Contents

1 Overview + Goals

This result of this assignment is to implement a `TextProcessor` which randomly generates an output string based off of trained data from the input string. The class is contained in `RandomWriter`, which can read an input and write to an output. My goal was to practice unit testing, as I'm not too familiar with it. I also wanted to practice the encapsulation and polymorphism techniques explained in lecture.

2 Solution Design

2.1 Algorithm

We make the following observations about the code.

1. For large seeds i.e. large values of `k`, string become quadratically more expensive to operate on. For this reason, we will store only the parsed input string. All remaining operations will be done with integer positions within the input string.
2. We reshuffle the seed after the first random selection again only if
   (a) The seed reaches the end of the input string
   (b) Such seed appears nowhere else in the input string

2.1.1 Underlying Data Structure

The code's underlying structure is a `HashMap` which stores each possible seed's following character. However, since all operations are done with an integer position and not the seed's substring, we create an internal wrapper class called `SeedIndex` which contains the position of a seed. We override the `.equals()` method of this `SeedIndex` class such that two seeds at different positions but containing the same seed will evaluate to true. Each key-value pair consists of a seed's `SeedIndex` as the key and an `ArrayList` containing the positions of all the characters following the seed or any equivalent seeds as the value.

2.1.2 Input Validation

In the following order, we check the inputs:

1. Check if the entire `args` array is null
2. Check if the length of the `args` array is exactly 4.
3. Check if any of the array elements are null
4. Check if level (`args[2]`) and length (`args[3]`) can be parsed to an Integer.
5. Check if level and length are nonnegative
6. Check if input file exists, is accessible, and is readable
7. Check if output file is accessible and writeable
8. Check that the level is strictly less than the input file size

2.1.3 The `main()` method

The `main()` method handles all of the errors if they exist. It will print them out to `System.err` and return if an error is detected. If not, it will proceed with instantiating a `RandomWriter`. It then reads the input file by calling `readText()` writes its generated text on the output file by calling `writeText()`.

2.1.4 The Constructor(s)

I've not modified any code in the factory constructor. In the private constructor, the member variables are instantiated. It also precompute powers of 31 which are used to calculate the hashes.

2.1.5 Reading in the Input File

The method `readtext()` does two things:

Parsing the File It uses a `BufferedReader` to read in lines and writes this to a `StringBuilder`.

Precomputation It calculates the hashes for each seed in the file. Since consecutive seeds overlap quite a bit, it uses a rolling hash function to compute the hashes.

With the hashes, it builds the `HashMap` by cycling through every seed in the input string. Once the `HashMap` is built, we can choose random characters following any seed.

2.1.6 Writing to Output

The method `writeText()` also does two things:

Random Generation Using the precomputed `HashMap`, it randomly selects an initial seed, randomly chooses the next character, and updates the seed. It does this until we reach a situation as described in Observation 2(a), in which we randomly reroll the seed, or we constructed an output string of length length.

Writing to the Output File It uses a `BufferedReader` to read in lines and writes this to the targetted output file.

2.2 Assumptions

Here are some assumptions I made with the code, along with what happens if these assumptions aren't satisfied:

- All characters must be storable in the `char` type.
- The input and output files must be readable. Although it's possible to handle file permissions, Java fails to build the code if it's unreadable because it can't copy the file to the binary folder.

- The public methods (including the factory constructor) have been passed the correct inputs . We do not parameter check in the factory constructor or the `readtext()` or `writetext()` methods, so we assume that those checks have already been taken out either automatically by the `main()` method or manually by the caller.

2.3 Design Decisions

I also made the following design decisions:

- Adding spaces to the end of new lines. This is so as to separate words in a more readable format. Also, it allows for more seeds to overlap, improving the behavior of the random string generation.
- In the situation which we need to reroll the seed, we append the rerolled seed to the end of the output string. This is to provide context for the randomly selected characters that follow it.

3 Reflection

3.1 Solution Scope

`RandomWriter` will generate text randomly from training data supplied by an input text file. This generation will work for all characters that can be stored in a Java Character. It only works for right-directed languages/files, and will not work for vertical languages/files, such as certain kanji books, or left-directed languages, written in Arabic, Hebrew, Farsi, etc. Also, the code takes quite a bit of reliance on `BufferedReader` and `BufferedWriter`, so we try our best to parameter handle it, but if for whatever reason either of those two classes break, either due to bugs there or strange file properties, such as permissions being off, almost certainly `RandomWriter` will not work. Finally, we can only call `readText()` once, or else the translation from seed to position will not hold.

3.2 Problems Encountered

I had a lot of trouble with indexing. It was particularly hard for me to figure out where the last seed was, and what to do with it, as I didn't need to read the following character. Unit testing was also a challenge, as I was not familiar with it. Setting up the unit tests and writing them indeed took about 70% of the time in this project, even though the test code is shorter than the source code. It is quite difficult to compare two `HashMaps` with values `ArrayLists`. Finally, I learned the hard way that changing file permissions on your computer to test if `FileReader` can open an unreadable file is a bad idea because a) you can't delete the file and b) the code doesn't even compile.

3.3 Code Repeatability/Quality

My code follows Java's coding standards. I'm pretty happy about the readability of my code, because I myself would forget what my code did, but I had good method and variable naming to help me figure it out. For quality, I'm also satisfied with the Javadoc commenting, method splicing, and practices used for member variables. However, I did feel that the class was a bit long, and I might've preferred if I had broken my methods into a different class. This would've also helped with readability as well.

My unit tests have a significant amount of repeated code. I could not figure how to condense this, as the `setUp` method doesn't take parameters. However, I'm sure I will get better at unit testing as I do it more.

3.4 Optimizations

I made three important optimizations:

1. The first is mentioned above, but its translating all the string operations into positional operations. As many string methods are generally inefficient, such as `.equals()` or `.substring()`, this optimization improves both run-time and memory usage.
2. Using a `StringBuilder` object optimizes spaces, as concatenation of strings is also very slow in Java.
3. Storing known equalities. When two seeds are compared and found to be equal, I memoize the latter of the two so if I ever have to compare them again, I can quickly search for the equality condition rather than parse through the seed and compare every character.

These three optimizations are, in my opinion, what makes my algorithm design unique.

3.5 Interesting Results

3.5.1 Character Freedom

I realized this code works for literally any character that can be stored in a Java char. It works for emojis, mandarin characters, weird LaTeX symbols, etc.

Less interesting, but putting the string

”abacadbcbdcda”

with level 1 would create a long string of nonconsecutive letters, such as

cdcadcdbacbdcdabcacbcdbcdbadabdcbdadbca a dadcbcbcda adababdbcadcb adbcadaca aca bcacbabcadbcbdadcdab

Finally, placing a string with no repeated seeds, such as

”abcdefghijklmnopqrstuvwxyz”

with level 4 will give substrings of such input string that always contain the last character, such as

”defghijklmnopqrstuvwxyz ghijklmnopqrstuvwxyz ijklmnopqrstuvwxyz op qrstuvwxyz abcdefghijklmnopqrstuvw”

The last part is spliced off so as to fit it into the length 100.

3.6 Room for Improvement

As mentioned above, my unit testing was very repeated. I also could've implemented automated black box testing and more statistical tests to supplement by unit testing. Unfortunately, I am not well versed enough in either coding or statistics, so I had to pass on those tests.

4 Testing

4.1 Motivations & Observations

I noticed that since the operations within `RandomWriter` hinge on random number generation, we cannot expect a single output from a single output. Instead, there is a wide range of outputs, whose frequencies form a distribution. However, we *can* control the output if we modify our input in a certain way to restrict the generation.

I felt that a combination of probability, black box testing and white box testing was the way to go. I focused more on unit testing because I was unfamiliar with it and also I had many complex methods that I wanted to check worked.

4.2 Black Box Testing

For Black Box Testing, I took a large text file, such as Cat in the Hat, and set the level to ten and allowed it to generate text. Then, I looked for a phrases in the output file with length 10 and checked if they existed in the input file. If such a phrase didn't exist, then my code successfully generated new text. I actually failed this the first couple of times because I found my code was just continuously rerolling a new seed.

4.3 Unit Testing

For Unit Testing, I tested all of my methods with custom inputs and outputs to make sure they worked properly. For example, to test my `validateParameters()` method, I passed in both parameters that did and didn't work and checked that the correct error message (or lack of) was received. To test my custom hash function and the rolling hash, I passed in some seeds and compared them to my own calculation of my hash function. I did these for all methods that I deemed complex enough that I need the confidence gained from the tests to modify them. I also unit test `readText()` by checking that it read the input file properly and `writeText()` by checking that it wrote to the output file properly.

4.4 Statistical Testing

I also tested if the seed generation was truly uniformly distributed. I used a very simple input, "abacadbcb-dcda" with level 1 and computed the probabilities that each character should appear. I then ran the output construction many many many times and computed their frequencies and compared them against by own probability calculations.

I know some of my classmates implemented a chi-square test/other statistical test to black box check as well. I tried this but could not get this to work within the time constraint, so I scrapped it.

4.5 Edge Cases

There were a couple of edge cases I needed to handle:

1. If level equals 0, then the output string should just be characters randomly selected in the input string.
2. If length equals 0, then the output string should be completely empty.
3. If level equals input string length - 1, the code should just reprint the input string multiple times until the output string reaches the specified length.
4. Validating Parameters to ensure that all of the arguments are non-null, parseable immutable, and read/writeable.
5. Checking to see if Unicode characters are encoded properly.

I visually inspect the outputs with black box testing to ensure that it works properly.

4.6 Sufficiency

I feel confident in my unit tests to ensure that all of my methods are working properly, even after modification. They are tested on very general, random inputs, so passing the test gives me the confidence that my code works. Therefore, during black box testing, the bulk of the general cases passed, and all I had to do was fine-tune for edge cases. I then black box tested the edge cases and saw if my results were correct, which they were. Since I tested on both random general cases and a broad range of edge cases, I feel that my testing was sufficient

5 Social Impact

If we continue to broaden our scope:

1. `RandomWriter` in specific has the potential to create very entertaining text. A very good implementation might be able to create a good book.
2. The karma may be able to create nice pictures, enjoyable music, or maybe even a great film.
3. Text generation can be used to write very descriptive texts on anything subject or topic.
4. Content generation can be used to provide entertainment value at great volume and for very cheap.

This list continues to broaden further. I will discuss text generation in particular.

Text Generation has both a lot of potential and value: It can generate funny stories about literally anything. It can be entertaining for those fond of reading. It can also be used to format lots of information on the internet to a single, concise source of information, if for example, a web crawler found all relevant pages to a topic and generated a survey article on it.

However, it poses numerous risks and ethical questions. Text Generation could be used for cyberbullying. It could be used for spam. And who gets the copyright for generated text? Is it the coder or the one who clicked generate text?

On a broader scale, most of AI and Ml has enormous potential. It can be used to predict cancer before it comes, or to promote self-driving. However, it also brings into question issues on data privacy, security, control, and more. I personally think that the direction society is heading to is one where we overlook ethics and attempt to fix these problems in the aftermath. I can't say if this is the right way to do it because frankly, I'm not sure.