Assignment 3

Decrypting a Substitution Cipher

Due: Friday, March 27, 2015
before midnight

Purpose:

C's speed makes it an idea language for computationally intensive tasks. Even in the age of 3.2 GHz quad processors, there are still applications that require a language that runs straight off the silicon. One such application is cryptography. C is used both to make cryptographic subsystems (the OpenSSL library, which is used by many Cisco products, is implemented in C) and to decrypt code written using these products.

In this assignment we'll use C to break a simple substitution cipher. In a real decryption package, we'd automate our code to run through billions or trillions of possible combinations while comparing the output against a dictionary of possible words. In this example, a human will take the place of the dictionary, and the encrypted text will need to be decoded by carefully deciding which two letters to swap during each iteration of the program. To implement your code, you'll need to deal directly with pointers to arrays.

You will have completed this assignment when you have:

- ☐ Implemented the instructions provided below, completely, as described;
- ☐ Used only pointers in the manipulation of data contained in arrays, as described below in Section III: Implementation;
- ☐ Submitted all the deliverables for this assignment as indicated at the end of this document, including the decrypted plaintext, along with your functioning C code;
- ☐ Documented your code according to the Documentation Standard for the course.

Assignment 3

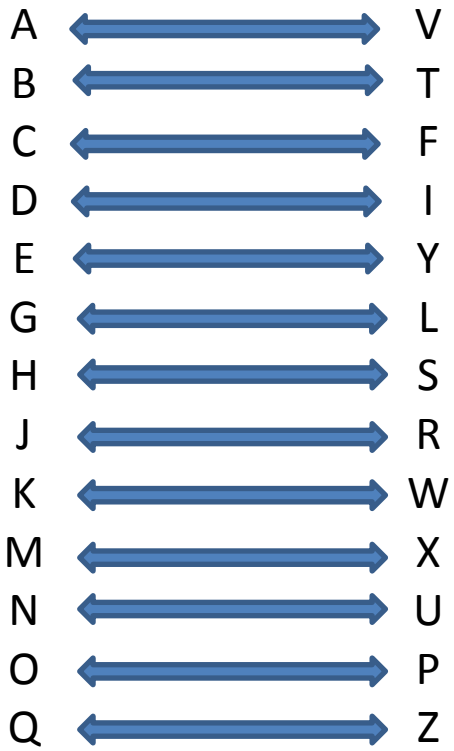
Decrypting a Substitution Cipher

I. Introduction

A substitution cipher is a simple encryption method involving swapping each of the characters in the unencrypted text—usually referred to as the *plaintext*—for another, resulting in an encrypted text, or *ciphertext*. For example, if the following is our plaintext:

All Gaul is divided into three parts, one of which the Belgae inhabit, the Aquitani another, those who in their own language are called Celts, in our Gauls, the third. All these differ from each other in language, customs and laws. The river Garonne…

and we then convert each character to uppercase and employ the following algorithm:



then the result is (minus the punctuation, but keeping the spaces):

VGG LVNG DH IDADIYI DUBP
BSJYY OVJBH PUY PC KSDFS
BSY TYGLVY DUSVTDB BSY
VZNDBVUD VUPBSYJ BSPHY KSP
DU BSYDJ PKU GVULNVLY VJY
FVGGYI FYGBH DU PNJ LVNGH
BSY BSDJI VGG BSYHY IDCCYJ
CJPX YVFS PBSYJ DU
GVULNVLY FNHBPXH VUI GVKH
BSY JDAYJ LVJPUUY…

This is the *ciphertext*, which will be written in capital letters throughout this document, to distinguish it from plaintext.

One of the first known uses of a substitution cipher was made by Julius Caesar and is recorded in the history of his campaigns, from which the above plaintext is a quote. A similar substitution cipher was used by Mary, Queen of Scots in the 16th century to smuggle coded messages to her supporters during her imprisonment by Elizabeth 1. Its decryption, using the method we'll use in this assignment, secured her execution in 1587. There are several good histories of cryptography, of which *The Code Book* by Simon Singh is one of the best.

II. Methodology

The standard method for decrypting a substitution cipher is called *frequency analysis*. In the English language, certain letters always occur with greater frequency than others, a fact reflected in the nonsense phrase/name made up of the 12 most common letters, in order: *ETAOIN SHRDLU*.

Legend has it that printers were the first to discover this pattern, since the most commonly-used characters are the ones whose typeface wears down the fastest. This term has passed into popular culture (see http://en.wikipedia.org/wiki/Etaoin_shrdlu), and has a listing in the *Oxford English Dictionary*.

ETAOIN SHRDLU shows up in other places as well, in ways you might not expect. For example, Morse code was designed to be fast and efficient, and so the most frequently-used characters are generally represented by the shortest combination of symbols: E is a single dot, T is a single dash, A is a dot-dash, O is three dashes, I is dot-dot, N is dash-dot and so on, using progressively more symbols for the less frequent characters. Another example: in the word game Scrabble, the value of the letters *ETAOIN* are each '1', reflecting the large number of tiles available for each of these characters in the standard game, which in turn is a reflection of the high frequency of each of these letters in the English language.

Imagine you'd been given the ciphertext shown above, in uppercase letters. If you sort the ciphertext by the frequency of each character, you'd find that the most common characters were, in order, something like YBVPDU HSJIGN. However, knowing *ETAOIN SHRDLU*, you could reasonably deduce that the Y's were being substituted for E's, B's for T's, and so on. Reversing the substitutions made above, you'd arrive back at the original text. That's how frequency analysis works.

Well, not quite. *ETAOIN SHRDLU* only works when you have a large enough ciphertext to work with, and then the statistics that frequency analysis relies on come to your aid. For shorter messages, you aren't guaranteed that the most common character will represent an 'E', the second most common a 'T', and so on; a certain amount of guesswork is required. So while a computer program that identifies the most frequent letters in a string and allows the user to make swaps between characters is an essential tool for success—that is your goal in this assignment, in a nutshell—a human brain is still required to make the correct guesses about which characters to substitute.

For example, consider the output shown below in Figure 3-1. As a first guess, the user has assumed that each of the V's in the above ciphertext are in fact T's. The output of the program is:

```
TGG LTNG DH IDADIYI DUBP BSJYY
OTJBH PUY PC KSDFS BSY VYGLTY
DUSTVDB BSY TZNDBTUD TUPBSYJ
BSPHY KSP DU BSYDJ PKU GTULNTLY
TJY FTGGYI FYGBH DU PNJ LTNGH
BSY BSDJI TGG BSYHY IDCCYJ CJPX
YTFS PBSYJ DU GTULNTLY FNHBPXH
TUI GTKH BSY JDAYJ LTJPUUY…

Frequency analysis:
Y: 25    T: 20   B:16    D:14
…etc.

Enter char to substitute: Y
Swap this with character: E
```

Figure 3-1

(Note: The newly swapped characters are highlighted in bold for emphasis; this feature will *not* be part of your program. Note also: T's become V's *and* V's become T's, not just one *or* the other.)

Here, the human decrypter has noticed that the character 'Y' has a very high number of occurrences, and guesses that it might stand for

the letter 'E'—the first character in *ETAOIN SHRDLU* and the most frequent letter in the English alphabet. This suspicion is supported by the appearance of the sequence 'TJE' in a few places, suggesting that the 'J' is substituting for 'H' and the word is really 'THE'.

In fact, these assumptions are incorrect, as we know from a comparison with the plaintext. 'V' is *not* being substituted for 'T'. Further analysis of the ciphertext in Figure 3-1 shows why this is probably not the case. The first word in the ciphertext is TGG. Assuming the first character is correct—we just substituted in the 'T'—the only two words in the English language that fit this pattern are TOO and TEE. Thus 'G' must stand for 'E' or 'O'. But 'G' has is a fairly low frequency letter in the ciphertext, showing up only 13 times in the ciphertext shown above (not shown in limited output fragment displayed), making it an unlikely substitute for high-frequency letters like 'E' or 'O'.

So the actual decryption of the ciphertext will rely not just on the straightforward application of frequency analysis or the competency of the code you write to assist in this process. Patience, intuition, attention to detail, and forethought will all play a role in your decryption of the ciphertext—qualities of a good programmer as well, it should be noted.

III. Implementation

Each student will be given a unique string of ciphertext to decrypt. This is found in the file Assignment3.zip. Download the file, unzip it, find the file with your student number on it, and copy this file into the folder where you do your C programming in Ubuntu. (Contact me immediately if your student number is not in the list of encrypted files.) Inside `main()`, copy the ciphertext string into your program, along with the two other arrays indicated below. Your code must have, and use, the following three arrays declared *inside the `main()` function* of your program:

```
char cipherText[] = "VGG LVNG DH
IDADIYI DUBP BSJYY OVJBH PUY PC
KSDFS BSY TYGLVY DUSVTDB
BSY…etc."; //use your ciphertext
           //from Assignment3.zip

int charFreq[26] = {0};

int* alphaSort[26] = {'\0'};
```

where the `cipherText[]` array will contain the unique ciphertext provided in the zip file for *you* to decrypt, and *not* the default one shown above. The second array, `charFreq[]`, will be used to store the number of occurrences of each character found in the `cipherText[]` array at each iteration of the decryption process. For example, the total number of A's in `cipherText` will be stored in `charFreq[0]`; the number of B's in `charFreq[1]`; and so on, up to `charFreq[25]` (for the Z's, if any).

The final array, `alphaSort[]`, is an array of pointers to `int`s, i.e. it is an array of 26 addresses. The first element of this array (index 0) stores the address of the `charFreq[]` element that contains the largest number i.e. it corresponds the most frequently used letter. The next element of `alphaSort[]` contains the address storing the next largest number, and so on. So `alphaSort[]` contains the frequency analysis information that will be displayed each time you make a substitution. But rather than store the letter associated with the frequency, we only need to store the address corresponding to that letter.

For example, given the above ciphertext, we obtain the following values, shown in Figure 3-2. As shown, the `charFreq[]` array contains the number of occurrences of each character, while `alphaSort[]` contains the address of the most frequent character, a 'Y' (character 24),

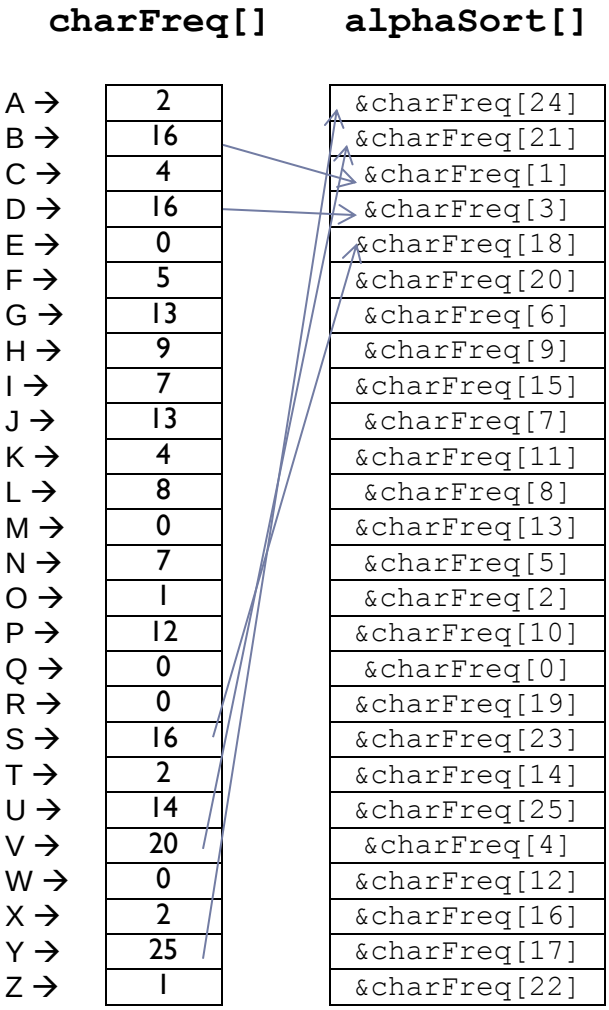


Figure 3-2

which appears 25 times in the ciphertext given above. The second most common character, the 'V' (character 21) occurs 20 times, while the third most common character, 'B' (character 1) occurs 16 times (as do 'D' and 'S'). The least used characters, E, M, Q, R and W show up exactly zero times in the ciphertext. Reference to the original algorithm at the start of this document indicates why: these letters correspond to J, K, X, Y, and Z.

Your code must contain (at minimum) the following four functions, operating as described below:

a. A function called `doFrequencyAnalysis()`, which takes as arguments a pointer to the `cipherText[]` array and a pointer to the `charFreq[]` array. Your code should loop through the former array and increment the corresponding element in the `charFreq[]` array. Note that this operation does not require the use of any `if` or `select` statements, merely a single `for` loop; it should be implemented in under a half dozen lines of code.

b. A function called `sortFrequencyAddresses()`. This function takes as parameters pointers to both the `charFreq[]` and `alphaSort[]` arrays. To implement this function, take your favorite sorting routine (acquired from your Data Structures course) and adapt it as follows. Sort through the values in `charFreq[]`, but rather than sorting `charFreq[]` by the values stored *inside* the array, use those values to sort the address in which the value is stored. If your algorithm requires it, you may create a temporary array at this stage, but you must use pointers to manipulate the elements and addresses of that array. Alternately, you may find it simpler to implement an operation that loops through `charFreq[]` looking for increasingly smaller numbers on each loop. (This is not a very efficient way to implement a sorting routine, but the decision is yours). Regardless of which

option you choose, your C code should implement the sort as efficiently as is possible given the method chosen.

c. A function called `displayFrequencyAnalysis()`, which takes in a pointer to the `alphaSort[]` array and displays the information contained in the array itself, as indicated in Figure 3-1. Note that each address stored in this array is directly tied to the character whose frequency it contains. To get the value of that character, subtract the value of `charFreq` (which is just the starting address of the array) from any address stored inside `alphaSort`, and add 'A' to it. (So if the address stored in an element of `alphaSort` *is* the value of `charFreq` itself, then the character returned by this operation will be an 'A', exactly as we'd expect, since the value stored in `charFreq[0]` corresponds to the number of 'A's in the ciphertext.)

d. A function called `swapCipherTextChars()`, which takes a pointer to the `cipherText[]` array, prompts the user to swap two characters in the cipherText string, and then does so. [Note: be sure your function swaps *both* characters, and does not merely substitute every occurrence of the first character for the second, neglecting to substitute every occurrence of the second for the first.]

NOTE THAT YOU MUST USE POINTERS THROUGHOUT EACH OF THESE FUNCTIONS; ASIDE FROM ASIGNING POINTERS TO THE THREE ARRAYS DECLARED INSIDE `main()` (THE ONES INDICATED ABOVE) AND USING AN ARRAY NAME AS THE STARTING ADDRESS OF THE ARRAY ITSELF, YOU CANNOT USE ARRAYS IN ANY OF THE FUNCTIONS IN THIS ASSIGNMENT:

USE POINTERS ONLY!

Your output should resemble the output shown in Figure 3-1: display the current ciphertext, the frequency analysis (from the `displayFrequencyAnalysis()` function) and the prompt (from the `swapCipherTextChars()` function). It is useful to clear the screen between character swaps; this makes it a bit easier to spot the changes each time the ciphertext is displayed.

IV. Testing

Test your code thoroughly prior to submission. Note that the zip file contains four encoded TestX.enc files, suitable for practice.*

**The original text on which the ciphertexts are based comes from a declassified NSA document on the history of German ciphers during the Second World War. Don't be surprised if you encounter the word ENIGMA in a few of the decrypted ciphertexts.*

V. Submission Guidelines

Your submission, must include:

1. Your working C code file, and,
2. The decrypted plaintext fragment for the original ciphertext in the file with your student ID on it. This should be included as a separate text file labelled `plaintext.txt`

This should be submitted on or before the deadline, March 27th, before midnight, to the dropbox set up in Canvas. Late submissions will be penalized according to the schedule outlined in Module 0, Slide 20, 'Late Assignments and Labs.' If there are any problems with Canvas,

please ship your code to me at houtmad@algonquincollege.com; however, please look in Canvas first—a number of students are *not using the link which is there in Canvas, and which works for everyone*.
When submitting, please be sure to zip all files. Your zipped file *must* have the following name:

Assignment3_*YourLastName*_*YourFirstName*.zip

Note that

1. Your code must be formatted according to the CST8234 Documentation Standard.
2. Pseudocode is not required for this assignment

VI. Marks/Evaluation

Your assignment will be marked as follows:

Requirement	Mark
Outputs the correct plaintext based on the given ciphertext	2
Code follows the Documentation Standard	6
Followed all the instructions in this document (including any late additions/changes announced in class or via email), and in particular used the functions correctly, i.e. for the purpose described	8
Code executes correctly, i.e. no obvious defects	4
Clarity of Code (terse is good, but not if applied to the point of incomprehensibility)	5
Total:	**25**

VIII. Notes

You may find the following fact useful in decrypting the ciphertext: the thirty most common words in the English language are:

1. the 2. of 3. and 4. a 5. to 6. in 7. is 8. you 9. that 10. it 11. he 12. was 13. for 14. on 15. are 16. as 17. with 18. his 19. they 20. I 21. at 22. be 23. this 24. have 25. from 26. or 27. one 28. had 29. by 30. word

(Source: Melvin Bragg, *The Adventure of English*, pg. 7. Baron Bragg actually lists the top one hundred words, if you're curious.)