CCPS 109 Weekly Labs

This document contains the specifications for the weekly labs of the course **CCPS 109 Computer Science I**, as taught by Ilkka Kokkarinen.

General instructions (READ THESE BEFORE DOING ANY LABS!)

What you must do: For every lab, **the instructor provides a JUnit test suite that you must use to verify that your code works correctly**. Create and copy that test class in your BlueJ project alongside the class that the lab specification asks you to write. Once you have written all required methods in your class, or at least written do-nothing stubs for them, you can compile the test class. BlueJ displays JUnit test classes in green instead of the usual yellow. The popup menu of the test class allows you to run either all tests at once, or only one individual test.

The test classes do not have syntax errors, so please don't waste our time by emailing me to tell me that my tester does not compile. **If the test class does not compile for you, it means that either some method in your class is missing or misspelled, or that you have copy-pasted the test class incompletely.** To allow the test class to compile and run, you must write every class name, method name, and the parameter list of every method **exactly** as it is specified in that lab question!

You are not allowed to modify the instructor-provided JUnit test class in any way! Especially any modification that intentionally attempts to make your non-working methods seem like they pass my tests is treated as blatant **academic dishonesty** and will automatically, with a zero tolerance policy, be punished by **forfeiture of all lab marks from this course**. Please do not attempt to trick me or the JUnit test class, not even as a prank.

Silence is golden! In every lab, make sure that **your methods are quiet** so that **they don't do any `System.out.println` console output when submitted**, but just quietly `return` the asked result. During the development, you can use console output for some "quick-and-dirty" debugging (although in many cases, it would be much better to use BlueJ **breakpoints** and stepping through the code with debugger for this purpose), but once you get your method working, **take out all those console output statements, or at least comment them out.** Otherwise, running the JUnit test might take a very long time to complete, since textual output to the graphical screen will definitely become very sluggish when millions of lines get printed.

What you may assume: In all problems, the **arguments passed to each method are always legal as defined in the problem specification**. **Your methods never need to be able to recover from any illegal argument values.** (Of course, in real world programming, it is good to make your methods **robust** so that they don't crash when given illegal inputs. However, proper error handling using exceptions will be taught in CCPS 209, so I cannot require you to do that in this course.)

After writing each method: Once you have written a method, give it a quick "dry run" by interactively creating an object in BlueJ and calling that method with some arguments of your choice so that they are simple enough so that you know what the correct result should be, but also not too simple, so that the method actually has to do some non-trivial work to produce the correct answer. If the method returns an incorrect answer, try to reason what is wrong with the logic of the method. Once you have tried out a sufficient number of simple test cases, it is time to subject your method to my automated JUnit tests.

Of course, you would certainly want to test some of your methods before writing all of them. Instead of modifying the JUnit test, which was forbidden, you can always implement every method initially as an **empty stub** that does nothing but only returns zero, or whatever is the simple default answer suitable for the method's return type. Of course this stub has no chance of passing the JUnit test, but **having the required method stubs in your class allows the test class to compile** so that you can use it to test your methods one by one as your write them. This is much easier than having to first write all methods to completion and then trying to test and debug them all at once. After you get the current method to work, move on to fill in the method body of the next method stub.

Lab 1: Objects and methods

JUnit: TestRectangle.java

In the first lab, we practice creating classes and objects and representing data and their operations in them, and writing methods that work on data given to them, similarly to the example class `Length`.

Write a class *Rectangle* that has two *private int* fields *width* and *height*, and a public constructor *Rectangle(int w, int h)* that assigns its arguments to these fields. Then, write the *public* accessor ("getter") methods *int getWidth()* and *int getHeight()* to return the values of these two fields, and public accessor methods *int getArea()* and *int getPerimeter()* that compute and return the area and perimeter length of the rectangle. Last, write a mutator method *void flip()* that internally swaps the values of the *width* and the *height* of the rectangle with each other. (For example, flipping a 2*5 rectangle would turn it into 5*2 rectangle, and flipping that would make it a 2*5 rectangle again.)

Try out your class in BlueJ by interactively creating an object from it (right click the class and select the item `new Rectangle`). Play around with your object by calling its methods and see if they return the correct answers. When you are satisfied with this, copy the JUnit test class TestRectangle.java into the same BlueJ project, compile it (since it is a JUnit test, the compiled class will show up as a green box instead of yellow) and run either one test or all tests from its right click popup menu. If some test does not give you the green check mark, debug your code.

After this question (each test is worth half point, as always), after you get a better idea how classes and objects work, register to CodingBat and start doing some warm-up problems to see how CodingBat works. These problems are not graded in the lab, and the warm-up problems will not become part of the 50 problems that you have to solve in CodingBat to get to enter the final exam. **Note that in CodingBat, you always write only one method whose signature and braces are already given to you. CodingBat then writes the rest of the class around your method for its internal compilation and testing purposes.**

Lab 2: Conditions

JUnit: TestCardProblems.java

IMPORTANT: THIS WEEK'S LAB CLASS `CardProblems.java` DOES NOT NEED AND SHOULD NOT HAVE ANY DATA FIELDS, A MAIN METHOD, OR A CONSTRUCTOR. WRITE ONLY THE FOUR METHODS THAT ARE ASKED, AND NOTHING ELSE! THE SAME INSTRUCTION IS TRUE FOR ALL REMAINING LABS IN THIS COURSE.

In this lab, we practice writing if-else conditions with a couple of methods inspired by familiar card games. An individual playing card is represented as a string of two characters so that the first character is the rank (from `"23456789TJQKA"`, and note that ten is encoded as letter `T` to make all ranks to be single characters) and the second character is the suit (from `"cdhs"`). For example, `"Jd"` would be the jack of diamonds, and `"4s"` would be the four of spades. A hand made up of multiple cards is given as string encoding all those cards consecutively, for example `"Kh3h7s8h2h"` for a five-card hand that happens to have one spade and four hearts. **Note that the cards can be listed inside the string in any order, not necessarily sorted by suit or rank.** The suits and ranks are also **case sensitive**, with the rank always given as a digit or an uppercase letter, and the suit always given as a lowercase letter from the four possible letters *cdhs*.

Important: when calling a method that takes a String argument interactively in BlueJ, you need to enter the argument string in double quotes in the BlueJ dialog! This is because the BlueJ method call dialog box expects you to give it a Java expression that it evaluates and passes the result to the method as an argument.

Write the following methods in the class named *CardProblems*. Remember that the *String* class has a handy method *charAt(int idx)* that returns the character in the position *idx* of that string.

1. The card ranks inside the hand are given as characters inside the string, but sometimes it would be more convenient to handle these ranks as numerical values from 2 to 14. The **spot cards** from two to ten have their numerical values, and the **face cards** jack, queen, king and ace have the values of 11, 12, 13 and 14, respectively. Write a method *int getRank(char c)* that returns the numerical value of the card given as the character parameter *c*. For example, when called with argument `'Q'`, this method would return 12. It is easiest to write this method as an if-else ladder, but as always in programming, many other ways are possible.

2. The complete hand classifier for **poker** would be far too complicated for us to write in this course, so let's just look at a couple of instructive special cases. First, write a method *boolean hasFlush(String hand)* that checks whether the five-card poker hand given as 10-character string is a **flush**, that is, all five cards are of the same suit. Ranks don't matter for this problem, since for simplicity, we don't care about the rare possibility of straight flushes. (Hint: The suits are given in

the positions 1, 3, 5, 7 and 9 of the parameter string.)

3. A bit trickier question is to determine whether the given five-card poker hand contains **four of a kind**, that is, four cards of the same rank. Write a method *boolean hasFourOfAKind(String hand)* to determine this. This time the suits don't matter, and the ranks are given in positions 0, 2, 4, 6 and 8 of the parameter string. There are many possible ways to organize the logic of this method. However, you have to look at all five possible ways to choose four cards from a hand of five cards. (Actually, you are choosing the one card to leave out in five different ways, and look at the remaining four to check if they are the same rank.) So this method will need plenty more equality comparisons than the previous method.

4. In the exotic draw poker variation of **badugi**, the players try to draw a four-card hand in which no two cards have the same suit or rank. Write a method *boolean hasFourCardBadugi(String hand)* that checks whether the given four-card hand is a badugi, that is, **all four ranks are different, and all four suits are different**. For example, given the hand `"2d7cKhJs"` this method would return *true*, but given the hands `"4d4cKhJs"` (that has two fours) or `"2d7cKhJd"` (that has two diamonds), the method would return *false*. (Your method needs to make **exactly 12 equality comparisons** in total, when you compare the rank and suit of each card to the rank and the suit of every other card that follows it.)

Lab 3: Loops

JUnit: TestMoreCardProblems.java

We continue with problems inspired by various classic and popular card games, this time demonstrating the use of loops and repetition to solve problems. Create a class named *MoreCardProblems* and write the following methods there.

1. In the game of **contract bridge**, each player is dealt a hand with thirteen cards. The basic "Milton Work point count" for hand evaluation, about the first thing that any beginning bridge player learns, estimates the strength of the hand for bidding purposes the following way: each ace is worth 4 points, each king is worth 3 points, each queen is worth 2 points, and each jack is worth 1 point. (Simple but surprisingly effective.) Write a method *public int bridgePointCount(String hand)* that, given a bridge hand as a string of 26 characters, computes and returns this point count.

2. In the game of **gin rummy**, once the hand is over after **knocking and melding**, each player scores "deadwood" penalty points for the cards that he could not get rid of. Same as in blackjack, aces count for one point, all face cards count for ten points, and the spot cards count according to their face value. Write a method *public int countDeadwood(String hand)* that computes the deadwood points in the given hand. The hand can now have *any* number of cards, so **you must use a for-loop** to go through the individual cards. (Note that "any number" here includes zero! Your method should correctly return zero deadwood points for an empty hand.)

3. Back to contract bridge. The shape of the hand (that is, how "balanced" or "unbalanced" it is) is often just as important for the hand evaluation purposes as its raw point count. Write a method *public String bridgeHandShape(String hand)* that creates and returns a four-element string that tells how many **spades, hearts, diamonds and clubs, in this exact order**, there are in the hand. This result string should list these four integer counts separated by commas, with exactly one space after each comma. There should be no trailing comma or space after the number of clubs. For example, when called with the argument "`Kh6c2cJdJcAd7h3d8dQdTsTh3h`", the method would create and return the string "`1, 4, 5, 3`" since this hand contains one spade, four hearts, five diamonds and three clubs.

4. In poker, whether an unsorted hand is **one pair, two pair, three of a kind, full house or four of a kind** can be determined surprisingly easily with some clever combinatorics by looping through all 4 + 3 + 2 + 1 = 10 possible ways to choose two cards from that hand, and counting how many of these two cards are pairs of equal rank. This count will always be 1 for a poker hand that contains one pair, 2 for two pair, 3 for three of a kind, 4 for full house and 6 for four-of-a-kind. (Try this out with some made up five card hands. You will also notice why is impossible for the internal pair count to equal 5 for a five card hand. Notice how shuffling the cards inside the hand does not affect this count.)

Write a method *public int countInternalPairs(String hand)* that, given a poker hand guaranteed to consist of exactly five cards, uses **two nested for-loops** with indices *i* and *j* to loop through all possible ways to choose two cards in position *i* and position *j* so that $i < j$. Count how many pairs of cards of equal ranks you find from the and, and return that count as the answer.

Lab 4: Using conditions and loop with Strings

JUnit: TestStringProblems.java, which also needs the data file *warandpeace.txt*. Do **NOT** copy-paste the contents of this file anywhere onto the BlueJ GUI. Also, do not open this file inside a text editor in between, since adding, removing or changing even one character in this file would cause the checksums of the JUnit test to fail.

In this lab, we shall take a closer look at the *String* class and its operations, and use text strings to practice writing logic with loops. Create a class *StringProblems* and write the following methods in it. In all the methods that ask you to create a result string by repeatedly adding individual characters to the result, for efficiency reasons you should use the mutable *StringBuilder* designed for this very purpose, and then convert the final result to a *String*. If you only use the bare *String* to which you add the characters one by one, your method will be inherently inefficient and the JUnit test methods will take quite a long time to complete.

1. *String removeDuplicates(String s)* that creates and returns a new string with all the consecutive occurrences of the same character turned into a single character. For example, when called with the string `"aabbbbbabbbbccccddd"`, this method would create and return the string `"ababcd"`. (Hint: loop through the characters of the string, and add each character to the result only if it is different from the previous character. Be careful at the beginning of the string.)

2. *int countWords(String s)* that counts how many words there are in the parameter string *s*. Each word starts at a non-whitespace character that either begins the entire string, or is immediately after some whitespace character. For example, the string "   **H**ɞllo **w**orld! **H**☺w   **a**re **y**ôu **t**oday?" contains six words, the boldface here indicating which characters begin a new word.

   (You **must** use the static utility method `Character.isWhitespace` in the utility class `Character` to identify the whitespace characters. There are way more possible whitespace characters than just the common one that you get by pressing the spacebar, and especially non-whitespace characters there are tens of thousands of, and in today's interconnected and globalized world you simply may not assume that only the 26 letters of English and the basic punctuation characters exist in text data!)

3. *String convertToTitleCase(String s)* that creates and returns a new string where the first letter of each word is converted to title case, using the same definition of a "word" as in the previous problem. All other characters are kept as they were. For example, when called with the string "The quick brown fox jumped ovɛr a lazy dog!", this method would return the string "The Quick Brown Fox Jumped Ovɛr A Lazy Dog!"

   Same way as in the previous question, you must use the static utility method `Character.toTitleCase` of `Character` to do this character conversion. Note that *title case* is not the same thing as *upper case*! (Explanation of the difference is available in the Unicode FAQ, for anybody interested in this kind of stuff.)

4. *String uniqueCharacters(String text)* that creates and returns a *String* that contains each character that occurs in the given *text*, so that each character is included in this result only once. These characters must occur in the result in the exact same order as they first occur in the original string. For example, when called with argument `"Hello there, world!"`, this method would return `"Helo thr,wd!"` (Again for efficiency, you should use a *StringBuilder* instance inside the method to build up the result.)

Lab 5: Arrays give loops something to do

JUnit: TestArrayProblems.java

This week, it is time to practice arrays so that our methods can process millions of items of data, giving our loops and conditions something useful to do. Create a class *ArrayProblems* and write the following array methods in it.

1. *int[] everyOther(int[] a)* that creates and returns an array that contains the elements from the **even-numbered positions** of the array *a*, that is, the elements *a[0], a[2], a[4],...* Make sure to compute the length of the result array exactly right so that there is no extra element in the end. (You need to consider the possible odd and even lengths of *a* separately. The integer arithmetic remainder operator `%` might come handy here.)

2. *int countInversions(int[] a)* that counts how many **inversions** there exist inside the array *a*. An inversion is a pair of indices (i, j) to the array so that $i < j$ and $a[i] > a[j]$. Use two nested *for*-loops to loop through all possible pairs of array indices *i* and *j*, with the inner loop variable *j* starting from $i + 1$.

   (In the analysis of sorting algorithms, the inversion count is an important measure of how "out of order" the elements of an array are compared to a fully sorted array that has zero inversions. There exists a whole bunch of **combinatorics** stuff about that.)

3. *void squeezeLeft(int[] a)* that **modifies the contents of its parameter array by** moving all its nonzero elements to the left as far as it can, writing over the zero elements that precede them. For example, if called with the parameter array {0, 1, 0, 0, 3, -2, 0, 7}, the contents of this array after the call would be {1, 3, -2, 7, 0, 0, 0, 0}. Note that **this method does not return anything, but it modifies the contents of the array object given to it as argument**. (For an extra challenge, try to make your method work **in place** so that it doesn't internally create another array to use as temporary workspace. For an even harder challenge for the hardcore algorists, make your method work in **one pass** through the array.)

4. *int[] runDecode(int[] a)* that takes an array that consists of a series of element values following the lengths of the run of those elements, listed in alternating locations. This method creates and returns a new array that contains these elements, each element repeated the number of times given by its run length. For example, when called with the array {3, 8, 1, -7, 0, 42, 4, -3} (read this out loud as "three eights, one minus seven, zero forty-twos, and four minus threes"), your method would create and return the array {8, 8, 8, -7, -3, -3, -3, -3}. Again, make sure that the array returned by your method has the exact correct length.

   (You can assume that the length of the parameter array is even. A run length can never be negative, although it can be zero. As a possibly tricky edge case, the result array could even end up being empty, if every run has zero length!)

Lab 6: Arrays give loops even more to do

JUnit: TestArrayShapeProblems.java

We continue writing code for arrays. First, let's define some useful terminology to use with this week's problems. Based on how the value in some array position *i* compares to the value in its **successor** position *i* + 1, each position *i* in the array *a* is classified as an **upstep** if *a*[*i*] < *a*[*i* + 1], a **downstep** if *a*[*i*] > a[*i* + 1], and a **plateau** if *a*[*i*] == *a*[*i* + 1]. The last position of the array has no such classification, since there is no successor element that we could compare its value with.

Armed with these definitions, create a class *ArrayShapeProblems* and write the following methods in it.

1. *int countUpsteps(int[] a)* that counts how many positions in the parameter array *a* are upsteps, and returns that count.

2. *boolean sameStepShape(int[] a, int[] b)* that checks whether the two parameter arrays *a* and *b* are the same overall shape in that every position *i* has the same classification (upstep, downstep or plateau) in both arrays. (Your method may assume that *a* and *b* have same `length`.)

3. *boolean isSawtooth(int[] a)* that checks whether the parameter array *a* has a **sawtooth shape**, meaning that it has no plateaus, and that each upstep is immediately followed by a downstep and vice versa. (Hint: it is much easier to determine whether the given array is **not** sawtooth, and then negate this answer.) As for the small edge cases, note that empty array and any array of one element are trivially sawtooth.

4. *boolean isMountain(int[] a)* that checks whether the parameter array *a* is **a mountain**, meaning that it contains no plateaus, and that it initially contains zero or more upsteps, which are then followed by zero or more downsteps. Note that by this definition, an array that contains nothing but upsteps is a mountain, as is an array that contains only downsteps. (Hint: this method should **operate in a single pass**. Don't try to make this too complicated: simply think of the sentence "Once you have gone down, you may not go up again" and build your logic on that.)

Lab 7: Expanding our dimensions

JUnit: TestTwoDeeArrayProblems.java

This week, it is time to wrangle two-dimensional arrays. Write the class *TwoDeeArrayProblems* with the following methods. As always, all these methods must be silent so that they print nothing on the console, but for your own debugging purposes during the development, note that the correct way to print out the contents of the two-dimensional array `arr` using only one line of code would be

```
System.out.println(java.util.Arrays.deepToString(arr));
```

1. *double[][] transpose(double[][] a)* that creates and returns a two-dimensional array *b* that has as many rows as *a* has columns, and as many columns as *a* has rows. (For this problem to be meaningful and possible, it is guaranteed that each row of *a* has the same number of elements.) Each element *b[i][j]* of the result should equal the value of the original element *a[j][i]* in the mirror image position. For example, if the original array contains 2 rows and 3 columns, with the elements

   ```
   1     4     -2
   -5    0     3
   ```

   the array returned by this method would contain 3 rows and 2 columns, with the elements

   ```
   1     -5
   4     0
   -2    3
   ```

   so in the array transpose operation, the rows become columns, and columns become rows.

2. *double[] minValues(double[][] a)* that creates and returns the one-dimensional array *b* whose length equals the number of rows in *a*. Each element *b[i]* should equal the smallest element in the *i*:th row of the parameter array *a*, or if that row is empty, equal 0.0. **Unlike the previous question, in this problem you may not assume that every row has the same length**.

3. *int[][] zigzag(int rows, int cols, int start)* that creates and returns a two-dimensional array of integers with the given number of *rows* and *cols*. This array should contain numbers ascending from *start* listed in order in the consecutive rows, except that every row whose row index is odd is filled in reverse order. For example, if *rows* is 3, *cols* is 5 and *start* is 14, this method would create and return a two-dimensional array with the elements

   ```
   14    15    16    17    18
   ```

```
23    22    21    20    19
24    25    26    27    28
```

4. *double maximumDistance(double[][] pts)* that is given a two-dimensional array *pts* that is guaranteed to have at least two rows, and each row is guaranteed to have exactly three elements. Each row of this array *pts* gives the (*x*, *y*, *z*) spatial coordinates of a point located in the three-dimensional space. This method should find the two points (that is, two rows of this array) that have the maximum distance from each other, and return this distance.

   To calculate the distance between two points `p1` and `p2` given as `double[]`, use the method

```
private double distance(double[] p1, double[] p2) {
  double sum = 0.0;
  for(int i = 0; i < p1.length; i++) {
    sum += (p1[i] - p2[i]) * (p1[i] - p2[i]);
  }
  return Math.sqrt(sum);
}
```

Lab 8: The problem practically solves itself

JUnit: TestRecursionProblems.java

In this lab, you get to try your hand with some recursion operating on arrays and subarrays. All of the following methods of the class *RecursionProblems* must be fully recursive, with **no loops allowed at all!** (If-else statements are allowed, as they are not loops. But no `for`, `while` or `do-while`.) Because these methods are quite short, this time there are exceptionally **six methods to write** instead of the usual four. The lab scoring rule changes for this week so that you start gaining any points for this lab at your *third* method that passes the JUnit test, your first two solved problems being warm-up freebies.

As in general with recursion, you should write these methods so that you don't use any additional fields in the class to store the intermediate results, but you pass all the data that your methods need as parameters going down, and return them as results coming up. (Using local variables inside the method is perfectly acceptable.)

As promised for all CCPS 109 labs, each method can assume that parameter values are legal, which in this lab means that the indices *start* and *end* are guaranteed to be legal indices within the bounds of *arr*. **Therefore the expression `arr.length` should not appear anywhere in any of your methods!** If you feel the urge to write `arr.length` in your method, that means that you are doing something wrong.

1. *boolean allEqual(int[] arr, int start, int end)* that checks if all elements in the subarray of *arr* from *start* to *end*, inclusive, are equal. Note that for the base case, all elements of an empty (that is, where *start* > *end*) or one-element subarray are equal by definition. After all, a subarray must by definition have at least two elements for it to have two different elements! (This principle generalizes to the counterintuitive but nonetheless true observation that any universal claim that you make about the members of an empty array is trivially true by definition.)

2. *void arraycopy(double[] src, int start, double[] tgt, int start2, int len)* that copies *len* elements from the location *start* of array *src* to the location *start2* of array *tgt*. In other words, this method works exactly the same way as the method *System.arraycopy*. (Of course you are not allowed to call this existing method.) Hint: when *len* is less than one, do nothing. Otherwise copy the first element, and then recursively copy the remaining *len* - 1 elements.

3. *boolean linearSearch(int[] arr, int x, int start, int end)* that checks if the unsorted array *arr* contains the element *x* anywhere in the subarray from *start* and *end*, inclusive.

4. *void reverse(int[] arr, int start, int end)* that reverses the order of the elements in the subarray of *arr* from index *start* to *end*, inclusive. Note that this method does not create and return a new array, but modifies the contents of the parameter array *arr* in place by reassigning its elements.

5. *public void parityPartition(int[] arr, int start, int end)* that modifies the subarray of *arr* from *start* to *end*, inclusive, in place so that all odd numbers are first together in one bunch, followed by all the even numbers in another bunch. The odd elements can end up in any order that you want inside the first bunch, as do the even elements for the second bunch. For example, if your array *a* is currently [2, -1, 3, 4, 8, 5, -7], after the call *parityPartition(a, 0, 6)*, the contents of this array might become [3, -1, -7, 5, 8, 4, 2]. (Hint: after testing for the base case of subarray that has only zero or one elements, parity status of both the first and the last element of the subarray, whichever way they might be, will somehow allow you to continue this recursion with a strictly smaller subarray.)

   Note that the same as in the above example, the array `arr` may contain negative numbers. For a negative integer `n`, the expression `n % 2` equals `-1`, not `+1`. So the easiest way to handle this possibility is to use the test `(n % 2 != 0)` to test whether `n` is odd.

6. *int countRuns(int[] arr, int start, int end)* that counts how many **runs** (maximal consecutive blocks of equal elements) there are in the subarray of *arr* from index *start* to index *end*, inclusive. For example, the array [4, 4, 4, -2, 0, -8, -8, 3, 3, 3] has five runs, whereas the arrays [5, 5, 5, 5] and [42] each have only one run. An empty subarray (that is, when *start* > *end*) has zero runs. (Hint: count how many elements are different from the next element.)

Lab 9: Those dreaded word problems

JUnit: TestWordProblems.java

In this week's lab, you write methods to operate on a list of words and practice using the `ArrayList` class in the Java Collection Framework. The JUnit test class needs the data file *words_alpha.txt* to run. Write the following methods in the class *WordProblems*. **From this wordlist file, the JUnit test class is guaranteed to only use the proper words consisting of only the English lowercase characters *a* to *z*.** Against my usually required principle of "Silence is golden", the JUnit test class should output a message `"Read 370103 words from words_alpha.txt."` at initialization so that you know that the wordlist has been successfully read in. (Your methods should still only "be seen and not heard", though.)

Your methods should not modify their parameter arraylist `words` in any way. Also, the JUnit test requires that every method must return the words that it finds in the same alphabetical order that they occur in the parameter list `words`. This should not be a problem, since your code will most likely process the words in alphabetical order.

1. Write the method `ArrayList<String> findMatchingWords(ArrayList<String> words, String pattern)` that creates and returns a new `ArrayList<String>` that contains all words in `words` that match the given `pattern` so that the found word and the `pattern` are the same length and contain the same characters in the same positions. The `pattern` can also contain question mark characters as **wildcard characters** that match **any letter that does not occur anywhere** in the `pattern`, in the spirit of the game of Hangman.

   For example, the words `"bridge"` and `"drudge"` would match the pattern `"?r??ge"`, but the words `"dredge"` and `"grunge"` would not. (Use the `String` method `indexOf` to quickly determine whether some character occurs inside the `pattern`, or precompute a 26-element `boolean` array to tell you this.)

2. Write the method `ArrayList<String> findSemordnilaps(ArrayList<String> words)` that creates and returns a new `ArrayList<String>` that contains all **semordnilaps** in `words`. A word is (slightly humorously) called a "semordnilap" (the word "palindromes" read backwards) if reading that word backwards gives you another word that is different from the original word. For example, the word `"tuba"` is a semordnilap, since reading it backwards gives you a new word `"abut"`. However, the word `"bob"` is not a semordnilap, since it is an ordinary palindrome and reading it backwards produces the same word.

   Remember that unlike `String` class, the class `StringBuilder` has a built in `reverse` method, which should eliminate one inner loop from your code. You still have to convert back and forth from `String` to `StringBuilder`, though. Also, since the list of `words` is known to be in sorted dictionary order, you can use the utility method `Collections.binarySearch` to quickly check whether the reverse of the given string is a word, instead of having to act like some "Shlemiel" and look for that reversed word by comparing it individually to every word in `words`. However, note that this method returns the negation of the insertion position whenever the word you search for is not in the list, so you need to test whether the returned result is greater than `-1` to check if that word is in this list.

3. Write a method `ArrayList<String> findAnagrams(ArrayList<String> words, String word)` that creates and returns a new `ArrayList<String>` that contains all words in `words` that are anagrams of `word`, that is, contain the exact same characters but possibly in a different order. (Note that each word is trivially an anagram of itself.)

   There are many ways to check whether two given strings are anagrams. Start by checking that both have the same length, which immediately determines for most pairs of strings that they cannot be anagrams. Then you can, for example, use an array of 26 integer counters to count how many times each character from *a* to *z* occurs in both strings. Alternatively, you can convert both strings to arrays (the class `String` has built in `toCharArray()` method to do this), sort these arrays and compare whether they are equal. The utility class `java.util.Arrays` offers pretty convenient methods for these last two operations. But there exist even faster ways for anagram checking out there...

4. Write a method `ArrayList<String> findOneLessWords(ArrayList<String> words, String word)` that creates and returns a new `ArrayList<String>` that contains all the words in `words` that can be constructed by **removing exactly one character** from the given `word` and keeping the rest of the characters in the order that they were. For example, when given the word "stopper", this method could return words such as "sopper", "stoper" and "topper", and when given the word "ledger", this method could return words such as "edger", "ledge" and "leger".

   Same as in previous problems, this method must return the list of words in sorted order and without duplicates. The fastest way to solve this problem is to loop through all the possible positions to remove a character from `word`, and then use `Collections.binarySearch` to find out whether removing that character results in a word, adding each word to the result unless it is already there. The list of words is then sorted using `Collections.sort` and returned to the caller.

   (As a fun additional exercise, can you find the English language word that produces the largest possible number of words as the answer to this method? Truly hardcore students could even try to find the longest possible **chain of words** so that each word is constructed from the previous word by removing one letter. The instructor has found several chains of eight words such as "*swaddler, waddler, wadder, wader, waer, war, ar, r*" or "*smaltine, saltine, saline, sline, sine, sie, ie, e*", but could there be a chain that contains nine words? How about in other languages whose wordlists you can download from the web?)

Lab 10: The Endgame

JUnit: TestChessProblems.java

No knowledge of chess or mastery of its play is required to solve the following problems, other than the knowledge (or the ability to look up) how some particular type of chess piece moves. (Considering the historical and social significance of chess in all cultures in both East and West, the basic chess pieces and their moves should pretty much be part of the assumed general knowledge of any educated person anyway.)

In this kind of problems, the chessboard can be of an arbitrary size $n * n$ instead of the usual 8 * 8. The chess pieces still move and capture like they would in the normal 8 * 8 game of chess (but they can now move as far as they can within the board, not just up to the distance of 8 tiles), but otherwise **any square can contain any piece of either colour**, whatever each problem happens to need. (In some wacky chess problem, there could even be a thousand queens and another thousand kings simultaneously on the board!) The squares of the board are numbered the same way as in Java two-dimensional arrays, with their row and column indices from the range 0 to $n - 1$.

Write the following methods in the class named *ChessProblems*. In each method, try to use as few levels of nested loops as possible. In each problem, the recommended number of nested loops is given in parentheses after the problem. (Try not to be a **multidimensional Shlemiel**, the *k*-dimensional generalization of the simple one-dimensional Shlemiel who uses two nested for-loops to do something to an array that could be done with just one loop. The JUnit test methods are of course written to be heavy enough so that they will get very slow for multidimensional Shlemiels who use more than the necessary number of levels of nested loops.)

Start by copy-pasting the following method to your *ChessProblems* class, since it will come handy for both questions 2 and 3. This method checks whether the coordinates (i, j) are inside the $n*n$ chessboard, to prevent your method from crashing because of index out of bounds error.

```
private static boolean inside(int i, int j, int n) {
    return 0 <= i && i < n && 0 <= j && j < n;
}
```

1. Write a method *public int countSafeSquaresRooks(int n, boolean[][] rooks)* that is given an $n * n$ board that is guaranteed to contain only **rooks of the same colour**, with the parameter array *rooks* telling whether a given square contains a rook. This method should count and return the number of empty squares on the board that are not threatened by any rook. A square is threatened by a rook in the same row or column. (**2 levels of nesting**. Hint: use two one-dimensional *boolean* arrays to keep track which rows and columns have been discovered to have a rook somewhere in them.)

2. Write a method *public int countKnightMoves(int n, boolean[][] knights)* that counts how many moves are possible on an *n* * *n* chessboard where every piece is a **knight of the same colour**, with the array *knights* telling whether the given square contains a knight. This method should count how many different moves are possible in the current board, adding up the number of moves that each knight can do. Unlike other chess pieces, chess knight can jump over other knights, but cannot move into a square that already contains another knight, or jump outside the board. (**3 levels of nesting**: two for the possible positions on the board, and one for the eight possible directions that each chess knight can potentially move to.)

   Hint: you might find the following two-dimensional array quite useful. Its each two-element row tells you how many steps the knight takes to the *x*- and *y*-directions when moving to the particular direction chosen from the eight possible directions of a chess knight.

   ```
   private static final int[][] knightMoves = {
      {2, 1}, {1, 2}, {-1, 2}, {-2, 1}, {-2, -1}, {-1, -2}, {1, -2}, {2, -1}
   };
   ```

3. Write a method *public int countPawnMoves(int n, char[][] board)* that, similar to previous method, counts how many moves are possible on a board that contains some number of white and black pawns, and no other pieces, in arbitrary positions. The board is now given as an *n***n* array of characters, with the possible values `'w'` for white pawn, `'b'` for black pawn, and the whitespace character `' '` for an empty square.

   Each chess pawn has up to three possible moves: it can move either **one step directly forward** into an empty square, or either **one step forward and left, or one step forward and right** to capture a pawn of opposite colour. (In this problem, the *en passant* captures for pawns do not exist, nor does the promotion to higher rank when reaching the row on the opposite side.) A pawn cannot move outside the board or into a square occupied by another pawn of the same colour.

   Note that unlike in actual chess diagrams whose row numbering system conventionally increases to the opposite direction than Java array indexing, for white pawns "forward" means the direction of **decreasing** row number (so a white pawn in square (i, j) might move to squares $(i - 1, j - 1)$, $(i - 1, j)$ and $(i - 1, j + 1)$, depending on what is in those squares). For black pawns, "forward" means the direction of **increasing** row number.

   (**Three nested loops**, with the outermost loop running for exactly two rounds, once for white pawns, once for the blacks, with the two inner loops looping through the positions of the chessboard. This could be done by unrolling from three loops to two by copy-pasting the entire logic for two separate blocks for white and black pawns, but that would be ugly.)

4. Write a method *public int countSafeSquaresQueens(int n, boolean[][] queens)* that works the same as the first method, but this time for chess queens instead of rooks. A queen threatens every square not just in the same row and column, but also in the same diagonal for all four diagonal

directions. (**2 levels of nesting, by generalizing the boolean array approach used for rooks**. See the hint below.)

Hint: In an *n* * *n* chessboard, there are *n* rows, *n* columns, 2 * *n* - 1 diagonals going northeast, and 2 * *n* - 1 diagonals going southeast. Allocate four boolean arrays, two of which are size *n* just like in the rooks problem, and two are of size 2 * *n* for the diagonals.

The square with coordinates (*i*, *j*) resides in row *i*, column *j*, SE diagonal (*n* - *i* + *j*) and NE diagonal (*i* + *j*). Your method should loop through the entire chessboard, and for each queen that you find, update the four boolean arrays using these indices. After this, loop through the entire chessboard again, and count how many squares are safe from attack from all directions.