Yecheng Yang
CIS 573 – Homework 1 Writeup

1. **Design**
   a. **Multiple classes with different roles**: Since reading file context and performing cross validation are logically independent, I designed separated classes for loading file context and performing cross validation so that the implementation details of two main tasks of the program are not tangled together. By specifying the way that frequency of each letter is stored and providing accessing methods, the whole cross validation, frequency analysis process is virtually independent of the methods written in FileInput class. Therefore, changes to be made in reading files will not affect anything during cross validation or frequency analysis. Vice versa, changes made for cross validation do not require corresponding changes in file input part. On top of those two classes, I also created Cipher class and AnaylsisResult class, and each of them has a very specific role in the program. Cipher class deals with ciphering the text and the AnalysisResult class deals with storing and outputting result of cross validation. This design ensures that result is independent of analysis and file io part of the program, especially CrossValidation class, which encapsulates frequency analysis and cross validation. By doing this, I logically separate analysis from outputting results. Therefore, the method chosen to decrypt the file will not affect the result representation. In case we wish to send the results to GUI or even save it to files, it could be done just inside the Analysis Result class.
   b. **Specific functionality of each method inside a class**: Inside FileInput class, I broke the functionality into two functions: one will check if the input directory is in a desired format and put files in the directory into a file array and the other actually go through each of the files and populate the data structure needed for frequency analysis in the next step. By doing so, I modularized the FileInput class. This really makes potential changes easy to implement. For example, the source of input data is changed to a database and file format is still plain text (and we still uses frequency analysis). The only function needs to be changed/overridden is openDir(), since this function needs to be able to get files from the database. But after that, the rest of the class will remain the same. In addition, I have created a wrapper method of openDir() called openSource() which will be called outside of the class. When the file array is populated through a different data source, such as a database, the openSource() function call from main class will not change, and I will only need to define a new openDir() function that can read each of the files and populate the data structure that keeps track of number of each character in each file for later analysis. Similarly, if the file format is changed to a new format, such as json, the only part of the class that needs modification is the loadFile() function. Overall, these design decisions will greatly reduce the changes both inside and outside of FileInput class to be made during changing input data and input file format.
   c. **Usage of Interface**: I created a Cipher interface and a Substitution class that implements Cipher interface. By creating an interface, I can guarantee that each

cipher, whether it is a simple substitution cipher or a more complex cipher, will have the same methods that performance cipher/decipher and have the same format of output. Using an interface instead of an abstract class, I do not specify any fields within the class since different ciphers may have different implementations and do not require the same fields. If substitution cipher will be changed to other ciphers, we need a new class that implements the same interface. The usage of the new cipher class will not change at all. And since the cipher is declared as interface in the class where it is used, the only thing I need to change is to replace Cipher cipher = new SubstitutionCipher() with new XXXcipher() and nothing else written in the main class or crossvalidation needs to be changed.

   d. **Minimal number of fields in each class**: Inside each of the aforementioned classes, I keep the number of fields to a minimum. Any other variables that were used in the class are kept as local variables for easy understanding of the main purpose of the class. Having too many fields may confuse the future users of the system and so I only keep the variables are absolutely necessary. For example, the ones that help define the class or will be used through a public access method outside of the class. By doing this, my class design is very concise and understandable.

2. **Readability/ Understandability**
   a. **Naming**: All the classes and field/local variables are named according to Java naming convention. In addition, all the variable names and methods names are dictionary words. Class names are all nouns and start with capital letters, such as CrossValidation and FileInput, whereas methods names are all verbs and start with lower case letters, such as crossValidate() and loadFile().
   b. **Punctuation**: I used punctuation and indentation throughout the program to ensure each line of code as well as each level of logical is represented in a very organized manner.
   c. **Comments**: I used proper amount of comments (every couple of lines) to ensure functionality of each section of the code is well described and people who wish to trace the code do not need to read every single line that was written.
   d. **Complexity**: I keep the data and structural complexity of each function to a minimal extend. I used as few variables and levels of logical (if/else and while loops) as possible so that each of the functions can perform its dedicated task while maintain very readable and easy to trace.
3. **Efficiency**
   a. **Counting character instead of cipher/decipher**: Given the nature of substitution cipher and frequency analysis, I realized that it is not necessary to cipher the entire file. Instead, it the only the count of each letter that we are interested in. Since substitution cipher only change one letter to another and creates a one-to-one relation between the original letter and the ciphered letter, it is only necessary to count the number of each letter in the original, non-ciphered text and then use those counts to find the count of each letter in the ciphered text.

For example, say there are 100 a's in the original text and the substitution cipher maps a to c. Then, in the ciphered text, the count of letter c will be 100.
Through this method, I only needed to read all the files once and count the number of each letter inside. Then, I can use a cipher to determine the corresponding letter in the ciphered text. By doing so, I was able to reduce both runtime and memory requirement on the program. Instead of applying cipher to each of the letter in the text to cipher, I only need to cipher 26 times (one time for each letter) for each of the files. Number of cipher calls reduces from number of total letters in a text to a constant. And therefore, my program avoids a lot of overheads

b. **Usage of 2D array**: I used a 2D array inside FileInput class to store the count of each letter inside each of the text file. I chose this data structure over a list of HashMap for the following reasons:
    i. Array access is faster than HashMap access even though they are both O(1). Since I was able to assume all the characters in the text are ASCII encoded, I only need to focus on the letters, which are a-z. Therefore, I was able to specify the length of the array to be 26 and be certain that will meet my need.
    ii. I chose to use another array to store the count of each character because, again, array access is faster than either ArrayList, which has amortized O(1) complexity. Since I was able to count the number of files when searching the entire directory, I can explicitly initiate a normal array to hold the counts for all characters of all the files.

c. **Usage of Priority Queue over other data structure**: During frequency analysis, I used two PriorityQueue's to find the character with highest frequency in the selected text and in all files remaining in the corpus. This is the most efficient way to conduct frequency analysis because priority queue is based on a priority heap and focuses only on the head of the queue. In this case, a heap satisfies our requirement as we only need to poll heads of both queue and map one onto another to establish a decipher mechanism. The time complexity for both add and poll is O(logn) for heap. It is much more efficient than sorting, which will take at best O(nlogn), depending on which sorting mechanism we choose. It is also better than data structures such as TreeMap even though a TreeMap also has O(logn) time complexity for add and poll. This is because a heap does not care about the internal order of all the elements and less work to be done during adding new elements leads to better performance.

4. **Testing**

a. I used Junit4 to test the functionality of the program. For file io part, I tested all the functions with incorrect inputs, such as empty directory, to see if exception is raised as expected. I used try- catch block and assert if the output is the correct exception. As for cross validation, I created two simple files and a substitution cipher. Then during every step of the process, I tested if the output or the changes made by the method is correct by calling assertEquals or assertTrue on the output with the correct answer, which was easy to find out given the simple example I created.

b. I stopped testing because I have tested all parts of file io and all the main functions of the program. And all of them are performing according to my expectation. I am 100% confident that my answer is correct.