This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 0

What to Do

1. Download and install the latest version of Chrome (https://www.google.com/chrome/), if you don't have it already.
2. Implement your very own Scratch project using Chrome, per this specification.

When to Do It

By 11:59pm on 31 December 2020.

Advice

Here are David's examples (https://scratch.mit.edu/studios/25128634/) from lecture if you'd like to review! To see the source code of each, click **See inside**.

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Scratch

It's time to choose your own adventure! Your assignment, quite simply, is to implement in Scratch any project of your choice, be it an interactive story, game, animation, or anything else, subject only to the following requirements:

- Your project must have at least two sprites, at least one of which must resemble something other than a cat.
- Your project must have at least three scripts total (i.e., not necessarily three per sprite).
- Your project must use at least one condition, one loop, and one variable.
- Your project must use at least one sound.
- Your project should be more complex than most of those demonstrated in lecture (many of which, though instructive, were quite short) but it can be less complex than *Ivy's Hardest Game*. As such, your project should probably use a few dozen puzzle pieces overall.

If you'd like to try out some Scratch projects from past students, here are a few:

- Ivy's Hardest Game (https://scratch.mit.edu/projects/326129433/), from lecture
- Soccer (https://scratch.mit.edu/projects/37413/), a game
- Cookie Love Story (https://scratch.mit.edu/projects/26329196/), an animation
- Gingerbread tales (https://scratch.mit.edu/projects/277536784/), an interactive story
- Intersection (https://scratch.mit.edu/projects/75390754/), a game

You might find these tutorials (https://scratch.mit.edu/projects/editor/?tutorial=all) or starter projects (https://scratch.mit.edu/starter-projects) helpful. And you're welcome to explore scratch.mit.edu (https://scratch.mit.edu/explore/projects/all) for inspiration. But try to think of an idea on your own, and then set out to implement it. However, don't try to implement the entirety of your project all at once: pluck off one piece at a time. In other words, take baby steps: write a bit of code (i.e., drag and drop a few puzzle pieces), test, write a bit more, test, and so forth. And select **File > Save now** every few minutes so that you don't lose any work!

If, along the way, you find it too difficult to implement some feature, try not to fret; alter your design or work around the problem. If you set out to implement an idea that you find fun, odds are you won't find it too hard to satisfy the above requirements.

Alright, off you go. Make us proud!

Once finished with your project, select **File > Save now** one last time. Then select **File > Save to your computer** and keep that file so that you can submit it.

Hello, World

Suffice it to say it's a bit harder to meet classmates when taking a course online. But, thanks to technology, everyone can at least say hello!

If you have a phone (or digital camera) and would like to say hello to classmates, record a 1- to 2-minute video of yourself saying hello, perhaps stating where in the world you are, why you're taking CS50x, and something interesting about you! Try to begin your video by saying "hello, world" and end it with "my name is ...., and this is CS50." But, ultimately, it's totally up to you.

If you do record a video, upload it to YouTube (unless blocked in your country, in which case you're welcome to upload it elsewhere) so that you can provide us with its URL when you submit!

How to Submit

Step 1 of 2

Submit this form (https://forms.cs50.io/bb5ace07-099c-405e-8da5-33ce6e242601).

Step 2 of 2

This step assumes that you've downloaded your Scratch project as a file whose name ends in `.sb3`. And this step also assumes that you've signed up for a GitHub account (https://github.com/join), per the above form.

1. Visit this link (https://submit.cs50.io/invites/9770b67479384c4d8c37790779e466d9), log in with your GitHub account, and click **Authorize cs50**.
2. Check the box indicating that you'd like to grant course staff access to your submissions, and click **Join course**.
3. Go to https://submit.cs50.io/upload/cs50/problems/2020/x/scratch (https://submit.cs50.io/upload/cs50/problems/2020/x/scratch).
4. Click "Choose File" and choose your `.sb3` file. Click **Submit**.

That's it! Once your submission uploads, you should be redirected to your submission page. Click the submission link and then the **check50** link to see which requirements your project met. You are welcome to resubmit as many times as you'd like (before the deadline)!

To view your current progress in the course, visit the course gradebook at cs50.me/cs50x (https://cs50.me/cs50x)!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 1

For this problem set, you'll use CS50 IDE, a cloud-based programming environment. This environment is similar to CS50 Sandbox and CS50 Lab, the programming environments that David discussed during lecture.

What to Do

1. Go to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE.
2. Submit Hello
3. Submit one of:
   - this version of Mario if feeling less comfortable
   - this version of Mario if feeling more comfortable
4. Submit one of:
   - Cash if feeling less comfortable
   - Credit if feeling more comfortable

If you submit both Marios, we'll record the higher of your two scores. If you submit both of Cash and Credit, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 1's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `hello`, and

  ```
  make hello
  ```

  is yielding errors, try running

  ```
  help50 make hello
  ```

  instead!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Hello

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/hello/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

Getting Started

CS50 IDE is a web-based "integrated development environment" that allows you to program "in the cloud," without installing any software locally. Indeed, CS50 IDE provides you with your very own "workspace" (i.e., storage space) in which you can save your own files and folders (aka directories).

Logging In

Head to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE. Once your IDE loads, you should see that (by default) it's divided into three parts. Toward the top of CS50 IDE is your "text editor", where you'll write all of your programs. Toward the bottom of is a "terminal window" (light blue, by default), a command-line interface (CLI) that allows you to explore your workspace's files and directories, compile code, run programs, and even install new software. And on the left is your "file browser", which shows you all of the files and folders currently in your IDE.

Start by clicking inside your terminal window. You should find that its "prompt" resembles the below.

```
~/ $
```

Click inside of that terminal window and then type

```
mkdir ~/pset1/
```

followed by Enter in order to make a directory (i.e., folder) called `pset1` in your home directory. Take care not to overlook the space between `mkdir` and `~/pset1` or any other character for that matter! Keep in mind that `~` denotes your home directory and `~/pset1` denotes a directory called `pset1` within `~`.

Here on out, to execute (i.e., run) a command means to type it into a terminal window and then hit Enter. Commands are "case-sensitive," so be sure not to type in uppercase when you mean lowercase or vice versa.

Now execute

```
cd ~/pset1/
```

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

```
~/pset1/ $
```

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

Now execute

```
mkdir ~/pset1/hello
```

to create a new directory called `hello` inside of your `pset1` directory. Then execute

```
cd ~/pset1/hello
```

to move yourself into that directory.

Shall we have you write your first program? From the *File* menu, click *New File*, and save it (as via the *Save* option in the *File* menu) as `hello.c` inside of your `~/pset1/hello` directory. Proceed to write your first program by typing precisely these lines into the file:

```
#include <stdio.h>

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

Notice how CS50 IDE adds "syntax highlighting" (i.e., color) as you type, though CS50 IDE's choice of colors might differ from this problem set's. Those colors aren't actually saved inside of the file itself; they're just added by CS50 IDE to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, CS50 IDE wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

Listing Files

Next, in your terminal window, immediately to the right of the prompt (`~/pset1/hello/ $`), execute

```
ls
```

You should see just `hello.c` ? That's because you've just listed the files in your `hello` folder. In particular, you *executed* (i.e., ran) a command called `ls` , which is shorthand for "list." (It's such a frequently used command that its authors called it just `ls` to save keystrokes.) Make sense?

Compiling Programs

Now, before we can execute the `hello.c` program, recall that we must *compile* it with a *compiler* (e.g., `clang`), translating it from *source code* into *machine code* (i.e., zeroes and ones). Execute the command below to do just that:

```
clang hello.c
```

And then execute this one again:

```
ls
```

This time, you should see not only `hello.c` but `a.out` listed as well? (You can see the same graphically if you click that folder icon again.) That's because `clang` has translated the source code in `hello.c` into machine code in `a.out` , which happens to stand for "assembler output," but more on that another time.

Now run the program by executing the below.

```
./a.out
```

Hello, world, indeed!

Naming Programs

Now, `a.out` isn't the most user-friendly name for a program. Let's compile `hello.c` again, this time saving the machine code in a file called,

```
submit50 cs50/problems/2020/x/hello
```

more aptly, `hello`. Execute the below.

```
clang -o hello hello.c
```

Take care not to overlook any of those spaces therein! Then execute this one again:

```
ls
```

You should now see not only `hello.c` (and `a.out` from before) but also `hello` listed as well? That's because `-o` is a *command-line argument*, sometimes known as a *flag* or a *switch*, that tells `clang` to output (hence the `o`) a file called `hello`. Execute the below to try out the newly named program.

```
./hello
```

Hello there again!

Making Things Easier

Recall that we can automate the process of executing `clang`, letting `make` figure out how to do so for us, thereby saving us some keystrokes. Execute the below to compile this program one last time.

```
make hello
```

You should see that `make` executes `clang` with even more command-line arguments for you? More on those, too, another time!

Now execute the program itself one last time by executing the below.

```
./hello
```

Phew!

Getting User Input

Suffice it to say, no matter how you compile or execute this program, it only ever prints `hello, world`. Let's personalize it a bit, just as we did in class.

Modify this program in such a way that it first prompts the user for their name and then prints `hello, so-and-so`, where `so-and-so` is their actual name.

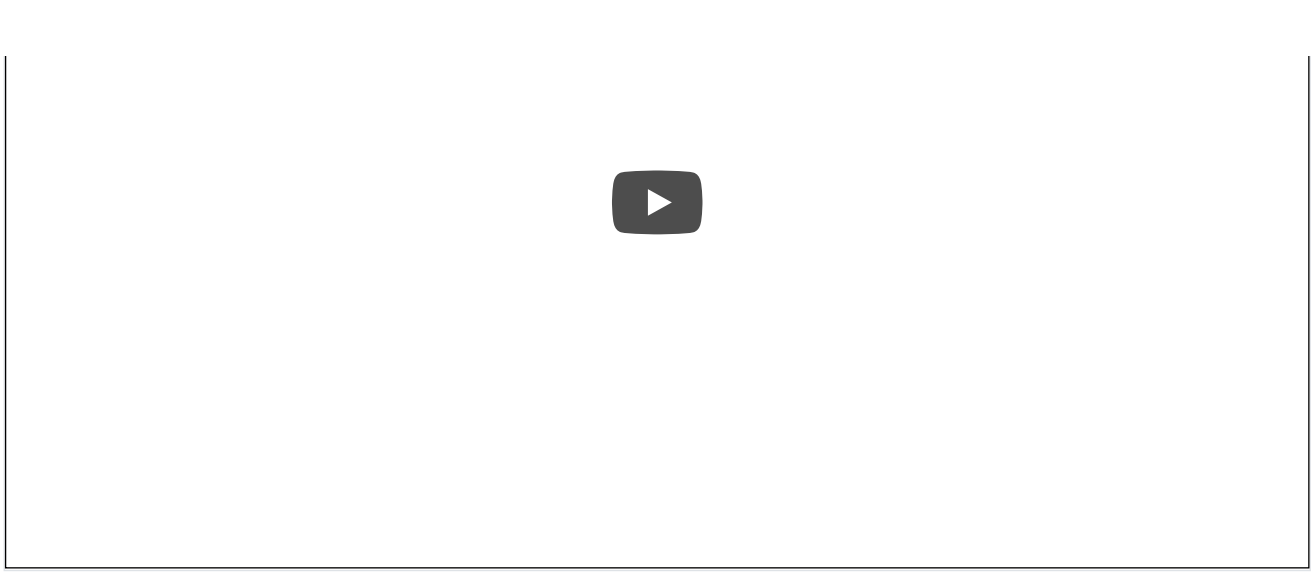
As before, be sure to compile your program with:

```
make hello
```

And be sure to execute your program, testing it a few times with different inputs, with:

```
./hello
```

Walkthrough



Hints

Don't recall how to prompt the user for their name?

Recall that you can use `get_string` as follows, storing its *return value* in a variable called `name` of type `string`.

```
string name = get_string("What is your name?\n");
```

Don't recall how to format a string?

Don't recall how to join (i.e., concatenate) the user's name with a greeting? Recall that you can use `printf` not only to print but to format a string (hence, the `f` in `printf`), a la the below, wherein `name` is a `string`.

```
printf("hello, %s\n", name);
```

Use of undeclared identifier?

Seeing the below, perhaps atop other errors?

```
error: use of undeclared identifier 'string'; did you mean 'stdin'?
```

Recall that, to use `get_string`, you need to include `cs50.h` (in which `get_string` is *declared*) atop a file, as with:

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

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/hello
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 hello.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

This is CS50x

OpenCourseWare

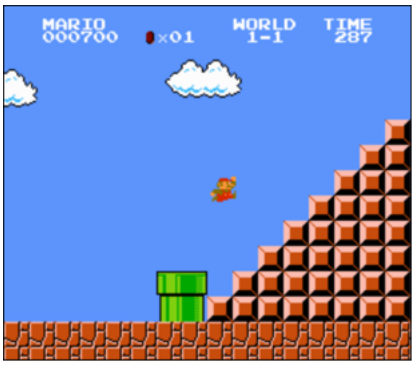
David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/mario/less/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

World 1-1

Toward the end of World 1-1 in Nintendo's Super Mario Brothers, Mario must ascend right-aligned pyramid of blocks, a la the below.



Let's recreate that pyramid in C, albeit in text, using hashes (`#`) for bricks, a la the below. Each hash is a bit taller than it is wide, so the pyramid itself is also be taller than it is wide.

```
      #
     ##
    ###
   ####
  #####
 ######
#######
########
```

The program we'll write will be called `mario` . And let's allow the user to decide just how tall the pyramid should be by first prompting them for a positive integer between, say, 1 and 8, inclusive.

Here's how the program might work if the user inputs `8` when prompted:

```
$ ./mario
Height: 8
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

Here's how the program might work if the user inputs `4` when prompted:

```
$ ./mario
Height: 4
   #
  ##
 ###
####
```

Here's how the program might work if the user inputs `2` when prompted:

```
$ ./mario
Height: 2
 #
##
```

And here's how the program might work if the user inputs `1` when prompted:

```
$ ./mario
Height: 1
#
```

If the user doesn't, in fact, input a positive integer between 1 and 8, inclusive, when prompted, the program should re-prompt the user until they cooperate:

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
   #
  ##
 ###
####
```

How to begin? Let's approach this problem one step at a time.

Pseudocode

First, create a new directory (i.e., folder) called `mario` inside of your `pset1` directory by executing

```
~/ $ mkdir ~/pset1/mario
```

Add a new file called `pseudocode.txt` inside of your `mario` directory.

Write in `pseudocode.txt` some pseudocode that implements this program, even if not (yet!) sure how to write it in code. There's no one right way to write pseudocode, but short English sentences suffice. Recall how we wrote pseudocode for finding Mike Smith (https://docs.google.com/presentation/d/17wRd8ksO6QkUq906SUgm17AqcI-Jan42jkY-EmufxnE/edit?usp=sharing). Odds are your pseudocode will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

▶ **Spoiler**

Prompting for Input

Whatever your pseudocode, let's first write only the C code that prompts (and re-prompts, as needed) the user for input. Create a new file called `mario.c` inside of your `mario` directory.

Now, modify `mario.c` in such a way that it prompts the user for the pyramid's height, storing their input in a variable, re-prompting the user again and again as needed if their input is not a positive integer between 1 and 8, inclusive. Then, simply print the value of that variable, thereby confirming (for yourself) that you've indeed stored the user's input successfully, a la the below.

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
Stored: 4
```

▶ **Hints**

Building the Opposite

Now that your program is (hopefully!) accepting input as prescribed, it's time for another step.

It turns out it's a bit easier to build a left-aligned pyramid than right-, a la the below.

```
#
##
###
####
#####
######
#######
########
```

So let's build a left-aligned pyramid first and then, once that's working, right-align it instead!

Modify `mario.c` at right such that it no longer simply prints the user's input but instead prints a left-aligned pyramid of that height.

▶ **Hints**

Right-Aligning with Dots

Let's now right-align that pyramid by pushing its hashes to the right by prefixing them with dots (i.e., periods), a la the below.

```
.......#
......##
.....###
....####
...#####
  ######
```

```
..######
.#######
########
```

Modify `mario.c` in such a way that it does exactly that!

▶ **Hint**

How to Test Your Code

Does your code work as prescribed when you input

- `-1` (or other negative numbers)?
- `0` ?
- `1` through `8` ?
- `9` or other positive numbers?
- letters or words?
- no input at all, when you only hit Enter?

Removing the Dots

All that remains now is a finishing flourish! Modify `mario.c` in such a way that it prints spaces instead of those dots!

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50` . But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/mario/less
```

Execute the below to evaluate the style of your code using `style50` .

```
style50 mario.c
```

▶ **Hint**

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/mario/less
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/mario/more/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

World 1-1

Toward the beginning of World 1-1 in Nintendo's Super Mario Brothers, Mario must hop over adjacent pyramids of blocks, per the below.

Let's recreate those pyramids in C, albeit in text, using hashes (`#`) for bricks, a la the below. Each hash is a bit taller than it is wide, so the pyramids themselves are also be taller than they are wide.

```
   #  #
  ##  ##
 ###  ###
####  ####
```

The program we'll write will be called `mario` . And let's allow the user to decide just how tall the pyramids should be by first prompting them for a positive integer between, say, 1 and 8, inclusive.

Here's how the program might work if the user inputs `8` when prompted:

```
$ ./mario
Height: 8
       #  #
      ##  ##
     ###  ###
    ####  ####
   #####  #####
  ######  ######
 #######  #######
########  ########
```

Here's how the program might work if the user inputs `4` when prompted:

```
$ ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
```

```
   #  #
  ##  ##
 ###  ###
####  ####
```

Here's how the program might work if the user inputs `2` when prompted:

```
$ ./mario
Height: 2
 #  #
##  ##
```

And here's how the program might work if the user inputs `1` when prompted:

```
$ ./mario
Height: 1
#  #
```

If the user doesn't, in fact, input a positive integer between 1 and 8, inclusive, when prompted, the program should re-prompt the user until they cooperate:

```
$ ./mario
Height: -1
Height: 0
Height: 42
Height: 50
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

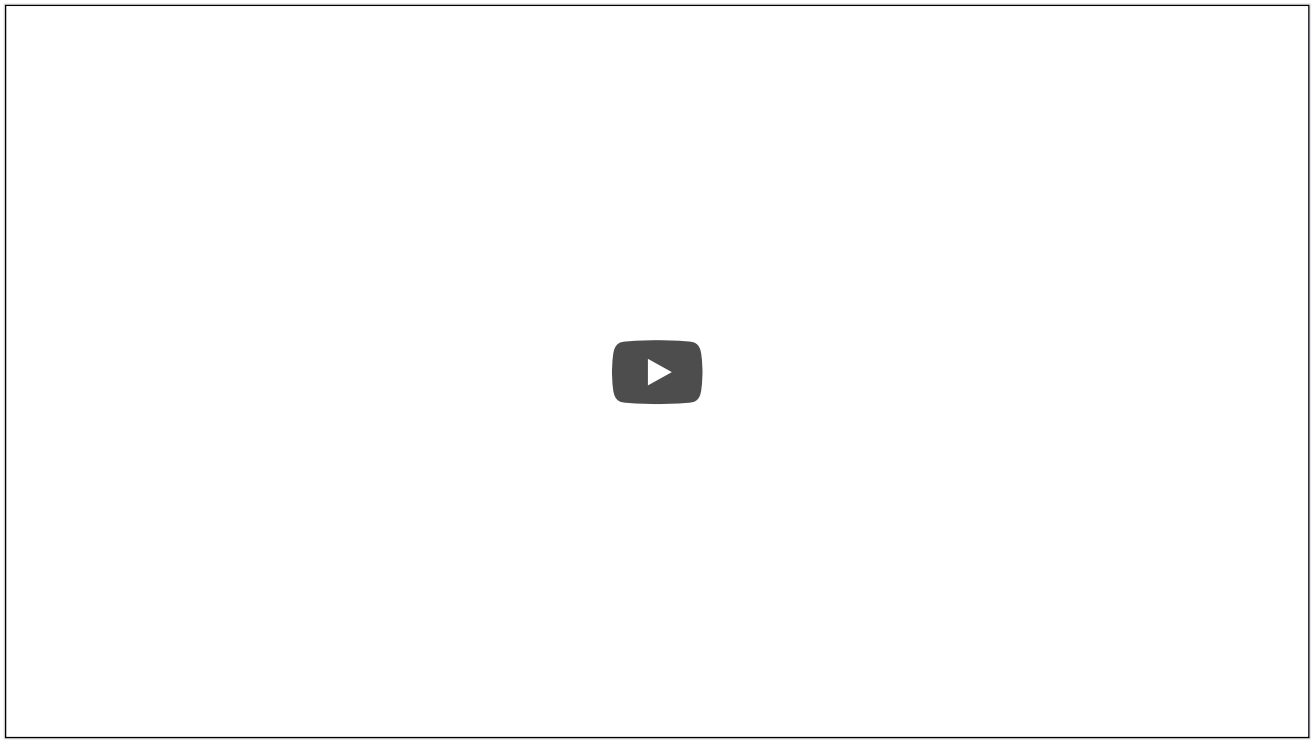
Notice that width of the "gap" between adjacent pyramids is equal to the width of two hashes, irrespective of the pyramids' heights.

Create a new directory called `mario` inside of your `pset1` directory by executing

```
~/ $ mkdir ~/pset1/mario
```

Create a new file called `mario.c` inside your `mario` directory. Modify `mario.c` in such a way that it implements this program as described!

Walkthrough



How to Test Your Code

Does your code work as prescribed when you input

- `-1` (or other negative numbers)?
- `0` ?
- `1` through `8` ?
- `9` or other positive numbers?
- letters or words?
- no input at all, when you only hit Enter?

You can also execute the below to evaluate the correctness of your code using `check50` . But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/mario/more
```

Execute the below to evaluate the style of your code using `style50` .

```
style50 mario.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/mario/more
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Cash

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/cash/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

Greedy Algorithms

When making change, odds are you want to minimize the number of coins you're dispensing for each customer, lest you run out (or annoy the customer!). 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."

What's all that mean? Well, suppose that a cashier owes a customer some change and in that cashier's drawer are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢). The problem to be solved is to decide which coins and how many of each to hand to the customer. Think of a "greedy" cashier as one who wants to take the biggest bite out of this problem as possible with each coin they take out of the drawer. For instance, if some customer is owed 41¢, the biggest first (i.e., best immediate, or local) bite that can be 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. How few? Well, you tell us!

Implementation Details

Implement, in a file called `cash.c` in a `~/pset1/cash` directory, a program that first asks the user how much change is owed and then prints the minimum number of coins with which that change can be made.

- Use `get_float` to get the user's input and `printf` to output your answer. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).
  - We ask that you use `get_float` 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.
- You need not try to check whether a user's input is too large to fit in a `float`. Using `get_float` 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.
- So that we can automate some tests of your code, be sure that your program's last line of output is only the minimum number of coins possible: an integer followed by `\n`.
- Beware the inherent imprecision of floating-point values. Recall `floats.c` from class, wherein, if `x` is `2`, and `y` is `10`, `x / y` is not precisely two tenths! And so, before making change, you'll probably want to convert the user's inputted dollars to cents (i.e., from a `float` to an `int`) to avoid tiny errors that might otherwise add up!
- Take care to round your cents to the nearest penny, as with `round`, which is declared in `math.h`. For instance, if `dollars` is a `float` with the user's input (e.g., `0.20`), then code like

  ```
  int cents = round(dollars * 100);
  ```

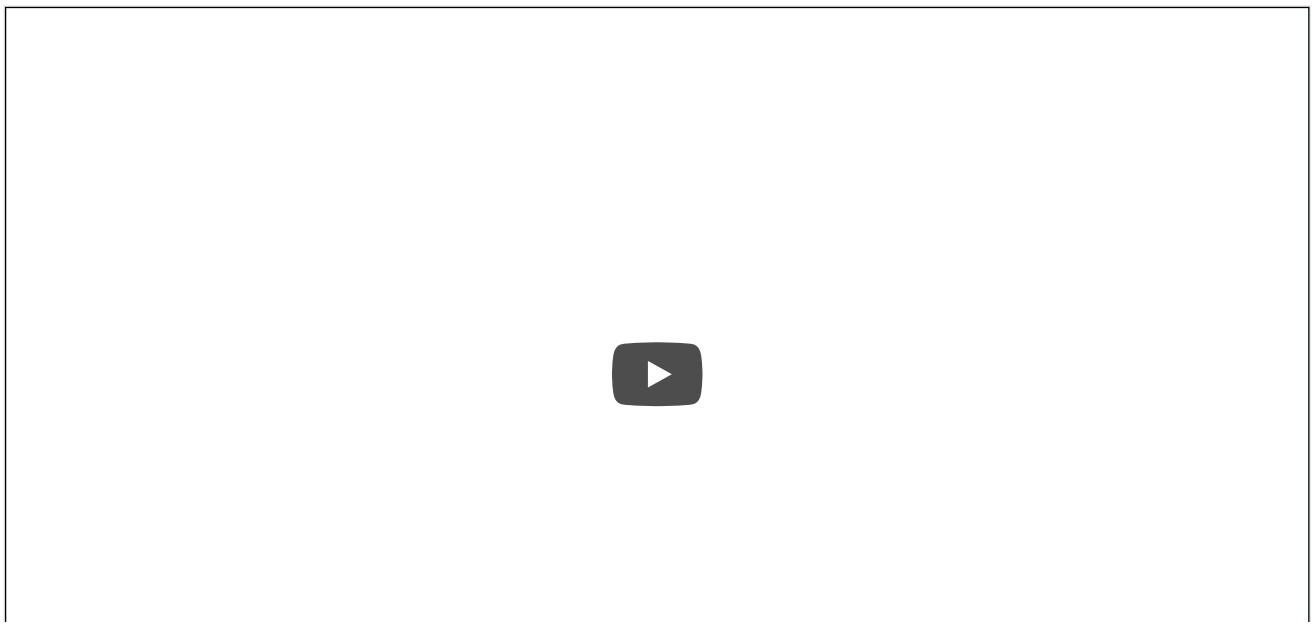
  will safely convert `0.20` (or even `0.20000000298023223876953125`) to `20`.

Your program should behave per the examples below.

```
$ ./cash
Change owed: 0.41
4
```

```
$ ./cash
Change owed: -0.41
Change owed: foo
Change owed: 0.41
4
```

Walkthrough



How to Test Your Code

Does your code work as prescribed when you input

- `-1.00` (or other negative numbers)?
- `0.00` ?
- `0.01` (or other positive numbers)?
- letters or words?
- no input at all, when you only hit Enter?

You can also execute the below to evaluate the correctness of your code using `check50` . But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/cash
```

Execute the below to evaluate the style of your code using `style50` .

```
style50 cash.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/cash
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Credit

If you already started to work on Problem Set 1 in CS50 Lab, you may **continue working on it (https://lab.cs50.io/cs50/labs/2020/x/credit/)** there. If you're just now starting to work in this problem, be sure to use CS50 IDE instead by following the instructions below!

A credit (or debit) card, of course, is a plastic card with which you can pay for goods and services. Printed on that card is a number that's 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! (That's, um, a quadrillion.)

Actually, that's a bit of an exaggeration, because credit card numbers actually have some structure to them. All American Express numbers start with 34 or 37; most MasterCard numbers start with 51, 52, 53, 54, or 55 (they also have some other potential starting numbers which we won't concern ourselves with for this problem); and all Visa numbers 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. 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.

Luhn's Algorithm

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

1. Multiply every other digit by 2, starting with the number's second-to-last digit, and then add those products' digits together.
2. Add the sum to the sum of the digits that weren't multiplied by 2.
3. 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 David's Visa: 4003600000000014.

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

   4003600000000014

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

   1•2 + 0•2 + 0•2 + 0•2 + 0•2 + 6•2 + 0•2 + 4•2

   That gives us:

   2 + 0 + 0 + 0 + 0 + 12 + 0 + 8

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

   2 + 0 + 0 + 0 + 0 + 1 + 2 + 0 + 8 = 13

2. Now let's add that sum (13) to the sum of the digits that weren't multiplied by 2 (starting from the end):

   13 + 4 + 0 + 0 + 0 + 0 + 0 + 3 + 0 = 20

3. Yup, the last digit in that sum (20) is a 0, so David's card is legit!

5. Yup, the last digit in that sum (20) is a 0, so David's card is legit!

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

Implementation Details

In a file called `credit.c` in a `~/pset1/credit/` directory, 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 `get_long` from CS50's library to get users' input. (Why?)

Consider the below representative of how your own program should behave when passed a valid credit card number (sans hyphens).

```
$ ./credit
Number: 4003600000000014
VISA
```

Now, `get_long` itself will reject hyphens (and more) anyway:

```
$ ./credit
Number: 4003-6000-0000-0014
Number: foo
Number: 4003600000000014
VISA
```

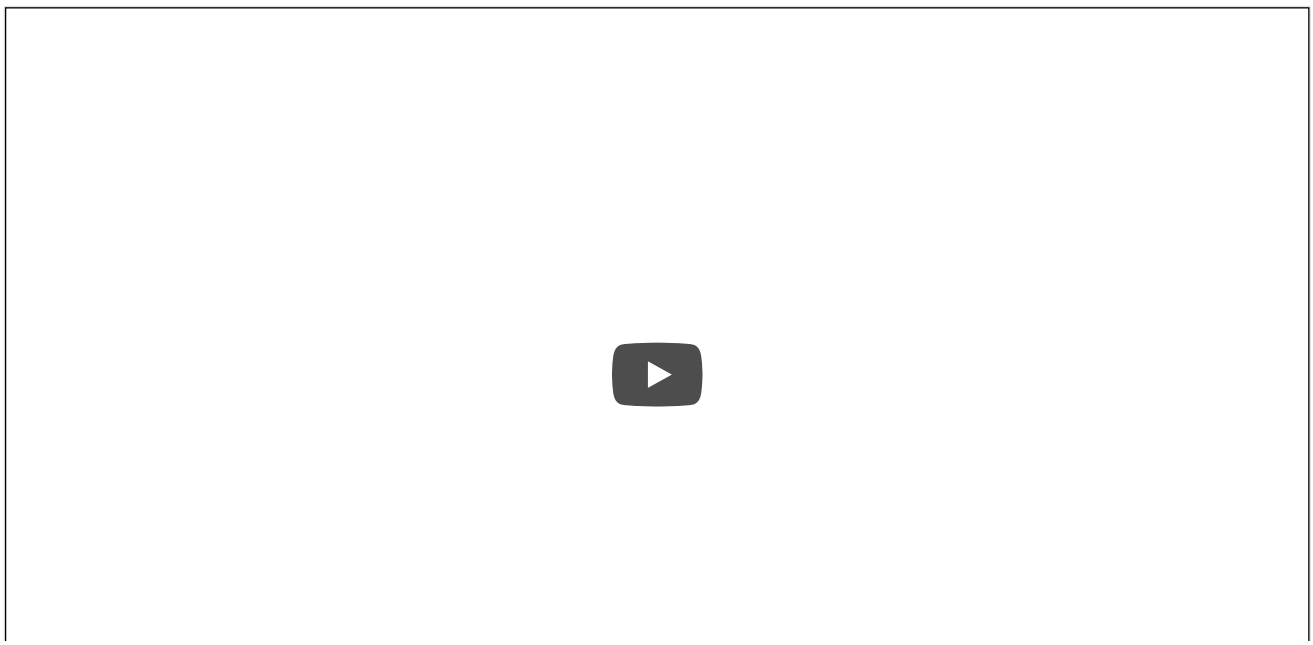
But it's up to you to catch inputs that are not credit card numbers (e.g., a phone number), even if numeric:

```
$ ./credit
Number: 6176292929
INVALID
```

Test out your program with a whole bunch of inputs, both valid and invalid. (We certainly will!) Here are a few card numbers (https://developer.paypal.com/docs/classic/payflow/payflow-pro/payflow-pro-testing/#credit-card-numbers-for-testing) that PayPal recommends for testing.

If your program behaves incorrectly on some inputs (or doesn't compile at all), time to debug!

Walkthrough



How to Test Your Code

You can also execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/credit
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 credit.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/credit
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 2

For this problem set, you'll use CS50 IDE, a cloud-based programming environment.

What to Do

1. Go to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE.
2. Submit Readability
3. Submit one of:
   - Caesar if feeling less comfortable
   - Substitution if feeling more comfortable

If you submit both Caesar and Substitution, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 2's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `hello`, and

  ```
  make readability
  ```

  is yielding errors, try running

  ```
  help50 make readability
  ```

  instead!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Readability

Implement a program that computes the approximate grade level needed to comprehend some text, per the below.

```
$ ./readability
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

Reading Levels

According to Scholastic (https://www.scholastic.com/teachers/teaching-tools/collections/guided-reading-book-lists-for-every-level.html), E.B. White's "Charlotte's Web" is between a second and fourth grade reading level, and Lois Lowry's "The Giver" is between an eighth grade reading level and a twelfth grade reading level. What does it mean, though, for a book to be at a "fourth grade reading level"?

Well, in many cases, a human expert might read a book and make a decision on the grade for which they think the book is most appropriate. But you could also imagine an algorithm attempting to figure out what the reading level of a text is.

So what sorts of traits are characteristic of higher reading levels? Well, longer words probably correlate with higher reading levels. Likewise, longer sentences probably correlate with higher reading levels, too. A number of "readability tests" have been developed over the years, to give a formulaic process for computing the reading level of a text.

One such readability test is the Coleman-Liau index. The Coleman-Liau index of a text is designed to output what (U.S.) grade level is needed to understand the text. The formula is:

```
index = 0.0588 * L - 0.296 * S - 15.8
```

Here, `L` is the average number of letters per 100 words in the text, and `S` is the average number of sentences per 100 words in the text.

Let's write a program called `readability` that takes a text and determines its reading level. For example, if user types in a line from Dr. Seuss:

```
$ ./readability
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

The text the user inputted has 65 letters, 4 sentences, and 14 words. 65 letters per 14 words is an average of about 464.29 letters per 100 words. And 4 sentences per 14 words is an average of about 28.57 sentences per 100 words. Plugged into the Coleman-Liau formula, and rounded to the nearest whole number, we get an answer of 3: so this passage is at a third grade reading level.

Let's try another one:

```
$ ./readability
Text: Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of
Grade 5
```

This text has 214 letters, 4 sentences, and 56 words. That comes out to about 382.14 letters per 100 words, and 7.14 sentences per 100 words. Plugged into the Coleman-Liau formula, we get a fifth grade reading level.

As the average number of letters and words per sentence increases, the Coleman-Liau index gives the text a higher reading level. If you were to take this paragraph, for instance, which has longer words and sentences than either of the prior two examples, the formula would give the text an eleventh grade reading level.

```
$ ./readability
Text: As the average number of letters and words per sentence increases, the Coleman-Liau index gives the text a higher reading l
Grade 11
```

Specification

Design and implement a program, `readability`, that computes the Coleman-Liau index of the text.

- Implement your program in a file called `readability.c` in a directory called `readability`.
- Your program must prompt the user for a `string` of text (using `get_string`).
- Your program should count the number of letters, words, and sentences in the text. You may assume that a letter is any lowercase character from `a` to `z` or any uppercase character from `A` to `Z`, any sequence of characters separated by spaces should count as a word, and that any occurrence of a period, exclamation point, or question mark indicates the end of a sentence.
- Your program should print as output `"Grade X"` where `X` is the grade level computed by the Coleman-Liau formula, rounded to the nearest integer.
- If the resulting index number is 16 or higher (equivalent to or greater than a senior undergraduate reading level), your program should output `"Grade 16+"` instead of giving the exact index number. If the index number is less than 1, your program should output `"Before Grade 1"`.

Getting Started

Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir readability` to make (i.e., create) a directory called `readability` in your home directory.
- Execute `cd readability` to change into (i.e., open) your new `readability` directory.
- Execute `open readability.c` to create and open in the editor an empty file called `readability.c` in your `readability` directory.

Getting User Input

Let's first write some C code that just gets some text input from the user, and prints it back out. Specifically, write code in `readability.c` such that when the user runs the program, they are prompted with `"Text: "` to enter some text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.
In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.
```

Letters

Now that you've collected input from the user, let's begin to analyze that input by first counting the number of letters that show up in the text. Modify `readability.c` so that, instead of printing out the literal text itself, it instead prints out a count of the number of letters in the text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she
235 letter(s)
```

Letters can be any uppercase or lowercase alphabetic characters, but shouldn't include any punctuation, digits, or other symbols.

You can reference https://man.cs50.io/ (https://man.cs50.io/) for standard library functions that may help you here! You may also find that writing a separate function, like `count_letters`, may be useful to keep your code organized.

Words

The Coleman-Liau index cares not only about the number of letters, but also the number of words in a sentence. For the purpose of this problem, we'll consider any sequence of characters separated by a space to be a word (so a hyphenated word like `"sister-in-law"` should be considered one word, not three).

Modify `readability.c` so that, in addition to printing out the number of letters in the text, also prints out the number of words in the text.

You may assume that a sentence will not start or end with a space, and you may assume that a sentence will not have multiple spaces in a row.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his breast i
250 letter(s)
55 word(s)
```

Sentences

The last piece of information that the Coleman-Liau formula cares about, in addition to the number of letters and words, is the number of sentences. Determining the number of sentences can be surprisingly trickly. You might first imagine that a sentence is just any sequence of characters that ends with a period, but of course sentences could end with an exclamation point or a question mark as well. But of course, not all periods necessarily mean the sentence is over. For instance, consider the sentence below.

```
Mr. and Mrs. Dursley, of number four Privet Drive, were proud to say that they were perfectly normal, thank you very much.
```

This is just a single sentence, but there are three periods! For this problem, we'll ask you to ignore that subtlety: you should consider any sequence of characters that ends with a `.` or a `!` or a `?` to be a sentence (so for the above "sentence", you may count that as three sentences). In practice, sentence boundary detection needs to be a little more intelligent to handle these cases, but we'll not worry about that for now.

Modify `readability.c` so that it also now prints out the number of sentences in the text.

The behavior of the resulting program should be like the below.

```
$ ./readability
Text: When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow. When it healed, and Jem's fears of never
295 letter(s)
70 word(s)
3 sentence(s)
```

Putting it All Together

Now it's time to put all the pieces together! Recall that the Coleman-Liau index is computed using the formula:

```
index = 0.0588 * L - 0.296 * S - 15.8
```

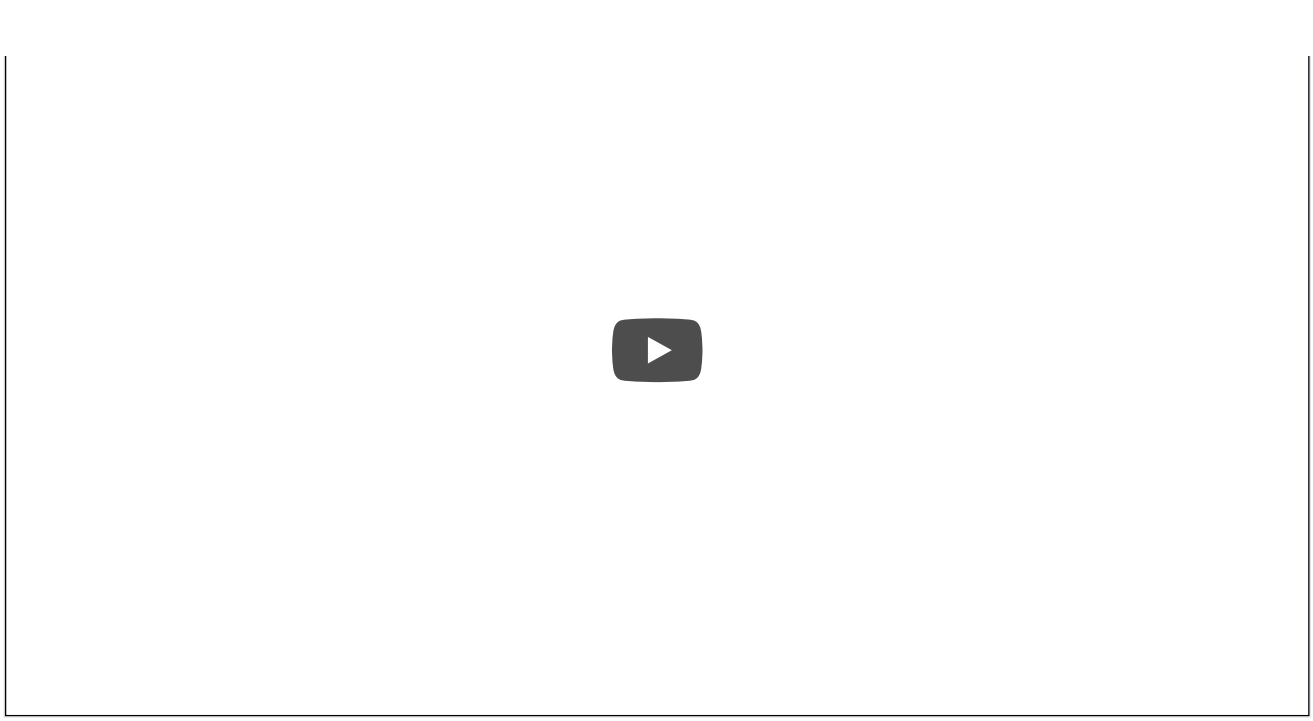
where `L` is the average number of letters per 100 words in the text, and `S` is the average number of sentences per 100 words in the text.

Modify `readability.c` so that instead of outputting the number of letters, words, and sentences, it instead outputs the grade level as given by the Coleman-Liau index (e.g. `"Grade 2"` or `"Grade 8"`). Be sure to round the resulting index number to the nearest whole number!

If the resulting index number is 16 or higher (equivalent to or greater than a senior undergraduate reading level), your program should output `"Grade 16+"` instead of giving the exact index number. If the index number is less than 1, your program should output `"Before Grade 1"`.

▶ **Hints**

Walkthrough



Note that while the walkthrough illustrates that words may be separated by more than one space, you may assume, per the specifications above, that no sentences will contain more than one space in a row.

How to Test Your Code

Try running your program on the following texts.

- `One fish. Two fish. Red fish. Blue fish.` (Before Grade 1)
- `Would you like them here or there? I would not like them here or there. I would not like them anywhere.` (Grade 2)
- `Congratulations! Today is your day. You're off to Great Places! You're off and away!` (Grade 3)
- `Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of year. For another, he really wanted to do his homework, but was forced to do it in secret, in the dead of the night. And he also happened to be a wizard.` (Grade 5)
- `In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.` (Grade 7)
- `Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice "without pictures or conversation?"` (Grade 8)
- `When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow. When it healed, and Jem's fears of never being able to play football were assuaged, he was seldom self-conscious about his injury. His left arm was somewhat shorter than his right; when he stood or walked, the back of his hand was at right angles to his body, his thumb parallel to his thigh.` (Grade 8)
- `There are more things in Heaven and Earth, Horatio, than are dreamt of in your philosophy.` (Grade 9)
- `It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his breast in an effort to escape the vile wind, slipped quickly through the glass doors of Victory Mansions, though not quickly enough to prevent a swirl of gritty dust from entering along with him.` (Grade 10)
- `A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains.` (Grade 16+)

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/readability
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 readability.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (*) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/readability
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Caesar

Implement a program that encrypts messages using Caesar's cipher, per the below.

```
$ ./caesar 13
plaintext:  HELLO
ciphertext: URYYB
```

Background

Supposedly, Caesar (yes, that Caesar) used to "encrypt" (i.e., conceal in a reversible way) confidential messages by shifting each letter therein by some number of places. For instance, he might write A as B, B as C, C as D, …, and, wrapping around alphabetically, Z as A. And so, to say HELLO to someone, Caesar might write IFMMP. Upon receiving such messages from Caesar, recipients would have to "decrypt" them by shifting letters in the opposite direction by the same number of places.

The secrecy of this "cryptosystem" relied on only Caesar and the recipients knowing a secret, the number of places by which Caesar had shifted his letters (e.g., 1). Not particularly secure by modern standards, but, hey, if you're perhaps the first in the world to do it, pretty secure!

Unencrypted text is generally called *plaintext*. Encrypted text is generally called *ciphertext*. And the secret used is called a *key*.

To be clear, then, here's how encrypting `HELLO` with a key of 1 yields `IFMMP`:

plaintext	**H**	**E**	**L**	**L**	**O**
+ key	1	1	1	1	1
= ciphertext	I	F	M	M	P

More formally, Caesar's algorithm (i.e., cipher) encrypts messages by "rotating" each letter by k positions. More formally, 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

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

wherein `% 26` here means "remainder when dividing by 26." This formula perhaps makes the cipher seem more complicated than it is, but it's really just a concise way of expressing the algorithm precisely. Indeed, for the sake of discussion, think of A (or a) as 0, B (or b) as 1, …, H (or h) as 7, I (or i) as 8, …, and Z (or z) as 25. Suppose that Caesar just wants to say Hi to someone confidentially using, this time, a key, k, of 3. And so his plaintext, p, is Hi, in which case his plaintext's first character, p_0, is H (aka 7), and his plaintext's second character, p_1, is i (aka 8). His ciphertext's first character, c_0, is thus K, and his ciphertext's second character, c_1, is thus L. Can you see why?

Let's write a program called `caesar` that enables you to encrypt messages using Caesar's cipher. At the time the user executes the program, they should decide, by providing a command-line argument, on what the key should be in the secret message they'll provide at runtime. We shouldn't necessarily assume that the user's key is going to be a number; though you may assume that, if it is a number, it will be a positive integer.

Here are a few examples of how the program might work. For example, if the user inputs a key of `1` and a plaintext of `HELLO`:

```
$ ./caesar 1
plaintext:  HELLO
ciphertext: IFMMP
```

Here's how the program might work if the user provides a key of `13` and a plaintext of `hello, world`:

```
$ ./caesar 13
plaintext:  hello, world
ciphertext: uryyb, jbeyq
```

Notice that neither the comma nor the space were "shifted" by the cipher. Only rotate alphabetical characters!

How about one more? Here's how the program might work if the user provides a key of `13` again, with a more complex plaintext:

```
$ ./caesar 13
plaintext:  be sure to drink your Ovaltine
ciphertext: or fher gb qevax lbhe Binygvar
```

▶ **Why?**

Notice that the case of the original message has been preserved. Lowercase letters remain lowercase, and uppercase letters remain uppercase.

And what if a user doesn't cooperate?

```
$ ./caesar HELLO
Usage: ./caesar key
```

Or really doesn't cooperate?

```
$ ./caesar
Usage: ./caesar key
```

Or even...

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

Specification

Design and implement a program, `caesar`, that encrypts messages using Caesar's cipher.

- Implement your program in a file called `caesar.c` in a directory called `caesar`.
- Your program must accept a single command-line argument, a non-negative integer. Let's call it *k* for the sake of discussion.
- If your program is executed without any command-line arguments or with more than one command-line argument, your program should print an error message of your choice (with `printf`) and return from `main` a value of `1` (which tends to signify an error) immediately.
- If any of the characters of the command-line argument is not a decimal digit, your program should print the message `Usage: ./caesar key` and return from `main` a value of `1`.
- Do 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`. (Recall that an `int` can overflow.) But, 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, per http://www.asciichart.com/[asciichart.com]; `A` should become `B`, since `B` is 27 positions away from `A`, provided you wrap around from `Z` to `A`.
- Your program must output `plaintext:` (without a newline) and then prompt the user for a `string` of plaintext (using `get_string`).
- Your program must output `ciphertext:` (without a newline) followed by the plaintext's corresponding ciphertext, with each alphabetical character in the plaintext "rotated" by *k* positions; non-alphabetical characters should be outputted unchanged.
- Your program must preserve case: capitalized letters, though rotated, must remain capitalized letters; lowercase letters, though rotated, must remain lowercase letters.
- After outputting ciphertext, you should print a newline. Your program should then exit by returning `0` from `main`.

How to begin? Let's approach this problem one step at a time.

Pseudocode

Pseudocode

First, write some pseudocode that implements this program, even if not (yet!) sure how to write it in code. There's no one right way to write pseudocode, but short English sentences suffice. Recall how we wrote pseudocode for finding Mike Smith (https://cdn.cs50.net/2018/fall/lectures/0/lecture0.pdf). Odds are your pseudocode will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

▶ **Spoiler**

Counting Command-Line Arguments

Whatever your pseudocode, let's first write only the C code that checks whether the program was run with a single command-line argument before adding additional functionality.

Specifically, modify `caesar.c` in such a way that: if the user provides exactly one command-line argument, it prints `Success`; if the user provides no command-line arguments, or two or more, it prints `Usage: ./caesar key`. Remember, since this key is coming from the command line at runtime, and not via `get_string`, we don't have an opportunity to re-prompt the user. The behavior of the resulting program should be like the below.

```
$ ./caesar 20
Success
```

or

```
$ ./caesar
Usage: ./caesar key
```

or

```
$ ./caesar 1 2 3
Usage: ./caesar key
```

▶ **Hints**

Accessing the Key

Now that your program is (hopefully!) accepting input as prescribed, it's time for another step.

Recall that in our program, we must defend against users who technically provide a single command-line argument (the key), but provide something that isn't actually an integer, for example:

```
$ ./caesar xyz
```

Before we start to analyze the key for validity, though, let's make sure we can actually read it. Further modify `caesar.c` such that it not only checks that the user has provided just one command-line argument, but after verifying that, prints out that single command-line argument. So, for example, the behavior might look like this:

```
$ ./caesar 20
Success
20
```

▶ **Hints**

Validating the Key

Now that you know how to read the key, let's analyze it. Modify `caesar.c` such that instead of printing out the command-line argument provided, your program instead checks to make sure that each character of that command line argument is a decimal digit (i.e., `0`, `1`, `2`, etc.) and, if any of them are not, terminates after printing the message `Usage: ./caesar key`. But if the argument consists solely of digit characters, you should convert that string (recall that `argv` is an array of strings, even if those strings happen to look like numbers) to an actual integer, and print out the *integer*, as via `%i` with `printf`. So, for example, the behavior might look like this:

```
$ ./caesar 20
Success
20
```

or

```
$ ./caesar 20x
Usage: ./caesar key
```

▶ **Hints**

Peeking Underneath the Hood

As human beings it's easy for us to intuitively understand the formula described above, inasmuch as we can say "H + 1 = I". But can a computer understand that same logic? Let's find out. For now, we're going to temporarily ignore the key the user provided and instead prompt the user for a secret message and attempt to shift all of its characters by just 1.

Extend the functionality of `caesar.c` such that, after validating the key, we prompt the user for a string and then shift all of its characters by 1, printing out the result. We can also at this point probably remove the line of code we wrote earlier that prints `Success`. All told, this might result in this behavior:

```
$ ./caesar 1
plaintext:  hello
ciphertext: ifmmp
```

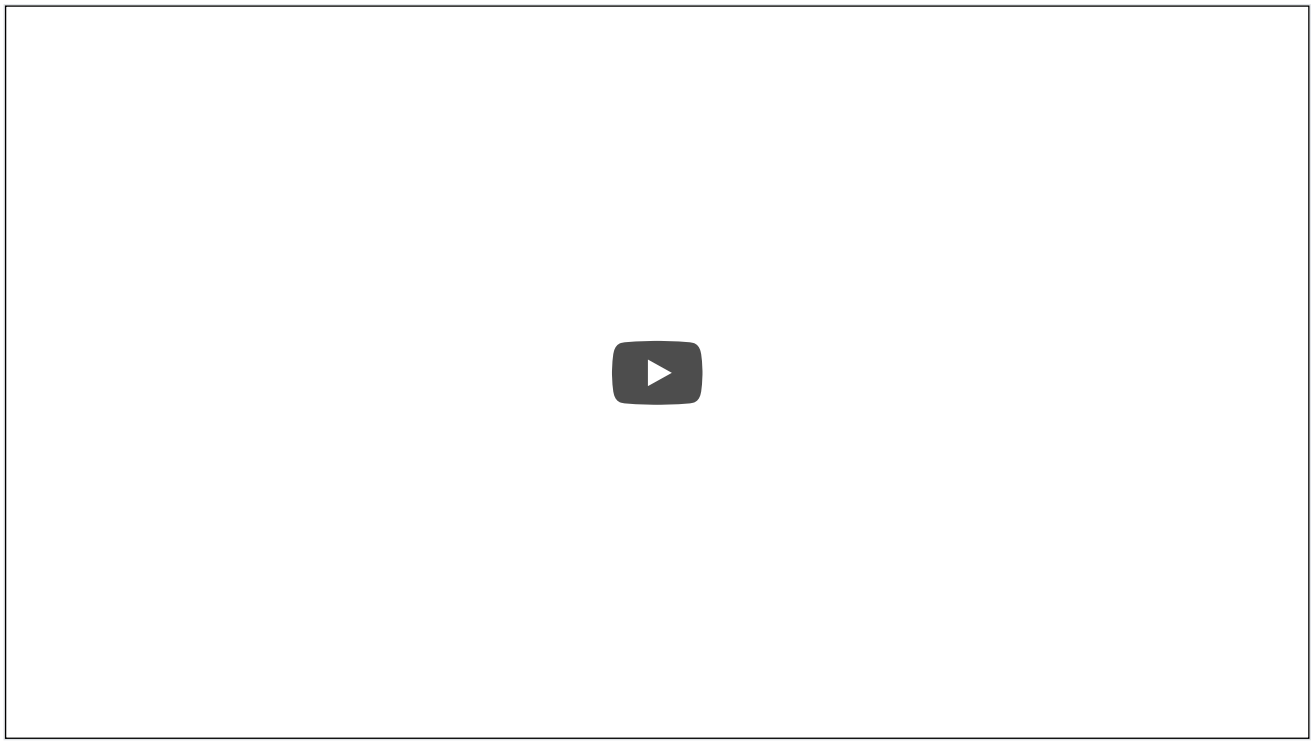
▶ **Hints**

Your Turn

Now it's time to tie everything together! Instead of shifting the characters by 1, modify `caesar.c` to instead shift them by the actual key value. And be sure to preserve case! Uppercase letters should stay uppercase, lowercase letters should stay lowercase, and characters that aren't alphabetical should remain unchanged.

▶ **Hints**

Walkthrough



How to Test Your Code

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/caesar
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 caesar.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/caesar
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Substitution

Implement a program that implements a substitution cipher, per the below.

```
$ ./substitution JTREKYAVOGDXPSNCUIZLFBMWHQ
plaintext:  HELLO
ciphertext: VKXXN
```

Background

In a substitution cipher, we "encrypt" (i.e., conceal in a reversible way) a message by replacing every letter with another letter. To do so, we use a *key*: in this case, a mapping of each of the letters of the alphabet to the letter it should correspond to when we encrypt it. To "decrypt" the message, the receiver of the message would need to know the key, so that they can reverse the process: translating the encrypt text (generally called *ciphertext*) back into the original message (generally called *plaintext*).

A key, for example, might be the string `NQXPOMAFTRHLZGECYJIUWSKDVB`. This 26-character key means that `A` (the first letter of the alphabet) should be converted into `N` (the first character of the key), `B` (the second letter of the alphabet) should be converted into `Q` (the second character of the key), and so forth.

A message like `HELLO`, then, would be encrypted as `FOLLE`, replacing each of the letters according to the mapping determined by the key.

Let's write a program called `substitution` that enables you to encrypt messages using a substitution cipher. At the time the user executes the program, they should decide, by providing a command-line argument, on what the key should be in the secret message they'll provide at runtime.

Here are a few examples of how the program might work. For example, if the user inputs a key of `YTNSHKVEFXRBAUQZCLWDMIPGJO` and a plaintext of `HELLO`:

```
$ ./substitution YTNSHKVEFXRBAUQZCLWDMIPGJO
plaintext:  HELLO
ciphertext: EHBBQ
```

Here's how the program might work if the user provides a key of `VCHPRZGJNTLSKFBDQWAXEUYMOI` and a plaintext of `hello, world`:

```
$ ./substitution VCHPRZGJNTLSKFBDQWAXEUYMOI
plaintext:  hello, world
ciphertext: jrssb, ybwsp
```

Notice that neither the comma nor the space were substituted by the cipher. Only substitute alphabetical characters! Notice, too, that the case of the original message has been preserved. Lowercase letters remain lowercase, and uppercase letters remain uppercase.

Whether the characters in the key itself are uppercase or lowercase doesn't matter. A key of `VCHPRZGJNTLSKFBDQWAXEUYMOI` is functionally identical to a key of `vchprzgjntlskfbdqwaxeuymoi` (as is, for that matter, `VcHpRzGjNtLsKfBdQwAxEuYmOi`).

And what if a user doesn't provide a valid key?

```
$ ./substitution ABC
Key must contain 26 characters.
```

Or really doesn't cooperate?

```
$ ./substitution
Usage: ./substitution key
```

Or even...

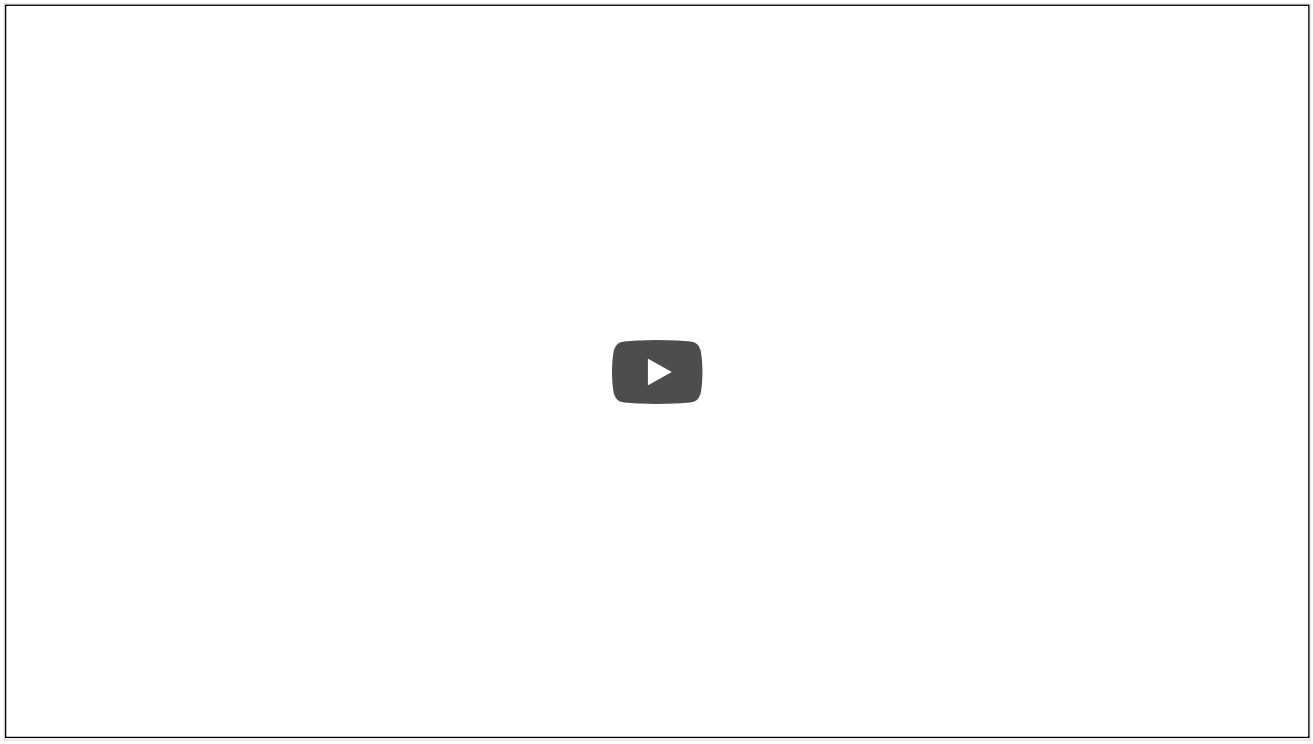
```
$ ./substitution 1 2 3
Usage: ./substitution key
```

Specification

Design and implement a program, `substitution`, that encrypts messages using a substitution cipher.

- Implement your program in a file called `substitution.c` in a directory called `substitution.
- Your program must accept a single command-line argument, the key to use for the substitution. The key itself should be case-insensitive, so whether any character in the key is uppercase or lowercase should not affect the behavior of your program.
- If your program is executed without any command-line arguments or with more than one command-line argument, your program should print an error message of your choice (with `printf`) and return from `main` a value of `1` (which tends to signify an error) immediately.
- If the key is invalid (as by not containing 26 characters, containing any character that is not an alphabetic character, or not containing each letter exactly once), your program should print an error message of your choice (with `printf`) and return from `main` a value of `1` immediately.
- Your program must output `plaintext:` (without a newline) and then prompt the user for a `string` of plaintext (using `get_string`).
- Your program must output `ciphertext:` (without a newline) followed by the plaintext's corresponding ciphertext, with each alphabetical character in the plaintext substituted for the corresponding character in the ciphertext; non-alphabetical characters should be outputted unchanged.
- Your program must preserve case: capitalized letters must remain capitalized letters; lowercase letters must remain lowercase letters.
- After outputting ciphertext, you should print a newline. Your program should then exit by returning `0` from `main`.

Walkthrough



How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/substitution
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 substitution.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/substitution
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 3

What to Do

1. Submit Plurality
2. Submit one of:
    - Runoff if feeling less comfortable
    - Tideman if feeling more comfortable

If you submit both Runoff and Tideman, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 3's sandboxes.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `plurality`, and

  ```
  make plurality
  ```

  is yielding errors, try running

  ```
  help50 make plurality
  ```

  instead!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Plurality

Implement a program that runs a plurality election, per the below.

```
$ ./plurality Alice Bob Charlie
Number of voters: 4
Vote: Alice
Vote: Bob
Vote: Charlie
Vote: Alice
Alice
```

Background

Elections come in all shapes and sizes. In the UK, the Prime Minister (https://www.parliament.uk/education/about-your-parliament/general-elections/) is officially appointed by the monarch, who generally chooses the leader of the political party that wins the most seats in the House of Commons. The United States uses a multi-step Electoral College (https://www.archives.gov/federal-register/electoral-college/about.html) process where citizens vote on how each state should allocate Electors who then elect the President.

Perhaps the simplest way to hold an election, though, is via a method commonly known as the "plurality vote" (also known as "first-past-the-post" or "winner take all"). In the plurality vote, every voter gets to vote for one candidate. At the end of the election, whichever candidate has the greatest number of votes is declared the winner of the election.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir pset3` to make (i.e., create) a directory called `pset3` in your home directory.
- Execute `cd pset3` to change into (i.e., open) that directory.
- Execute `mkdir plurality` to make (i.e., create) a directory called `plurality` in your `pset3` directory.
- Execute `cd plurality` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/plurality/plurality.c` to download this problem's distribution code.
- Execute `ls`. You should see this problem's distribution code, in a file called `plurality.c`.

Understanding

Let's now take a look at `plurality.c` and read through the distribution code that's been provided to you.

The line `#define MAX 9` is some syntax used here to mean that `MAX` is a constant (equal to `9`) that can be used throughout the program. Here, it represents the maximum number of candidates an election can have.

The file then defines a `struct` called a `candidate`. Each `candidate` has two fields: a `string` called `name` representing the candidate's name, and an `int` called `votes` representing the number of votes the candidate has. Next, the file defines a global array of `candidates`, where each element is itself a `candidate`.

Now, take a look at the `main` function itself. See if you can find where the program sets a global variable `candidate_count` representing the

Now, take a look at the `main` function itself. See if you can find where the program sets a global variable `candidate_count` representing the number of candidates in the election, copies command-line arguments into the array `candidates`, and asks the user to type in the number of voters. Then, the program lets every voter type in a vote (see how?), calling the `vote` function on each candidate voted for. Finally, `main` makes a call to the `print_winner` function to print out the winner (or winners) of the election.

If you look further down in the file, though, you'll notice that the `vote` and `print_winner` functions have been left blank. This part is up to you to complete!

Specification

Complete the implementation of `plurality.c` in such a way that the program simulates a plurality vote election.

- Complete the `vote` function.
  - `vote` takes a single argument, a `string` called `name`, representing the name of the candidate who was voted for.
  - If `name` matches one of the names of the candidates in the election, then update that candidate's vote total to account for the new vote. The `vote` function in this case should return `true` to indicate a successful ballot.
  - If `name` does not match the name of any of the candidates in the election, no vote totals should change, and the `vote` function should return `false` to indicate an invalid ballot.
  - You may assume that no two candidates will have the same name.
- Complete the `print_winner` function.
  - The function should print out the name of the candidate who received the most votes in the election, and then print a newline.
  - It is possible that the election could end in a tie if multiple candidates each have the maximum number of votes. In that case, you should output the names of each of the winning candidates, each on a separate line.

You should not modify anything else in `plurality.c` other than the implementations of the `vote` and `print_winner` functions (and the inclusion of additional header files, if you'd like).

Usage

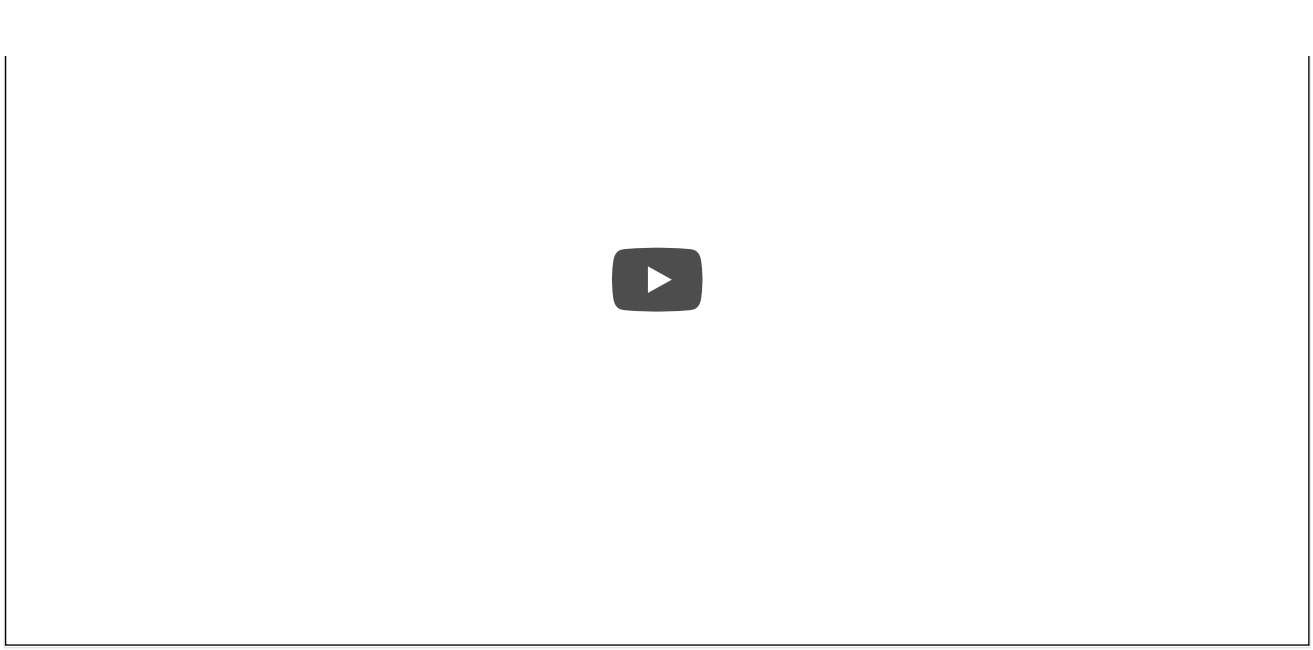
Your program should behave per the examples below.

```
$ ./plurality Alice Bob
Number of voters: 3
Vote: Alice
Vote: Bob
Vote: Alice
Alice
```

```
$ ./plurality Alice Bob
Number of voters: 3
Vote: Alice
Vote: Charlie
Invalid vote.
Vote: Alice
Alice
```

```
$ ./plurality Alice Bob Charlie
Number of voters: 5
Vote: Alice
Vote: Charlie
Vote: Bob
Vote: Bob
Vote: Alice
Alice
Bob
```

Walkthrough



Testing

Be sure to test your code to make sure it handles…

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election if there is only one
- Printing the winner of the election if there are multiple winners

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/plurality
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 plurality.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/plurality
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Runoff

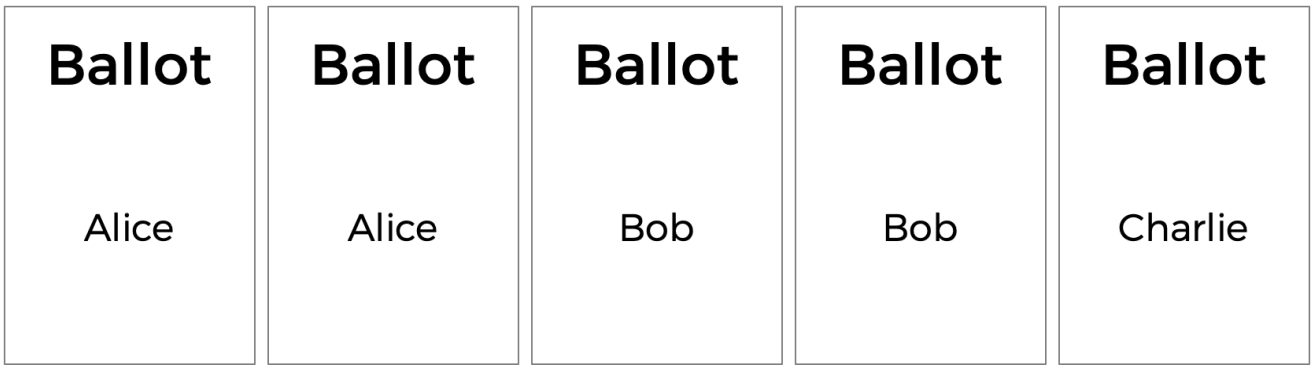
Implement a program that runs a runoff election, per the below.

```
./runoff Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Bob
Rank 3: Charlie

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Alice
Rank 3: Charlie

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Alice
```

Background

You already know about plurality elections, which follow a very simple algorithm for determining the winner of an election: every voter gets one vote, and the candidate with the most votes wins.

But the plurality vote does have some disadvantages. What happens, for instance, in an election with three candidates, and the ballots below are cast?



A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

There's another kind of voting system known as a ranked-choice voting system. In a ranked-choice system, voters can vote for more than one candidate. Instead of just voting for their top choice, they can rank the candidates in order of preference. The resulting ballots might therefore look like the below

look like the below:

Ballot	Ballot	Ballot	Ballot	Ballot
1. Alice	1. Alice	1. Bob	1. Bob	1. Charlie
2. Bob	2. Charlie	2. Alice	2. Alice	2. Alice
3. Charlie	3. Bob	3. Charlie	3. Charlie	3. Bob

Here, each voter, in addition to specifying their first preference candidate, has also indicated their second and third choices. And now, what was previously a tied election could now have a winner. The race was originally tied between Alice and Bob, so Charlie was out of the running. But the voter who chose Charlie preferred Alice over Bob, so Alice could here be declared the winner.

Ranked choice voting can also solve yet another potential drawback of plurality voting. Take a look at the following ballots.

Ballot	Ballot	Ballot	Ballot	Ballot
1. Alice	1. Alice	1. Bob	1. Bob	1. Bob
2. Bob	2. Bob	2. Alice	2. Alice	2. Alice
3. Charlie	3. Charlie	3. Charlie	3. Charlie	3. Charlie

Ballot	Ballot	Ballot	Ballot
1. Charlie	1. Charlie	1. Charlie	1. Charlie
2. Alice	2. Alice	2. Bob	2. Bob
3. Bob	3. Bob	3. Alice	3. Alice

Who should win this election? In a plurality vote where each voter chooses their first preference only, Charlie wins this election with four votes compared to only three for Bob and two for Alice. But a majority of the voters (5 out of the 9) would be happier with either Alice or Bob instead of Charlie. By considering ranked preferences, a voting system may be able to choose a winner that better reflects the preferences of the voters.

One such ranked choice voting system is the instant runoff system. In an instant runoff election, voters can rank as many candidates as they wish. If any candidate has a majority (more than 50%) of the first preference votes, that candidate is declared the winner of the election.

If no candidate has more than 50% of the vote, then an “instant runoff” occurrs. The candidate who received the fewest number of votes is eliminated from the election, and anyone who originally chose that candidate as their first preference now has their second preference considered. Why do it this way? Effectively, this simulates what would have happened if the least popular candidate had not been in the election to begin with.

The process repeats: if no candidate has a majority of the votes, the last place candidate is eliminated, and anyone who voted for them will instead vote for their next preference (who hasn't themselves already been eliminated). Once a candidate has a majority, that candidate is declared the winner.

Let's consider the nine ballots above and examine how a runoff election would take place.

Alice has two votes, Bob has three votes, and Charlie has four votes. To win an election with nine people, a majority (five votes) is required. Since nobody has a majority, a runoff needs to be held. Alice has the fewest number of votes (with only two), so Alice is eliminated. The voters who originally voted for Alice listed Bob as second preference, so Bob gets the extra two vote. Bob now has five votes, and Charlie still has four votes. Bob now has a majority, and Bob is declared the winner.

What corner cases do we need to consider here?

One possibility is that there's a tie for who should get eliminated. We can handle that scenario by saying all candidates who are tied for last place will be eliminated. If every remaining candidate has the exact same number of votes, though, eliminating the tied last place candidates means eliminating everyone! So in that case, we'll have to be careful not to eliminate everyone, and just declare the election a tie between all remaining candidates.

Some instant runoff elections don't require voters to rank all of their preferences — so there might be five candidates in an election, but a voter might only choose two. For this problem's purposes, though, we'll ignore that particular corner case, and assume that all voters will rank all of the candidates in their preferred order.

Sounds a bit more complicated than a plurality vote, doesn't it? But it arguably has the benefit of being an election system where the winner of the election more accurately represents the preferences of the voters.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `cd pset3` to change into (i.e., open) your `pset3` directory that should already exist.
- Execute `mkdir runoff` to make (i.e., create) a directory called `runoff` in your `pset3` directory.
- Execute `cd runoff` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/runoff/runoff.c` to download this problem's distribution code.
- Execute `ls`. You should see this problem's distribution code, in a file called `runoff.c`.

Understanding

Let's open up `runoff.c` to take a look at what's already there. We're defining two constants: `MAX_CANDIDATES` for the maximum number of candidates in the election, and `MAX_VOTERS` for the maximum number of voters in the election.

Next up is a two-dimensional array `preferences`. The array `preferences[i]` will represent all of the preferences for voter number `i`, and the integer `preferences[i][j]` here will store the index of the candidate who is the `j` th preference for voter `i`.

Next up is a `struct` called `candidate`. Every `candidate` has a `string` field for their `name`, and `int` representing the number of `votes` they currently have, and a `bool` value called `eliminated` that indicates whether the candidate has been eliminated from the election. The array `candidates` will keep track of all of the candidates in the election.

The program also has two global variables: `voter_count` and `candidate_count`.

Now onto `main`. Notice that after determining the number of candidates and the number of voters, the main voting loop begins, giving every voter a chance to vote. As the voter enters their preferences, the `vote` function is called to keep track of all of the preferences. If at any point, the ballot is deemed to be invalid, the program exits.

Once all of the votes are in, another loop begins: this one's going to keep looping through the runoff process of checking for a winner and eliminating the last place candidate until there is a winner.

The first call here is to a function called `tabulate`, which should look at all of the voters' preferences and compute the current vote totals, by looking at each voter's top choice candidate who hasn't yet been eliminated. Next, the `print_winner` function should print out the winner if there is one; if there is, the program is over. But otherwise, the program needs to determine the fewest number of votes anyone still in the election received (via a call to `find_min`). If it turns out that everyone in the election is tied with the same number of votes (as determined by the `is_tie` function), the election is declared a tie; otherwise, the last-place candidate (or candidates) is eliminated from the election via a call to the `eliminate` function.

to the `eliminate` function.

If you look a bit further down in the file, you'll see that these functions — `vote`, `tabulate`, `print_winner`, `find_min`, `is_tie`, and `eliminate` — are all left up to you to complete!

Specification

Complete the implementation of `runoff.c` in such a way that it simulates a runoff election. You should complete the implementations of the `vote`, `tabulate`, `print_winner`, `find_min`, `is_tie`, and `eliminate` functions, and you should not modify anything else in `runoff.c` (except you may include additional header files, if you'd like).

`vote`

Complete the `vote` function.

- The function takes arguments `voter`, `rank`, and `name`. If `name` is a match for the name of a valid candidate, then you should update the global preferences array to indicate that the voter `voter` has that candidate as their `rank` preference (where `0` is the first preference, `1` is the second preference, etc.).
- If the preference is successfully recorded, the function should return `true`; the function should return `false` otherwise (if, for instance, `name` is not the name of one of the candidates).
- You may assume that no two candidates will have the same name.

▶ **Hints**

`tabulate`

Complete the `tabulate` function.

- The function should update the number of `votes` each candidate has at this stage in the runoff.
- Recall that at each stage in the runoff, every voter effectively votes for their top-preferred candidate who has not already been eliminated.

▶ **Hints**

`print_winner`

Complete the `print_winner` function.

- If any candidate has more than half of the vote, their name should be printed to `stdout` and the function should return `true`.
- If nobody has won the election yet, the function should return `false`.

▶ **Hints**

`find_min`

Complete the `find_min` function.

- The function should return the minimum vote total for any candidate who is still in the election.

▶ **Hints**

`is_tie`

Complete the `is_tie` function.

- The function takes an argument `min`, which will be the minimum number of votes that anyone in the election currently has.
- The function should return `true` if every candidate remaining in the election has the same number of votes, and should return `false` otherwise.

▶ **Hints**

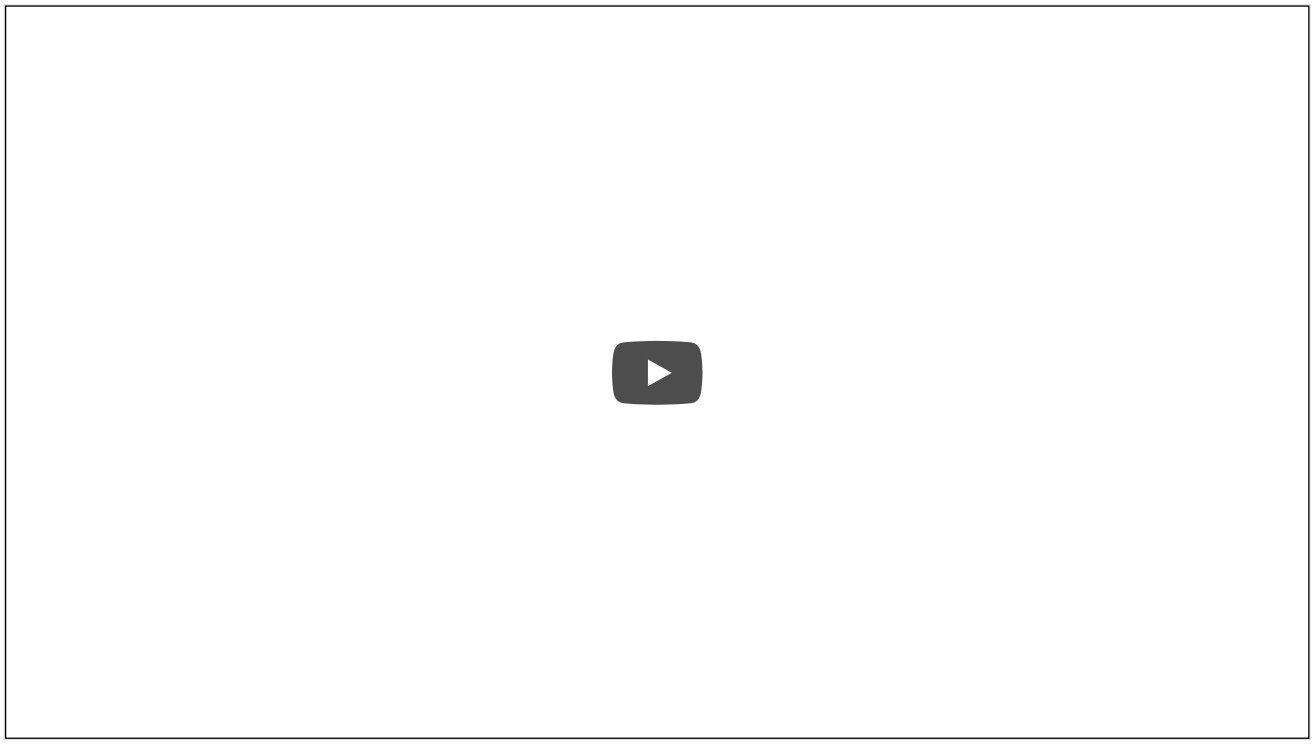
`eliminate`

Complete the `eliminate` function.

- The function takes an argument `min`, which will be the minimum number of votes that anyone in the election currently has

- The function takes an argument `min`, which will be the minimum number of votes that anyone in the election currently has.
- The function should eliminate the candidate (or candidates) who have `min` number of votes.

Walkthrough



Usage

Your program should behave per the example below:

```
./runoff Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Alice
```

Testing

Be sure to test your code to make sure it handles...

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election if there is only one

- Printing the winner of the election if there is only one
- Not eliminating anyone in the case of a tie between all remaining candidates

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/runoff
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 runoff.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/runoff
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Tideman

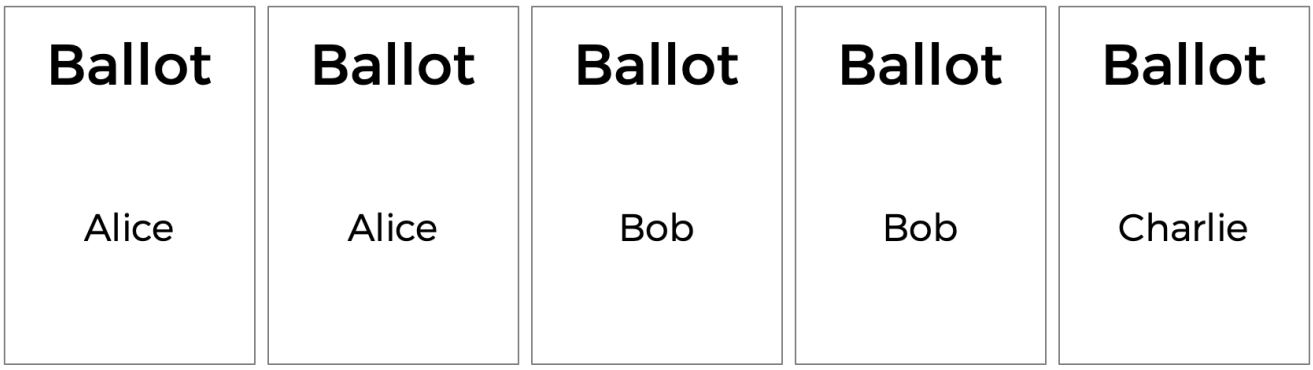
Implement a program that runs a Tideman election, per the below.

```
./tideman Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Charlie
```

Background

You already know about plurality elections, which follow a very simple algorithm for determining the winner of an election: every voter gets one vote, and the candidate with the most votes wins.

But the plurality vote does have some disadvantages. What happens, for instance, in an election with three candidates, and the ballots below are cast?



A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

There's another kind of voting system known as a ranked-choice voting system. In a ranked-choice system, voters can vote for more than one candidate. Instead of just voting for their top choice, they can rank the candidates in order of preference. The resulting ballots might therefore look like the below

look like the below:

Ballot	Ballot	Ballot	Ballot	Ballot
1. Alice	1. Alice	1. Bob	1. Bob	1. Charlie
2. Bob	2. Charlie	2. Alice	2. Alice	2. Alice
3. Charlie	3. Bob	3. Charlie	3. Charlie	3. Bob

Here, each voter, in addition to specifying their first preference candidate, has also indicated their second and third choices. And now, what was previously a tied election could now have a winner. The race was originally tied between Alice and Bob. But the voter who chose Charlie preferred Alice over Bob, so Alice could here be declared the winner.

Ranked choice voting can also solve yet another potential drawback of plurality voting. Take a look at the following ballots.

Ballot	Ballot	Ballot	Ballot	Ballot
1. Alice	1. Alice	1. Bob	1. Bob	1. Bob
2. Charlie	2. Charlie	2. Alice	2. Alice	2. Alice
3. Bob	3. Bob	3. Charlie	3. Charlie	3. Charlie

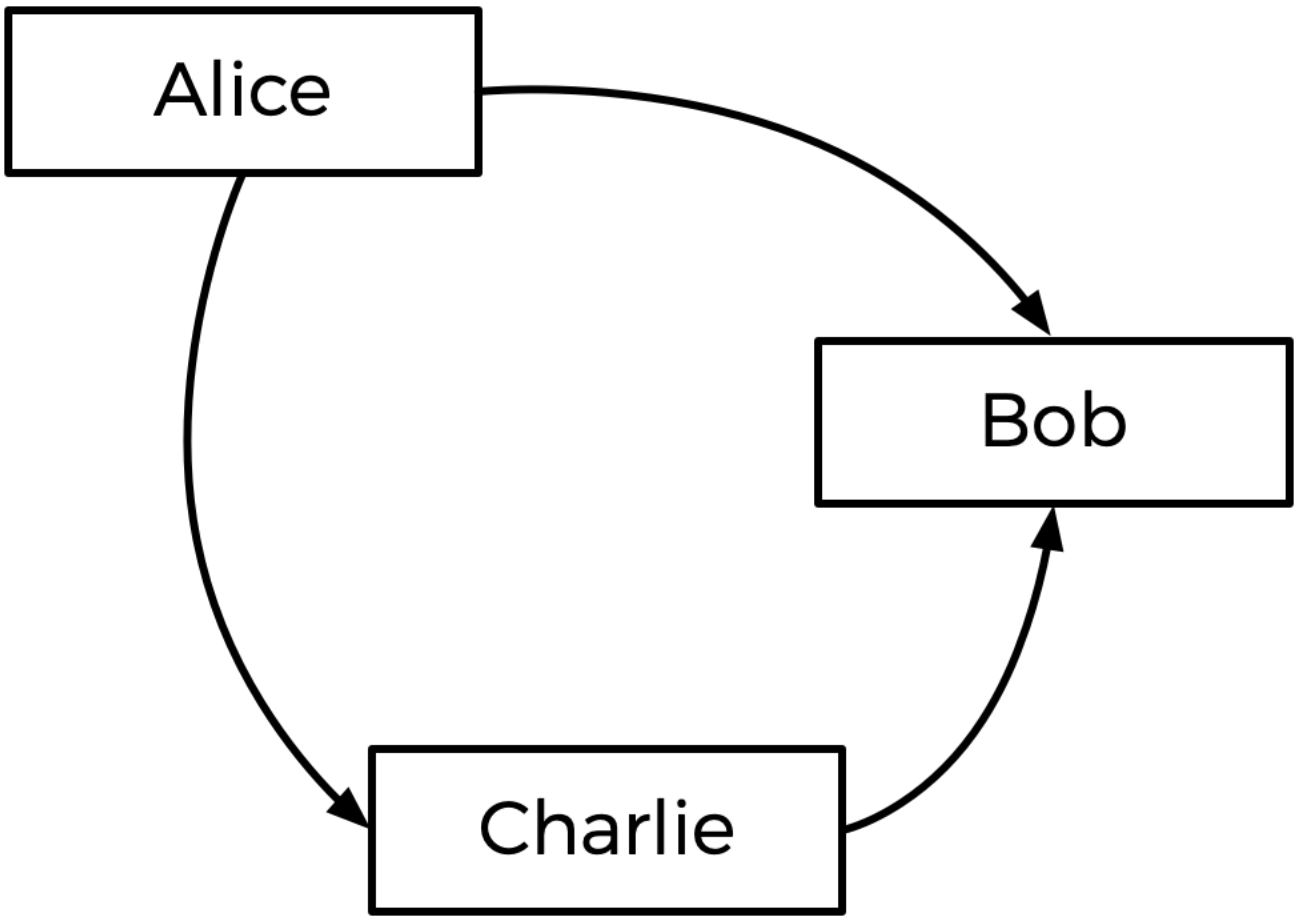
Ballot	Ballot	Ballot	Ballot
1. Charlie	1. Charlie	1. Charlie	1. Charlie
2. Alice	2. Alice	2. Alice	2. Bob
3. Bob	3. Bob	3. Bob	3. Alice

Who should win this election? In a plurality vote where each voter chooses their first preference only, Charlie wins this election with four votes compared to only three for Bob and two for Alice. (Note that, if you're familiar with the instant runoff voting system, Charlie wins here under that system as well). Alice, however, might reasonably make the argument that she should be the winner of the election instead of Charlie: after all, of the nine voters, a majority (five of them) preferred Alice over Charlie, so most people would be happier with Alice as the winner instead of Charlie.

Alice is, in this election, the so-called "Condorcet winner" of the election: the person who would have won any head-to-head matchup against another candidate. If the election had been just Alice and Bob, or just Alice and Charlie, Alice would have won.

The Tideman voting method (also known as "ranked pairs") is a ranked-choice voting method that's guaranteed to produce the Condorcet winner of the election if one exists.

Generally speaking, the Tideman method works by constructing a "graph" of candidates, where an arrow (i.e. edge) from candidate A to candidate B indicates that candidate A wins against candidate B in a head-to-head matchup. The graph for the above election, then, would look like the below.



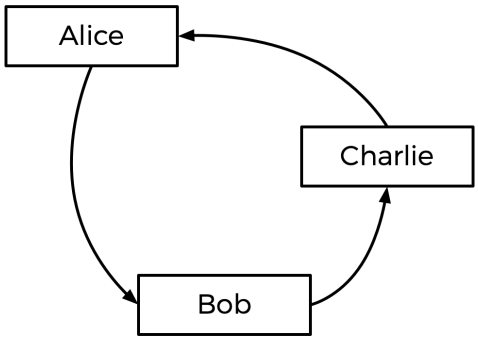
The arrow from Alice to Bob means that more voters prefer Alice to Bob (5 prefer Alice, 4 prefer Bob). Likewise, the other arrows mean that more voters prefer Alice to Charlie, and more voters prefer Charlie to Bob.

Looking at this graph, the Tideman method says the winner of the election should be the "source" of the graph (i.e. the candidate that has no arrow pointing at them). In this case, the source is Alice — Alice is the only one who has no arrow pointing at her, which means nobody is preferred head-to-head over Alice. Alice is thus declared the winner of the election.

It's possible, however, that when the arrows are drawn, there is no Condorcet winner. Consider the below ballots.

Ballot	Ballot	Ballot	Ballot	Ballot
1. Alice	1. Alice	1. Alice	1. Bob	1. Bob
2. Bob	2. Bob	2. Bob	2. Charlie	2. Charlie
3. Charlie	3. Charlie	3. Charlie	3. Alice	3. Alice

Ballot	Ballot	Ballot	Ballot
1. Charlie	1. Charlie	1. Charlie	1. Charlie
2. Alice	2. Alice	2. Alice	2. Alice
3. Bob	3. Bob	3. Bob	3. Bob



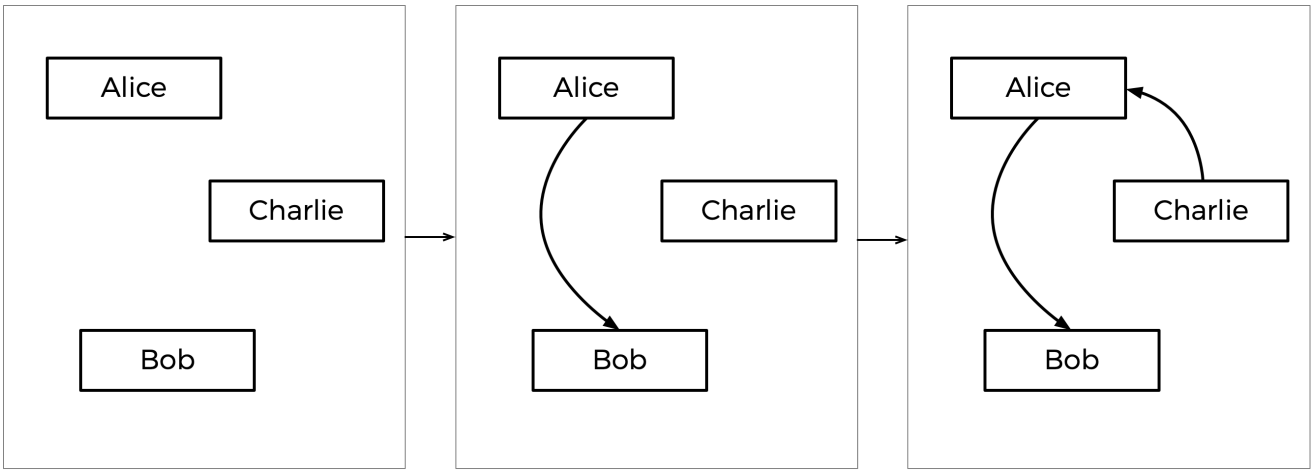
Between Alice and Bob, Alice is preferred over Bob by a 7-2 margin. Between Bob and Charlie, Bob is preferred over Charlie by a 5-4 margin. But between Charlie and Alice, Charlie is preferred over Alice by a 6-3 margin. If we draw out the graph, there is no source! We have a cycle of candidates, where Alice beats Bob who beats Charlie who beats Alice (much like a game of rock-paper-scissors). In this case, it looks like there's no way to pick a winner.

To handle this, the Tideman algorithm must be careful to avoid creating cycles in the candidate graph. How does it do this? The algorithm locks in the strongest edges first, since those are arguably the most significant. In particular, the Tideman algorithm specifies that matchup edges should be "locked in" to the graph one at a time, based on the "strength" of the victory (the more people who prefer a candidate over their opponent, the stronger the victory). So long as the edge can be locked into the graph without creating a cycle, the edge is added; otherwise, the edge is ignored.

How would this work in the case of the votes above? Well, the biggest margin of victory for a pair is Alice beating Bob, since 7 voters prefer Alice over Bob (no other head-to-head matchup has a winner preferred by more than 7 voters). So the Alice-Bob arrow is locked into the graph first. The next biggest margin of victory is Charlie's 6-3 victory over Alice, so that arrow is locked in next.

Next up is Bob's 5-4 victory over Charlie. But notice: if we were to add an arrow from Bob to Charlie now, we would create a cycle! Since the graph can't allow cycles, we should skip this edge, and not add it to the graph at all. If there were more arrows to consider, we would look to those next, but that was the last arrow, so the graph is complete.

This step-by-step process is shown below, with the final graph at right.



Based on the resulting graph, Charlie is the source (there's no arrow pointing towards Charlie), so Charlie is declared the winner of this election.

Put more formally, the Tideman voting method consists of three parts:

- **Tally**: Once all of the voters have indicated all of their preferences, determine, for each pair of candidates, who the preferred candidate is and by what margin they are preferred.
- **Sort**: Sort the pairs of candidates in decreasing order of strength of victory, where strength of victory is defined to be the number of voters who prefer the preferred candidate.
- **Lock**: Starting with the strongest pair, go through the pairs of candidates in order and "lock in" each pair to the candidate graph, so long as locking in that pair does not create a cycle in the graph.

Once the graph is complete, the source of the graph (the one with no edges pointing towards it) is the winner!

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `cd pset3` to change into (i.e., open) your `pset3` directory that should already exist.
- Execute `mkdir tideman` to make (i.e., create) a directory called `tideman` in your `pset3` directory.
- Execute `cd tideman` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/3/tideman/tideman.c` to download this problem's distribution code.
- Execute `ls`. You should see this problem's distribution code, in a file called `tideman.c`.

Understanding

Let's open up `tideman.c` to take a look at what's already there.

First, notice the two-dimensional array `preferences`. The integer `preferences[i][j]` will represent the number of voters who prefer candidate `i` over candidate `j`.

The file also defines another two-dimensional array, called `locked`, which will represent the candidate graph. `locked` is a boolean array, so `locked[i][j]` being `true` represents the existence of an edge pointing from candidate `i` to candidate `j`; `false` means there is no edge. (If curious, this representation of a graph is known as an "adjacency matrix").

Next up is a `struct` called `pair`, used to represent a pair of candidates: each pair includes the `winner`'s candidate index and the `loser`'s candidate index.

The candidates themselves are stored in the array `candidates`, which is an array of `string`s representing the names of each of the candidates. There's also an array of `pairs`, which will represent all of the pairs of candidates (for which one is preferred over the other) in the election.

The program also has two global variables: `pair_count` and `candidate_count`, representing the number of pairs and number of candidates in the arrays `pairs` and `candidates`, respectively.

Now onto `main`. Notice that after determining the number of candidates, the program loops through the `locked` graph and initially sets all of the values to `false`, which means our initial graph will have no edges in it.

Next, the program loops over all of the voters and collects their preferences in an array called `ranks` (via a call to `vote`), where `ranks[i]` is the index of the candidate who is the `i` th preference for the voter. These ranks are passed into the `record_preference` function, whose job it is to take those ranks and update the global `preferences` variable.

Once all of the votes are in, the pairs of candidates are added to the `pairs` array via a called to `add_pairs`, sorted via a call to `sort_pairs`, and locked into the graph via a call to `lock_pairs`. Finally, `print_winner` is called to print out the name of the election's winner!

Further down in the file, you'll see that the functions `vote`, `record_preference`, `add_pairs`, `sort_pairs`, `lock_pairs`, and `print_winner` are left blank. That's up to you!

Specification

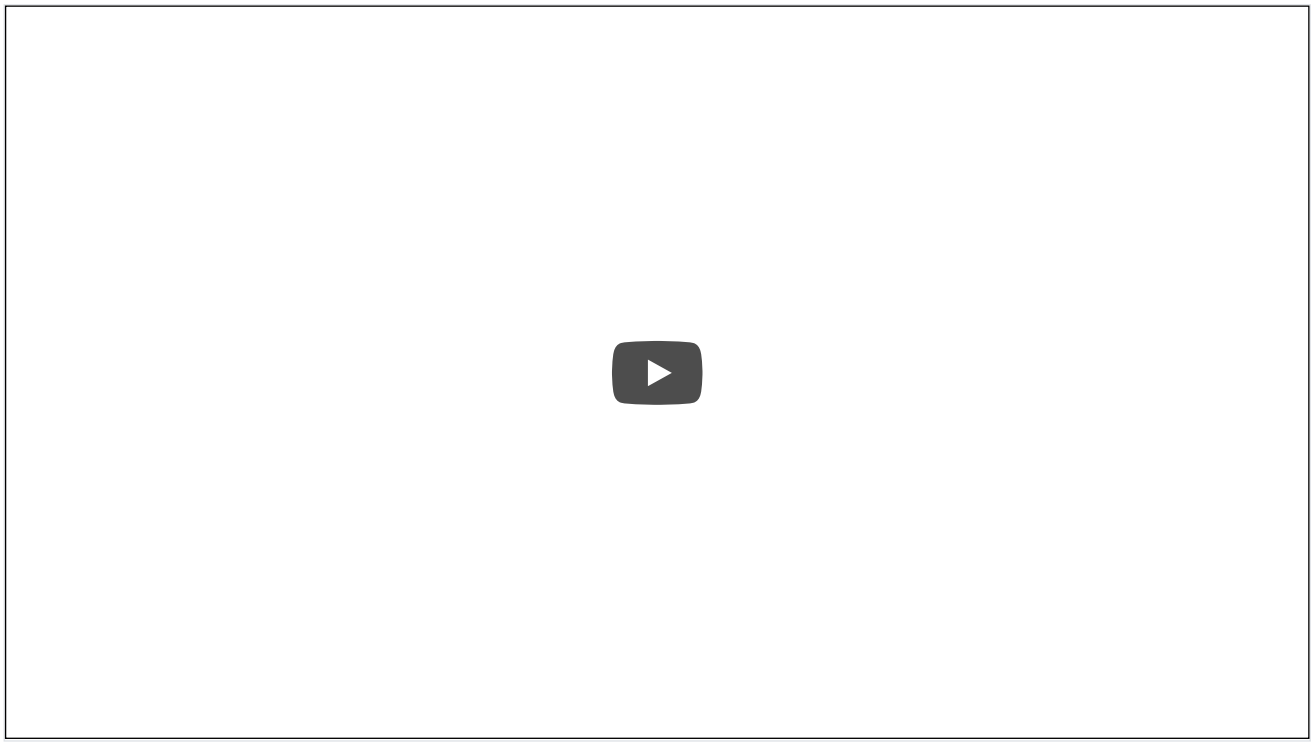
Complete the implementation of `tideman.c` in such a way that it simulates a Tideman election.

- Complete the `vote` function.
  - The function takes arguments `rank`, `name`, and `ranks`. If `name` is a match for the name of a valid candidate, then you should update the `ranks` array to indicate that the voter has the candidate as their `rank` preference (where `0` is the first preference, `1` is the second preference, etc.)
  - Recall that `ranks[i]` here represents the user's `i` th preference.
  - The function should return `true` if the rank was successfully recorded, and `false` otherwise (if, for instance, `name` is not the name of one of the candidates).
  - You may assume that no two candidates will have the same name.
- Complete the `record_preferences` function.
  - The function is called once for each voter, and takes as argument the `ranks` array, (recall that `ranks[i]` is the voter's `i` th preference, where `ranks[0]` is the first preference).
  - The function should update the global `preferences` array to add the current voter's preferences. Recall that `preferences[i][j]` should represent the number of voters who prefer candidate `i` over candidate `j`.
  - You may assume that every voter will rank each of the candidates.
- Complete the `add_pairs` function.
  - The function should add all pairs of candidates where one candidate is preferred to the `pairs` array. A pair of candidates who are tied (one is not preferred over the other) should not be added to the array.
  - The function should update the global variable `pair_count` to be the number of pairs of candidates. (The pairs should thus all be stored between `pairs[0]` and `pairs[pair_count - 1]`, inclusive).
- Complete the `sort_pairs` function.
  - The function should sort the `pairs` array in decreasing order of strength of victory, where strength of victory is defined to be the number of voters who prefer the preferred candidate. If multiple pairs have the same strength of victory, you may assume that the order does not matter.
- Complete the `lock_pairs` function.
  - The function should create the `locked` graph, adding all edges in decreasing order of victory strength so long as the edge would not create a cycle.

- Complete the `print_winner` function.
    - The function should print out the name of the candidate who is the source of the graph. You may assume there will not be more than one source.

You should not modify anything else in `tideman.c` other than the implementations of the `vote`, `record_preferences`, `add_pairs`, `sort_pairs`, `lock_pairs`, and `print_winner` functions (and the inclusion of additional header files, if you'd like). You are permitted to add additional functions to `tideman.c`, so long as you do not change the declarations of any of the existing functions.

Walkthrough



Usage

Your program should behave per the example below:

```
./tideman Alice Bob Charlie
Number of voters: 5
Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Alice
Rank 2: Charlie
Rank 3: Bob

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Bob
Rank 2: Charlie
Rank 3: Alice

Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Charlie
```

Testing

Be sure to test your code to make sure it handles

Be sure to test your code to make sure it handles…

- An election with any number of candidate (up to the `MAX` of `9`)
- Voting for a candidate by name
- Invalid votes for candidates who are not on the ballot
- Printing the winner of the election

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/tideman
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 tideman.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/tideman
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 4

What to Do

1. Submit one of:
   - this version of Filter if feeling less comfortable
   - this version of Filter if feeling more comfortable
2. Submit Recover

If you submit both versions of Filter, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 4's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `filter`, and

  ```
  make filter
  ```

  is yielding errors, try running

  ```
  help50 make filter
  ```

  instead!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Filter

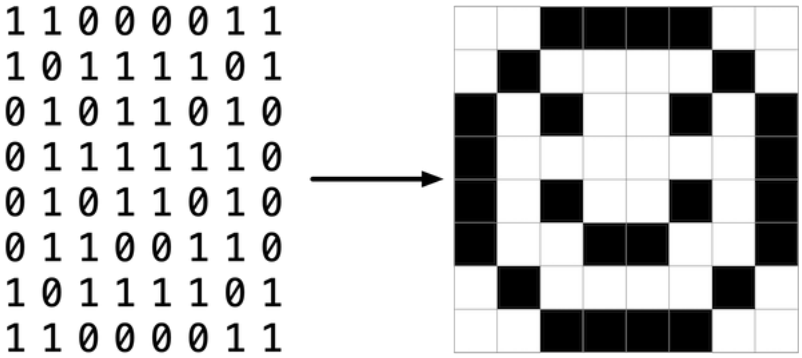
Implement a program that applies filters to BMPs, per the below.

```
$ ./filter -r image.bmp reflected.bmp
```

Background

Bitmaps

Perhaps the simplest way to represent an image is with a grid of pixels (i.e., dots), each of which can be of a different color. For black-and-white images, we thus need 1 bit per pixel, as 0 could represent black and 1 could represent white, as in the below.



In this sense, then, is an image just a bitmap (i.e., a map of bits). For more colorful images, you simply need more bits per pixel. A file format (like BMP (https://en.wikipedia.org/wiki/BMP_file_format), JPEG (https://en.wikipedia.org/wiki/JPEG), or PNG (https://en.wikipedia.org/wiki/Portable_Network_Graphics)) that supports "24-bit color" uses 24 bits per pixel. (BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.)

A 24-bit BMP uses 8 bits to signify the amount of red in a pixel's color, 8 bits to signify the amount of green in a pixel's color, and 8 bits to signify the amount of blue in a pixel's color. If you've ever heard of RGB color, well, there you have it: red, green, blue.

If the R, G, and B values of some pixel in a BMP are, say, `0xff`, `0x00`, and `0x00` in hexadecimal, that pixel is purely red, as `0xff` (otherwise known as `255` in decimal) implies "a lot of red," while `0x00` and `0x00` imply "no green" and "no blue," respectively.

A Bit(map) More Technical

Recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, (almost) every 24 of which happen to represent some pixel's color. But a BMP file also contains some "metadata," information like an image's height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as "headers," not to be confused with C's header files. (Incidentally, these headers have evolved over time. This problem uses the latest version of Microsoft's BMP format, 4.0, which debuted with Windows 95.)

The first of these headers, called `BITMAPFILEHEADER`, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called `BITMAPINFOHEADER`, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel's color. However, BMP stores these triples backwards (i.e., as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's

red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's BMPs as described herein, with each bitmap's top row first and bottom row last.) In other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would store this bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

```
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
```

Because we've presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, `ffffff` in hexadecimal actually signifies `111111111111111111111111` in binary.

Notice that you could represent a bitmap as a 2-dimensional array of pixels: where the image is an array of rows, each row is an array of pixels. Indeed, that's how we've chosen to represent bitmap images in this problem.

Image Filtering

What does it even mean to filter an image? You can think of filtering an image as taking the pixels of some original image, and modifying each pixel in such a way that a particular effect is apparent in the resulting image.

Grayscale

One common filter is the "grayscale" filter, where we take an image and want to convert it to black-and-white. How does that work?

Recall that if the red, green, and blue values are all set to `0x00` (hexadecimal for `0`), then the pixel is black. And if all values are set to `0xff` (hexadecimal for `255`), then the pixel is white. So long as the red, green, and blue values are all equal, the result will be varying shades of gray along the black-white spectrum, with higher values meaning lighter shades (closer to white) and lower values meaning darker shades (closer to black).

So to convert a pixel to grayscale, we just need to make sure the red, green, and blue values are all the same value. But how do we know what value to make them? Well, it's probably reasonable to expect that if the original red, green, and blue values were all pretty high, then the new value should also be pretty high. And if the original values were all low, then the new value should also be low.

In fact, to ensure each pixel of the new image still has the same general brightness or darkness as the old image, we can take the average of the red, green, and blue values to determine what shade of grey to make the new pixel.

If you apply that to each pixel in the image, the result will be an image converted to grayscale.

Sepia

Most image editing programs support a "sepia" filter, which gives images an old-timey feel by making the whole image look a bit reddish-brown.

An image can be converted to sepia by taking each pixel, and computing new red, green, and blue values based on the original values of the three.

There are a number of algorithms for converting an image to sepia, but for this problem, we'll ask you to use the following algorithm. For each pixel, the sepia color values should be calculated based on the original color values per the below.

```
sepiaRed = .393 * originalRed + .769 * originalGreen + .189 * originalBlue
sepiaGreen = .349 * originalRed + .686 * originalGreen + .168 * originalBlue
sepiaBlue = .272 * originalRed + .534 * originalGreen + .131 * originalBlue
```

Of course, the result of each of these formulas may not be an integer, but each value could be rounded to the nearest integer. It's also possible that the result of the formula is a number greater than 255, the maximum value for an 8-bit color value. In that case, the red, green, and blue values should be capped at 255. As a result, we can guarantee that the resulting red, green, and blue values will be whole numbers between 0 and 255, inclusive.

Reflection

Some filters might also move pixels around. Reflecting an image, for example, is a filter where the resulting image is what you would get by placing the original image in front of a mirror. So any pixels on the left side of the image should end up on the right, and vice versa.

Note that all of the original pixels of the original image will still be present in the reflected image, it's just that those pixels may have rearranged to be in a different place in the image.

Blur

There are a number of ways to create the effect of blurring or softening an image. For this problem, we'll use the "box blur," which works by taking each pixel and, for each color value, giving it a new value by averaging the color values of neighboring pixels.

Consider the following grid of pixels, where we've numbered each pixel.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The new value of each pixel would be the average of the values of all of the pixels that are within 1 row and column of the original pixel (forming a 3x3 box). For example, each of the color values for pixel 6 would be obtained by averaging the original color values of pixels 1, 2, 3, 5, 6, 7, 9, 10, and 11 (note that pixel 6 itself is included in the average). Likewise, the color values for pixel 11 would be be obtained by averaging the color values of pixels 6, 7, 8, 10, 11, 12, 14, 15 and 16.

For a pixel along the edge or corner, like pixel 15, we would still look for all pixels within 1 row and column: in this case, pixels 10, 11, 12, 14, 15, and 16.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.
- Execute `cd pset4` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/4/filter/less/filter.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip filter.zip` to uncompress that file.
- Execute `rm filter.zip` followed by `yes` or `y` to delete that ZIP file.

- Execute `ls`. You should see a directory called `filter`, which was inside of that ZIP file.
- Execute `cd filter` to change into that directory.
- Execute `ls`. You should see this problem's distribution, including `bmp.h`, `filter.c`, `helpers.h`, `helpers.c`, and `Makefile`. You'll also see a directory called `images`, with some sample Bitmap images.

Understanding

Let's now take a look at some of the files provided to you as distribution code to get an understanding for what's inside of them.

`bmp.h`

Open up `bmp.h` (as by double-clicking on it in the file browser) and have a look.

You'll see definitions of the headers we've mentioned (`BITMAPINFOHEADER` and `BITMAPFILEHEADER`). In addition, that file defines `BYTE`, `DWORD`, `LONG`, and `WORD`, data types normally found in the world of Windows programming. Notice how they're just aliases for primitives with which you are (hopefully) already familiar. It appears that `BITMAPFILEHEADER` and `BITMAPINFOHEADER` make use of these types.

Perhaps most importantly for you, this file also defines a `struct` called `RGBTRIPLE` that, quite simply, "encapsulates" three bytes: one blue, one green, and one red (the order, recall, in which we expect to find RGB triples actually on disk).

Why are these `struct`s useful? Well, recall that a file is just a sequence of bytes (or, ultimately, bits) on disk. But those bytes are generally ordered in such a way that the first few represent something, the next few represent something else, and so on. "File formats" exist because the world has standardized what bytes mean what. Now, we could just read a file from disk into RAM as one big array of bytes. And we could just remember that the byte at `array[i]` represents one thing, while the byte at `array[j]` represents another. But why not give some of those bytes names so that we can retrieve them from memory more easily? That's precisely what the structs in `bmp.h` allow us to do. Rather than think of some file as one long sequence of bytes, we can instead think of it as a sequence of `struct`s.

`filter.c`

Now, let's open up `filter.c`. This file has been written already for you, but there are a couple important points worth noting here.

First, notice the definition of `filters` on line 11. That string tells the program what the allowable command-line arguments to the program are: `b`, `g`, `r`, and `s`. Each of them specifies a different filter that we might apply to our images: blur, grayscale, reflection, and sepia.

The next several lines open up an image file, make sure it's indeed a BMP file, and read all of the pixel information into a 2D array called `image`.

Scroll down to the `switch` statement that begins on line 102. Notice that, depending on what `filter` we've chosen, a different function is called: if the user chooses filter `b`, the program calls the `blur` function; if `g`, then `grayscale` is called; if `r`, then `reflect` is called; and if `s`, then `sepia` is called. Notice, too, that each of these functions take as arguments the height of the image, the width of the image, and the 2D array of pixels.

These are the functions you'll (soon!) implement. As you might imagine, the goal is for each of these functions to edit the 2D array of pixels in such a way that the desired filter is applied to the image.

The remaining lines of the program take the resulting `image` and write them out to a new image file.

`helpers.h`

Next, take a look at `helpers.h`. This file is quite short, and just provides the function prototypes for the functions you saw earlier.

Here, take note of the fact that each function takes a 2D array called `image` as an argument, where `image` is an array of `height` many rows, and each row is itself another array of `width` many `RGBTRIPLE`s. So if `image` represents the whole picture, then `image[0]` represents the first row, and `image[0][0]` represents the pixel in the upper-left corner of the image.

`helpers.c`

Now, open up `helpers.c`. Here's where the implementation of the functions declared in `helpers.h` belong. But note that, right now, the implementations are missing! This part is up to you.

`Makefile`

Finally, let's look at `Makefile`. This file specifies what should happen when we run a terminal command like `make filter`. Whereas programs

```
$ ./filter -s infile.bmp outfile.bmp
```

```
$ ./filter -r infile.bmp outfile.bmp
```

```
$ ./filter -b infile.bmp outfile.bmp
```

Hints

- The values of a pixel's `rgbtRed`, `rgbtGreen`, and `rgbtBlue` components are all integers, so be sure to round any floating-point numbers to the nearest integer when assigning them to a pixel value!

Testing

Be sure to test all of your filters on the sample bitmap files provided!

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/filter/less
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 helpers.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/filter/less
```

Finally, let's look at `Makefile`. This file specifies what should happen when we run a terminal command like `make filter`. Whereas programs you may have written before were confined to just one file, `filter` seems to use multiple files: `filter.c`, `bmp.h`, `helpers.h`, and `helpers.c`. So we'll need to tell `make` how to compile this file.

Try compiling `filter` for yourself by going to your terminal and running

```
$ make filter
```

Then, you can run the program by running:

```
$ ./filter -g images/yard.bmp out.bmp
```

which takes the image at `images/yard.bmp`, and generates a new image called `out.bmp` after running the pixels through the `grayscale` function. `grayscale` doesn't do anything just yet, though, so the output image should look the same as the original yard.

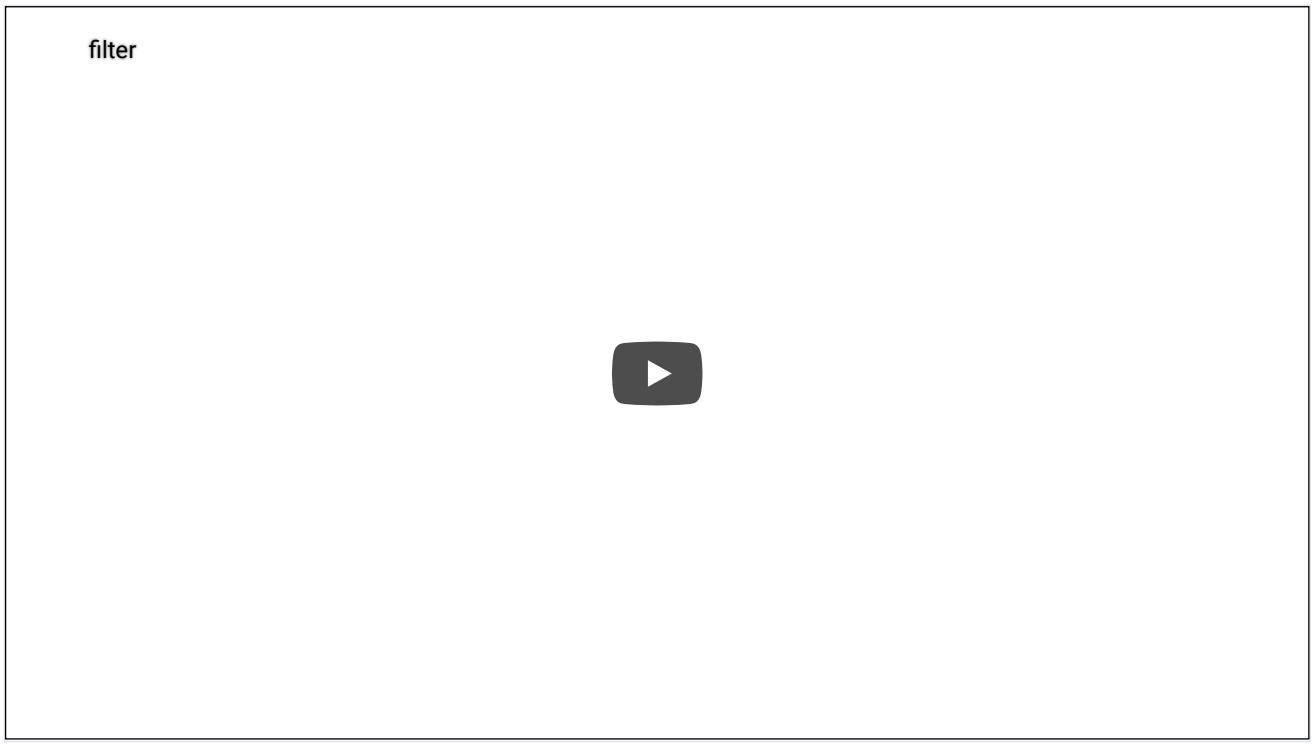
Specification

Implement the functions in `helpers.c` such that a user can apply grayscale, sepia, reflection, or blur filters to their images.

- The function `grayscale` should take an image and turn it into a black-and-white version of the same image.
- The function `sepia` should take an image and turn it into a sepia version of the same image.
- The `reflect` function should take an image and reflect it horizontally.
- Finally, the `blur` function should take an image and turn it into a box-blurred version of the same image.

You should not modify any of the function signatures, nor should you modify any other files other than `helpers.c`.

Walkthrough



Usage

Your program should behave per the examples below.

```
$ ./filter -g infile.bmp outfile.bmp
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Filter

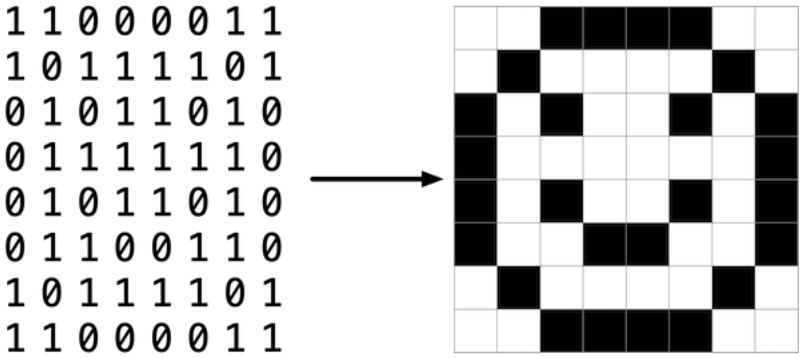
Implement a program that applies filters to BMPs, per the below.

```
$ ./filter -r image.bmp reflected.bmp
```

Background

Bitmaps

Perhaps the simplest way to represent an image is with a grid of pixels (i.e., dots), each of which can be of a different color. For black-and-white images, we thus need 1 bit per pixel, as 0 could represent black and 1 could represent white, as in the below.



In this sense, then, is an image just a bitmap (i.e., a map of bits). For more colorful images, you simply need more bits per pixel. A file format (like BMP (https://en.wikipedia.org/wiki/BMP_file_format), JPEG (https://en.wikipedia.org/wiki/JPEG), or PNG (https://en.wikipedia.org/wiki/Portable_Network_Graphics)) that supports "24-bit color" uses 24 bits per pixel. (BMP actually supports 1-, 4-, 8-, 16-, 24-, and 32-bit color.)

A 24-bit BMP uses 8 bits to signify the amount of red in a pixel's color, 8 bits to signify the amount of green in a pixel's color, and 8 bits to signify the amount of blue in a pixel's color. If you've ever heard of RGB color, well, there you have it: red, green, blue.

If the R, G, and B values of some pixel in a BMP are, say, `0xff`, `0x00`, and `0x00` in hexadecimal, that pixel is purely red, as `0xff` (otherwise known as `255` in decimal) implies "a lot of red," while `0x00` and `0x00` imply "no green" and "no blue," respectively.

A Bit(map) More Technical

Recall that a file is just a sequence of bits, arranged in some fashion. A 24-bit BMP file, then, is essentially just a sequence of bits, (almost) every 24 of which happen to represent some pixel's color. But a BMP file also contains some "metadata," information like an image's height and width. That metadata is stored at the beginning of the file in the form of two data structures generally referred to as "headers," not to be confused with C's header files. (Incidentally, these headers have evolved over time. This problem uses the latest version of Microsoft's BMP format, 4.0, which debuted with Windows 95.)

The first of these headers, called `BITMAPFILEHEADER`, is 14 bytes long. (Recall that 1 byte equals 8 bits.) The second of these headers, called `BITMAPINFOHEADER`, is 40 bytes long. Immediately following these headers is the actual bitmap: an array of bytes, triples of which represent a pixel's color. However, BMP stores these triples backwards (i.e., as BGR), with 8 bits for blue, followed by 8 bits for green, followed by 8 bits for red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's

red. (Some BMPs also store the entire bitmap backwards, with an image's top row at the end of the BMP file. But we've stored this problem set's BMPs as described herein, with each bitmap's top row first and bottom row last.) In other words, were we to convert the 1-bit smiley above to a 24-bit smiley, substituting red for black, a 24-bit BMP would store this bitmap as follows, where `0000ff` signifies red and `ffffff` signifies white; we've highlighted in red all instances of `0000ff`.

```
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff ffffff ffffff ffffff ffffff 0000ff
0000ff ffffff 0000ff ffffff ffffff 0000ff ffffff 0000ff
0000ff ffffff ffffff 0000ff 0000ff ffffff ffffff 0000ff
ffffff 0000ff ffffff ffffff ffffff ffffff 0000ff ffffff
ffffff ffffff 0000ff 0000ff 0000ff 0000ff ffffff ffffff
```

Because we've presented these bits from left to right, top to bottom, in 8 columns, you can actually see the red smiley if you take a step back.

To be clear, recall that a hexadecimal digit represents 4 bits. Accordingly, `ffffff` in hexadecimal actually signifies `111111111111111111111111` in binary.

Notice that you could represent a bitmap as a 2-dimensional array of pixels: where the image is an array of rows, each row is an array of pixels. Indeed, that's how we've chosen to represent bitmap images in this problem.

Image Filtering

What does it even mean to filter an image? You can think of filtering an image as taking the pixels of some original image, and modifying each pixel in such a way that a particular effect is apparent in the resulting image.

Grayscale

One common filter is the "grayscale" filter, where we take an image and want to convert it to black-and-white. How does that work?

Recall that if the red, green, and blue values are all set to `0x00` (hexadecimal for `0`), then the pixel is black. And if all values are set to `0xff` (hexadecimal for `255`), then the pixel is white. So long as the red, green, and blue values are all equal, the result will be varying shades of gray along the black-white spectrum, with higher values meaning lighter shades (closer to white) and lower values meaning darker shades (closer to black).

So to convert a pixel to grayscale, we just need to make sure the red, green, and blue values are all the same value. But how do we know what value to make them? Well, it's probably reasonable to expect that if the original red, green, and blue values were all pretty high, then the new value should also be pretty high. And if the original values were all low, then the new value should also be low.

In fact, to ensure each pixel of the new image still has the same general brightness or darkness as the old image, we can take the average of the red, green, and blue values to determine what shade of grey to make the new pixel.

If you apply that to each pixel in the image, the result will be an image converted to grayscale.

Reflection

Some filters might also move pixels around. Reflecting an image, for example, is a filter where the resulting image is what you would get by placing the original image in front of a mirror. So any pixels on the left side of the image should end up on the right, and vice versa.

Note that all of the original pixels of the original image will still be present in the reflected image, it's just that those pixels may have rearranged to be in a different place in the image.

Blur

There are a number of ways to create the effect of blurring or softening an image. For this problem, we'll use the "box blur," which works by taking each pixel and, for each color value, giving it a new value by averaging the color values of neighboring pixels.

Consider the following grid of pixels, where we've numbered each pixel.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

The new value of each pixel would be the average of the values of all of the pixels that are within 1 row and column of the original pixel (forming a 3x3 box). For example, each of the color values for pixel 6 would be obtained by averaging the original color values of pixels 1, 2, 3, 5, 6, 7, 9, 10, and 11 (note that pixel 6 itself is included in the average). Likewise, the color values for pixel 11 would be be obtained by averaging the color values of pixels 6, 7, 8, 10, 11, 12, 14, 15 and 16.

For a pixel along the edge or corner, like pixel 15, we would still look for all pixels within 1 row and column: in this case, pixels 10, 11, 12, 14, 15, and 16.

Edges

In artificial intelligence algorithms for image processing, it is often useful to detect edges in an image: lines in the image that create a boundary between one object and another. One way to achieve this effect is by applying the Sobel operator (https://en.wikipedia.org/wiki/Sobel_operator) to the image.

Like image blurring, edge detection also works by taking each pixel, and modifying it based on the 3x3 grid of pixels that surrounds that pixel. But instead of just taking the average of the nine pixels, the Sobel operator computes the new value of each pixel by taking a weighted sum of the values for the surrounding pixels. And since edges between objects could take place in both a vertical and a horizontal direction, you'll actually compute two weighted sums: one for detecting edges in the x direction, and one for detecting edges in the y direction. In particular, you'll use the following two "kernels":

Gx

−1	0	1
−2	0	2
−1	0	1

Gy

−1	−2	−1
0	0	0
1	2	1

How to interpret these kernels? In short, for each of the three color values for each pixel, we'll compute two values `Gx` and `Gy`. To compute `Gx` for the red channel value of a pixel, for instance, we'll take the original red values for the nine pixels that form a 3x3 box around the pixel, multiply them each by the corresponding value in the `Gx` kernel, and take the sum of the resulting values.

Why these particular values for the kernel? In the `Gx` direction, for instance, we're multiplying the pixels to the right of the target pixel by a positive number, and multiplying the pixels to the left of the target pixel by a negative number. When we take the sum, if the pixels on the right are a similar color to the pixels on the left, the result will be close to 0 (the numbers cancel out). But if the pixels on the right are very different from the pixels on the left, then the resulting value will be very positive or very negative, indicating a change in color that likely is the result of a boundary between objects. And a similar argument holds true for calculating edges in the `y` direction.

Using these kernels, we can generate a `Gx` and `Gy` value for each of the red, green, and blue channels for a pixel. But each channel can only take on one value, not two: so we need some way to combine `Gx` and `Gy` into a single value. The Sobel filter algorithm combines `Gx` and `Gy` into a final value by calculating the square root of `Gx^2 + Gy^2`. And since channel values can only take on integer values from 0 to 255, be sure the resulting value is rounded to the nearest integer and capped at 255!

And what about handling pixels at the edge, or in the corner of the image? There are many ways to handle pixels at the edge, but for the purposes of this problem, we'll ask you to treat the image as if there was a 1 pixel solid black border around the edge of the image: therefore, trying to access a pixel past the edge of the image should be treated as a solid black pixel (values of 0 for each of red, green, and blue). This will effectively ignore those pixels from our calculations of `Gx` and `Gy`.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
- Execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.
- Execute `cd pset4` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/4/filter/more/filter.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip filter.zip` to uncompress that file.
- Execute `rm filter.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `filter`, which was inside of that ZIP file.
- Execute `cd filter` to change into that directory.
- Execute `ls`. You should see this problem's distribution, including `bmp.h`, `filter.c`, `helpers.h`, `helpers.c`, and `Makefile`. You'll also see a directory called `images`, with some sample Bitmap images.

Understanding

Let's now take a look at some of the files provided to you as distribution code to get an understanding for what's inside of them.

`bmp.h`

Open up `bmp.h` (as by double-clicking on it in the file browser) and have a look.

You'll see definitions of the headers we've mentioned (`BITMAPINFOHEADER` and `BITMAPFILEHEADER`). In addition, that file defines `BYTE`, `DWORD`, `LONG`, and `WORD`, data types normally found in the world of Windows programming. Notice how they're just aliases for primitives with which you are (hopefully) already familiar. It appears that `BITMAPFILEHEADER` and `BITMAPINFOHEADER` make use of these types.

Perhaps most importantly for you, this file also defines a `struct` called `RGBTRIPLE` that, quite simply, "encapsulates" three bytes: one blue, one green, and one red (the order, recall, in which we expect to find RGB triples actually on disk).

Why are these `struct` s useful? Well, recall that a file is just a sequence of bytes (or, ultimately, bits) on disk. But those bytes are generally ordered in such a way that the first few represent something, the next few represent something else, and so on. "File formats" exist because the world has standardized what bytes mean what. Now, we could just read a file from disk into RAM as one big array of bytes. And we could just remember that the byte at `array[i]` represents one thing, while the byte at `array[j]` represents another. But why not give some of those bytes names so that we can retrieve them from memory more easily? That's precisely what the structs in `bmp.h` allow us to do. Rather than think of some file as one long sequence of bytes, we can instead think of it as a sequence of `struct` s.

`filter.c`

Now, let's open up `filter.c`. This file has been written already for you, but there are a couple important points worth noting here.

First, notice the definition of `filters` on line 11. That string tells the program what the allowable command-line arguments to the program are: `b`, `e`, `g`, and `r`. Each of them specifies a different filter that we might apply to our images: blur, edge detection, grayscale, and reflection.

The next several lines open up an image file, make sure it's indeed a BMP file, and read all of the pixel information into a 2D array called `image`.

Scroll down to the `switch` statement that begins on line 102. Notice that, depending on what `filter` we've chosen, a different function is called: if the user chooses filter `b`, the program calls the `blur` function; if `e`, then `edges` is called; if `g`, then `grayscale` is called; and if `r`, then `reflect` is called. Notice, too, that each of these functions take as arguments the height of the image, the width of the image, and the 2D array of pixels.

These are the functions you'll (soon!) implement. As you might imagine, the goal is for each of these functions to edit the 2D array of pixels in such a way that the desired filter is applied to the image.

The remaining lines of the program take the resulting `image` and write them out to a new image file.

`helpers.h`

Next, take a look at `helpers.h`. This file is quite short, and just provides the function prototypes for the functions you saw earlier.

Here, take note of the fact that each function takes a 2D array called `image` as an argument, where `image` is an array of `height` many rows, and each row is itself another array of `width` many `RGBTRIPLE`s. So if `image` represents the whole picture, then `image[0]` represents the first row, and `image[0][0]` represents the pixel in the upper-left corner of the image.

`helpers.c`

Now, open up `helpers.c`. Here's where the implementation of the functions declared in `helpers.h` belong. But note that, right now, the implementations are missing! This part is up to you.

`Makefile`

Finally, let's look at `Makefile`. This file specifies what should happen when we run a terminal command like `make filter`. Whereas programs you may have written before were confined to just one file, `filter` seems to use multiple files: `filter.c`, `bmp.h`, `helpers.h`, and `helpers.c`. So we'll need to tell `make` how to compile this file.

Try compiling `filter` for yourself by going to your terminal and running

```
$ make filter
```

Then, you can run the program by running:

```
$ ./filter -g images/yard.bmp out.bmp
```

which takes the image at `images/yard.bmp`, and generates a new image called `out.bmp` after running the pixels through the `grayscale` function. `grayscale` doesn't do anything just yet, though, so the output image should look the same as the original yard.

Specification

Implement the functions in `helpers.c` such that a user can apply grayscale, reflection, blur, or edge detection filters to their images.

- The function `grayscale` should take an image and turn it into a black-and-white version of the same image.
- The `reflect` function should take an image and reflect it horizontally.
- The `blur` function should take an image and turn it into a box-blurred version of the same image.
- The `edges` function should take an image and highlight the edges between objects, according to the Sobel operator.

You should not modify any of the function signatures, nor should you modify any other files other than `helpers.c`.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (*) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/filter/more
```

Walkthrough

Usage

Your program should behave per the examples below.

```
$ ./filter -g infile.bmp outfile.bmp
```

```
$ ./filter -r infile.bmp outfile.bmp
```

```
$ ./filter -b infile.bmp outfile.bmp
```

```
$ ./filter -e infile.bmp outfile.bmp
```

Hints

- The values of a pixel's `rgbtRed`, `rgbtGreen`, and `rgbtBlue` components are all integers, so be sure to round any floating-point numbers to the nearest integer when assigning them to a pixel value!

Testing

Be sure to test all of your filters on the sample bitmap files provided!

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/filter/more
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 helpers.c
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Recover

Implement a program that recovers JPEGs from a forensic image, per the below.

```
$ ./recover card.raw
```

Background

In anticipation of this problem, we spent the past several days taking photos of people we know, all of which were saved on a digital camera as JPEGs on a memory card. (Okay, it's possible we actually spent the past several days on Facebook instead.) Unfortunately, we somehow deleted them all! Thankfully, in the computer world, "deleted" tends not to mean "deleted" so much as "forgotten." Even though the camera insists that the card is now blank, we're pretty sure that's not quite true. Indeed, we're hoping (er, expecting!) you can write a program that recovers the photos for us!

Even though JPEGs are more complicated than BMPs, JPEGs have "signatures," patterns of bytes that can distinguish them from other file formats. Specifically, the first three bytes of JPEGs are

```
0xff 0xd8 0xff
```

from first byte to third byte, left to right. The fourth byte, meanwhile, is either `0xe0`, `0xe1`, `0xe2`, `0xe3`, `0xe4`, `0xe5`, `0xe6`, `0xe7`, `0xe8`, `0xe9`, `0xea`, `0xeb`, `0xec`, `0xed`, `0xee`, or `0xef`. Put another way, the fourth byte's first four bits are `1110`.

Odds are, if you find this pattern of four bytes on media known to store photos (e.g., my memory card), they demarcate the start of a JPEG. To be fair, you might encounter these patterns on some disk purely by chance, so data recovery isn't an exact science.

Fortunately, digital cameras tend to store photographs contiguously on memory cards, whereby each photo is stored immediately after the previously taken photo. Accordingly, the start of a JPEG usually demarks the end of another. However, digital cameras often initialize cards with a FAT file system whose "block size" is 512 bytes (B). The implication is that these cameras only write to those cards in units of 512 B. A photo that's 1 MB (i.e., 1,048,576 B) thus takes up 1048576 ÷ 512 = 2048 "blocks" on a memory card. But so does a photo that's, say, one byte smaller (i.e., 1,048,575 B)! The wasted space on disk is called "slack space." Forensic investigators often look at slack space for remnants of suspicious data.

The implication of all these details is that you, the investigator, can probably write a program that iterates over a copy of my memory card, looking for JPEGs' signatures. Each time you find a signature, you can open a new file for writing and start filling that file with bytes from my memory card, closing that file only once you encounter another signature. Moreover, rather than read my memory card's bytes one at a time, you can read 512 of them at a time into a buffer for efficiency's sake. Thanks to FAT, you can trust that JPEGs' signatures will be "block-aligned." That is, you need only look for those signatures in a block's first four bytes.

Realize, of course, that JPEGs can span contiguous blocks. Otherwise, no JPEG could be larger than 512 B. But the last byte of a JPEG might not fall at the very end of a block. Recall the possibility of slack space. But not to worry. Because this memory card was brand-new when I started snapping photos, odds are it'd been "zeroed" (i.e., filled with 0s) by the manufacturer, in which case any slack space will be filled with 0s. It's okay if those trailing 0s end up in the JPEGs you recover; they should still be viewable.

Now, I only have one memory card, but there are a lot of you! And so I've gone ahead and created a "forensic image" of the card, storing its contents, byte after byte, in a file called `card.raw`. So that you don't waste time iterating over millions of 0s unnecessarily, I've only imaged the first few megabytes of the memory card. But you should ultimately find that the image contains 50 JPEGs.

Getting Started

Here's how to download this problem's "distribution code" (i.e., starter code) into your own CS50 IDE. Log into CS50 IDE (https://cs50.io/) and then, in a terminal window, execute each of the below.

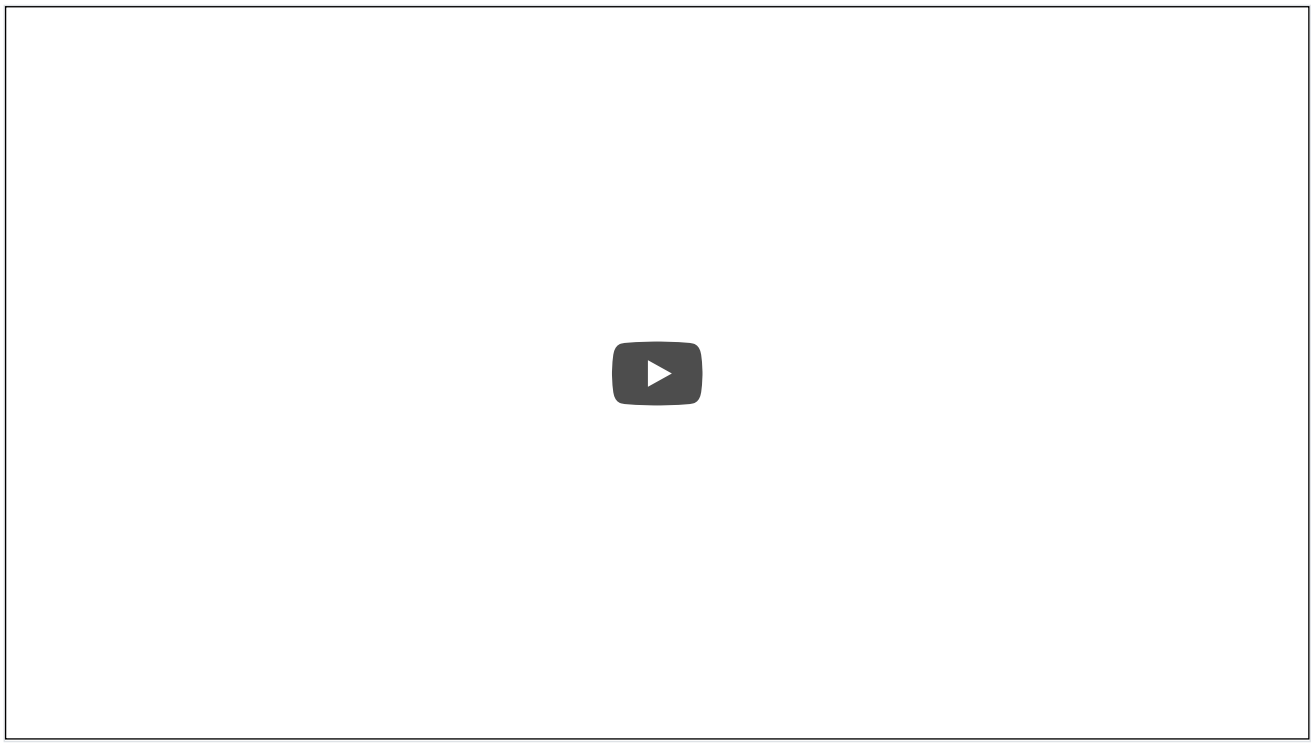
1. Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
2. If you haven't already, execute `mkdir pset4` to make (i.e., create) a directory called `pset4` in your home directory.
3. Execute `cd pset4` to change into (i.e., open) your `pset4` directory.
4. Execute `wget https://cdn.cs50.net/2019/fall/psets/4/recover/recover.zip` to download a (compressed) ZIP file with this problem's distribution.
5. Execute `unzip recover.zip` to uncompress that file.
6. Execute `rm recover.zip` followed by `yes` or `y` to delete that ZIP file.
7. Execute `ls` . You should see a directory called `recover` , which was inside of that ZIP file.
8. Execute `cd recover` to change into that directory.
9. Execute `ls` . You should see this problem's distribution, including `card.raw` and `recover.c` .

Specification

Implement a program called `recover` that recovers JPEGs from a forensic image.

- Implement your program in a file called `recover.c` in a directory called `recover` .
- Your program should accept exactly one command-line argument, the name of a forensic image from which to recover JPEGs.
- If your program is not executed with exactly one command-line argument, it should remind the user of correct usage, and `main` should return `1` .
- If the forensic image cannot be opened for reading, your program should inform the user as much, and `main` should return `1` .
- Your program, if it uses `malloc` , must not leak any memory.

Walkthrough



Usage

Your program should behave per the examples below.

```
$ ./recover
Usage: ./recover image
```

```
$ ./recover card.raw
```

Hints

Keep in mind that you can open `card.raw` programmatically with `fopen`, as with the below, provided `argv[1]` exists.

```
FILE *file = fopen(argv[1], "r");
```

When executed, your program should recover every one of the JPEGs from `card.raw`, storing each as a separate file in your current working directory. Your program should number the files it outputs by naming each `###.jpg`, where `###` is three-digit decimal number from `000` on up. (Befriend `sprintf` (https://man.cs50.io/3/sprintf).) You need not try to recover the JPEGs' original names. To check whether the JPEGs your program spit out are correct, simply double-click and take a look! If each photo appears intact, your operation was likely a success!

Odds are, though, the JPEGs that the first draft of your code spits out won't be correct. (If you open them up and don't see anything, they're probably not correct!) Execute the command below to delete all JPEGs in your current working directory.

```
$ rm *.jpg
```

If you'd rather not be prompted to confirm each deletion, execute the command below instead.

```
$ rm -f *.jpg
```

Just be careful with that `-f` switch, as it "forces" deletion without prompting you.

If you'd like to create a new type to store a byte of data, you can do so via the below, which defines a new type called `BYTE` to be a `uint8_t` (a type defined in `stdint.h`, representing an 8-bit unsigned integer).

```
typedef uint8_t BYTE;
```

Keep in mind, too, that you can read data from a file using `fread` (https://man.cs50.io/3/fread), which will read data from a file into a location in memory and return the number of items successfully read from the file.

Testing

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2020/x/recover
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 recover.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/recover
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Speller

Be sure to read this specification in its entirety before starting so you know what to do and how to do it!

Implement a program that spell-checks a file, a la the below, using a hash table.

```
$ ./speller texts/lalaland.txt
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

Distribution

Downloading

Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

1. Execute `cd` to ensure that you're in `~/` (i.e., your home directory).
2. Execute `mkdir pset5` to make (i.e., create) a directory called `pset5` in your home directory.
3. Execute `cd pset5` to change into (i.e., open) that directory.
4. Execute `wget https://cdn.cs50.net/2019/fall/psets/5/speller/speller.zip` to download a (compressed) ZIP file with this problem's distribution.
5. Execute `unzip speller.zip` to uncompress that file.
6. Execute `rm speller.zip` followed by `yes` or `y` to delete that ZIP file.
7. Execute `ls`. You should see a directory called `speller`, which was inside of that ZIP file.
8. Execute `cd speller` to change into that directory.
9. Execute `ls`. You should see this problem's distribution:

```
dictionaries/  dictionary.c  dictionary.h  keys/  Makefile  speller.c  texts/
```

Understanding

Theoretically, on input of size *n*, an algorithm with a running time of *n* is "asymptotically equivalent," in terms of *O*, to an algorithm with a running time of *2n*. Indeed, when describing the running time of an algorithm, we typically focus on the dominant (i.e., most impactful) term (i.e., *n* in this case, since *n* could be much larger than 2). In the real world, though, the fact of the matter is that *2n* feels twice as slow as *n*.

The challenge ahead of you is to implement the fastest spell checker you can! By "fastest," though, we're talking actual "wall-clock," not asymptotic, time.

In `speller.c`, we've put together a program that's designed to spell-check a file after loading a dictionary of words from disk into memory. That dictionary, meanwhile, is implemented in a file called `dictionary.c`. (It could just be implemented in `speller.c`, but as programs get more complex, it's often convenient to break them into multiple files.) The prototypes for the functions therein, meanwhile, are defined not in `dictionary.c` itself but in `dictionary.h` instead. That way, both `speller.c` and `dictionary.c` can `#include` the file. Unfortunately, we didn't quite get around to implementing the loading part. Or the checking part. Both (and a bit more) we leave to you! But first, a tour.

`dictionary.h`

Open up `dictionary.h`, and you'll see some new syntax, including a few lines that mention `DICTIONARY_H`. No need to worry about those, but, if curious, those lines just ensure that, even though `dictionary.c` and `speller.c` (which you'll see in a moment) `#include` this file, `clang` will only compile it once.

Next notice how we `#include` a file called `stdbool.h`. That's the file in which `bool` itself is defined. You've not needed it before, since the CS50 Library used to `#include` that for you.

Also notice our use of `#define`, a "preprocessor directive" that defines a "constant" called `LENGTH` that has a value of `45`. It's a constant in the sense that you can't (accidentally) change it in your own code. In fact, `clang` will replace any mentions of `LENGTH` in your own code with, literally, `45`. In other words, it's not a variable, just a find-and-replace trick.

Finally, notice the prototypes for five functions: `check`, `hash`, `load`, `size`, and `unload`. Notice how three of those take a pointer as an argument, per the `*`:

```
bool check(const char *word);
unsigned int hash(const char *word);
bool load(const char *dictionary);
```

Recall that `char *` is what we used to call `string`. So those three prototypes are essentially just:

```
bool check(const string word);
unsigned int hash(const string word);
bool load(const string dictionary);
```

And `const`, meanwhile, just says that those strings, when passed in as arguments, must remain constant; you won't be able to change them, accidentally or otherwise!

`dictionary.c`

Now open up `dictionary.c`. Notice how, atop the file, we've defined a `struct` called `node` that represents a node in a hash table. And we've declared a global pointer array, `table`, which will (soon) represent the hash table you will use to keep track of words in the dictionary. The array contains `N` node pointers, and we've set `N` equal to `1` for now, meaning this hash table has just 1 bucket right now. You'll likely want to increase the number of buckets, as by changing `N`, to something larger!

Next, notice that we've implemented `load`, `hash`, `check`, `size`, and `unload`, but only barely, just enough for the code to compile. Your job, ultimately, is to re-implement those functions as cleverly as possible so that this spell checker works as advertised. And fast!

`speller.c`

Okay, next open up `speller.c` and spend some time looking over the code and comments therein. You won't need to change anything in this file, and you don't need to understand its entirety, but do try to get a sense of its functionality nonetheless. Notice how, by way of a function called `getrusage`, we'll be "benchmarking" (i.e., timing the execution of) your implementations of `check`, `load`, `size`, and `unload`. Also notice how we go about passing `check`, word by word, the contents of some file to be spell-checked. Ultimately, we report each misspelling in that file along with a bunch of statistics.

Notice, incidentally, that we have defined the usage of `speller` to be

```
Usage: speller [dictionary] text
```

where `dictionary` is assumed to be a file containing a list of lowercase words, one per line, and `text` is a file to be spell-checked. As the

brackets suggest, provision of `dictionary` is optional; if this argument is omitted, `speller` will use `dictionaries/large` by default. In other words, running

```
$ ./speller text
```

will be equivalent to running

```
$ ./speller dictionaries/large text
```

where `text` is the file you wish to spell-check. Suffice it to say, the former is easier to type! (Of course, `speller` will not be able to load any dictionaries until you implement `load` in `dictionary.c` ! Until then, you'll see `Could not load` .)

Within the default dictionary, mind you, are 143,091 words, all of which must be loaded into memory! In fact, take a peek at that file to get a sense of its structure and size. Notice that every word in that file appears in lowercase (even, for simplicity, proper nouns and acronyms). From top to bottom, the file is sorted lexicographically, with only one word per line (each of which ends with `\n`). No word is longer than 45 characters, and no word appears more than once. During development, you may find it helpful to provide `speller` with a `dictionary` of your own that contains far fewer words, lest you struggle to debug an otherwise enormous structure in memory. In `dictionaries/small` is one such dictionary. To use it, execute

```
$ ./speller dictionaries/small text
```

where `text` is the file you wish to spell-check. Don't move on until you're sure you understand how `speller` itself works!

Odds are, you didn't spend enough time looking over `speller.c` . Go back one square and walk yourself through it again!

`texts/`

So that you can test your implementation of `speller` , we've also provided you with a whole bunch of texts, among them the script from *La La Land*, the text of the Affordable Care Act, three million bytes from Tolstoy, some excerpts from *The Federalist Papers* and Shakespeare, the entirety of the King James V Bible and the Koran, and more. So that you know what to expect, open and skim each of those files, all of which are in a directory called `texts` within your `pset5` directory.

Now, as you should know from having read over `speller.c` carefully, the output of `speller` , if executed with, say,

```
$ ./speller texts/lalaland.txt
```

will eventually resemble the below. For now, try the staff's solution (using the default dictionary) by executing

```
$ ~cs50/2019/fall/pset5/speller texts/lalaland.txt
```

Below's some of the output you'll see. For information's sake, we've excerpted some examples of "misspellings." And lest we spoil the fun, we've omitted our own statistics for now.

```
MISSPELLED WORDS

[...]
AHHHHHHHHHHHHHHHHHHHHHHHHHHHHT
[...]
Shangri
[...]
fianc
[...]
Sebastian's
[...]

WORDS MISSPELLED:
WORDS IN DICTIONARY:
WORDS IN TEXT:
TIME IN load:
TIME IN check:
TIME IN size:
TIME IN unload:
TIME IN TOTAL:
```

`TIME IN load` represents the number of seconds that `speller` spends executing your implementation of `load`. `TIME IN check` represents the number of seconds that `speller` spends, in total, executing your implementation of `check`. `TIME IN size` represents the number of seconds that `speller` spends executing your implementation of `size`. `TIME IN unload` represents the number of seconds that `speller` spends executing your implementation of `unload`. `TIME IN TOTAL` is the sum of those four measurements.

Note that these times may vary somewhat across executions of `speller`, depending on what else CS50 IDE is doing, even if you don't change your code.

Incidentally, to be clear, by "misspelled" we simply mean that some word is not in the `dictionary` provided.

`Makefile`

And, lastly, recall that `make` automates compilation of your code so that you don't have to execute `clang` manually along with a whole bunch of switches. However, as your programs grow in size, `make` won't be able to infer from context anymore how to compile your code; you'll need to start telling `make` how to compile your program, particularly when they involve multiple source (i.e., `.c`) files, as in the case of this problem. And so we'll utilize a `Makefile`, a configuration file that tells `make` exactly what to do. Open up `Makefile`, and you should see four lines:

1. The first line tells `make` to execute the subsequent lines whenever you yourself execute `make speller` (or just `make`).
2. The second line tells `make` how to compile `speller.c` into machine code (i.e., `speller.o`).
3. The third line tells `make` how to compile `dictionary.c` into machine code (i.e., `dictionary.o`).
4. The fourth line tells `make` to link `speller.o` and `dictionary.o` in a file called `speller`.

Be sure to compile `speller` by executing `make speller` (or just `make`). Executing `make dictionary` won't work!

Specification

Alright, the challenge now before you is to implement, in order, `load`, `hash`, `size`, `check`, and `unload` as efficiently as possible using a hash table in such a way that `TIME IN load`, `TIME IN check`, `TIME IN size`, and `TIME IN unload` are all minimized. To be sure, it's not obvious what it even means to be minimized, inasmuch as these benchmarks will certainly vary as you feed `speller` different values for `dictionary` and for `text`. But therein lies the challenge, if not the fun, of this problem. This problem is your chance to design. Although we invite you to minimize space, your ultimate enemy is time. But before you dive in, some specifications from us.

- You may not alter `speller.c` or `Makefile`.
- You may alter `dictionary.c` (and, in fact, must in order to complete the implementations of `load`, `hash`, `size`, `check`, and `unload`), but you may not alter the declarations (i.e., prototypes) of `load`, `hash`, `size`, `check`, or `unload`. You may, though, add new functions and (local or global) variables to `dictionary.c`.
- You may change the value of `N` in `dictionary.c`, so that your hash table can have more buckets.
- You may alter `dictionary.h`, but you may not alter the declarations of `load`, `hash`, `size`, `check`, or `unload`.
- Your implementation of `check` must be case-insensitive. In other words, if `foo` is in dictionary, then `check` should return true given any capitalization thereof; none of `foo`, `foO`, `fOo`, `fOO`, `fOO`, `Foo`, `FoO`, `FOo`, and `FOO` should be considered misspelled.
- Capitalization aside, your implementation of `check` should only return `true` for words actually in `dictionary`. Beware hard-coding common words (e.g., `the`), lest we pass your implementation a `dictionary` without those same words. Moreover, the only possessives allowed are those actually in `dictionary`. In other words, even if `foo` is in `dictionary`, `check` should return `false` given `foo's` if `foo's` is not also in `dictionary`.
- You may assume that any `dictionary` passed to your program will be structured exactly like ours, alphabetically sorted from top to bottom with one word per line, each of which ends with `\n`. You may also assume that `dictionary` will contain at least one word, that no word will be longer than `LENGTH` (a constant defined in `dictionary.h`) characters, that no word will appear more than once, that each word will contain only lowercase alphabetical characters and possibly apostrophes, and that no word will start with an apostrophe.
- You may assume that `check` will only be passed words that contain (uppercase or lowercase) alphabetical characters and possibly apostrophes.
- Your spell checker may only take `text` and, optionally, `dictionary` as input. Although you might be inclined (particularly if among those more comfortable) to "pre-process" our default dictionary in order to derive an "ideal hash function" for it, you may not save the output of any such pre-processing to disk in order to load it back into memory on subsequent runs of your spell checker in order to gain an advantage.
- Your spell checker must not leak any memory. Be sure to check for leaks with `valgrind`.
- You may search for (good) hash functions online, so long as you cite the origin of any hash function you integrate into your own code.

Alright, ready to go?

- Implement `load`.
- Implement `hash`.
- Implement `size`.
- Implement `check`.
- Implement `unload`.

Walkthroughs

Please note that there are 6 videos in this playlist.

Hints

To compare two strings case-insensitively, you may find `strcasecmp` (https://man.cs50.io/3/strcasecmp) (declared in `strings.h`) useful!

Ultimately, be sure to `free` in `unload` any memory that you allocated in `load`! Recall that `valgrind` is your newest best friend. Know that `valgrind` watches for leaks while your program is actually running, so be sure to provide command-line arguments if you want `valgrind` to analyze `speller` while you use a particular `dictionary` and/or text, as in the below. Best to use a small text, though, else `valgrind` could take quite a while to run.

```
$ valgrind ./speller texts/cat.txt
```

If you run `valgrind` without specifying a `text` for `speller`, your implementations of `load` and `unload` won't actually get called (and thus analyzed).

If unsure how to interpret the output of `valgrind`, do just ask `help50` for help:

```
$ help50 valgrind ./speller texts/cat.txt
```

Testing

How to check whether your program is outting the right misspelled words? Well, you're welcome to consult the "answer keys" that are inside of the `keys` directory that's inside of your `speller` directory. For instance, inside of `keys/lalaland.txt` are all of the words that your program *should* think are misspelled.

You could therefore run your program on some text in one window, as with the below.

```
$ ./speller texts/lalaland.txt
```

And you could then run the staff's solution on the same text in another window, as with the below.

```
$ ~cs50/2019/fall/pset5/speller texts/lalaland.txt
```

And you could then compare the windows visually side by side. That could get tedious quickly, though. So you might instead want to "redirect" your program's output to a file, as with the below.

```
$ ./speller texts/lalaland.txt > student.txt
$ ~cs50/2019/fall/pset5/speller texts/lalaland.txt > staff.txt
```

You can then compare both files side by side in the same window with a program like `diff`, as with the below.

```
$ diff -y student.txt staff.txt
```

Alternatively, to save time, you could just compare your program's output (assuming you redirected it to, e.g., `student.txt`) against one of the answer keys without running the staff's solution, as with the below.

```
$ diff -y student.txt keys/lalaland.txt
```

If your program's output matches the staff's, `diff` will output two columns that should be identical except for, perhaps, the running times at the bottom. If the columns differ, though, you'll see a `>` or `|` where they differ. For instance, if you see

```
MISSPELLED WORDS                                        MISSPELLED WORDS

TECHNO                                                  TECHNO
L                                                       L
                                                      > Thelonious
Prius                                                   Prius
                                                      > MIA
L                                                       L
```

that means your program (whose output is on the left) does not think that `Thelonious` or `MIA` is misspelled, even though the staff's output (on the right) does, as is implied by the absence of, say, `Thelonious` in the lefthand column and the presence of `Thelonious` in the righthand column.

`check50`

To test your code less manually (though still not exhaustively), you may also execute the below.

```
$ check50 cs50/problems/2020/x/speller
```

Note that `check50` will also check for memory leaks, so be sure you've run `valgrind` as well.

Staff's Solution

How to assess just how fast (and correct) your code is? Well, as always, feel free to play with the staff's solution, as with the below, and compare its numbers against yours.

```
$ ~cs50/2019/fall/pset5/speller texts/lalaland.txt
```

Big Board

And if you'd like to put your code to the test against classmates' code (just for fun), execute the command below to challenge the Big Board before or after you submit.

▶ **Submit to Big Board**

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (*) instead of the actual characters in your password.

```
$ submit50 cs50/problems/2020/x/speller
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 5

What to Do

1. Submit Speller

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 5's source code.
- If you see any errors when compiling your code with `make`, focus first on fixing the very first error you see, scrolling up as needed. If unsure what it means, try asking `help50` for help. For instance, if trying to compile `hello`, and

  ```
  make speller
  ```

  is yielding errors, try running

  ```
  help50 make speller
  ```

  instead!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 6

What to Do

1. Submit Hello in Python
2. Submit one of:
    - this version of Mario in Python, if feeling less comfortable
    - this version of Mario in Python, if feeling more comfortable
3. Submit one of:
    - Cash in Python, if feeling less comfortable
    - Credit in Python, if feeling more comfortable
4. Submit Readability in Python
5. Submit DNA in Python

If you submit both versions of Mario, we'll record the higher of your two scores. If you submit both Cash and Credit, we'll record the higher of your two scores.

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Try out any of David's programs from class via Week 6's source code.

Academic Honesty

- For Hello, Mario, Cash, Credit, and Readability, it is **reasonable** to look at your own implementations thereof in C and others' implementations thereof *in C*, including the staff's implementations thereof in C.
- It is **not reasonable** to look at others' implementations of the same *in Python*.
- Insofar as a goal of these problems is to teach you how to teach yourself a new language, keep in mind that these acts are not only **reasonable**, per the syllabus, but encouraged toward that end:
    - Incorporating a few lines of code that you find online or elsewhere into your own code, provided that those lines are not themselves solutions to assigned problems and that you cite the lines' origins.
    - Turning to the web or elsewhere for instruction beyond the course's own, for references, and for solutions to technical difficulties, but not for outright solutions to problem set's problems or your own final project.

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Hello

Implement a program that prints out a simple greeting to the user, per the below.

```
$ python hello.py
What is your name?
David
hello, David
```

Specification

Write, in a file called `hello.py` in `~/pset6/hello`, a program that prompts a user for their name, and then prints `hello, so-and-so`, where `so-and-so` is their provided name, exactly as you did in Problem Set 1, except that your program this time should be written (a) in Python and (b) in CS50 IDE.

Usage

Your program should behave per the example below.

```
$ python hello.py
What is your name?
Emma
hello, Emma
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python hello.py`, and wait for a prompt for input. Type in `Emma` and press enter. Your program should output `hello, Emma`.
- Run your program as `python hello.py`, and wait for a prompt for input. Type in `Rodrigo` and press enter. Your program should output `hello, Rodrigo`.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (*) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/hello
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

Implement a program that prints out a half-pyramid of a specified height, per the below.

```
$ ./mario
Height: 4
   #
  ##
 ###
####
```

Specification

- Write, in a file called `mario.py` in `~/pset6/mario/less/`, a program that recreates the half-pyramid using hashes (`#`) for blocks, exactly as you did in Problem Set 1, except that your program this time should be written (a) in Python and (b) in CS50 IDE.
- To make things more interesting, first prompt the user with `get_int` for the half-pyramid's height, a positive integer between `1` and `8`, inclusive.
- If the user fails to provide a positive integer no greater than `8`, you should re-prompt for the same again.
- Then, generate (with the help of `print` and one or more loops) the desired half-pyramid.
- Take care to align the bottom-left corner of your half-pyramid with the left-hand edge of your terminal window.

Usage

Your program should behave per the example below.

```
$ ./mario
Height: 4
   #
  ##
 ###
####
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python mario.py` and wait for a prompt for input. Type in `-1` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Type in `0` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Type in `1` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
#
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `2` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
 #
##
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `8` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
       #
      ##
     ###
    ####
   #####
  ######
 #######
########
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `9` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number. Then, type in `2` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
 #
##
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `foo` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Do not type anything, and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/mario/less
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Mario

Implement a program that prints out a double half-pyramid of a specified height, per the below.

```
$ ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

Specification

- Write, in a file called `mario.py` in `~/pset6/mario/more/`, a program that recreates these half-pyramids using hashes (`#`) for blocks, exactly as you did in Problem Set 1, except that your program this time should be written (a) in Python and (b) in CS50 IDE.
- To make things more interesting, first prompt the user with `get_int` for the half-pyramid's height, a positive integer between `1` and `8`, inclusive. (The height of the half-pyramids pictured above happens to be `4`, the width of each half-pyramid `4`, with a gap of size `2` separating them).
- If the user fails to provide a positive integer no greater than `8`, you should re-prompt for the same again.
- Then, generate (with the help of `print` and one or more loops) the desired half-pyramids.
- Take care to align the bottom-left corner of your pyramid with the left-hand edge of your terminal window, and ensure that there are two spaces between the two pyramids, and that there are no additional spaces after the last set of hashes on each row.

Usage

Your program should behave per the example below.

```
$ ./mario
Height: 4
   #  #
  ##  ##
 ###  ###
####  ####
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python mario.py` and wait for a prompt for input. Type in `-1` and press enter. Your program should reject this input

- Run your program as `python mario.py` and wait for a prompt for input. Type in `-1` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Type in `0` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Type in `1` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
#  #
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `2` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
 #  #
##  ##
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `8` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
       #  #
      ##  ##
     ###  ###
    ####  ####
   #####  #####
  ######  ######
 #######  #######
########  ########
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `9` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number. Then, type in `2` and press enter. Your program should generate the below output. Be sure that the pyramid is aligned to the bottom-left corner of your terminal, and that there are no extra spaces at the end of each line.

```
 #  #
##  ##
```

- Run your program as `python mario.py` and wait for a prompt for input. Type in `foo` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python mario.py` and wait for a prompt for input. Do not type anything, and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/mario/more
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Cash

Implement a program that calculates the minimum number of coins required to give a user change.

```
$ python cash.py
Change owed: 0.41
4
```

Specification

- Write, in a file called `cash.py` in `~/pset6/cash/`, a program that first asks the user how much change is owed and then spits out the minimum number of coins with which said change can be made, exactly as you did in Problem Set 1, except that your program this time should be written (a) in Python and (b) in CS50 IDE.
- Use `get_float` from the CS50 Library to get the user's input and `print` to output your answer. Assume that the only coins available are quarters (25¢), dimes (10¢), nickels (5¢), and pennies (1¢).
  - We ask that you use `get_float` 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.
- 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, 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 a newline.

Usage

Your program should behave per the example below.

```
$ python cash.py
Change owed: 0.41
4
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python cash.py`, and wait for a prompt for input. Type in `0.41` and press enter. Your program should output `4`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `0.01` and press enter. Your program should output `1`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `0.15` and press enter. Your program should output `2`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `1.60` and press enter. Your program should output `7`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `23` and press enter. Your program should output `92`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `4.2` and press enter. Your program should output `18`.
- Run your program as `python cash.py`, and wait for a prompt for input. Type in `-1` and press enter. Your program should reject this input

as invalid, as by re-prompting the user to type in another number.

- Run your program as `python cash.py`, and wait for a prompt for input. Type in `foo` and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.
- Run your program as `python cash.py`, and wait for a prompt for input. Do not type anything, and press enter. Your program should reject this input as invalid, as by re-prompting the user to type in another number.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/cash
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Credit

Implement a program that determines whether a provided credit card number is valid according to Luhn's algorithm.

```
$ python credit.py
Number: 378282246310005
AMEX
```

Specification

- In `credit.py` in `~/pset6/credit/`, write a program that prompts the user for a credit card number and then reports (via `print`) whether it is a valid American Express, MasterCard, or Visa card number, exactly as you did in Problem Set 1, except that your program this time should be written (a) in Python and (b) in CS50 IDE.
- 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).
- Best to use `get_int` or `get_string` from CS50's library to get users' input, depending on how you to decide to implement this one.

Usage

Your program should behave per the example below.

```
$ python credit.py
Number: 378282246310005
AMEX
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python credit.py`, and wait for a prompt for input. Type in `378282246310005` and press enter. Your program should output `AMEX`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `371449635398431` and press enter. Your program should output `AMEX`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `5555555555554444` and press enter. Your program should output `MASTERCARD`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `5105105105105100` and press enter. Your program should output `MASTERCARD`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `4111111111111111` and press enter. Your program should output `VISA`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `4012888888881881` and press enter. Your program should output `VISA`.
- Run your program as `python credit.py`, and wait for a prompt for input. Type in `1234567890` and press enter. Your program should output `INVALID`

output `INVALID`.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (*) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/credit
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Readability

Implement a program that computes the approximate grade level needed to comprehend some text, per the below.

```
$ python readability.py
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

Specification

- Write, in a file called `readability.py` in `~/pset6/readability/`, a program that first asks the user to type in some text, and then outputs the grade level for the text, according to the Coleman-Liau formula, exactly as you did in Problem Set 2, except that your program this time should be written in Python.
  - Recall that the Coleman-Liau index is computed as `0.0588 * L - 0.296 * S - 15.8`, where `L` is the average number of letters per 100 words in the text, and `S` is the average number of sentences per 100 words in the text.
- Use `get_string` from the CS50 Library to get the user's input, and `print` to output your answer.
- Your program should count the number of letters, words, and sentences in the text. You may assume that a letter is any lowercase character from `a` to `z` or any uppercase character from `A` to `Z`, any sequence of characters separated by spaces should count as a word, and that any occurrence of a period, exclamation point, or question mark indicates the end of a sentence.
- Your program should print as output `"Grade X"` where `X` is the grade level computed by the Coleman-Liau formula, rounded to the nearest integer.
- If the resulting index number is 16 or higher (equivalent to or greater than a senior undergraduate reading level), your program should output `"Grade 16+"` instead of giving the exact index number. If the index number is less than 1, your program should output `"Before Grade 1"`.

Note that the specification here is only a summary of the requirements, so if you didn't do Readability in C, or if you are still unsure, we'd recommend that you review the **C specification and walkthrough** for clarification.

Usage

Your program should behave per the example below.

```
$ python readability.py
Text: Congratulations! Today is your day. You're off to Great Places! You're off and away!
Grade 3
```

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python readability.py`, and wait for a prompt for input. Type in `One fish. Two fish. Red fish. Blue fish.` and press enter. Your program should output `Before Grade 1`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `Would you like them here or there? I would not like them here or there. I would not like them anywhere.` and press enter. Your program should output `Grade 2`.

- Run your program as `python readability.py`, and wait for a prompt for input. Type in `Congratulations! Today is your day. You're off to Great Places! You're off and away!` and press enter. Your program should output `Grade 3`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of year. For another, he really wanted to do his homework, but was forced to do it in secret, in the dead of the night. And he also happened to be a wizard.` and press enter. Your program should output `Grade 5`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `In my younger and more vulnerable years my father gave me some advice that I've been turning over in my mind ever since.` and press enter. Your program should output `Grade 7`.

- Run your program as `python readability.py`, and wait for a prompt for input. Type in `Alice was beginning to get very tired of sitting by her sister on the bank, and of having nothing to do: once or twice she had peeped into the book her sister was reading, but it had no pictures or conversations in it, "and what is the use of a book," thought Alice "without pictures or conversation?"` and press enter. Your program should output `Grade 8`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `When he was nearly thirteen, my brother Jem got his arm badly broken at the elbow. When it healed, and Jem's fears of never being able to play football were assuaged, he was seldom self-conscious about his injury. His left arm was somewhat shorter than his right; when he stood or walked, the back of his hand was at right angles to his body, his thumb parallel to his thigh.` and press enter. Your program should output `Grade 8`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `There are more things in Heaven and Earth, Horatio, than are dreamt of in your philosophy.` and press enter. Your program should output `Grade 9`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `It was a bright cold day in April, and the clocks were striking thirteen. Winston Smith, his chin nuzzled into his breast in an effort to escape the vile wind, slipped quickly through the glass doors of Victory Mansions, though not quickly enough to prevent a swirl of gritty dust from entering along with him.` and press enter. Your program should output `Grade 10`.
- Run your program as `python readability.py`, and wait for a prompt for input. Type in `A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains.` and press enter. Your program should output `Grade 16+`.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/sentimental/readability
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

DNA

Implement a program that identifies a person based on their DNA, per the below.

```
$ python dna.py databases/large.csv sequences/5.txt
Lavender
```

Getting Started

Here's how to download this problem into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

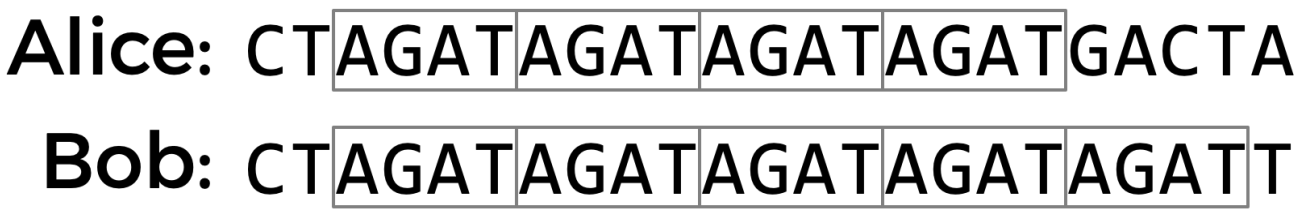
- Execute `cd` to ensure that you're in `~/` (i.e., your home directory, aka `~`).
- If you haven't already, execute `mkdir pset6` to make (i.e., create) a directory called `pset6` in your home directory.
- Execute `cd pset6` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/6/dna/dna.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip dna.zip` to uncompress that file.
- Execute `rm dna.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `dna`, which was inside of that ZIP file.
- Execute `cd dna` to change into that directory.
- Execute `ls`. You should see a directory of sample `databases` and a directory of sample `sequences`.

Background

DNA, the carrier of genetic information in living things, has been used in criminal justice for decades. But how, exactly, does DNA profiling work? Given a sequence of DNA, how can forensic investigators identify to whom it belongs?

Well, DNA is really just a sequence of molecules called nucleotides, arranged into a particular shape (a double helix). Each nucleotide of DNA contains one of four different bases: adenine (A), cytosine (C), guanine (G), or thymine (T). Every human cell has billions of these nucleotides arranged in sequence. Some portions of this sequence (i.e. genome) are the same, or at least very similar, across almost all humans, but other portions of the sequence have a higher genetic diversity and thus vary more across the population.

One place where DNA tends to have high genetic diversity is in Short Tandem Repeats (STRs). An STR is a short sequence of DNA bases that tends to repeat consecutively numerous times at specific locations inside of a person's DNA. The number of times any particular STR repeats varies a lot among individuals. In the DNA samples below, for example, Alice has the STR `AGAT` repeated four times in her DNA, while Bob has the same STR repeated five times.



Using multiple STRs, rather than just one, can improve the accuracy of DNA profiling. If the probability that two people have the same number

of repeats for a single STR is 5%, and the analyst looks at 10 different STRs, then the probability that two DNA samples match purely by chance is about 1 in 1 quadrillion (assuming all STRs are independent of each other). So if two DNA samples match in the number of repeats for each of the STRs, the analyst can be pretty confident they came from the same person. CODIS, The FBI's DNA database (https://www.fbi.gov/services/laboratory/biometric-analysis/codis/codis-and-ndis-fact-sheet), uses 20 different STRs as part of its DNA profiling process.

What might such a DNA database look like? Well, in its simplest form, you could imagine formatting a DNA database as a CSV file, wherein each row corresponds to an individual, and each column corresponds to a particular STR.

```
name,AGAT,AATG,TATC
Alice,28,42,14
Bob,17,22,19
Charlie,36,18,25
```

The data in the above file would suggest that Alice has the sequence `AGAT` repeated 28 times consecutively somewhere in her DNA, the sequence `AATG` repeated 42 times, and `TATC` repeated 14 times. Bob, meanwhile, has those same three STRs repeated 17 times, 22 times, and 19 times, respectively. And Charlie has those same three STRs repeated 36, 18, and 25 times, respectively.

So given a sequence of DNA, how might you identify to whom it belongs? Well, imagine that you looked through the DNA sequence for the longest consecutive sequence of repeated `AGAT` s and found that the longest sequence was 17 repeats long. If you then found that the longest sequence of `AATG` is 22 repeats long, and the longest sequence of `TATC` is 19 repeats long, that would provide pretty good evidence that the DNA was Bob's. Of course, it's also possible that once you take the counts for each of the STRs, it doesn't match anyone in your DNA database, in which case you have no match.

In practice, since analysts know on which chromosome and at which location in the DNA an STR will be found, they can localize their search to just a narrow section of DNA. But we'll ignore that detail for this problem.

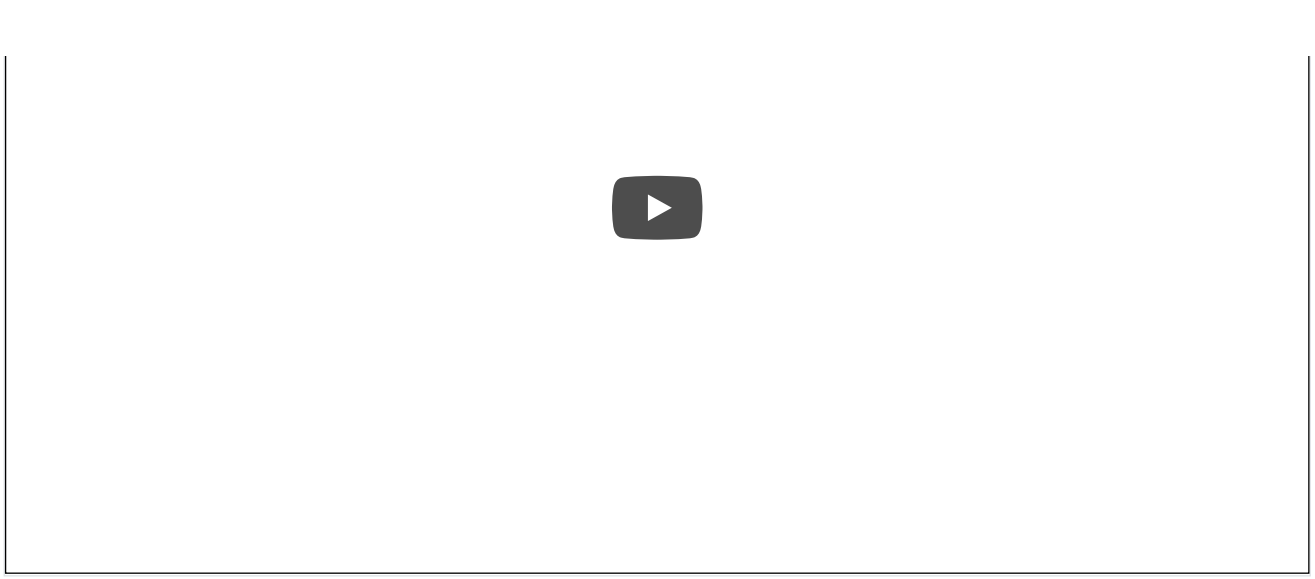
Your task is to write a program that will take a sequence of DNA and a CSV file containing STR counts for a list of individuals and then output to whom the DNA (most likely) belongs.

Specification

In a file called `dna.py` in `~/pset6/dna/` , implement a program that identifies to whom a sequence of DNA belongs.

- The program should require as its first command-line argument the name of a CSV file containing the STR counts for a list of individuals and should require as its second command-line argument the name of a text file containing the DNA sequence to identify.
  - If your program is executed with the incorrect number of command-line arguments, your program should print an error message of your choice (with `print`). If the correct number of arguments are provided, you may assume that the first argument is indeed the filename of a valid CSV file, and that the second argument is the filename of a valid text file.
- Your program should open the CSV file and read its contents into memory.
  - You may assume that the first row of the CSV file will be the column names. The first column will be the word `name` and the remaining columns will be the STR sequences themselves.
- Your program should open the DNA sequence and read its contents into memory.
- For each of the STRs (from the first line of the CSV file), your program should compute the longest run of consecutive repeats of the STR in the DNA sequence to identify.
- If the STR counts match exactly with any of the individuals in the CSV file, your program should print out the name of the matching individual.
  - You may assume that the STR counts will not match more than one individual.
  - If the STR counts do not match exactly with any of the individuals in the CSV file, your program should print `"No match"` .

Walkthrough



Usage

Your program should behave per the example below:

```
$ python dna.py databases/large.csv sequences/5.txt
Lavender
```

```
$ python dna.py
Usage: python dna.py data.csv sequence.txt
```

```
$ python dna.py data.csv
Usage: python dna.py data.csv sequence.txt
```

Hints

- You may find Python's `csv` (https://docs.python.org/3/library/csv.html) module helpful for reading CSV files into memory. You may want to take advantage of either `csv.reader` (https://docs.python.org/3/library/csv.html#csv.reader) or `csv.DictReader` (https://docs.python.org/3/library/csv.html#csv.DictReader).
- The `open` (https://docs.python.org/3.3/tutorial/inputoutput.html#reading-and-writing-files) and `read` (https://docs.python.org/3.3/tutorial/inputoutput.html#methods-of-file-objects) functions may prove useful for reading text files into memory.
- Consider what data structures might be helpful for keeping tracking of information in your program. A `list` (https://docs.python.org/3/tutorial/introduction.html#lists) or a `dict` (https://docs.python.org/3/tutorial/datastructures.html#dictionaries) may prove useful.

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

- Run your program as `python dna.py databases/small.csv sequences/1.txt`. Your program should output `Bob`.
- Run your program as `python dna.py databases/small.csv sequences/2.txt`. Your program should output `No match`.
- Run your program as `python dna.py databases/small.csv sequences/3.txt`. Your program should output `No match`.
- Run your program as `python dna.py databases/small.csv sequences/4.txt`. Your program should output `Alice`.
- Run your program as `python dna.py databases/large.csv sequences/5.txt`. Your program should output `Lavender`.
- Run your program as `python dna.py databases/large.csv sequences/6.txt`. Your program should output `Luna`.
- Run your program as `python dna.py databases/large.csv sequences/7.txt`. Your program should output `Ron`.
- Run your program as `python dna.py databases/large.csv sequences/8.txt`. Your program should output `Ginny`.
- Run your program as `python dna.py databases/large.csv sequences/9.txt`. Your program should output `Draco`.
- Run your program as `python dna.py databases/large.csv sequences/10.txt`. Your program should output `Albus`.
- Run your program as `python dna.py databases/large.csv sequences/11.txt`. Your program should output `Hermione`.

- Run your program as `python dna.py databases/large.csv sequences/12.txt`. Your program should output `Lily`.
- Run your program as `python dna.py databases/large.csv sequences/13.txt`. Your program should output `No match`.
- Run your program as `python dna.py databases/large.csv sequences/14.txt`. Your program should output `Severus`.
- Run your program as `python dna.py databases/large.csv sequences/15.txt`. Your program should output `Sirius`.
- Run your program as `python dna.py databases/large.csv sequences/16.txt`. Your program should output `No match`.
- Run your program as `python dna.py databases/large.csv sequences/17.txt`. Your program should output `Harry`.
- Run your program as `python dna.py databases/large.csv sequences/18.txt`. Your program should output `No match`.
- Run your program as `python dna.py databases/large.csv sequences/19.txt`. Your program should output `Fred`.
- Run your program as `python dna.py databases/large.csv sequences/20.txt`. Your program should output `No match`.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/dna
```

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Problem Set 7

What to Do

1. Submit Movies
2. Submit Houses

When to Do It

By 11:59pm on 31 December 2020.

Advice

- Head to w3schools.com/sql (https://www.w3schools.com/sql/) for a handy reference!

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Movies

Write SQL queries to answer questions about a database of movies.

Getting Started

Here's how to download this problem into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory, aka `~`).
- If you haven't already, execute `mkdir pset7` to make (i.e., create) a directory called `pset7` in your home directory.
- Execute `cd pset7` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/7/movies/movies.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip movies.zip` to uncompress that file.
- Execute `rm movies.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `movies`, which was inside of that ZIP file.
- Execute `cd movies` to change into that directory.
- Execute `ls`. You should see a `movies.db` file, and some empty `.sql` files as well.

Alternatively, you're welcome to download and unzip cdn.cs50.net/2019/fall/psets/7/movies/movies.zip (https://cdn.cs50.net/2019/fall/psets/7/movies/movies.zip) on your own computer and then open it in DB Browser for SQLite (https://sqlitebrowser.org/dl/). But be sure to upload your `.sql` files to CS50 IDE ultimately so that you can submit them via `submit50`.

Understanding

Provided to you is a file called `movies.db`, a SQLite database that stores data from IMDb (https://www.imdb.com/) about movies, the people who directed and starred in them, and their ratings. In a terminal window, run `sqlite3 movies.db` so that you can begin executing queries on the database.

First, when `sqlite3` prompts you to provide a query, type `.schema` and press enter. This will output the `CREATE TABLE` statements that were used to generate each of the tables in the database. By examining those statements, you can identify the columns present in each table.

Notice that the `movies` table has an `id` column that uniquely identifies each movie, as well as columns for the `title` of a movie and the `year` in which the movie was released. The `people` table also has an `id` column, and also has columns for each person's `name` and `birth` year.

Movie ratings, meanwhile, are stored in the `ratings` table. The first column in the table is `movie_id`: a foreign key that references the `id` of the `movies` table. The rest of the row contains data about the `rating` for each movie and the number of `votes` the movie has received on IMDb.

Finally, the `stars` and `directors` tables match people to the movies in which they acted or directed. (Only principal (https://www.imdb.com/interfaces/) stars and directors are included.) Each table has just two columns: `movie_id` and `person_id`, which reference a specific movie and person, respectively.

The challenge ahead of you is to write SQL queries to answer a variety of different questions by selecting data from one or more of these tables.

Specification

For each of the following problems, you should write a single SQL query that outputs the results specified by each problem. Your response must take the form of a single SQL query, though you may nest other queries inside of your query. You **should not** assume anything about the `id`s of any particular movies or people: your queries should be accurate even if the `id` of any particular movie or person were different. Finally, each query should return only the data necessary to answer the question: if the problem only asks you to output the names of movies, for example, then your query should not also output the each movie's release year.

You're welcome to check your queries' results against IMDb (https://www.imdb.com/) itself, but realize that ratings on the website might differ from those in `movies.db`, as more votes might have been cast since we downloaded the data!

1. In `1.sql`, write a SQL query to list the titles of all movies released in 2008.
    - Your query should output a table with a single column for the title of each movie.
2. In `2.sql`, write a SQL query to determine the birth year of Emma Stone.
    - Your query should output a table with a single column and a single row (plus optional header) containing Emma Stone's birth year.
    - You may assume that there is only one person in the database with the name Emma Stone.
3. In `3.sql`, write a SQL query to list the titles of all movies with a release date on or after 2018, in alphabetical order.
    - Your query should output a table with a single column for the title of each movie.
    - Movies released in 2018 should be included, as should movies with release dates in the future.
4. In `4.sql`, write a SQL query to determine the number of movies with an IMDb rating of 10.0.
    - Your query should output a table with a single column and a single row (plus optional header) containing the number of movies with a 10.0 rating.
5. In `5.sql`, write a SQL query to list the titles and release years of all Harry Potter movies, in chronological order.
    - Your query should output a table with two columns, one for the title of each movie and one for the release year of each movie.
    - You may assume that the title of all Harry Potter movies will begin with the words "Harry Potter", and that if a movie title begins with the words "Harry Potter", it is a Harry Potter movie.
6. In `6.sql`, write a SQL query to determine the average rating of all movies released in 2012.
    - Your query should output a table with a single column and a single row (plus optional header) containing the average rating.
7. In `7.sql`, write a SQL query to list all movies released in 2010 and their ratings, in descending order by rating. For movies with the same rating, order them alphabetically by title.
    - Your query should output a table with two columns, one for the title of each movie and one for the rating of each movie.
    - Movies that do not have ratings should not be included in the result.
8. In `8.sql`, write a SQL query to list the names of all people who starred in Toy Story.
    - Your query should output a table with a single column for the name of each person.
    - You may assume that there is only one movie in the database with the title Toy Story.
9. In `9.sql`, write a SQL query to list the names of all people who starred in a movie released in 2004, ordered by birth year.
    - Your query should output a table with a single column for the name of each person.
    - People with the same birth year may be listed in any order.
    - No need to worry about people who have no birth year listed, so long as those who do have a birth year are listed in order.
    - If a person appeared in more than one movie in 2004, they should only appear in your results once.
10. In `10.sql`, write a SQL query to list the names of all people who have directed a movie that received a rating of at least 9.0.
    - Your query should output a table with a single column for the name of each person.
11. In `11.sql`, write a SQL query to list the titles of the five highest rated movies (in order) that Chadwick Boseman starred in, starting with the highest rated.
    - Your query should output a table with a single column for the title of each movie.
    - You may assume that there is only one person in the database with the name Chadwick Boseman.
12. In `12.sql`, write a SQL query to list the titles of all movies in which both Johnny Depp and Helena Bonham Carter starred.
    - Your query should output a table with a single column for the title of each movie.
    - You may assume that there is only one person in the database with the name Johnny Depp.
    - You may assume that there is only one person in the database with the name Helena Bonham Carter.
13. In `13.sql`, write a SQL query to list the names of all people who starred in a movie in which Kevin Bacon also starred.
    - Your query should output a table with a single column for the name of each person.
    - There may be multiple people named Kevin Bacon in the database. Be sure to only select the Kevin Bacon born in 1958.
    - Kevin Bacon himself should not be included in the resulting list.

Walkthrough

Walkthrough

Usage

To test your queries on CS50 IDE, you can query the database by running

```
$ cat filename.sql | sqlite3 movies.db
```

where `filename.sql` is the file containing your SQL query.

Or you can paste them into DB Browser for SQLite's **Execute SQL** tab and click ▶.

Hints

- See this SQL keywords reference (https://www.w3schools.com/sql/sql_ref_keywords.asp) for some SQL syntax that may be helpful!

Testing

No `check50` for this problem! But be sure to test each query and ensure that the output is what you expect. You can run `sqlite3 movies.db` to run additional queries on the database to ensure that your result is correct.

If you're using the `movies.db` database provided in this problem set's distribution, you should find that

- Executing `1.sql` results in a table with 1 column and 9,480 rows.
- Executing `2.sql` results in a table with 1 column and 1 row.
- Executing `3.sql` results in a table with 1 column and 35,755 rows.
- Executing `4.sql` results in a table with 1 column and 1 row.
- Executing `5.sql` results in a table with 2 columns and 10 rows.
- Executing `6.sql` results in a table with 1 column and 1 row.
- Executing `7.sql` results in a table with 2 columns and 6,835 rows.
- Executing `8.sql` results in a table with 1 column and 4 rows.
- Executing `9.sql` results in a table with 1 column and 18,013 rows.
- Executing `10.sql` results in a table with 1 column and 1,841 rows.
- Executing `11.sql` results in a table with 1 column and 5 rows.
- Executing `12.sql` results in a table with 1 column and 6 rows.
- Executing `13.sql` results in a table with 1 column and 176 rows.

- Executing `13.sql` results in a table with 1 column and 176 rows.

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2020/x/movies
```

Acknowledgements

Information courtesy of IMDb (imdb.com (http://www.imdb.com)). Used with permission.

This is CS50x

OpenCourseWare

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Houses

Implement a program to import student data into a database, and then produce class rosters.

```
$ python import.py characters.csv
$ python roster.py Gryffindor

Lavender Brown, born 1979
Colin Creevey, born 1981
Seamus Finnigan, born 1979
Hermione Jean Granger, born 1979
Neville Longbottom, born 1980
Parvati Patil, born 1979
Harry James Potter, born 1980
Dean Thomas, born 1980
Romilda Vane, born 1981
Ginevra Molly Weasley, born 1981
Ronald Bilius Weasley, born 1980
```

Getting Started

Here's how to download this problem into your own CS50 IDE. Log into CS50 IDE (https://ide.cs50.io/) and then, in a terminal window, execute each of the below.

- Execute `cd` to ensure that you're in `~/` (i.e., your home directory, aka `~`).
- If you haven't already, execute `mkdir pset7` to make (i.e., create) a directory called `pset7` in your home directory.
- Execute `cd pset7` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2019/fall/psets/7/houses/houses.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip houses.zip` to uncompress that file.
- Execute `rm houses.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `houses`, which was inside of that ZIP file.
- Execute `cd houses` to change into that directory.
- Execute `ls`. You should see a `characters.csv` file and a `students.db` database.

Background

Hogwarts is in need of a student database. For years, the professors have been maintaing a CSV file containing all of the students' names and houses and years. But that file didn't make it particularly easy to get access to certain data, such as a roster of all the Ravenclaw students, or an alphabetical listing of the students enrolled at the school.

The challenge ahead of you is to import all of the school's data into a SQLite database, and write a Python program to query that database to get house rosters for each of the houses of Hogwarts.

Specification

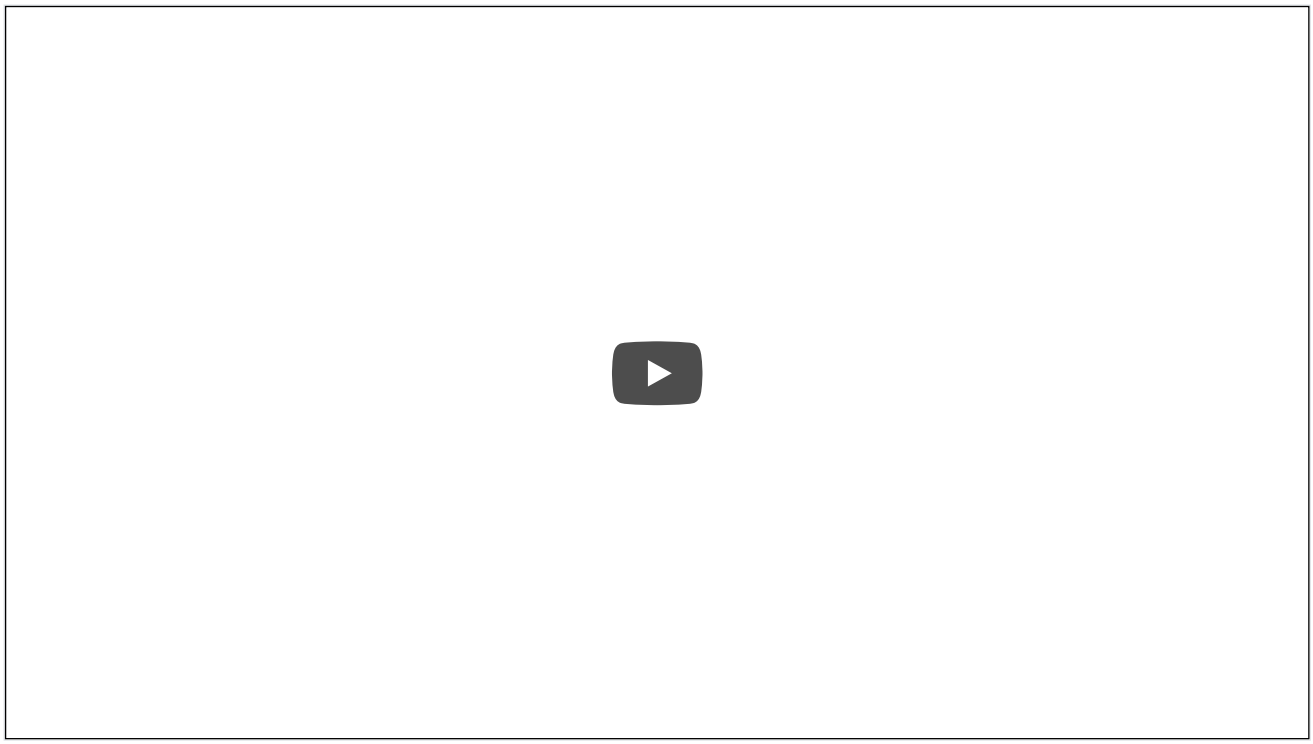
In `import.py`, write a program that imports data from a CSV spreadsheet.

- Your program should accept the name of a CSV file as a command-line argument.
  - If the incorrect number of command-line arguments are provided, your program should print an error and exit.
  - You may assume that the CSV file will exist, and will have columns `name`, `house`, and `birth`.
- For each student in the CSV file, insert the student into the `students` table in the `students.db` database.
  - While the CSV file provided to you has just a `name` column, the database has separate columns for `first`, `middle`, and `last` names. You'll thus want to first parse each name and separate it into first, middle, and last names. You may assume that each person's name field will contain either two space-separated names (a first and last name) or three space-separated names (a first, middle, and last name). For students without a middle name, you should leave their `middle` name field as `NULL` in the table.

In `roster.py`, write a program that prints a list of students for a given house in alphabetical order.

- Your program should accept the name of a house as a command-line argument.
  - If the incorrect number of command-line arguments are provided, your program should print an error and exit.
- Your program should query the `students` table in the `students.db` database for all of the students in the specified house.
- Your program should then print out each student's full name and birth year (formatted as, e.g., `Harry James Potter, born 1980` or `Luna Lovegood, born 1981`).
  - Each student should be printed on their own line.
  - Students should be ordered by last name. For students with the same last name, they should be ordered by first name.

Walkthrough



Usage

Your program should behave per the example below:

```
$ python import.py characters.csv
```

```
$ python roster.py Gryffindor
Hermione Jean Granger, born 1979
Harry James Potter, born 1980
Ginevra Molly Weasley, born 1981
Ronald Bilius Weasley, born 1980
...
```

Hints

- Recall that after you've imported `SQL` from `cs50`, you can set up a database connection using syntax like

```
db = SQL("sqlite:///students.db")
```

Then, you can use `db.execute` to execute SQL queries from inside of your Python script.

- Recall that when you call `db.execute` and perform a `SELECT` query, the return value will be a `list` of rows that are returned, where each row is represented by a Python `dict`.

Testing

No `check50` for this problem, but be sure to test your code for each of the following.

```
$ python import.py characters.csv
$ python roster.py Gryffindor
Lavender Brown, born 1979
Colin Creevey, born 1981
Seamus Finnigan, born 1979
Hermione Jean Granger, born 1979
Neville Longbottom, born 1980
Parvati Patil, born 1979
Harry James Potter, born 1980
Dean Thomas, born 1980
Romilda Vane, born 1981
Ginevra Molly Weasley, born 1981
Ronald Bilius Weasley, born 1980

$ python roster.py Hufflepuff
Hannah Abbott, born 1980
Susan Bones, born 1979
Cedric Diggory, born 1977
Justin Finch-Fletchley, born 1979
Ernest Macmillan, born 1980

$ python roster.py Ravenclaw
Terry Boot, born 1980
Mandy Brocklehurst, born 1979
Cho Chang, born 1979
Penelope Clearwater, born 1976
Michael Corner, born 1979
Roger Davies, born 1978
Marietta Edgecombe, born 1978
Anthony Goldstein, born 1980
Robert Hilliard, born 1974
Luna Lovegood, born 1981
Isobel MacDougal, born 1980
Padma Patil, born 1979
Lisa Turpin, born 1979

$ python roster.py Slytherin
Millicent Bulstrode, born 1979
Vincent Crabbe, born 1979
Tracey Davis, born 1980
Marcus Flint, born 1975
Gregory Goyle, born 1980
Terence Higgs, born 1979
Draco Lucius Malfoy, born 1980
Adelaide Murton, born 1982
Pansy Parkinson, born 1979
Adrian Pucey, born 1977
Blaise Zabini, born 1979
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
```

```
submit50 cs50/problems/2020/x/houses
```