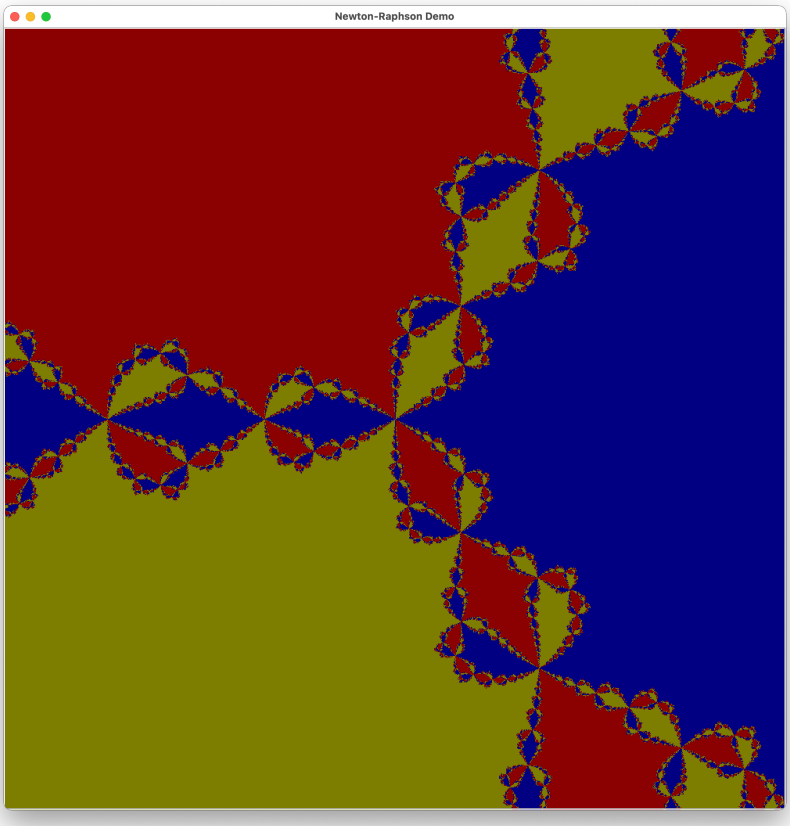
Once you have defined the class to represent `Complex` numbers, create a class `NewtonRaphson` in your labs project. This class needs to have just one method that computes the Newton-Raphson fractal, based on the seven parameters that define the finite square peephole into the infinitely large complex plane, the function whose root we are looking for, the first derivative of the said function, and finally a function that maps `Complex` points to the hues used to colour that partition with. (The last function allows us to render this fractal without having to solve these roots beforehand.)

```
public static BufferedImage createNewtonRaphson(double topRe, double topIm, int pixelN, double pixelSize, Function<Complex, Complex> f, Function<Complex, Complex> fp, Function<Complex, Float> hueFunction)
```

The first two parameters `topRe` and `topIm` are the top left corner of the peephole that is being rendered into a new `BufferedImage` instance created by this method, so that the width and the height of this square image are `pixelN` pixels. The parameter `pixelSize` gives the size of the square on the complex plane represented by each individual pixel. The `Function` object `f` is a black box that contains the function whose root is being numerically iterated, and the similar black box `fp` gives its first derivative. Once the iteration from the given starting point z_0 has converged numerically into some z_n, the pixel that corresponds to the point z_0 is coloured using the **hue** acquired by applying the `hueFunction` to the point of convergence z_n.

This method should start by creating the new `BufferedImage` instance that will be returned from this method in the end. Then loop through the pixels `(x,y)` of that peephole, and compute for each pixel which complex number z_0 on the underlying infinite complex plane it corresponds to. Iterate the Newton-Raphson formula $z_{n+1} = z_n - f(z_n)/f'(z_n)$ until convergence, using the methods of your `Complex` class to perform the arithmetic operations in absence of **operator overloading** that for some mysterious reason, the Java language still does not have, at least as of this writing.

After convergence, colour that pixel with a colour whose **hue** comes from applying the given `hueFunction` to the point that this iteration ended up, whose **saturation** is 1.0f, and whose **brightness** is computed from the number of iterations needed to reach convergence within some



reasonably small `EPSILON` that you have defined as a named constant in your class. Note the method `HSBtoRGB` in the `Color` class that makes this colour computation a breeze.

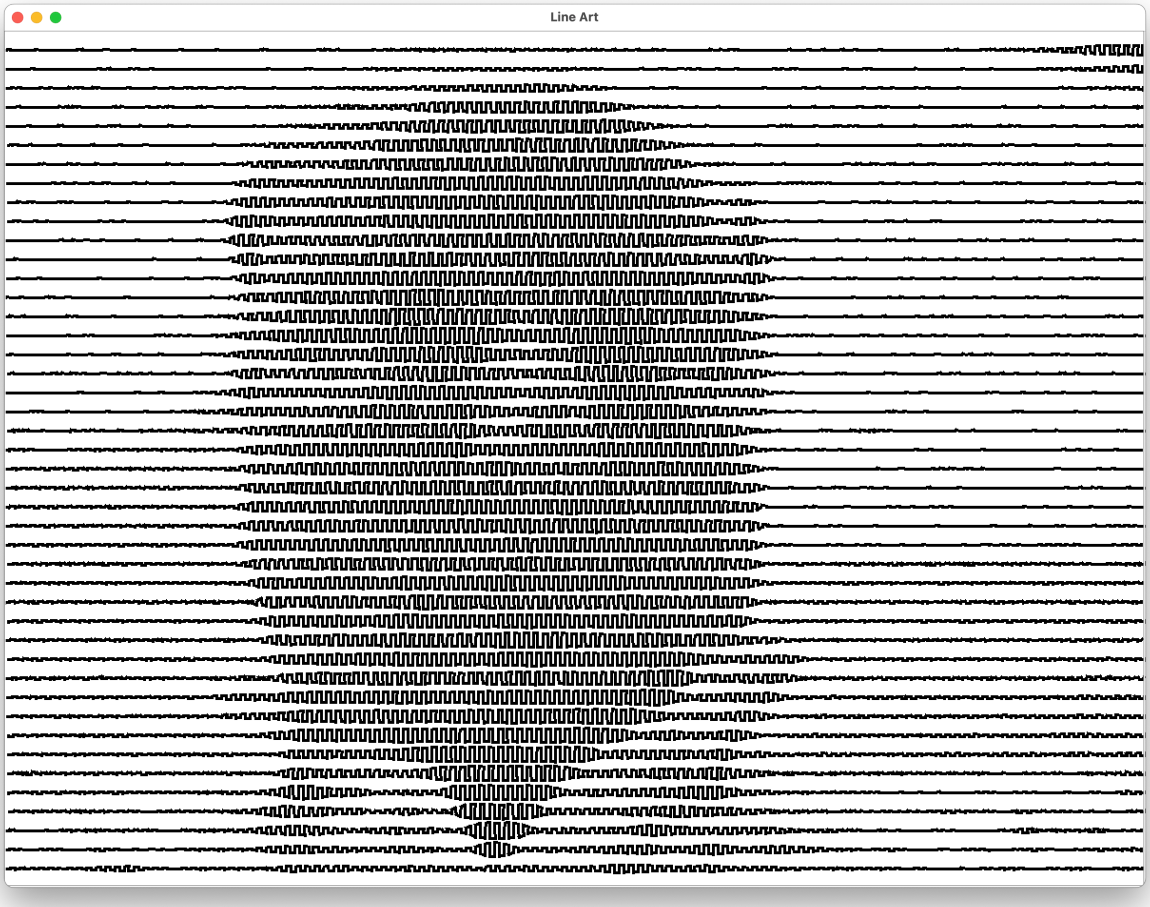
The class `NewtonRaphsonMain` uses your method to plot the Newton-Raphson fractal for the simple example polynomial $z^3 - 1$. According to the iron hand of the Fundamental Theorem of Algebra, this polynomial has exactly three roots, one of which is real and the other two are each other's complex conjugates with nonzero imaginary parts. Running the `main` method of this class should therefore soon enough pop up the result window that looks like the above screenshot. If the colours of your result image end up being mirrored or rotated, remember that the y-coordinates are mirrored inside a `BufferedImage` so that they increase downwards, as is conventional in computer graphics. Therefore, make sure to use the formula with a correct sign to calculate the starting point z_0 for each pixel `(x, y)` from the given top left corner point `(topRe, topIm)` that should be anchored to map to the top left corner pixel `(0, 0)`.

The Wikipedia page and countless other places on the web showcase pictures of Newton-Raphson fractals computed from the roots of more complex (heh) functions than the mere three roots of unity. As also explained on that Wikipedia page, the iterative formula itself can also be generalized to produce wackier results that no longer correspond to actual root finding, but nonetheless prove their worth by being pretty to look at.

Lab 74: Between The Lines

For testing: LineArtMain.java

Op art exploits the tendency of the human visual system to interpolate patterns from simple cues, so that complex three-dimensional shapes emerge out from the interplay of simple two-dimensional shapes, usually rendered in black and white. As seen in the example screenshot, the method written in this lab can convert the given colour pixel image into a bunch of jaggy horizontal curves drawn with black on a white background, so that squinting at this black and white op art picture from a sufficient distance will make the visual sensation of the original image emerge.



In your labs project, create a new class named `LineArt` where you will first be writing two brief utility methods followed by the actual method to perform the above conversion to the pixel image. The first one of these utility methods is used to calculate the **luminance** of the given RGB colour, that is, how bright this colour will appear to the human eye relative to other RGB colours.

```
private static double getLuminance(int rgb)
```

With the RGB colour given as a single int value `rgb` whose three lowest bytes encode the red, green and blue components of that colour, respectively, this method should return a `double` between 0

and 1 that estimates its luminance. You should use **bitwise arithmetic** to extract the individual colour components from the `rgb` integer (you can consult either one of the two example programs `ImageOpsDemo.java` and `FloydSteinberg.java` for the formulas to do this), and from these use one of the possible luminance formulas given in the Stack Overflow page "Formula to determine perceived brightness of RGB color", or some other equivalent online resource. Then, you should write the second required utility method

```
private static double averageLuminance(BufferedImage source, int x, int y, int r)
```

that calculates the average luminance of the pixels inside the square box in the `source` image whose *x*-coordinates start and end at `x-r` and `x+r`, and whose *y*-coordinates start and end at `y-r` and `y+r`. Be careful at the edges of the image where the boundaries of this box would reach outside the boundaries of the `source` image. With this utility function available, the actual conversion of the pixel image into the black and white op art image will be done by the method

```
public static BufferedImage lineArt(BufferedImage source, double density, double frequency, double amplitude)
```

This method should start by creating the `result` image as a `BufferedImage` with the same width and height as the original `source`. Acquire the `Graphics2D` object needed to render things into the image, and fill this image with a white background to act as your canvas. Then set the stroke to be a solid black pen whose width equals `0.15*density`, converted to a `float`.

Each curve starts at the left edge of the image, and corresponds to some *y*-coordinate baseline. The `density` parameter defines how many pixels apart the *y*-coordinate baselines of these individual curves are spaced in the vertical direction. Each individual curve should be rendered with a loop that starts with *x* equal to zero, and terminates once *x* reaches the right edge of the image. Every other round of this loop, the value of *x* should increase by `frequency` for the horizontal curve segments, whereas every other round *x* should stay the same, for the vertical curve segments.

The loop should also maintain another local variable `int sign`, initially set to equal to +1. This `sign` should alternate between the values +1 and -1 every two rounds, flipping its sign in precisely those rounds when the *x*-coordinate stays put. Each round of the loop, draw a line segment from the previous point to the new point whose *x*-coordinate is given by your variable *x*, and whose *y*-coordinate is computed with the formula

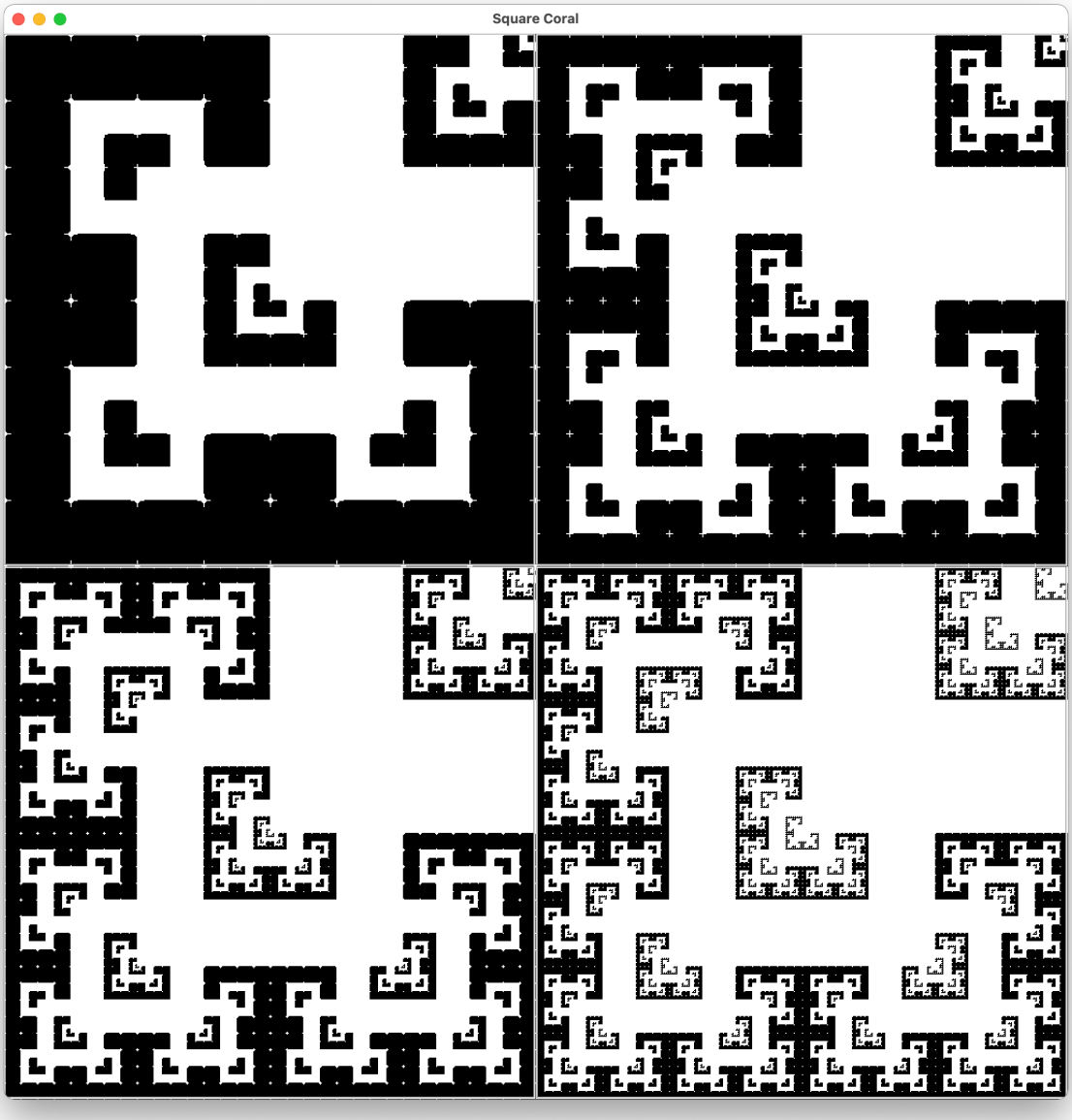
```
y + sign * amplitude * (1 - luminance)
```

where `y` is the baseline coordinate of the curve, and `luminance` is the average luminance computed over the box centered at the pixel (*x*, *y*) with *r* equal half of `density`. Once you have drawn this line segment, the new point becomes the previous point for the next round of this loop. Finally, to give your result some kind of artisanal hand-drawn feel of the quaint past when everything was analog and had a constant component of noise, you should also perturb each individual corner point of your jaggy curves by some small random amount.

Lab 75: Square Coral

For testing: SquareCoralMain.java

The page "Fractals, Chaos and Self-Similarity" by Paul Bourke has always offered plenty of formulas and visualizations of various exotic fractals to explore once all those basic Sierpinski gaskets and Mandelbrot sets start getting boring. Most of the fractals seen on that page emerge from iterating all kinds of inventive formulas on a complex plane, same way as we did in the earlier lab about Newton-Raphson fractals. However, in that collection the page "Square Coral" is a rare example of a **subdivision fractal** where some simple geometric shape is recursively cut into smaller pieces whose areas together comprise the original shape, and some smaller pieces are systematically discarded during this process so that the resulting shape will meander around fractal-shaped holes.



The earlier lab problem of imitating the famous **Mondrian art** also concerned subdividing a rectangle into smaller rectangles. However, in that lab, the cutting points inside the rectangle were chosen randomly and none of the smaller rectangles were thrown away, so the distinction between these pieces was made with black outlines filled in with simple primary colours. Construction of the Square Coral fractal begins with a square that is subdivided into four square quadrants, the side length of each quadrant thus exactly one half of the original square. However, depending on one of the four possible **orientations** of the square currently being subdivided, one of these four quadrants is shrunk to half size, so that its side length is only one quarter of the original, to reveal some background colour from behind the shape.

Each of the four quadrants is also given an orientation that depends on the compass position of that quadrant inside the current square, and the given orientation of the current square. This orientation will affect the recursive subdivision of that particular quadrant. By choosing the orientations of the quadrants according to the rule illustrated with the example on the page linked above, the resulting fractal ends up having the overall coral shape seen in the example screenshot.

In your lab project, create a new class named `SquareCoral`, and there the recursive method

```
public static void render(Graphics2D g2, int depth, int x, int y, int
width, int orientation)
```

This method receives the *x*- and *y*-coordinates of the top left corner of the current square, along with its `width` in pixels. The base case of this recursion is when `depth` equals zero, indicating that that shape is small enough and should not be subdivided any further, but should be rendered as an instance of `RoundRectangle.Double`. (For a more aesthetic appearance, let's give its corners both the horizontal and vertical roundness of `width/5.0`.) Otherwise, depending on the `orientation` of the current square, this method should recursively render the four smaller squares to their rightful positions with `depth-1`. (To simplify these formulas and make them more readable, you might want to first define local variables `h` and `q` to denote the one half and the one quarter of the given `width`, respectively.)

The above screenshot shows the expected visual outcome of performing the Square Coral subdivision up to depths from three to six. To ensure that the result of our SquareCoralMain.java testbed will match this screenshot when executed with your implementation of the recursive `render` method, the `orientation` of 0 should mean that the northeast quadrant contains the quarter-sized square; 1 means southeast; 2 means southwest; and 3 means northwest. Again, make sure to pass on the correct orientation to every one of the four recursive calls for the subdivided quarters. To simplify your coding and debugging, you should note that for each of the four orientations, exactly two of the subdivided quadrants will have that same orientation, and the quadrant that was shrunk to half the size should be one of these two. When the shape has an underlying symmetry, so will the rules that define it, even if it happens to be hiding behind the veil of some abstraction layers.