Computer Science 50: Introduction to Computer Science I
Harvard College
Fall 2007

Problem Set 2: Crypto

out of 51 points

due by 7:00 P.M. on Friday, 12 October 2007

Be sure that your code is thoroughly commented
to such an extent that lines' functionality is apparent from comments alone.

Goals.

The goals of this problem set are to:

- Better acquaint you with functions and libraries.
- Allow you to dabble in cryptography.

Recommended Reading.

Per the syllabus, no books are required for this course. If you feel that you would benefit from some supplementary reading, though, below are some recommendations.

- Sections 11 – 14 and 39 of `http://www.howstuffworks.com/c.htm`.
- Chapters 6, 7, 10, 17, 19, 21, 22, 30, and 32 of *Absolute Beginner's Guide to C*.
- Chapters 7, 8, and 10 of *Programming in C*.

Academic Honesty.

All work that you do toward fulfillment of this course's expectations must be your own unless collaboration is explicitly allowed (*e.g.*, by some problem set or the final project). Viewing or copying another individual's work (even if left by a printer, stored in an executable directory, or accidentally shared in the course's virtual terminal room) or lifting material from a book, magazine, website, or other source—even in part—and presenting it as your own constitutes academic dishonesty, as does showing or giving your work, even in part, to another student.

Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this class that you have submitted or will submit to another. Moreover, submission of any work that you intend to use outside of the course (*e.g.*, for a job) must be approved by the staff.

All forms of cheating will be dealt with harshly.

You are welcome to discuss the course's material with others in order to better understand it. You may even discuss problem sets with classmates, but you may not share code. In other words, you may communicate with classmates in English, but you may not communicate in, say, C. If in doubt as to the appropriateness of some discussion, contact the staff.

Getting Started.

0. SSH to `nice.fas.harvard.edu`, create a directory called `ps2` in your `~/cs50/` directory, and then navigate your way to that directory. (Remember how?) All of the work that you do for this problem set must ultimately reside in this directory for submission.

   Your prompt should now resemble the below.

   ```
   username@nice (~/cs50/ps2):
   ```

1. (5 points.) If you haven't seen the iPhone in action, take a peek at one or more of the videos at `http://www.apple.com/iphone/ads/`. Better yet, find someone on campus with an iPhone and ask for a demo!

   The screen on the iPhone (and iPod Touch) responds to the touch of your finger. You can touch once to click, touch and drag to scroll, and even pinch to zoom. But, in engineering terms, how does it work? In a paragraph of your own words in a file called `iphone.txt`, explain as technically as you can (*i.e.*, you should understand your own explanation) how the iPhone's "multi-touch" screen works.[1] Realize that it's a bit different from, say, your local ATM's touchscreen!

[1] For this question, you're welcome to consult *How Computers Work*, Google, Wikipedia, a friend, or anyone else, so long as your words are ultimately your own!

Let's Warm Up with a Song.

2. (10 points.) Recall the following song from childhood. (Mine, at least.)

This old man, he played one
He played knick-knack on my thumb
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played two
He played knick-knack on my shoe
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played three
He played knick-knack on my knee
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played four
He played knick-knack on my door
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played five
He played knick-knack on my hive
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played six
He played knick-knack on my sticks
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played seven
He played knick-knack up in heaven
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played eight
He played knick-knack on my gate
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played nine
He played knick-knack on my spine
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

This old man, he played ten
He played knick-knack once again
Knick-knack paddywhack, give your dog a bone
This old man came rolling home

Oddly enough, the lyrics to this song don't seem to be standardized. In fact, if you'd like to be overwhelmed with variations, search for some with Google. And then stop procrastinating.

Your first challenge this week is to write, in `oldman.c`, a program that prints, verbatim, the above version of "This Old Man."

Notice, though, the repetition in this song's verses. Clearly your program must make use of some sort of loop to generate repeated lyrics. Your program must also make use of one or more conditions in order to print the number referenced in each verse's first line as well as where this old man played knick-knack in each verse's second line. Allow us to recommend but not require that you somehow employ `switch`.

Before writing any code, think about how you might use hierarchical decomposition (*i.e.*, more than one function) to solve this problem as efficiently and elegantly as possible. Then go solve it, taking care to comment your code!

Hail, Caesar!

3. (15 points.) Your next challenge this week is to write, in `caesar.c`, a program that encrypts messages using Caesar's cipher. Your program must accept a single command-line argument: a non-negative integer, *k*. If your program is executed without any command-line arguments, with more than one command-line argument, or with one command-line argument that contains any characters other than `'0'` through `'9'`, your program should complain and exit immediately. Otherwise, your program must proceed to prompt the user for a `string` of plaintext and then output that text with each alphabetical character "rotated" by *k* positions; non-alphabetical characters should be outputted unchanged. After outputting this ciphertext, your program should exit.

   Although there exist only 26 letters in the English alphabet, you may not assume that *k* will be less than or equal to 26; your program should work for all non-negative integral values of *k* less than 2^{32}. Even if *k* is greater than 26, alphabetical characters in your program's input should remain alphabetical characters in your program's output. For instance, if *k* is 27, `'A'` should not become `'['`, even though `'['` is 27 positions away from `'A'` in ASCII; `'A'` should become `'B'`, since 27 modulo 26 is 1. In other words, values like *k* = 1 and *k* = 27 are effectively equivalent.

   Your program must preserve case: capitalized letters, though rotated, must remain capitalized letters; lowercase letters, though rotated, must remain lowercase letters.

Functionally, your program might resemble the below. Underlined are some sample inputs.

```
username@nice (~/cs50/ps2): caesar 13
Plaintext:  Be sure to drink your Ovaltine!
Ciphertext: Or fher gb qevax lbhe Binygvar!
```

So that you don't reinvent the wheel, do scour

```
http://www.cppreference.com/stdstring/index.html
```

for any functions that might be of assistance to you. As will be often the case, there is more than one way to solve the problem at hand. But know that `isdigit` and `atoi` might prove particularly good friends. You should probably dig up Week 0's ASCII chart. And best not to forget about an operator like `%`.

Parlez-vous français?

4. (20 points.) Well that last cipher was hardly secure. Let's implement a more sophisticated algorithm. Needless to say, it's French. Your final challenge this week is to write, in `vigenere.c`, a program that encrypts messages using Vigenère's cipher. This program must accept a single command-line argument: a keyword, *k*, composed entirely of alphabetical characters. If your program is executed without any command-line arguments, with more than one command-line argument, or with one command-line argument that contains any non-alphabetical character, your program should complain and exit immediately. Otherwise, your program must proceed to prompt the user for a `string` of plaintext, *p*, which it must then encrypt according to Vigenère's cipher with *k*, ultimately printing the result and exiting.

   As for the characters in *k*, you must treat 'A' and 'a' as 0, 'B' and 'b' as 1, . . . , and 'Z' and 'z' as 25, just as we did in lecture. In addition, your program must only apply Vigenère's cipher to a character in *p* if that character is a letter. All other characters (numbers, symbols, spaces, punctuation marks, *etc.*) must be outputted unchanged. Moreover, if your code is about to apply the i^{th} character of *k* to the j^{th} character of *p*, but the latter proves to be a non-alphabetical character, you must wait to apply that i^{th} character of *k* to the next alphabetical character in *p*—you must not yet advance to the next character in *k*. Finally, your program must preserve the case of each letter in *p*.

   Functionally, your program might resemble the below. Underlined are some sample inputs.

```
username@nice (~/cs50/ps2): vigenere FOOBAR
Plaintext:  HELLO, WORLD
Ciphertext: MSZMO, NTFZE
```

   How to test your program, besides predicting what it should output, given some input? Well, recall that we're nice people. And so we've written a program called `devigenere` that also takes one and only one command-line argument (a keyword) but whose job is to take ciphertext as input and produce plaintext as output.

To use our program, execute

```
devigenere k
```

at your prompt, where `k` is some keyword.[2] Presumably you'll want to paste your program's output as input to our program; be sure, of course, to use the same key.

Submitting Your Work.

5. (1 point.) Okay, the free point is back.

6. Ensure that your work is in `~/cs50/ps2/`. Submit your work by executing the command below.

   ```
   cs50submit ps2
   ```

   Thereafter, follow any on-screen instructions until you receive visual confirmation of your work's successful submission. You will also receive a "receipt" via email to your FAS account, which you should retain until term's end. You may re-submit as many times as you'd like; each resubmission will overwrite any previous submission. But take care not to re-submit after the problem set's deadline, as only your latest submission's timestamp is retained.

[2] Even though `devigenere` is installed in our account, we've configured your FAS account to know where it is.