**Algorithm Study Template**

**Algorithm**: Caesar’s Cipher

**aka**: The Shift Cipher, Caesar’s Code, Caesar Shift

**Techniques**: Substitution Cipher

**Categories**: Cryptography

**Problem**: In ancient times, Julius Caesar required a way to send confidential messages, especially those concerning military matters, without them being able to be read by an enemy if intercepted. To prevent this, he used a cipher to change the ordering of the letters of the alphabet. Different shifts are possible, but Caesar shifted three letters to the left, so that ‘D’ would become ‘A’, and so on.

**Applications**: Caesar used the cipher to send encrypted messages, but it has very different uses today. For example, Caesar ciphers can be found in children’s toys such as secret decoder rings. A Caesar shift is also performed in the ROT13 algorithm, a method of obscuring text that is sometimes found obscuring joke punch lines and story spoilers on discussion boards by shifting all the letters by thirteen.

**References**:

* <http://mathworld.wolfram.com/CaesarsMethod.html>
* <http://www.murky.org/2004/09/the-caesar-shift/>
* <http://www.princeton.edu/~achaney/tmve/wiki100k/docs/Caesar_cipher.html>
* <http://www.nku.edu/~christensen/section%202%20caesar%20ciphers.pdf>
* <http://practicalcryptography.com/ciphers/caesar-cipher/>

**Implementation details**:

* **Big Idea**: Iterate through a string of text (the un-encoded message) and shift each letter down one side of the alphabet according to a specified shift value. For example, with a shift of negative three, ‘D’ becomes ‘A’. With a shift of 12, ‘F’ becomes ‘R’. Thus, negatives values shift to the left through the alphabet and positive values shift to the right. The resulting string, with all letters shifted, is the encoded message.
* **Description**: The documentation I found for the cipher does not handle spaces or punctuation in the un-encoded message, so I modified the algorithm to handle these cases. The cipher iterates though the un-encoded message, checking each character in the string to see if it is a space, punctuation mark, or other character, which is handled as an alphabetic letter. The allowed punctuation marks are period, comma, exclamation mark, and question mark. I chose these punctuation marks because they were the ones I thought would most likely occur in a message to be encoded.

If the current character is a space or punctuation mark, it is simply appended into the encoded message without being modified. Other characters (once again, assumed to be alphabetic letters) are handled by first finding their position in the English alphabet. This is done by taking the current letter and locating its index in a string containing the English alphabet. Then, the position of the corresponding encoded character is found by taking the sum of the alphabetic position of the un-encoded character and the specified shift value and taking that sum modulus twenty six. The modulus twenty six is to ensure that we don’t have with alphabetic indices greater than what we can look up in the alphabet string. For example, if the letter is ‘X’, the alphabetic index would be twenty-three (range of 0-25). If the shift value is three, the sum of the alphabetic index and the shift would be twenty-six. Obviously, this is outside the range of alphabetic letters. So, we take twenty-six modulus twenty-six and end up with an index of zero. This leads us to find the encoded letter ‘A’ inside the alphabet string.

If we are shifting left with a character toward the beginning of the alphabet, we could have a negative alphabetic index. To combat this, if the index is found to be less than zero, the length of the alphabet (26) is added to the index. For example, if we are shifting ‘B’ (index 1) by a value of negative five, the index becomes negative four. When we add twenty-six, we get an index of twenty-two, which is ‘W’. One may wonder why it is necessary to check for this condition when the encoded index is calculated using modulus. The best answer I can give is that there are different definitions of the modulus operation, and the version which Java runs with the ‘%’ operator doesn’t automatically add the mod value to a negative result. Thus, I added my own check.

At this point, we look up the encoded letter by its calculated position in the alphabet and append it into the encoded message. This process is repeated until the entire un-encoded message has been iterated through. The result is a string containing the encoded message, which is then returned.

* **Pseudo-code**: \*Note that in my actual code I used case-statements instead of an if-else for the different character possibilities. The pseudo-code here is logically the same.

**function** caesarEncrypt(message,shift){

//create alphabet string

String alphabet = “abcdefghijklmnopqrstuvwxyz”

//iterate through message

for(i = 0; i < length of message; i++){

if(current character == space, comma, period, exclamation mark, or question mark){

append character into encoded message

}else{

charPos = index of current letter in alphabet string

keyVal = (shift + charPos) % 26

if(keyVal < 0){

keyVal = length of alphabet string + keyVal

}

encVal = letter at index keyVal in alphabet

append encVal into encoded message

}

}

return encoded message

}

* **Specific implementation**: (see CaesarCipher.java)

**Correctness**:

**Theoretical**: The shift can be described mathematically as e(x) = (x + k) mod 26, where x is the index of the value we’re encrypting and k is the shift value applied to that index. As mentioned in the “description” section above, the modulus operation ensures that the calculated index of the encoded letter is not greater than the range of letters in the alphabet (0-25). Also, as previously mentioned, the check for indices less than zero and subsequent addition if found handles cases where left shifting causes us to circularly move from the beginning of the alphabet to the end (such as when encoding ‘B’ to ‘W’ with a shift of -5).

**Empirical**: To ensure that the message being input by the user is acceptable for use with the cipher, I ran a few checks on the input prior to encoding. First, if the message contains numeric values, the user is notified of unacceptable input and the program is terminated. This is because the cipher is only designed for use with alphabetic letters. Next, I make sure that the user did not enter an empty string, with the same consequences as a numeric value if found. This is an unlikely edge case, but I didn’t like the idea of getting a message containing nothing and giving back an encoded version that also contains nothing. The text of the message can be upper-case or lower-case when input because the cipher will convert it to upper-case anyway. I wrestled with the decision over which case to use for a while, and ultimately decided that most people have an easier time reading upper-case as output. Note that I have not handled cases where a character other than an alphabetic letter, number, space, period, comma, exclamation mark, or question mark is used as the message to be encoded. This means that characters such as ampersands or number signs would break the program, but it’s something I accepted because the user is given explicit instructions about what to input and the program itself is only intended for use with simple text messages.

The user is also prompted for a shift value to use with the cipher. The only check I ran on this input is that the value received is an integer. This means that shifts of zero are permitted, but I’m allowing it because it doesn’t break the program and isn’t likely to be used often. The try-catch statement that does the integer check also ensures that a value must be entered before running the cipher. However, if the shift value entered is not an integer, the user is notified of unacceptable input and the program is terminated.

In order to properly test the encryption, I decided to write a method for decryption. My thinking was that if I encrypt with a certain shift value, then I should be able to decrypt with the same shift value and get my original message back. The decryption method was simple to write and is very similar to the encryption method. The key difference is that calculating the index of the un-encoded character (as opposed to the encoded character before) requires taking the difference of the alphabetic position of the encoded character and the shift value modulus twenty-six instead of the sum modulus twenty-six. Besides that, the methods are essentially the same, which is why I didn’t see a reason to discuss them as separate algorithms in this report. Instead, I used the decryption method to provide the original message back as proof that the encryption worked.

My test results are as follows:

Test 1:

Your message: There are one thousand centurions moving to attack the enemy!

Cipher shift: -3

Encrypted message: QEBOB XOB LKB QELRPXKA ZBKQROFLKP JLSFKD QL XQQXZH QEB BKBJV!

Original message: THERE ARE ONE THOUSAND CENTURIONS MOVING TO ATTACK THE ENEMY!

Test 2:

Your message: Is it just me, or does Brutus seem kind of shifty?

Cipher shift: 5

Encrypted message: NX NY OZXY RJ, TW ITJX GWZYZX XJJR PNSI TK XMNKYD?

Original message: IS IT JUST ME, OR DOES BRUTUS SEEM KIND OF SHIFTY?

Test 3:

Your message: Tomorrow, we will cross the Rubicon.

Cipher shift: 13

Encrypted message: GBZBEEBJ, JR JVYY PEBFF GUR EHOVPBA.

Original message: TOMORROW, WE WILL CROSS THE RUBICON.

(These are just a sampling of possible messages that utilize all the punctuation marks I accounted for. For authenticity, I attempted to encrypt messages that Caesar himself might have sent.)

**Performance**:

**Theoretical**: The Caesar Cipher is of linear complexity, O(n), because it is based on a single for-loop and spends close to the same amount of time processing each character in the un-encrypted message.

**Empirical**: The following results are the execution times of the encryption and decryption methods. No printed output or other “noise” is intentionally included in the measurements. The test results shown in this report, as well as others I ran during development, indicate that encryption usually takes longer than decryption. There are two reasons I can think of as to why that would be. The first is that when encrypting, the characters have to all be changed to upper-case. When decrypting the output of the encryption, the characters are already upper-case, and do not need to be changed. The second reason is that when decrypting, the calculated index value for the un-encrypted character doesn’t hit the “below zero” condition as often, and thus less code is executed. These are just my observations, but I think they have some validity.

My test results are as follows:

Test 1:

Encryption finished in: 0.73 milliseconds.

Decryption finished in: 0.475 milliseconds.

Test 2:

Encryption finished in: 0.718 milliseconds.

Decryption finished in: 0.366 milliseconds.

Test 3:

Encryption finished in: 0.499 milliseconds.

Decryption finished in: 0.228 milliseconds.

(These measurements are taken from the same tests in the “empirical correctness” section of this report)

**Anecdotes**: It is not known how effective this cipher was in Julius Caesar’s time, but it is likely to have worked quite well against his many illiterate enemies. Even a more sophisticated enemy would likely have assumed that the text was written in a foreign language.

**History**: As previously mentioned, this cipher was used by Julius Caesar to send encrypted messages, particularly those of military importance. Caesar always used a shift of negative three when encoding, but his nephew, Augustus, preferred a shift of positive one. Over time, the basic idea of the cipher has remained the same, but different users have typically employed different shift values. As recently as 2011, a man in the United Kingdom was convicted of “terrorism offenses” for using the Caesar cipher to communicate with Bangladeshi Islamic activists to discuss blowing up British Airways planes or disrupt their IT network.

**Variations**: As previously mentioned, the only real variation that’s been seen in the Caesar cipher is different shift values being utilized.

**Alternatives**: The Vigenere cipher is a series of Caesar ciphers that is used to encrypt alphabetic text. It could be used in the same contexts that a Caesar cipher would be suited for, but is considered to be more secure.

**Credits:**

* <http://rosettacode.org/wiki/Caesar_cipher>
* <http://en.wikipedia.org/wiki/Caesar_cipher>
* <http://en.wikipedia.org/wiki/Substitution_cipher>