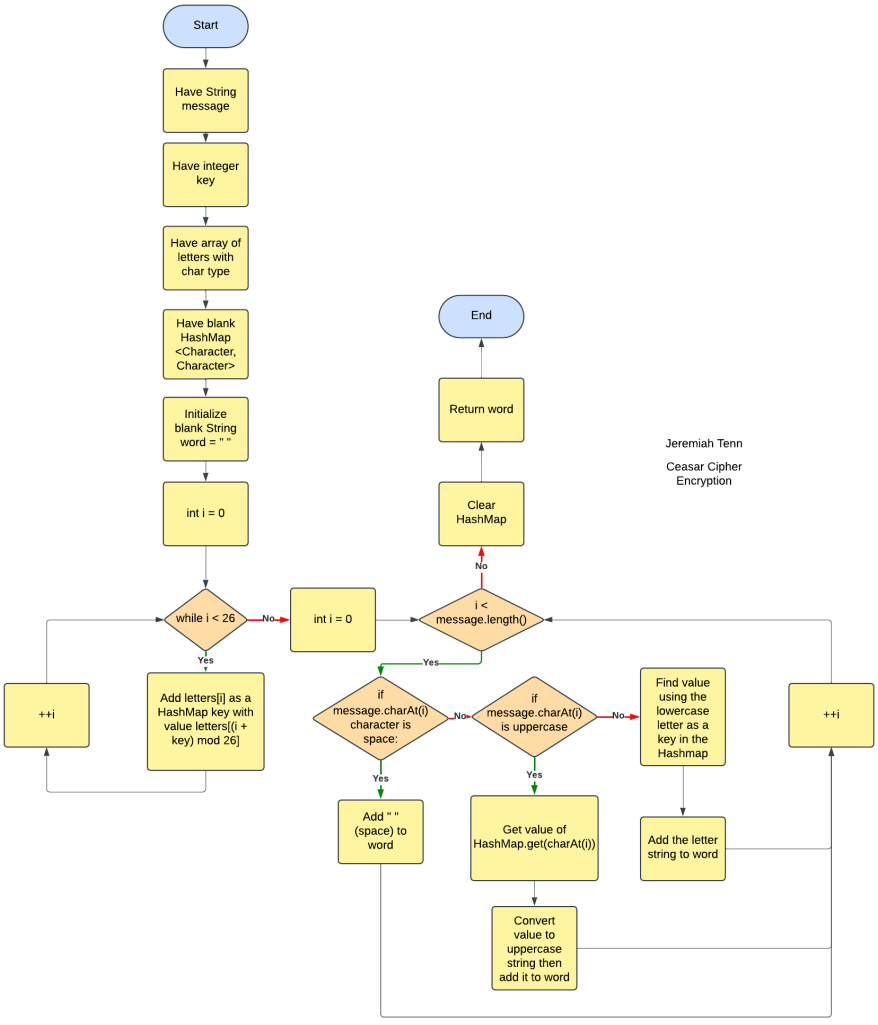
Final Programming Project Report - Jeremiah Tenn

My program allows users to encrypt messages and words in the Ceasar cipher, ROT13 encryption (based on the Ceasar cipher), and the Vigenère cipher. The Caesar and Vigenère ciphers require a user-inputted key to encode and decode. If the user does not have a key, a brute-force decryption method is available, which finds all possibilities given an encoded string. Though these ciphers are considered weak in modern cryptography, they can still be used for fun, hiding messages that are not meant to be interpreted by the casual observer. The Caesar cipher shifts all of the letters in a message by a certain value. ROT13 encryption is a specific instance of the Ceasar cipher where the amount shifted is 13, and only 13. The Vigenère cipher is a more sophisticated cipher based off of the Ceasar cipher, where each letter is shifted by a different amount. Here, the key given by the user is another String of the same length rather than an integer.

The three algorithms that I will discuss here will be the most complex ones: the Caesar encryption, Vigenère encryption, and Vigenère brute-force decryption algorithms.

The Caesar encryption algorithm flowchart:

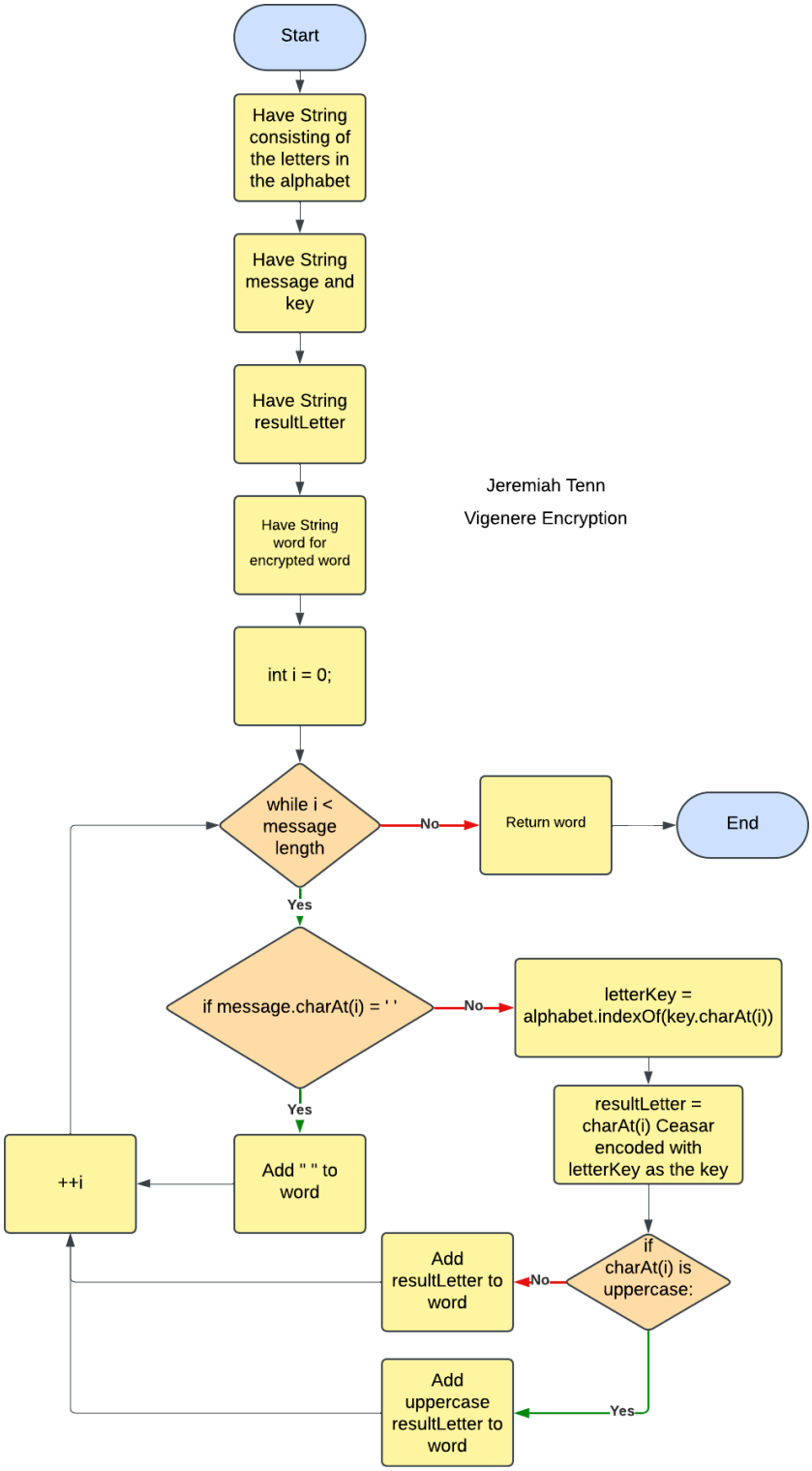


Caesar cipher encryption code snapshot:
Note: The array and HashMap are static variables, so they are not declared in the method.

```
// Returns a string encoded with the Ceasar cipher
public static String encodeCeasar(String str, int key) {
        // Base word; encoded letters are added to this
        String encryptedWord = "";

        // Fills altAlphabet so each "regular" letter would be associated with its corresponding encoded letter
        for (int i = 0; i < 26; ++i) {
                // (i + key) % 26 is used so that keys of greater than 26 are allowed
                altAlphabet.put(letters[i], letters[(i + key) % 26]);
        }

        // Goes through each character in str and puts the corresponding encoded letter in encryptedWord
        for (int i = 0; i < str.length(); ++i) {
                // If the char is a space then do not change it
                if (str.charAt(i) == ' ') {
                        encryptedWord = encryptedWord + " ";
                        continue;
                }
                // If the character is uppercase: convert it to lowercase, find the corresponding letter and convert that to uppercase
                if (str.charAt(i) != Character.toLowerCase(str.charAt(i))) {
                        encryptedWord = encryptedWord + Character.toString(Character.toUpperCase(altAlphabet.get(Character.toLowerCase(str.charAt(i)))));
                        continue;
                }

                // Character.toString used since the individual letters are of type char
                encryptedWord = encryptedWord + Character.toString(altAlphabet.get(str.charAt(i)));
        }

        // Clear altAlphabet so that it may be used for other operations
        altAlphabet.clear();

        return encryptedWord;
}
```

This method is used to encrypt a message using the Caesar Cipher. It is called when the user chooses Caesar Cipher encryption from the directory method, with a user-inputted String and integer used as arguments. The Caesar cipher shifts each letter in the String up by the integer “key”, resulting in distinct messages when the key is in the range 0-25. With key values greater than 25, the possibilities repeat themselves. Here, the calculation (i + key) mod 26 is used to get the new value of a letter, representing the letter at index i being shifted up by the value of key. Modulo 26 is used so that large keys will accurately encrypt the message. With this being the principle idea of the Ceasar cipher, the method loops through the letters in the given string, finds a new letter in a HashMap of the shifted alphabet, and adds the new letter to the string. Once all of the letters have been encoded, the string is returned. In creating the method, I utilized a static HashMap data type with the Character type for both the key and value. Populating this HashMap with the original letters as the keys and the shifted letters as the values, accessing the necessary letter during encryption is made quick and efficient. If the character is a space, a space is simply added to the word String and the for loop continues. The method has a time complexity of n time, since it grows linearly as the size of the string grows. For each additional letter in the string, one additional encryption operation is performed. This time complexity is acceptable, as each operation is very quick and huge input sizes will not be common.

The Vigenère encryption algorithm flowchart:

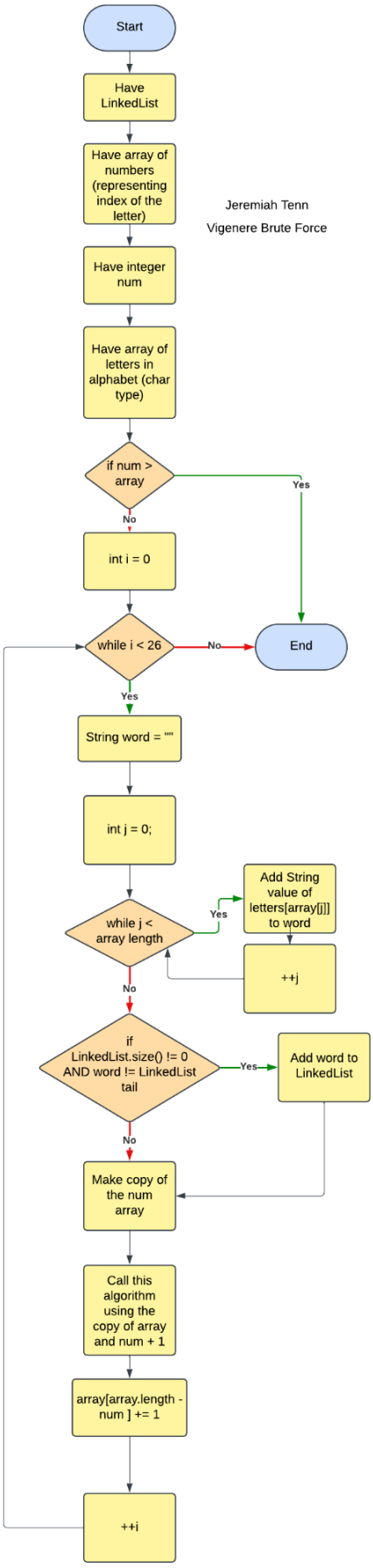


The Vigenère encryption algorithm code snapshot:
Note: This method is used for encryption and decryption due to the amount of repeated code in them. A boolean determines which operation is performed.

```
// Encodes or decodes a word or phrase in Vigenere given a key of letters the same length
public static String encodeDecodeVigenere(String str, String key, boolean encrypt) {
        String alphabet = "abcdefghijklmnopqrstuvwxyz"; // Used to locate the numerical value of each letter
        boolean encode = encrypt;                       // true if in encode mode, false if in decode mode
        String word = "";                               // String that will be returned as the encoded phrase
        String resultLetter = "";                       // Stores the encoded/decoded letter

        // Makes sure that the key is sufficient for the string message
        if (str.length() != key.length()) {
                return "Message and string to not have the same length. Check the input.";
        }

        // Goes through each letter of the word and encodes/decodes it using the key
        for (int i = 0; i < str.length(); ++i) {

                // If the current character is a space simply add a space the string and continue
                if (str.charAt(i) == ' ') {
                        word = word + " ";
                        continue;
                }

                // Gets number value of the corresponding letter in the key
                int letterKey = alphabet.indexOf(key.charAt(i));
                // Encodes or decodes the letter and stores it in resultLetter
                if (encode) {
                        resultLetter = encodeCeasar(str.substring(i, i + 1), letterKey);
                } else {
                        resultLetter = decodeCeasar(str.substring(i, i + 1), letterKey);
                }
                // Checks if the letter should be uppercase and then adds it to word
                if (str.charAt(i) != Character.toLowerCase(str.charAt(i))) {
                        word = word + resultLetter.toUpperCase(); // Converted to uppercase and added to word
                } else {
                        word = word + resultLetter; // Supposed to be lowercase so added without changes
                }
        }

        // Returns the final encoded string
        return word;
}
```

As mentioned earlier, the Vigenère cipher utilizes the Ceasar cipher, with each letter in the message having a key. A String holding the lowercase alphabet is used to find the numerical values for the letters in the key String. In the actual encryption method, letterKey is assigned with the alphabet array index of the current key letter. This is passed into the encodeCeasar() method, which uses the letterKey value for the Caesar shifting operation on the letter, which is represented as a substring of the original message. The encoded letter is added to the final word String, and once all of the letters have been encoded, this variable is returned. If the character is a space, a space is added to the word String and the encodeCeasar() method is not called. The method has a time complexity of n, similar to the Ceasar cipher methods. The time is linear; each additional letter in the string results in one additional operation in the Vigenère method. Again, this is an acceptable time complexity as the operations are very fast and the sizes of the inputs are not exceptionally large. Because it uses the Ceasar cipher methods, this method also utilizes a HashMap data structure.

The Vigenère brute force decryption algorithm flowchart:



The Vigenère brute force algorithm code snapshot:
Note: The LinkedList and array of numbers are static variables, so they are not declared in this method.

```
public static void recursiveVigenere(int[] array, int num) {
        // If the index number is greater than the array (resulting in a -1 index) then stop the current branch of recursion
        if (num > array.length) {
                return;
        }

        // for each possible letter combination:
        for (int i = 0; i < 26; ++i) {
                String word = "";
                // Uses the contents of the array to create a String key
                for (int j = 0; j < array.length; ++j) {
                        // Finds the associated letter wth the current number int the array; adds it the the word string
                        word  = word + Character.toString(letters[array[j]]);

                }

                // Places conditions for adding the word combination:
                // Add the word as long as the LinkedList is empty and not the same as the last node (we do not want duplicates)
                if (combo.size() == 0 || !word.equals(combo.getLast())) {
                        combo.add(word);
                }

                // Create a copy of the array to be used in the recursive function; otherwise the array will be changed for all of the running threads
                int[] arrayCopy = new int[array.length];
                for (int j = 0; j < arrayCopy.length; ++j) {
                        arrayCopy[j] = array[j];
                }

                // Run this function again using the copied list, except using the n-th + 1 to the last letter
                recursiveVigenere(arrayCopy, num + 1);
                // Adds one to the current index (letter) so that the next letter will be used.
                array[array.length - num] += 1;
        }
}
```

The Vigenère brute-force method is one of the more complex methods in my program. First of all, in order to generate all possible combinations using this cipher, the method must function recursively. After each possibility is generated, that specific possibility is again used when the method runs recursively. The method takes an array of integers and an integer called num in as arguments. The array has the length of the desired message to decrypt, and initially has zeros in each of its indexes. Basically, the array initially reflects a string the same length as the original one, except with only the letter “a”. This method must find all possible letter combinations given only the length of the string, so all possible letter combinations must be found. This is where the num argument’s purpose comes in. The possibilities begin with the *last* letter (or index) of the string, and the recursive methods work towards the first letter. The num integer is subtracted from the length of the array to find the current index that is being examined. Starting with 1, this num variable eventually increments until it is greater than the length of the array; at this point the method stops. Basically, for each possible letter from the right to the left, the method calls itself to find all of the possible letters of the next leftmost index – keeping the letters that have already been found. For each set of possible letters, a String is constructed by finding the letters associated with the numbers in the array. If this String is not equal to the last one (or if the LinkedList is empty), then add the possibility to the static LinkedList variable. This prevents duplicate combinations in the final list. Because of the recursive nature of this algorithm, it has an exponential time complexity of 26^n. This is because each additional letter has 26 possibilities,

and with each additional letter, there will be 26 times more combinations and operations. Though this is very slow and inefficient, especially for lengths of longer than 4, the method works as a brute-force operation. Finding all combinations of large sets will always result in massive sizes, so even though it may be very slow, it still is functional and fulfills its task.

Three of the data structures that I used in my program were the HashMap, LinkedList, and array. I chose the HashMap for the Caesar cipher encryption and decryption because it allowed for key value pairs. This feature was necessary for setting each of the letters in the alphabet with the corresponding shifted letter. The HashMap data structure also allows for quick retrieving of the data. It is also space saving, since it is cleared after each Caesar Cipher operation, thus using only one HashMap instance for all of the operations. If the user has many requests, no additional space is taken up by the HashMap. I chose the LinkedList type for the Vigenère algorithm for a similar reason, since it can be easily cleared to save space. Another reason why I choose this specific type of list is how it does not have a set size. Because the amount of results varies on the length of the original string, size adaptability is very important. Only one instance of a LinkedList is used in the whole program regardless of the number of operations because it is cleared to a blank state after each is finished. Arrays are used in my program to hold sets of non-changing variables, including the alphabet of characters and the array of numbers for Vigenère brute-force decryption. These situations do not require the array to change its size, so a LinkedList is not used. Additionally, arrays are easily parsed for searching for and copying values.

I started writing the program that first implemented the Caesar cipher encryption method. I initially constructed the method so that a binary search is used to find the index of the letter in the alphabet. However, I soon implemented a static HashMap where each letter of the alphabet is associated with its shifted value. This was more efficient and easier and made the code cleaner. I then implemented the Caesar decryption method, which was very similar to the encryption method. The crucial difference was that the HashMap was populated with the shifted letter as the key and the regular letter as the value. One problem that occurred was that errors arose when I tried to add letters to the final String. This was because String objects are immutable, meaning that they cannot be changed. To successfully add the letter, I had to create a new string that consists of the letter added to the String that we already have. This is then stored under the same name as the original word. After I finished the Caesar methods, I added the ROT13 methods. These were simply calling the Caesar encryption and decryption methods with 13 being the default and only key. The Vigenère encryption and decryption methods were created next. As I coded the encryption method, I realized that the decryption method would have almost exactly the same code, so I added a boolean parameter that would indicate what operation should be used. Inside the method, I declared a String with the letters of the alphabet, so the letterKey variable would easily be assigned with the index of a specific letter using the String.charAt() method. Using this key and a specified letter in the Caesar encryption/decryption methods

returns the encrypted letter, which was then added to a String representing the word. There were no particular errors in implementing this method. The next methods, bruteForceDecodeVigenere() and recursiveVigenere() methods were difficult to implement properly. In order to generate all of the possible combinations of a word, I had to use a recursive function. The bruteForceDecodeVigenere() method sets up an array of the same length as the given string. This is then passed into the recursive function with the integer argument of 1. This method was difficult because it took me a long time to figure out exactly what I needed to do, but writing notes and drawings helped me to understand my task. Once I actually completed the first incarnation of the recursive method, I noticed that there were more results than there were supposed to be. This was because some words were generated duplicate times. To solve this, I required either the LinkedList to be empty or the word to be unequal to the tail of the LinkedList in order to be added. This resulted in the correct amount of combinations in the LinkedList. The last method that I created was the directory() method, which prompted the user to pick a choice and then called the method that the user wishes to use. Here, I used a switch statement to easily direct the flow of the program based on the value of the integer user input. However, I came across NoSuchElement exceptions among other errors. These were a result of having user input returned from scanner.nextLine() after a scanner.nextInt(). The error arose because the nextLine() method took in the newline character from the nextInt() line. In order to read the newline character, I placed a scanner.nextLine() method that does not return its results. Following scanner input worked as expected. In order for the user to make multiple choices, the directory() method calls itself after either the chosen operation is finished or the user uses invalid input. The program ends when the user writes that they want to quit.

In my next version of this cipher program, I would add support for a multi-word phrase in the Vigenère brute-force method. In development, I could not find an efficient and accurate way to do this, but given more time this task would be possible. I would also add support for more complex ciphers, such as the Playfair cipher.