CS 100M Fall 2005 Project 3 due Thursday, Oct 13, at 6pm

Introduction

Did you read the news reports on the hacker who exploited a computer system's vulnerabilities to access more than 40 million MasterCard accounts earlier this year? How about the waves of viruses spread by email that crippled not only personal computers but large corporations, government agencies, and other institutions? Computer Research & Technology estimated that viruses are discovered at the rate of 15 per *day* (http://www.crt.net.au/etopics/virus.htm). If you do any Internet shopping, you probably have noticed that most companies have (or claim to have) a "secure" site where customer data are *encrypted* for safe transmission. *Security* is an important aspect of today's digital world: our personal information (e.g., your student records, medical history), business transactions (e.g., credit cards), and government operations (e.g., social security payments, military information) all require our computing infrastructure to be secure.

The term "security" has many meanings when applied to computers and computing technology. Three commonly discussed dimensions are *integrity*, *accessibility*, and *confidentiality*. When a virus or unwanted software is installed on your computer, it violates the *integrity* of your computer system. If a group of people (or a group of machines controlled by a person) launches a denial of service attack on a website by overwhelming it with search queries, it is violating *accessibility* to the service of that website. Maintaining the *confidentiality* of digital data can mean keeping unauthorized people from accessing that data and *making the data unusable even if they are stolen*. *Encryption* is the primary way to render information "unusable"—keep it secret—to unauthorized parties. Encrypted text is plain text scrambled in some fashion—according to a *key*. The authorized user of the information will need to know this key to decode or "*decrypt*" the encrypted information.

In this project, we will deal with a simple method of maintaining confidentiality through cryptography, the process of secret communications. We will learn about simple substitution ciphers, how to use them, and how to break them.

Objectives

In this project, you will learn to work with vectors and matrices in MATLAB. In part A of this project, you will write a function that generates a key for the cipher. In part B of this project, you will write code that encrypts and decrypts text. Part C requires you to implement and use frequency analysis to decode text that was encrypted using a simple substitution cipher. Part D asks you to answer several questions based on your experiments with the simple substitution cipher. Read the entire project statement before starting to write code.

Follow the specifications!

You must follow all the specifications. For example, if a function is supposed to return a vector containing numbers, then write a function to return a vector of numbers, not a vector of letters. In this project, you will mostly write your own code without calling a lot of built-in functions. Here is a list of the built-in functions that you may use:

```
length  size  rand  ones  zeros  char  lower  upper  isletter
                floor  ceil  sum  sortrows
```

You may of course use functions that you write and the ones we have provided. If there's a built-in function *not* listed above that you really, really, think you need to use, check with the course staff first. Download and read all the (skeleton) M-files we have provided, but you don't have to read the text (.txt) files now—they are very, very long. At the end of the project you'll read parts of these files.

Part A: Creating a Substitution Cipher Key

Simple substitution ciphers are a method of encryption in which individual letters of unencrypted text are replaced with individual letters of encrypted text. Each letter of the unencrypted alphabet is replaced with a unique letter in the encrypted alphabet. As an example, consider the following 5-letter alphabet:

`abcde:` unencrypted alphabet or plaintext alphabet
`debca:` encrypted alphabet or ciphertext alphabet

Then the word 'bead' in plaintext would be replaced with 'eadc' in ciphertext. (Note: Your solution must work on the full 26-letter alphabet).

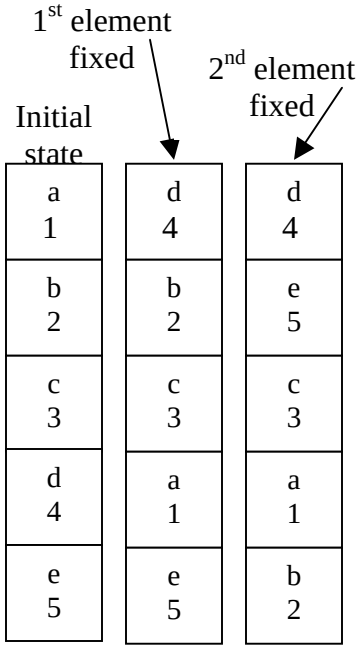
Write a function **`genKey(n)`** to return a *substitution cipher key* for an alphabet with **n** letters. The substitution cipher key is an **n**-by-2 matrix of *numbers*, not letters. The first column of the key is simply the numbers 1 to **n**, representing the order of the letters of the original alphabet. The second column contains the numbers 1 to **n** in a scrambled order, representing the ordering of the letters in the ciphertext alphabet. An example is shown below.

One way to scramble the alphabet is to use the Fisher-Yates shuttle algorithm. In the diagrams below, we show both the numeric values and the letters to which they correspond for clarity. In your solution, the matrix should store only the numbers!

1. Randomly select one of the elements of the array and swap it with the element in the first position. Suppose the 4th element is swapped with the first, then you have the 2nd vector shown below. We then say the first element is fixed. (If the first element of the vector was randomly selected to be swapped into the first position, then the resulting array would look unchanged from the original state. This is fine—it is just one iteration.)

2. Next, randomly select one of the elements in the 2nd through last positions and swap it with the element in the 2nd position. Now the second element in your vector is "fixed." Suppose the 5th element was randomly selected to be swapped in to position 2. Then your array would look as follows:

3. Continue this procedure until all elements in your array are "fixed."



An example on a 5-letter alphabet:

```
key= genKey(5)
key =
  1 4
  2 5
  3 3
  4 1
  5 2
```

Part B: Encrypting and Decrypting a Substitution Cipher

Now that you have a method to generate keys, you can start encrypting and decrypting messages.

Recall the key we generated in part A:

`abcde:` unencrypted alphabet or plaintext alphabet
`debca:` encrypted alphabet or ciphertext alphabet

To encrypt a message, you replace all occurrences of 'a' with 'd', 'b' with 'e', and so on in your unencrypted message. The resulting message after all of these substitutions is the encrypted message.

To decrypt a message, you would perform the opposite substitution. In other words, you replace all occurrences of 'd' with 'a', 'e' with 'b', and so on in your encrypted message. The resulting message after these substitutions is the decrypted message. Note that if you encrypt a message and then decrypt that same message with the same key, you obtain your original result. This is a good way to check if your encryption and decryption functions work.

Write a function **`encrypt`** that has two input parameters:

1. **`filename`**: the name of a text file containing plaintext
2. **`K`**: a substitution cipher key generated in part A

and returns an array of **`lower case`** characters containing the encrypted message using the key.

Write a function **`decrypt`** that has two input parameters:

1. **`filename`**: the name of a text file containing encrypted text
2. **`K`**: a substitution cipher key generated in part A

and returns an array of **`lower case`** characters containing the decrypted message using the key.

Make sure you only encrypt/decrypt the alphabetic characters and ignore punctuation, numbers, etc. Also, make sure both upper- and lower-case letters get encrypted/decrypted by the same key.

MATLAB treats characters as their ASCII values when performing numerical operations. *Review the lecture notes and examples for 9/27.* Also, try these examples at the command prompt to get a feel for it:

```
>>'a' + 1
>>'A' + 2
>>'c' – 'a'
```

The `char` function converts ASCII values back to characters in MATLAB. Try this example:

```
>>char('a' + 1)
```

Example for the 5-letter alphabet:

`abcde:` unencrypted alphabet or plaintext alphabet
`debca:` encrypted alphabet or ciphertext alphabet
In MATLAB:

```
>>secret= encrypt(['cAb.', key)
secret =
  bde.
>>message= decrypt('bdE.',key)
message =
  cab.
```

Part C: Decoding Text using Frequency Analysis

What if you have lost the key you were given for decrypting a message? Well, you can try to "break the code." *Frequency analysis* is a method that can be used to break simple substitution ciphers. In a natural language based on an alphabet, the frequencies of occurrence of the letters are roughly the same across all texts. By looking at the frequencies of different letters in an encrypted text, a cryptanalyst can make educated guesses about which ciphertext characters correspond to which plaintext characters.

Complete the skeleton function **`crack`**. This function has two input parameters:

1. **`trainer`** – name of a file containing a sample of English text.
2. **`mystery`** – name of a file containing a mysterious sample of English text encrypted using a simple substitution cipher.

The function returns a substitution cipher key obtained by performing frequency analysis on the **`trainer`** text.

How does frequency analysis work? First we must determine the letter frequencies in the English language. The *frequency* of a letter is the ratio of its number of occurrences to the total number of letters in some text. You will estimate the frequency of each letter based on the sample text in `trainer`. Put this code in a function called **`freq`** . See the skeleton code for a description of the parameters.

Suppose a file **`sample.txt`** contains the training text and has following characters:

`Cab deaD dad.`

```
>>f= freq('sample.txt')
f =
  1  0.3          %1 through 5 corresponds to 'a' through 'e'
  2  0.1
  3  0.1
  4  0.4
  5  0.1
```

Our table has only 5 rows because we're using our 5-letter alphabet example. Yours should have 26 rows. Be careful not to let non-alphabetic characters ruin your frequency counts. Make sure you understand where the numbers above come from.

Next, you use your function **`freq`** again to calculate the frequencies of each letter in the encrypted text.

Now, you have two frequency tables: one from the training text and one from the mystery text. Our method assumes that both of these texts have similar character frequencies so the most frequent character in **`mystery`** must correspond to the most frequent character in **`trainer`**. Similarly, the second most frequent character in **`mystery`** must correspond to the second most frequent character in **`trainer`**, and so forth.

Suppose the frequency table for the mystery text is:

```
1  0.1
2  0.1
3  0.4
4  0.3
5  0.2
```

Matching the frequencies between the ciphertext and plaintext, we see that the enciphered 'c' (row 3) corresponds to 'd' (row 4) in plaintext. Continuing with the analysis, we see that the enciphered 'd' (row 4), refers to 'a' (row 1) in plaintext. The next highest frequency of the enciphered text belongs to 'e' (row 5), but its match is ambiguous since all the remaining letters in plaintext have the same frequency. We will have to simply choose one of these three remaining plaintext letters as the match for the enciphered 'e'.

Hint: The **`sortrows`** function may be helpful. Use **`help`** in MATLAB to learn about it.

Function **`crack`** will create and return the hypothesized key in the same format as specified in the earlier function **`genKey`**. In function **`crack`**, call function **`decrypt`** to attempt to "crack" the **`mystery`** text. Keep in mind, however, that the frequency analysis shown here is very simple and won't always return the correct key, therefore the result of decryption may not be perfect.

Part D: Testing and experiments

Create or find some text files for testing your code. For the **`encrypt`** and **`decrypt`** functions, you don't need anything long. For the **`crack`** function, however, you need something of reasonable length so that the frequency analysis has a chance to succeed. Try perhaps a page of text as you develop this code. When you are satisfied with your product, run the **`crack`** function on the *large* posted mystery text files **`mys1.txt`** and **`mys2.txt`**. These text files come from *Project Gutenberg*, an online e-book site (http://www.gutenberg.org/).

1. Use your genKey function to create a key and then encrypt **`mys1.txt`**
2. Next use **`mys2.txt`** as the training text in **`crack`** and see if you can decrypt the encrypted version of **`mys1.txt`**.

At the end of your function **`crack`**, answer the following questions in the form of comments:

1. How successful was **`crack`**, based on a simple frequency analysis, in decrypting the encrypted version of **`mys1.txt`**? (I.e., can you read the decrypted text? How similar is it to the original plaintext version?)
2. Compare the key that you generated to encrypt **`mys1.txt`** and the key that was guessed from function **`crack`**. For which letters, if any, was the guessed key incorrect? If you repeat the experiment on other long text files, do you expect the same errors in the guessed key? Does it matter what kind of text you train on and later try to decrypt?
3. Will our method work given text in another language?
4. Describe at least one way in which we can improve the algorithm for cracking the code.

Submit the following files through CMS before the project deadline: `genKey.m`, `encrypt.m`, `decrypt.m`, `freq.m`, `crack.m`.