


Lab 5 - Veni, vidi, vici

Due on 2/26/2024 at 11:59PM


Introduction:

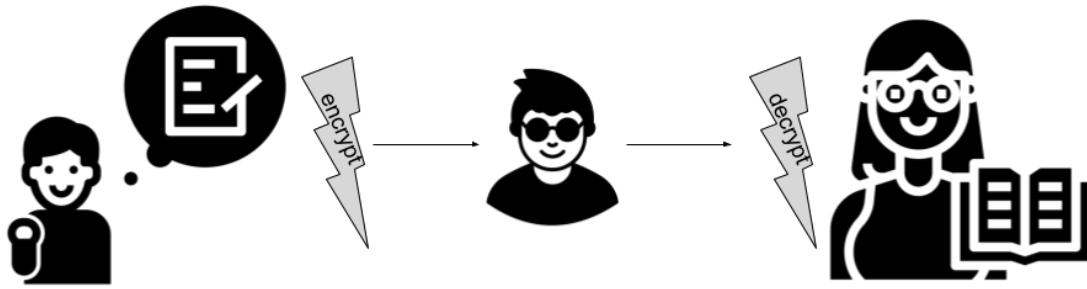
Since the beginning of time, people have sent messages that they only wanted the intended recipient to understand. To maintain privacy, sometimes the speaker would tell the message to a trusted courier who would deliver the message to the speaker's counterpart. Sometimes the writer would mail a letter and seal it with a wax stamp to indicate to the reader whether the message had been opened in transit. No matter the mechanism, there have always been people who wanted to keep certain communications secret.

And, for as long as there have been secrets, there have been people dedicated to exposing them. The battle between the coy and the nosy has been fought back and forth throughout the ages. Whether by hiding the truth or using trusted intermediaries, early furtive senders would rely on surreptitiousness to keep a message secret. The technique of "**security through obscurity**"  (https://en.wikipedia.org/wiki/Security_through_obscurity) did not hold up long when exposed to dedicated adversaries.

What the secretive sender needed to communicate with the reluctant reader was a method of communicating that could protect the information whether a spy knew the means for its protection or not. In other words, the sender and receiver had to assume that the spy knew the method by which they were keeping a secret and still be able to communicate privately. The need to communicate privately under these conditions led to the development of *encryption*.

Encryption/decryption methods are known as *schemes*. In an encryption/decryption scheme, there is an algorithm for taking the original message (the *plaintext*) and transforming it to something that an eavesdropper can't understand (the *ciphertext*). This process is known as encryption and is performed by the sender. In an encryption/decryption scheme, there is a corresponding algorithm for taking the *ciphertext* and reversing the encryption -- the process is known as *decryption* and is performed by the receiver.

Encryption/decryption schemes are publicly known, well documented and **standardized**  (<https://csrc.nist.gov/Projects/cryptographic-standards-and-guidelines>). If the methods of encryption and decryption are known, why can't the sender's foes use that information to recover the plaintext? Because encryption/decryption schemes usually rely on the sender and the intended recipient sharing some data that is used to *configure* the encryption/decryption scheme in a particular way. After configuration, the *ciphertext* the scheme generates cannot be recovered by knowing only the scheme's algorithm -- the attacker must also know the configuration! The data shared between communicating parties that can be used to configure the encryption/decryption scheme is called a *key*.



Combining a good encryption/decryption scheme with a key to protect a message means that the sunglassed snooper has no way of knowing the contents of the message *even if they know the algorithm used to protect the message*.

One of the earliest encryption schemes was employed by **Julius Caesar** [↗](https://en.wikipedia.org/wiki/Caesar_cipher) (https://en.wikipedia.org/wiki/Caesar_cipher) to keep his messages safe. A type of *substitution cipher*, in the Caesar Cipher, the sender and receiver agree on the key -- a shift number and direction (left or right) -- and, to encrypt, the sender shifts each letter in the plaintext in the chosen direction and by the number of spaces indicated by the shift number to create the ciphertext. In the opposite direction, to decrypt, the receiver shifts each letter in the ciphertext in the opposite direction and same amount to restore the plaintext. It is called a substitution cipher because each letter in the plaintext is substituted for a letter in the ciphertext.

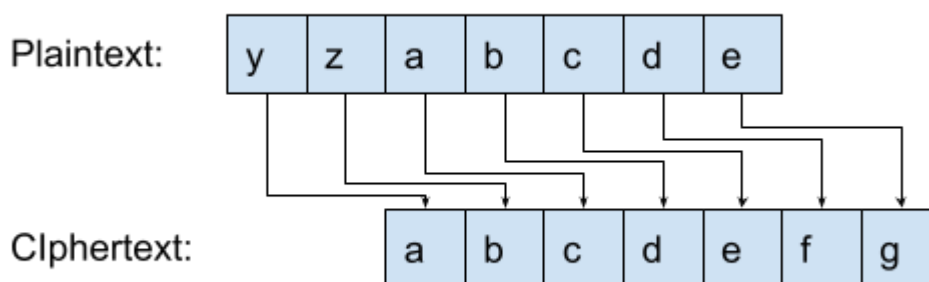


Figure 1: Substituting characters in plaintext for ciphertext when using a key with a shift number of 3 and right direction.

An example will help!

Assume that I want to communicate with my parents. We would agree that I would encrypt my messages by shifting each letter 5 (the shift number) characters to the right (the direction). I would encrypt the message "hello" as "mjqqt". Why? Because m is five letters to the right of h in the alphabet, j is five letters to the right of e in the alphabet, and so on. To be able to read the secured message, my parents would shift each letter of the ciphertext to the left by 5 and recover the original plaintext. Hello, indeed.

Implementing the Caesar Cipher is done with modulo arithmetic and by equating each letter with a number according to its position in the alphabet. Assume that we have a function named `number_from_letter` that turns a letter into its position in the alphabet: a is 0, b is 1, c is 2, etc. For example, if we call `number_from_letter` with the argument d, the return value is 3.

Assume that we have another function named `letter_from_number` that turns a number between 0 and 25, inclusive, into a letter from the alphabet: 0 is a, 1 is b, 2 is c, etc. For example, if we call `letter_from_number` with the argument 1, the value of the expression is b. We can use these functions to come up with the ciphertext given the key and the direction.

What is the general process for encrypting a single character? Let's use a variable, *c*, to represent the plaintext letter that we are encrypting. **Note carefully that we are using *c* as a variable to represent any character.** We will name the result of a call to `number_from_letter` with the argument *c* the *index* of the plaintext letter. If the direction of encryption is *right*, we will *add* the shift number from the key to the index of the plaintext letter. That gives us a number that represents the position of the ciphertext character in the alphabet, what we will call the *index* of the ciphertext character. To get the actual ciphertext letter, we call the `letter_from_number` function with the index of the ciphertext letter as the argument. Voila! The result of that function call is the enciphered character.












If the direction of the encryption is *left*, we will *subtract* the key from the index of the plaintext letter. Otherwise, the process is the same!

That works fine for *most* cases. However, there are two corner cases that we have to consider.

First, what happens if the index of the ciphertext character is past the end of the alphabet? For example, if we attempt to encrypt z with a shift number of 1 and a direction of right? We *know* that we want the ciphertext letter to be a. But, how do we calculate it algorithmically? Using the algorithm from above, we would calculate the index of the ciphertext character as 26 in this case. However, we can't call `letter_from_number` with an argument of 26 because our numbering scheme equates 25 with z! Despair! Wait: we can use our friend the modulus operator! Taking the index of the ciphertext character and "modding it" by 26, the number of characters in the alphabet, will solve the problem. In our example, $26 \% 26$ is 0 and that becomes the updated index of the ciphertext character. Using the updated index of the ciphertext character as the argument in the call to `letter_from_function` means, in our working example, that the ciphertext character is a. Just what we wanted!


Second, what happens if we calculate an index of the ciphertext character that is less than 0? For example, if we attempt to encrypt the letter a with a key of 1 and a direction of left? Intuitively, we know that we want the ciphertext character to be z. But, how do we write a process to do it? Using the algorithm from above, we would calculate an index of the ciphertext character of -1 in this case. Uh oh, there is no letter of the alphabet equated with the number -1. To solve the problem, when the index of the ciphertext character is negative, let's overwrite it with an updated index of the ciphertext character calculated by subtracting the old index from the number of letters in the alphabet (26). Therefore, in the working example, the index of the ciphertext character becomes 25 ($26 - 1$). Using that updated index of the ciphertext character as the argument to `letter_from_function` means that the ciphertext character is z! Bingo!

Practice the algorithm using paper/pencil by filling in the blank entries in the following table:

Plaintext Character (p)	p's position in the alphabet	Shift Number (sn)	Shift Direction	C's Position in the alphabet	C's Position in the alphabet (% 26)	C's Position in the alphabet (account for negatives)	Ciphertext Character
b	1	5		1  sn=6	$6 \% 26 = 6$	6	g
x	23	9		23  sn=32	$32 \% 26 = 6$	6	g
d	3	7		3  sn=-4	$-4 \% 26 = -4$	$-4 + 26 = 22$	w
m	12	200		12  sn=-188	$-188 \% 26 = -6$	$-6 + 26 = 20$	u
t		500					
c		13					
h		47					

When you practice with this chart, feel free to use the [PDF version](#)

(<https://uc.instructure.com/courses/1657742/files/177496673?wrap=1>) -- it'll be a good thing to have handy when you make your submission for the lab!

In this lab, you will develop an encryption program named *1d3s* to read in a key and plaintext message from a file and print the encrypted version which will be safe from **prying eyes** 
(<https://www.youtube.com/watch?v=2PTEqZURh4o>).

As always, read the entire lab document before beginning!

Good luck!

Program Design Task:

As my dad always said, "If you want something done wrong, you have to do it yourself." Before you start writing code, please write the pseudocode or draw a flow chart for your implementation of the *1d3s* application.

Program Design Requirements:

Your pseudocode or flow chart must describe the entirety of the solution. You may choose to write the flow chart or the pseudocode at any level of detail but remember that this is a tool for you!

Your pseudocode or flow chart must include a description of how you plan to

1. Read in the encryption key (shift number and direction) and plaintext from a file,
2. Handle any errors in the formatting of the input file,
3. For each character in the plaintext
 1. Calculate the index of the plaintext character,
 2. Calculate the index of the ciphertext character,
 3. Handle situations where the index of the ciphertext character is longer than the alphabet,
 4. Handle the situation where the index of the ciphertext character is shorter than the alphabet,

5. Convert the (possibly adjusted) index of the ciphertext character to the actual ciphertext character, and
4. Print the ciphertext to the screen.

In addition to submitting pseudocode for this lab, you *must* submit a completed version of the chart from the Introduction. Your skeleton code for the lab has a fillable PDF or you can download it [here \(https://uc.instructure.com/courses/1657742/files/177496673?wrap=1\)](https://uc.instructure.com/courses/1657742/files/177496673?wrap=1).

Programming Task:

Your programming task is to implement the *1d3s* application. In this lab you will not prompt the user for input. You will read all input from a file named `input.txt`. Like Lab 4, a skeleton file is provided (below) and you *must* use the skeleton.

The input file will be formatted like so:

<`l` or `r`><shift number (an integer)><first character of plaintext><second character of plaintext>...<last character of plaintext>

The first character is the direction (`l` for left and `r` for right) and the number is the shift number. For example, an input file might contain:

```
l4hello
```

which would tell *1d3s* that

1. the encryption direction is left;
2. the key is four (4); and
3. the plaintext is `hello`.

Your program will calculate the ciphertext and print it to the screen on a single line. For example, if you run *1d3s* with an input file whose contents are

```
l4hello
```

the output would be

```
dahhk
```

As another example, if you run *1d3s* with a file whose contents are

```
l28goodbye
```

the output would be

```
emmbzwc
```

The *1d3s* program must also handle input files that are improperly encoded. If there is no direction provided, *1d3s* must output

Oops: Could not read the direction from the input file.

If the first character in the file is not an **l** or an **r**, 1d3s must output

Oops: Invalid direction in the input file.

If there is no shift number provided, 1d3s must output

Oops: Could not read the shift number from the input file.

You may test your code using these test input files: **one**

(<https://uc.instructure.com/courses/1657742/files/177485799?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485799/download?download_frd=1) , **two**

(<https://uc.instructure.com/courses/1657742/files/177485797?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485797/download?download_frd=1) , **three**

(<https://uc.instructure.com/courses/1657742/files/177485795?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485795/download?download_frd=1) , **four**

(<https://uc.instructure.com/courses/1657742/files/177485793?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485793/download?download_frd=1) , **five**

(<https://uc.instructure.com/courses/1657742/files/177485789?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485789/download?download_frd=1) . Don't worry, you also have these files in your skeletons!

To check whether your program is operating correctly, you may want to use one of these online

Caesar → (<https://www.dcode.fr/caesar-cipher>) **Cipher** → (<https://cryptii.com/pipes/caesar-cipher>) **implementations** → (<http://practicalcryptography.com/ciphers/caesar-cipher/>)_.

Note: Your program will be tested against other test cases. Your program must compute properly in all cases in order to receive full points! Check the output of the autograder to make sure that your program behaves as expected. Read the autograder's output carefully!

Programming Requirements:

Start with **this skeleton** (<https://uc.instructure.com/courses/1657742/files/177391353?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177391353/download?download_frd=1) if you are a Windows-based developer. If you are a macOS-based developer, use **this skeleton**

(<https://uc.instructure.com/courses/1657742/files/177391473?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177391473/download?download_frd=1) . If you are a Linux-based developer, start with **this skeleton**

(<https://uc.instructure.com/courses/1657742/files/177485773?wrap=1>)_ ↓

(https://uc.instructure.com/courses/1657742/files/177485773/download?download_frd=1) code. The skeleton code provides the functions you will need to use to successfully complete this lab. If you do not use this skeleton code you will not be able to complete this lab. The skeleton code contains two useful functions:

- `number_from_letter`: This function takes a single parameter, a character. The output from the function is an integer according to the equation between number and letters described in the introduction (above).
- `letter_from_number`: This function takes a single parameter, an integer. The output from the function is a character according to the equation between number and letters described in the introduction (above).



You may make the following assumptions:

1. The file `input.txt` will always be available and readable.
2. The plaintext will be entirely lowercase.
3. The plaintext will only contain valid alphabetic letters.
4. The plaintext will contain no numbers.

You must use the provided constant anywhere that you need to use 26, the number of letters in the alphabet.

Getting Started:

Let's think about how we could get started! First, let's explore how to use some of the functions that we have been given. To do that, let's code up the first row from the practice table. As a reminder, that row helped us walk through the steps of encrypting the letter `b` with a shift number of `5` and a *right* shift direction.

Plaintext Character (p)	p's position in the alphabet	Shift Number (sn)	Shift Direction	C's Position in the alphabet	C's Position in the alphabet (% 26)	C's Position in the alphabet (account for negatives)	Ciphertext Character
b	1	5		1  sn=6	6 % 26 = 6	6	g

In our main function, let's write some code!

First, we will start by declaring some important variables that we will need to use. For our input, let's declare a variable for the plaintext character that we will encrypt (a `char`), the shift number that we will use (an `int`) and the shift direction (a `char`):

```
int main() {
    char plaintext{ ' ' };
    int shift_number{ 0 };
    char shift_direction{ ' ' };
}
```

Great! Now, let's add code to update those variables so that they hold the values that we want to use:

```
plaintext = 'b';
shift_number = 5;
shift_direction = 'r';
```


From the practice chart, we know that `b`'s position in the alphabet is `1`. How can we get that value programmatically? Well, we have a function named `number_from_letter` that appears to perform that very translation! If we pass a character as an argument, we will get back its spot in the alphabet. The spot in the alphabet is a number so, of course, the return type of that function is `int`. If we want a variable to hold the result of that function call, we would give it a type of `int`. Because it holds the position of the plaintext character in the alphabet, a good variable name might be `plaintext_position`:

```
int plaintext_position{number_from_letter(plaintext)};
```

Fantastic!

Now comes the first part of our code where we will have to *select* which piece of code to execute based on the contents of a variable. If we want to select which piece of code to execute based on a condition, I bet that we will want to use a *selection statement*! According to our encryption algorithm, if the shift direction is left, then we will subtract the shift number from the plaintext character's position in the alphabet. On the other hand, if the shift direction is to the right, then we will want to add the shift number to the plaintext character's position in the alphabet.

Let's try a fun trick! Let's declare/initialize a variable named `actual_shift` that will hold the number/direction of the actual shift. What do I mean by "actual shift"? Well, let's assume that we want the actual shift value to be a variant of the shift number that will accomplish the shift using addition, no matter whether the direction is right or left. Well, if the given shift direction is right, then there's no need to modify the shift number -- according to the algorithm we would want to add the shift number to the plaintext character's position in the alphabet. On the other hand, if the given shift direction is left, then we will need to make the given shift number negative -- according to the algorithm we would want to subtract (i.e., *add the negative*) the shift number from the plaintext character's position in the alphabet.

To write our code succinctly, we will start by making an assumption that the given shift direction is right. If that were the case, then the declaration/initialization of our `actual_shift` variable would look something like:

```
int actual_shift{shift_number};
```

Great! But, you know what happens when we assume. So, let's write some code that will selectively update `actual_shift` to a different value when reality does not match our assumption:

```
if (shift_direction != 'r') {  
    actual_shift *= -1;  
}
```



```
}
```

How cool is that? When our program has passed the `if` statement, we are guaranteed to have a value that we can *always* add to the plaintext character's position in the alphabet to find the encrypted character's position in the alphabet:

```
int encrypted_position{plaintext_position + actual_shift};
```

We are so close to what we want! But, there are a few corner cases that we need to consider. The encryption algorithm specifies that we must guarantee that the encrypted position corresponds to an actual character in the alphabet, even if that value happens to be greater than or less than the number of characters in the alphabet. If the position of the encrypted character ends up being greater than the number of characters in the alphabet, we will have to wrap around back to the beginning of the alphabet. As described above, we can accomplish that with the modulus operator. Remember from an early lecture that we said we can write down the modulus operation in C++ using the `%` operator? Watch how we use that in practice:

```
encrypted_position = encrypted_position % 26;
```

Note: Aren't there 26 letters in the alphabet? Why are we using 26? Doesn't 26 look like a magic number? Is there a constant given to us in the skeleton code that we can use instead of that magic number? Is it safe to always perform the modulus operation? Or, should we only do it when the `encrypted_position` is greater than the number of letters in the alphabet? All good questions!

I can taste success, but we are still a few steps away. The second caveat to consider is an encrypted position that is negative. In that case, our encryption algorithm specifies that we again need to wrap around. This time, though, we will have to wrap around to the end of the alphabet. We can achieve that mathematically by simply adding 26 to a *negative* encryption position. Should we wrap around if our encryption position is positive? No! In other words, we will need to wrap around selectively:

```
if (encrypted_position < 0) {  
    encrypted_position += 26;  
}
```

Nicely done! We are at the point now where our encrypted position holds the position in the alphabet of the encrypted character, even after we consider all the corner cases. All we need to do is convert that position to the character. What function are we given that will do exactly that? The `number_to_letter` function looks like a great candidate!

```
char encrypted{letter_from_number(encrypted_position)};
```

Wow! It took a while, but we did it! Sprinkle a few `std::cout` s through your code and run it! Does it work like you expect? Change the values of `plaintext`, `shift_number` and `shift_direction` to match the values in the practice sheet and see if you get what you expected. Does your code need to be adjusted? Does your practice sheet need to be adjusted?

Just Getting Warmed Up

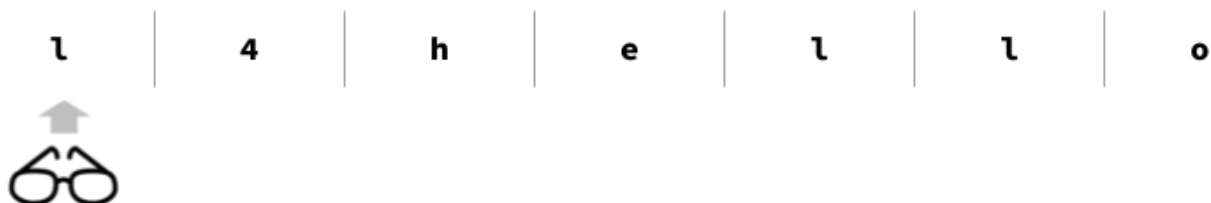
The code we wrote above is fantastic! However, it would be nice if our application worked the way that it was specified -- in other words, it would be great if it read in the encryption specifications and the data to encrypt from a file. How could we do something like that?

Well, the first thing that we will have to do is open the input file so that we can read it! Our specifications say that the input data is always stored in a file named `input.txt`. We should write code that declares a `std::ifstream` variable and then associate that variable with a file named `input.txt`.

We don't need to check whether the file could be opened successfully -- I told you that you can safely assume that file is available and readable!

The first bit of information that should be in a properly formatted input file is the shift direction.

input.txt:



The shift direction is a character and we already have a variable to hold that (`shift_direction`). So, we should use the `>>` operator and the `std::ifstream-typed` variable (from above) to read the shift direction from the file into the variable!

This is really easy, right? Well, not so fast! We have to check for two things: First, what would happen if the file we opened doesn't have any data in it at all? In that case, our read from the file into the `shift_direction` would fail. To detect and handle that condition, we should probably wrap the statement that we just wrote in an `if` statement.

Second, even if we were able to successfully read a character from the file into the `shift_direction`, it could be something other than `l` or `r`, the only two valid values. To detect and handle that condition, we should probably write another `if` statement!

Let's assume that we got a valid shift direction from the input file.

input.txt:

The next step is to read the shift number! We already have a variable that will hold the shift number (`shift_number`) so we can use our `>>` operator to read from the file into that variable. Again, however, we need to make sure that the file contained an integer where we expected it! We should probably use another `if` statement!

If everything has succeeded to this point, where should the read head be positioned in our input file?

input.txt:

That's right, it should be pointing to the first character of the text to be encrypted! Let's use our `>>` one more time to read in the plaintext character into our plaintext variable.

Given the `shift_direction` and `shift_number` that we have already read (and validated), we know have enough information to use the code that we already wrote to encrypt that plaintext character. The final step is to add some code that will make our program repeatedly execute the steps of the encryption until there are no more characters to encrypt! That sounds like a job for a while loop!

Critical Thinking Task:

We talked in class about an interesting property of the `for` loop: Every `for` loop can be rewritten as a `while` loop. How cool is that? Your Critical Thinking Task is to describe (graphically or textually) a mechanical (algorithmic) process for converting between `for` loops and `while` loops. You may use any means you choose to describe the transformation. Bonus points may be awarded for creative presentations. You may use the following terminology to refer to different parts of the `for` loop when describing the conversion:

```
for (initialization statement; condition expression; update expression) {
    body
}
```

Critical Thinking Requirement:

As stated above, you may use any means you choose to describe the transformation between for loops and while loops. You must describe the transformation process generally -- you cannot simply describe how you would translate a particular for loop into a particular while loop. Bonus points may be awarded for creative presentations at the discretion of the professor and TAs.

Question Task:

For this lab you exercised your skills writing loops and worked with input from files for the first time! Was there anything surprising? Something that didn't make sense? Please feel free to ask anything related to anything that we've learned so far (or will learn)!

Question Requirement:

Submit a file named `question.pdf` which contains your question. Take particular notice of the fact that the question does not have to be subtle, or deep or philosophical for you to receive full points. The goal of the question is for me to be able to get a sense whether there are common questions about the material which would indicate where I failed to teach the material appropriately.

Deliverables:

1. The pseudocode describing the algorithm of your *1d3s* program in PDF format (named `design.pdf`).
2. The completed algorithm practice chart in PDF format (named `cipher-practice.pdf`).
3. The C++ source code for your *1d3s* application (named `1d3s.cpp`).
4. A written response to the Critical Thinking Task prompt in PDF format (named `equivalent.pdf`). If you chose to satisfy the Critical Thinking Task using a means that is not easily transmissible as a PDF, please upload an appropriately named PDF with instructions on how to access your masterpiece!
5. A written question about the material covered so far in this class in PDF format (name the file `question.pdf`).

Rubric:

Points	Task	Criteria
12	Design	Pseudocode uses algorithmic thinking to express a high-level description of how to implement the <i>1d3s</i> application and meets the specified criteria. Each of the four (4) top-level bullets in the Program Design Requirements above are worth 3 points.
8	Design	The algorithm practice chart is completed accurately.

20	Critical Thinking	Critical Thinking Task response describes (graphically, textually or otherwise) a mechanical (algorithmic) process for converting between generic <code>for</code> loops and <code>while</code> loops.
10	Programming	Program generates appropriate output when the input contains no direction.
10	Programming	Program generates appropriate output when the input contains no shift number.
30	Programming	Program correctly calculates outputs when given properly formatted input.
5	Programming	Code is appropriately commented to match the solution's pseudocode and variables are given appropriate/meaningful names.
5	Question	Your question shows an engagement with the material.

Related Learning Objectives:

1. Writing boolean expressions using relational and logical operators
2. Using if-statements to implement selective program execution
3. Using algorithmic thinking to design programs
4. Write syntactically correct for, while and do-while loops
5. Identify the components of for, while and do-while loops
6. Understand the difference between pre- and post-test loops
7. Use each of the three types of loops in the appropriate situation
8. Use methods on file stream objects to read/write files

Credits

writer by Kamin Ginkaew from the Noun Project

writer by Smashing Stocks from the Noun Project

reader by Komkrit Noenpoempisut from the Noun Project