**Cosine Distance Computer – Matthew Sloyan G00348036**

For this project I made various decisions from brainstorming, writing it out on paper and trial and error. I also used the lecture notes for guidance regarding Big O notation and some of the other implementations. Some of the decisions and a guide for each feature be found below under their relevant headings.

**To run the Jar file:**

java -cp .;./four-square.jar;./jsoup.jar ie.gmit.sw.Runner

or

java -cp .;./four-square.jar ie.gmit.sw.Runner

as it implements the Jsoup.jar file.

**How it works:**

When running initially you will be presented with the option to enter your own cipher key (1) or generate one randomly (2). If 1 is selected it then gives you the options to enter one large (>=50) keyword or two shorter (>=25) character keywords which will populate key 1 & 2.

**User entered keys:**

From researching and reading the lecture notes I found that a LinkedHashSet would be the best fit for this as it allowed me to add inputs which are in order, unique and inserted quickly. The user is given the option to enter a string of characters which can be anything, as it will only ever add one of each letter ranging from A-Z excluding J in order of input. For example, if you input “ffjjsss234kk” it would add f, s, k.

The same method is used for both options the to cut down on code, so for (>=50) it is called twice with each half of the string passed in and for (>=25) option it is called for each key. If the LinkedHashSet is less than <25 then it will look through the hardcoded alphabet ArrayList while using the contains method to check if the letter is in the LinkedHashSet or not. If it’s not, then it will add the letter essentially filling up the rest of the key with the remaining letters of the alphabet to always create a 25-letter key.

**Randomly Generated Keys:**

Using Collection.shuffle a random key is set for use throughout the program. The keys are set in the CipherKeys class. To allow the same keys to be accessible across the program I have created a method which creates an instance of the class so that the keys are saved when assessing for encryption or decryption.

**Menu Options:**

After selecting an initial key, you will be presented with the overall menu options listed below.

**(1) & (2) Encrypt/Decrypt:** First you are presented with the option to select a file name to output to, so if you input “fileName” it will save the encrypted/decrypted text file to filename.txt in the same directory as the .jar file. Next the JFileChooser window will open allowing you to choose a text file to encrypt/decrypt from anywhere on your PC. Once selected it will call the parse method which will encrypt/decrypt your document. This is described further below. Lastly you are provided with an option to print out the encrypted/decrypted document to the console.

**(3) Set Keys** – Uses the same class as initial key set (above).

**(4) Exit –** exits the program on input of four or any other value.

**Parser Class:**

When this class is called the parameters, it’s passed are determined by the class it’s called from and whether it’s for encrypting a file, encrypting a URL or decrypting. For example, the call for encrypting a file would be “p.parse(fileName, false, false);” with the first false meaning it’s not a URL and the second meaning that it’s for encrypting. The while loop reads through the file line by line stripping the text of all characters except A-Z, spaces and numbers 0-9. If it’s a URL JSoup parses the line and gets just the text from it removing the HTML tags. The line is passed to the encrypt/decrypt class as a StringBuilder which is returned and appended to an overall StringBuilder of the encrypted/decrypted file. This is then passed to the PrintDisplay class which prints to the selected file and gives the option to print to screen.

**Encryption/Decryption:**

An overall for loop runs through the line two characters at a time to create the bigrams. Also, a X is appended to each line if uneven to ensure all characters are encrypted, this is removed when decrypting. A series of if else statements which skips over spaces or numbers until it finds two suitable characters to create a bigram. I did it this way as it makes it easier to keep the source formatting of the document. When it finds two suitable characters for Encryption it gets the ascii value and takes away either 66 or 65 to account for the missing J, this gives the position of the letter in the array as it’s in order of A-Z. For decryption a for loop must be used to search through the array till it finds the value as it’s unordered. To find the final position of the value to swap it with some maths is used such dividing by 5 to find the row and modulus 5 to find the column, this is then added to row \* 5 to get the final position. Lastly the values are swapped using the calculated final position. By doing it this way I eliminated a nested loop and decreased space complexity by not using a 2D array.

**Overall goals and thoughts:**

I originally started by using the two-dimensional array cipher key and had nested for loops to find and encrypt/decrypt values. But I felt there was a more efficient way to do it with less loops especially for encryption. So, with time I devised a way using one dimensional arrays and ascii values which cut out the need for a nested for loop to find the positions making the overall running time for encryption linear rather than quadratic. Another thing I changed to increase efficiency was to read and encrypt the file on the fly line by line rather than the whole file at once. This only increased the speed slightly for WarAndPeace but for ultra large text files this would make it considerably faster. I have also put all the menus and spread out the code as much as possible throughout different classes to enhance readability and code cleanliness. I have further extended this with using a lot of the same classes for multiple things especially for encrypting and decrypting as majority of the code is the same. I tried to keep speed and space complexity in mind by using byte declarations where possible and StringBuilder’s rather than strings.

**User interface:**

Overall I wanted to keep it simple, clear , easy to use and understand.

**Additions/Extras:**

* JFileChooser implementation.
* JSoup implementation which removes HTML tags.
* Validation on all menu inputs and when inputting keys.
* Extended the encryption/decryption to included numbers.
* Kept source formatting of document (all spaces, new-line etc.)
* I added the option to be able to save the key to a file when setting the keys, and then load this key in again. So, say if you encrypted a document using a random key and wanted to decrypt it later you wouldn’t be able to as the key would be lost. This uses much the same methods as setting the keys.