Problem Set 1: C

due by noon on Thu 9/15

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

Do take advantage of Week 2's office hours as well Week 1's supersections (or video thereof at `https://www.cs50.net/sections/`).

If you have any questions or trouble, head to `http://help.cs50.net/`.

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

Goals.

- Get comfortable with Linux.
- Start thinking more carefully.
- Solve some problems in C.

Recommended Reading.

- Sections 1 – 7, 9, and 10 of `http://www.howstuffworks.com/c.htm`.
- Chapters 1 – 6 of *Programming in C*.

diff hacker1.pdf hacker1.pdf.

- Hacker Edition expects commas in currency.
- Hacker Edition plays with credit cards instead of coins.
- Hacker Edition demands vertical bars instead of horizontal ones.

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.

Getting Started.

☑ Take CS50.

☐ Recall that the CS50 Appliance is a "virtual machine" (running an operating system called Fedora, which itself is a flavor of Linux) that you can run inside of a window on your own computer, whether you run Windows, Mac OS, or even Linux itself. To do so, all you need is a "hypervisor" (otherwise known as a "virtual machine monitor"), software that tricks the appliance into thinking that it's running on "bare metal." One such hypervisor is called VirtualBox, made by Oracle (formerly Sun Microsystems), which just so happens to be free!

Alternatively, you could buy a new computer, install Fedora on it (*i.e.*, bare metal), and use that! But VirtualBox lets you do all that for free with whatever computer you already have. Plus, the CS50 Appliance is pre-configured for CS50, so, as soon as you install it, you can hit the ground running.

So let's get both VirtualBox and the CS50 Appliance installed. Head to

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

where instructions await. If you run into any problems whatsoever, simply post to the **Appliance** category at `help.cs50.net`!

☐ Once you have the CS50 Appliance installed, go ahead and start it (as by launching VirtualBox, clicking the appliance in VirtualBox's lefthand menu, then clicking **Start**). A small window should open, inside of which the appliance should boot. A few seconds or minutes later, you should find yourself logged in as John Harvard (whose username is **jharvard** and whose password is **crimson**), with John Harvard's desktop before you.

If you find that the appliance runs unbearably slow on your computer, particularly if several years old or a somewhat slow netbook, let us know via `help.cs50.net`, and we'll offer some tips on boosting its speed.

By default, the appliance's resolution is only 800 × 600 (*i.e.*, 800 pixels wide by 600 pixels tall), in case you have a small screen. But you can increase it to 1024 × 768 via **Menu > Preferences > Display** if you'd like.[1]

Feel free to poke around, particularly the CS50 Menu in the appliance's bottom-left corner. You should find the graphical user interface (GUI), called Xfce, reminiscent of both Mac OS and Windows. Linux actually comes with a bunch of GUIs; Xfce is just one. If you're already familiar with Linux, you're welcome to install other software via **Menu > Administration > Add/Remove Software**, but the appliance should have everything you need for now. You're also welcome to play with the appliance's various features, per the instructions at

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

[1] To increase its resolution further, see `https://manual.cs50.net/Appliance#How_to_Change_Resolution`.

but this problem set will explicitly mention anything that you need know or do.

Notice, though, that the appliance will "capture" your trackpad or mouse, whereby once you've clicked inside of the appliance, you can no longer move your cursor (*i.e.*, arrow) outside of the appliance's window! Not to worry. To release your cursor from the appliance's clutches, simply hit VirtualBox's "host key" on your keyboard: on Mac OS, hit left-⌘; on Windows or Linux, hit right-Ctrl. Once you do, you should have full control of your trackpad or mouse again.

☐ Even if you just downloaded the appliance, ensure that it's completely up-to-date by selecting **Menu > Administration > Software Update**. If updates are indeed available, click **Install Updates**. If prompted with **Additional confirmation required**, click **Continue**. If warned that the **software is not from a trusted source** and prompted for a password, input **crimson**, then click **Authenticate**. If prompted a few seconds or minutes later to **log out and back in**, click **Log Out** and then log back in as John Harvard, when prompted, with username **jharvard** and password **crimson**.

☐ Okay, let's create a folder (otherwise known as a "directory") in which your code for this problem set will soon live. Go ahead and double-click **Home** on John Harvard's desktop (in the appliance's top-left corner). A window entitled **jharvard - File Manager** should appear, indicating that you're inside of John Harvard's "home directory" (*i.e.*, personal folder). Be sure that **jharvard** is indeed highlighted in the window's top-left corner, then select **File > Create Folder…** and input **hacker1** (in all lowercase, with no spaces) when prompted for a new name. Then click **Create**. A new folder called **hacker1** should appear in the window. Go ahead and double-click it. The window's title should change to **hacker1 - File Manager**, and you should see an otherwise empty folder (since you just created it). Notice, though, that atop the window are two buttons, **jharvard** and **hacker1**, that indicate where you were and where you are, respectively; you can click buttons like those to navigate back and forth easily.

☐ Okay, go ahead and close any open windows, then select **Menu > Programming > gedit**. (Recall that the CS50 Menu is in the appliance's bottom-left corner.) A window entitled **Unsaved Document 1 - gedit** should appear, inside of which is a tab entitled **Unsaved Document 1**. Clearly the document is just begging to be saved. Go ahead and type `hello` (or the ever-popular `asdf`) in the tab, and then notice how the tab's name is now prefixed with an asterisk (*), indicating that you've made changes since the file was first opened. Select **File > Save**, and a window entitled **Save As** should appear. Input `hello.txt` next to **Name**, then click **jharvard** under **Places**. You should then see the contents of John Harvard's home directory, namely **Desktop** and **hacker1**. Double-click **hacker1**, and you should find yourself inside that empty folder you created. Now, at the bottom of this same window, you should see that the file's default **Character Encoding** is **Unicode (UTF-8)** and that the file's default **Line Ending** is **Unix/Linux**. No need to change either; just notice they're there. That the file's **Line Ending** is **Unix/Linux** just means that gedit will insert (invisibly) `\n` at the end of any line of text that you type. Windows, by contrast, uses `\r\n`, and Mac OS uses `\r`, but more on those details some other time.

Okay, click **Save** in the window's bottom-right corner. The window should close, and you should see that the original window's title is now **hello.txt (~/hacker1) - gedit**. The parenthetical just means that **hello.txt** is inside of **hacker1**, which itself is inside of **~**, which is shorthand notation for

John Harvard's home directory. A useful reminder is all. The tab, meanwhile, should now be entitled **hello.txt** (with no asterisk, unless you accidentally hit the keyboard again).

☐ Okay, with **hello.txt** still open in gedit, notice that beneath your document is a "terminal window," a command-line (*i.e.*, text-based) interface via which you can navigate the appliance's hard drive and run programs (by typing their name). Notice that the window's "prompt" is

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

which means that you are logged into the appliance as John Harvard and that you are currently inside of **~** (*i.e.*, John Harvard's home directory). If that's the case, there should be a **hacker1** directory somewhere inside. Let's confirm as much.

Click somewhere inside of that terminal window, and the prompt should start to blink. Type

```
ls
```

and then Enter. That's a lowercase L and a lowercase S, which is shorthand notation for "list." Indeed, you should then see a (short!) list of the folders inside of John Harvard's home directory, namely **Desktop** and **hacker1**! Let's open the latter. Type

```
cd hacker1
```

or even

```
cd ~/hacker1
```

followed by Enter to change your directory to **hacker1** (ergo, `cd`). You should find that your prompt changes to

```
jharvard@appliance (~/hacker1):
```

confirming that you are indeed now inside of **~/hacker1** (*i.e.*, a directory called **hacker1** inside of John Harvard's home directory). Now type

```
ls
```

followed by Enter. You should see **hello.txt**! Now, you can't click or double-click on that file's name there; it's just text. But that listing does confirm that **hello.txt** is where we hoped it would be.

Let's poke around a bit more. Go ahead and type

```
cd
```

and then Enter. If you don't provide `cd` with a "command-line argument" (*i.e.*, a directory's name), it whisks you back to your home directory by default. Indeed, your prompt should now be:

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

Phew, home sweet home. Make sense? If not, no worries; it soon will! It's in this terminal window that you'll soon be compiling your first program! For now, though, close gedit (via **File > Quit**) and, with it, **hello.txt**.

Incidentally, if you encounter an issue whereby clicking icons on John Harvard's desktop (or in John Harvard's home directory or in **hacker1**) fails to trigger gedit to open, even if those files end in .c or .txt. (Instead, you may only see a spinning cursor.) If so, not to worry. Simply launch gedit via **Menu > Programming > gedit**, and then open the file in question manually via **File > Open**.

O hai, world!

☐ Shall we have you write your first program?

Okay, go ahead and launch gedit. (Remember how?) You should find yourself faced with another **Unsaved Document 1**. Go ahead and save the file as **hello.c** (not **hello.txt**) inside of **hacker1**, just as before. (Remember how?) Once the file is saved, the window's title should change to **hello.c (~/hacker1) - gedit**, and the tab's title should change to **hello.c**. (If either does not, best to close gedit and start fresh! Or ask for help!)

Go ahead and write your first program by typing these lines into the file (though you're welcome to change the words between quotes to whatever you'd like):[2]

```
#include <stdio.h>

int
main(void)
{
    printf("hello, world!\n");
}
```

Notice how gedit adds "syntax highlighting" (*i.e.*, color) as you type. Those colors aren't actually saved inside of the file itself; they're just added by gedit to make certain syntax stand out. Had you not saved the file as **hello.c** from the start, gedit wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent. Notice, too, that gedit sometimes tries to help you along by completing your thought: you should find that, when you type that first curly parenthesis (and curly brace), the second appears for you automatically.[3]

Do be sure that you type in this program just right, else you're about to experience your first bug! In particular, capitalization matters, so don't accidentally capitalize words (unless they're between those two quotes). And don't overlook that one semicolon. C is quite nitpicky!

[2] Do type in this program keystroke by keystroke inside of the appliance; don't try to copy/paste from the PDF! Odds are copy/paste won't work yet anyway until you install "Guest Additions," but more on those some other time!
[3] If you find that annoying, you can disable the feature by unchecking **Edit > Preferences > Plugins > Bracket Completion**.

When done typing, select **File > Save** (or hit ctrl-s), but don't quit. Recall that the leading asterisk in the tab's name should then disappear. Click anywhere in the terminal window beneath your code, and its prompt should start blinking. But odds are the prompt itself is just

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

which means that, so far as the terminal window's concerned, you're still inside of John Harvard's home directory, even though you saved the program you just wrote inside of **~/hacker1** (per the top of gedit's window). No problem, go ahead and type

```
cd hacker1
```

or

```
cd ~/hacker1
```

at the prompt, and the prompt should change to

```
jharvard@appliance (~/hacker1):
```

in which case you're where you should be! Let's confirm that `hello.c` is there. Type

```
ls
```

at the prompt followed by Enter, and you should see both `hello.c` and `hello.txt`? If not, no worries; you probably just missed a small step. Best to restart these past several steps or ask for help!

Assuming you indeed see `hello.c`, let's try to compile! Cross your fingers and then type

```
make hello
```

at the prompt, followed by Enter. (Well, maybe don't cross your fingers whilst typing.) To be clear, type only `hello` here, not `hello.c`. If all that you see is another, identical prompt, that means it worked! Your source code has been translated to 0s and 1s that you can now execute. Type

```
./hello
```

at your prompt, followed by Enter, and you should see whatever message you wrote between quotes in your code! Indeed, if you type

```
ls
```

followed by Enter, you should see a new file, `hello`, alongside `hello.c` and `hello.txt`.

If, though, upon running `make`, you instead see some error(s), it's time to debug! (If the terminal window's too small to see everything, click and drag its top border upward to increase its height.)

If you see an error like **expected declaration** or something no less mysterious, odds are you made a syntax error (*i.e.*, typo) by omitting some character or adding something in the wrong place. Scour your code for any differences vis-à-vis the template above. It's easy to miss the slightest of things when learning to program, so do compare your code against ours character by character; odds are the mistake(s) will jump out! Anytime you make changes to your own code, just remember to re-save via **File > Save** (or ctrl-s), then re-click inside of the terminal window, and then re-type

```
make hello
```

at your prompt, followed by Enter. (Just be sure that you are inside of **~/hacker1** within your terminal window, as your prompt will confirm or deny.) If you see no more errors, try running your program by typing

```
./hello
```

at your prompt, followed by Enter! Hopefully you now see the greeting you wrote? If not, reach out to `help.cs50.net` for help! In fact, if you log into `help.cs50.net` within the appliance itself (via **Menu > Internet > Firefox**), you can even attach your code to your post; just take care to flag it as private.

Incidentally, if you find gedit's built-in terminal window too small for your tastes, know that you can open one in its own window via **Menu > Programming > Terminal**. You can then alternate between gedit and Terminal as needed, as by clicking either's name along the appliance's bottom.

Woo hoo! You've begun to program!

Let's take a short break.

Story Time.

☐ We explored in Week 1 how hard drives work, but computers actually have a few types of memory (*i.e.*, storage), among them level-1 cache, level-2 cache, RAM, and ROM. Curl up with the article below to learn a bit about each:

```
http://computer.howstuffworks.com/computer-memory.htm
```

Odds are you'll want to peruse, at least, pages 1 through 5 of that article.

That's it for now. Bet this topic comes up again, though!

☐ Recall that "style" generally refers to source code's aesthetics, the extent to which code is readable (*i.e.*, commented and indented with variables aptly named). Odds are you didn't have to give too much thought to style when writing `hello.c`, given its brevity, but you're about to start writing programs where you'll need to make some stylistic decisions.

Before you do, read over CS50's Style Guide:

`https://manual.cs50.net/Style`

Take the Pennies.

☐ Recall from Week 0 that, if given the choice between $10,000,000 or a month's worth of pennies, whereby you receive a penny the first day, two pennies the second, four pennies the third, and so forth… take the pennies. (By contrast, if you receive an email from some stranger on the Internet offering you an opportunity to double your money, maybe think twice.)

Anyhow, why the pennies? Exponentiation. Those pennies add up! Consider how many pennies you'd receive on the 31st day alone, not to mention on the days leading up to it:

$1 \times 2 = 1073741824$

Put more succinctly, that's 1×2^{30}. Convert those pennies to dollars (by dividing by 100) and you get, what, over $10,000,000? Crazy.

What if you were given more than one penny on that first day? Or the month were February, in which case you'd get shortchanged a few million? (Best to take the pennies in January, March, May, July, August, October, or December.[4]) Let's find out.

Implement, in a file called `pennies.c`, a program that first asks the user how many days there are in the month and then asks the user how many pennies he or she will receive on the first day of that month. The program should then calculate the amount that the user will have received in total by the end of the month (not just on the last day) if that amount is doubled on every day but the first, expressed not as pennies but as dollars and cents, with dollars formatted with commas every three digits. If the user does not type in 28, 29, 30, or 31 for the number of days in the month, the program should prompt the user to retry. If the user does not input a positive integer for the first day's number of pennies, the program should prompt the user to retry.

For instance, your program might behave as follows, whereby boldfaced text represents some user's input.

```
jharvard@appliance (~/hacker1): ./pennies
Days in month: 32
Days in month: 31
Pennies on first day: 1
$21,474,836.47
```

[4] I'll admit it. I had to look at a calendar.

Notice how this output suggests that the program should indeed re-prompt the user if he or she fails to cooperate with these rules (as by inputting too many days). And notice how the dollars have been formatted with commas.

How to begin? Well, as before, start by opening gedit and saving an otherwise blank file as `pennies.c`. Then, fill the file with some "boilerplate" code like the below:

```
#include <cs50.h>
#include <stdio.h>

int
main(void)
{

}
```

Save the file and, just to be safe, try compiling it with

```
make pennies
```

in your terminal window, just to make sure you didn't make any syntactical mistakes, in which case you'll see errors that will need to be fixed! Then dive back into your code.

Odds are you'll want a couple of loops, one with which to prompt (and potentially re-prompt) the user for a number of days, and another with which to prompt (and potentially re-prompt) the user for a number of first-day pennies. How to get both those numbers? Perhaps the CS50 Library offers some options?

Of course, if you store the user's amount due in an `int` (which is only 32 bits in the CS50 Appliance), the total will be bounded by $2^{31} - 1$ pennies. (Why 2^{31} and not 2^{32}? And why 1 less than 2^{31}?) Best, then, to store your total in a `long long`, so that the user benefits from 64 bits. (Don't worry if users' totals overflow 64 bits and even go negative; consider it punishment for greed!)

Do take care to format the user's total as dollars and cents (to just 2 decimal places), prefixed with a dollar sign, just as we did in the output above. And do remember to insert commas after every 3 digits to the left of the decimal, as you might normally do. (You must implement those commas in code and not via a locale environment variable.) So that we can automate some tests of your code, we ask that your program's last line of output be the amount owed to a user, followed by `\n`. The rest of your program's personality we leave entirely to you!

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

```
~cs50/hacker1/pennies
```

Bad Credit.

☐ Odds are you have a credit card in your wallet. Though perhaps the bill does not (yet) get sent to you! That card has a number, both printed on its face and embedded (perhaps with some other data) in the magnetic stripe on back. That number is also stored in a database somewhere, so that when your card is used to buy something, the creditor knows whom to bill. There are a lot of people with credit cards in this world, so those numbers are pretty long: American Express uses 15-digit numbers, MasterCard uses 16-digit numbers, and Visa uses 13- and 16-digit numbers. And those are decimal numbers (0 through 9), not binary, which means, for instance, that American Express could print as many as 10^{15} = 1,000,000,000,000,000 unique cards![5]

Now that's a bit of an exaggeration, because credit card numbers actually have some structure to them. American Express numbers all start with 34 or 37; MasterCard numbers all start with 51, 52, 53, 54, or 55; and Visa numbers all start with 4. But credit card numbers also have a "checksum" built into them, a mathematical relationship between at least one number and others. That checksum enables computers (or humans who like math) to detect typos (*e.g.*, transpositions), if not fraudulent numbers, without having to query a database, which can be slow. (Consider the awkward silence you may have experienced at some point whilst paying by credit card at a store whose computer uses a dial-up modem to verify your card.) Of course, a dishonest mathematician could certainly craft a fake number that nonetheless respects the mathematical constraint, so a database lookup is still necessary for more rigorous checks.

So what's the secret formula? Well, most cards use an algorithm invented by Hans Peter Luhn, a nice fellow from IBM. According to Luhn's algorithm, you can determine if a credit card number is (syntactically) valid as follows:

i) Multiply every other digit by 2, starting with the number's second-to-last digit, and then add those products' digits together.
ii) Add the sum to the sum of the digits that weren't multiplied by 2.
iii) If the total's last digit is 0 (or, put more formally, if the total modulo 10 is congruent to 0), the number is valid!

That's kind of confusing, so let's try an example with my own AmEx: 378282246310005.

i) For the sake of discussion, let's first underline every other digit, starting with the number's second-to-last digit:

3̲7̲8̲2̲8̲2̲2̲4̲6̲3̲1̲0̲0̲5̲

Okay, let's multiply each of the underlined digits by 2:

7·2 + 2·2 + 2·2 + 4·2 + 3·2 + 0·2 + 0·2

[5] That's, ahem, a quadrillion.

That gives us:

14 + 4 + 4 + 8 + 6 + 0 + 0

Now let's add those products' digits (*i.e.*, not the products themselves) together:

1 + 4 + 4 + 4 + 8 + 6 + 0 + 0 = 27

ii) Now let's add that sum (27) to the sum of the digits that weren't multiplied by 2:

27 + 3 + 8 + 8 + 2 + 6 + 1 + 0 + 5 = 60

iii) Yup, the last digit in that sum (60) is a 0, so David's card is legit![6]

So, validating credit card numbers isn't hard, but it does get a bit tedious by hand. Let's write a program.

In `credit.c`, write a program that prompts the user for a credit card number and then reports (via `printf`) whether it is a valid American Express, MasterCard, or Visa card number, per the definitions of each's format herein. So that we can automate some tests of your code, we ask that your program's last line of output be `AMEX\n` or `MASTERCARD\n` or `VISA\n` or `INVALID\n`, nothing more, nothing less. For simplicity, you may assume that the user's input will be entirely numeric (*i.e.*, devoid of hyphens, as might be printed on an actual card). But do not assume that the user's input will fit in an `int`! Best to use `GetLongLong` from CS50's library to get users' input. (Why?)

Of course, to use `GetLongLong`, you'll need to tell `gcc` about CS50's library. Be sure to put

```
#include <cs50.h>
```

toward the top of `credit.c`. And be sure to compile your code with a command like the below.

```
gcc -o credit credit.c -lcs50
```

Note that `-lcs50` must come at this command's end because of how `gcc` works.

Incidentally, recall that `make` can invoke `gcc` for you and provide that flag for you, as via the command below.

```
make credit
```

Assuming your program compiled without errors (or, ideally, warnings) via either command, run your program with the command below.

```
./credit
```

[6] Hm, maybe this wasn't the best idea.

Consider the below representative of how your own program should behave when passed a valid credit card number (sans hyphens); highlighted in bold is some user's input.

```
jharvard@appliance (~/hacker1): ./credit
Number: 378282246310005
AMEX
```

Of course, `GetLongLong` itself will reject hyphens (and more) anyway:

```
jharvard@appliance (~/hacker1): ./credit
Number: 3782-822-463-10005
Retry: foo
Retry: 378282246310005
AMEX
```

But it's up to you to catch inputs that are not credit card numbers (*e.g.*, my phone number), even if numeric:

```
jharvard@appliance (~/hacker1): ./credit
Number: 6175230925
INVALID
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) Here are a few card numbers that PayPal recommends for testing:

```
https://www.paypalobjects.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm
```

Google (or perhaps a roommate's wallet) should turn up more.[7] If your program behaves incorrectly on some inputs (or doesn't compile at all), have fun debugging!

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

```
~cs50/hacker1/credit
```


[7] If your roommate asks what you're doing, don't mention us.

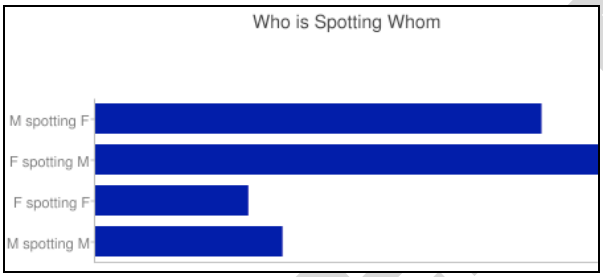
I Saw You.

☐ Surf on over to

`http://isawyouharvard.com/`

where you'll find "your source for posting and browsing missed connections," a website created by CS50's own Tej Toor '10 as her final project her year. Want to let someone special know that you saw him or her the other day? Here's your chance! We won't know it's you.[8]

Anyhow, once we have your attention again, follow the link to **Statistics** atop the site, where you'll find some neat visuals, among them a bar chart. As of the end of Week 1, here's who is spotting whom:



It turns out it's quite easy to integrate such things into a website these days. Tej happens to be using the Google Chart API (a free library of sorts) to generate those visuals:

`http://code.google.com/apis/chart/`

If curious, documentation for bar charts specifically lives at:

`http://code.google.com/apis/chart/image/docs/gallery/bar_charts.html`

We actually use a similar service, the Google Visualization API, for HarvardEnergy, a CS50 App with which you can explore Harvard's energy consumption and greenhouse effects:

`http://energy.cs50.net/`

Select your own dorm or house via the drop-down menus at top-left to see all sorts of interesting data. Here's what else you can do with that particular API:

`http://code.google.com/apis/chart/interactive/docs/gallery.html`

Suffice it to say, by term's end, you'll be empowered to implement ISawYouHarvard and HarvardEnergy alike! For the moment, though, we're confined to a command-line environment. But not to worry, we can still do some pretty neat things. In fact, we can certainly generate bar

[8] Or will we? Okay, we won't.

charts with "ASCII art," even with vertical bars (as opposed to Tej's horizontal ones). Let's give it a try.

Implement, in `chart.c`, a program that prompts a user for four non-negative integers (one for each of **M spotting F**, **F spotting M**, **F spotting F**, and **M spotting M**), after which it should generate a vertical bar chart depicting those values, with the first value's bar on the left and the fourth value's bar on the right. You may assume that the user's terminal window is at least 80 characters wide by 24 characters tall. (You might want to open a terminal window of your own, separate from gedit, as via **Menu > Programming > Terminal**, so that you can see more output at once.) Each bar should be represented as a vertical sequence of 0 or more pound signs (`#`), up to a maximum of 20. The length of each bar should be proportional to the corresponding value and relative to the four values' sum. For instance, if the user inputs 10, 0, 0, and 0, the leftmost bar should be 20 pound signs in height, since 10 is 100% of 10 + 0 + 0 + 0 = 10 and 100% of 20 is 20, and the remaining three bars should be 0 pound signs in length. By contrast, if the user inputs 5, 5, 0, and 0, each of the left two bars should be 10 pound signs in height, since 5 is 50% of 5 + 5 + 0 + 0 = 10 and 50% of 20 is 10, and the rightmost two bars should be 0 pound signs in height. Accordingly, if the user inputs 2, 2, 2, 2, each of the four bars should be 5 pound signs in length, since 2 is 25% of 2 + 2 + 2 + 2 = 8 and 25% of 20 is 5. And so forth. When computing proportions, go ahead and round down to the nearest `int` (as by simply casting any floating-point values to `int`'s). You needn't worry about overflow; you can assume users' inputs will be reasonably small.

Rather than label each bar on the left as Google does for horizontal bars, place abbreviated labels (**M4F**, **F4M**, **F4F**, and **M4M**) immediately below corresponding bar; each bar should be 3 pound signs in width, with two white spaces separating each bar; and the leftmost bar should be flush with the terminal window's lefthand side. Consider the sample output below; assume that the boldfaced text is what some user has typed.

```
jharvard@appliance (~/hacker1): ./chart

M spotting F: 3
F spotting M: 4
F spotting F: 1
M spotting M: 2

Who is Spotting Whom

     ###
     ###
###  ###
###  ###
###  ###       ###
###  ###       ###
###  ###  ###  ###
###  ###  ###  ###
M4F  F4M  F4F  M4M
```

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

```
~cs50/hacker1/chart
```

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.[9]

☐ First, head to

```
https://www.cs50.net/me/
```

to find out which username and password you should use to submit in the following step.

☐ Next, open a terminal window (as via **Menu > Programming > Terminal** or within gedit) then execute:

```
cd ~/hacker1
```

Then execute:

```
ls
```

At a minimum, you should see `pennies.c`, `credit.c`, and `chart.c`. If not, odds are you skipped some step(s) earlier! If you do see those files, you are ready to submit your source code to us. Execute:

```
submit50 ~/hacker1
```

When prompted for **Course**, input **cs50**; when prompted for **Repository**, input **hacker1**. When prompted for a username and password, input the values you found at `https://www.cs50.net/me/`. That command will essentially upload your entire `~/hacker1` 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.

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:

```
http://www.cs50.net/psets/1/
```

If not already logged in, you'll be prompted to log into the course's website. Once you have submitted that form (as well as your source code), you are done! This was Problem Set 1.

[9] This one's much shorter than Problem Set 0's!