Problem Set 2: Crypto

due by noon on Thu 9/22

Per the directions at this document's end, submitting this problem set involves submitting source code via `submit50` as well as filling out a Web-based, which may take a few minutes, so best not to wait until the very last minute, lest you spend a late day unnecessarily.

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

Goals.

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

Recommended Reading.

- 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 in writing by the course's instructor. Collaboration in the completion of problem sets is not permitted unless otherwise stated by some problem set's specification.

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, 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 or soliciting the work of another individual. Similarly is dual submission academic dishonesty: you may not submit the same or similar work to this course that you have submitted or will submit to another. Nor may you provide or make available solutions to problem sets to individuals who take or may take this course in the future. 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.

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 course's instructor.

You may turn to the Web for instruction beyond the course's lectures and sections, for references, and for solutions to technical difficulties, but not for outright solutions to problems on problem sets or your own final project. However, failure to cite (as with comments) the origin of any code or technique that you do discover outside of the course's lectures and sections (even while respecting these constraints) and then integrate into your own work may be considered academic dishonesty.

All forms of academic dishonesty are dealt with harshly. If the course refers some matter to the Administrative Board and the outcome for some student is *Admonish*, *Probation*, *Requirement to Withdraw*, or *Recommendation to Dismiss*, the course reserves the right to impose local sanctions on top of that outcome for that student that may include, but not be limited to, a failing grade for work submitted or for the course itself.

Grades.

Your work on this problem set will be evaluated along four axes primarily.

Scope. To what extent does your code implement the features required by our specification?
Correctness. To what extent is your code consistent with our specifications and free of bugs?
Design. To what extent is your code written well (*i.e.*, clearly, efficiently, elegantly, and/or logically)?
Style. To what extent is your code readable (*i.e.*, commented and indented with variables aptly named)?

All students, whether taking the course Pass/Fail or for a letter grade, must ordinarily submit this and all other problem sets to be eligible for a passing grade (*i.e.*, Pass or A to D–) unless granted an exception in writing by the course's instructor.

Help!

☐ Surf on over to

`http://help.cs50.net/`

and log in if prompted. Then take a look around!

Henceforth, consider `help.cs50.net` *the* place to turn to anytime you have questions. Not only can you post questions of your own, you can also search for or browse answers to questions already asked by others.

It is expected, of course, that you respect the course's policies on academic honesty. Posting snippets of code about which you have questions is generally fine. Posting entire programs, even if broken, is definitely not. If in doubt, simply flag your discussion as "private," particularly if you need to show us most or all of your code. But the more questions you ask publicly, the more others will benefit as well!

Lest you feel uncomfortable posting, know that students' posts to the course's bulletin board are anonymized. Only the staff, not fellow students, will know who you are. Certainly don't hesitate to post a question because you think that it's "dumb." It is not!

Sanity Check.

☐ You should already have the CS50 Appliance installed, per Problem Set 1. But be sure that you have version 2.3. To check, launch VirtualBox (as by double-clicking its icon wherever it's installed), and in VirtualBox's lefthand menu should be **CS50 Appliance 2.3**. If you instead see an older version (*e.g.*, **CS50 Appliance 2.1**), head to `https://manual.cs50.net/FAQs` for instructions on how to upgrade to version 2.3.

Tips.

☐ If you have a computer that's a few years old or a netbook (in which case your CPU might be on the slower side and your RAM on the lower side), head to `https://manual.cs50.net/FAQs` for tips on how to improve the appliance's performance if you're finding it slow. If you're finding that the appliance is too slow to even be usable on your computer, email `sysadmins@cs50.net` to inquire about options.

☐ Realize that the CS50 Appliance is a computer, albeit a virtual one. For better or for worse (mostly worse), computers don't like to be forcibly shut down or otherwise interrupted while in the middle of something. Do take care, then, not to quit VirtualBox, shutdown your own computer, or even close your laptop's lid while the appliance is in the middle of something (*e.g.*, downloading or installing updates, submitting your work*, etc.*) Best to wait until the appliance isn't doing anything important, then shut it down (as via the green icon in the appliance's bottom-right corner). Bad things can happen, too, if your own computer runs out of disk space, so beware

downloading big files on your own computer if you know you're low on disk space while the appliance is running.

☐ When running, the CS50 Appliance "borrows" some of your computer's own RAM and CPU cycles, which can slow down programs on your computer and vice versa. For maximum performance, try to launch VirtualBox and the appliance before launching other programs on your computer, and try to minimize the number of programs running on your computer while the appliance is running.

With that said, if you have lots of RAM (*e.g.*, 4GB) and lots of CPU cycles (*e.g.*, 2.0GHz), you might not need to give any of this a second thought!

☐ We've chased down and corrected almost all of the problems that folks ran into with the appliance during Problem Set 1. But at least one quirk remains, whereby if you double-click a `.c` file on your desktop (or somewhere in John Harvard's home directory), gedit might not actually open, though your cursor might start to spin. If you run into that issue, simply launch gedit via **Menu > Programming > gedit** (or via its icon in the appliance's bottom-left corner), then open the file in question via **File > Open…**.

☐ If you run into some technical difficulty with the appliance itself, unrelated to C code, do consult `https://manual.cs50.net/FAQs` first, followed by `http://help.cs50.net/`. Anytime you post to `help.cs50.net`, do take care to be as specific as possible, mentioning your OS (and version thereof), your symptoms, and what interventions you've already tried. "It's not working" isn't quite enough detail for us to know how we can help!

☐ Just to be safe, do get into the habit of backing up your `.c` files from time to time, whether to your own hard drive, to `dropbox.com`, or to CS50's servers. To back up your files to your own hard drive (outside of the appliance), see:

`https://manual.cs50.net/CS50_Appliance_2.3#How_to_Transfer_Files_between_Appliance_and_Your_Computer`

To synchronize with `dropbox.com`, see:

`https://manual.cs50.net/Appliance#How_to_Synchronize_Files_with_Dropbox`

To back up your files to CS50's servers, simply run `submit50` (per this document's end), as though you're submitting your work. You can re-submit as many times as you'd like (prior to the problem set's deadline), so you might as well submit as you go, just in case something goes wrong with your computer or appliance! If you need to retrieve your files from CS50's servers, consult `https://manual.cs50.net/FAQs`!

We'll soon introduce you to other techniques!

Getting Started.

☐ Alright, here we go!

Launch VirtualBox (as by double-clicking its icon wherever it's installed), and then boot the appliance (as by single-clicking it in VirtualBox's lefthand menu, and then clicking **Start**).

Upon reaching John Harvard's desktop, open a terminal window (remember how?) and type the below, followed by Enter:

```
sudo yum -y update
```

Input **crimson** if prompted for John Harvard's password. For security, you won't see any characters as you type. That command essentially does what **Menu > Administration > Software Update** does, but you'll see in more detail what's going on.

Realize that updating the appliance in this manner requires Internet access. If on a slow connection (or computer), it might take a few minutes to update the appliance. Don't worry if the process seems to hang if it decides to update a "package" called `cs50-appliance`; that one can take several minutes.

If you see messages like **Couldn't resolve host** or **Cannot retrieve metalink for repository**, those simply mean that the appliance doesn't currently have Internet access. Sometimes that happens if you've just awakened your computer from sleep or perhaps changed from wireless to wired Internet or vice versa. If your own computer does have Internet access (which you can confirm by trying to visit some website in a browser on your own computer) but the appliance does not (which you can confirm by trying to visit the same with Firefox within the appliance), try restarting the appliance (as by clicking the green icon in its bottom-right corner, then clicking **Restart**). If, upon restart, the appliance still doesn't have Internet access, head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` for help!

Once the appliance has been updated, you should see **Complete!** in your terminal window. If there was nothing to update, you'll see **No packages marked for Update** instead.

☐ Juuuuuuuuust to be sure that everything worked, go ahead and execute that very same command again in a terminal window (though not while its first invocation is still running):

```
sudo yum -y update
```

Again, input **crimson** if prompted for John Harvard's password. (If only a few minutes have passed since the last update, you might not even be prompted.) You should now see **No packages marked for Update**, which means that your appliance is now up-to-date! If you see some error instead, try once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ Alright, here we go for real! Open a terminal window if not open already (whether by opening gedit via **Menu > Programming > gedit** or by opening Terminal itself via **Menu > Programming > Terminal**). Then execute

```
mkdir ~/pset2
```

at your prompt in order to make a directory called `pset2` in your home directory.[1] Take care not to overlook the space between `mkdir` and `~/pset2` or any other character for that matter! Recall that `~` denotes your home directory, and thus `~/pset2` denotes a directory called `pset2` therein.

Now execute

```
cd ~/pset2
```

to move yourself into (*i.e.*, open) that directory. Your prompt should now resemble the below.

```
jharvard@appliance (~/pset2):
```

If not, retrace your steps and see if you can determine where you went wrong. You can actually execute

```
history
```

at the prompt to see your last several commands in chronological order if you'd like to do some sleuthing. You can also scroll through the same one line at a time by hitting your keyboard's up and down arrows; hit Enter to re-execute any command that you'd like. If still unsure how to fix, remember that `help.cs50.net` is your friend!

All of the work that you do for this problem set must ultimately reside in your `pset2` directory for submission.

Let's Warm Up with a Song.

☐ 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
```

[1] If you decide to use `dropbox.com`, you can instead store your files in, say, `~/Dropbox/pset2`.

```
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." Your version should be capitalized and spelled exactly as ours is.

Notice, though, the repetition in this song's verses. Perhaps you could leverage a loop that iterates from 1 to 10 (or 0 to 9) to generate them? Though they do vary somewhat, so you might need some conditions? Seems you could even implement a couple of functions that take, as input, an integer and return, as output, a string? Or maybe you could store all those strings in arrays? Hm. So many possibilities!

There are, as you may be increasingly aware, many ways to solve problems like this one. Pick an approach, implement it, test it, then go back and see if you can improve it before moving on! Ultimately, not only should your code be correct (*i.e.*, work right), it should also manifest good design and good style. Rest assured that there are many ways to implement this song.

Style is easy. Ask yourself questions like these: Is my code well commented, without being excessively so? Is my code "pretty-printed" (*i.e.*, consistently indented)? Are my variables aptly named? Refer back to `https://manual.cs50.net/Style` as needed.

As for design, ask yourself questions like these: Is my code straightforward to read? Am I wasting CPU cycles unnecessarily? Is my code more complicated than it need be to get this job done?

And don't forget scope: be sure that you've checked off (if mentally) each of this specification's checkboxes!

Consider yourself done with this problem when you feel there's no more room for improvement!

Your program will need, at least, a `main` function. It's up to you to decide whether or not you want to write one or more additional functions that `main` calls. Remember that you can compile your program with:

```
make oldman
```

And you can run it with:

```
./oldman
```

As this usage implies, `oldman` need not accept any "command-line arguments." And so it suffices to declare `main` with

```
int
main(void)
```

without any mention of `argc` or `argv`. If you'd like to play with the staff's own implementation of `oldman` in the appliance, you may execute the below.

```
~cs50/pset2/oldman
```

Alright, off you go! Don't forget to back up your files (as to your own hard drive, to `dropbox.com`, or to CS50's servers with `submit50`)!

Hail, Caesar!

☐ Recall from the end of Week 2 that Caesar's cipher encrypts (*i.e.*, scrambles in a reversible way) messages by "rotating" each letter by *k* positions, wrapping around from 'Z' to 'A' as needed:

`http://en.wikipedia.org/wiki/Caesar_cipher`

In other words, if *p* is some plaintext (*i.e.*, an unencrypted message), p_i is the i^{th} character in *p*, and *k* is a secret key (*i.e.*, a non-negative integer), then each letter, c_i, in the ciphertext, *c*, is computed as:

$c_i = (p_i + k) \% 26$

This formula perhaps makes the cipher seem more complicated than it is, but it's really just a nice way of expressing the algorithm precisely and concisely. And computer scientists love precision and, er, concision.[2]

For example, suppose that the secret key, *k*, is 13 and that the plaintext, *p*, is "be sure to drink your Ovaltine." Let's encrypt that *p* with that *k* in order to get the ciphertext, *c*, by rotating each of the letters in *p* by 13 places:

```
plaintext:  Be sure to drink your Ovaltine!
ciphertext: Or fher gb qevax lbhe Binygvar!
```

We've deliberately printed the above in a monospaced font so that all of the letters line up nicely. Notice how `O` (the first letter in the ciphertext) is 13 letters away from `B` (the first letter in the plaintext). Similarly is `r` (the second letter in the ciphertext) 13 letters away from `e` (the second letter in the plaintext). Meanwhile, `f` (the third letter in the ciphertext) is 13 letters away from `s` (the third letter in the plaintext), though we had to wrap around from `Z` to `A` to get there. And so on. Not the most secure cipher, to be sure, but fun to implement!

Incidentally, a Caesar cipher with a key of 13 is generally called ROT13:

`http://en.wikipedia.org/wiki/ROT13`

In the real world, though, it's probably best to use ROT26, which is believed to be twice as secure.[3]

Anyhow, your next goal 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. Let's call it *k*. If your program is executed without any command-line arguments or with more than one command-line argument, your program should yell at the user and return a value of `1` (which tends to signify an error) immediately as via the statement below:

```
return 1;
```

[2] Okay, fine, conciseness. So much for parallelism.

[3] `http://www.urbandictionary.com/define.php?term=ROT26`

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, with `main` returning `0`.

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 $2^{31} - 26$. (In other words, you don't need to worry if your program eventually breaks if the user chooses a value for *k* that's too big or almost too big to fit in an `int`. Now, 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, as a computer scientists might say. 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.

Where to begin? Well, this program needs to accept a command-line argument, *k*, so this time you'll want to declare `main` with:

```
int
main(int argc, char *argv[])
```

Recall that `argv` is an "array" of strings (which are otherwise known as "char stars" for reasons we'll soon see). In fact, because `string` is just a synonym for `char *`, thanks to the CS50 Library, you could actually declare `main` with

```
int
main(int argc, string argv[])
```

if you find that syntax more clear. Either way, you can think of an array as row of gym lockers, inside each of which is some value (and maybe some socks). In this case, inside each such locker is a `string`. To open (*i.e.*, "index into") the first locker, you use syntax like `argv[0]`, since arrays are "zero-indexed." To open the next locker, you use syntax like `argv[1]`. And so on. Of course, if there are `n` lockers, you'd better stop opening lockers once you get to `argv[n-1]`, since `argv[n]` doesn't exist! (That or it belongs to someone else, in which case you still shouldn't open it.)

And so you can access *k* with code like

```
string k = argv[1];
```

assuming it's actually there! Recall that `argc` is an `int` that equals the number of strings that are in `argv`, so you'd best check the value of `argc` before opening a locker that might not exist! Ideally, `argc` will be 2. Why? Well, recall that inside of `argv[0]`, by default, is a program's own name. So `argc` will always be at least `1`. But for this program you want the user to provide a command-line argument, *k*, in which case `argc` should be `2`. Of course, if the user provides more

than one command-line argument at the prompt, `argc` could be greater than `2`, in which case it's time for some yelling.

Now, just because the user types an integer at the prompt, that doesn't mean their input will be automatically stored in an `int`. Au contraire, it will be stored as a `string` that just so happens to look like an `int`! And so you'll need to convert that `string` to an actual `int`. As luck would have it, a function, `atoi`, exists for exactly that purposes. Here's how you might use it:

```
int k = atoi(argv[1]);
```

Notice, this time, we've declared *k* as an actual `int` so that you can actually do some arithmetic with it. Ah, much better. Incidentally, you can assume that the user will only type integers at the prompt. You don't have to worry about them typing, say, `foo`, just to be difficult; `atoi` will just return `0` in such cases. Incidentally, you'll need to `#include` a header file other than `cs50.h` and `stdio.h` in order to use of `atoi` without getting yelled at by `gcc`. We leave it to you to figure out which one![4]

Okay, so once you've got *k* stored as an `int`, you'll need to ask the user for some plaintext. Odds are CS50's own `GetString` can help you with that.

Once you have both *k* and some plaintext, it's time to encrypt the latter with the former. Recall that you can iterate over the characters in a `string`, printing each one at a time, with code like the below:

```
for (int i = 0, n = strlen(p); i < n; i++)
{
    printf("%c", p[i]);
}
```

In other words, just as `argv` is an array of strings, so is a `string` an array of characters. And so you can use square brackets to access individual characters in strings just as you can individual strings in `argv`. Neat, eh? Of course, printing each of the characters in a string one at a time isn't exactly cryptography. Well, maybe technically if *k* = 0. But the above should help you help Caesar implement his cipher! For Caesar!

Incidentally, you'll need to `#include` yet another header file in order to use `strlen`.[5]

So that we can automate some tests of your code, your program must behave per the below. Assumed that the boldfaced text is what some user has typed.

```
jharvard@appliance (~/pset2): ./caesar 13
Be sure to drink your Ovaltine!
Or fher gb qevax lbhe Binygvar!
```

[4] https://www.cs50.net/resources/cppreference.com/stdstring/atoi.html
[5] https://www.cs50.net/resources/cppreference.com/stdstring/strlen.html

Besides `atoi`, you might find some handy functions documented at:

`http://www.cs50.net/resources/cppreference.com/stdstring/`

For instance, `isdigit` sounds interesting. And, with regard to wrapping around from `Z` to `A`, don't forget about `%`. You might also want to check out `http://asciitable.com/`, which reveals the ASCII codes for more than just alphabetical characters, just in case you find yourself printing some characters accidentally.

If you'd like to play with the staff's own implementation of `caesar` in the appliance, you may execute the below.

`~cs50/pset2/caesar`

Don't forget to back up your files (as to your own hard drive, to `dropbox.com`, or to CS50's servers with `submit50`)!

☐ `uggc://jjj.lbhghor.pbz/jngpu?i=bUt5FWLEUN0`

Parlez-vous français?

☐ Well that last cipher was hardly secure. Fortunately, per Week 3's first lecture, there's a more sophisticated algorithm out there: Vigenère's. It is, of course, French:[6]

`http://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher`

Vigenère's cipher improves upon Caesar's by encrypting messages using a sequence of keys (or, put another way, a keyword). In other words, if p is some plaintext and k is a keyword (*i.e.*, an alphbetical string, whereby `A` and `a` represent 0, while `Z` and `z` represent 25), then each letter, c_i, in the ciphertext, c, is computed as:

$c_i = (p_i + k_j) \% 26$

Note this cipher's use of k_j as opposed to just k. And recall that, if k is shorter than p, then the letters in k must be reused cyclically as many times as it takes to encrypt p.

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, with main returning `1` (thereby signifying an error that our own tests can detect). Otherwise, your program must proceed to prompt the user for a string of plaintext, p,

[6] Do not be mislead by the article's discussion of a *tabula recta*. Each c_i can be computed with relatively simple arithmetic! You do not need a two-dimensional array.

which it must then encrypt according to Vigenère's cipher with *k*, ultimately printing the result and exiting, with `main` returning `0`.

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. 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 j^{th} character of *k* to the i^{th} character of *p*, but the latter proves to be a non-alphabetical character, you must wait to apply that j^{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*.

Not sure where to begin? As luck would have it, this program's pretty similar to `caesar`! Only this time, you need to decide which character in *k* to use as you iterate from character to character in *p*.

So that we can automate some tests of your code, your program must behave per the below; highlighted in bold are some sample inputs.

```
jharvard@appliance (~/pset2): ./vigenere FOOBAR
HELLO, WORLD
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

```
~cs50/pset2/devigenere k
```

at your prompt, where `k` is some keyword. Presumably you'll want to paste your program's output as input to our program; be sure, of course, to use the same key. Note that you do not need to implement `devigenere` yourself, only `vigenere`.

If you'd like to play with the staff's own implementation of `vigenere` in the appliance, you may execute the below.

```
~cs50/pset2/vigenere
```

Don't forget to back up your files (as to your own hard drive, to `dropbox.com`, or to CS50's servers with `submit50`)!

How to Submit.

In order to submit this problem set, you must first execute a command in the appliance and then submit a (brief) form online.

☐ Recall that you obtained a CS50 Cloud account (*i.e.*, username and password) for Problem Set 1. If you don't remember your username and/or password, head to `https://cloud.cs50.net/` look up the former and/or change the latter. You'll be prompted to log in with your HUID (or XID) and PIN.

☐ Just in case we updated `submit50` since you started this problem set, open a terminal window and execute the below, inputting **crimson** if prompted for John Harvard's password.

```
sudo yum -y update
```

If there was something to update, you should see **Complete!** after a few seconds or minutes. If there was nothing to update, you should instead see **No packages marked for Update**. If you see any errors, try the command once more, try to restart the appliance and then try once more, then head to `https://manual.cs50.net/FAQs` followed by `http://help.cs50.net/` as needed for help!

☐ To actually submit, first open a terminal window and execute:[7]

```
cd ~/pset2
```

Then execute:

```
ls
```

At a minimum, you should see `oldman.c`, `caesar.c`, and `vigenere.c`, capitalized and spelled exactly like that. If not, odds are you skipped some step(s) earlier! In particular, if you misnamed some file (*e.g.*, `vigenere.c` as `Vigenere.c`), know that you can rename it with a command like

```
mv Vigenere.c vigenere.c
```

where `mv`'s first command-line argument is the file's current name and `mv`'s second command-line argument is the file's new name. If you don't see any or all of your files, you might have saved them somewhere else accidentally. Poke around John Harvard's desktop and home directory, and drag files as needed into the `pset2` directory that should be in his home directory.

If everything is as it should be, you are ready to submit your source code to us. Execute:[8]

```
submit50 ~/pset2
```

[7] Unless you decided to use `dropbox.com` and stored your files in, say, `~/Dropbox/pset2`.
[8] *Ibid.*

When prompted for **Course**, input **cs50**; when prompted for **Repository**, input **pset2**. When prompted for a username and password, input your CS50 Cloud username and password. For security, you won't see your password as you type it. That command will essentially upload your entire `~/pset2` directory to CS50's repository, where your TF will be able to access it. The command will inform you whether your submission was successful or not. If provided with the URL of a PDF of your code (which further confirms its submission), right-click (or ctrl-click) the link, then choose **Open Link** from the menu that appears to open the PDF in Document Viewer.

You may re-submit as many times as you'd like; we'll grade your most recent submission. But take care not to submit after the problem set's deadline, lest you spend a late day unnecessarily or risk rejection entirely.

If you run into any trouble at all, let us know via `help.cs50.net` and we'll try to assist! Just take care to seek help well before the problem set's deadline, as we can't always reply within minutes!

☐ Head to the URL below where a short form awaits:

`https://www.cs50.net/psets/2/`

Once you have submitted that form (as well as your source code), you are done!

This was Problem Set 2.