COMS W1007 Assignment 2
Due Oct. 16, 2018

1 Theory Part (50 points)

The following problems are worth the points indicated. They are generally based on the lectures about CRC and UML analysis, testing, clean code (up to and including Chapter 4), class design, and some patterns.(Chapters 1 and 4).

"Paper" programs are those which are composed (handwritten or typed) on the same sheet that the rest of the theory problems are composed on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work for paper programs beyond the design and coding necessary to manually produce the objects or object fragments that are asked for, so text editor or computer output will have no effect on your grade. The same goes for the Theory Part in general: scanned clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

An important note about academic honesty: If you use *any* resource, including `stackoverflow.com`, you must properly attribute the source of your answer with the proper citation. Cutting and pasting without attribution is plagiarism. The CAs will report to the instructor any suspected violations. This even goes for `wikipedia.org`–which for this course often turns out to be inaccurate or overly complicated, anyway–since plagiarizing from Wikipedia is still plagiarism. Much better to read, think, and create your own sentences.

1.1 Primitives (6 points)

Your tiny smart watch has room for integers that can fit into 6 bytes. What range of integers can it store? Suppose we use the same 6 bytes to store some floating point numbers, in a 1 + 1 + 8 + 38 configuration (sign of number + sign of exponent + space for exponent + space for mantissa). What is its base-10 range of exponents? what is the number of significant decimal digits?

1.2 Total order (6 points)

Since a lot of Java uses the Comparable interface to order things for sorting, consider the following. Suppose you define a VeryRightTriangle (VRT) by its four parameters. These are triangles that have one leg of the right angle parallel to the horizontal axis, and one leg parallel to the vertical axis. So, a complete description of such a VRT is given by (x, y, w, h). Alternatively, you can draw the triangle between the three points (x+w, y), (x, y), and (x, y+h). Now, define the relationship "less than or equal" in the following way. VRT A is less than or equal to VRT B if all the points in A (including the border) fit in or on the points in B (including the border). Is this relationship reflexive? antisymmetric? transitive? total? If it is all four, implement Comparable by writing a simple Java method compareTo that takes in the 8 values of two VRTs and outputs the usual -1, 0, or 1. If it is not a total order and such a method is not possible, say (but do not code) some way

in which a compareTo might still be written, based on considering (x, y, w, h) to be like the four names of famous people, like Zendaya Maree Stoermer Coleman.

1.3 Critiquing a class (5 points)

Critique the class String as it is given in the API, Give an answer for each of: cohesion, completeness, convenience, clarity, consistency. Make sure you carefully examine: any constructors and/or factory methods, whether parameters can be out of bounds, any other exceptions that can be thrown, and in particular what happens with empty objects.

1.4 Test case documentation (5 points)

Using Assignment 1, fill out the following template for an actual test case for RPSLK.that you used in your solution. For a bit more information, you can see:

`http://www.softwaretestinghelp.com/test-case-template-examples/`

```
Test case ID
Test Designed By
Test Summary
Pre-conditions
Test Data
Expected Result
Actual Result
Post-condition
```

1.5 Class categories (6 points)

Here are the six common kinds of classes: Information Holders, Service Providers, Controllers (Sequencers), Structurers, User Interfaces, Coordinators. For each class, explain what constructs of Java you would expect to see within each class that makes it worthy of its label? For example, one of these classes is easily notable because of its many import statements; another by its many try/catch statements, etc.

1.6 Factory method for the Programming Part (6 points)

Using your RLEConverter class of Step 2, rewrite it as a paper program–or, at least as much of it that proves the point– so that any instances of the class are only accessible via a Factory method. You can use the ListIterator class as a guide. (If you didn't get to Step 2, do this for some other class of your choice.)

1.7 UML for the Programming Part (8 points)

Give your UML for the Programming Part: Show the UML diagrams that you developed, depending on how far you got. These can be free-hand. There is no extra credit for using any design aids. Please do these *before* you start coding! You only have to turn one set, in the final form

that documents whatever you developed and submitted as your final system(s) in the Programming part. NOTE: Also compute the complexity of your UML (2 * lines / boxes). Also indicate if any of your classes have less than two methods or more than seven methods, and if they do, justify why you used them in your design anyway.

1.8 Clean code for the Programming Part (8 points)

Justify the cleanliness of your code in the following way. What is the *best name* you have chosen in your implementation? Justify it according to the suggestions in Chapter 2. What is the *second best name* you have chosen, and why? What is the *best method* you have written in your implementation? Justify it according to the suggestions in Chapter 3. What is the *worst method*? Explain why you thought you could get away with it, even though you read Chapter 3. Finally, what *comment* of yours would the Clean Code author most like, according to Chapter 4, and why?

2 Programming Part (50 points)

This assignment is intended to explore *alternative* implementations to the *same* use case. The differences come from the choice of algorithms and data structures, which are invisible to the user.

Your eventual goal is to design, document, code, and test three classes that provide exactly the same functionality, with the first class taking advantage of algorithms and data structures in Java's API, but the second and third requiring the implementation of something new that you write yourself.. The classes should look and feel exactly the same to a user, and you need to demonstrate that by showing that the same tests get exactly the same results. And, for this assignment, you must also generate the javadoc html pages for both versions.

All three classes implement the same class RLESequence ("Run Length Encoded Sequence"), which uses some of the techniques that are actually used for JPEG and MPEG encoding of images, and sometimes for the compressing of files . Although a real user won't know what algorithms and data structures they are getting when they use RLESequence, for purposes of the assignment we ask you to provide RLESequenceV1 (version 1, using the API) and RLESequenceV2 (version 2, using your own algorithms and data structures to save space), and RLESequence V3 (version 3, saving space and time). Note that CRC, UML, Javadoc, and test strategies are necessary for all three versions, including the javadoc html for all three versions.

2.1 Step 1: Building the infrastructure (20 points)

This assignment should demonstrate that it is possible to write a class so that a user *doesn't know*–and in fact, *can't know*–how a sequence is stored internally, since on the surface it will still *operate* as if it were not compressed at all. Anything you would want to do to a sequence–for example: initialize it, set an element to a value, modify an element at an index, get a value at an index, toString, etc.–is provided by means of methods. But these methods hide from the user that the class may have encapsulated secret algorithms that make the implementation very efficient. This encapsulation also permits the class to *modify* any of its internal representations and techniques at any time, without the user knowing that that has happened.

So, this step produces a straightforward implementation of the Class, but one that is inefficient in both space and time. But it does two useful things: First, it helps explore the syntax rules of the class: what, exactly is legal to do, and what exceptions should be thrown otherwise. Second, it builds some of the classes that are useful in testing any future improvements that are "behind" the API and invisible to the user–but not a maintainer.

For RLESequenceV1, start by considering a one-dimensional array of ints. This one dimensional array can capture the visual information on a single line of image pixels, where 0 usually means pure black and higher values represent lighter shades of gray, usually stopping at 255 for pure white. You can assume throughout this assignment that image pixels *must be* within this range.

For this starting step, you will need to write *without using any compression techniques*: (a) some constructors, (b) some public methods for storing, modifying, retrieving, comparing, and printing, etc., and (c) some more exotic methods that users are likely to request, like a counterpart to the String method "concat", and (d) any necessary internal private methods to help make the above methods more DRY: "don't repeat yourself". You should use only the data structures readily available in Java and/or the API: String, a[], ArrayList, LinkedList. (Yes, there are other API data structures that are more complex, but it would be a design error to use them.for such a simple problem.)

For (a): You need to write the constructor RLESequenceV1(int length), which establishes a RLESequence that can hold "length" number of values. You need to decide if you should preload it with anything, since a lot of these sequences may just be all black or all white. And, you need to consider a reasonable way to record an empty RLESequence. You also need to establish some default constructors, and to defend your choices of default length, and, perhaps, the default content.

For (a): It is also helpful if you write the accessor method length(). And, you should consider that having a constructor RLESequenceV1(int[] entireSequence) would be helpful. This last constructor takes the usual one-dimensional array, checks it for correctness (e.g., what if one of the values is negative?), and creates a RLESequence from it, which may not be stored internally as an array at all

For (b): Write public methods–make sure your methods follow the usual naming conventions!– that allow a user to insert a value at an index, to modify the value at an index (if it exists: what do you do if it does not exist?), to retrieve the value at an index, to see if two of these sequences are the same ("equals()"), and to create a printable String of the contents ("toString()"), which includes the leading and trailing square brackets and which separates values by using exactly one space. For example, toString(myRLES) should output something like "[5 5 5 5 1 1 3 3 3 3]".

For (b): You will also have to decide on whether the IOOBE (index out of bounds exception) applies. For example, starting with an empty user sequence, can the user ask to insert a value at any possible index?

For (c): It is also helpful to provide the methods addToHead() and addToTail(), which are like "concat()", except that the first method *pre*pends an RLESequence, and the second *ap*pends an RLESequence. Note that these methods, as well as every other method, should work properly even if one or both RLESequences are empty, or are overly long..

Then, test your class by creating some RLESequences, exercising every method, and printing out some results. As usual, you should have a Runner class, appropriately named. And, probably a Tester class that simply walks through a script to do the testing.

It is up to you to analyze and refine the above use case and record your decisions on what

kinds of assumptions you are making. (As usual, we will use Piazza to resolve any ambiguities or contradictions.) Doing this analysis, you should be also building up the test cases that will be used to validate the eventual class, in both this version and the next. Note that this test case analysis can be done even before the CRC is written. For example, you must decide what happens when a user does a command that may not make the usual sense, like attempting to replace a value if the sequence is empty.

Then, write the CRC, the UML, and the Javadoc that summarizes your choice of algorithms and data structures. In particular, it is up to you to select how the file is represented inside your system (one big String? an array? an ArrayList<something>? a LinkedList<something>?), and how to explain these choices in the class and method comments. You should select your underlying data structure carefully, and justify it in the javadoc. Note that many of the required methods should be relatively simple to write, except for the various exceptions they can throw, for example, if a user asks to set a value to a negative amount..

Do these steps *first*, because you will next be writing the implementation of the system a second time in Step 2. Except, the guts of that class (those internal decisions and code that the API never talks about) will have to be changed. But the basic look and feel must remain the same to the user.

For this Step, you must submit CRC, UML, *Javadoc html files*, code, and test runs.

2.2 Step 2: Saving space (15 points)

This class, RLESequenceV2, will demonstrate a particular way of compressing the data in a file, which is particularly useful when much of a file repeats or reuses many of the same values. For example, think of a cartoon like The Simpsons: there are very large regions of an image that have exactly the same color. Since many regions of an image, such as the sky, or even skin in close-up, have many identical values next to each other in any given row of pixels, this encoding usually results in a great space saving. This is also true of graphics and animations like video games, where it can lead to compression factors of over 100. Note that if the class can cut down on space, it also cuts down on transmission time to mobile devices, since the data structure is more compact, and transmission time is severely limited, whereas computation time in the device generally is not.

The way that RLESequenceV2 squeezes data storage is by carefully noting those values which repeat. For example, it could represent a user sequence of numbers such as [5 5 5 5 1 1 3 3 3 3] as an internal representation of ”field code pairs”, the first code of the pair giving a count, and the second code of the pair giving a value. So, the above example could become something like: [4 5 2 1 4 3]. That is, there are four 5s, then two 1s, then four 3s. Alternatively, it can represent the sequence as [item1 item2 item3], where item1 is [4 5], item2 is [2 1], and item3 is [4 3]. Or, alternatively again, it can represent the user sequence as [count value] where count is [4 2 4] and value is [5 1 3]. Part of your job as a designer is to select which data structures are best. Note that yet other representations are also possible, and are valid for this assignment if you choose to use–and defend!–one of them.

You need to write the class RLESequenceV2, which is based on RLESequenceV1, except that it uses a compressed internal representation of the sequence. You should find it useful to write a second class, RLEConverter, whose methods can convert whatever you were using in RLESequenceV1 into whatever you are using internally in RLESequenceV2, and the reverse.

This new class has to have the usual constructor(s), two methods (you can call them toAPI() and toSpace()) which do the conversions, and whatever private helping methods you need. One of those helping methods, useful for debugging, would be a toString() method whose job it is is to "dump the guts" of the internal representation. For example, if your internal representation was based on items, it would produce the string "[[4 5] [2 1] [4 3]]". Note that in general no user would ever need to see this internal representation, but developers and maintainers probably would.

For Step 2, you should consider that it is difficult to predict how much compression will happen, since this can be come quite subtle. There are at least four important design issues to consider.

First, note that sometimes a long user sequence can be compressed severely. For example, if the user sequence is just 1000 repetitions of 0, it can become internally [1000 0]. But sometimes a sequence stays the same, or actually becomes *longer*, for example, user [6 7 6 7 6 7 6 7] becomes internally [1 6 1 7 1 6 1 7 1 6 1 7 1 6 1 7]. Otherwise, a user could recursively keep compressing the result of the compression until everything was compressed to nothing!

Second, note that although the values are limited to be non-negative integers less than 256, the counts are not. This can affect your choices of data structures and algorithms.

Third, your methods to insert, modify, and delete can have unusual effects. For example, starting with the sequence [5 5 5 5 1 1 3 3 3 3], changing any of the 5s to something that is not a 5 will make the internal representation longer–except if the last 5 is changed to a 1 (why?). So, one easy way to accommodate these weird "corner cases" is to first convert the internal representation to that of V1, then use V1 code to make the changes, then convert back to V2. Obviously, this take time, and sometimes can be very wasteful, but we will allow it for V2. For example, if you start with internal [1000 1] and then the user wants to change any of these items to a 1, it takes a very long time to end up with [1000 1] again. It even takes a very long time if the user wants to change any of those items to a non-1. Note that these problems can also occur with addToHead() and addToTail(), for example, when the ends of the two sequences match in value.

Fourth, equals() has to handle some unusual cases, too. Suppose the internal representation of [4 4 3] had become internally [1 4 1 4 1 3] instead of the proper internal [2 4 1 3]. This is probably an error. Should equals() say both internal representations represent the same sequence?

Note that this would be a good time to back to V1, to make sure your Tester checks for all of these tricky (but legal) cases.

If you do this Step, you must submit CRC, UML, *Javadoc html files*, code, and test runs for both Step 1 and Step 2. Note that the Javadoc html should be identical, as the user interface should be unchanged, and that the CRCs and UMLs should be similar if not identical.

2.3 Step 3: Saving time (10 points)

Write RLESequenceV3, which saves both space and *time* by operating on the internal representation *directly*, without converting it to V1 and back again. Note that V3 still has to provide the same functionality as V1 (and V2), and uses the same improved representation as V2, but uses different *algorithms*. For example, if the internal representation is [1 2 4 3] and the user want to change the value at index 0 to a 3, then the algorithm verifies both the index and the value, then finds that value in the internal representation, and then checks to see what needs to be done to produce the proper [5 3], *without calling on help from V1*.

Note that this would be a good time to back to both V1 and V2, to make sure your Tester checks for all of these additional tricky (but legal) cases.

If you do this Step, you must submit CRC, UML, *Javadoc html files*, code, and test runs for all three Steps. Note that the Javadoc html should be identical, as the user interface should be unchanged, and that the CRCs and UMLs should be similar if not identical.

2.4 Step 4 (or: 1a, 2a, 3a): Creativity (10 points)

This is an exercise in adding additional functionality to whatever Version (V1, V2, V3) you completed. You only have to augment your highest version. So, you can submit for this Step even if you haven't done all of the Steps previously.

Pick *one* of the following. No extra credit for doing more than one.

a) Implement the method changeValues(byWhat). This method says that every value in the user sequence is converted to v+byWhat, that is, it is increased by byWhat–which may be zero or negative. Make sure you decide, defend, and test what happens if the argument is illegal, or if the resulting values are "too big" or "too small". Depending on your version (V1, V2, V3) and your data structures, this could be relatively clean and easy, or messy and hand, so it might be a good idea to consider this possible your prior design decisions.

a) Add the method tailReplace() which takes an index and an RLESequence as parameters, and substitutes the new incoming RLESequence at the index position. For example, if the user sequence is given by [5 5 5 5 1 1 3 3 3 3], and the user wants tailReplace(2, myNewTail), where myNewTail is given by the user sequence [7 7], then the result should.be [5 5 7 7]. Note that tailReplace() is useful when you also add the method insertAt(), which inserts a user sequence at an index, but keeping and moving the rest of the original sequence "over to the right".

c) Consider that color images don't just have a black-and-white value in the range $0 <= value <= 255$ for each pixel. They have *three* such values, one for red, one for green, one for blue. Augment your highest version so that users can use represent RLESequences of the form [(0,0,0) (100,100,100) (50, 150, 250)]. Then create the class TesterRGB that tests your methods.

If you do this Step, you must submit CRC, UML, *Javadoc html files*, code, and test runs only for the whatever of Steps 1, 2, or 3 you completed–but the last Step should include your creative modifications required by this Step. So, for example, if you did the first two Steps, then submit what is required for Step 1 before any creativity, and submit Step 2 as creatively augmented.

3 General Notes

For each Step, design and document the system, text edit its components into files using Eclipse, compile them, debug them, and execute them on your own test data. When you are ready, submit the text of its code, a copy of your testing, and whatever additional documentation is required. Make sure that the source code is clear and its user interface is comprehensive and informative.

Write your classes clearly, with a Javadoc comment at the beginning describing what the class does, how it does it (i.e., in some form, talk about its CRC and UML), and justify how it will be tested. Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. For one reasonable set of suggested practices and stylistic guidelines (but one that does not refer to the Eclipse defaults), see the website of the textbook used in COMS W1004, at: `http://www.horstmann.com/bigj/style.html`

4 Checklist

Here is an approximate checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below.

4.1 For the Theory Part

Answers, either handwritten then scanned, or in machine-readable form. No extra points for using a text editor or other formatting tools; hand-drawn diagrams are fine. Submitted electronically in Courseworks by the time specified (i.e., by the beginning of class, exactly). The theory file should be separate from the files for the programming. Name and UNI must be prominently included.

4.2 For the Programming Part

Answers submitted electronically in Courseworks by the time specified (i.e., by the beginning of class, exactly). All code (*.java files) documented according to Javadoc conventions and all trial runs. These files should be separate from the files for the theory. Name and UNI must be prominently included in each class (use "@author") and on each output. Rather than providing a separate ReadMe file, one of your classes should have an obvious name that will attract the attention of the TAs (like "HW2Runner", for example), and *inside* it there should be an overview of the components of your system. The classes, in whatever state of completion they are, *must compile*. Pay special attention to the completeness of your Tester class.

4.3 For both Parts

Please put all your submission files (theory, *.java, example runs) in one directory (named, for example,"HW2"), and then compress the contents of that directory into one file. Submit that compressed file to Coursework2. The name of that compressed file should be "myUNI_HW2", and it should have the appropriate extension like ".zip", except that "myUNI" should of course be replaced with your real Columbia UNI. *Please* use this convention as it makes the TAs' job much easier:

You may submit electronically as often as you wish, but it will be your *last* electronic submission before the deadline which is the official one that will be graded and (if need be) executed.