ENGG1003 - Programming Assignment 1
English Text Ciphers

Brenton Schulz

Due date: 5pm Monday of Week 8 (29th April)
Submission: Submit a GitHub repository link to Blackboard any time prior to the due date
Marking: During your Week 8 lab
Weighting: 12.5% of your final grade

1 Introduction

A *cipher* is an algorithm which encrypts a message so that it can be safely transmitted without an eavesdropper being able to (easily) read it. For the purposes of this assignment the "message" will be a block of English text stored as a string of ASCII characters.

Cipher algorithms always perform two tasks: encryption and decryption. The encryption process takes a "message" and "key" as inputs and produces *cipher text*. The decryption process performs the reverse, it turns cipher text and the key back into the original message.

The exact details of how the key and message are processed to produce the cipher text vary between algorithms. However, the general principle is that algorithms employ a mathematical function which is "easy" to calculate with the key but "difficult" to invert without it. A rigorous discussion of modern cryptographic techniques can be found in the excellent (but *highly* mathematical) text, An Introduction to Mathematical Cryptography (available in the Auchmuty library).

This assignment will cover two ancient[1] cipher algorithms:

1. The rotation cipher (also known as *Caesar cipher*)
2. The substitution cipher.

Your task is to write a C program which performs the following broad tasks:

- Encryption given an algorithm, message text, and key
- Decryption given an algorithm, cipher text, and key
- Decryption given cipher text and some assumptions of its contents *without* the key

[1] Literally ancient. The Caesar cipher is attributed to the Roman general & emperor Julius Caesar.

2 Cipher Algorithms

2.1 Rotation Cipher

A rotation cipher encrypts a text message by substituting each letter in the message with a letter a fixed number of places away in the alphabet. The "key" is the number of letters by which the alphabet is shifted when calculating a substitution. When performing the shift any letter that "falls off the end" is rotated back to the start of the alphabet.

Example with a shift of 1 to the right:

Message letter:	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
Cipher text letter:	`ZABCDEFGHIJKLMNOPQRSTUVWXY`

A message is encrypted by substituting letters in the top row with ones from the bottom. For example:

Message:	`ATTACK AT SUNRISE`
Cipher text :	`ZSSZBJ ZS RTMQHRD`

This cipher can be described mathematically using the modulus operator. The "key" is an integer, k, between 0 and 26 (0 and 26 imply "no encryption") and each letter is allocated a number:

$$A = 0, B = 1, C = 2, ..., Z = 25.$$

Encryption can then be performed on a message letter, m, by defining the encryption function, $e(m)$, as:

$$e(x) = (m + k)(\text{mod } 26)$$

where "mod" means the modulus operator (`%` symbol in C).

The decryption of a cipher text letter, c, can be defined by a decryption function, $d(c)$, as follows:

$$d(c) = (c - k)(\text{mod } 26)$$

NB: In C, the modulus operator has undefined behaviour when dealing with negative numbers. If a negative occurs in the calculation of $(m + k)$ or $(c - k)$ then you can add 26 to make the result positive without impacting the result.

2.1.1 Rotation Cipher Attacks

Because the encryption key is an alphabetical rotation there are only 25 different substitutions to choose from (the 26th being "no rotation"). Furthermore, because the encrypted alphabet is still in alphabetical order a full decryption is possible if any one letter substitution is discovered.

Decryption *without* the key can therefore be done with two methods:

- A *brute force* attack, where every possible key is tested and all the algorithm outputs tested for intelligibility (eg: How many spelling mistakes are there?) or a search for some known phrase (eg: a recipient's name)
- A statistical attack, where each different letter in the cipher text is counted, the most frequent letter assumed to be 'e' (or 't', then 'a'), and the rotation deduced and tested from that.

To perform either of the above attacks you only need knowledge of the cipher algorithm and the language of the original message. Even if done by hand, brute forcing 25 different encryption keys is quite straightforward.

The security of the rotation cipher mostly comes from the algorithm being kept secret. In the modern world "security by obscurity" is not considered to be of much use so this cipher is studied purely out of historic curiosity.

2.2 Substitution Cipher

A substitution cipher encrypts a text message by replacing each of the 26 message letters to an encrypted letter. Each letter is only chosen once. The "key" is therefore knowledge of all 26 different substitutions. The number of possible key combinations is: $26! \approx 4 \times 10^{26}$.

Example with a substitution chosen from the qwerty keyboard layout:

Message letter:	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
Cipher text letter:	`QWERTYUIOPASDFGHJKLZXCVBNM`

As before, a message is encrypted by substituting each letter in the top row with the one below it:

Message:	`PLEASE GET MILK AT THE SHOPS`
Cipher text :	`HSTQLT UTZ DOSA QZ ZIT LIGHL`

There is no neat algebraic method to write the encryption function. It is effectively a "look-up-table" where each letter, x_n, becomes a different letter, y_n based on a fixed substitution rule.

2.3 Substitution Cipher Attacks

With 26! different encryption keys a brute force attack is not possible. Even when testing a million combinations *per second* it would take almost 1000 times longer than the *age of the universe* to test every encryption key. Aside: the German Enigma machine (used in WWII) was a form of substitution cipher and a new encryption key was used every day. Even with modern computers a naive brute force attack would not have been possible. You can read about the Enigma decryption efforts on Wikipedia.

Decryption must therefore rely on extra information which reduces the key search space. It is possible, for example, to perform a statistical analysis of the cipher text to estimate which letters were used for 'e', 't', 'a', 'z', etc.

If the message is assumed to be normal English text then other assumptions can be made. Any single letter word is likely to be 'a' or 'i', the latter being easier to spot if the message is case sensitive. Likewise, the most common three letter word is likely to be 'the'.

By making educated guesses about the most common short words and letters a subset of the encryption key can be deduced. This knowledge, coupled with a dictionary, and some kind of "spell checker" algorithm such as *Levenshtein distance*, can then be used in an attempt to work out further letter substitutions.

The WWII efforts to decrypt Enigma relied heavily on *cribs*. These were known, or assumed, sections of plain text for a given encrypted message. For example, many German messages would include regular weather reports in the same format and would reveal several of the day's letter substitutions.

3 Programming Task

Please read the full marking guide before reacting emotionally to the task difficulty. A pass will be "reasonably easy" but a high distinction will be "unreasonably difficult".

Write a C program which performs the following tasks:

1. Encryption of a message with a rotation cipher given the message text and rotation amount
2. Decryption of a message encrypted with a rotation cipher given cipher text and rotation amount
3. Encryption of a message with a substitution cipher given message text and alphabet substitution
4. Decryption of a message encrypted with a substitution cipher given cipher text and substitutions
5. Decryption of a message encrypted with a rotation cipher given cipher text only
6. Decryption of a message encrypted with a substitution cipher given cipher text only

3.1 User Interface Specification

3.1.1 Inputs

For all data inputs (message, keys, cipher text, algorithm selection, etc) you may choose from the following methods (**NB:** methods are *not* worth equal marks):

- Hard-coded variable initialisation
- Read from `stdin` with `scanf()`
- Read from a file using the C standard file I/O library

It is *strongly* recommended that variable initialisation be used while debugging your programs and file I/O only be implemented as a final feature.

If file I/O is implemented the file name may be hard coded. You are permitted to read the file name(s) as command line arguments but, at time of writing, this is beyond the scope of ENGG1003 and will not attract extra marks.

When being graded, you should ensure that multiple inputs can be tested quickly. This is especially true for the decryption tasks which will be tested with multiple blocks of cipher text.

3.1.2 Outputs

All program output should be sent to `stdout`. You are encouraged to also use file I/O for outputs but, due to time constraints while marking, all output should be sent to *both* the file and `stdout`.

3.1.3 Task Selection

It should be easy for a *demonstrator* to quickly test functionality of each task listed in Section 3. It is up to you to decide exactly how this is implemented but the following options are recommended (NB: not all options are worth equal marks):

- Hard code an initialised integer variable which selects between the different tasks.
- Take user input from `stdin` to select each task in a menu system
- (Advanced) Define the task as part of a *header* inside an input file which contains the message and key (or key and cipher text, or just cipher text). The header could be something like:
  - A single integer placed on the first line to indicate the task to be performed, followed by
  - A 2nd, optional, line which contains a key (perhaps start the line with a "weird" character like # to indicate that it is a key), followed by
  - The message or cipher text
- (Advanced-Beyond ENGG1003) Use command line arguments

Each task should be implemented as a different function. Beyond that you are free to choose how many functions are implemented.

All tasks must be accessible from running a **single** `.c` file. The GitHub repository should contain this `.c` file and any other required files (eg: input text).

3.2 Message Text Specification

The ciphers above are only defined for letters. This leaves a few potential ambiguities:

- What should be done to punctuation and white space?
- What about numerals?
- Should upper and lower case letters be handled differently?

For the purposes of this assignment you should apply the following rules:

1. Do not encrypt white space, punctuation, or numerals. If an input character is not a letter it should be copied to the output unmodified.
2. All input data should use UPPER CASE letters only
   - (Advanced) If a lower case letter is found in the input it should be converted to upper case before encryption

3.3 Key Format Specification

The rotation cipher key is to be a single integer in the range [0,25]. "Encryption" with a shift of zero should process the plain text but produce cipher text equal to plain text.

The substitution cipher key is a string of 26 UPPER CASE letters ordered as-per their alphabetical substitution.

eg: The string `"QAZXSWEDCVFRTGBNHYUJMKILOP"` would cause 'A' to become 'Q', 'B' to become 'A', ..., 'Z' to become 'P'. Substitution of a letter with itself (ie: no change for one or more letters) should be allowed.

3.3.1 ASCII Code

All input data is to be encoded with the ASCII standard. ASCII encoding defines that upper and lower case letters be stored as the following 8-bit integers:

A	65	a	97
B	66	b	98
C	67	c	100
...		...	
Y	89	y	121
Z	90	z	122

If an input byte is outside of the ranges $[65, 90]$ and $[97, 122]$ then it can be copied to the output without modification. If an input byte is in the lower case range, $[97, 122]$, then you should subtract 32 from its value to make it an upper case letter prior to encryption.

4 Tackling the Problem

To complete all parts of this assignment some project management techniques are required.

This is a big task. It is intentionally large because I encourage you to discuss potential algorithms with your peers while writing your program. Implementation in C, however, should all be your own work.

It is also expected that some level of independent research will be needed before you can fully implement all the specifications.

Each programming task listed in Section 3 gets progressively more difficult. I am not expecting anybody to 100% "finish" this task but have designed it so that achieving a pass or credit is a "reasonable" amount of work for an average student. By contrast, achieving a high distinction should be an "extensive" amount of work for the high achievers.

You will need to do some planning before doing too much coding. This will involve understanding the problem at a high level (eg: message and key in, encrypted text out) then breaking it down into smaller sub-tasks (eg: read message in, read key, loop over message doing substitutions, store message somewhere).

Each time you define a sub-task try to describe it as a C function. Can you clearly define the inputs, the processing, and the outputs? If so, it is a good candidate for an *actual* function in your code.

After sub-tasks have been defined the details can be filled in. What variables do you need? Should you use 26 `char`'s to represent 26 letters or should they be in an array?

5 Marking and Submission

Marking will be performed by a demonstrator during your enrolled lab time in week 8.

Your code should be submitted to Blackboard as a GitHub repository link. An effort will be made to detect plagiarism using a mixture of computer tools and human judgement.

Code committed after 5pm Monday 29th April will not be marked.

Progress marks (0.5 ea) are awarded for performing a git commit at the following times:

- At the beginning of your Week 6 lab
  - Initial commit. Simply create the nearly-empty main `.c` file
- At the end of your Week 6 lab
- At the end of your Week 7 lab
- On submission day: Monday of Week 8 (29th April) between 9am and 5pm

Show the git commit log to your demonstrator to be awarded these marks.

Any required files should be included in your GitHub repository (eg: plain and cipher text files). Your code should run by performing a `git clone` followed by compilation (`gcc *.c`) and execution (`./a.out`). Code developed with Eclipse Che in a workspace created with the provided factory link meets these criteria. **You must ensure that any other development platform also meets these criteria when uploading to GitHub**.

6 Commenting

Your ability to comment will be assessed in this task. The target audience for the comments is a student who has previously failed ENGG1003 and is trying to learn how your program works by reading the source code.

This implies the following guidelines:

- Most lines will need a comment.
  - Tell me *why* your code is doing something. For example, this comment is useless:

    ```
    a = 1; // Set a to 1
    ```

    but this one could be the difference between a working program and a totally broken one:

    ```
    a = 1; // Re-initialise loop variable to skip first array element
    ```

- A large block comment should be at the top of your code which describes the high-level operation of the program. It should also include any user-interface notes (eg: how should the demonstrator choose between encryption and decryption?)
- Every function needs to be documented in a block comment above the function definition:
  - What are the inputs?
  - What is the return value?
  - What does the function do?
  - Are there limitations to the function? Must strings be less than a certain length? Are there data type restrictions? etc.
- Program flow control needs to be briefly described.

7 Code Indentation

Your program must follow a code indentation standard. Exactly which one is up to you. A list of common standards can be found on Wikipedia. If in doubt, use "Allman" (as it places matching pairs of braces in vertical lines) or K&R (because I like it).

8 External Resources

This list may be extended after the assignment is published.

For decryption purposes you may find a list of the 10 000 most common English words useful. One can be found on GitHub here: `https://github.com/first20hours/google-10000-english/blob/master/google-10000-english.txt`

A more complete list (466k words) is here: `https://github.com/dwyl/english-words`

Text analysis (counting words, etc) can be done here: `http://textalyser.net/`

There will likely be other online text analysers available, that was just one near the top of the Google results.

9 Mark Allocation

Total marks: 12.5

Git commits (as-per Section 5): 2 marks

Code commenting: 1 mark

Encryption with a rotation cipher given plain text and key: 1 mark

Decryption with a rotation cipher given cipher text and key: 1 mark

Encryption with a substitution cipher given plain text and key: 1 mark

Decryption with a substitution cipher given cipher text and key: 1 mark

Completing all of the above achieves a passing grade of 7/12.5 (56%).

Decryption of a previously unseen cipher text encrypted with a rotation cipher: 1.5 marks

Decryption of a day-1 provided block of cipher text encrypted with a substitution cipher: 1 mark

Decryption of an unseen block of cipher text on marking day: 1.5 marks

The substitution cipher decryption marks will be allocated based on how many letters are correctly worked out by your program.

The following marks will be awarded for various cipher and plain text input methods (if multiple methods are used demonstrators may, at their digression, award marks based on the "most advanced" method used at any point in the program):

- Hard-coded cipher and plain text: +0 marks
- Cipher and plain text read from `stdin`: +0.5 mark
- Cipher and plain text read from, and written to, files: +0.75 marks

The following marks will be awarded for various task selection methods:

- Task selection with a hard-coded variable: +0 marks
- Task selection with a "simple" method utilising `stdin` (eg: read a number without prompting the user): +0.25 marks
- Task selection with a "user friendly" menu system: +0.5 marks
- Task selection with a file header or command line arguments: +0.75 marks

At the demonstrator's discretion, the following marks may be deducted:

- Comments are woefully inadequate, uninformative, or missing: -1 mark
- Code indentation is confusing, not attempted, or is in some way unnecessarily difficult to read: -1 mark