ECE 190 MP3 – Breaking the Caesar Cipher
Spring 2010

Due dates:

Checkpoint 1 – Wednesday, 03/17/2010 at 5:00pm
Checkpoint 2 – Wednesday, 03/31/2010 at 5:00pm

Overview:

The goal of this MP is to create a program that can analyze enciphered text and decipher it based on the assumption that it was enciphered using the Caesar cipher.

The Caesar cipher is a simple shift cipher where every letter in the alphabet is simply shifted by some offset (i.e., key) modulo 26 (the number of letters in the alphabet). For example, suppose we use the following plain text,

```
Linux
```

and we want to encipher it with the offset 'd' or 4, then the text after enciphering (i.e., the cipher text) would be,

```
Pmryb
```

Notice how 'x' wrapped around the alphabet. There are 26 possible keys for the Caesar cipher since there are 26 letters in the English alphabet.

One way to break the Caesar cipher is by analyzing the frequency of characters in the cipher text, which is called frequency analysis. The letters in the English alphabet appear in the English language with a well-known frequency, statistically speaking. By matching the expected frequencies from the cipher text and the plain text, the cipher text can be deciphered without knowing the key beforehand. That is, one can deduce the key used to encipher the plain text by choosing the key with letter frequencies closest to that of the English language (i.e., by minimizing the error). In other words, the key that results in the lowest error is the key that was used to encipher the plain text. These frequencies can easily be found online, but will be provided for you.

Frequency analysis example:

Suppose we encipher the text,

```
In cryptography, a Caesar cipher, also known as a Caesar's
cipher, the shift cipher, Caesar's code or Caesar shift, is
one of the simplest and most widely known encryption
techniques.
```

The letter frequency of the plain text is shown in column 2 of Table 1. If we encipher this text with an offset/key of 3, then the cipher text is,

```
Lq fubswrjudskb, d Fdhvdu flskhu, dovr nqrzq dv d Fdhvdu'v
flskhu, wkh vkliw flskhu, Fdhvdu'v frgh ru Fdhvdu vkliw, lv
rqh ri wkh vlpsohvw dqg prvw zlghob nqrzq hqfubswlrq
whfkqltxhv.
```

The corresponding frequency of the cipher text is given in column 3 of Table 1. To use frequency analysis to break the cipher and decipher the text, we need to find the key with the minimum error, i.e.,

$$k^* = \arg\min_k \left(\sum_{i=0}^{25} \left(f_p(i) - f_E((i+k) \mod 26) \right)^2 \right)$$

where $f_p()$ and $f_E()$ are the letter frequencies of the plain text and English language, respectively. From this equation, the key with the minimum error (i.e., k^*) is 3.

Table 1 – Frequencies of example plain text, cipher text, and the English language

Letter	Plain text frequency	Cipher text frequency	English frequency
a	0.09589	0.00000	0.08167
b	0.00000	0.02740	0.01492
c	0.07534	0.00000	0.02782
d	0.02055	0.09589	0.04253
e	0.10959	0.00000	0.12702
f	0.02055	0.07534	0.02228
g	0.00685	0.02055	0.02015
h	0.06164	0.10959	0.06094
i	0.07534	0.02055	0.06966
j	0.00000	0.00685	0.00153
k	0.01370	0.06164	0.00772
l	0.02055	0.07534	0.04025
m	0.01370	0.00000	0.02406
n	0.06849	0.01370	0.06749
o	0.06849	0.02055	0.07507
p	0.04795	0.01370	0.01929
q	0.00685	0.06849	0.00095
r	0.07534	0.06849	0.05987
s	0.10274	0.04795	0.06327
t	0.06164	0.00685	0.09056
u	0.00685	0.07534	0.02758
v	0.00000	0.10274	0.00978
w	0.02055	0.06164	0.02360
x	0.00000	0.00685	0.00150
y	0.02740	0.00000	0.01974
z	0.00000	0.02055	0.00074

Program description:

In this MP, you need to implement 2 separate C programs, one for each checkpoint. The first program, to be submitted for MP3.1, reads a string from the user, counts the number of each letter, divides each of these numbers by the total number of letters, and displays the result to the user. The second program, to be submitted for MP3.2, does everything that MP3.1 does, decides how the input text should be deciphered, and finally outputs the deciphered text (i.e., the plain text).

Given code:

You will be given a skeleton file named mp3.c, which contains necessary declarations, an empty main function, empty subroutines, an implemented function PrintMsg, an implemented function calcError, and some comments. You should use PrintMsg to print some messages. For example, use PrintMsg(0) to print message 0. The given code in mp3.c contains the declarations of global variables cntLetters, totalLetters, cipherText, table (the English character frequency table), estimatedKey, and minimumError. This given code also contains a function calcError that calculates the error between the standard English character distribution and the distribution of characters as counted by your program given an offset (a proposed key value), which is passed to the function.

Checkpoint 1 tasks:

For checkpoint 1, you need to modify the file mp3.c and implement the Count function, which reads in a character one at a time, using getchar, until the 'Enter' key or '\n' has been entered. For help on using the getchar function, see the man page (i.e., `man getchar`). The user input will be cipher text and should be stored in the cipherText variable that is already declared. At the end of the user input, this function displays the frequency of each letter that was entered by the user (i.e., PrintMsg with message code 1). For this function to work, you must store the count of each letter in the cntLetters global variable and the total number of letters in the totalLetters global variable.

For checkpoint 1, we will only test your code with lower case letters and spaces, i.e., we will not use non-alphabetic characters or upper case characters. Table 2 shows all of the possible message codes you can use, but for checkpoint 1, you only need to use codes 0 and 1. You should use message code 0 to prompt the user for input, then you should use message code 1 to display the result of analyzing the letter frequencies. In the main function, you should include a function call to Count, and any initializations, if necessary.

Table 2 – MP3.1 message information for the PrintMsg function.

Message code	Message
0	Please enter cipher text:
1	Letter frequency table: <letter,frequency> <letter,frequency> …
2	Estimated key = <deduced encryption key>
3	Estimated error = <estimation error>
4	Plain text:

Checkpoint 2 tasks:

For checkpoint 2, you need to modify your mp3.c file and implement the printPlainText function. Your code for checkpoint 2 must be able to handle non-alphabetic characters and upper case characters.

There are 3 global variables that you should use in MP3.2 that were not used in MP3.1.

1. table – This array holds the frequencies of letters in the English language.
2. estimatedKey – You should store the key that you deduce was used to create the cipher text in this variable.
3. minimumError – You should store the calculated error for estimatedKey in this variable.

The error calculating function calcError is already implemented for you. You need to call it and decide on the cipher key used to create the cipher text based on its output. The calcError function calculates the error between the letter frequency you count with your MP3.1 code and the frequency of the English language if it had been enciphered with a given key. You must pass this key to the function. The function returns the error between the two frequency distributions.

Finally, you need to implement the printPlainText function, which decriphers the cipher text that will be entered and analyzed through the code you write for MP3.1. In the printPlainText function will need to output the deciphered plain text to the screen. Before you print the plain text to the screen, you should print the estimated key using PrintMsg and code 2, and display "Plain text:" by calling PrintMsg using code 4.

In addition to your code from checkpoint 1, the main function should include code to decide on a key, a function call to printPlainText, and any initializations, if necessary.

The PrintMsg function has a message code (i.e., code 3) which you do not have to use but may be helpful for testing. The error referred to in message code 3 is the error between the frequency of the cipher text assuming the key stored in estimatedKey and the frequency of English. These messages should not be displayed in the code you hand in; they are only available to you to help you test.

Remember that there are only 26 possible keys, and the key that produces the smallest error is most likely the key used to encipher the plain text.

Testing:

Before you can test your program you must create an executable. You **must** do so by exactly typing

```
gcc -Wall -g -ansi mp3.c -o mp3
```

at the terminal, when you are in the same directory as your code. If there is no error found, it will generate a binary executable named mp3. Otherwise you need to fix all errors and warnings that are reported. If you make the mistake of putting mp3.c after the -o argument, your source will be overwritten. Keep copies and be careful. You may execute the binary by simply typing "./mp3" or "./mp3" at the terminal. We will provide gold binary files for both checkpoints and some test cases to help you test your program. You should use the gold binary to generate the standard output and compare it with the output from your own program. Your output must match the gold output EXACTLY. You

can compare your output to the gold output using diff or vimdiff.

Specifics:

- When you hand your code in, your program files must be called **mp3.1.c** and **mp3.2.c**.
- You must implement the required functions. You are not required to use additional functions. You may if you think they are helpful.
- You must use the predefined message codes.
- You may not include any additional header files (other than the ones already included).
- You may not use any additional global variables (other than the ones provided).
- You may define additional constants to improve the readability of your code. (i.e., using #define statements).
- Your code should be in good coding style, with proper indentation on each line to indicate functional blocks and explanatory comments. A brief header describing the program and headers for main function and support functions are important. You should not have any lines longer than 80 characters.
- Your code must pass compilation. You will not receive any functionality points on a code that fails to generate a binary. You will also lose points for compiler warnings.

Handing in:

Turn in your code using the ECE190 handin script. Your files must be called mp3.1.c and mp3.2.c. We will NOT grade files with any other name. To hand in the code, first login to any EWS machine and type ece190. Change to the directory containing your code and type

```
handin --MP 3.1 mp3.1.c
```

or

```
handin --MP 3.2 mp3.2.c
```

You may hand in as many times as you like, but only the last submission will be time stamped and graded. Note that only your .c file is submitted.

Grading:

Your code functionality will be graded by a computer script. Use input/output prompting messages provided in the tables and test your program using our test cases. You must ensure that your program terminates properly and does not loop forever.

- Functionality (65%)
  - 5% - Uses predefined global variables as counters.
  - 10% - Program terminates correctly.
  - 10% - Uses predefined message codes.
  - 15% - (**Checkpoint 1**) Program reads each character and increments appropriate counters correctly.
  - 10% - (**Checkpoint 2**) Checks for lowercase, uppercase, and non-alphabetic characters.
  - 15% - (**Checkpoint 2**) Program reads each string and deciphers it correctly.
- Style (20%)
  - 10% - Program compiles without errors or warnings.
  - 5% - Good and consistent indentations.

- 5% - No lines longer than 80 characters.
- Documentation (15%)
  - 5% - Introductory paragraph explaining program's purpose and approach.
  - 10% - Good comments and variable names.