Problem Set 1: C

due by 7:00pm on Fri 9/17

Per the directions at this document's end, submitting this problem set involves submitting source code on `cloud.cs50.net` 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 as the video of Week 1's supersection (at `http://www.cs50.net/sections/`) and the video of this problem set's walkthrough (at `http://www.cs50.net/psets/`). 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 – 5, 9, and 11 – 17 of *Absolute Beginner's Guide to C*.
- Chapters 1 – 6 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. 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 *Warn*, *Admonish*, or *Disciplinary Probation*, 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 three primary axes.

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)?

Getting Started.

☑ Take CS50.

☐ Recall that, for Problem Set 0, you obtained (if you didn't have one already) an FAS account. With that account can you log on to computers on campus, particularly those in Science Center B14 (the "terminal room"), where office hours are generally held.

It's now time to obtain an account on CS50's own cluster of servers, otherwise known as `cloud.cs50.net`, on which you'll soon be writing and compiling code (perhaps for the very first time!).[1] Within CS50's cloud will you have your own "home directory" (*i.e.*, personal folder) in which you'll store all the code that you write. The websites you develop for Problem Sets 7 and 8 (and possibly your Final Project) will also live in the cloud.

Not sure what all that means? Not to worry, you soon will!

We've taken the liberty of creating an account on `cloud.cs50.net` for you: for simplicity, your username is identical to your FAS username. Because we don't know your FAS password, though, we've generated a "pseudorandom" password for you. To find out what it is, head to

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

and log in with your FAS username and password if prompted. You should see the password we've generated for you, among other things.[2]

☐ Okay, at this point in the story, you should know your CS50 username and password. It's now time to use them! Just like you saw me do in Week 1, you're about to "SSH to `cloud.cs50.net`," which means you're about to log into some server in CS50's cluster via a "protocol" (or, really, a program) called "Secure Shell." Even though CS50 has multiple servers (so that we can handle lots of users at once), your home directory is "mounted" (*i.e.*, accessible) on all of them. SSH allows you to access the contents of your home directory as well as run programs via a "command-line interface" (CLI) in a "terminal window," which might feel pretty retro at first, inasmuch as you can only command the computer to do your bidding by actually typing commands. But, it turns out, once you get the hang of it, a CLI is a wonderfully efficient interface. We'll teach you plenty of tricks along the way.

Even though CS50's servers happen to run an operating system called Linux, there exist SSH clients (*i.e.*, programs) for Mac OS and Windows alike, which means you can connect to the cloud from

[1] "Cloud" is a buzzword that's very much in vogue these days. It pretty much means "cluster of servers managed by someone else that lives somewhere on the Internet." The word is new. The idea isn't. :-) Just sounds sexier than "cluster of servers managed by someone else that lives somewhere on the Internet."

[2] If you don't see any such password, you're probably just not in our database yet. That may be because you're not yet officially registered for the course, because we haven't imported your answers to Problem Set 0's questions yet, because there was a typo among your answers to Problem Set 0 (the result of which is that a program we wrote overlooked your submission), or because we've made some mistake! Simply email `help@cs50.net` to inquire; be sure to mention your full name, your `@fas.harvard.edu` address, and, if you have one, your `@college.harvard.edu` address so that we can investigate.

any computer (with an Internet connection). Macs tend to come with an SSH client, whereas Windows users must download one (for free). Whatever your operating system, head to

```
http://www.cs50.net/resources/
```

where you'll find all sorts of resources, among them some HOWTOs (*i.e.*, tutorials) for SSH. Mac users should read the article on **Terminal**; Windows users should read the article on **PuTTY**. [3,4]

Alright, ready to give SSH a try? Go ahead and SSH to `cloud.cs50.net` (using Terminal or PuTTY). After providing your username and password, you should reach the "command line" whose "prompt," where you'll soon be typing commands, looks like

```
username@cloud (~):
```

where `username` is your CS50 (and FAS) username; the tilde (~) means that you're currently inside your home directory. That prompt (or, really, the program that runs by default when you SSH to a server) is called your "shell." A shell is just an interpreter of commands. You type something, it does something. For instance, go ahead and type

```
echo hi
```

followed by Enter. Nice! The cloud just said hi back. It turns out that `echo` is a program (that someone else wrote long ago) that takes a "command-line argument" and simply echoes it back. You'll soon be using and writing much more interesting programs!

☐ Let's now create a directory for Problem Set 1 where your code will soon live. Go ahead and "execute" the command below (by typing it and then hitting Enter):[5]

```
mkdir ~/pset1
```

Recall from lecture that ~ denotes your home directory, so the above means "make a directory called `pset1` inside my home directory."

Now open that new directory (or, put another way, change your "current working directory" to that new directory) by executing this command:

```
cd ~/pset1
```

[3] Harvard actually has a "site license" for SecureCRT, which is technically commercial software that Harvard pays for on your (and everyone else's) behalf. But we'll generally recommend PuTTY for Windows users this semester, if only because it's popular and free. But you're welcome to use any SSH client.
[4] If you happen to be running some OS other than Mac OS and Windows on your own computer, odds are you already know how to SSH! But, if not, simply drop `help@cs50.net` a note for advice.
[5] Don't omit the space between `mkdir` and the tilde.

Notice how your prompt just changed to

```
username@cloud (~/pset1):
```

since you're now inside your `pset1` directory. It's worth mentioning that you could have just executed

```
cd pset1
```

a moment ago, because `cd` would simply assume that `pset1` must be inside whichever directory you're currently in. Want to give it a try? Go ahead and just execute

```
cd
```

without any arguments, as they say. You should find yourself back in your home directory, as per your prompt's parenthetical:

```
username@cloud (~):
```

In fact, if you ever get lost inside your account, just execute `cd` without any arguments, and you'll be whisked back to home, sweet home.[6] In any case, now execute

```
cd pset1
```

without any tilde. You should find yourself back inside `~/pset1`.

Make sense? All you're doing, really, is opening and closing folders, albeit via a command-line interface (as opposed to a GUI).

Incidentally, that tilde is really just shorthand notation for your home directory's actual "path" (*i.e.*, location on the server's hard drive). Go ahead and execute

```
pwd
```

which stands for "print working directory." You should see output like:

```
/home/username/pset1
```

On a Linux server, `/` denotes the "root" of the hard drive. And so the implication here is that there's a directory called `home` in the root of the server's hard drive, inside of which is your home directory (and everybody else's), inside of which is that `pset1` directory that you just created. Okay, let's get you home. You could just execute `cd`, but let's take another route home this time. Go ahead and execute:

```
cd ..
```

[6] Too corny? I worried as much.

Recall from lecture that a single dot (`.`) represents your current working directory. It turns out that "dot dot" (`..`) represents your current working directory's "parent directory" (*i.e.*, the directory that it's inside of). And so, if you're in `~/pset1`, changing your current working directory to `..` takes you back to `~`, your home directory. In other words, if you now execute

```
pwd
```

you should find that you're indeed back home (in `/home/username`).[7] Make sense?

Okay, how about "dot dot dot" (`...`)? What does it probably represent?[8]

Alright, make sure you're indeed in your home directory. Let's now take a quick look around. Go ahead and execute this command:[9]

```
ls
```

That command will list the contents of (*i.e.*, the files and directories in) your current working directory. Because the only thing you've created so far is `pset1`, you should only see `pset1`. However, go ahead and re-execute `ls`, this time with a "switch" (a command-line argument that influences a program's behavior):

```
ls -a
```

The `-a` implies "all," and so this invocation of `ls` lists absolutely everything in your home directory, including "dotfiles," files and directories that are normally hidden by default. Never mind those dotfiles for now, but know that they exist; they're generally configuration files.

Incidentally, if you'd like to change that default password of yours, go ahead and execute:[10]

```
passwd
```

When prompted for your `login(LDAP)` password, provide your current (*i.e.*, default) password for the cloud. You'll then be prompted for your new password twice; be sure to type the same thing both times.[11]

Okay, need a short break? Go ahead and execute the below:

```
logout
```

Alternatively, you can hit ctrl-D or execute `exit` instead. You've just disconnected from the cloud. Always best to log out in this manner before you quit Terminal or PuTTY or put your computer to sleep.

---

[7] If not, just execute `cd` by itself and take that shortcut home!
[8] Nope, nothing. :-)
[9] Note that the `l` is a lowercase L.
[10] Indeed, there's no `o` or `r` in `passwd`. Why? Eh, slightly shorter to type is all.
[11] If you forget your new password, email `help@cs50.net`, and we'll reset it to your default.

O hai, Nano!

☐ Back so soon?? Okay, go ahead and SSH to the cloud, if you aren't still there. (Remember how?) Then navigate your way to `~/pset1`. (Remember how?) Once there, execute the command below:

```
nano hello.c
```

Recall that `nano` is a (relatively) simple text editor with which you can write code (or any text, for that matter). Proceed to write your own version of "hello, world." It suffices to re-type, nearly character for character, Week 1's `hai1.c`, but do at least replace "O hai, world!" with your own argument to `printf`.

Once done with your recreation, hit ctrl-x to save, followed by Enter, and you should be returned to your prompt. Proceed to execute the command below.

```
gcc hello.c
```

If you've made no mistakes, you should just see another prompt. If you've made some mistake, you'll instead see one or more warning and/or error messages. Even if cryptic, think about what they might mean, then go find your mistake(s)! To edit `hello.c`, re-execute `nano` as before. Once your code is correct and compiles successfully, look for your program in your current working directory by typing the following command.

```
ls
```

You should see output resembling the below.

```
a.out  hello.c
```

Actually, some more details would be nice. Go ahead and execute the command below instead.

```
ls -l
```

More than just list the contents of your current working directory, this command lists their sizes, dates and times of creation, and more. The output you see should resemble the below.[12]

```
-rwx------ 1 username student 4647 2010-09-10 19:01 a.out
-rw------- 1 username student   66 2010-09-10 19:01 hello.c
```

It turns out that `-l` is another switch that controls the behavior of `ls`. To look up more switches for `ls` in the cloud's user manual, execute the command below.

```
man ls
```

[12] Don't worry if the numbers you see don't match ours.

You can scroll up and down in this manual using your keyboard's arrow keys and space bar. In general, anytime you'd like more information about some command, try checking its "man page" by executing `man` followed by the command's name! Let's now confirm that your program does work. Execute the command below.

```
./a.out
```

You should see your greeting. Recall that `.` denotes your current working directory, and so this command means "execute the program called `a.out` in my current working directory."

Before moving on, let's give your program a more interesting name than `a.out`. Go ahead and execute the following command.[13]

```
gcc -o hello hello.c
```

In this case, `-o` is but a switch for `gcc`. The effect of this switch is to name `gcc`'s output `hello` instead of `a.out`. Let's now get rid of your first compilation. To delete `a.out`, execute the following command.

```
rm a.out
```

If prompted to confirm, hit y followed by Enter.

Welcome to Linux and C!

Story Time.

☐ We explored in Week 1 how hard drives (and floppies) 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.

[13] Be careful not to transpose `hello` and `hello.c`, else you'll end up deleting your code!

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

`http://wiki.cs50.net/Style_Guide`

Inasmuch as style is, to some extent, a matter of personal preference, CS50 doesn't mandate that you mimic the styles that you see in lecture and section. But we do expect that you model your own style after common conventions. You'll find that CS50's Style Guide introduces you to some of those common conventions. Do keep them in mind as you start churning out code!

Free Candy.

☐ So, one of the best things about Maxwell Dworkin (the CS building) is the free-candy machine in the Lounge in room G123:[14,15]

Well, that and the cat in the ceiling. Do drop by sometime! Anyhow, there's a whole lot of Skittles, Mike and Ike's, and M&M's in that machine.[16] Want to guess how many Skittles?

Glad you said yes! Implement, in a file called `skittles.c`, a program that first picks a (pseudorandom) number between 0 and 1023, inclusive, and then asks you (the human) to guess what it is.[17] The program should keep asking you to guess until you guess right, at which point it should thank you for playing.

Where to begin? Allow us to hand you some puzzle pieces.

[14] It might have been us who hacked it to be free.
[15] Photograph by Dan Armendariz.
[16] Actually, there's probably fewer now that we've mentioned the machine's existence.
[17] To be clear, that range includes 1024 integers, from and including 0, to and including 1023.

To generate a random number, you can use a function called `rand`. Take a peek at its man page by executing the command below:[18]

```
man 3 rand
```

The `3` instructs `man` to consult section 3 of the server's user manual: whereas programs are generally documented in section 1 (the default if you specify no number at all), functions are often documented in section 3. Per the man page for `rand`, under **SYNOPSIS**, the function is declared in `stdlib.h`. So you'll want to put

```
#include <stdlib.h>
```

atop `skittles.c` along with

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

as usual. The order of such includes tends not to matter, but alphabetical is probably good style.

Also notice that `rand` "returns a value between 0 and `RAND_MAX`." It turns out that `RAND_MAX` is a "constant" (a symbol that represents some value) that's defined in `stdlib.h`. Its value can vary by server, and so it's not hard-coded into the manual. Let's assume that `RAND_MAX` is greater than 1023. How, though, do we map a number that's between 0 and `RAND_MAX` to a number that's between 0 and 1023?

Turns out that modulo operator (`%`) we saw in Week 1 is useful for more than arithmetic remainders alone! Consider this line of code:

```
int skittles = rand() % 1024;
```

The effect of that line is to divide the "return value" of `rand` by 1024 and store the remainder in `skittles`. What might the remainder be, though, when dividing some integer by 1024? Well, there might be no remainder, in which case the answer is 0. Or there might be a big remainder, in which case the answer is 1023. And, of course, a whole bunch of other remainders are possible in between those bounds.

Well there you have it, a way of generating a pseudorandom number between 0 and 1023, inclusive! There's a catch, though. It turns out that, by default, `rand` always returns the same number (odds are it's `1804289383`) the first time it's called in a program, in which case your program's always going to be filled with the same number of Skittles. That's because, per the man page for `rand`, "If no seed value is provided, the `rand()` function is automatically seeded with a value of 1."

A "seed" is simply a value that influences the sequence of values returned by a "pseudorandom number generator" (PRNG) like `rand`. To be clear, it's not the first number returned by a PRNG but, rather, an influence thereon. (See why we say "pseudorandom" all the time instead of

[18] Recall that, to quit `man`, you can hit q.

"random"? Computers can't really generate numbers that are truly random: they have to start somewhere.) How can you override this default seed of 1? Before you call `rand`, call `srand` with your choice of seed (*e.g.*, 2):

```
srand(2);
```

Better yet, call `srand` with a seed that actually changes over time (literally), without your having to recompile your code each time you want it to change:

```
srand(time(NULL));
```

Never mind what `NULL` is for now, but know that

```
time(NULL)
```

returns the current time in seconds; that's not a bad seed (unlike some people). No need to store the return value of `time` in some variable first; we can pass it directly to `srand` between those parentheses. It's worth noting, though, that time is declared in `time.h`, so you'll need to include that header file too.

Alright, what other puzzle pieces do we need? Well, your program will need to tell the user what to do, for which `printf` should be helpful. And you'll want to allow the user an infinite number of guesses, for which some looping construct is probably necessary. And you'll also want to get integers from the user, for which `GetInt`, declared in `cs50.h`, is definitely handy.

Okay, where to begin? Allow us to suggest that you begin by filling `skittles.c` with this code:

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

int
main(void)
{
    // seed PRNG
    srand(time(NULL));

    // pick pseudorandom number in [0, 1023]
    int skittles = rand() % 1024;

    // TODO
}
```

We'll leave the `TODO` to you! Remember, don't try to implement the whole program at once. Perhaps start by printing (with `printf`) the value of `skittles`, just to be sure that you didn't make any typos. Then save your code and quit `nano` (with ctrl-x) and proceed to compile it with:[19]

```
make skittles
```

Recall that this command saves you the trouble of executing `gcc` directly with its `-o` flag. To run your program (assuming it compiles with no errors), execute

```
./skittles
```

to see how many Skittles there are. Wait one second and then run your program again: odds are the number of Skittles will differ. Then re-open `skittles.c` with `nano` and take another bite out of this problem. Perhaps next implement your program's instructions that explain to the user how to play this guessing game.

What should your program's output be, once fully implemented? We leave your program's personality entirely up to you, but below's one possible design. Assume that the boldfaced text is what some user has typed.

```
username@cloud (~/pset1): ./skittles
O hai! I'm thinking of a number between 0 and 1023. What is it?
0
Nope! There are way more Skittles than that. Guess again.
1
Nope! There are way more Skittles than that. Guess again.
-1
Nope! Don't be difficult. Guess again.
1023
Nope! There are fewer Skittles than that. Guess again.
42
That's right! Nom nom nom.
```

Your program should end once the user has guessed right. The above design happens to respond to the user's input in a few different ways, but we leave it to you to decide how much to vary your program's output.

Incidentally, know that you can generally force a program to quit prematurely by hitting ctrl-c. For efficiency's sake, you might find it helpful to SSH to the cloud twice in two separate windows, so that you can keep `nano` open in one and use the other to compile and test things. And recall from Week 0 that finding a value between 0 and 1023 doesn't actually require that many guesses. Odds are you can test your program fairly efficiently. You can certainly use temporarily a modulus less than 1024 to save even more time; just be sure that your final version does pick a number in [0, 1023].

[19] If only it were that easy to make Skittles.

If you'd like to play with the staff's own implementation of `skittles` on `cloud.cs50.net`, you may execute the below.

```
~cs50/pub/solutions/pset1/skittles
```

Time for Change.

☐ "Counting out change is a blast (even though it boosts mathematical skills) with this spring-loaded changer that you wear on your belt to dispense quarters, dimes, nickels, and pennies into your hand." Or so says the website on which we found this here fashion accessory.[20]

Of course, the novelty of this thing quickly wears off, especially when some jerk wants to pay for his newspaper with a hundred-dollar bill. Fortunately, computer science has given cashiers everywhere ways to minimize numbers of coins due: greedy algorithms.

According to the National Institute of Standards and Technology (NIST), a greedy algorithm is one "that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems."[21]

What's all that mean? Well, suppose that a cashier owes a customer some change and on that cashier's belt are levers that dispense quarters, dimes, nickels, and pennies. Solving this "problem" requires one or more presses of one or more levers. Think of a "greedy" cashier as one who wants to take, with each press, the biggest bite out of this problem as possible. For instance, if some customer is owed 41¢, the biggest first (*i.e.*, best immediate, or local) bite that can be

[20] Description and image from `hearthsong.com`. For ages 5 and up.
[21] `http://www.nist.gov/dads/HTML/greedyalgo.html`

taken is 25¢. (That bite is "best" inasmuch as it gets us closer to 0¢ faster than any other coin would.) Note that a bite of this size would whittle what was a 41¢ problem down to a 16¢ problem, since 41 – 25 = 16. That is, the remainder is a similar but smaller problem. Needless to say, another 25¢ bite would be too big (assuming the cashier prefers not to lose money), and so our greedy cashier would move on to a bite of size 10¢, leaving him or her with a 6¢ problem. At that point, greed calls for one 5¢ bite followed by one 1¢ bite, at which point the problem is solved. The customer receives one quarter, one dime, one nickel, and one penny: four coins in total.

It turns out that this greedy approach (*i.e.*, algorithm) is not only locally optimal but also globally so for America's currency (and also the European Union's). That is, so long as a cashier has enough of each coin, this largest-to-smallest approach will yield the fewest coins possible.[22]

How few? Well, you tell us. Write, in `greedy.c`, a program that first asks the user how much change is owed and then spits out (via `printf`) the minimum number of coins with which said change can be made. Use `GetFloat` from CS 50's library to get the user's input and `printf` from the Standard I/O library to output your answer.

We ask that you use `GetFloat` so that you can handle dollars and cents, albeit sans dollar sign. In other words, if some customer is owed $9.75 (as in the case where a newspaper costs 25¢ but the customer pays with a $10 bill), assume that your program's input will be **`9.75`** and not **`$9.75`** or **`975`**. However, if some customer is owed $9 exactly, assume that your program's input will be **`9.00`** or just **`9`** but, again, not **`$9`** or **`900`**. Of course, by nature of floating-point values, your program will likely work with inputs like **`9.0`** and **`9.000`** as well; you need not worry about checking whether the user's input is "formatted" like money should be. And you need not try to check whether a user's input is too large to fit in a `float`. But you should check that the user's input makes cents! Er, sense. Using `GetFloat` alone will ensure that the user's input is indeed a floating-point (or integral) value but not that it is non-negative. If the user fails to provide a non-negative value, your program should re-prompt the user for a valid amount again and again until the user complies. Incidentally, do beware the inherent imprecision of floating-point values.[23] Before doing any math, you'll probably want to convert the user's input entirely to cents (*i.e.*, from a `float` to an `int`) to avoid tiny errors that might otherwise add up![24] Be careful to round and not truncate your pennies!

[22] By contrast, suppose that a cashier runs out of nickels but still owes some customer 41¢. How many coins does that cashier, if greedy, dispense? How about a "globally optimal" cashier?

[23] For instance, `0.01` cannot be represented exactly as a `float`. Try printing its value to, say, ten decimal places with code like the below:

```
float f = 0.01;
printf("%.10f\n", f);
```

[24] Don't just cast the user's input from a `float` to an `int`! After all, how many cents does one dollar equal?

So that we can automate some tests of your code, we ask that your program's last line of output be only the minimum number of coins possible: an integer followed by `\n`. Consider the below representative of how your own program should behave; highlighted in bold is some user's input.

```
username@cloud (~/pset1): ./greedy
O hai!  How much change is owed?
0.41
4
```

By nature of floating-point values, that user could also have inputted just **`.41`**. (Were they to input **`41`**, though, they'd get many more coins!)

Of course, more difficult users (let's call them, oh, n00bs) might experience something more like the below.

```
username@nice (~/pset1): ./greedy
O hai!  How much change is owed?
-0.41
Er, how much change is owed?
-0.41
Er, how much change is owed?
foo
Retry: 0.41
4
```

Per these requirements (and the sample above), your code will likely have some sort of loop. If, while testing your program, you find yourself looping forever, remember that you can kill your program (*i.e.*, short-circuit its execution) by hitting ctrl-c (sometimes a lot).

We leave it to you to determine how to compile and run and debug this particular program!

If you'd like to play with the staff's own implementation of `greedy` on `cloud.cs50.net`, you may execute the below.

```
~cs50/pub/solutions/pset1/greedy
```

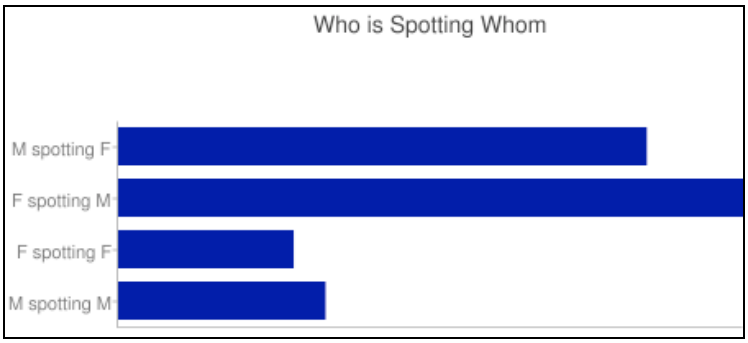
I Saw You.

☐ Surf on over to

`http://isawyouharvard.com/`

where you'll find a website created by CS50's own Tej Toor '10 as her Final Project last year, "your source for posting and browsing missed connections." 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.[25]

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/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/visualization/documentation/gallery.html`

Suffice it to say, by term's end, you'll be empowered to implement ISawYouHarvard and HarvardEnergy alike!

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

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 charts with "ASCII art". Let's do it.

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 horizontal bar chart depicting those values, with the first value's bar on top and the fourth value's bar on the bottom. Assume that the user's terminal window is at least 80 characters wide by 24 characters tall. (Best to ensure that your own window boasts at least those dimensions, as by dragging the window's bottom-right corner as needed.) Each bar should be represented as a horizontal sequence of 0 or more pound signs (`#`), up to a maximum of 80. 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 first bar should be 80 pound signs in length, since 10 is 100% of 10 + 0 + 0 + 0 = 10 and 100% of 80 is 80, 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 top two bars should be 40 pound signs in length, since 5 is 50% of 5 + 5 + 0 + 0 = 10 and 50% of 80 is 40, and the bottom two bars should be 0 pound signs in length. Accordingly, if the user inputs 2, 2, 2, 2, each of the four bars should be 20 pound signs in length, since 2 is 25% of 2 + 2 + 2 + 2 = 8 and 25% of 80 is 20. And so forth. When computing proportions, go ahead and round <u>down</u> to the nearest `int` (as by simply casting any floating-point values to `int`'s).

Rather than label each bar on the left as Google does, place each label immediately above the corresponding bar; you're welcome to output some blank lines for clarity so long as the last 8 lines of your output are the bars and their labels. Unlike Week 2's progress bar, you chart need not be animated. Consider the sample output below; assume that the boldfaced text is what some user has typed.

```
username@cloud (~/pset1): ./chart

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

Who is Spotting Whom

M spotting F
#######################
F spotting M
##############################
F spotting F
########
M spotting M
################
```

If you'd like to play with the staff's own implementation of `chart` on `cloud.cs50.net`, you may execute the below.

```
~cs50/pub/solutions/pset1/chart
```

How to Submit.

In order to submit this problem set, you must first execute a command on `cloud.cs50.net` and then submit a (brief) form online.[26]

☐ SSH to `cloud.cs50.net`, if not already there, and then execute:

```
cd ~/pset1
```

If informed that there is "no such file or directory," odds are you skipped some step(s) earlier! Not a problem, we'll help you fix in a moment. If the command indeed works, proceed to execute this command next:

```
ls
```

At a minimum, you should see `skittles.c`, `greedy.c`, and `chart.c`. If not, odds are you skipped some more steps earlier! If you do see those files, you're ready to submit your source code to us:[27]

```
~cs50/pub/bin/submit pset1
```

That command will essentially copy your entire `~/pset1` directory into CS50's own account, where your TF will be able to access it. You'll know that the command worked if you are informed that your "work HAS been submitted." If you instead encounter an error that doesn't appear to be a mistake on your part, do try running the command one or more additional times. (The `submit` program's a bit old and annoyingly flakey.) But if informed that your "work HAS been submitted," it indeed has.

Now, if you instead see some message informing you that your code is not where it should be, you'll need to fix that before you submit. Recall that you can list the contents of your current working directory simply by executing `ls`. Recall that you can create a directory (*e.g.*, `pset1`) with `mkdir`. Recall that you can move to and from directories with `cd`. And know that you can rename or move files with `mv`. For instance,

```
mv foo bar
```

will rename a directory (or file) called `foo` in your current working directory to `bar`, which is handy if you accidentally named `bar` `foo`! Meanwhile,

```
mv foo.c pset1
```

will move `foo.c` into a directory called `pset1` in your current working directory, assuming `pset1` already exists (else `foo.c` will be renamed `pset1`)! In other words, `mv` moves things if the

[26] This one's much shorter than Problem Set 0's!
[27] Note that there is no slash between this tilde (~) and `cs50`. This particular path implies that a program called `submit` lives in a directory called `bin`, which is in a directory called `pub`, which is in CS50's own home directory (the shorthand notation for which is `~cs50`).

destination (*i.e.*, the second command-line argument) already exists, else it renames things. Just so you know,

```
rm foo.c
```

will remove (*i.e.*, delete) `foo.c`. Do be careful if you use that command! No need to delete any files, though, before submitting, even if you have more than just three files in `pset1`.

If these tips don't help you solve some problem you're having, not to worry. Email `help@cs50.net` for assistance, but be sure to tell us your username, and be sure to detail the problem you're having. Take care to seek help if you need it well before the problem set's deadline, lest you spend a late day unnecessarily; 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.