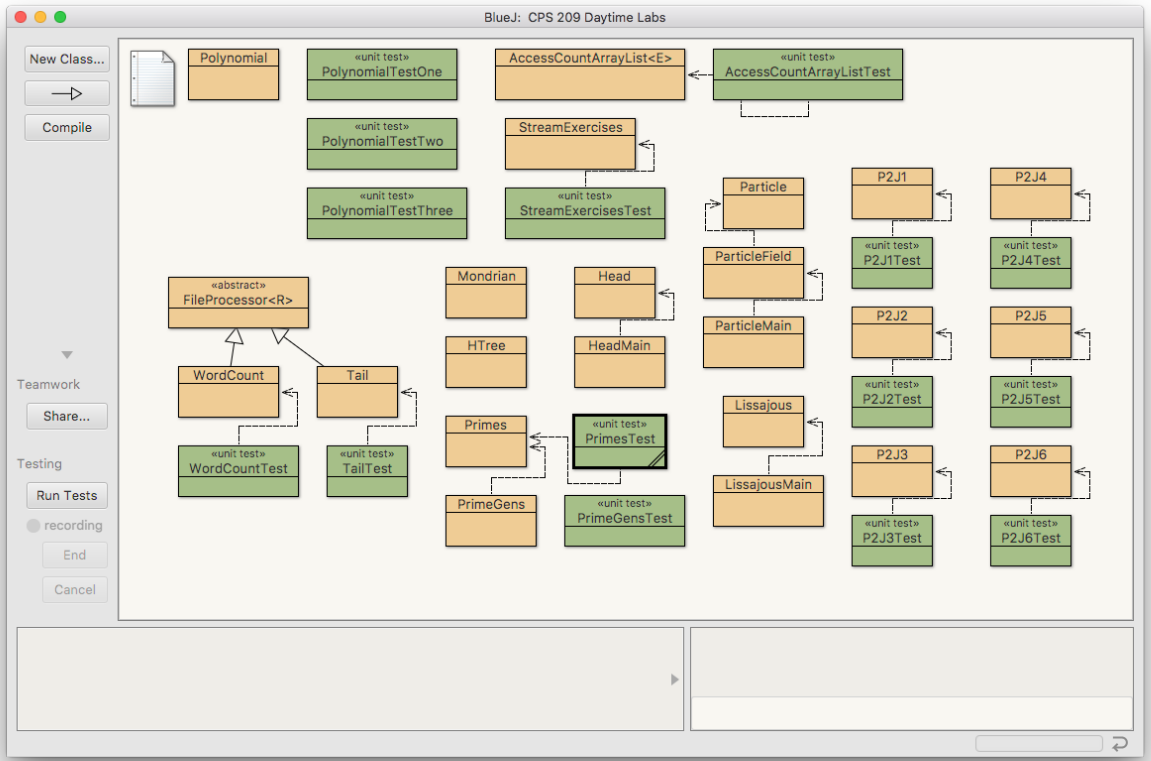
CCPS 209 Labs

This document contains the graded labs for the course **CCPS 209 Computer Science II,** as taught by Ilkka Kokkarinen. This document contains a total of **twenty** lab specifications. Out of these twenty, you can do **up to any ten labs of your choice** to earn the maximum lab mark of 20 points. However, even with all this choice available, **the instructor strongly recommends that you complete at least the first three transition labs 0(A), 0(B), 0(C)** to become proficient in Java so that you learn to think and type in that language without having to use conscious thought on the language itself. This will allow you to see the forest instead of just the individual trees and concentrate on the actual important topics in the remaining labs, instead of getting mired in the nitty gritty language details. Once you have done these three transition labs, you can pick and choose the other seven based on your interest.

You must create one BlueJ project folder named "209 Labs" inside which you write all of these labs. That's right, **all your labs will go in one and the same project folder.** Into this same BlueJ project, you should also manually add the JUnit test classes provided by the instructor for the labs that you choose to implement. The automated testers for these labs and the data file `warandpeace.txt` are available in the instructor's GitHub repository CCPS209Labs.

The BlueJ project that you work on, once it is ready for submission, should look something like the following screenshot taken from the private model solution of the instructor, except that you need to submit only ten labs of your choice instead of all twenty. (Of course the boxes representing the individual classes can be positioned any which way inside your project window.)



Each of these labs is worth the same two points of your total course grade. For those labs that ask you to write a Swing component, these two marks are given for these components passing the visual inspection of the behaviour specified in the lab. The other labs must cleanly pass the entire JUnit tester to receive the two points for that lab. These two points are **all or nothing** for that lab, and **no partial marks whatsoever are given for labs that do not not pass these tests.**

To compensate for this strict policy of correctness, **all the labs have the same deadline of one day after the final exam**. Before that day, there is no real rush to complete any particular lab, so you can take your time to get each lab to work smoothly so that it will pass the JUnit tests. You can then simply come to each lab session to work on and get help for whichever lab you are currently working on, to make the most efficient use of your time.

In all these labs, silence is golden. Since many of these JUnit testers will test your code with a large number of pseudo-randomly generated test cases, your methods should be absolutely silent and print nothing on the console during their execution. **Any lab solution that prints anything at all on the console during the execution will be unconditionally rejected and receive a zero mark.**

You may not modify the JUnit testers provided by the instructor in any way whatsoever, but your classes must pass these tests exactly the way that your instructor wrote them. Modifying a JUnit tester to make it look like that your class passes that test, even though it really does not do so, is considered **serious academic dishonesty** and will be automatically penalized by the forfeiture of all lab marks in the course to begin with. These labs have now been used for one semester and several students, working independently of each other, have successfully implemented their code to pass all these tests.

Once you have completed all the labs that you think you are going to complete before the deadline, you will **submit all your completed labs in one swoop** as the entire "209 Labs" BlueJ project folder that contains all the source files and the provided JUnit testers, compressed into a zip file that you upload into the assignment tab on D2L. Even if you write your labs working on Eclipse or some other IDE, it is compulsory to submit these labs as a single BlueJ project folder. **No other form of submission is acceptable**.

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

Lab 0(A): Arrays and Arithmetic

JUnit: P2J1Test.java

The transition labs from 0(A) to 0(F) help you translate your existing Python knowledge into equivalent Java knowledge in both your brain and fingertips. You may already be familiar with these problems from the same instructor's CCPS 109 course, but even if you are not, these problems are self-contained and can be solved by anybody aspiring to learn Java from either Java or some other language background. Each transition lab consists of up to four `static` methods for you to write, so that there is no object oriented thinking involved in them. All methods of the individual lab must pass the entire JUnit test suite to receive the two points for that lab. There are no partial marks given.

Inside your fresh new BlueJ project, create the first class that **must be named** exactly `P2J1`. Erase the nonsense template that BlueJ fills inside the class body in its misguided effort to "help" you, and in its place, write the following four methods.

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

Python has the integer exponentiation operator ** conveniently built in the language, whereas Java unfortunately does not have that operator. (In both languages, the caret character ^ denotes the **bitwise exclusive or** operation that has nothing to do with integer exponentiation.)

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

This method should compute and return the falling power $n^{\underline{k}}$ where n can be any integer, and k can be any nonnegative integer. (Analogous to ordinary powers, $n^{\underline{0}} = 1$ for any n.) The automated tester is designed so that your method does not need to worry about potential integer overflow as long as you perform computations using `long` type of 64-bit integers.

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

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

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

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

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

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

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

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

Inside an array `a`, an **inversion** is a pair of positions `i` and `j` inside the array that satisfy simultaneously both `i < j` and `a[i] > a[j]`. In combinatorics, the inversion count inside an array is a rough measure how "out of order" that array is. If an array is sorted in ascending order, it has zero inversions, whereas an n-element array sorted in reverse order has $n(n-1)/2$ inversions, the largest number possible. This method should count the inversions inside the given array `arr`, and return that count. (As you should always do when writing methods that operate on arrays, make sure that your method works correctly for arrays of any length, including the important special cases of zero and one.)

Once you have written all four methods, you can download and add the above **JUnit test** class P2J1Test.java to be placed separately inside the same BlueJ project. In the BlueJ project display window, the JUnit test classes show up as green boxes, as opposed to the usual yellow box like the ordinary classes. **The JUnit test class cannot be compiled until your class contains all four methods exactly as they are specified.** If you want to test one method without having to first write also the other three, you can implement the other three methods as one-liners with some placeholder return statement that returns zero or some other convenient dummy value. These methods will of course fail their respective tests, but having them around allows the tester to compile for you to test the one method that you have properly written. Once that is done, move on to replace the placeholder body of the next method with its actual body.

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

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

Lab 0(B): Putting Details Together, Part I

JUnit: P2J2Test.java

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

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

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

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

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

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

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

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

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

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

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

Lab 0(C): Putting Details Together, Part II

JUnit: P2J3Test.java

One last batch of transitional problems taken from the instructor's Python graded labs. Only three methods to write this time, though. The automated tester for these labs uses the `warandpeace.txt` text file, so make sure that this text file has been properly copied into your course labs project folder. This text file does not show up in the BlueJ project screen, even though it is part of the project directory.

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

Rearrange the elements of the given an 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 colours here 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 the array `{ 5, 4, 3, 2, 1 }`, it would become `{ 5, 4, 3, 2, 1 }` since each element by itself is a maximal ascending subarray of length one.

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

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

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

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

A bit convoluted, but does what is needed.

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

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

Furthermore, to make this problem more interesting and the result look prettier, this method **must maintain the capitalization of vowels** based on the vowel character that was originally in the position that each new vowel character is moved into. For example, `"Ilkka Markus"` should become `"Ulkka Markis"` instead of `"ulkka MarkIs"`. Use the handy utility methods in the

`Character` wrapper class to determine whether some particular character is upper- or lowercase, and to convert a character to upper- or lowercase as needed.

Lab 0(D): Lists of Integers

JUnit: P2J4Test.java

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

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

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

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

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

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

Sort the elements of the given list in decreasing order of **frequency**, that is, how many times each element appears in this list. If two elements appear in the parameter list the same number of times, those elements should end up in ascending order of values, the same way as we do in ordinary sorting of list of integers. For example, given a list object that prints out as `[4, 99999, 2, 2, 99999, 4, 4, 4]`, after calling this method that list object would print out as `[4, 4, 4, 4, 2, 2, 99999, 99999]`. Note that the return type of this method is `void`, so this method rearranges the elements of `items` in place instead of creating and returning a separate result list.

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

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

Compute and return the list of prime factors of the **factorial** of `n` (that is, the product of all positive integers up to `n`), with those prime factors sorted in ascending order and with each factor appearing in this list exactly as many times as it would appear in the prime factorization of that factorial. For example, when called with `n = 10`, this method would create and return a list that prints out as `[2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 5, 5, 7]`. (Multiplying all those little prime numbers together would produce the result of 3628800, which equals the product of the first ten positive integers.)

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

Lab 0(E): All Integers Great and Small

JUnit: P2J5Test.java

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

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

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

Fibonacci numbers, that dusty and tired example of silly recursions for teaching introductory computer science, turn out to have more interesting combinatorial properties that make up better problems for us to solve. Relevantly for the current moment, any positive integer `n` can be broken down exactly one way as a sum of distinct Fibonacci numbers that add up to `n`, under the additional 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.)

The unique breakdown into Fibonacci numbers can be constructed with a **greedy algorithm** that simply finds the largest Fibonacci number `f` that is less than or equal to `n`. Add this `f` to the result list, and break down the rest of the number given by the expression `n-f` the same way. This method should create and return some kind of `List<BigInteger>` that contains the selected Fibonacci numbers in **descending order**. For example, when called with `n = 1000000`, this method would return a list that prints as `[832040, 121393, 46368, 144, 55]`.

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

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

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

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

Since seven is a famously lucky number in the Western culture, whereas zero is what nobody wants to be, let us look at those positive integers whose digits consist of some sequence of sevens, followed by some (possibly empty) sequence of zeros. For example, 0, 7, 77777, 7700000, 77777700, or 700000000000000000000000000000000. Note that all sevens must be in one consecutive bunch, followed by all zeros in another consecutive bunch.

One of the examples of combinatorial thinking given in the excellent MIT online textbook "*Mathematics for Computer Science*" (PDF link to the updated 2018 version, for anybody who is interested in that sort of stuff) points out that for any positive integer `n`, there exists at least one positive integer thus constrained to sevens and zeroes that is divisible by `n`. This integer made of sevens and zeroes can get pretty humongous as `n` grows larger, but at least one such number must always exist for any integer `n`.

This method should find and return the **smallest** such integer that is divisible by the given `n`. The easiest way to do this would be to use two nested loops. The outer `while`-loop iterates through all possible lengths (that is, how many digits the number contains) of the number, and for each length, the inner `for`-loop iterates through all legal sequences of sevens followed by zeros of that total length. Keep going up until you find such number that is exactly divisible by `n`.

This process will always eventually terminate for any `n`, since such a number is always guaranteed to exist, although these numbers can easily become gargantuan. For example, for the argument value `n = 12345`, the resulting number consists of 822 copies of the digit seven followed by a single zero digit. To speed up the search, you can utilize an additional theorem given in the same book, that says that unless `n` is divisible by either 2 or 5, the sequence of sevens and zeros that is divisible by `n` is guaranteed to contain only sevens but no zeros. This ought to speed up your search by at least an order of magnitude for such easy values of `n`, since the quadratic number of possibilities is pruned down into a single linear branch 7, 77, 777, 7777, ...

Lab 0(F): Two Branching Recursions

JUnit: P2J6Test.java

In typical computer science education, teaching of recursion tends to "nerf down" this mighty sword by using it only as a toy to simulate some linear loop to compute factorials or Fibonacci numbers, thus gaining nothing over using a for-loop the way any reasonable person would have automatically done there anyway. This tends to leave the students understandably confused about the general usefulness of recursion, since from their point of view, recursion is only ever used to solve toy problems in a needlessly convoluted manner.

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

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

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

Many integers can be broken down into a sum of cubes in several different ways. This method must always return the breakdown that is **lexicographically highest**, that is, starts with the largest possible working value for the first element, and then follow the same principle for the remaining elements that break down the rest of the number into a list of distinct cubes. For example, when called with `n = 1072`, this method must return the list `[10, 4, 2]` instead of `[9, 7]`, even though $9^3 + 7^3 = 1072$.

Hint: the easiest way to implement this recursion is again to have a second private method

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

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

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

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

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

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

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

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

that receives two additional arguments `soFar` and `result`, where `soFar` is the partial string generated so far (this should initially equal the empty string at the top level call) and `result` is the list of strings that the recursion has already discovered. The base case for failure is when the string `soFar` ends with one of the strings in `tabu` list. The base case for success is when `soFar.length() == n`, in which case the string soFar is added to `result`. Otherwise, the recursion should loop through the characters in `alphabet`, appending each one to soFar for the recursive call.

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

Lab 1: Polynomial Data Type: Basics

JUnit: PolynomialTestOne.java

After learning the basics of Java language, we may proceed to designing our own data types as **classes**, and writing the operations on these data types as **methods** that the outside users of our data type can then call to get the job done.

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

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 cannot change their internal state after they have been constructed. Immutability has many advantages in programming, even though those advantages might not be fully evident yet. 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 testers, 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 define the polynomial. For a polynomial of degree *n*, the array `coefficients` 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 as argument a coefficient array whose highest terms are zeroes, it should simply ignore those 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}` without those pesky higher order zeros.

To guarantee that the `Polynomial` class is immutable so that no outside code can ever change the internal state of an object after its construction (well, at least not without resorting to underhanded Java tricks such as *reflection*), the constructor should not assign only 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.

```
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. Constant polynomials have a degree of zero.

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

Returns the coefficient for the term of order *k*. For example, the term of order 3 of the previous polynomial equals 5, and the term of order 0 equals 42. This method should work correctly even when *k* is negative or greater than the actual degree of the polynomial, and simply return zero in such cases of nonexistent terms. (This policy will also make some methods of Lab 2 much easier to implement.)

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

Evaluates the polynomial using the value `x` for the unknown symbolic variable of the polynomial. For example, when called with `x = 2` for the previous example polynomial, this method would return `68`. To keep this simple, your method does not have to worry about potential *integer overflows*, but can assume that the final and intermediate results of this computation will always stay within the range of the primitive data type `long`.

Lab 2: Polynomial Data Type: Arithmetic

JUnit: PolynomialTestTwo.java

The second lab continues with the `Polynomial` class from the first lab by adding new methods for *polynomial arithmetic* to its source code. (There is no inheritance or polymorphism taking place yet in this lab.) Since the class `Polynomial` is designed to be immutable, none of the following methods should modify the objects `this` or `other` in any way, but return the result of that arithmetic operation as a brand 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`. This method should not modify `this` or `other` polynomial in any way. Make sure that just like with the constructor, the coefficient of the highest term of the result is nonzero, so that adding the two polynomials $5x^{10} - x^2 + 3x$ and $-5x^{10} + 7$, each having a degree 10, produces the result $-x^2 + 3x + 7$ that has a degree of only 2 instead of 10.

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

Creates and returns a new `Polynomial` object that represents the result of *polynomial multiplication* of two polynomials `this` and `other`. Polynomial multiplication works by multiplying all possible pairs of terms between the two polynomials and adding them together, combining terms of equal degree together into a single term.

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 in an internal data field `int count` of how many times the methods `get` and `set` have been called. (The same `int` 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 current 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 Data Type: Comparisons

JUnit: PolynomialTestThree.java

In this fourth lab, we continue 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 the class definition so that this class `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` polynomial are pairwise equal. If the `other` object is anything else, this method should return `false`.

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

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

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

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

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

`public int compareTo(Polynomial other)`

Implements the *ordering comparison* between `this` and `other` polynomial, 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 different possible rules. Here 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**. Otherwise the automated tester will reject your code, even if your code happened to define some other perfectly legal ordering relation from the infinitely many possible ordering relations!

Lab 5: Introduction to Swing

Historically, **graphical user interface (GUI) programming** was the first "killer app" of object oriented programming that propelled the entire paradigm into the mainstream after two decades of purely theoretical academic research. (Most things in computer science are at least two decades older than what even most working programmers would casually assume them to be.) Inheritance and polymorphism make writing GUI's 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 nicely illustrating both Greenspun's Tenth Rule and Blub Paradox in practice.

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

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

`public Head()`

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

`@Override public void paintComponent(Graphics g)`

Renders some kind of image of a simple human head on the surface of this component. This is not an art class so this head does not need to look fancy, but a couple of simple rectangles and ellipses suffice. (Of course, interested students of more artistic bent might want to check out more complicated shapes from the package `java.awt.geom`, or perhaps `Image` objects read from some GIF or JPEG files, to generate a prettier image.)

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

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

Lab 6: Processing Text Files, Part I

JUnit: WordCountTest.java

This lab lets you try out reading in text data from an instance of `BufferedReader` and performing computations on that data. To practice working inside a small but entirely proper object-oriented *framework*, we design an entire class hierarchy that reads in text one line at the time and performs some operation for each line, and having read in all the lines, emits a result in the end. The subclasses can then decide what exactly that operation is by overriding the *template methods* used by 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();
```

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

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

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

This abstract superclass defines a template for a class that performs some computation for a text file that is processed one line at the time, returning a result whose type `R` can be freely chosen by the concrete subclasses of this class. Different subclasses can now 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 file contains. Those of you who have taken the Unix and C programming course CCPS 393 should already recognize this handy little text processing tool, but even if you have not yet taken that course, you can still implement this tool based on the following specification of its behaviour.

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

`protected void startFile()`

Initializes the character, word and line counts to zero.

`protected void processLine(String line)`

Increments the character, word and line counts appropriately. In the given `line`, every character increments the count by one, regardless of whether that character is a whitespace character. To properly count the words in the given line, count the non-whitespace characters for which the previous character on that `line` is a whitespace character. (You can imagine that the first character is preceded by an invisible whitespace character.) You must 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.

`protected List<Integer> endFile()`

Creates and returns a new `List<Integer>` instance (you can 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

This seventh lab lets you create a Swing component that displays a real-time animation using Java concurrency to execute an *animation thread* that runs at constant pace independent of what the human user happens to be doing with that component. This thread will animate a classic *particle field* where a swarm of thousands of independent little *particles* buzzes randomly around the component.

(This same basic architecture could then, with surprisingly little additional modification, be used to implement some real-time game, with this animation thread moving the game entities according to the rules of the game 50 frames per second, and the event listeners giving orders to the game entity that happens to represent the human player. As it seems to be 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!)

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

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

The class should then have the following methods:

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

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

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

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

```
public void move()
```

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

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

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

The class should have the following `public` methods:

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

The constructor that first sets the preferred size of this component to be `width`-by-`height`. Next, it creates `n` separate instances of `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. The body of this loop should first `sleep` for 20 milliseconds. After waking up from its sleep, it should loop through all the `particles` and call the method `move()` for each of them. Then call `repaint()` and go to the next round of the while-loop.

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

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

```
public void terminate()
```

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

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

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

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

Lab 8: Processing Text Files, Part II

JUnit: TailTest.java

In this lab, we return to the `FileProcessor<R>` framework from Lab 6 for writing tools that process text files one line at the time. This time this framework will be used to implement another

Unix command line tool `tail` that extracts the last `n` lines of its input and discards the rest. Write a class `Tail` that extends `FileProcessor<List<String>>` and has the following methods:

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

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

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

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

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

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

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

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

Lab 9: Swinging Some Curves

In this lab you write a Swing component that displays a Lissajous curve on its surface, allowing the user can control the parameters *a*, *b* and *delta* that define the shape of this curve by entering their values into three `JTextField` components placed inside this component. Write a class `Lissajous extends JPanel`, and the following methods in it:

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

The constructor that sets the preferred size of this component to be `size`-by-`size` pixels. Then, three instances of `JTextField` are created and added inside `this` component. Initialize these text fields with values 2, 3 and 0.5, with some extra spaces added to these strings so that these text fields have a decent size for the user to enter other numbers. Add an [ActionListener]() to each of the three text fields whose method `actionPerformed` simply calls `repaint` for this component.

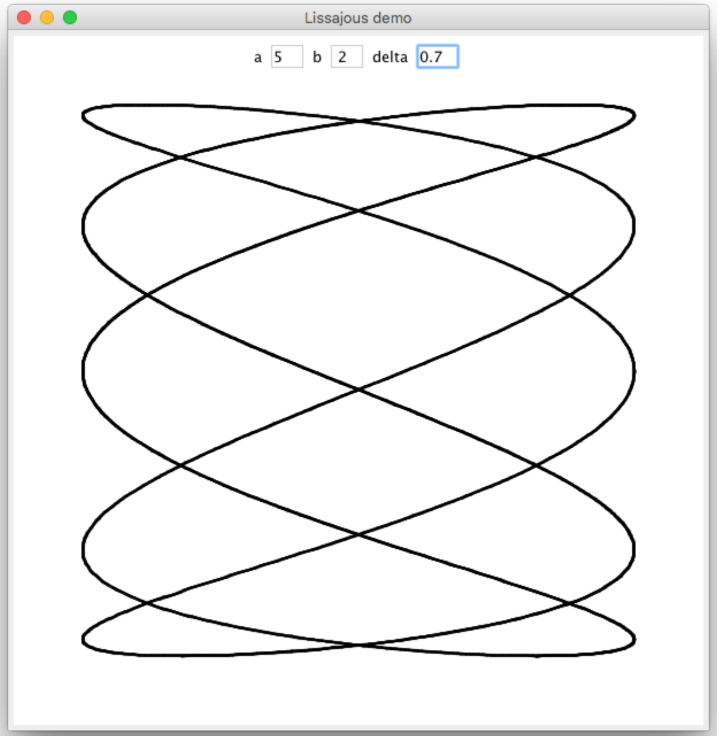
```
@Override public void paintComponent(Graphics g)
```

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

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

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

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



Motivated students can take on as an extra challenge to make the displayed image look smoother by eliminating some visual jagginess, and possibly even make this rendering look more artistic in other ways. (For example, instead of subdividing the curve into linear line segments, you could rather subdivide it into much smoother cubic curves that connect seamlessly at the point and the direction vector that the previous curve left off...)

Lab 10: Computation Streams

JUnit: StreamExercisesTest.java

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

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

```
public 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 `filter` to keep only those whose `length` is greater or equal to the threshold value `thres`, and in the end, return a count how many such lines the file contains.

```
public 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 using the separator regex `"[^a-z]+"` for the method `split` in `String`. In the next stages of the computation stream, discard all empty words, and `sort` the remaining words in alphabetical order. Multiple consecutive occurrences of the same word should then be removed (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 only if it was the first one or distinct from the previous argument), and to wrap up this computation stream, `collect`ed into a `List<String>` that is returned as the final answer.

Lab 11: Prime Numbers and Factorization

JUnit: PrimesTest.java

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

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

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

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

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

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

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

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

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

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

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

To qualify for the two lab marks, the automated tester must successfully **finish all three tests within one minute** when run concurrently on instructor's Mac desktop bought in 2012. Speed is of the essence here. (Currently, the instructor's model solution takes about 7 seconds to complete the automated tester in that environment.)

Lab 12: Interesting Prime Number Sequences

JUnit: PrimeGensTest.java

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

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

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

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

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

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

`public static class TwinPrimes implements Iterator<Integer>`

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

`public static class SafePrimes implements Iterator<Integer>`

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

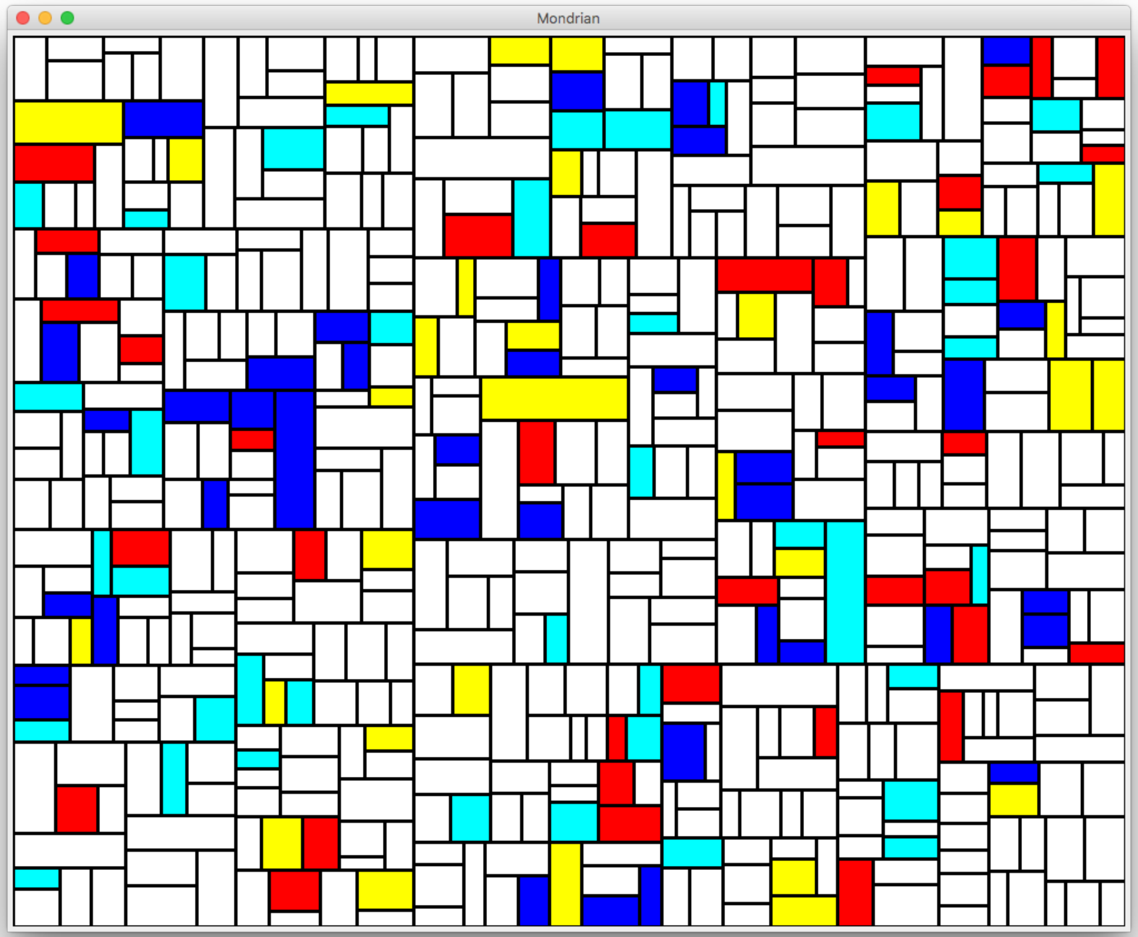
`public static class StrongPrimes implements Iterator<Integer>`

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

Lab 13: Recursive Mondrian Art

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

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 following image, 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)
```

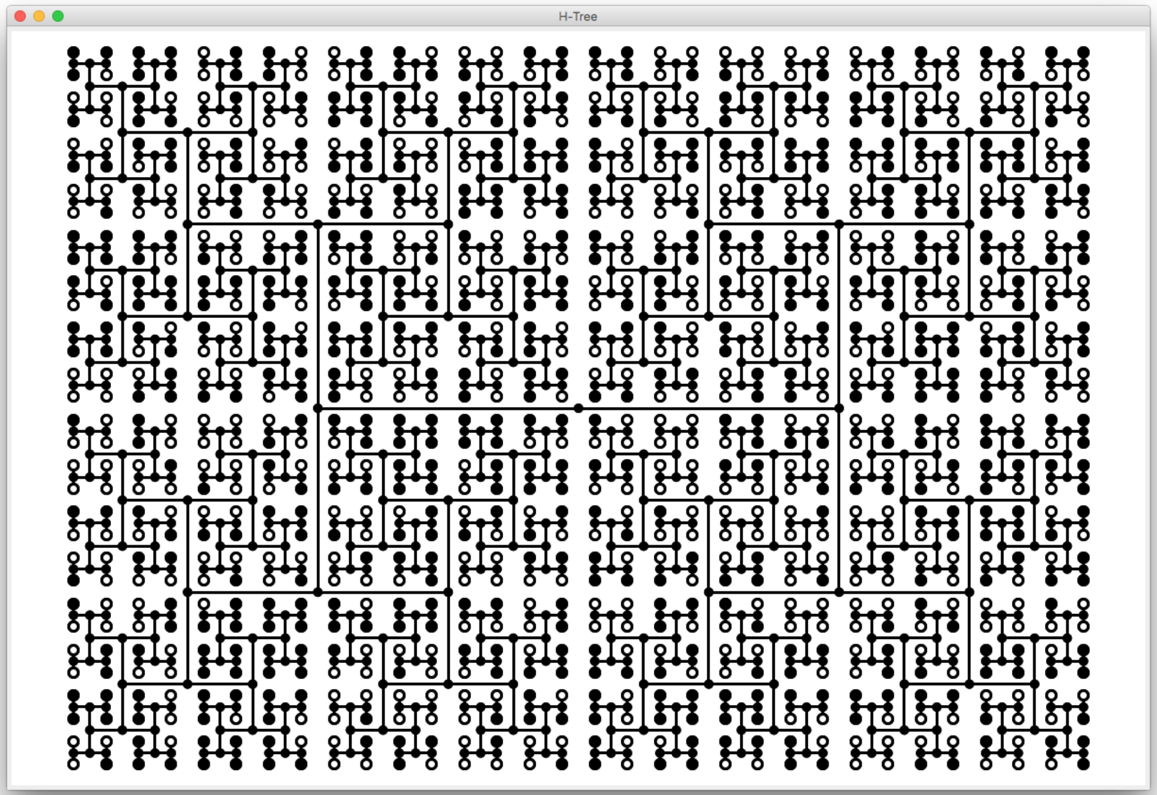
Recursively subdivides the given rectangle whose top left corner is in coordinates `(tx, ty)` and whose width and height are `(w, h)`. The base case of the recursion is when either `w` or `h` is less than `CUTOFF`, in which case this rectangle is drawn on the `Graphics2D` object provided. (The top-level 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 tend to be perceived as ugly and disharmonious, the same way as when somebody writes cursive by hand and has to squeeze the words to be more narrow at the end of the line to fit within the margins. In all walks of life, that precarious sweet spot of balance between order and randomness tends to be perceived as most aesthetic.)

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.

Lab 14: H-Tree Fractal

In the spirit of the recursive Mondrian subdivision of the previous problem, here is a similar (but different enough) problem of rendering the H Tree fractal on the surface of a Swing component, with some additional decorative little dots that indicate the intersections and the caps at the end pieces of this exponentially branching structure. An example run of the instructor's private model solution produced the following result that your outcome should also resemble:



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

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

This class should then have the following methods.

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

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

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

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

The base case of the recursion is when `len < CUTOFF`, in which case this method should `fill` a disk of radius `R` centered at coordinates `(x, y)`, flipping a random coin to decide whether this disk should be black or white. (If it is white, you should still `draw` the black outline around it.)

Otherwise, draw a black disk to the current coordinates `(x, y)`, and then calculate the endpoint `(nx, ny)` of the line segment that this fractal is currently heading. Draw the line segment from `(x, y)` to `(nx, ny)` on the image, and follow with two recursive calls at `(nx, ny)` using the directions `i + 1` and `i - 1`, making sure that these indices stay within the `DIRS` array. The new value of `len` in the recursive call should equal the current `len` divided by the square root of two, so that the branches become shorter down the line and now matter how deep you recourse, the fractal shape stays within the finite boundary.

(Note how, due to the magic of subdivision fractals in general, these exponentially spreading branches never touch or overlap each other, no matter how deep you continue this recursion. In the limit of `CUTOFF` equal to zero, as the recursion depth increases towards infinity, this **space-filling fractal** fills its entire two-dimensional bounding box perfectly not leaving any holes in which you could fit any circle with any arbitrarily small positive radius ε of your choice, despite consisting only of one-dimensional line segments and not even drawing the decorative disks on the intersections! Should some students catch a case of "fractal fever" from this exercise, they might want to later dive into "Brainfilling Curves: A Fractal Bestiary", your instructor's personal favourite that teaches you to render an infinite variety of such fractals with **turtle graphics**.)

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

To get to admire your fractal artwork, once again create a `main` method to open a new `JFrame` instance, inside which you add a new `HTree` instance of dimensions (1000, 800).

Again, same as with the Mondrian subdivision, once you get this program to work, feel free to experiment to make the result look more aesthetic and artistic, which is perfectly fine with the instructor as long as you maintain the overall shape and spirit of the original H-Tree fractal. (One fun variation makes the direction vectors to be the four diagonal directions instead of being axis-aligned. Also, 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 some small random amount, as if somebody were drawing this shape by hand...)