CS50x

https://cs50.harvard.edu/x/2021/

The intellectual enterprises of computer science and the art of programming.

Оглавление

Week 0 Scratch

Lecture 0

- Welcome
- What is computer science?
- Representing numbers
- Text
- Images, video, sounds
- Algorithms
- Pseudocode
- Scratch

Welcome

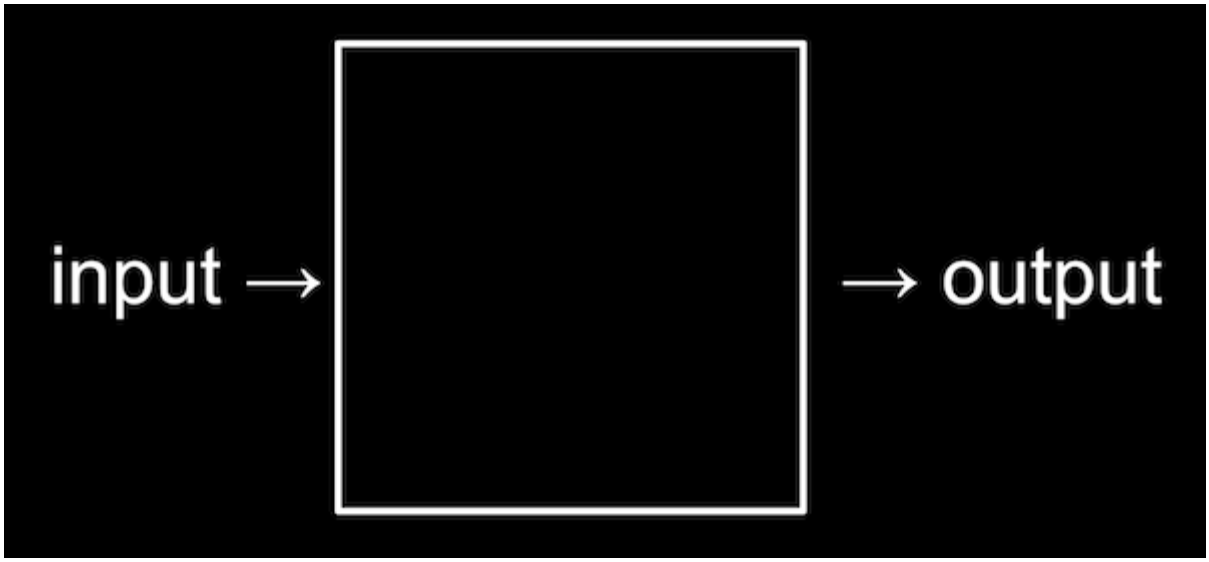
- This year, we'll be in the Loeb Drama Center at Harvard University, where, thanks to our close collaboration with the American Repertory Theater, we have an amazing stage and even props for demonstrations.
- We turned an 18th century watercolor painting of Harvard's campus by a student, Jonathan Fisher, into the backdrop for the stage.
- Twenty years ago as an undergraduate, David overcame his own trepidation, stepping outside his comfort zone and took CS50 himself, finding that the course was less about programming than about problem solving.
- In fact, two-thirds of CS50 students have never taken a computer science course before.
- And importantly, too:

  > what ultimately matters in this course is not so much where you end up relative to your classmates but where you end up relative to yourself when you began

- We'll start off the course recreating a component of a Super Mario game, later building a web application called CS50 Finance that will allow users to buy and sell stocks virtually, and ending the course with the creation of your very own final project.

What is computer science?

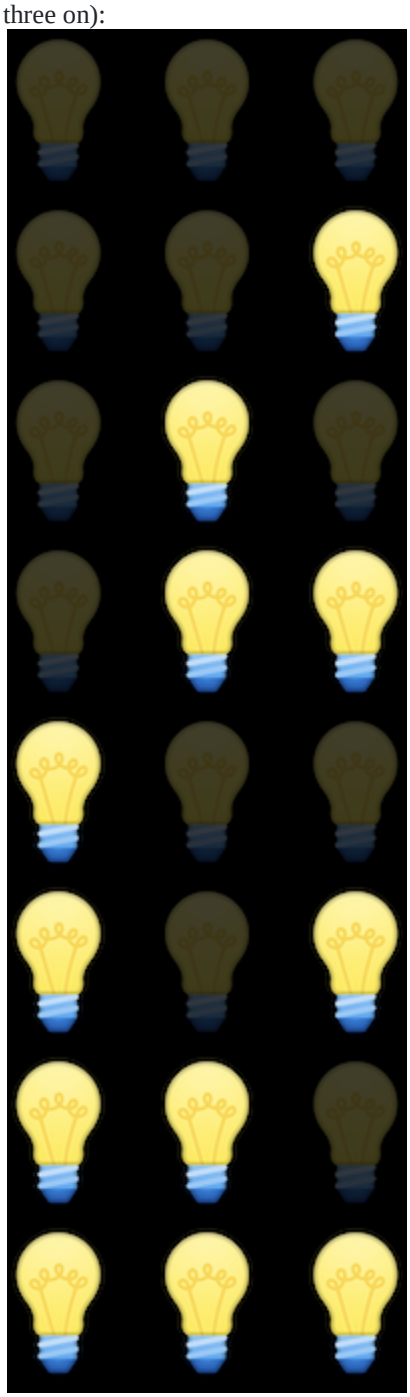
- Computer science is fundamentally problem solving.
- We can think of **problem solving** as the process of taking some input (details about our problem) and generate some output (the solution to our problem). The "black box" in the middle is computer science, or the code we'll learn to write.



- To begin doing that, we'll need a way to represent inputs and outputs, so we can store and work with information in a standardized way.

Representing numbers

- We might start with the task of taking attendance by counting the number of people in a room. With our hand, we might raise one finger at a time to represent each person, but we won't be able to count very high. This system is called **unary**, where each digit represents a single value of one.
- We've probably learned a more efficient system to represent numbers, where we have ten digits, 0 through 9:
- 0 1 2 3 4 5 6 7 8 9
    - This system is called decimal, or **base 10**, since there are ten different values that a digit can represent.
- Computers use a simpler system called **binary**, or base two, with only two possible digits, 0 and 1.
    - Each *binary digit* is also called a **bit**.
- Since computers run on electricity, which can be turned on or off, we can conveniently represent a bit by turning some switch on or off to represent a 0 or 1.
    - With one light bulb, for example, we can turn it on to count to 1.
- With three light bulbs, we can turn them on in different patterns, and count from 0 (with all three off) to 7 (with all three on):



- Inside modern computers, there are not light bulbs but million of tiny switches called **transistors** that can be turned on and off to represent different values.

- For example, we know the following number in decimal represents one hundred and twenty-three.
- 1 2 3
    - The 3 is in the ones column, the 2 is in the tens column, and the 1 is in the hundreds column.
    - So 123 is $100 \times 1 + 10 \times 2 + 1 \times 3 = 100 + 20 + 3 = 123$.
    - Each place for a digit represents a power of ten, since there are ten possible digits for each place. The rightmost place is for 10^0, the middle one 10^1, and the leftmost place 10^2:
    - 10^2 10^1 10^0
    - 1 2 3
- In binary, with just two digits, we have powers of two for each place value:
- 2^2 2^1 2^0
- # # #
    - This is equivalent to:
    - 4 2 1
    - # # #
- With all the light bulbs or switches off, we would still have a value of 0:
- 4 2 1
- **0 0 0**
- Now if we change the binary value to, say, 0 1 1, the decimal value would be 3, since we add the 2 and the 1:
- 4 2 1
- **0 1 1**
- If we had several more light bulbs, we might have a binary value of 110010, which would have the equivalent decimal value of 50:
- 32 16 8 4 2 1
- **1 1 0 0 1 0**
    - Notice that 32 + 16 + 2 = 50.
- With more bits, we can count up to even higher numbers.

Text

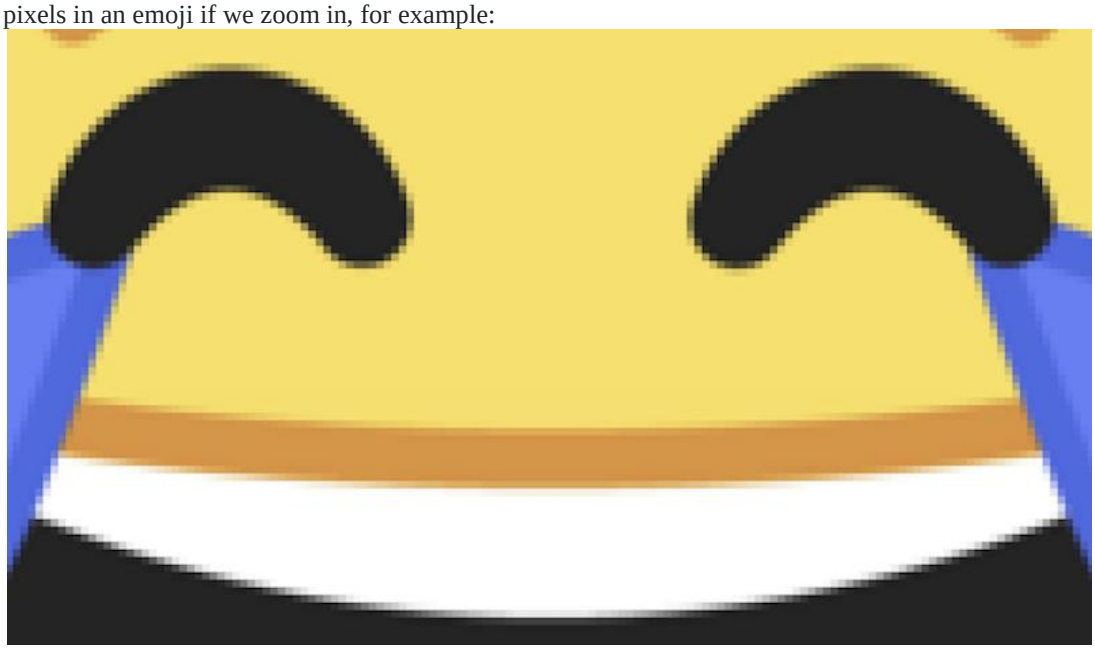
- To represent letters, all we need to do is decide how numbers map to letters. Some humans, many years ago, collectively decided on a standard mapping of numbers to letters. The letter "A", for example, is the number 65, and "B" is 66, and so on. By using context, like whether we're looking at a spreadsheet or an email, different programs can interpret and display the same bits as numbers or text.
- The standard mapping, **ASCII**, also includes lowercase letters and punctuation.
- If we received a text message with a pattern of bits that had the decimal values 72, 73, and 33, those bits would map to the letters HI!. Each letter is typically represented with a pattern of eight bits, or a **byte**, so the sequences of bits we would receive are 01001000, 01001001, and 00100001.
    - We might already be familiar with using bytes as a unit of measurement for data, as in megabytes or gigabytes, for millions or billions of bytes.
- With eight bits, or one byte, we can have 2^8, or 256 different values (including zero). (The *highest value* we can count up to would be 255.)
- Other characters, such as letters with accent marks and symbols in other languages, are part of a standard called **Unicode**, which uses more bits than ASCII to accommodate all these characters.
    - When we receive an emoji, our computer is actually just receiving a number in binary that it then maps to the image of the emoji based on the Unicode standard.
        - For example, the "face with tears of joy" emoji is just the bits 00000001111011000000010:

Images, video, sounds

- An image, like the picture of the emoji, are made up of colors.
- With only bits, we can map numbers to colors as well. There are many different systems to represent colors, but a common one is **RGB**, which represents different colors by indicating the amount of red, green, and blue within each color.

- For example, our pattern of bits earlier, 72, 73, and 33 might indicate the amount of red, green, and blue in a color. (And our programs would know those bits map to a color if we opened an image file, as opposed to receiving them in a text message.)
    - Each number might be a byte, with 256 possible values, so with three bytes, we can represent millions of colors. Our three bytes from above would represent a dark shade of yellow:

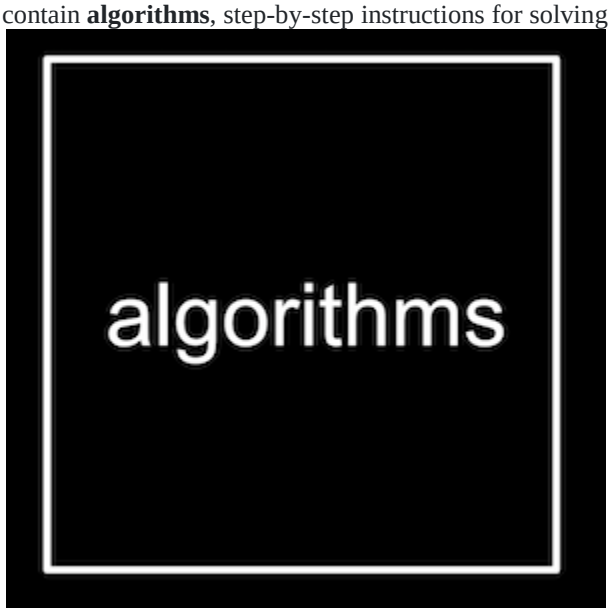
- The dots, or squares, on our screens are called **pixels**, and images are made up of many thousands or millions of those pixels as well. So by using three bytes to represent the color for each pixel, we can create images. We can see pixels in an emoji if we zoom in, for example:



- The **resolution** of an image is the number of pixels there are, horizontally and vertically, so a high-resolution image will have more pixels and require more bytes to be stored.
- Videos are made up of many images, changing multiple times a second to give us the appearance of motion, as an old-fashioned flipbook might do.
- Music can be represented with bits, too, with mappings of numbers to notes and durations, or more complex mappings of bits to sound frequencies at each moment of time.
- File formats, like JPEG and PNG, or Word or Excel documents, are also based on some standard that some humans have agreed on, for representing information with bits.

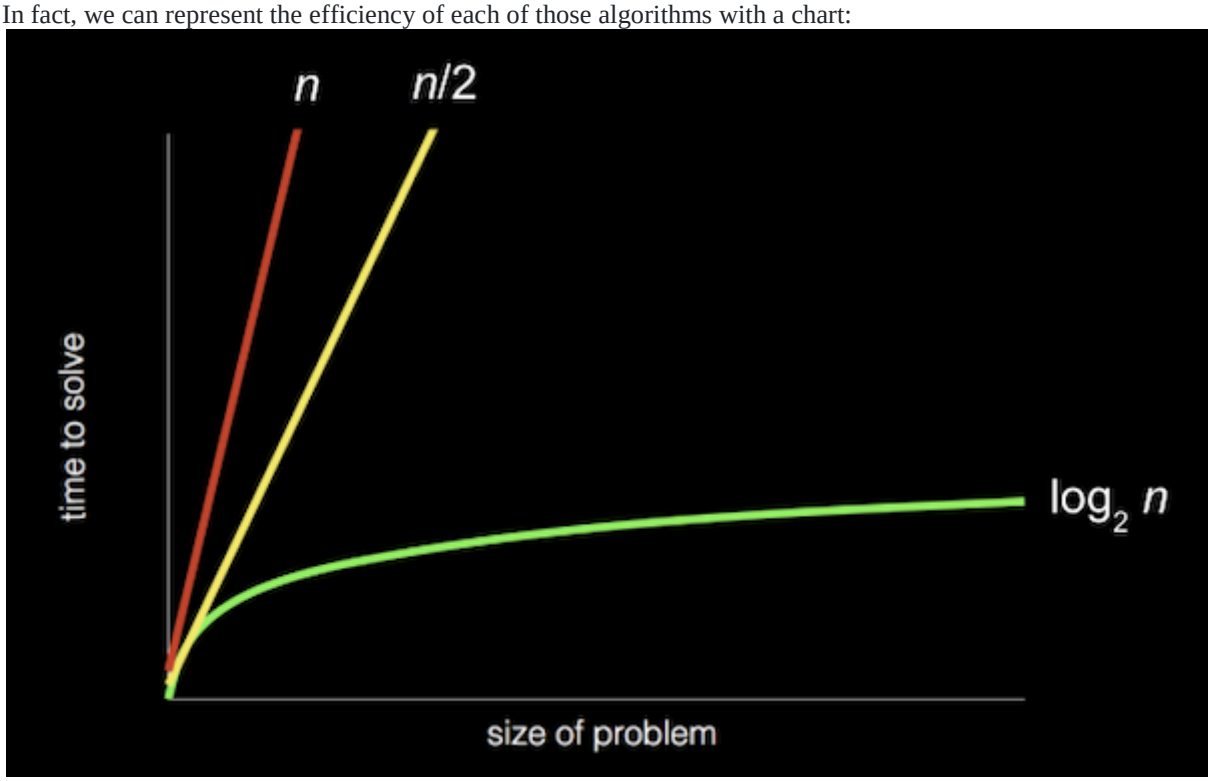
Algorithms

- Now that we can represent inputs and outputs, we can work on problem solving. The black box earlier will contain **algorithms**, step-by-step instructions for solving problems:



- Humans can follow algorithms too, such as recipes for cooking. When programming a computer, we need to be more precise with our algorithms so our instructions aren't ambiguous or misinterpreted.
- We might have an application on our phones that store our contacts, with their names and phone numbers sorted alphabetically. The old-school equivalent might be a phone book, a printed copy of names and phone numbers.
- Our input to the problem of finding someone's number would be the phone book and a name to look for. We might open the book and start from the first page, looking for a name one page at a time. This algorithm would be **correct**, since we will eventually find the name if it's in the book.
- We might flip through the book two pages at a time, but this algorithm will not be correct since we might skip the page with our name on it. We can fix this **bug**, or mistake, by going back one page if we flipped too far, since we know the phone book is sorted alphabetically.
- Another algorithm would be opening the phone book to the middle, decide whether our name will be in the left half or right half of the book (because the book is alphabetized), and reduce the size of our problem by half. We can repeat this until we find our name, dividing the problem in half each time. With 1024 pages to start, we would only need 10 steps of dividing in half before we have just one page remaining to check. We can see this visualized in an animation of dividing a phone book in half repeatedly, compared to the animation of searching one page at a time.

- In fact, we can represent the efficiency of each of those algorithms with a chart:



    - Our first solution, searching one page at a time, can be represented by the red line: our time to solve increases linearly as the size of the problem increases. *n* is a some number representing the size of the problem, so with *n* pages in our phone books, we have to take up to *n* steps to find a name.
    - The second solution, searching two pages at a time, can be represented by the yellow line: our slope is less steep, but still linear. Now, we only need (roughly) *n* / 2 steps, since we flip two pages at a time.
    - Our final solution, dividing the phone book in half each time, can be represented by the green line, with a fundamentally different relationship between the size of the problem and the time to solve it: **logarithmic**, since our time to solve rises more and more slowly as the size of the problem increases. In other words, if the phone book went from 1000 to 2000 pages, we would only need one more step to find our name. If the size doubled again from 2000 to 4000 pages, we would still only need one more step. The green line is labeled $\log_2 n$, or log base 2 of *n*, since we're dividing the problem by two with each step.
- When we write programs using algorithms, we generally care not just how correct they are, but how **well-designed** they are, considering factors such as efficiency.

Pseudocode

- We can write **pseudocode**, which is a representation of our algorithm in precise English (or some other human language):

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If person is on page
5      Call person
6  Else if person is earlier in book
7      Open to middle of left half of book
8      Go back to line 3
9  Else if person is later in book
10     Open to middle of right half of book
11     Go back to line 3
12 Else
13     Quit
```

    - With these steps, we check the middle page, decide what to do, and repeat. If the person isn't on the page, and there's no more pages in the book left, then we stop. And that final case is particularly important to remember. When other programs on our computers forgot that final case, they might appear to freeze or stop responding, since they've encountered a case that wasn't accounted for, or continue to repeat the same work over and over behind the scenes without making any progress.

- Some of these lines start with verbs, or actions. We'll start calling these *functions*:
- 1 **Pick up** phone book
- 2 **Open to** middle of phone book
- 3 **Look at** page
- 4 If person is on page
- 5 **Call** person
- 6 Else if person is earlier in book
- 7 **Open to** middle of left half of book
- 8 Go back to line 3
- 9 Else if person is later in book
- 10 **Open to** middle of right half of book
- 11 Go back to line 3
- 12 Else
- 13 **Quit**
- We also have branches that lead to different paths, like forks in the road, which we'll call *conditions*:
- 1 Pick up phone book
- 2 Open to middle of phone book
- 3 Look at page
- 4 **If** person is on page
- 5 Call person
- 6 **Else if** person is earlier in book
- 7 Open to middle of left half of book
- 8 Go back to line 3
- 9 **Else if** person is later in book
- 10 Open to middle of right half of book
- 11 Go back to line 3
- 12 **Else**
- 13 Quit
- And the questions that decide where we go are called *Boolean expressions*, which eventually result in a value of yes or no, or true or false:
- 1 Pick up phone book
- 2 Open to middle of phone book
- 3 Look at page
- 4 If **person is on page**
- 5 Call person
- 6 Else if **person is earlier in book**
- 7 Open to middle of left half of book
- 8 Go back to line 3
- 9 Else if **person is later in book**
- 10 Open to middle of right half of book
- 11 Go back to line 3
- 12 Else
- 13 Quit
- Lastly, we have words that create cycles, where we can repeat parts of our program, called *loops*:
- 1 Pick up phone book
- 2 Open to middle of phone book
- 3 Look at page
- 4 If person is on page
- 5 Call person
- 6 Else if person is earlier in book
- 7 Open to middle of left half of book
- 8 **Go back to line 3**
- 9 Else if person is later in book
- 10 Open to middle of right half of book
- 11 **Go back to line 3**
- 12 Else
- 13 Quit

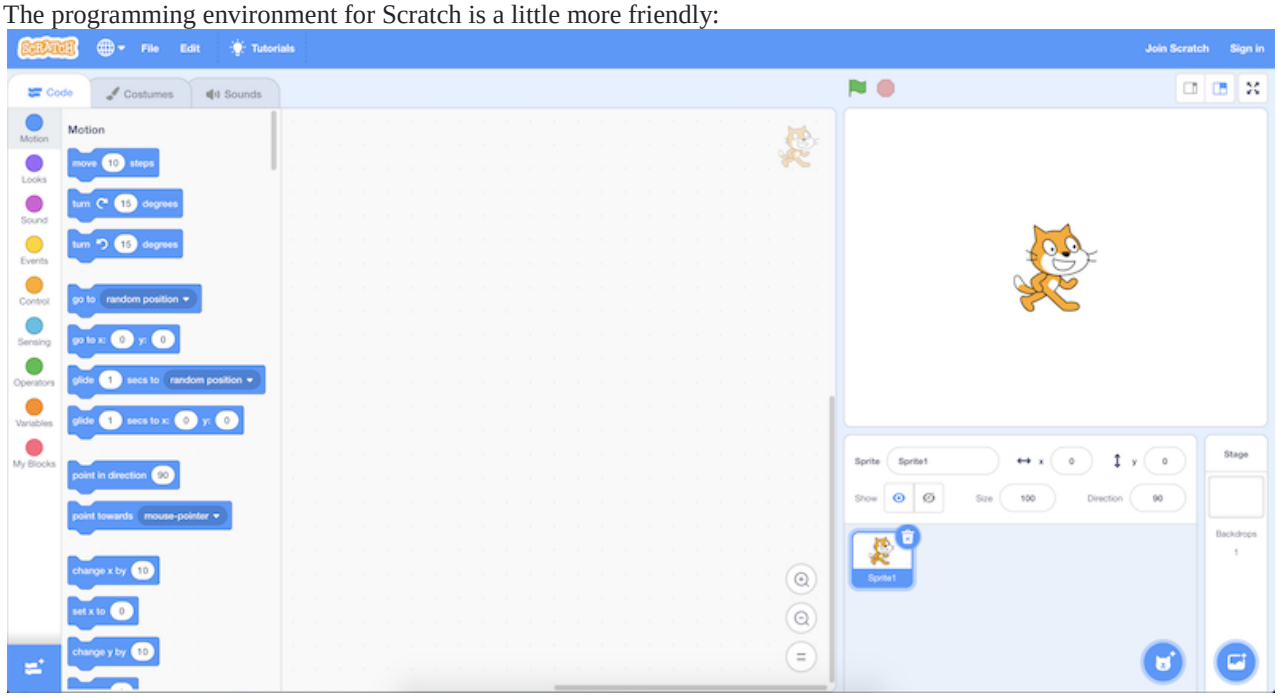
Scratch

- We can write programs with the building blocks we just discovered:
  - functions
  - conditions
  - Boolean expressions
  - loops
- And we'll discover additional features including:
  - variables
  - threads
  - events
  - …
- Before we learn to use a text-based programming language called C, we'll use a graphical programming language called Scratch, where we'll drag and drop blocks that contain instructions.
- A simple program in C that prints out "hello, world", would look like this:

```
#include <stdio.h>

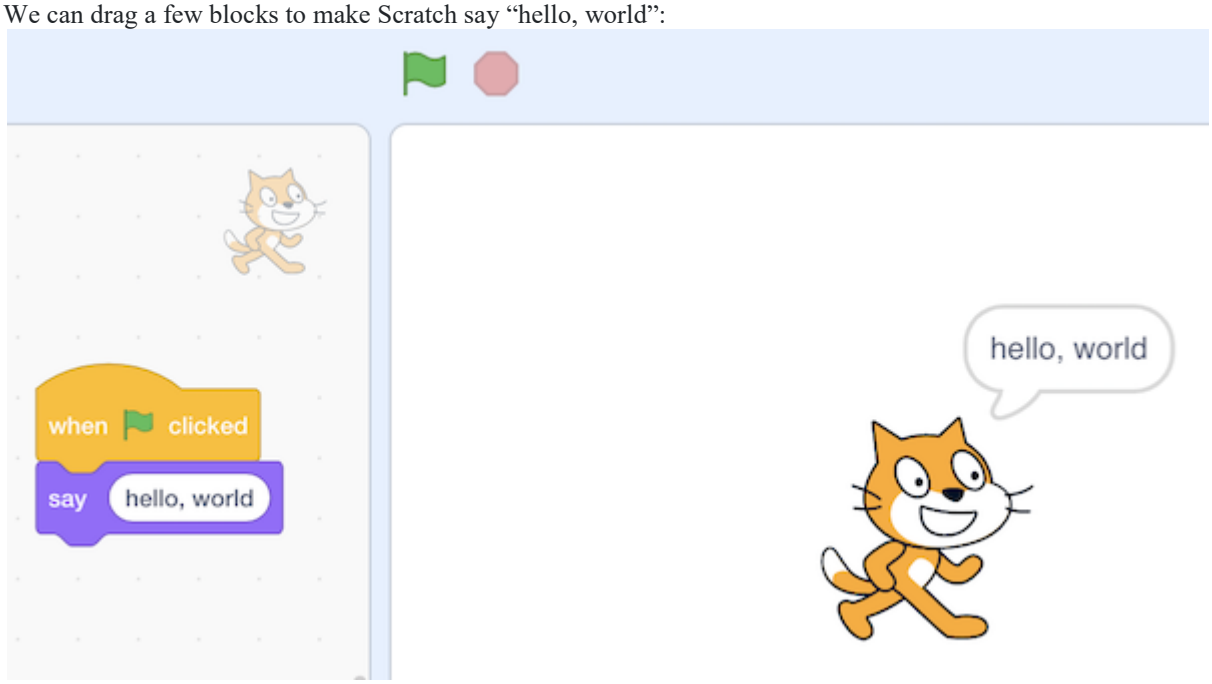
int main(void)
{
    printf("hello, world\n");
}
```

  - There's a lot of symbols and syntax, or arrangement of these symbols, that we would have to figure out.
- The programming environment for Scratch is a little more friendly:

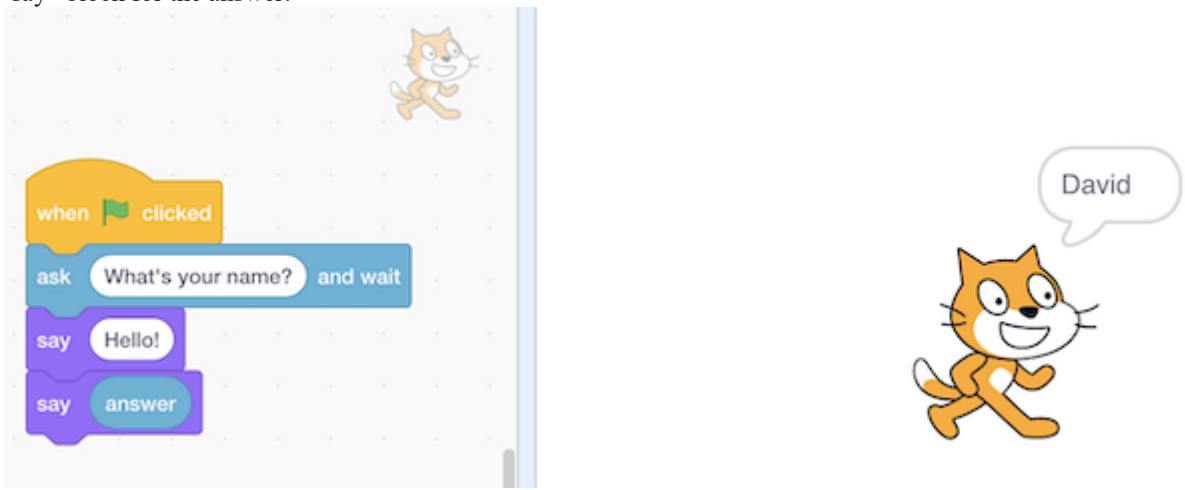


  - On the top right, we have a stage that will be shown by our program, where we can add or change backgrounds, characters (called sprites in Scratch), and more.
  - On the left, we have puzzle pieces that represent functions or variables, or other concepts, that we can drag and drop into our instruction area in the center.
  - On the bottom right, we can add more characters for our program to use.

- We can drag a few blocks to make Scratch say “hello, world”:



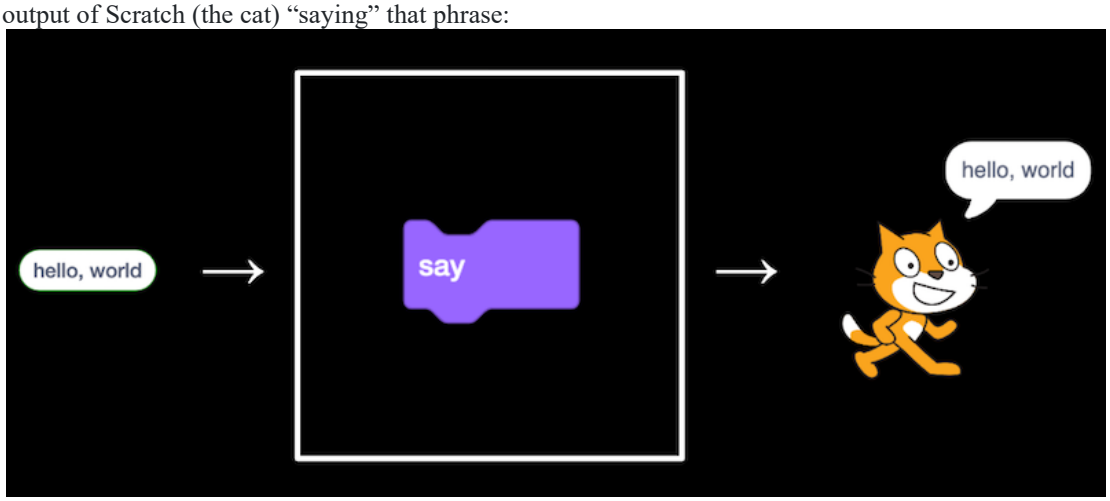
  - The “when green flag clicked” block refers to the start of our program (since there is a green flag above the stage that we can use to start it), and below it we’ve snapped in a “say” block and typed in “hello, world”. And we can figure out what these blocks do by exploring the interface and experimenting.
- We can also drag in the “ask and wait” block, with a question like “What’s your name?”, and combine it with a “say” block for the answer:



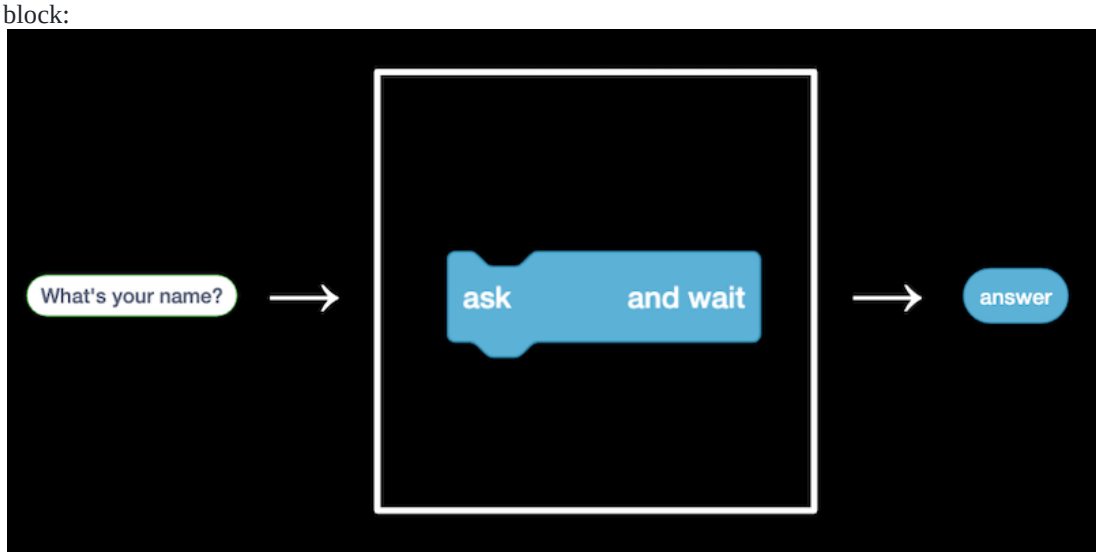
  - The “answer” block is a variable, or value, that stores what the program’s user types in, and we can place it in a “say” block by draggind and dropping as well.
- But we didn’t wait after we said “Hello” with the first block, so we can use the “join” block to combine two phrases so our cat can say “hello, David”:

  - When we try to nest blocks, or place them one inside the other, Scratch will help us by expanding places where they can be used.

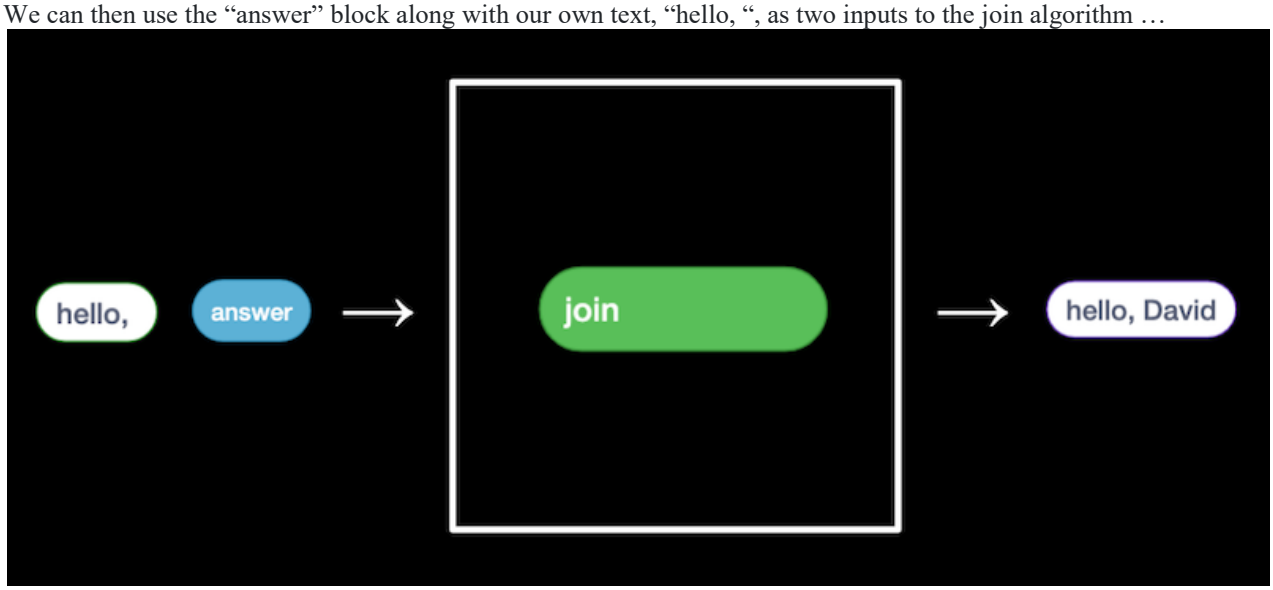
- In fact, the “say” block itself is like an algorithm, where we provided an input of “hello, world” and it produced the output of Scratch (the cat) “saying” that phrase:



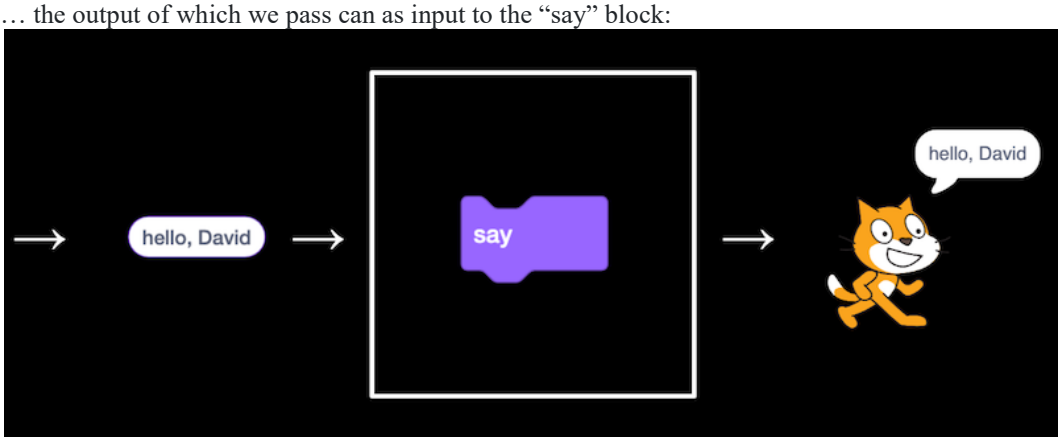
- The “ask” block, too, takes in an input (the question we want to ask), and produces the output of the “answer” block:



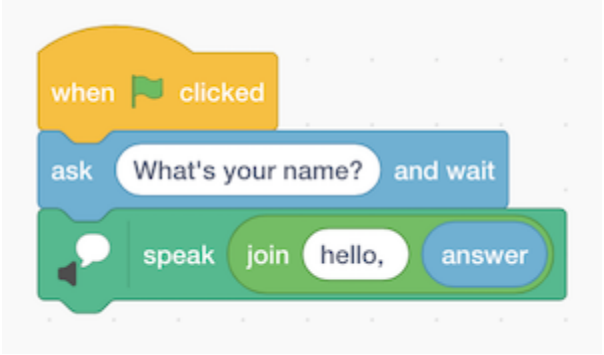
- We can then use the “answer” block along with our own text, “hello, “, as two inputs to the join algorithm …



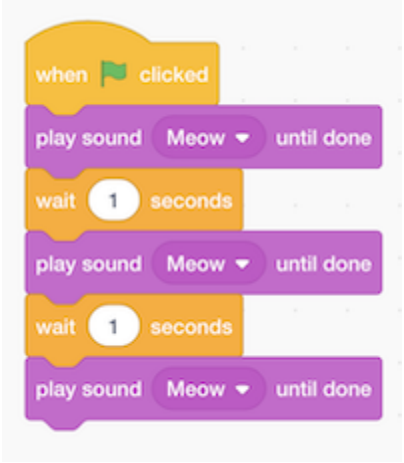
- … the output of which we pass can as input to the “say” block:



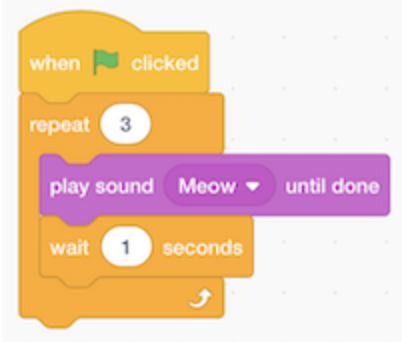
- At the bottom left of the screen, we see an icon for extensions, and one of them is called Text to Speech. After we add it, we can use the “speak” block to hear our cat speak:



- The Text to Speech extension, thanks to the cloud, or computer servers on the internet, is converting our text to audio.
- We can try to make the cat say meow:

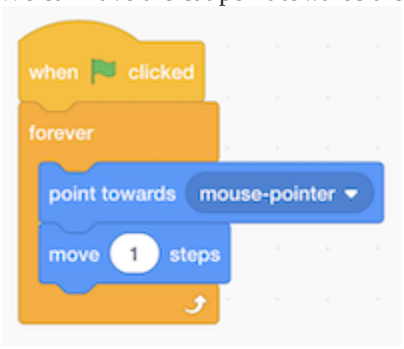


  - We can have it say meow three times, but now we’re repeating blocks over and over.
- Let’s use a loop, or a “repeat” block:

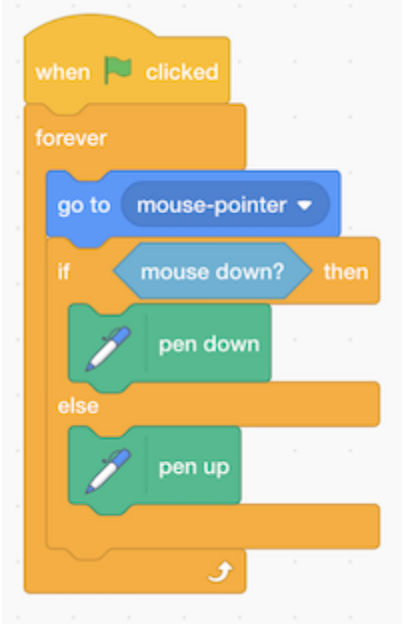


  - Now our program achieves the same results, but with fewer blocks. We can consider it to have a better design: if there’s something we wanted to change, we would only need to change it in one place instead of three.

- We can have the cat point towards the mouse and move towards it:



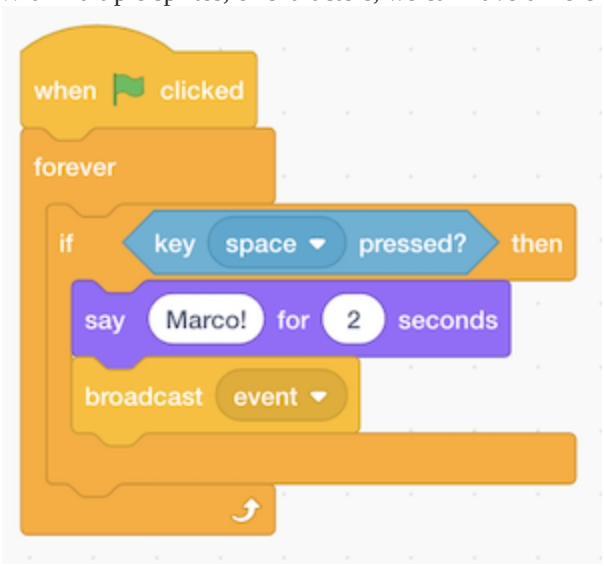
- We try out the Pen extension, by using the "pen down" block with a condition:



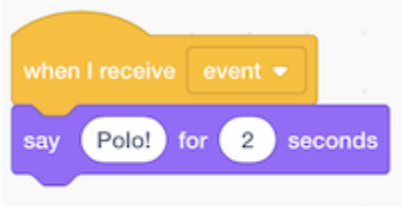
  - Here, we move the cat to the mouse pointer, and if the mouse is clicked, or down, we put the "pen down", which draws. Otherwise, we put the pen up. We repeat this very quickly, over and over again, so we end up with the effect of drawing whenever we have the mouse held down.
- Scratch also has different costumes, or images, that we can use for our characters.
- We'll make a program that can count:

  - Here, counter is a variable, the value of which we can set, use, and change.
- We look at some more programs, like bounce, where the cat moves back and forth on the screen forever, by turning around whenever we're at the edge of the screen.
  - We can improve the animation by having the cat change to a different costumes after every 10 steps in bounce1. Now when we click the green flag to run our program, we see the cat alternate the movement of its legs.
- We can even record our own sounds with our computer's microphone, and play them in our program.
- To build more and more complex programs, we start with each of these simpler features, and layer them atop one another.
- We can also have Scratch meow if we touch it with the mouse pointer, in pet0.

- In bark, we have not one but two programs in the same Scratch project. Both of these programs will run at the same time after the green flag is clicked. One of these will play a sea lion sound if the `muted` variable is set to `false`, and the other will set the `muted` variable from either `true` to `false`, or `false` to `true`, if the space key is pressed.
- Another extension looks at the video as captured by our computer's webcam, and plays the meow sound if the video has motion above some threshold.
- With multiple sprites, or characters, we can have different sets of blocks for each of them:

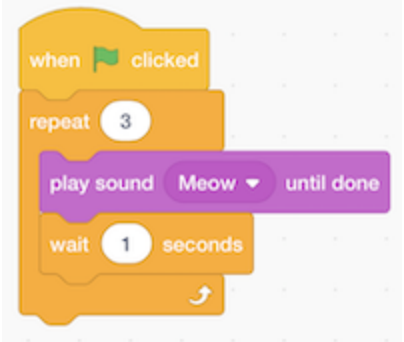


  - For one puppet, we have these blocks that say "Marco!", and then a "broadcast event" block. This "event" is used for our two sprites to communicate with each other, like sending a message behind the scenes. So our other puppet can just wait for this "event" to say "Polo!":



- We can use the Translate extension to say something in other languages, too:

  - Here, the output of the "join" block is use as the input to the "translate" block, the output of which is passed as input to the "say" block.
- Now that we know some basics, we can think about the design, or quality of our programs. For example, we might want to have the cat meow three times with the "repeat" block:



- We can use **abstraction**, which simplifies a more complex concept. In this case, we can define our own "meow" block in Scratch, and reuse it elsewhere in our program, as seen in meow3. The advantage is that we don't need to know how meowing is implemented, or written in code, but rather just use it in our program, making it more readable.
- We can even define a block with an input in meow4, where we have a block that make the cat meow a certain number of times. Now we can reuse that block in our program to meow any number of times, much like how we can use the "translate" or "speak" blocks, without knowing the **implementation details**, or how the block actually works.
- We take a look at a few more demos, including Gingerbread tales remix and Oscartime, both of which combine loops, conditions, and movement to make an interactive game.
- Oscartime was actually made by David many years ago, and he started by adding one sprite, then one feature at a time, and so on, until they added up to the more complicated program.
- A former student, Andrew, created Raining Men. Even though Andrew ultimately ended up not pursuing computer science as a profession, the problem-solving skills, algorithms, and ideas we'll learn in the course are applicable everywhere.
- Until next time!

Problem Set 0

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

What to Do

1. Download and install the latest version of Chrome, if you don't have it already.
2. Submit this problem on Scratch.

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

Here are David's examples from lecture if you'd like to review! To see the source code of each, click **See inside**.

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.
- Your project must use at least one loop.
- Your project must use at least 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:

- It's Raining Men, from lecture
- Ivy's Hardest Game, a game, Harvard edition
- Soccer, a game
- Cookie Love Story, an animation
- Gingerbread tales, an interactive story
- Intersection, a game
- Oscartime, a game

You might find these tutorials or starter projects helpful. And you're welcome to explore scratch.mit.edu 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. If prompted by your computer to **Open** or **Save** the file, be sure to **Save** 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

Be sure to complete **both** steps below, in order!

Step 1 of 2

Submit this form.

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, per the above form.

1. Visit this link, 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/2021/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!

Week 1 C

Lecture 1

- C
- CS50 IDE
- Compiling
- Functions and arguments
- main, header files
- Tools
- Commands
- Types, format codes,
- Operators, limitations, truncation
- Variables, syntactic sugar
- Conditions
- Boolean expressions, loops
- Abstraction
- Mario
- Memory, imprecision, and overflow

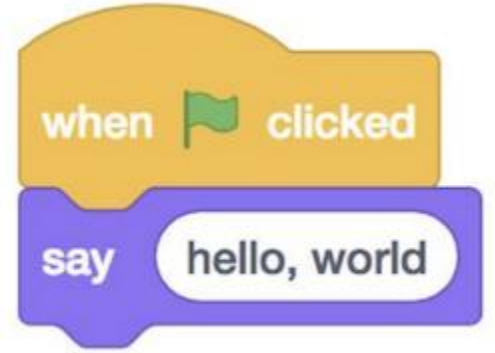
C

- Today we'll learn a new language, **C**: a programming language that has all the features of Scratch and more, but perhaps a little less friendly since it's purely in text:

```
#include <stdio.h>

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

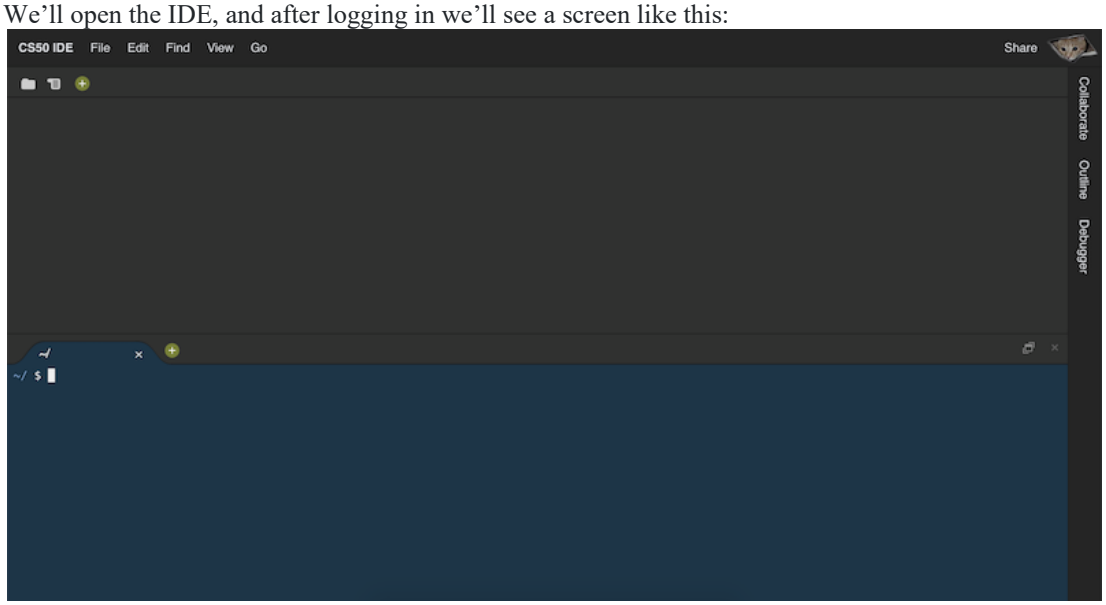
- Though at first, to borrow a phrase from MIT, trying to absorb all these new concepts may feel like drinking from a fire hose, be assured that by the end of the semester we'll be empowered by and experienced at learning and applying these concepts.
- We can compare a lot of the programming features in C to blocks we've already seen and used in Scratch. The details of the syntax are far less important than the ideas, which we've already been introduced to.
- In our example, though the words are new, the ideas are exactly as same as the "when green flag clicked" and "say (hello, world)" blocks in Scratch:



- When writing code, we might consider the following qualities:
  - **Correctness**, or whether our code works correctly, as intended.
  - **Design**, or a subjective measure of how well-written our code is, based on how efficient it is and how elegant or logically readable it is, without unnecessary repetition.
  - **Style**, or how aesthetically formatted our code is, in terms of consistent indentation and other placement of symbols. Differences in style don't affect the correctness or meaning of our code, but affect how readable it is visually.

CS50 IDE

- To start writing our code quickly, we'll use a tool for the course, **CS50 IDE**, an *integrated development environment* which includes programs and features for writing code. CS50 IDE is built atop a popular cloud-based IDE used by general programmers, but with additional educational features and customization.
- We'll open the IDE, and after logging in we'll see a screen like this:



  - The top panel, blank, will contain text files within which we can write our code.
  - The bottom panel, a **terminal** window, will allow us to type in various commands and run them, including programs from our code above.
- Our IDE runs in the cloud and comes with a standard set of tools, but know that there are many desktop-based IDEs as well, offering more customization and control for different programming purposes, at the cost of greater setup time and effort.
- In the IDE, we'll go to File > New File, and then File > Save to save our file as `hello.c`, indicating that our file will be code written in C. We'll see that the name of our tab has indeed changed to `hello.c`, and now we'll paste our code from above:

```
#include <stdio.h>

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

- To run our program, we'll use a CLI, or **command-line interface**, a prompt at which we need to enter text commands. This is in contrast to a **graphical user interface**, or GUI, like Scratch, where we have images, icons, and buttons in addition to text.

Compiling

- In the terminal in the bottom pane of our IDE, we'll **compile** our code before we can run it. Computers only understand binary, which is also used to represent instructions like printing something to the screen. Our **source code** has been written in characters we can read, but it needs to be compiled: converted to **machine code**, patterns of zeros and ones that our computer can directly understand.
- A program called a **compiler** will take source code as input and produce machine code as output. In the CS50 IDE, we have access to a compiler already, through a command called **make**. In our terminal, we'll type in `make hello`, which will automatically find our `hello.c` file with our source code, and compile it into a program called `hello`. There will be some output, but no error messages in yellow or red, so our program compiled successfully.
- To run our program, we'll type in another command, `./hello`, which looks in the current folder, `.`, for a program called `hello`, and runs it.
- The $ in the terminal is an indicator of where the prompt is, or where we can type in more commands.

Functions and arguments

- We'll use the same ideas we've explored in Scratch.
- **Functions** are small actions or verbs that we can use in our program to do something, and the inputs to functions are called **arguments**.
    - For example, the "say" block in Scratch might have taken something like "hello, world" as an argument. In C, the function to print something to the screen is called `printf` (with the `f` standing for "formatted" text, which we'll soon see). And in C, we pass in arguments within parentheses, as in `printf("hello, world");`. The double quotes indicate that we want to print out the letters `hello, world` literally, and the semicolon at the end indicates the end of our line of code.
- Functions can also have two kinds of outputs:
    - **side effects**, such as something printed to the screen,
    - and **return values**, a value that is passed back to our program that we can use or store for later.
        - The "ask" block in Scratch, for example, created an "answer" block.
- To get the same functionality as the "ask" block, we'll use a **library**, or a set of code already written. The CS50 Library will include some basic, simple functions that we can use right away. For example, `get_string` will ask the user for a **string**, or some sequence of text, and return it to our program. `get_string` takes in some input as the prompt for the user, such as `What's your name?`, and we'll have to save it in a variable with:

```
string answer = get_string("What's your name? ");
```

    - In C, the single `=` indicates **assignment**, or setting the value on the right to the variable on the left. And C will call the `get_string` function in order to get its output first.
    - And we also need to indicate that our variable named `answer` has a **type** of string, so our program knows to interpret the zeros and ones as text.
    - Finally, we need to remember to add a semicolon to end our line of code.
- In Scratch, we also used the "answer" block within our "join" and "say" blocks. In C, we'll do this:

```
printf("hello, %s", answer);
```

    - The `%s` is called a **format code**, which just means that we want the `printf` function to substitute a variable where the `%s` placeholder is. And the variable we want to use is `answer`, which we give to `printf` as another argument, separated from the first one with a comma. (`printf("hello, answer")` would literally print out `hello, answer` every time.)
- Back in the CS50 IDE, we'll add what we've discovered:

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

int main(void)
{
    string answer = get_string("What's your name? ");
    printf("hello, %s", answer);
}
```

    - We need to tell the compiler to include the CS50 Library, with `#include <cs50.h>`, so we can use the `get_string` function.
    - We also have an opportunity to use better style here, since we could name our `answer` variable anything, but a more descriptive name will help us understand its purpose better than a shorter name like `a` or `x`.
- After we save the file, we'll need to recompile our program with `make hello`, since we've only changed the source code but not the compiled machine code. Other languages or IDEs may not require us to manually recompile our code after we change it, but here we have the opportunity for more control and understanding of what's happening under the hood.
- Now, `./hello` will run our program, and prompt us for our name as intended. We might notice that the next prompt is printed immediately after our program's output, as in `hello, Brian~/ $`. We can add a new line after our program's output, so the next prompt is on its own line, with `\n`:

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

    - `\n` is an example of an **escape sequence**, or some text that represents some other text.

main, header files

- The "when green flag clicked" block in Scratch starts what we would consider to be the main program. In C, the first line for the same is `int main(void)`, which we'll learn more about over the coming weeks, followed by an open curly brace `{`, and a closed curly brace `}`, wrapping everything that should be in our program.

```
int main(void)
{
```

```
}
```

  - We'll learn more about ways we can modify this line in the coming weeks, but for now we'll simply use this to start our program.
- **Header files** that end with .h refer to some other set of code, like a library, that we can then use in our program. We *include* them with lines like #include <stdio.h>, for example, for the *standard input/output* library, which contains the printf function.

Tools

- With all of the new syntax, it's easy for us to make mistakes or forget something. We have a few tools written by the staff to help us.
- We might forget to include a line of code, and when we try to compile our program, see a lot of lines of error messages that are hard to understand, since the compiler might have been designed for a more technical audience. **help50** is a command we can run to explain problems in our code in a more user-friendly way. We can run it by adding help50 to the front of a command we're trying, like help50 make hello, to get advice that might be more understandable.
- It turns out that, in C, new lines and indentation generally don't affect how our code runs. For example, we can change our main function to be one line, int main(void){printf("hello, world");}, but it's much harder to read, so we would consider it to have bad style. We can run **style50**, as with style50 hello.c, with the name of the file of our source code, to see suggestions for new lines and indentation.
- Additionally, we can add **comments**, notes in our source code for ourselves or other humans that don't affect how our code runs. For example, we might add a line like // Greet user, with two slashes // to indicate that the line is a comment, and then write the purpose of our code or program to help us remember later on.
- **check50** will check the correctness of our code with some automated tests. The staff writes tests specifically for some of the programs we'll be writing in the course, and instructions for using check50 will be included in each problem set or lab as needed. After we run check50, we'll see some output telling us whether our code passed relevant tests.
- The CS50 IDE also gives us the equivalent of our own computer in the cloud, somewhere on the internet, with our own files and folders. If we click the folder icon in the top left, we'll see a file tree, a GUI of the files in our IDE:

  - To open a file, we can just double-click it. hello.c is the source code that we just wrote, and hello itself will have lots of red dots, each of which are unprintable characters since they represent binary instructions for our computers.

Commands

- Since the CS50 IDE is a virtual computer in the cloud, we can also run commands available in Linux, an operating system like macOS or Windows.
- In the terminal, we can type in ls, short for list, to see a list of files and folder in the current folder:

```
~/ $ ls
hello*  hello.c
```

  - hello is in green with an asterisk to indicate that we can run it as a program.
- We can also *remove* files with rm, with a command like rm hello. It will prompt us for a confirmation, and we can respond with y or n for yes or no.
- With mv, or *move*, we can rename files. With mv hello.c goodbye.c, we've renamed our hello.c file to be named goodbye.c.
- With mkdir, or *make directory*, we can create folders, or directories. If we run mkdir lecture, we'll see a folder called lecture, and we can move files into directories with a command like mv hello.c lecture/.

- To *change directories* in our terminal, we can use `cd`, as with `cd lecture/`. Our prompt will change from `~/` to `~/lecture/`, indicating that we're in the `lecture` directory inside `~`. `~` stands for our home directory, or our account's default, top-level folder.
- We can also use `..` as shorthand for the parent, or containing folder. Within `~/lecture/`, we can run `mv hello.c ..` to move it back up to `~`, since it's the parent folder of `lecture/`. `cd ..`, similarly, will change our terminal's directory to the current one's parent. A single dot, `.`, refers to the current directory, as in `./hello`.
- Now that our `lecture/` folder is empty, we can remove it with `rmdir lecture/` as well.

Types, format codes,

- There are many data **types** we can use for our variables, which indicate to the computer what type of data they represent:
  - `bool`, a Boolean expression of either `true` or `false`
  - `char`, a single ASCII character like `a` or `2`
  - `double`, a floating-point value with more digits than a `float`
  - `float`, a floating-point value, or real number with a decimal value
  - `int`, integers up to a certain size, or number of bits
  - `long`, integers with more bits, so they can count higher than an `int`
  - `string`, a string of characters
- And the CS50 library has corresponding functions to get input of various types:
  - `get_char`
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long`
  - `get_string`
- For `printf`, too, there are different placeholders for each type:
  - `%c` for chars
  - `%f` for floats, doubles
  - `%i` for ints
  - `%li` for longs
  - `%s` for strings

Operators, limitations, truncation

- There are several mathematical operators we can use, too:
  - `+` for addition
  - `-` for subtraction
  - `*` for multiplication
  - `/` for division
  - `%` for remainder
- We'll make a new program, `addition.c`:

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

int main(void)
{
    int x = get_int("x: ");

    int y = get_int("y: ");

    printf("%i\n", x + y);
}
```

  - We'll include header files for libraries we know we want to use, and then we'll call `get_int` to get integers from the user, storing them in variables named `x` and `y`.
  - Then, in `printf`, we'll print a placeholder for an integer, `%i`, followed by a new line. Since we want to print out the sum of `x` and `y`, we'll pass in `x + y` for `printf` to substitute in the string.
  - We'll save, run `make addition` in the terminal, and then `./addition` to see our program working. If we type in something that's not an integer, we'll see `get_int` asking us for an integer again. If we type in a really big number, like `4000000000`, `get_int` will prompt us again too. This is because, like on many computer systems, an `int` in CS50 IDE is 32 bits, which can only contain about four billion different values. And since integers

can be positive or negative, the highest positive value for an `int` can only be about two billion, with a lowest negative value of about negative two billion, for a total of about four billion total values.

- We can change our program to use the `long` type:

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

int main(void)
{
    long x = get_long("x: ");

    long y = get_long("y: ");

    printf("%li\n", x + y);
}
```

    - Now we can type in bigger integers, and see a correct result as expected.
- Whenever we get an error while compiling, it's a good idea to scroll up to the top to see the first error and fix that first, since sometimes a mistake early in the program will lead to the rest of the program being interpreted with errors as well.
- Let's look at another example, `truncation.c`:

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

int main(void)
{
    // Get numbers from user
    int x = get_int("x: ");
    int y = get_int("y: ");

    // Divide x by y
    float z = x / y;
    printf("%f\n", z);
}
```

    - We'll store the result of `x` divided by `y` in `z`, a floating-point value, or real number, and print it out as a float too.
    - But when we compile and run our program, we see `z` printed out as whole numbers like `0.000000` or `1.000000`. It turns out that, in our code, `x / y` is divided as two integers *first*, so the result given back by the division operation is an integer as well. The result is **truncated**, with the value after the decimal point lost. Even though `z` is a `float`, the value we're storing in it is already an integer.
    - To fix this, we **cast**, or convert, our integers to floats before we divide them:
    - `float z = (float) x / (float) y;`
    - The result will be a float as we expect, and in fact we can cast only one of `x` or `y` and get a float as well.

Variables, syntactic sugar

- In Scratch, we had blocks like "set [counter] to (0)" that set a **variable** to some value. In C, we would write `int counter = 0;` for the same effect.
- We can increase the value of a variable with `counter = counter + 1;`, where we look at the right side first, taking the original value of `counter`, adding 1, and then storing it into the left side (back into `counter` in this case).
- C also supports **syntactic sugar**, or shorthand expressions for the same functionality. In this case, we could equivalently say `counter += 1;` to add one to `counter` before storing it again. We could also just write `counter++;`, and we can learn this (and other examples) through looking at documentation or other references online.

Conditions

- We can translate conditions, or "if" blocks, with:

```
if (x < y)
{
    printf("x is less than y\n");
}
```

    - Notice that in C, we use { and } (as well as indentation) to indicate how lines of code should be nested.
- We can have “if” and “else” conditions:

```
if (x < y)
{
    printf("x is less than y\n");
}
else
{
    printf("x is not less than y\n");
}
```

- And even “else if”:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else if (x == y)
{
    printf("x is equal to y\n");
}
```

    - Notice that, to compare two values in C, we use ==, two equals signs.
    - And, logically, we don’t need the if (x == y) in the final condition, since that’s the only case remaining, so we can just say else:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

- Let’s take a look at another example, conditions.c:

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

int main(void)
{
    // Prompt user for x
    int x = get_int("x: ");

    // Prompt user for y
    int y = get_int("y: ");

    // Compare x and y
    if (x < y)
    {
        printf("x is less than y\n");
    }
    else if (x > y)
    {
        printf("x is greater than y\n");
    }
    else
```

```
    {
        printf("x is equal to y\n");
    }
}
```

- We've included the conditions we just saw, along with two calls, or uses, of `get_int` to get `x` and `y` from the user.
- We'll compile and run our program to see that it indeed works as intended.

- In agree.c, we can ask the user to confirm or deny something:

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

int main(void)
{
    char c = get_char("Do you agree? ");

    // Check whether agreed
    if (c == 'Y' || c == 'y')
    {
        printf("Agreed.\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("Not agreed.\n");
    }
}
```

- With `get_char`, we can get a single character, and since we only have a single one in our program, it seems reasonable to call it `c`.
- We use two vertical bars, `||`, to indicate a logical "or", whether either expression can be true for the condition to be followed. (Two ampersands, `&&`, indicate a logical "and", where both conditions would have to be true.) And notice that we use two equals signs, `==`, to compare two values, as well as single quotes, `'`, to surround our values of single characters.
- If neither of the expressions are true, nothing will happen since our program doesn't have a loop.

Boolean expressions, loops

- We can translate a "forever" block in Scratch with:

```
while (true)
{
    printf("hello, world\n");
}
```

- The `while` keyword requires a condition, so we use `true` as the Boolean expression to ensure that our loop will run forever. `while` will tell the computer to check whether the expression evaluates to `true`, and then run the lines inside the curly braces. Then it will repeat that until the expression isn't true anymore. In this case, `true` will always be true, so our loop is an **infinite loop**, or one that will run forever.

- We could do something a certain number of times with `while`:

```
int i = 0;
while (i < 50)
{
    printf("hello, world\n");
    i++;
}
```

- We create a variable, `i`, and set it to 0. Then, while `i` is less than 50, we run some lines of code, including one where we add 1 to `i` each time. This way, our loop will eventually end when `i` reaches a value of 50.
- In this case, we're using the variable `i` as a counter, but since it doesn't serve any additional purpose, we can simply name it `i`.

- Even though we *could* do the following and start counting at 1, by convention we should start at 0:

```
int i = 1;
while (i <= 50)
{
    printf("hello, world\n");
```

```
    i++;
}
```

- Another correct, but arguably less well-designed solution might be starting at 50 and counting backwards:

```
int i = 50;
while (i > 0)
{
    printf("hello, world\n");
    i--;
}
```

    - In this case, the logic for our loop is harder to reason about without serving any additional purpose, and might even confuse readers.
- Finally, more commonly we can use the `for` keyword:

```
for (int i = 0; i < 50; i++)
{
    printf("hello, world\n");
}
```

    - Again, first we create a variable named `i` and set it to 0. Then, we check that `i < 50` every time we reach the top of the loop, before we run any of the code inside. If that expression is true, then we run the code inside. Finally, after we run the code inside, we use `i++` to add one to `i`, and the loop repeats.
    - The `for` loop is more elegant than a `while` loop in this case, since everything related to the loop is in the same line, and only the code we actually want to run multiple times is inside the loop.
- Notice that for many of these lines of code, like `if` conditions and `for` loops, we don't put a semicolon at the end. This is just how the language of C was designed, many years ago, and a general rule is that only lines for actions or verbs have semicolons at the end.

Abstraction

- We can write a program that prints `meow` three times:

```
#include <stdio.h>

int main(void)
{
    printf("meow\n");
    printf("meow\n");
    printf("meow\n");
}
```

- We could use a `for` loop, so we don't have to copy and paste so many lines:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("meow\n");
    }
}
```

- We can move the `printf` line to its own function, like our own puzzle piece:

```
#include <stdio.h>

void meow(void) {
  printf("meow\n");
}

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
```

```
}
```

- We defined a function, `meow`, above our `main` function.

- But conventionally, our `main` function should be the first function in our program, so we need a few more lines:

```
#include <stdio.h>

void meow(void);

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}

void meow(void)
{
    printf("meow\n");
}
```

- It turns out that we need to declare our `meow` function first with a **prototype**, before we use it in `main`, and actually define it after. The compiler reads our source code from top to bottom, so it needs to know that `meow` will exist later in the file.

- We can even change our `meow` function to take in some input, `n`, and meow `n` times:

```
#include <stdio.h>

void meow(int n);

int main(void)
{
    meow(3);
}

void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

- The `void` before the `meow` function means that it doesn't return a value, and likewise in `main` we can't do anything with the result of `meow`, so we just call it.

- The abstraction here leads to better design, since we now have the flexibility to reuse our `meow` function in multiple places in the future.

- Let's look at another example of abstraction, `get_positive_int.c`:

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

int get_positive_int(void);

int main(void)
{
    int i = get_positive_int();
    printf("%i\n", i);
}

// Prompt user for positive integer
int get_positive_int(void)
{
    int n;
```

```
    do
    {
        n = get_int("Positive Integer: ");
    }
    while (n < 1);
    return n;
}
```

- We have our own function that calls `get_int` repeatedly until we have some integer that's *not* less than 1. With a do-while loop, our program will do something first, then check some condition, and repeat while the condition is true. A while loop, on the other hand, will check the condition first.
- We need to declare our integer `n` outside the do-while loop, since we need to use it after the loop ends. The **scope** of a variable in C refers to the context, or lines of code, within which it exists. In many cases, this will be the curly braces surrounding the variable.
- Notice that the function `get_positive_int` now starts with `int`, indicating that it has a return value of type `int`, and in `main` we indeed store it in `i` after calling `get_positive_int()`. In `get_positive_int`, we have a new keyword, `return`, to return the value `n` to wherever the function was called.

Mario

- We might want a program that prints part of a screen from a video game like Super Mario Bros. In `mario.c`, we can print four question marks, simulating blocks:

```
#include <stdio.h>

int main(void)
{
    printf("????\n");
}
```

- With a loop, we can print a number of question marks, following them with a single new line after the loop:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 4; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

- We can get a positive integer from the user, and print out that number of question marks, by using `n` for our loop:

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

int main(void)
{
    // Get positive integer from user
    int n;
    do
    {
        n = get_int("Width: ");
    }
    while (n < 1);

    // Print out that many question marks
    for (int i = 0; i < n; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

- And we can print a two-dimensional set of blocks with nested loops, one inside the other:

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

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

    - We have two nested loops, where the outer loop uses `i` to do everything inside 3 times, and the inner loop uses `j`, a different variable, to do something 3 times for each of *those* times. In other words, the outer loop prints 3 "rows", or lines, ending each of them with a new line, and the inner loop prints 3 "columns", or `#` characters, *without* a new line.

Memory, imprecision, and overflow

- Our computer has memory, in hardware chips called RAM, random-access memory. Our programs use that RAM to store data while they're running, but that memory is finite.
- With `imprecision.c`, we can see what happens when we use floats:

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

int main(void)
{
    float x = get_float("x: ");
    float y = get_float("y: ");

    printf("%.50f\n", x / y);
}
```

    - With `%.50f`, we can specify the number of decimal places displayed.
    - Hmm, now we get …

```
x: 1
y: 10
0.10000000149011611938476562500000000000000000000000
```

    - It turns out that this is called **floating-point imprecision**, where we don't have enough bits to store all possible values. With a finite number of bits for a `float`, we can't represent all possible real numbers (of which there are an *infinite* number of), so the computer has to store the closest value it can. And this can lead to problems where even small differences in value add up, unless the programmer uses some other way to represent decimal values as accurately as needed.
- Last week, when we had three bits and needed to count higher than seven (or `111`), we added another bit to get eight, `1000`. But if we only had three bits available, we wouldn't have a place for the extra `1`. It would disappear and we would be back at `000`. This problem is called **integer overflow**, where an integer can only be so big before it runs out of bits.
- The Y2K problem arose because many programs stored the calendar year with just two digits, like 98 for 1998, and 99 for 1999. But when the year 2000 approached, the programs had to store only 00, leading to confusion between the years 1900 and 2000.
- In 2038, we'll also run out of bits to track time, since many years ago some humans decided to use 32 bits as the standard number of bits to count the number of seconds since January 1st, 1970. But with 32 bits representing only positive numbers, we can only count up to about four billion, and in 2038 we'll reach that limit unless we upgrade the software in all of our computer systems.

Lab 1

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

What to Do

1. Go to ide.cs50.io and click "Log in" to access your CS50 IDE.
2. Submit Hello
3. Submit Population

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Hello

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 and click "Log in" 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` inside of 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 `~` always denotes your home directory and `~/pset1` denotes a directory called `pset1`, which is inside of `~`.

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, 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/2021/x/hello
```

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

```
style50 hello.c
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/problems/2021/x/hello
```

Population Growth

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

Determine how long it takes for a population to reach a particular size.

```
$ ./population
Start size: 100
End size: 200
Years: 9
```

Background

Say we have a population of `n` llamas. Each year, `n / 3` new llamas are born, and `n / 4` llamas pass away.

For example, if we were to start with `n = 1200` llamas, then in the first year, `1200 / 3 = 400` new llamas would be born and `1200 / 4 = 300` llamas would pass away. At the end of that year, we would have `1200 + 400 - 300 = 1300` llamas.

To try another example, if we were to start with `n = 1000` llamas, at the end of the year, we would have `1000 / 3 = 333.33` new llamas. We can't have a decimal portion of a llama, though, so we'll truncate the decimal to get `333` new llamas born. `1000 / 4 = 250` llamas will pass away, so we'll end up with a total of `1000 + 333 - 250 = 1083` llamas at the end of the year.

Getting Started

- Copy the "distribution code" (i.e., starter code) from cdn.cs50.net/2020/fall/labs/1/population.c into a new file in your IDE called `population.c`.

Implementation Details

Complete the implementation of `population.c`, such that it calculates the number of years required for the population to grow from the start size to the end size.

- Your program should first prompt the user for a starting population size.
  - If the user enters a number less than 9 (the minimum allowed population size), the user should be re-prompted to enter a starting population size until they enter a number that is greater than or equal to 9. (If we start with fewer than 9 llamas, the population of llamas will quickly become stagnant!)
- Your program should then prompt the user for an ending population size.
  - If the user enters a number less than the starting population size, the user should be re-prompted to enter an ending population size until they enter a number that is greater than or equal to the starting population size. (After all, we want the population of llamas to grow!)
- Your program should then calculate the (integer) number of years required for the population to reach at least the size of the end value.
- Finally, your program should print the number of years required for the llama population to reach that end size, as by printing to the terminal `Years: n`, where `n` is the number of years.

Walkthrough

Hints

- If you want to repeatedly re-prompt the user for the value of a variable until some condition is met, you might want to use a `do ... while` loop. For example, recall the following code from lecture, which prompts the user repeatedly until they enter a positive integer.

```
int n;
do
{
    n = get_int("Positive Integer: ");
}
while (n < 1);
```

How might you adapt this code to ensure a start size of at least 9, and an end size of at least the start size?

- To declare a new variable, be sure to specify its data type, a name for the variable, and (optionally) what its initial value should be.
    - For example, you might want to create a variable to keep track of how many years have passed.
- To calculate how many years it will take for the population to reach the end size, another loop might be helpful! Inside the loop, you'll likely want to update the population size according to the formula in the Background, and update the number of years that have passed.
- To print an integer `n` to the terminal, recall that you can use a line of code like
- ```
  printf("The number is %i\n", n);
  ```

  to specify that the variable `n` should fill in for the placeholder `%i`.

How to Test Your Code

Your program should behave per the examples below.

```
$ ./population
Start size: 1200
End size: 1300
Years: 1
$ ./population
Start size: -5
Start size: 3
Start size: 9
End size: 5
End size: 18
Years: 8
$ ./population
Start size: 20
End size: 1
End size: 10
End size: 100
Years: 20
$ ./population
Start size: 100
End size: 1000000
Years: 115
```

Not sure how to solve?

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/labs/2021/x/population
```

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

```
style50 population.c
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/population
```
```

Problem Set 1

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 1** (both the Hello and Population problems) before beginning this problem set.

1. Log into Ed via this link and then reply to this thread to say hello (and, if you'd like, a little about yourself!) to other students and staff. If you don't yet have an account, you should first sign up for Ed.
2. Go to ide.cs50.io and click "Log in" to access your CS50 IDE.
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

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

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!

Mario-less

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. Odds are your pseudocode will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

Spoiler

There's more than one way to do this, so here's just one!

1. Prompt user for height
2. If height is less than 1 or greater than 8 (or not an integer at all), go back one step
3. Iterate from 1 through height:
    1. On iteration *i*, print *i* hashes and then a newline

It's okay to edit your own after seeing this pseudocode here, but don't simply copy/paste ours into your own!

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

- Recall that you can compile your program with `make`.
- Recall that you can print an `int` with `printf` using `%i`.
- Recall that you can get an integer from the user with `get_int`.
- Recall that `get_int` is declared in `cs50.h`.
- Recall that we prompted the user for a positive integer in class via `positive.c`.

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

- Keep in mind that a hash is just a character like any other, so you can print it with `printf`.
- Just as Scratch has a Repeat block, so does C have a `for` loop, via which you can iterate some number times. Perhaps on each iteration, *i*, you could print that many hashes?
- You can actually "nest" loops, iterating with one variable (e.g., `i`) in the "outer" loop and another (e.g., `j`) in the "inner" loop. For instance, here's how you might print a square of height and width `n`, below. Of course, it's not a square that you want to print!

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        printf("#");
    }
    printf("\n");
}
```

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

Notice how the number of dots needed on each line is the "opposite" of the number of that line's hashes. For a pyramid of height 8, like the above, the first line has but 1 hash and thus 7 dots. The bottom line, meanwhile, has 8 hashes and thus 0 dots. Via what formula (or arithmetic, really) could you print that many dots?

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/2021/x/mario/less
```

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

```
style50 mario.c
```

Hint

A space is just a press of your space bar, just as a period is just a press of its key! Just remember that `printf` requires that you surround both with double quotes!

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/2021/x/mario/less
```

Mario-more

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 (i.e., folder) 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/2021/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/2021/x/mario/more
```

Cash

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 `0.2000000029802322387695312 50`) 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/2021/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/2021/x/cash
```

Credit

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!

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 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/2021/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/2021/x/credit
```

Week 2 Arrays

Lecture 2

Compiling

- Last time, we learned to write our first program in C, printing "hello, world" to the screen.
- We compiled it with `make hello` first, turning our source code into machine code before we could run the compiled program with `./hello`.
- `make` is actually just a program that calls `clang`, a compiler, with options. We could compile our source code file, `hello.c`, ourselves by running the command `clang hello.c`. Nothing seems to happen, which means there were no errors. And if we run `ls`, now we see an `a.out` file in our directory. The filename is still the default, so we can actually run a more specific command: `clang -o hello hello.c`.
- We've added another **command-line argument**, or an input to a program on the command-line as extra words after the program's name. `clang` is the name of the program, and `-o`, `hello`, and `hello.c` are additional arguments. We're telling `clang` to use `hello` as the *output* filename, and use `hello.c` as the source code. Now, we can see `hello` being created as output.
- If we wanted to use CS50's library, via `#include <cs50.h>`, for the `get_string` function, we also have to add a flag: `clang -o hello hello.c -lcs50`:

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


int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

    - The `-l` flag links the `cs50` file, which is already installed in the CS50 IDE, and includes the machine code for `get_string` (among other functions) that our program can then refer to and use as well.
- With `make`, these arguments are generated for us since the staff has configured `make` in the CS50 IDE already as well.
- Compiling source code into machine code is actually made up of smaller steps:
    - preprocessing
    - compiling
    - assembling
    - linking
- **Preprocessing** generally involves lines that start with a `#`, like `#include`. For example, `#include <cs50.h>` will tell `clang` to look for that header file, since it contains content that we want to include in our program. Then, `clang` will essentially replace the contents of those header files into our program.
- For example …

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

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

- … will be preprocessed into:
- ...

```
string get_string(string prompt);
int printf(string format, ...);
...

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- This includes the prototypes of all the functions from those libraries we included, so we can then use them in our code.
- **Compiling** takes our source code, in C, and converts it to another type of source code called **assembly code**, which looks like this:

```
...
main:                   # @main
    .cfi_startproc
# BB#0:
    pushq   %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    xorl    %eax, %eax
    movl    %eax, %edi
    movabsq    $.L.str, %rsi
    movb    $0, %al
    callq   get_string
    movabsq    $.L.str.1, %rdi
    movq    %rax, -8(%rbp)
    movq    -8(%rbp), %rsi
    movb    $0, %al
    callq   printf
    ...
```

  - These instructions are lower-level and is closer to the binary instructions that a computer's processor can directly understand. They generally operate on bytes themselves, as opposed to abstractions like variable names.
- The next step is to take the assembly code and translate it to instructions in binary by **assembling** it. The instructions in binary are called **machine code**, which a computer's CPU can run directly.
- The last step is **linking**, where previously compiled versions of libraries that we included earlier, like `cs50.c`, are actually combined with the binary of our program. So we end up with one binary file, `a.out` or `hello`, that is the combined machine code for `hello.c`, `cs50.c`, and `stdio.c`. (In the CS50 IDE, precompiled machine code for `cs50.c` and `stdio.c` has already been installed, and `clang` has been configured to find and use them.)
- These four steps have been abstracted away, or simplified, by `make`, so all we have to implement is the code for our programs.

Debugging

- **Bugs** are mistakes or problems in programs that cause them to behave differently than intended. And debugging is the process of finding and fixing those bugs.
- Last week, we learned about a few tools that help us with writing code that compiles, has good style, and is correct:
  - `help50`
  - `style50`
  - `check50`
- We can use another "tool", the `printf` function, to print messages and variables to help us debug.
- Let's take a look at `buggy0.c`:

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

```
int main(void)
{
   // Print 10 hashes
   for (int i = 0; i <= 10; i++)
   {
      printf("#\n");
   }
}
```

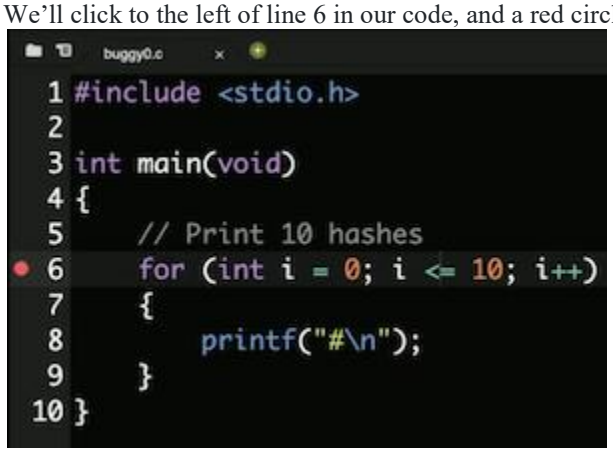
- Hmm, we want to print only 10 #s, but there are 11. If we didn't know what the problem is (since our program is compiling without any errors, and we now have a logical error), we could add another `printf` temporarily:

```
#include <stdio.h>

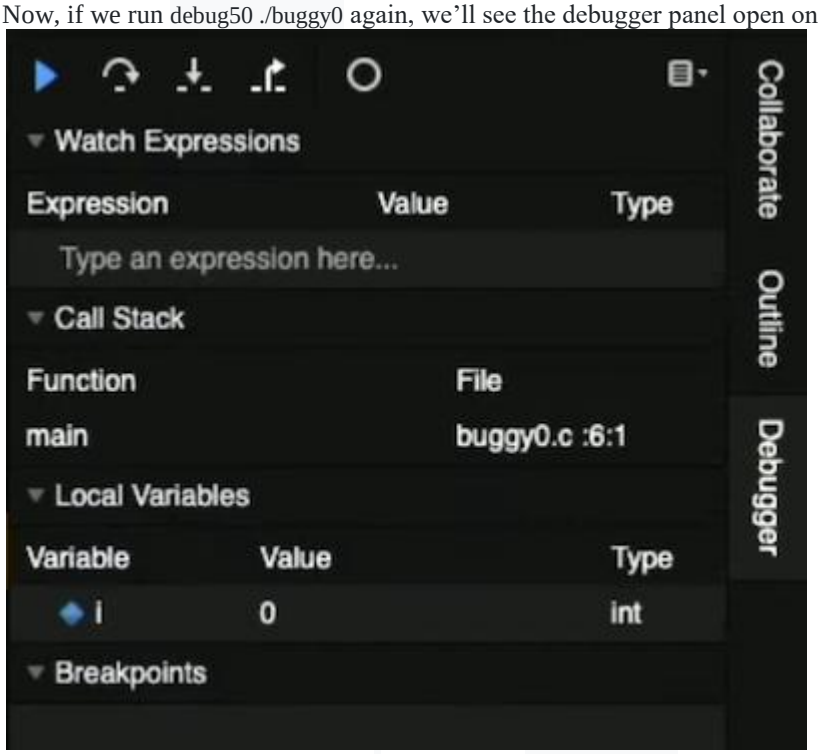
int main(void)
{
   for (int i = 0; i <= 10; i++)
   {
      printf("i is now %i\n", i);
      printf("#\n");
   }
}
```

- Now, we can see that `i` started at 0 and continued until it was 10, but we should have our `for` loop stop once it's at 10, with `i < 10` instead of `i <= 10`.

- In the CS50 IDE, we have another tool, **debug50**, to help us debug programs. This is a tool written by staff that's built on a standard tool called `gdb`. Both of these **debuggers** are programs that will run our own programs step-by-step and let us look at variables and other information while our program is running.
- We'll run the command `debug50 ./buggy0`, and it will tell us to recompile our program since we changed it. Then, it'll tell us to add a **breakpoint**, or indicator for a line of code where the debugger should pause our program.
  - By using the up and down keys in the terminal, we can reuse commands from the past without typing them again.
- We'll click to the left of line 6 in our code, and a red circle will appear:



- Now, if we run `debug50 ./buggy0` again, we'll see the debugger panel open on the right:



- We see that the variable we made, `i`, is under the `Local Variables` section, and see that there's a value of `0`.
- Our breakpoint has paused our program on line 6, highlighting that line in yellow. To continue, we have a few controls in the debugger panel. The blue triangle will continue our program until we reach another breakpoint or the end of our program. The curved arrow to its right, Step Over, will "step over" the line, running it and pausing our program again immediately after.
- So, we'll use the curved arrow to run the next line, and see what changes after. We're at the `printf` line, and pressing the curved arrow again, we see a single # printed to our terminal window. With another click of the arrow, we see the value of `i` change to `1`. We can keep clicking the arrow to watch our program run, one line at a time.
- To exit the debugger, we can press `control + C` to stop the running program.
- Let's look at another example, `buggy1.c`:

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

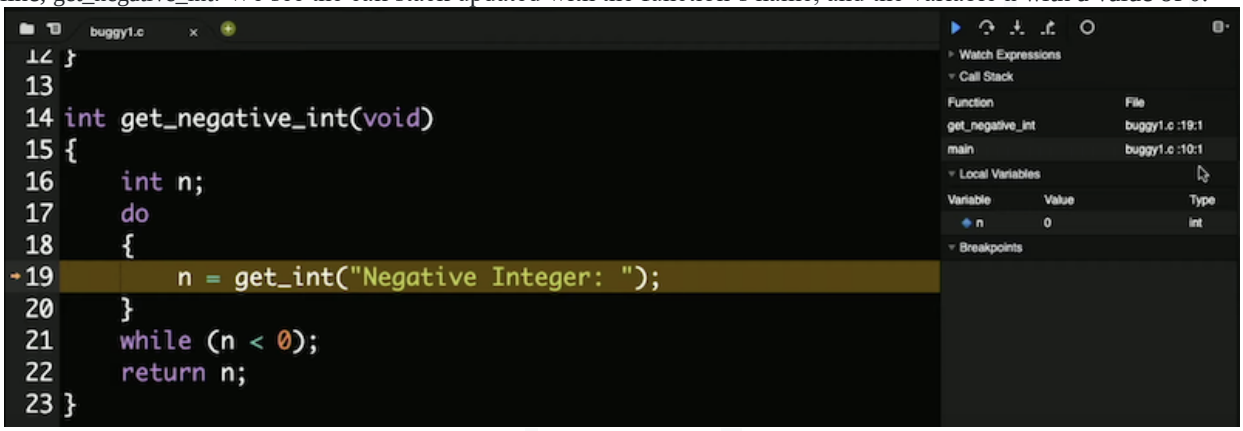
// Prototype
int get_negative_int(void);

int main(void)
{
    // Get negative integer from user
    int i = get_negative_int();
    printf("%i\n", i);
}

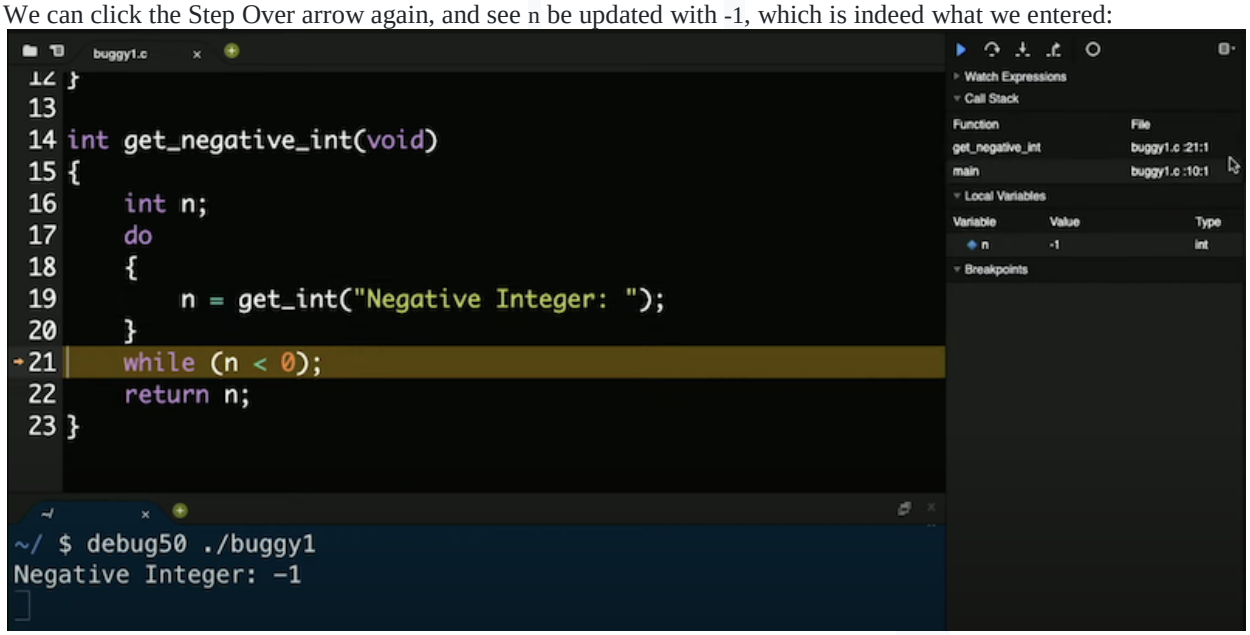
int get_negative_int(void)
{
    int n;
    do
    {
        n = get_int("Negative Integer: ");
    }
    while (n < 0);
    return n;
}
```

    - We've implemented another function, `get_negative_int`, to get a negative integer from the user. We need to remember the prototype before our `main` function, and then our code compiles.

- But when we run our program, it keeps asking us for a negative integer, even after we provide one. We'll set a breakpoint on line 10, `int i = get_negative_int();`, since it's the first interesting line of code. We'll run `debug50 ./buggy1`, and see in the debugging panel's Call Stack section that we're in the `main` function. (The "call stack" refers to all the functions that have been called in our program at the time, and not yet returned from. So far, only the `main` function has been called.)
- We'll click the arrow pointing down, Step Into, and the debugger brings us *into* the function called on that line, `get_negative_int`. We see the call stack updated with the function's name, and the variable `n` with a value of `0`:



- We can click the Step Over arrow again, and see `n` be updated with `-1`, which is indeed what we entered:



- We click Step Over again, and we see our program going back inside the loop. Our `while` loop is still running, so the condition that it checks must be `true` still. And we do see that `n < 0` is true even if we entered a negative integer, so we should fix our bug by changing it to `n >= 0`.
- We can save lots of time in the future by investing a little bit now to learn how to use `debug50`!
- We can also use `ddb`, short for "duck debugger", a real technique where we explain what we're trying to do to a rubber duck, and oftentimes we'll realize our own mistake in logic or implementation as we're explaining it.

Memory

- In C, we have different types of variables we can use for storing data, and each of them take up a fixed amount of space. Different computer systems actually vary in the amount of space actually used for each type, but we'll work with the amounts here, as used in the CS50 IDE:
  - `bool` 1 byte
  - `char` 1 byte
  - `double` 8 bytes
  - `float` 4 bytes
  - `int` 4 bytes
  - `long` 8 bytes
  - `string` ? bytes
  - …

- Inside our computers, we have chips called RAM, random-access **memory**, that stores data for short-term use, like a program's code while it's running, or a file while it's open. We might save a program or file to our hard drive (or SSD, solid state drive) for long-term storage, but use RAM because it is much faster. However, RAM is volatile, or requires power to keep data stored.
- We can think of bytes stored in RAM as though they were in a grid:

  - In reality, there are millions or billions of bytes per chip.
  - Each byte will have a location on the chip, like the first byte, second byte, and so on.
- In C, when we create a variable of type `char`, which will be sized one byte, it will physically be stored in one of those boxes in RAM. An integer, with 4 bytes, will take up four of those boxes.

Arrays

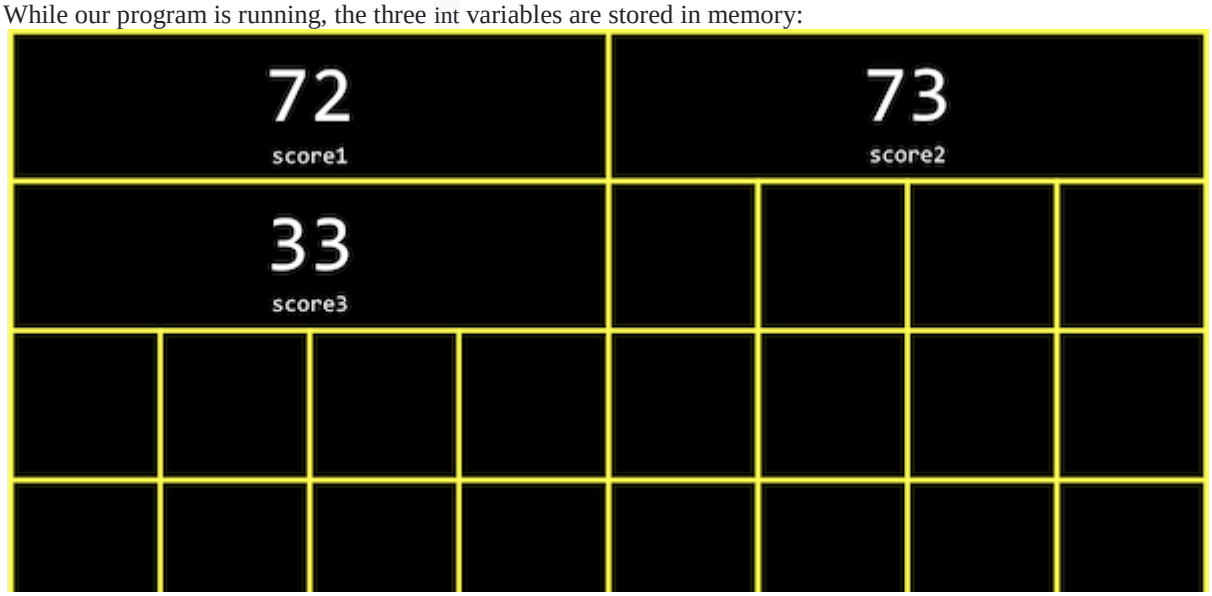
- Let's say we wanted to take the average of three variables:

```c
#include <stdio.h>

int main(void)
{
    int score1 = 72;
    int score2 = 73;
    int score3 = 33;

    printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
}
```

  - We divide by not `3`, but `3.0` so the result is also a float.
  - We can compile and run our program, and see an average printed.
- While our program is running, the three `int` variables are stored in memory:



    - Each `int` takes up four boxes, representing four bytes, and each byte in turn is made up of eight bits, 0s and 1s stored by electrical components.
- It turns out, in memory, we can store variables one after another, back-to-back, and access them more easily with loops. In C, a list of values stored one after another contiguously is called an **array**.
- For our program above, we can use `int scores[3];` to declare an array of three integers instead.
- And we can assign and use variables in an array with `scores[0] = 72`. With the brackets, we're indexing into, or going to, the "0th" position in the array. Arrays are zero-indexed, meaning that the first value has index 0, and the second value has index 1, and so on.
- Let's update our program to use an array:

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

int main(void)
{
    int scores[3];
    scores[0] = get_int("Score: ");
    scores[1] = get_int("Score: ");
    scores[2] = get_int("Score: ");

    // Print average
    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

    - Now, we're asking the user for three values, and printing the average as before, but using the values stored in the array.
- Since we can set and access items in an array based on their position, and that position can *also* be the value of some variable, we can use a loop:

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

int main(void)
{
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
      scores[i] = get_int("Score: ");
    }

    // Print average
    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

    - Now, instead of hard-coding, or manually specifying each element three times, we use a `for` loop and `i` as the index of each element in the array.
- And we repeated the value 3, representing the length of our array, in two different places. So we can use a **constant**, or variable with a fixed value in our program:

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

const int TOTAL = 3;

int main(void)
{
    int scores[TOTAL];
    for (int i = 0; i < TOTAL; i++)
    {
      scores[i] = get_int("Score: ");
    }

    printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / TOTAL);
}
```

    - We can use the `const` keyword to tell the compiler that the value of `TOTAL` should never be changed by our program. And by convention, we'll place our declaration of the variable outside of the `main` function and capitalize its name, which isn't necessary for the compiler but shows other humans that this variable is a constant and makes it easy to see from the start.
    - But now our average will be incorrect or broken if we don't have exactly three values.
- Let's add a function to calculate the average:

```
float average(int length, int array[])
{
    int sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }
    return sum / (float) length;
}
```

    - We'll pass in the length and an array of `int`s (which could be any size), and use another loop inside our helper function to add up the values into a `sum` variable. We use `(float)` to cast `length` into a float, so the result we get from dividing the two is also a float.
    - Now, in our `main` function, we can call our new `average` function with `printf("Average: %f\n", average(TOTAL, scores);`. Notice that the names of the variables in `main` don't need to match what `average` calls them, since only the *values* are passed in.
    - We need to pass in the length of the array to the `average` function, so it knows how many values there are.

Characters

- We can print out a single character with a simple program:

```
#include <stdio.h>

int main(void)
{
    char c = '#';

    printf("%c\n", c);
}
```

    - When we run this program, we get `#` printed in the terminal.
- Let's see what happens if we change our program to print `c` as an integer:

```
#include <stdio.h>

int main(void)
{
    char c = '#';

    printf("%i\n", (int) c);
}
```

    - When we run this program, we get `35` printed. It turns out that `35` is indeed the ASCII code for a # symbol.
    - In fact, we don't need to cast `c` to an `int` explicitly; the compiler can do that for us in this case.
- A `char` is a single byte, so we can picture it as being stored in one box in the grid of memory above.

Strings

- We can print out a string, or some text, by creating a variable for each character and printing them out:

```
#include <stdio.h>

int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

```

```
    printf("%c%c%c\n", c1, c2, c3);
}
```

- Here, we'll see `HI!` printed out.

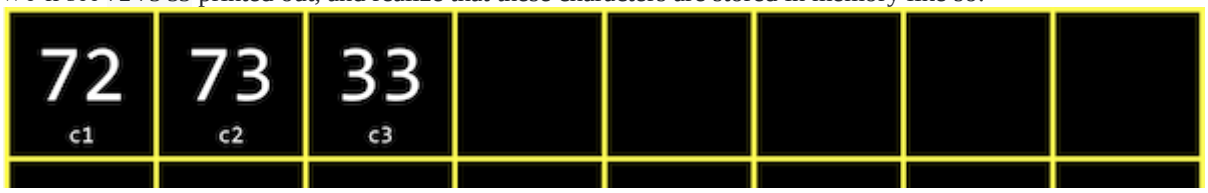
- Now let's print out the integer values of each character:

```
#include <stdio.h>

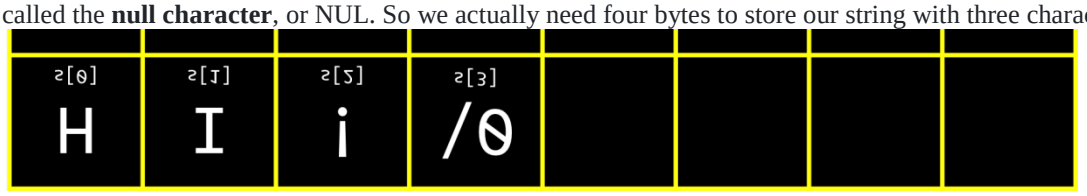
int main(void)
{
    char c1 = 'H';
    char c2 = 'I';
    char c3 = '!';

    printf("%i %i %i\n", c1, c2, c3);
}
```

- We'll see `72 73 33` printed out, and realize that these characters are stored in memory like so:



- **Strings** are actually just arrays of characters, and defined not in C but by the CS50 library. If we had an array called `s`, each character can be accessed with `s[0]`, `s[1]`, and so on.
- And it turns out that a string ends with a special character, `'\0'`, or a byte with all bits set to 0. This character is called the **null character**, or NUL. So we actually need four bytes to store our string with three characters:



- We can use a string as an array in our program, and print out the ASCII codes, or integer values, of each character in the string:

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

int main(void)
{
    string s = "HI!";
    printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);
}
```

- And as we might have expected, we see `72 73 33 0` printed.
- In fact, we could try to access `s[4]`, and see some unexpected symbol printed. With C, our code has the ability to access or change memory that it otherwise shouldn't, which is both powerful and dangerous.

- We can use a loop to print out every character in a string:

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

int main(void)
{
    string s = get_string("Input:  ");
    printf("Output: ");
    for (int i = 0; s[i] != '\0'; i++)
    {
        printf("%c", s[i]);
    }
    printf("\n");
}
```

- We can change our loop's condition to continue regardless of what `i` is, but rather only when `s[i] != '\0'`, or when the character at the current position in `s` *isn't* the null character.
- We can use a function that comes with C's `string` library, `strlen`, to get the length of the string for our loop:

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

int main(void)
{
    string s = get_string("Input:  ");
    printf("Output: ");
    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c", s[i]);
    }
    printf("\n");
}
```

- We have an opportunity to improve the design of our program. Our loop was a little inefficient, since we check the length of the string, after each character is printed, in our condition. But since the length of the string doesn't change, we can check the length of the string once:

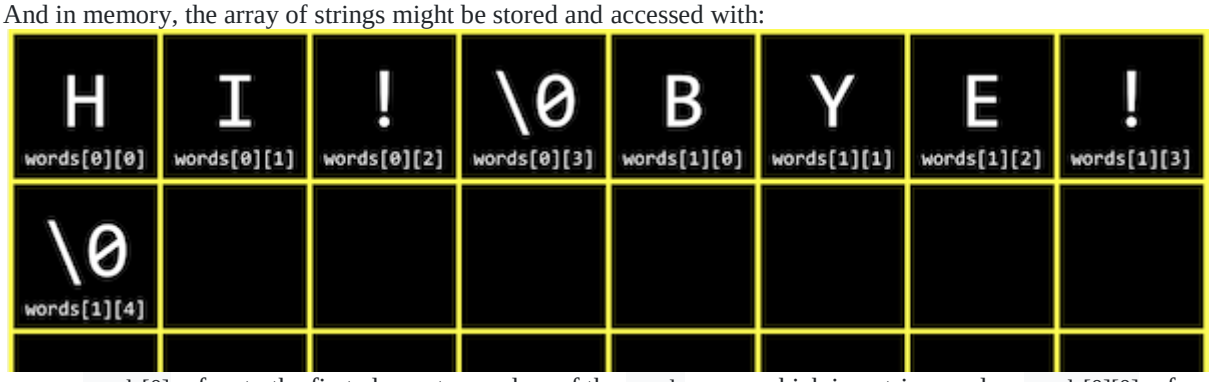
```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("Input: ");
    printf("Output:\n");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c\n", s[i]);
    }
}
```

  - Now, at the start of our loop, we initialize both an `i` and `n` variable, and remember the length of our string in `n`. Then, we can check the values without having to call `strlen` to calculate the length of the string each time.
  - And we did need to use a little more memory to store `n`, but this saves us some time with not having to check the length of the string each time.
- We might declare an array of two strings:

```
string words[2];
words[0] = "HI!";
words[1] = "BYE!";
```

- And in memory, the array of strings might be stored and accessed with:



  - `words[0]` refers to the first element, or value, of the `words` array, which is a string, and so `words[0][0]` refers to the first element in that string, which is a character.
  - So an array of strings is just an array of arrays of characters.
- We can now combine what we've seen, to write a program that can capitalize letters:

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

int main(void)
```

```
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            printf("%c", s[i] - 32);
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

- First, we get a string `s` from the user. Then, for each character in the string, if it's lowercase (which means it has a value between that of `a` and `z`), we convert it to uppercase. Otherwise, we just print it.
- We can convert a lowercase letter to its uppercase equivalent by subtracting the difference between their ASCII values. (We know that lowercase letters have a higher ASCII value than uppercase letters, and the difference is the same between the same letters, so we can subtract to get an uppercase letter from a lowercase letter.)

- It turns out that there's another library, `ctype.h`, that we can use:

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

int main(void)
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (islower(s[i]))
        {
            printf("%c", toupper(s[i]));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

- Now, our code is more readable and likely to be correct, since others have written and tested these functions for us.

- We can simplify even further, and just pass in each character to `toupper`, since it doesn't change non-lowercase characters:

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

int main(void)
{
    string s = get_string("Before: ");
    printf("After:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
```

```
    {
        printf("%c", toupper(s[i]));
    }
    printf("\n");
}
```

- We can use CS50's **manual pages** to find and learn about common library functions. From searching the man pages, we see `toupper()` is a function, among others, from a library called `ctype`, that we can use.

Command-line arguments

- Programs of our own can also take in command-line arguments, or words added after our program's name in the command itself.
- In `argv.c`, we change what our `main` function looks like:

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

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

  - `argc` and `argv` are two variables that our `main` function will now get automatically when our program is run from the command line. `argc` is the *argument count*, or number of arguments, and `argv`, *argument vector* (or argument list), an array of strings.
  - The first argument, `argv[0]`, is the name of our program (the first word typed, like `./hello`). In this example, we check if we have two arguments, and print out the second one if so.
  - For example, if we run `./argv David`, we'll get `hello, David` printed, since we typed in `David` as the second word in our command.
- We can print out each character individually, too:

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

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        for (int i = 0, n = strlen(argv[1]); i < n; i++)
        {
            printf("%c\n", argv[1][i]);
        }
    }
}
```

  - We'll use `argv[1][i]` to access each character in the first argument to our program.
- It turns out that our `main` function also returns an integer value. By default, our `main` function returns `0` to indicate nothing went wrong, but we can write a program to return a different value:

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

int main(int argc, string argv[])
{
    if (argc != 2)
    {
```

```
        printf("missing command-line argument\n");
        return 1;
    }
    printf("hello, %s\n", argv[1]);
    return 0;
}
```

    - The return value of `main` in our program is called an **exit code**, usually used to indicate error codes. (We'll write `return 0` explicitly at the end of our program here, even though we don't technically need to.)
- As we write more complex programs, error codes like this can help us determine what went wrong, even if it's not visible or meaningful to the user

Applications

- Now that we know how to work with strings in our programs, as well code written by others in libraries, we can analyze paragraphs of text for their level of readability, based on factors like how long and complicated the words and sentences are.
- **Cryptography** is the art of scrambling, or hiding information. If we wanted to send a message to someone, we might want to **encrypt**, or somehow scramble that message so that it would be hard for others to read. The original message, or input to our algorithm, is called **plaintext**, and the encrypted message, or output, is called **ciphertext**. And the algorithm that does the scrambling is called a **cipher**. A cipher generally requires another input in addition to the plaintext. A **key**, like a number, is some other input that is kept secret.
- For example, if we wanted to send a message like `I L O V E Y O U`, we can first convert it to ASCII: `73 76 79 86 69 89 79 85`. Then, we can encrypt it with a key of just `1` and a simple algorithm, where we just add the key to each value: `74 77 80 87 70 90 80 86`. Then, the ciphertext after we convert the values back to ASCII would be `J M P W F Z P V`. To decrypt this, someone would have to know the key is `1`, and to subtract it from each character!
- We'll apply these concepts in our sections and problem set!

Lab 2: Scrabble

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

Determine which of two Scrabble words is worth more.

```
$ ./scrabble
Player 1: COMPUTER
Player 2: science
Player 1 wins!
```

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Background

In the game of Scrabble, players create words to score points, and the number of points is the sum of the point values of each letter in the word.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	3	3	2	1	4	2	4	1	8	5	1	3	1	1	3	10	1	1	1	1	4	4	8	4	10

For example, if we wanted to score the word Code, we would note that in general Scrabble rules, the C is worth 3 points, the o is worth 1 point, the d is worth 2 points, and the e is worth 1 point. Summing these, we get that Code is worth 3 + 1 + 2 + 1 = 7 points.

Getting Started

- Copy the "distribution code" (i.e., starter code) from cdn.cs50.net/2020/fall/labs/2/scrabble.c into a new file in your IDE called scrabble.c.
    - You can also download the distribution code by running the command wget https://cdn.cs50.net/2020/fall/labs/2/scrabble.c in CS50 IDE.

Implementation Details

Complete the implementation of scrabble.c, such that it determines the winner of a short scrabble-like game, where two players each enter their word, and the higher scoring player wins.

- Notice that we've stored the point values of each letter of the alphabet in an integer array named POINTS.
    - For example, A or a is worth 1 point (represented by POINTS[0]), B or b is worth 3 points (represented by POINTS[1]), etc.
- Notice that we've created a prototype for a helper function called compute_score() that takes a string as input and returns an int. Whenever we would like to assign point values to a particular word, we can call this function. Note that this prototype is required for C to know that compute_score() exists later in the program.
- In main(), the program prompts the two players for their words using the get_string() function. These values are stored inside variables named word1 and word2.
- In compute_score(), your program should compute, using the POINTS array, and return the score for the string argument. Characters that are not letters should be given zero points, and uppercase and lowercase letters should be given the same point values.
    - For example, ! is worth 0 points while A and a are both worth 1 point.
    - Though Scrabble rules normally require that a word be in the dictionary, no need to check for that in this problem!
- In main(), your program should print, depending on the players' scores, Player 1 wins!, Player 2 wins!, or Tie!.

Walkthrough

Hints

- You may find the functions `isupper()` and `islower()` to be helpful to you. These functions take in a character as the argument and return a nonzero value if the character is uppercase (for `isupper`) or lowercase (for `islower`).
- To find the value at the `n`th index of an array called `arr`, we can write `arr[n]`. We can apply this to strings as well, as strings are arrays of characters.
- Recall that computers represent characters using ASCII, a standard that represents each character as a number.

How to Test Your Code

Your program should behave per the examples below.

```
$ ./scrabble
Player 1: Question?
Player 2: Question!
Tie!
$ ./scrabble
Player 1: Oh,
Player 2: hai!
Player 2 wins!
$ ./scrabble
Player 1: COMPUTER
Player 2: science
Player 1 wins!
$ ./scrabble
Player 1: Scrabble
Player 2: wiNNeR
Player 1 wins!
```

Not sure how to solve?

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/labs/2021/x/scrabble
```

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

```
style50 scrabble.c
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/scrabble
```

Problem Set 2

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 2** before beginning this problem set.

1. Go to ide.cs50.io and click "Log in" 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 Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Try out any of David's programs from class via Week 2's examples.
- 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!

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, 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 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
```

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 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.
Grade 11
```

Try It

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 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 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?"
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/ 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 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.
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 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.
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

- Recall that `math.h` declares a function called `round` that might be useful here.
- Recall that, when dividing values of type `int` in C, the result will also be an `int`, with any remainder (i.e., digits after the decimal point) discarded. Put another way, the result will be "truncated." You might want to cast your one or more values to `float` before performing division when calculating `L` and `S`!

Walkthrough

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/2021/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/2021/x/readability
```

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
```

Try It

To try out the staff’s implementation of this problem, execute

```
./caesar key
```

substituting a valid integer in place of key, within this sandbox.

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

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. Odds are your pseudocode will use (or imply using!) one or more functions, conditions, Boolean expressions, loops, and/or variables.

Spoiler

There's more than one way to do this, so here's just one!

1. Check that program was run with one command-line argument
2. Iterate over the provided argument to make sure all characters are digits
3. Convert that command-line argument from a `string` to an `int`
4. Prompt user for plaintext
5. Iterate over each character of the plaintext:
    1. If it is an uppercase letter, rotate it, preserving case, then print out the rotated character
    2. If it is a lowercase letter, rotate it, preserving case, then print out the rotated character
    3. If it is neither, print out the character as is
6. Print a newline

It's okay to edit your own after seeing this pseudocode here, but don't simply copy/paste ours into your own!

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

- Recall that you can compile your program with `make`.
- Recall that you can print with `printf`.
- Recall that `argc` and `argv` give you information about what was provided at the command line.
- Recall that the name of the program itself (here, `./caesar`) is in `argv[0]`.

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

- Recall that argc and argv give you information about what was provided at the command line.
- Recall that argv is an array of strings.
- Recall that with printf we can print a string using %s as the placeholder.
- Recall that computer scientists like counting starting from 0.
- Recall that we can access individual elements of an array, such as argv using square brackets, for example: argv[0].

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

- Recall that argv is an array of strings.
- Recall that a string, meanwhile, is just an array of chars.
- Recall that the string.h header file contains a number of useful functions that work with strings.
- Recall that we can use a loop to iterate over each character of a string if we know its length.
- Recall that the ctype.h header file contains a number of useful functions that tell us things about characters.
- Recall that we can return nonzero values from main to indicate that our program did not finish successfully.
- Recall that with printf we can print an integer using %i as the placeholder.
- Recall that the atoi function converts a string that looks like a number into that number.

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

- Try to iterate over every character in the plaintext and literally add 1 to it, then print it.
- If `c` is a variable of type `char` in C, what happens when you call `printf("%c", c + 1)`?

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

- Best to use the modulo (i.e., remainder) operator, `%`, to handle wraparound from `Z` to `A`! But how?
- Things get weird if we try to wrap `Z` or `z` by 1 using the technique in the previous section.
- Things get weird also if we try to wrap punctuation marks using that technique.
- Recall that ASCII maps all printable characters to numbers.
- Recall that the ASCII value of `A` is 65. The ASCII value of `a`, meanwhile, is 97.
- If you're not seeing any output at all when you call `printf`, odds are it's because you're printing characters outside of the valid ASCII range from 0 to 127. Try printing characters as numbers (using `%i` instead of `%c`) at first to see what values you're printing, and make sure you're only ever trying to print valid characters!

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/2021/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/2021/x/caesar
```

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
```

Try It

To try out the staff's implementation of this problem, execute

```
./substitution key
```

substituting a valid key in place of `key`, within this sandbox.

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/2021/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/2021/x/substitution
```

Week 3 Algorithms

Lecture 3

Last week

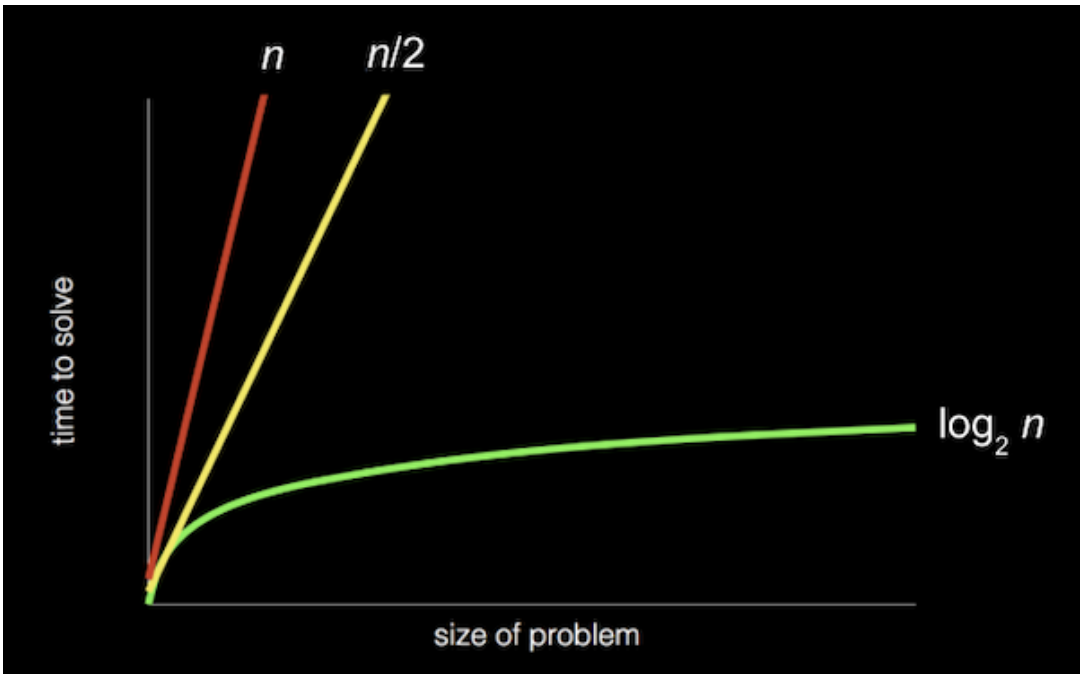
- We learned about tools to solve problems, or bugs, in our code. In particular, we discovered how to use a debugger, a tool that allows us to step slowly through our code and look at values in memory while our program is running.
- Another powerful, if less technical, tool is rubber duck debugging, where we try to explain what we're trying to do to a rubber duck (or some other object), and in the process realize the problem (and hopefully solution!) on our own.
- We looked at memory, visualizing bytes in a grid and storing values in each box, or byte, with variables and arrays.

Searching

- It turns out that, with arrays, a computer can't look at all of the elements at once. Instead, a computer can only to look at them one at a time, though the order can be arbitrary. (Recall in Week 0, David could only look at one page at a time in a phone book, whether he flipped through in order or in a more sophisticated way.)
- **Searching** is how we solve the problem of finding a particular value. A simple case might have an input of some array of values, and the output might simply be a bool, whether or not a particular value is in the array.
- Today we'll look at algorithms for searching. To discuss them, we'll consider **running time**, or how long an algorithm takes to run given some size of input.

Big O

- In week 0, we saw different types of algorithms and their running times:



    - Recall that the red line is searching linearly, one page at a time; the yellow line is searching two pages at a time; and the green line is searching logarithmically, dividing the problem in half each time.
    - And these running times are for the worst case, or the case where the value takes the longest to find (on the last page, as opposed to the first page).
- The more formal way to describe each of these running times is with **big O notation**, which we can think of as "on the order of". For example, if our algorithm is linear search, it will take approximately O(n) steps, read as "big O of n" or "on the order of n". In fact, even an algorithm that looks at two items at a time and takes n/2 steps has O(n). This is because, as n gets bigger and bigger, only the dominant factor, or largest term, n, matters. In the chart above, if we zoomed out and changed the units on our axes, we would see the red and yellow lines end up very close together.
- A logarithmic running time is O(log$_{f_0}$n), no matter what the base is, since this is just an approximation of what fundamentally happens to running time if n is very large.
- There are some common running times:
    - O(n2)
    - O(nlog$_{f_0}$n)
    - O(n)
        - (searching one page at a time, in order)
    - O(log$_{f_0}$n)
        - (dividing the phone book in half each time)
    - O(1)
        - An algorithm that takes a **constant** number of steps, regardless of how big the problem is.
- Computer scientists might also use big Ω, big Omega notation, which is the lower bound of number of steps for our algorithm. Big O is the upper bound of number of steps, or the worst case.
- And we have a similar set of the most common big Ω running times:
    - Ω(n2)
    - Ω(nlog$_{f_0}$n)
    - Ω(n)
    - Ω(log$_{f_0}$n)
    - Ω(1)
        - (searching in a phone book, since we might find our name on the first page we check)

Linear search, binary search

- On stage, we have a few prop doors, with numbers hidden behind them. Since a computer can only look at one element in an array at a time, we can only open one door at a time as well.
- If we want to look for the number zero, for example, we would have to open one door at a time, and if we didn't know anything about the numbers behind the doors, the simplest algorithm would be going from left to right.
- So, we might write pseudocode for **linear search** with:

```
For i from 0 to n–1
    If number behind i'th door
        Return true
Return false
```

    - We label each of n doors from 0 to n–1, and check each of them in order.
    - "Return false" is *outside* the for loop, since we only want to do that after we've looked behind *all* the doors.
    - The big O running time for this algorithm would be O(n), and the lower bound, big Omega, would be Ω(1).
- If we know that the numbers behind the doors are sorted, then we can start in the middle, and find our value more efficiently.
- For binary search, our algorithm might look like:

```
If no doors
    Return false
If number behind middle door
    Return true
Else if number < middle door
    Search left half
Else if number > middle door
    Search right half
```

    - The upper bound for binary search is O(log$_{f_0}$n), and the lower bound also Ω(1), if the number we're looking for is in the middle, where we happen to start.
- With 64 light bulbs, we notice that linear search takes much longer than binary search, which only takes a few steps.

- We turned off the light bulbs at a frequency of one **hertz**, or cycle per second, and a processor's speed might be measured in gigahertz, or billions of operations per second.

Searching with code

- Let's take a look at `numbers.c`:

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

int main(void)
{
    int numbers[] = {4, 6, 8, 2, 7, 5, 0};

    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == 0)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - Here we initialize an array with some values in curly braces, and we check the items in the array one at a time, in order, to see if they're equal to zero (what we were originally looking for behind the doors on stage).
  - If we find the value of zero, we return an exit code of 0 (to indicate success). Otherwise, *after* our for loop, we return 1 (to indicate failure).
- We can do the same for names:

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

int main(void)
{
    string names[] = {"Bill", "Charlie", "Fred", "George", "Ginny", "Percy", "Ron"};

    for (int i = 0; i < 7; i++)
    {
        if (strcmp(names[i], "Ron") == 0)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - Note that `names` is a sorted array of strings.
  - We can't compare strings directly in C, since they're not a simple data type but rather an array of many characters. Luckily, the `string` library has a `strcmp` ("string compare") function which compares strings for us, one character at a time, and returns 0 if they're the same.
  - If we only check for `strcmp(names[i], "Ron")` and not `strcmp(names[i], "Ron") == 0`, then we'll print `Found` even if the name isn't found. This is because `strcmp` returns a value that isn't 0 if two strings *don't* match, and any nonzero value is equivalent to true in a condition.

Structs

- If we wanted to implement a program that searches a phone book, we might want a data type for a "person", with their name and phone number.
- It turns out in C that we can define our own data type, or data *structure*, with a **struct** in the following syntax:

```
typedef struct
{
    string name;
    string number;
}
person;
```

    - We use `string` for the `number`, since we want to include symbols and formatting, like plus signs or hyphens.
    - Our struct contains other data types inside it.
- Let's try to implement our phone book without structs first:

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

int main(void)
{
    string names[] = {"Brian", "David"};
    string numbers[] = {"+1-617-495-1000", "+1-949-468-2750"};

    for (int i = 0; i < 2; i++)
    {
        if (strcmp(names[i], "David") == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

    - We'll need to be careful to make sure that the firstname in `names` matches the first number in `numbers`, and so on.
    - If the name at a certain index `i` in the `names` array matches who we're looking for, we can return the phone number in the `numbers` array at the same index.
- With structs, we can be a little more confident that we won't have human errors in our program:

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

typedef struct
{
    string name;
    string number;
}
person;

int main(void)
{
    person people[2];

    people[0].name = "Brian";
    people[0].number = "+1-617-495-1000";

    people[1].name = "David";
    people[1].number = "+1-949-468-2750";
```

```
    for (int i = 0; i < 2; i++)
    {
        if (strcmp(people[i].name, "David") == 0)
        {
            printf("Found %s\n", people[i].number);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

  - We create an array of the `person` struct type, and name it `people` (as in `int numbers[]`, though we could name it arbitrarily, like any other variable). We set the values for each field, or variable, inside each `person` struct, using the dot operator, `.`.
  - In our loop, we can now be more certain that the `number` corresponds to the `name` since they are from the same `person` struct.
  - We can also improve the design of our program with a constant, like `const int NUMBER = 10;`, and store our values not in our code but in a separate file or even a database, which we'll soon see.
- Soon too, we'll write our own header files with definitions for structs, so they can be shared across different files for our program.

Sorting

- If our input is an unsorted list of numbers, there are many algorithms we could use to produce an output of a sorted list, where all the elements are in order.
- With a sorted list, we can use binary search for efficiency, but it might take more time to write a sorting algorithm for that efficiency, so sometimes we'll encounter the tradeoff of time it takes a human to write a program compared to the time it takes a computer to run some algorithm. Other tradeoffs we'll see might be time and complexity, or time and memory usage.

Selection sort

- Brian is backstage with a set of numbers on a shelf, in unsorted order:

```
6 3 8 5 2 7 4 1
```

- Taking some numbers and moving them to their right place, Brian sorts the numbers pretty quickly.
- Going step-by-step, Brian looks at each number in the list, remembering the smallest one we've seen so far. He gets to the end, and sees that 1 is the smallest, and he knows that must go at the beginning, so he'll just swap it with the number at the beginning, 6:

```
6 3 8 5 2 7 4 1
–             –
1 3 8 5 2 7 4 6
```

- Now Brian knows at least the first number is in the right place, so he can look for the smallest number among the rest, and swap it with the next unsorted number (now the second number):

```
1 3 8 5 2 7 4 6
  –   –
1 2 8 5 3 7 4 6
```

- And he repeats this again, swapping the next smallest, 3, with the 8:

```
1 2 8 5 3 7 4 6
    -   -
1 2 3 5 8 7 4 6
```

- After a few more swaps, we end up with a sorted list.
- This algorithm is called **selection sort**, and we can be a bit more specific with some pseudocode:

```
For i from 0 to n–1
    Find smallest item between i'th item and last item
    Swap smallest item with i'th item
```

  - The first step in the loop is to look for the smallest item in the unsorted part of the list, which will be between the i'th item and last item, since we know we've sorted up to the "i-1'th" item.
  - Then, we swap the smallest item with the i'th item, which makes everything up to the item at i sorted.
- We look at a visualization online with animations for how the elements move for selection sort.

- For this algorithm, we were looking at roughly all n elements to find the smallest, and making n passes to sort all the elements.
- More formally, we can use some math formulas to show that the biggest factor is indeed n2. We started with having to look at all n elements, then only n−1, then n−2:
  n+(n–1)+(n–2)+...+1
  n(n+1)/2
  (n2+n)/2
  n2/2+n/2
  O(n2)
    - Since n2 is the biggest, or dominant, factor, we can say that the algorithm has running time of O(n2).

Bubble sort

- We can try a different algorithm, one where we swap pairs of numbers repeatedly, called **bubble sort**.
- Brian will look at the first two numbers, and swap them so they are in order:

```
6 3 8 5 2 7 4 1
– –
3 6 8 5 2 7 4 1
```

- The next pair, 6 and 8, are in order, so we don't need to swap them.
- The next pair, 8 and 5, need to be swapped:

```
3 6 8 5 2 7 4 1
  – –
3 6 5 8 2 7 4 1
```

- Brian continues until he reaches the end of the list:

```
3 6 5 2 8 7 4 1
    – –
3 6 5 2 7 8 4 1
      – –
3 6 5 2 7 4 8 1
        – –
3 6 5 2 7 4 1 8
          -
```

- Our list isn't sorted yet, but we're slightly closer to the solution because the biggest value, 8, has been shifted all the way to the right. And other bigger numbers have also moved to the right, or "bubbled up".
- Brian will make another pass through the list:

```
3 6 5 2 7 4 1 8
– –
3 6 5 2 7 4 1 8
 – –
3 5 6 2 7 4 1 8
  – –
3 5 2 6 7 4 1 8
    – –
3 5 2 6 7 4 1 8
      – –
3 5 2 6 4 7 1 8
        - –
3 5 2 6 4 1 7 8
          - -
```

    - Note that we didn't need to swap the 3 and 6, or the 6 and 7.
    - But now, the next biggest value, 7, moved all the way to the right.
- Brian will repeat this process a few more times, and more and more of the list becomes sorted, until we have a fully sorted list.
- With selection sort, the best case with a sorted list would still take just as many steps as the worst case, since we only check for the smallest number with each pass.
- The pseudocode for bubble sort might look like:

```
Repeat until sorted
  For i from 0 to n–2
    If i'th and i+1'th elements out of order
```

- Swap them
    - Since we are comparing the i'th and i+1'th element, we only need to go up to n–2 for i. Then, we swap the two elements if they're out of order.
    - And we can stop as soon as the list is sorted, since we can just remember whether we made any swaps. If not, the list must be sorted already.
- To determine the running time for bubble sort, we have n–1 comparisons in the loop, and at most n–1 loops, so we get n2–2n+2 steps total. But the largest factor, or dominant term, is again n2 as n gets larger and larger, so we can say that bubble sort has O(n2). So it turns out that fundamentally, selection sort and bubble sort have the same upper bound for running time.
- The lower bound for running time here would be Ω(n), once we look at all the elements once.
- So our upper bounds for running time that we've seen are:
    - O(n2)
        - selection sort, bubble sort
    - O(nlog$_{f0}$n)
    - O(n)
        - linear search
    - O(log$_{f0}$n)
        - binary search
    - O(1)
- And for lower bounds:
    - Ω(n2)
        - selection sort
    - Ω(nlog$_{f0}$n)
    - Ω(n)
        - bubble sort
    - Ω(log$_{f0}$n)
    - Ω(1)
        - linear search, binary search

Recursion

- **Recursion** is the ability for a function to call itself. We haven't seen this in code yet, but we've seen something in pseudocode in week 0 that we might be able to convert:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If Smith is on page
5     Call Mike
6  Else if Smith is earlier in book
7     Open to middle of left half of book
8     Go back to line 3
9  Else if Smith is later in book
10    Open to middle of right half of book
11    Go back to line 3
12 Else
13    Quit
```

    - Here, we're using a loop-like instruction to go back to a particular line.
- We could instead just repeat our entire algorithm on the half of the book we have left:

```
1  Pick up phone book
2  Open to middle of phone book
3  Look at page
4  If Smith is on page
5     Call Mike
6  Else if Smith is earlier in book
7     Search left half of book
8
9  Else if Smith is later in book
10    Search right half of book
11
12 Else
```

- 13 Quit
    - This seems like a cyclical process that will never end, but we're actually changing the input to the function and dividing the problem in half each time, stopping once there's no more book left.
- In week 1, too, we implemented a "pyramid" of blocks in the following shape:
- #
- ##
- ###
- ####
- But notice that a pyramid of height 4 is actually a pyramid of height 3, with an extra row of 4 blocks added on. And a pyramid of height 3 is a pyramid of height 2, with an extra row of 3 blocks. A pyramid of height 2 is a pyramid of height 1, with an extra row of 2 blocks. And finally, a pyramid of height 1 is just a single block.
- With this idea in mind, we can write a recursive function to draw a pyramid, a function that calls itself to draw a smaller pyramid before adding another row.

Merge sort

- We can take the idea of recusion to sorting, with another algorithm called **merge sort**. The pseudocode might look like:
- If only one number
- Return
- Else
- Sort left half of number
- Sort right half of number
- Merge sorted halves
- We'll best see this in practice with two sorted lists:
- 3 5 6 8 | 1 2 4 7
- We'll *merge* the two lists for a final sorted list by taking the smallest element at the front of each list, one at a time:
- 3 5 6 8 | _ 2 4 7
-
- 1
- The 1 on the right side is the smallest between 1 and 3, so we can start our sorted list with it.
- 3 5 6 8 | _ _ 4 7
-
- 1 2
- The next smallest number, between 2 and 3, is 2, so we use the 2.
- _ 5 6 8 | _ _ 4 7
-
- 1 2 3
- _ 5 6 8 | _ _ _ 7
-
- 1 2 3 4
- _ _ 6 8 | _ _ _ 7
-
- 1 2 3 4 5
- _ _ _ 8 | _ _ _ 7
-
- 1 2 3 4 5 6
- _ _ _ 8 | _ _ _ _
-
- 1 2 3 4 5 6 7
- _ _ _ _ | _ _ _ _
-
- 1 2 3 4 5 6 7 8
    - Now we have a completely sorted list.
- We've seen how the final line in our pseudocode can be implemented, and now we'll see how the entire algorithm works:
- If only one number
- Return
- Else

- Sort left half of number
- Sort right half of number
- Merge sorted halves
- We start with another unsorted list:
- 6 3 8 5 2 7 4 1
- To start, we need to sort the left half first:
- 6 3 8 5
- Well, to sort that, we need to sort the left half of the left half first:
- 6 3
- Now both of these halves just have one item each, so they're both sorted. We merge these two lists together, for a sorted list:
- _ _ 8 5 2 7 4 1
- 3 6
- We're back to sorting the right half of the left half, merging them together:
- _ _ _ _ 2 7 4 1
- 3 6 5 8
- Both halves of the *left half* have been sorted individually, so now we need to merge them together:
- _ _ _ _ 2 7 4 1
- _ _ _ _
- 3 5 6 8
- We'll do what we just did, with the right half:
- _ _ _ _ _ _ _ _
- _ _ _ _ 2 7 1 4
- 3 5 6 8
    - First, we sort both halves of the right half.
    - _ _ _ _ _ _ _ _
    - _ _ _ _ _ _ _ _
    - 3 5 6 8 1 2 4 7
    - Then, we merge them together for a sorted right half.
- Finally, we have two sorted halves again, and we can merge them for a fully sorted list:
- _ _ _ _ _ _ _ _
- _ _ _ _ _ _ _ _
- _ _ _ _ _ _ _ _
- 1 2 3 4 5 6 7 8
- Each number was moved from one shelf to another three times (since the list was divided from 8, to 4, to 2, and to 1 before merged back together into sorted lists of 2, 4, and finally 8 again). And each shelf required all 8 numbers to be merged together, one at a time.
- Each shelf required n steps, and there were only $\log_2 n$ shelves needed, so we multiply those factors together. Our total running time for binary search is $O(\log_2 n)$:
    - O(n2)
        - selection sort, bubble sort
    - $O(n \log_2 n)$
        - merge sort
    - O(n)
        - linear search
    - $O(\log_2 n)$
        - binary search
    - O(1)
- (Since $\log_2 n$ is greater than 1 but less than n, $n \log_2 n$ is in between n (times 1) and n2.)
- The best case, Ω, is still $n \log_2 n$, since we still have to sort each half first and then merge them together:
    - Ω(n2)
        - selection sort
    - $\Omega(n \log_2 n)$
        - merge sort
    - Ω(n)
        - bubble sort
    - $\Omega(\log_2 n)$
    - Ω(1)
        - linear search, binary search

- Even though merge sort is likely to be faster than selection sort or bubble sort, we did need another shelf, or more memory, to temporarily store our merged lists at each stage. We face the tradeoff of incurring a higher cost, another array in memory, for the benefit of faster sorting.
- Finally, there is another notation, Θ, Theta, which we use to describe running times of algorithms if the upper bound and lower bound is the same. For example, merge sort has Θ(nlogf_0n) since the best and worst case both require the same number of steps. And selection sort has Θ(n2):
    - Θ(n2)
        - selection sort
    - Θ(nlogf_0n)
        - merge sort
    - Θ(n)
    - Θ(logf_0n)
    - Θ(1)
- We look at a final visualization of sorting algorithms with a larger number of inputs, running at the same time.

Lab 3: Sort

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Analyze three sorting programs to determine which algorithms they use.

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Background

Recall from lecture that we saw a few algorithms for sorting a sequence of numbers: selection sort, bubble sort, and merge sort.

- Selection sort iterates through the unsorted portions of a list, selecting the smallest element each time and moving it to its correct location.
- Bubble sort compares pairs of adjacent values one at a time and swaps them if they are in the incorrect order. This continues until the list is sorted.
- Merge sort recursively divides the list into two repeatedly and then merges the smaller lists back into a larger one in the correct order.

Getting Started

1. Log into ide.cs50.io using your GitHub account.
2. In your terminal window, run `wget https://cdn.cs50.net/2020/fall/labs/3/lab3.zip` to download a Zip file of the lab distribution code.
3. In your terminal window, run `unzip lab3.zip` to unzip (i.e., decompress) that Zip file.
4. In your terminal window, run `cd lab3` to change directories into your `lab3` directory.

Instructions

Provided to you are three already-compiled C programs, `sort1`, `sort2`, and `sort3`. Each of these programs implements a different sorting algorithm: selection sort, bubble sort, or merge sort (though not necessarily in that order!). Your task is to determine which sorting algorithm is used by each file.

- `sort1`, `sort2`, and `sort3` are binary files, so you won't be able to view the C source code for each. To assess which sort implements which algorithm, run the sorts on different lists of values.
- Multiple `.txt` files are provided to you. These files contain `n` lines of values, either reversed, shuffled, or sorted.
  - For example, `reversed10000.txt` contains 10000 lines of numbers that are reversed from `10000`, while `random100000.txt` contains 100000 lines of numbers that are in random order.
- To run the sorts on the text files, in the terminal, run `./[program_name] [text_file.txt]`.
  - For example, to sort `reversed10000.txt` with `sort1`, run `./sort1 reversed10000.txt`.
- You may find it helpful to time your sorts. To do so, run `time ./[sort_file] [text_file.txt]`.
  - For example, you could run `time ./sort1 reversed10000.txt` to run `sort1` on 10,000 reversed numbers. At the end of your terminal's output, you can look at the `real` time to see how much time actually elapsed while running the program.
- Record your answers in `answers.txt`, along with an explanation for each program, by filling in the blanks marked `TODO`.

Walkthrough

Hints

- The different types of `.txt` files may help you determine which sort is which. Consider how each algorithm performs with an already sorted list. How about a reversed list? Or shuffled list? It may help to work through a smaller list of each type and walk through each sorting process.

Not sure how to solve?

How to Check Your Answers

Execute the below to evaluate the correctness of your answers using `check50`. But be sure to fill in your explanations as well, which `check50` won't check here!

```
check50 cs50/labs/2021/x/sort
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/sort
```

Problem Set 3

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 3** before beginning this problem set.

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. **But you do not need to submit both. It is not necessary to solve Tideman if you solve Runoff instead.**

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Try out any of David's programs from class via Week 3's examples.
- 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 plurality`

  is yielding errors, try running

  `help50 make plurality`

  instead!

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 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 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 and then, in a terminal window, execute each of the below.

- Execute `cd ~` (or simply `cd` with no arguments) to ensure that you're in your home directory).
- Execute `mkdir pset3` to make (i.e., create) a directory called `pset3`.
- 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/2020/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 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/2021/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/2021/x/plurality
```

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?

Ballot	Ballot	Ballot	Ballot	Ballot
Alice	Alice	Bob	Bob	Charlie

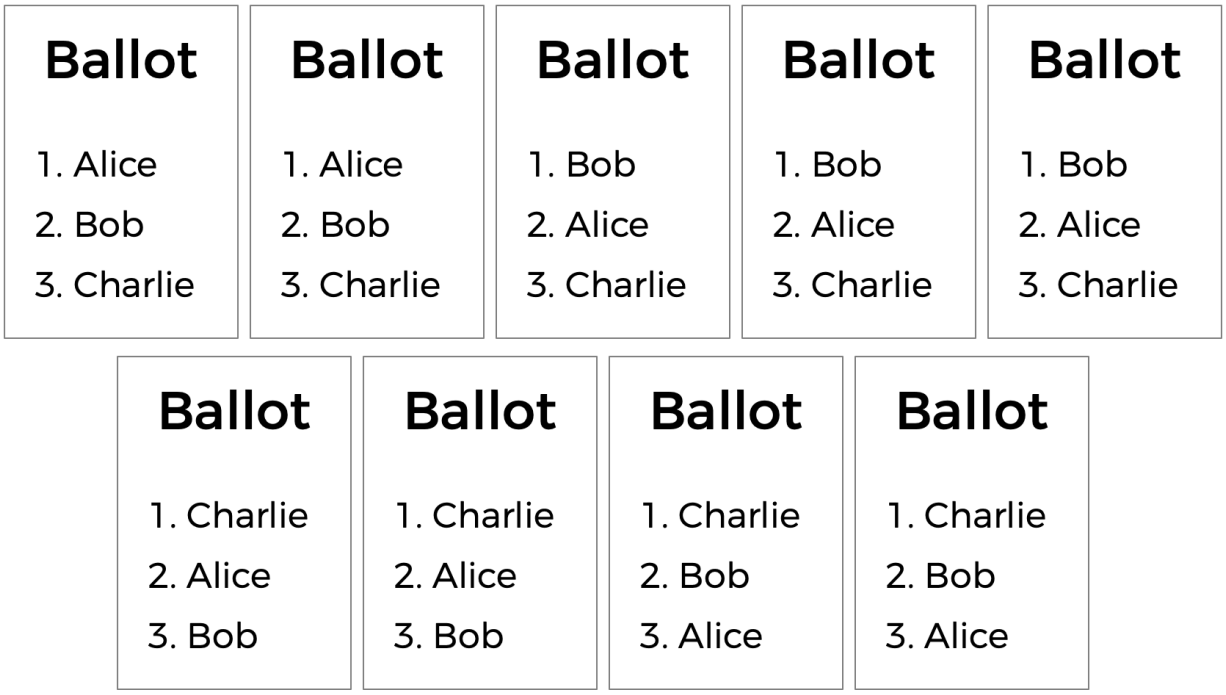
A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

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.

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



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 and then, in a terminal window, execute each of the below.

- Navigate to 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/2020/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.

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 to 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` (and the inclusion of 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

- Recall that `candidate_count` stores the number of candidates in the election.
- Recall that you can use `strcmp` to compare two strings.
- Recall that `preferences[i][j]` stores the index of the candidate who is the `j`th ranked preference for the `i`th voter.

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

- Recall that `voter_count` stores the number of voters in the election.
- Recall that for a voter `i`, their top choice candidate is represented by `preferences[i][0]`, their second choice candidate by `preferences[i][1]`, etc.
- Recall that the `candidate` struct has a field called `eliminated`, which will be `true` if the candidate has been eliminated from the election.
- Recall that the `candidate` struct has a field called `votes`, which you'll likely want to update for each voter's preferred candidate.

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

- Recall that `voter_count` stores the number of voters in the election. Given that, how would you express the number of votes needed to win the election?

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

- You'll likely want to loop through the candidates to find the one who is both still in the election and has the fewest number of votes. What information should you keep track of as you loop through the candidates?

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

- Recall that a tie happens if every candidate still in the election has the same number of votes. Note, too, that the `is_tie` function takes an argument `min`, which is the smallest number of votes any candidate currently has. How might you use that information to determine if the election is a tie (or, conversely, not a tie)?

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 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
- 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/2021/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/2021/x/runoff
```

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

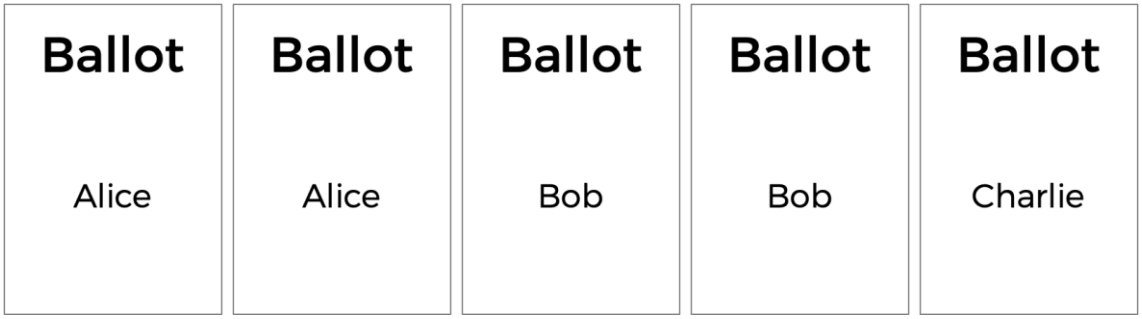
Rank 1: Charlie
Rank 2: Alice
Rank 3: Bob

Charlie
```

Background

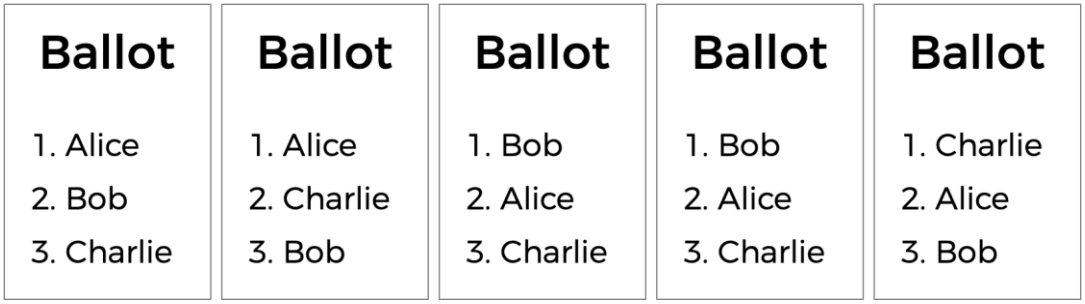
You already know about plurality elections, which follow a very simple algorithm for determining the winner of an election: every voter gets one vote, and the candidate with the most votes wins.

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



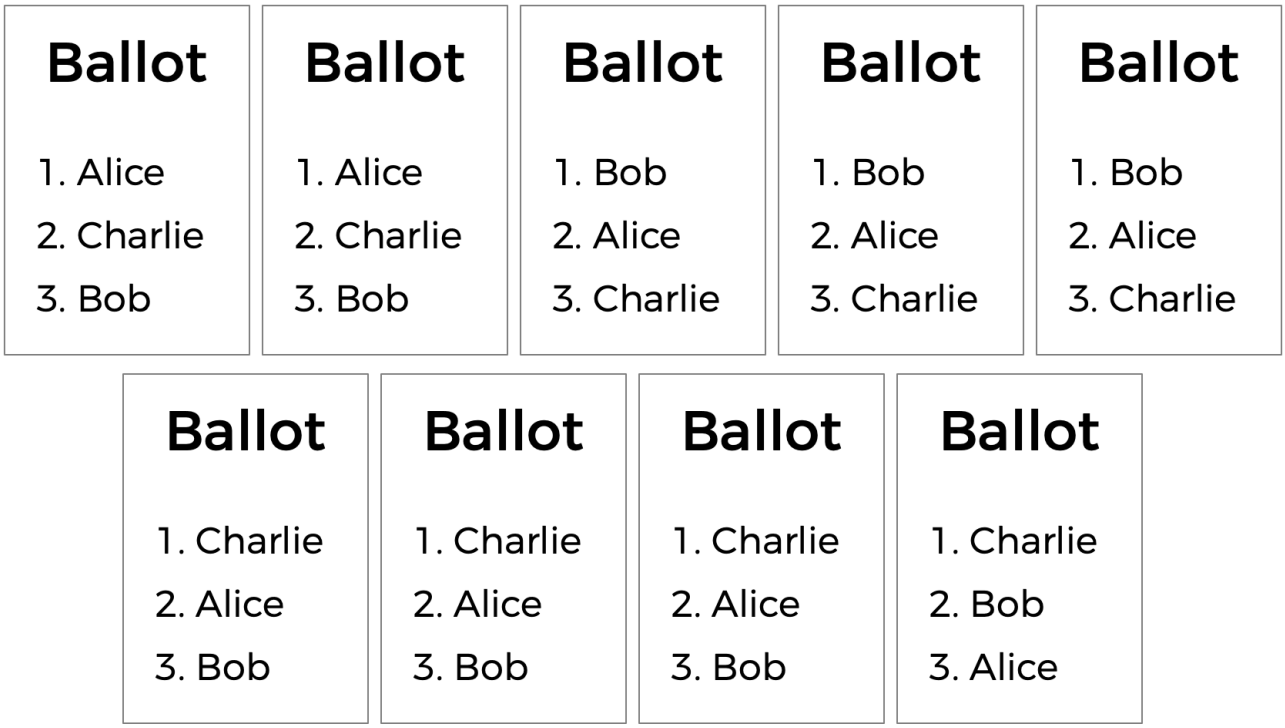
A plurality vote would here declare a tie between Alice and Bob, since each has two votes. But is that the right outcome?

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.



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.

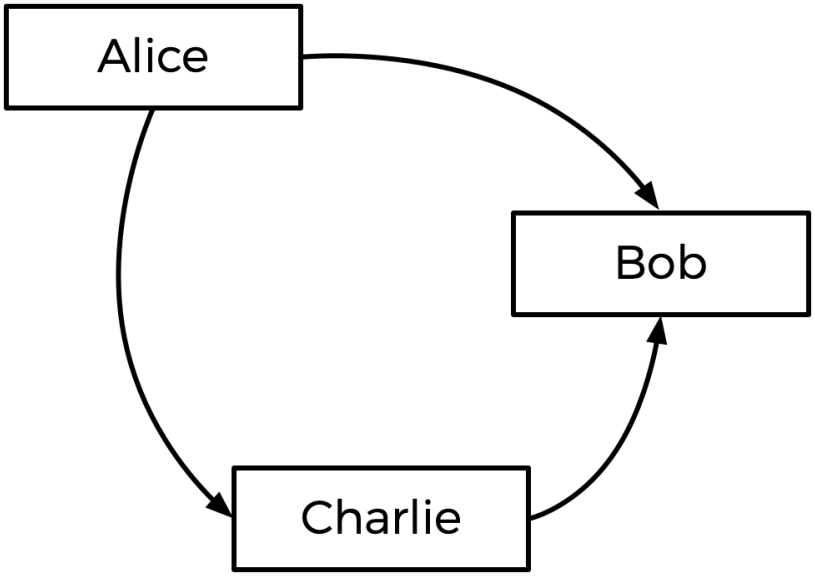


Who should win this election? In a plurality vote where each voter chooses their first preference only, Charlie wins this election with four votes compared to only three for Bob and two for Alice. (Note that, if you're familiar with the instant runoff voting system, Charlie wins here under that system 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.

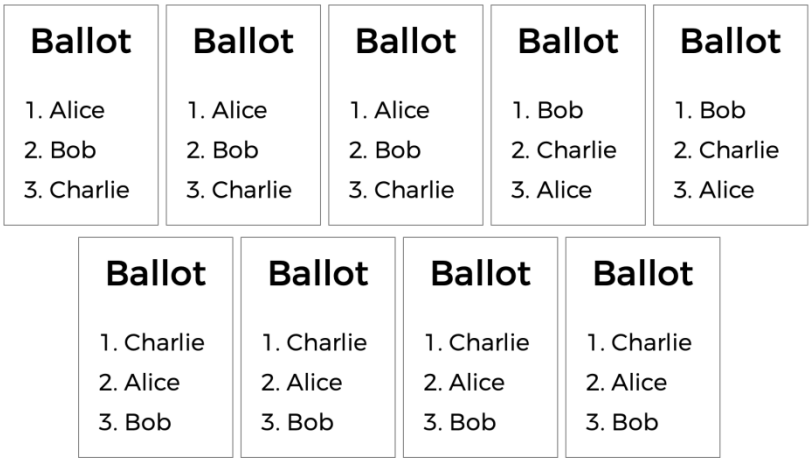
Generally speaking, the Tideman method works by constructing a "graph" of candidates, where an arrow (i.e. edge) from candidate A to candidate B indicates that candidate A wins against candidate B in a head-to-head matchup. The graph for the above election, then, would look like the below.

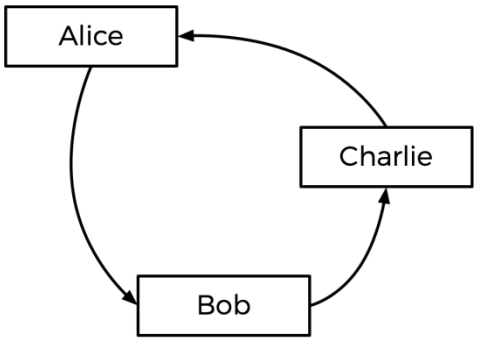


The arrow from Alice to Bob means that more voters prefer Alice to Bob (5 prefer Alice, 4 prefer Bob). Likewise, the other arrows mean that more voters prefer Alice to Charlie, and more voters prefer Charlie to Bob.

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.





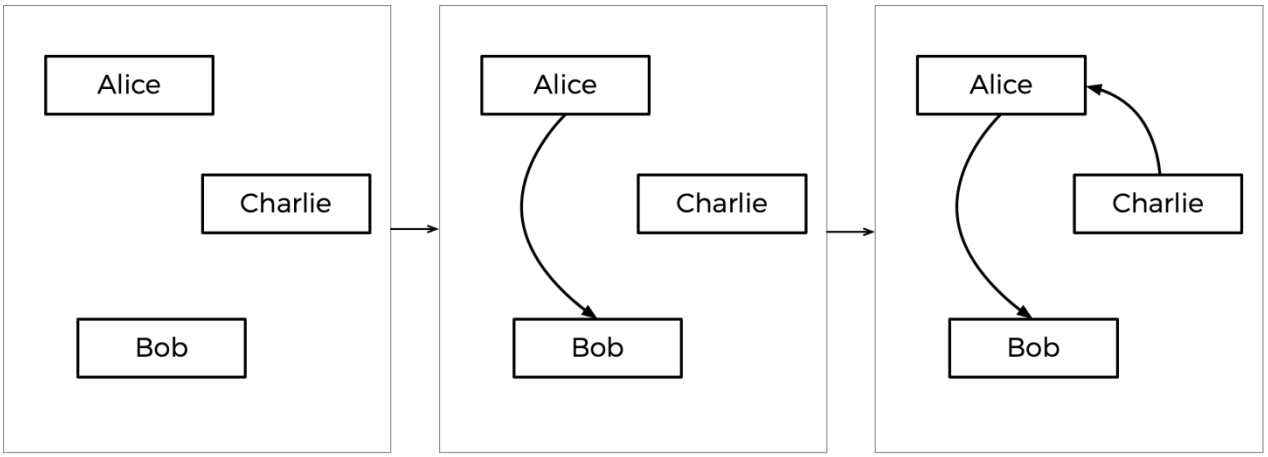
Between Alice and Bob, Alice is preferred over Bob by a 7-2 margin. Between Bob and Charlie, Bob is preferred over Charlie by a 5-4 margin. But between Charlie and Alice, Charlie is preferred over Alice by a 6-3 margin. If we draw out the graph, there is no source! We have a cycle of candidates, where 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 and then, in a terminal window, execute each of the below.

- Navigate to 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/2020/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 strings 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 ith 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 ith 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 ith 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…

- 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/2021/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/2021/x/tideman
```

Week 4 Memory

Lecture 4

Hexadecimal

- In week 2, we talked about memory and how each byte has an address, or identifier, so we can refer to where our data are actually stored.
- It turns out that, by convention, the addresses for memory use the counting system **hexadecimal**, or base-16, where there are 16 digits: 0-9, and A-F as equivalents to 10-15.
- Let's consider a two-digit hexadecimal number:

```
16^1 16^0
  0   A
```

  - Here, the A in the ones place (since 16^0 = 1) has a decimal value of 10. We can keep counting until `0F`, which is equivalent to 15 in decimal.

- After `0F`, we need to carry the one, as we would go from 09 to 10 in decimal:

```
16^1 16^0
  1   0
```

  - Here, the `1` has a value of 16^1 * 1 = 16, so `10` in hexadecimal is 16 in decimal.

- With two digits, we can have a maximum value of `FF`, or 16^1 * 15 + 16^0 * 15 = 240 + 15 = 255, which is the same maximum value with 8 bits of binary. So two digits in hexadecimal can conveniently represent the value of a byte in binary. (Each digit in hexadecimal, with 16 values, maps to four bits in binary.)
- In writing, we indicate a value is in hexadecimal by prefixing it with `0x`, as in `0x10`, where the value is equal to 16 in decimal, as opposed to 10.
- The RGB color system conventionally uses hexadecimal to describe the amount of each color. For example, `000000` in hexadecimal represents 0 for each of red, green, and blue, for a combined color of black. And `FF0000` would be 255, or the highest possible, amount of red. `FFFFFF` would indicate the highest value of each color, combining to be the brightest white. With different values for each color, we can represent millions of different colors.
- For our computer's memory, too, we'll use hexadecimal for each address or location.

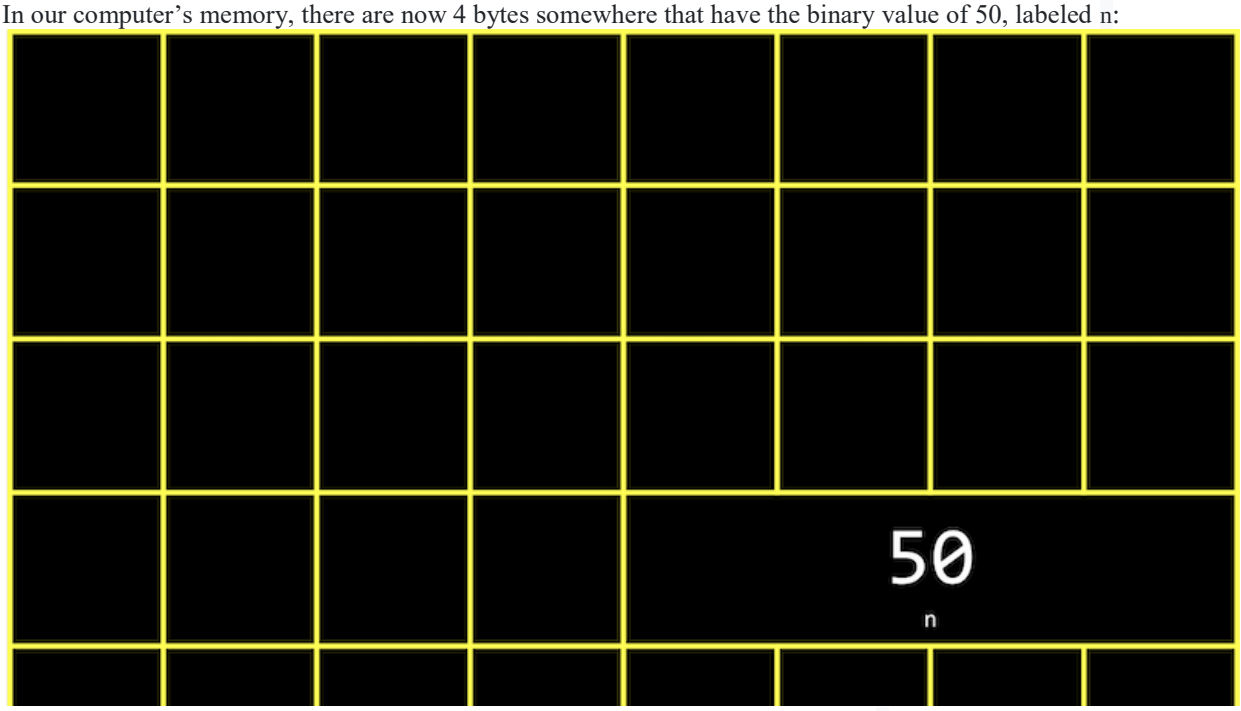
Addresses

- We might create a value `n`, and print it out:

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%i\n", n);
}
```

- In our computer's memory, there are now 4 bytes somewhere that have the binary value of 50, labeled `n`:



- It turns out that, with the billions of bytes in memory, those bytes for the variable `n` starts at some location, which might look something like `0x12345678`.
- In C, we can actually see the address with the `&` operator, which means "get the address of this variable":

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%p\n", &n);
}
```

  - `%p` is the format code for an address.
  - In the CS50 IDE, we see an address like `0x7ffd80792f7c`. The value of the address in itself is not useful, since it's just some location in memory that the variable is stored in; instead, the important idea is that we can *use* this address later.
- The `*` operator, or the dereference operator, lets us "go to" the location that a pointer is pointing to.
- For example, we can print `*&n`, where we "go to" the address of `n`, and that will print out the value of `n`, `50`, since that's the value at the address of `n`:

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%i\n", *&n);
}
```

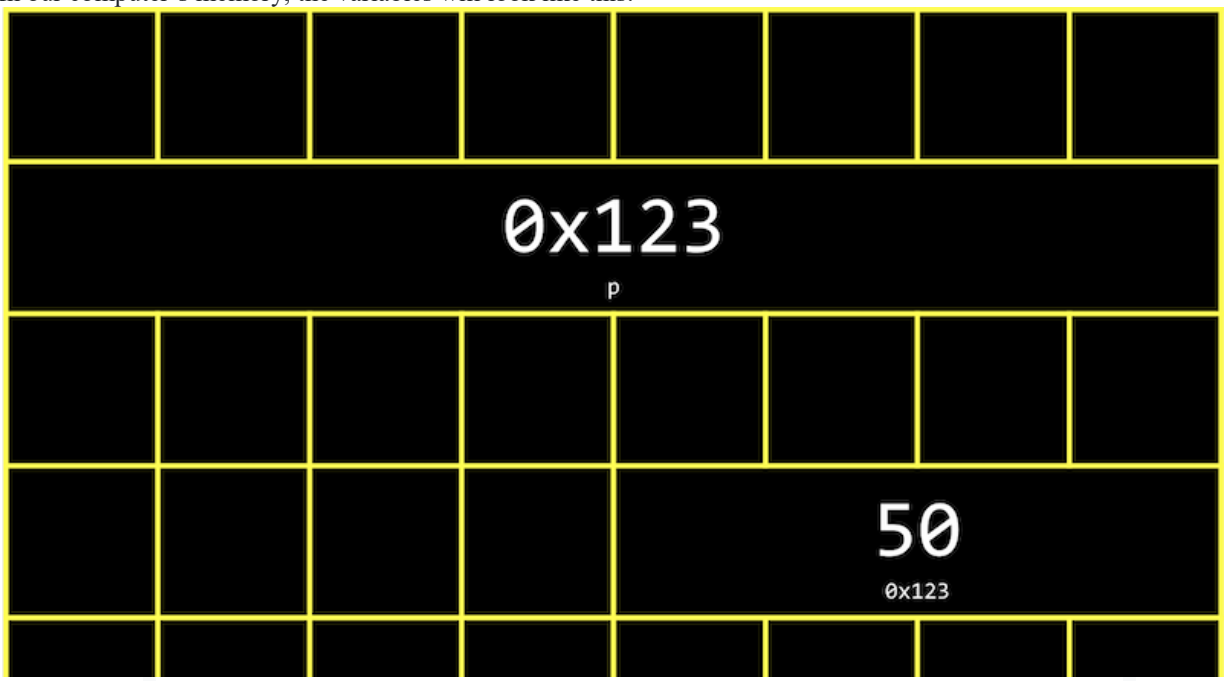
Pointers

- A variable that stores an address is called a **pointer**, which we can think of as a value that "points" to a location in memory. In C, pointers can refer to specific types of values.
- We can use the `*` operator (in an unfortunately confusing way) to declare a variable that we want to be a pointer:

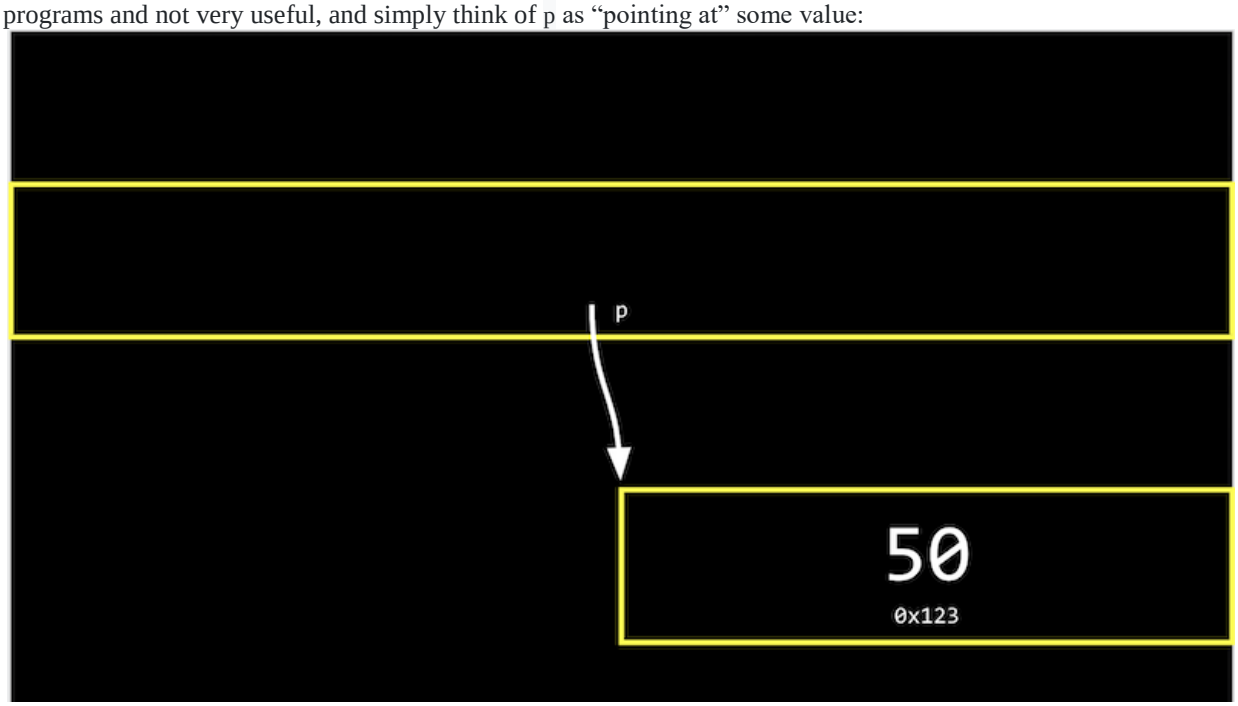
```
#include <stdio.h>

int main(void)
{
   int n = 50;
   int *p = &n;
   printf("%p\n", p);
```

- }
    - Here, we use `int *p` to declare a variable, `p`, that has the type of `*`, a pointer, to a value of type `int`, an integer. Then, we can print its value (an address, something like `0x12345678`), or print the *value at* its location with `printf("%i\n", *p);`.
- In our computer's memory, the variables will look like this:



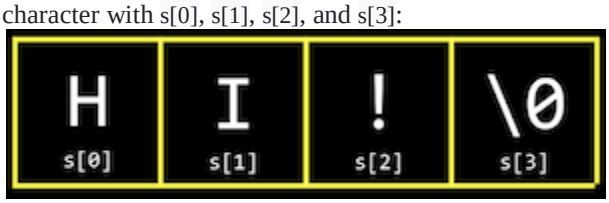
    - Since `p` is a variable itself, it's somewhere in memory, and the value stored there is the address of `n`.
    - Modern computer systems are "64-bit", meaning that they use 64 bits to address memory, so a pointer will in reality be 8 bytes, twice as big as an integer of 4 bytes.
- We can abstract away the actual value of the addresses, since they'll be different as we declare variables in our programs and not very useful, and simply think of `p` as "pointing at" some value:



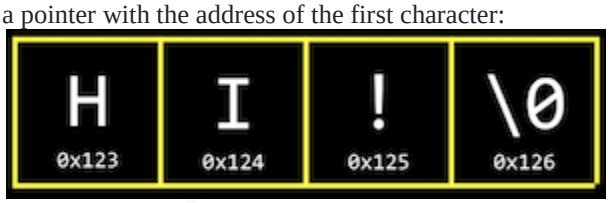
- In the real world, we might have a mailbox labeled "p", among many mailboxes with addresses. Inside our mailbox, we can put a value like `0x123`, which is the address of some other mailbox `n`, with the address `0x123`.

Strings

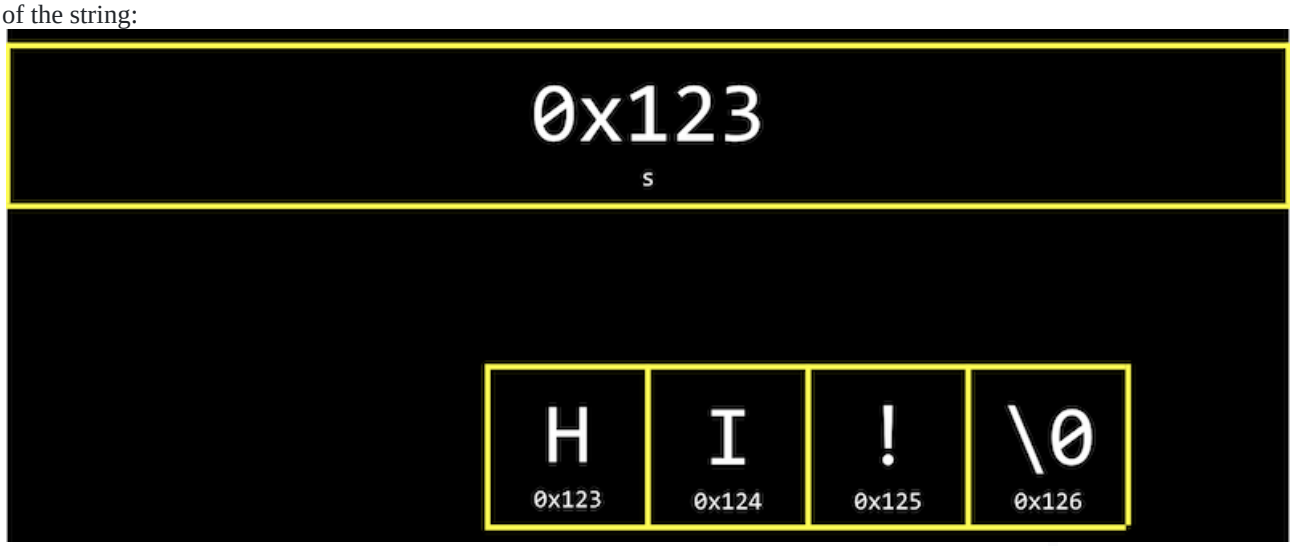
- A variable declared with `string s = "HI!";` will be stored one character at a time in memory. And we can access each character with `s[0]`, `s[1]`, `s[2]`, and `s[3]`:



- But it turns out that each character, since it's stored in memory, *also* has some unique address, and `s` is actually just a pointer with the address of the first character:



- And the variable `s` stores the address of the first character of the string. The value `\0` is the only indicator of the end of the string:



  - Since the rest of the characters are in an array, back-to-back, we can start at the address in `s` and continue reading one character at a time from memory until we reach `\0`.
- Let's print out a string:

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

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

- We can see the value stored in `s` with `printf("%p\n", s);`, and we see something like `0x4006a4` since we're printing the address in memory of the first character of the string.
- If we add another line, `printf("%p\n", &s[1]);`, we indeed see the next address in memory: `0x4006a5`.
- It turns out that `string s` is just a pointer, an address to some character in memory.
- In fact, the CS50 library defines a type that doesn't exist in C, `string`, as `char *`, with `typedef char *string;`. The custom type, `string`, is defined as just a `char *` with `typedef`. So `string s = "HI!"` is the same as `char *s = "HI!";`. And we can use strings in C in the exact same way without the CS50 library, by using `char *`.

Pointer arithmetic

- **Pointer arithmetic** is mathematical operations on addresses with pointers.
- We can print out each character in a string (using `char *` directly):

```
#include <stdio.h>

```

```
int main(void)
{
    char *s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
}
```

- But we can go to addresses directly:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", *s);
    printf("%c\n", *(s+1));
    printf("%c\n", *(s+2));
}
```

  - `*s` goes to the address stored in `s`, and `*(s+1)` goes to the location in memory with an address one byte higher, or the next character. `s[1]` is syntactic sugar for `*(s+1)`, equivalent in function but more human-friendly to read and write.
- We can even try to go to addresses in memory that we shouldn't, like with `*(s+10000)`, and when we run our program, we'll get a **segmentation fault**, or crash as a result of our program touching memory in a segment it shouldn't have.

Compare and copy

- Let's try to compare two integers from the user:

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

int main(void)
{
    int i = get_int("i: ");
    int j = get_int("j: ");

    if (i == j)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

  - We compile and run our program, and it works as we'd expect, with the same values of the two integers giving us "Same" and different values "Different".
- When we try to compare two strings, we see that the same inputs are causing our program to print "Different":

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

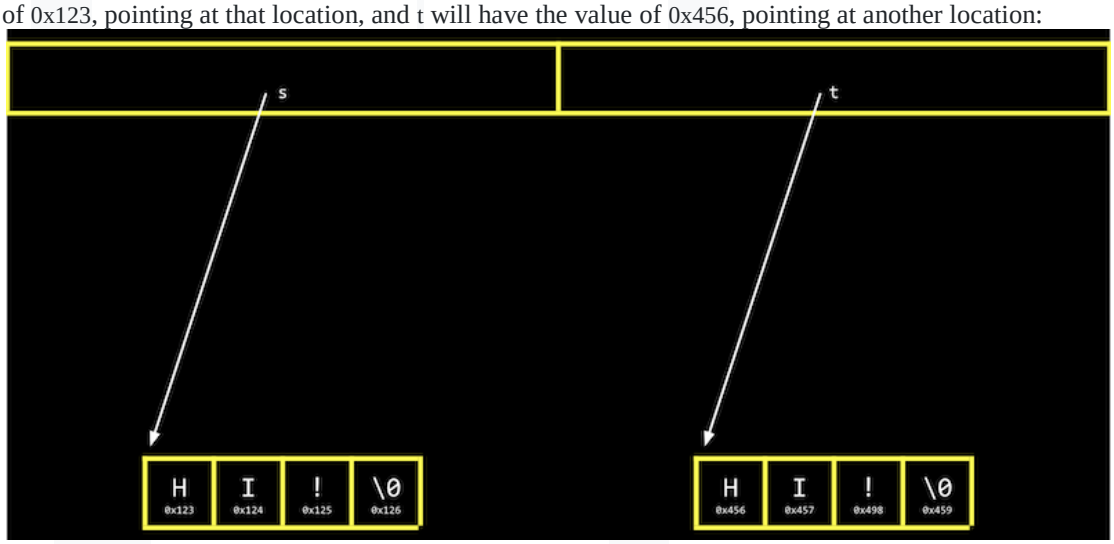
int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    if (s == t)
    {
        printf("Same\n");
    }
```

```
    else
    {
        printf("Different\n");
    }
}
```

- Even when our inputs are the same, we see “Different” printed.
- Each “string” is a pointer, `char *`, to a different location in memory, where the first character of each string is stored. So even if the characters in the string are the same, this will always print “Different”.

- For example, our first string might be at address 0x123, our second might be at 0x456, and `s` will have the value of 0x123, pointing at that location, and `t` will have the value of 0x456, pointing at another location:



- And `get_string`, this whole time, has been returning just a `char *`, or a pointer to the first character of a string from the user. Since we called `get_string` twice, we got two different pointers back.
- Let’s try to copy a string:

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

int main(void)
{
    char *s = get_string("s: ");

    char *t = s;

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

- We get a string `s`, and copy the value of `s` into `t`. Then, we capitalize the first letter in `t`.
- But when we run our program, we see that both `s` and `t` are now capitalized.
- Since we set `s` and `t` to the same value, or the same address, they’re both pointing to the same character, and so we capitalized the same character in memory!

- To actually make a copy of a string, we have to do a little more work, and copy each character in `s` to somewhere else in memory:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

```

```
    char *t = malloc(strlen(s) + 1);

    for (int i = 0, n = strlen(s); i < n + 1; i++)
    {
        t[i] = s[i];
    }

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

- We create a new variable, `t`, of the type `char *`, with `char *t`. Now, we want to point it to a new chunk of memory that's large enough to store the copy of the string. With **malloc**, we *allocate* some number of bytes in memory (that aren't already used to store other values), and we pass in the number of bytes we'd like to mark for use. We already know the length of `s`, and we add 1 to that for the terminating null character. So, our final line of code is `char *t = malloc(strlen(s) + 1);`.
- Then, we copy each character, one at a time, with a `for` loop. We use `i < n + 1`, since we actually want to go *up to* `n`, the length of the string, to ensure we copy the terminating character in the string. In the loop, we set `t[i] = s[i]`, copying the characters. While we could use `*(t+i) = *(s+i)` to the same effect, it's arguably less readable.
- Now, we can capitalize just the first letter of `t`.

- We can add some error-checking to our program:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

    char *t = malloc(strlen(s) + 1);
    if (t == NULL)
    {
        return 1;
    }

    for (int i = 0, n = strlen(s); i < n + 1; i++)
    {
        t[i] = s[i];
    }

    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    printf("s: %s\n", s);
    printf("t: %s\n", t);

    free(t);
}
```

- If our computer is out of memory, `malloc` will return `NULL`, the null pointer, or a special value that indicates there isn't an address to point to. So we should check for that case, and exit if `t` is `NULL`.
- We could also check that `t` has a length, before trying to capitalize the first character.
- Finally, we should **free** the memory we allocated earlier, which marks it as usable again by some other program. We call the `free` function and pass in the pointer `t`, since we're done with that chunk of memory.

(get_string, too, calls malloc to allocate memory for strings, and calls free just before the main function returns.)

- We can actually also use the strcpy function, from the C's string library, with strcpy(t, s); instead of our loop, to copy the string s into t.

valgrind

- valgrind is a command-line tool that we can use to run our program and see if it has any **memory leaks**, or memory we've allocated without freeing, which might eventually cause out computer to run out of memory.
- Let's build a string but allocate less than what we need in memory.c:

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

int main(void)
{
    char *s = malloc(3);
    s[0] = 'H';
    s[1] = 'I';
    s[2] = '!';
    s[3] = '\0';
    printf("%s\n", s);
}
```

  - We also don't free the memory we've allocated.
  - We'll run valgrind ./memory after compiling, and we'll see a lot of output, but we can run help50 valgrind ./memory to help explain some of those messages. For this program, we see snippets like "Invalid write of size 1", "Invalid read of size 1", and finally "3 bytes in 1 blocks are definitely lost", with line numbers nearby. Indeed, we're writing to memory, s[3], which is not part of what we originally allocated for s. And when we print out s, we're reading all the way to s[3] as well. And finally, s isn't freed at the end of our program.

- We can make sure to allocate the right number of bytes, and free memory at the end:

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

int main(void)
{
    char *s = malloc(4);
    s[0] = 'H';
    s[1] = 'I';
    s[2] = '!';
    s[3] = '\0';
    printf("%s\n", s);
    free(s);
}
```

  - Now, valgrind doesn't show any warning messages.

Garbage values

- Let's take a look at the following:

```
int main(void)
{
    int *x;
    int *y;

    x = malloc(sizeof(int));

    *x = 42;
    *y = 13;

    y = x;
```

```
    *y = 13;
}
```

- We declare two pointers to integers, `x` and `y`, but don't assign them values. We use `malloc` to allocate enough memory for an integer with `sizeof(int)`, and store it in `x`. `*x = 42` goes to the address `x` points to, and sets that location in memory to the value 42.
- With `*y = 13`, we're trying to put the value 13 at the address `y` points to. But since we never assigned `y` a value, it has a **garbage value**, or whatever unknown value that was in memory, from whatever program was running in our computer before. So when we try to go to the garbage value in `y` as an address, we're going to some unknown address, which is likely to cause a segmentation fault, or segfault.

- We watch Pointer Fun with Binky, an animated video demonstrating the concepts in the code above.
- We can print out garbage values, by declaring an array but not setting any of its values:

```
#include <stdio.h>

int main(void)
{
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", scores[i]);
    }
}
```

- When we compile and run this program, we see various values printed.

Swap

- Let's try to swap the values of two integers.

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

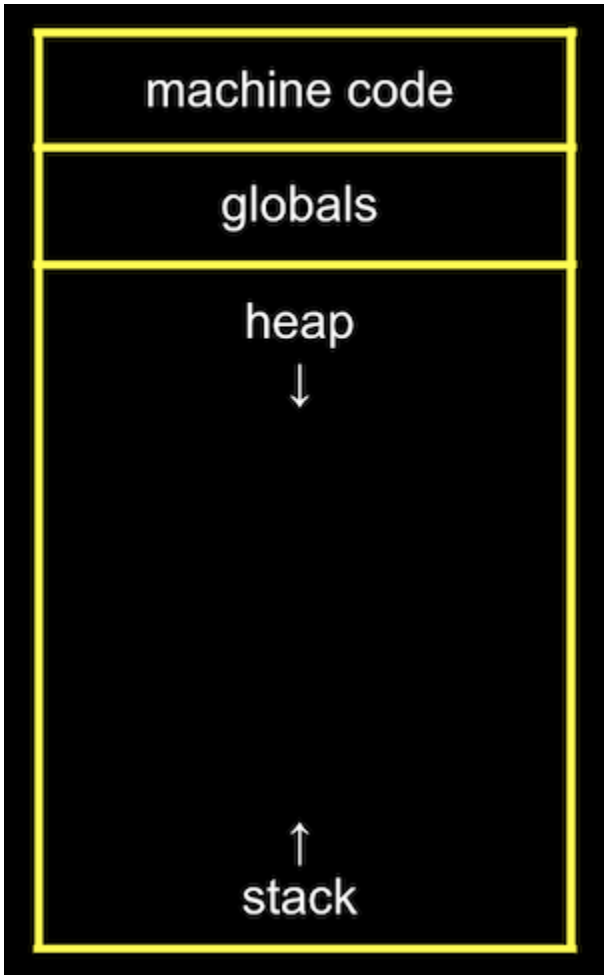
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

- In the real world, if we had a red liquid in one glass, and a blue liquid in another, and we wanted to swap them, we would need a third glass to temporarily hold one of the liquids, perhaps the red glass. Then we can pour the blue liquid into the first glass, and finally the red liquid from the temporary glass into the second one.
- In our `swap` function, we have a third variable to use as temporary storage space as well. We put `a` into `tmp`, and then set `a` to the value of `b`, and finally `b` can be changed to the original value of `a`, now in `tmp`.

- But, if we tried to use that function in a program, we don't see any changes. It turns out that the `swap` function gets its own variables, `a` and `b` when they are passed in, that are copies of `x` and `y`, and so changing those values don't change `x` and `y` in the `main` function.

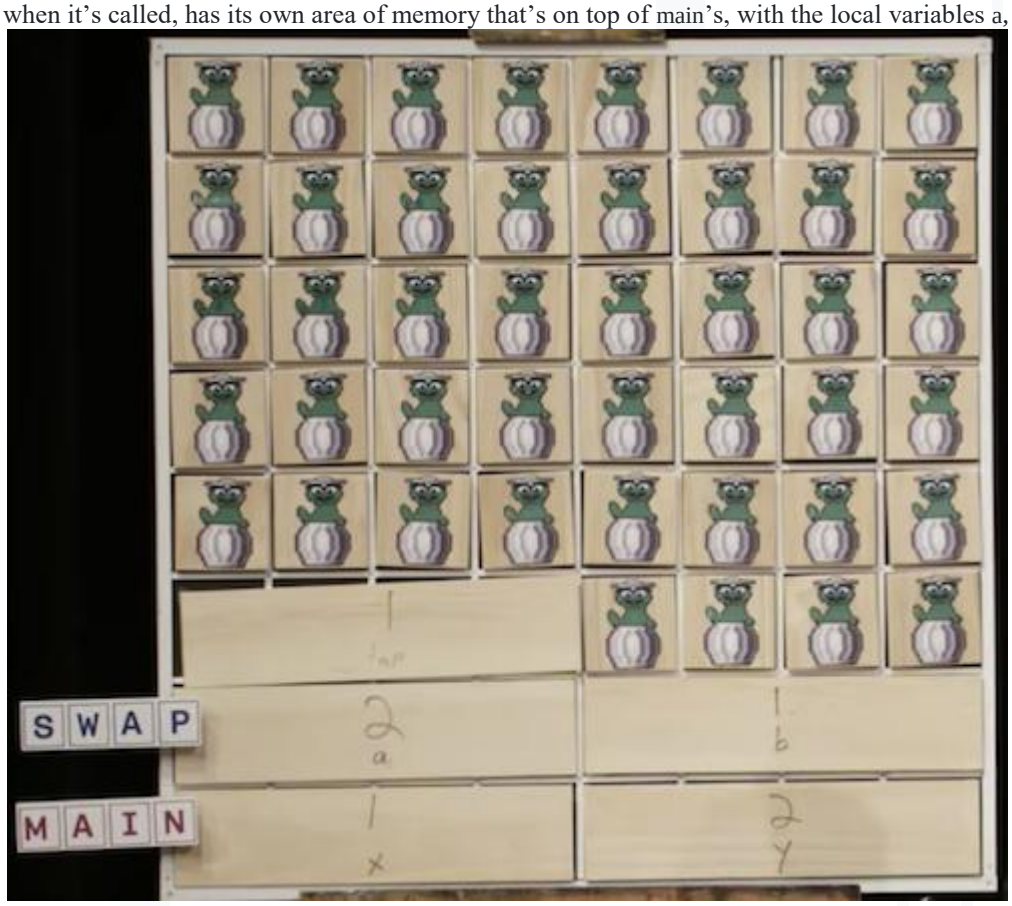
Memory layout

- Within our computer's memory, the different types of data that need to be stored for our program are organized into different sections:



  - The **machine code** section is our compiled program's binary code. When we run our program, that code is loaded into the "top" of memory.
  - Just below, or in the next part of memory, are **global variables** we declare in our program.
  - The **heap** section is an empty area from where `malloc` can get free memory for our program to use. As we call `malloc`, we start allocating memory from the top down.
  - The **stack** section is used by functions in our program as they are called, and grows upwards. For example, our `main` function is at the very bottom of the stack and has the local variables `x` and `y`. The `swap` function,

when it's called, has its own area of memory that's on top of main's, with the local variables a, b, and tmp:



- Once the function swap returns, the memory it was using is freed for the next function call. x and y are arguments, so they're copied as a and b for swap, so we don't see our changes back in main.
- By passing in the address of x and y, our swap function can actually work:

```
#include <stdio.h>

void swap(int *a, int *b);

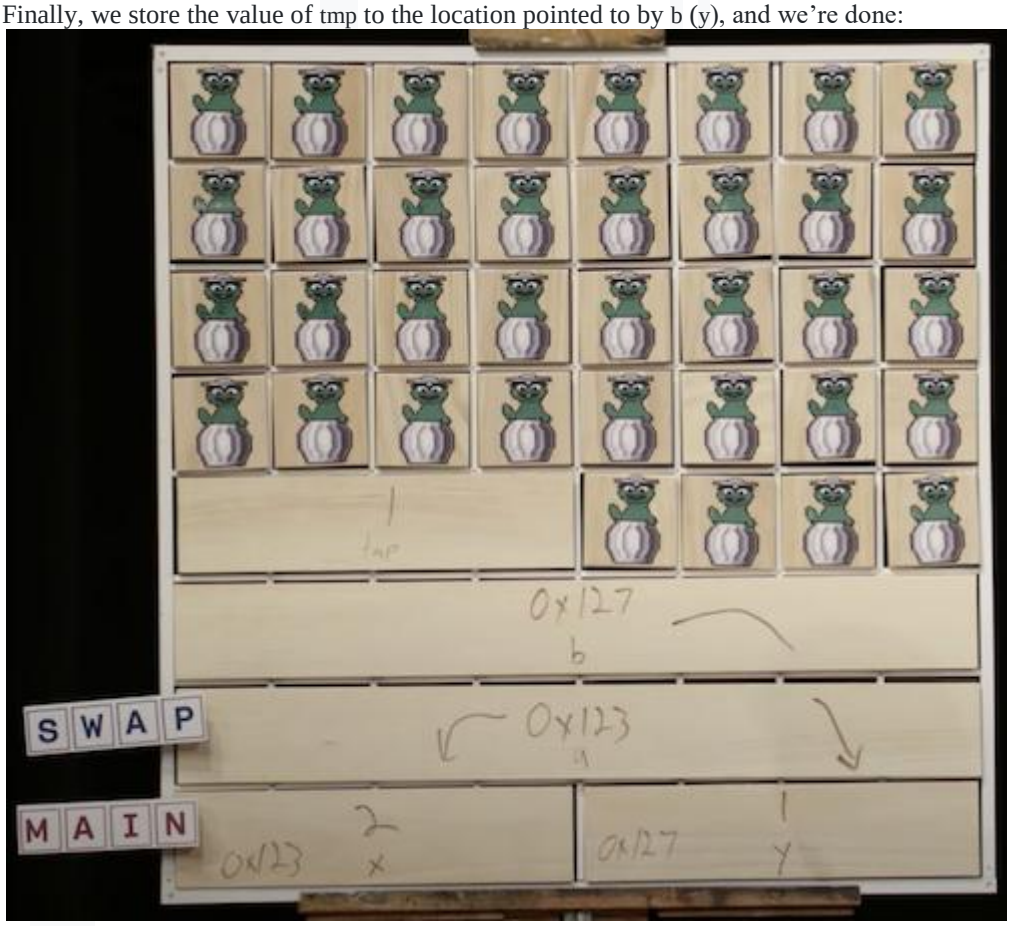
int main(void)
{
   int x = 1;
   int y = 2;

   printf("x is %i, y is %i\n", x, y);
   swap(&x, &y);
   printf("x is %i, y is %i\n", x, y);
}

void swap(int *a, int *b)
{
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

  - The addresses of x and y are passed in from main to swap with &x and &y, and we use the int *a syntax to declare that our swap function takes in pointers. We save the value of x to tmp by following the pointer a, and then take the value of y by following the pointer b, and store that to the location a is pointing to (x).

Finally, we store the value of `tmp` to the location pointed to by `b` (`y`), and we're done:



- If we call `malloc` for too much memory, we will have a **heap overflow**, since we end up going past our heap. Or, if we call too many functions without returning from them, we will have a **stack overflow**, where our stack has too much memory allocated as well.
- Let's implement drawing Mario's pyramid, by calling a function:

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

void draw(int h);

int main(void)
{
    int height = get_int("Height: ");
    draw(height);
}

void draw(int h)
{
    for (int i = 1; i <= h; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

- We can change `draw` to be recursive:

```
void draw(int h)
{
    draw(h - 1);

```

```
    for (int i = 0; i < h; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- When we try to compile this with `make`, we see a warning that the `draw` function will call itself recursively without stopping. So we'll use `clang` without the extra checks, and when we run this program, we get a segmentation fault right away. `draw` is calling itself over and over, and we ran out of memory on the stack.

- By adding a base case, the `draw` function will stop calling itself at some point:

```
void draw(int h)
{
    if (h == 0)
    {
        return;
    }

    draw(h - 1);

    for (int i = 0; i < h; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- But if we enter a large enough value for the height, like `2000000000`, we'll still run out of memory, since we're calling `draw` too many times without returning.

- A **buffer overflow** occurs when we go past the end of a buffer, some chunk of memory we've allocated like an array, and access memory we shouldn't be.

scanf

- We can implement `get_int` ourselves with a C library function, `scanf`:

```
#include <stdio.h>

int main(void)
{
    int x;
    printf("x: ");
    scanf("%i", &x);
    printf("x: %i\n", x);
}
```

- `scanf` takes a format, `%i`, so the input is "scanned" for that format. We also pass in the address in memory where we want that input to go. But `scanf` doesn't have much error checking, so we might not get an integer.

- We can try to get a string the same way:

```
#include <stdio.h>

int main(void)
{
    char *s;
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

- But we haven't actually allocated any memory for `s`, so we need to call `malloc` to allocate memory for characters for our string. We could also use `char s[4];` to declare an array of four characters. Then, `s` will be treated as a pointer to the first character in `scanf` and `printf`.

- Now, if the user types in a string of length 3 or less, our program will work safely. But if the user types in a longer string, `scanf` might be trying to write past the end of our array into unknown memory, causing our program to crash.
- `get_string` from the CS50 library continuously allocates more memory as `scanf` reads in more characters, so it doesn't have that issue.

Files

- With the ability to use pointers, we can also open files, like a digital phone book:

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

int main(void)
{
    FILE *file = fopen("phonebook.csv", "a");
    if (file == NULL)
    {
        return 1;
    }

    char *name = get_string("Name: ");
    char *number = get_string("Number: ");

    fprintf(file, "%s,%s\n", name, number);

    fclose(file);
}
```

  - `fopen` is a new function we can use to open a file. It will return a pointer to a new type, `FILE`, that we can read from and write to. The first argument is the name of the file, and the second argument is the mode we want to open the file in (`r` for read, `w` for write, and `a` for append, or adding to).
  - We'll add a check to exit if we couldn't open the file for some reason.
  - After we get some strings, we can use `fprintf` to print to a file.
  - Finally, we close the file with `fclose`.
- Now we can create our own CSV files, a file of comma-separated values (like a mini-spreadsheet), programmatically.

Graphics

- We can read in binary and map them to pixels and colors, to display images and videos. With a finite number of bits in an image file, though, we can only zoom in so far before we start seeing individual pixels.
  - With artificial intelligence and machine learning, however, we can use algorithms that can generate additional details that weren't there before, by guessing based on other data.
- Let's look at a program that opens a file and tells us if it's a JPEG file, an image file in a particular format:

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

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    // Check usage
    if (argc != 2)
    {
        return 1;
    }

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

```
    if (!file)
    {
        return 1;
    }

    // Read first three bytes
    BYTE bytes[3];
    fread(bytes, sizeof(BYTE), 3, file);

    // Check first three bytes
    if (bytes[0] == 0xff && bytes[1] == 0xd8 && bytes[2] == 0xff)
    {
        printf("Maybe\n");
    }
    else
    {
        printf("No\n");
    }

    // Close file
    fclose(file);
}
```

- First, we define a `BYTE` as 8 bits, so we can refer to a byte as a type more easily in C.
- Then, we try to open a file (checking that we indeed get a non-NULL file back), and read the first three bytes from the file with `fread`, into a buffer called `bytes`.
- We can compare the first three bytes (in hexadecimal) to the three bytes required to begin a JPEG file. If they're the same, then our file is likely to be a JPEG file (though, other types of files may still begin with those bytes). But if they're not the same, we know it's definitely not a JPEG file.

- We can even copy files ourselves, one byte at a time now:

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

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    // Ensure proper usage
    if (argc != 3)
    {
        fprintf(stderr, "Usage: copy SOURCE DESTINATION\n");
        return 1;
    }

    // open input file
    FILE *source = fopen(argv[1], "r");
    if (source == NULL)
    {
        printf("Could not open %s.\n", argv[1]);
        return 1;
    }

    // Open output file
    FILE *destination = fopen(argv[2], "w");
    if (destination == NULL)
    {
        fclose(source);
        printf("Could not create %s.\n", argv[2]);
        return 1;
```

```
    }

    // Copy source to destination, one BYTE at a time
    BYTE buffer;
    while (fread(&buffer, sizeof(BYTE), 1, source))
    {
        fwrite(&buffer, sizeof(BYTE), 1, destination);
    }

    // Close files
    fclose(source);
    fclose(destination);
    return 0;
}
```

    - We use argv to get arguments, using them as filenames to open files to read from and one to write to.
    - Then, we read one byte from the source file into a buffer, and write that byte to the destination file. We can use a while loop to call fread, which will stop once there are no more bytes to read.
- We can use these abilities to read and write files, recovering images from a file, and adding filters to images by changing the bytes in them, in this week's problem set!

Lab 4: Volume

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Write a program to modify the volume of an audio file.

```
$ ./volume input.wav output.wav 2.0
```

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

WAV Files

WAV files are a common file format for representing audio. WAV files store audio as a sequence of "samples": numbers that represent the value of some audio signal at a particular point in time. WAV files begin with a 44-byte "header" that contains information about the file itself, including the size of the file, the number of samples per second, and the size of each sample. After the header, the WAV file contains a sequence of samples, each a single 2-byte (16-bit) integer representing the audio signal at a particular point in time.

Scaling each sample value by a given factor has the effect of changing the volume of the audio. Multiplying each sample value by 2.0, for example, will have the effect of doubling the volume of the origin audio. Multiplying each sample by 0.5, meanwhile, will have the effect of cutting the volume in half.

Types

So far, we've seen a number of different types in C, including `int`, `bool`, `char`, `double`, `float`, and `long`. Inside a header file called `stdint.h` are the declarations of a number of other types that allow us to very precisely define the size (in bits) and sign (signed or unsigned) of an integer. Two types in particular will be useful to us in this lab.

- `uint8_t` is a type that stores an 8-bit unsigned (i.e., not negative) integer. We can treat each byte of a WAV file's header as a `uint8_t` value.
- `int16_t` is a type that stores a 16-bit signed (i.e., positive or negative) integer. We can treat each sample of audio in a WAV file as an `int16_t` value.

Getting Started

1. Log into ide.cs50.io using your GitHub account.
2. In your terminal window, run `wget https://cdn.cs50.net/2020/fall/labs/4/lab4.zip` to download a Zip file of the lab distribution code.
3. In your terminal window, run `unzip lab4.zip` to unzip (i.e., decompress) that Zip file.
4. In your terminal window, run `cd lab4` to change directories into your `lab4` directory.

Implementation Details

Complete the implementation of `volume.c`, such that it changes the volume of a sound file by a given factor.

- The program accepts three command-line arguments: `input` represents the name of the original audio file, `output` represents the name of the new audio file that should be generated, and `factor` is the amount by which the volume of the original audio file should be scaled.
    - For example, if `factor` is `2.0`, then your program should double the volume of the audio file in `input` and save the newly generated audio file in `output`.
- Your program should first read the header from the input file and write the header to the output file. Recall that this header is always exactly 44 bytes long.
    - Note that `volume.c` already defines a variable for you called `HEADER_SIZE`, equal to the number of bytes in the header.
- Your program should then read the rest of the data from the WAV file, one 16-bit (2-byte) sample at a time. Your program should multiply each sample by the `factor` and write the new sample to the output file.
    - You may assume that the WAV file will use 16-bit signed values as samples. In practice, WAV files can have varying numbers of bits per sample, but we'll assume 16-bit samples for this lab.
- Your program, if it uses `malloc`, must not leak any memory.

Walkthrough

Hints

- You'll likely want to create an array of bytes to store the data from the WAV file header that you'll read from the input file. Using the `uint8_t` type to represent a byte, you can create an array of `n` bytes for your header with syntax like

```
uint8_t header[n];
```

replacing `n` with the number of bytes. You can then use `header` as an argument to `fread` or `fwrite` to read into or write from the header.

- You'll likely want to create a "buffer" in which to store audio samples that you read from the WAV file. Using the `int16_t` type to store an audio sample, you can create a buffer variable with syntax like

```
int16_t buffer;
```

You can then use `&buffer` as an argument to `fread` or `fwrite` to read into or write from the buffer. (Recall that the `&` operator is used to get the address of the variable.)

- You may find the documentation for `fread` and `fwrite` helpful here.
  - In particular, note that both functions accept the following arguments:
    - `ptr`: a pointer to the location in memory to store data (when reading from a file) or from which to write data (when writing data to a file)
    - `size`: the number of bytes in an item of data
    - `nmemb`: the number of items of data (each of `size` bytes) to read or write
    - `stream`: the file pointer to be read from or written to
  - Per its documentation, `fread` will return the number of items of data successfully read. You may find this useful to check for when you've reached the end of the file!

How to Test Your Code

Your program should behave per the examples below.

```
$ ./volume input.wav output.wav 2.0
```

When you listen to `output.wav` (as by double-clicking on `output.wav` in the file browser or running `open output.wav` from the terminal), it should be twice as loud as `input.wav`!

```
$ ./volume input.wav output.wav 0.5
```

When you listen to `output.wav`, it should be half as loud as `input.wav`!

Not sure how to solve?

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/labs/2021/x/volume
```

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

```
style50 volume.c
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/volume
```

Problem Set 4

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 4** before beginning this problem set.

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 Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Try out any of David's programs from Week 4.
- 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 filter`

  is yielding errors, try running

  ```
  help50 make filter
  ```

  instead!

Filter-less

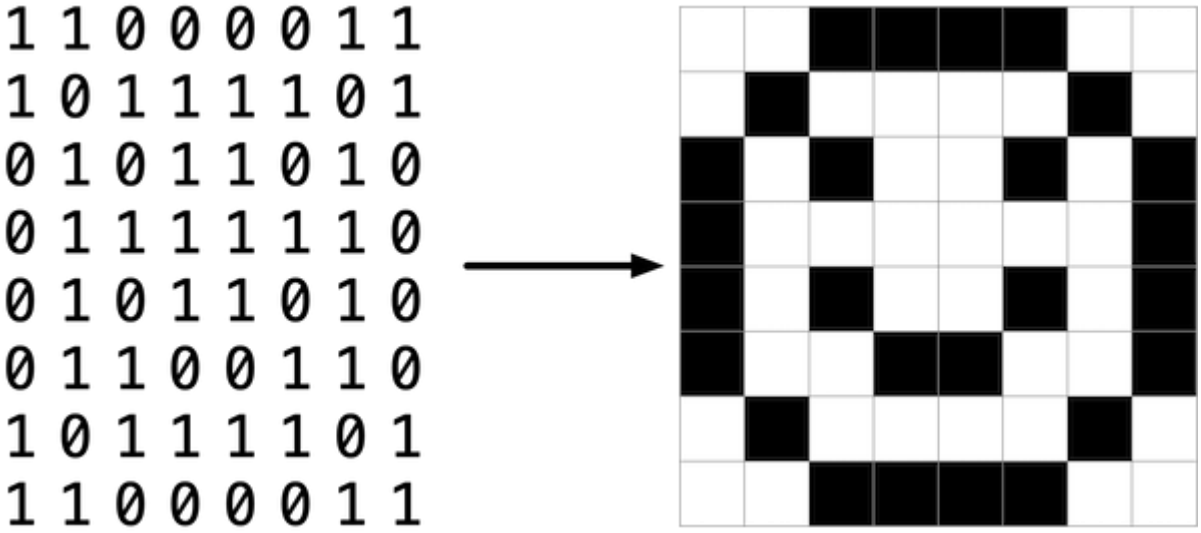
Implement a program that applies filters to BMPs, per the below.

```
$ ./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, JPEG, or PNG) 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 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 and then, in a terminal window, execute each of the below.

- Execute `cd ~` (or simply `cd` with no arguments) to ensure that you're in your home directory.
- Execute `mkdir pset4` to make (i.e., create) a directory called `pset4`.
- Execute `cd pset4` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2020/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 `structs` 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 `structs`.

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 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

Please note that there are 5 videos in this playlist.

Usage

Your program should behave per the examples below.

```
$ ./filter -g infile.bmp outfile.bmp
$ ./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/2021/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/2021/x/filter/less
```

Filter-more

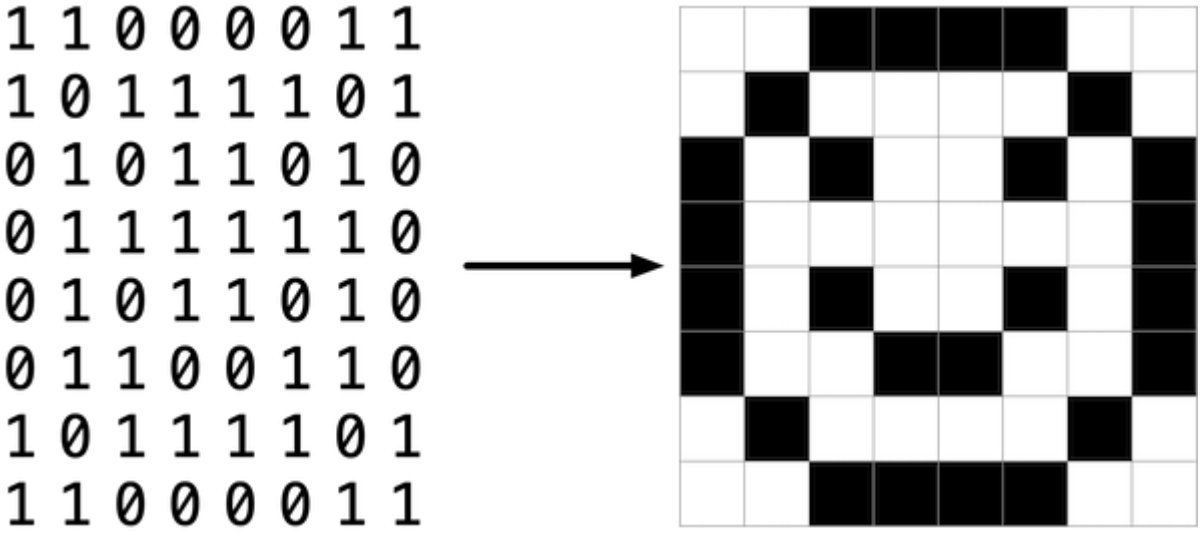
Implement a program that applies filters to BMPs, per the below.

```
$ ./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, JPEG, or PNG) 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 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`.

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 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 and then, in a terminal window, execute each of the below.

- Execute `cd ~` (or simply `cd` with no arguments) to ensure that you're in your home directory.
- Execute `mkdir pset4` to make (i.e., create) a directory called `pset4`.
- Execute `cd pset4` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2020/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 structs 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 structs.

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 RGBTRIPLEs. 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.

Walkthrough

Please note that there are 5 videos in this playlist.

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/2021/x/filter/more
```

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/2021/x/filter/more
```

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 and then, in a terminal window, execute each of the below.

1. Navigate to your `pset4` directory that should already exist.
2. Execute `wget http://cdn.cs50.net/2020/fall/psets/4/recover/recover.zip` to download a (compressed) ZIP file with this problem's distribution.
3. Execute `unzip recover.zip` to uncompress that file.
4. Execute `rm recover.zip` followed by `yes` or `y` to delete that ZIP file.

5. Execute `ls`. You should see a directory called `recover`, which was inside of that ZIP file.
6. Execute `cd recover` to change into that directory.
7. 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`.
- The files you generate should each be named `###.jpg`, where `###` is a three-digit decimal number, starting with `000` for the first image and counting up.
- 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`.) 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`, 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/2021/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.

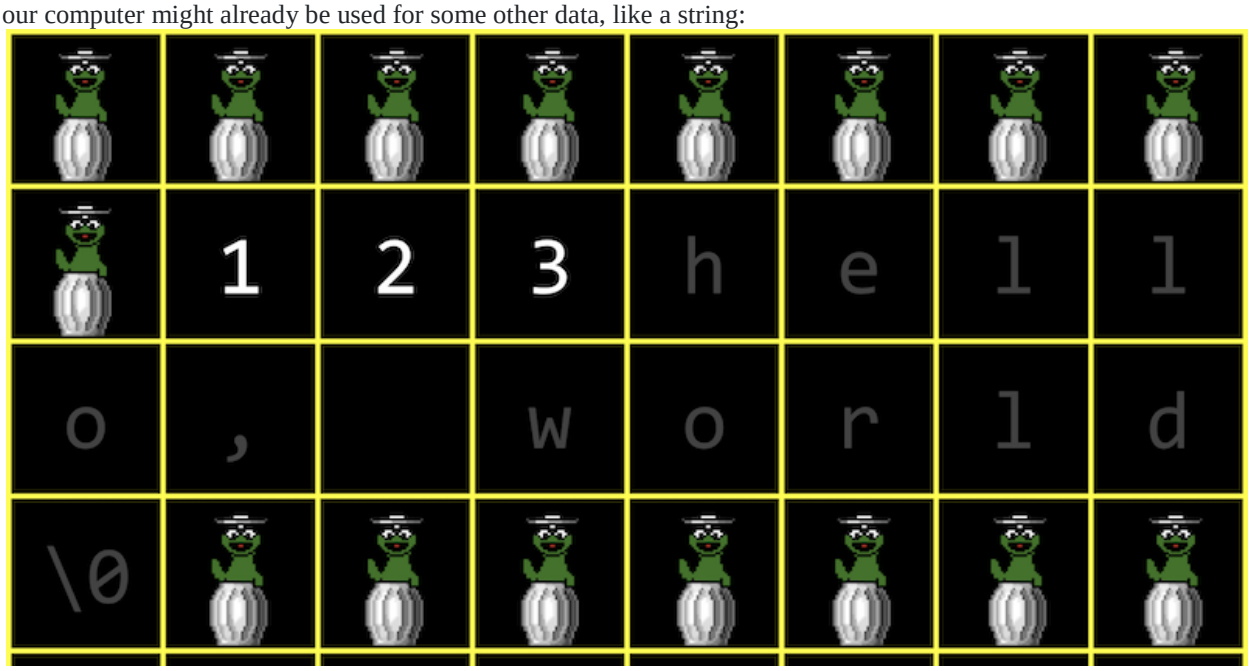
```
submit50 cs50/problems/2021/x/recover
```

Week 5 Data Structures

Lecture 5

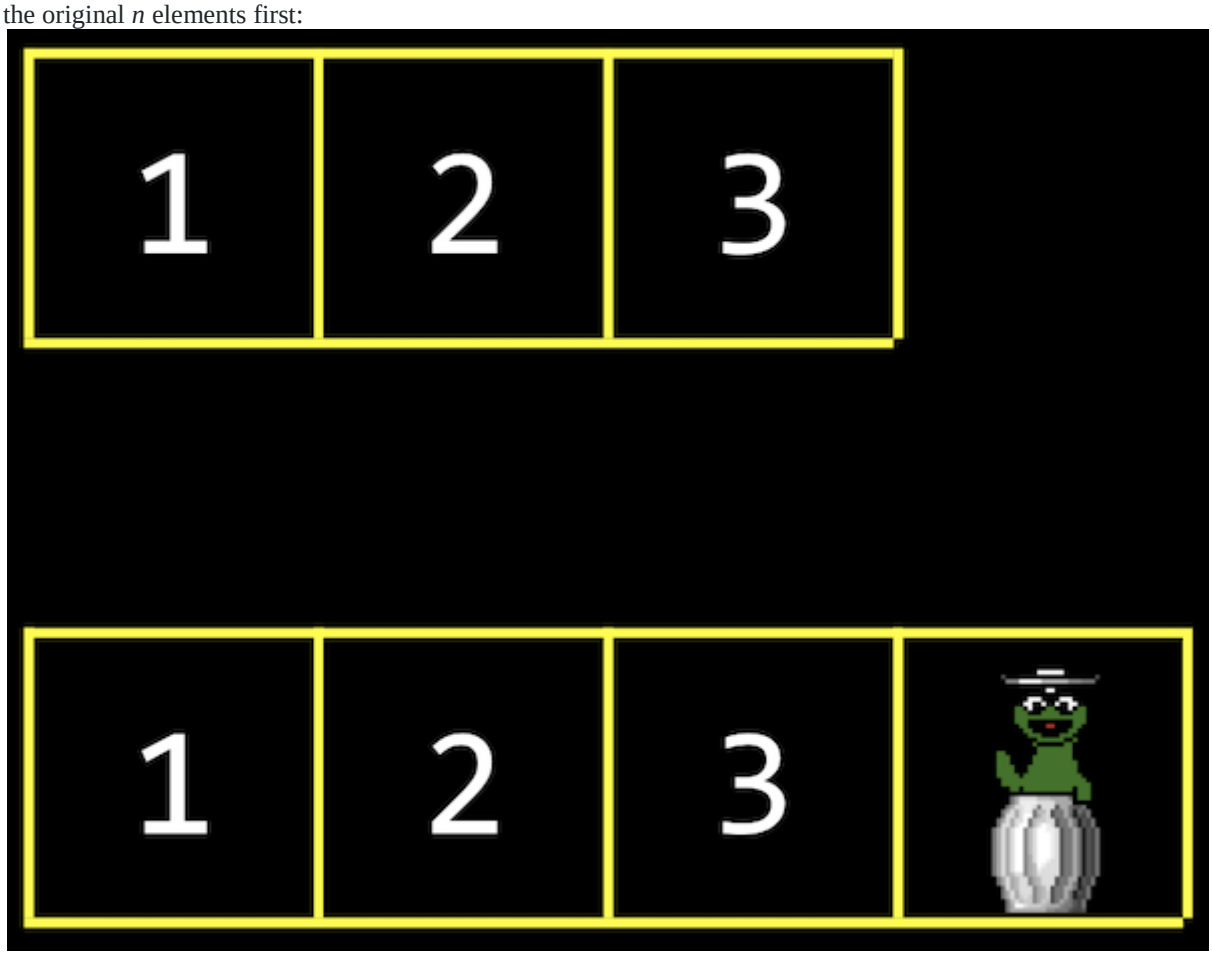
Resizing arrays

- Last time, we learned about pointers, `malloc`, and other useful tools for working with memory.
- In week 2, we learned about arrays, where we could store the same kind of value in a list, back-to-back in memory. When we need to insert an element, we need to increase the size of the array as well. But, the memory after it in our computer might already be used for some other data, like a string:



- One solution might be to allocate more memory where there's enough space, and move our array there. But we'll need to copy our array there, which becomes an operation with running time of $O(n)$, since we need to copy each of

the original *n* elements first:



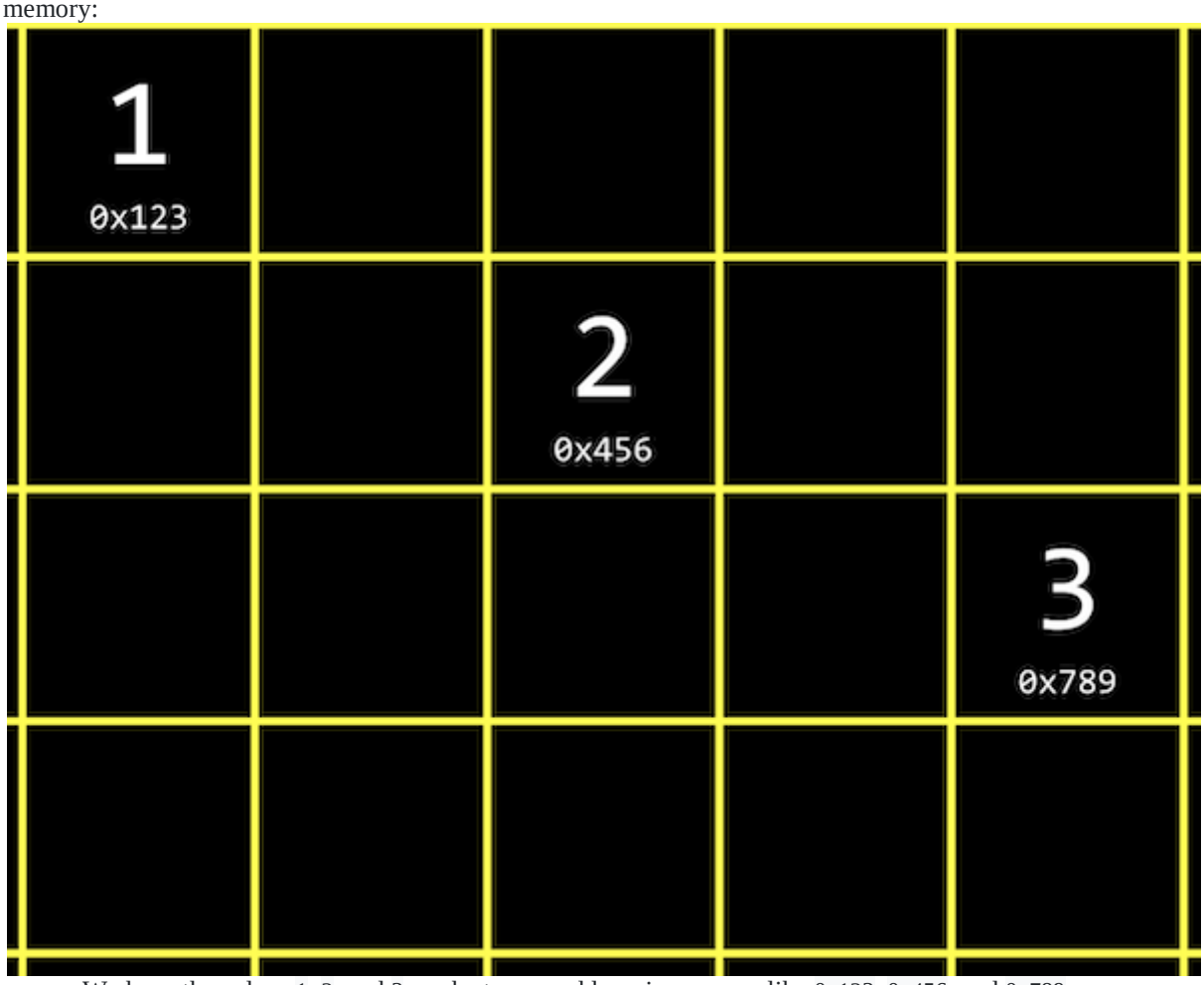
- The lower bound of inserting an element into an array would be O(1) since we might already have space in the array for it.

Data structures

- **Data structures** are more complex ways to organize data in memory, allowing us to store information in different layouts.
- To build a data structure, we'll need some tools:
  - `struct` to create custom data types
  - `.` to access properties in a structure
  - `*` to go to an address in memory pointed to by a pointer
  - `->` to access properties in a structure pointed to by a pointer

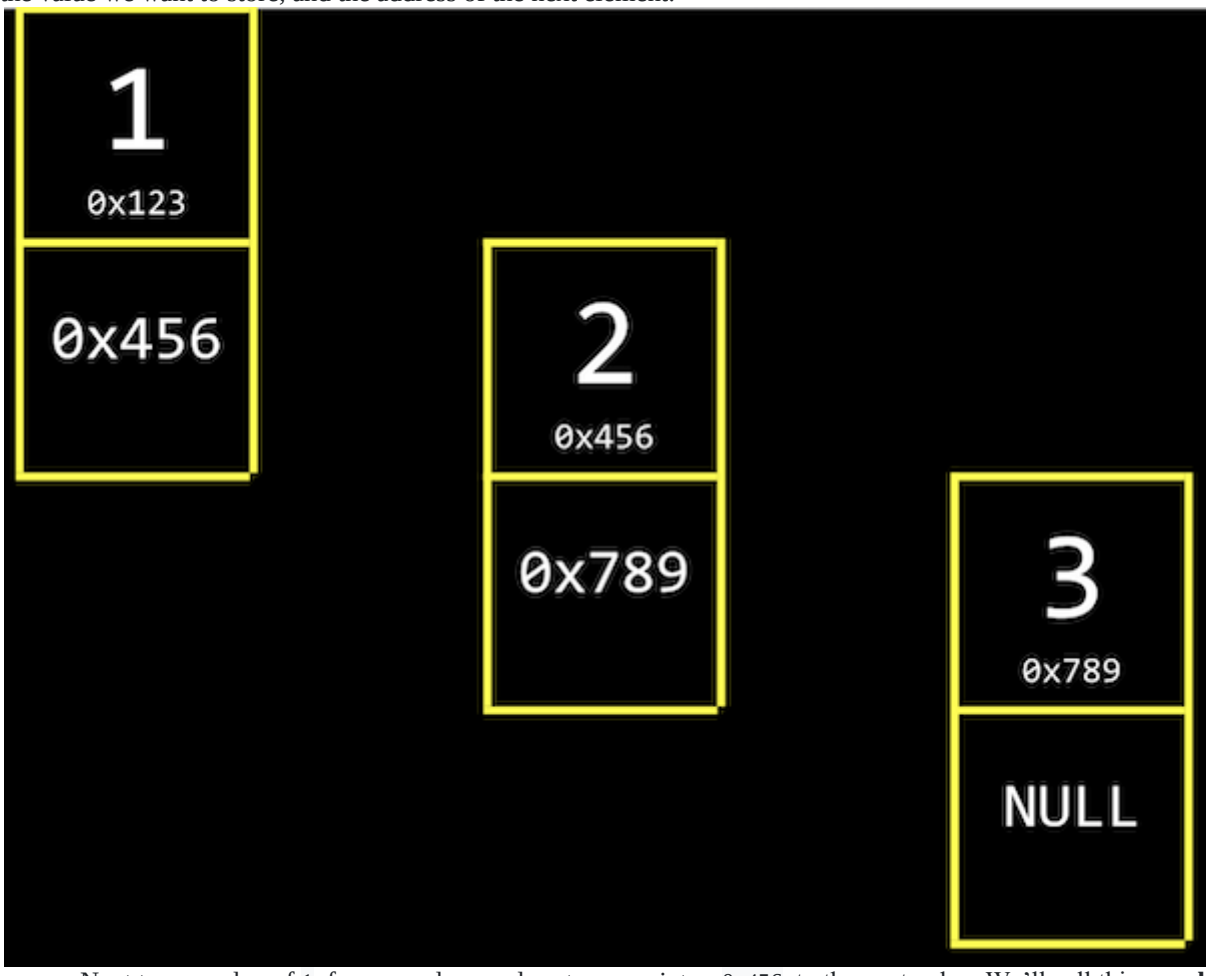
Linked Lists

- With a **linked list**, we can store a list of values that can easily be grown by storing values in different parts of memory:



  - We have the values `1`, `2`, and `3`, each at some address in memory like `0x123`, `0x456`, and `0x789`.
  - This is different than an array since our values are no longer next to one another in memory. We can use whatever locations in memory that are free.

- To track all of these values, we need link our list together by allocating, for each element, enough memory for both the value we want to store, and the address of the next element:



  - Next to our value of 1, for example, we also store a pointer, 0x456, to the next value. We'll call this a **node**, a component of our data structure that stores both a value and a pointer. In C, we'll implement our nodes with a struct.
  - For our last node with value 3, we have the null pointer, since there's no next element. When we need to insert another node, we can just change that single null pointer to point to our new value.
- We have the tradeoff of needing to allocate twice as much memory for each element, in order to spend less time adding values. And we can no longer use binary search, since our nodes might be anywhere in memory. We can only access them by following the pointers, one at a time.
- In code, we might create our own struct called `node`, and we need to store both our value, an `int` called `number`, and a pointer to the next `node`, called `next`:

```
typedef struct node
{
    int number;
    struct node *next;
}
node;
```

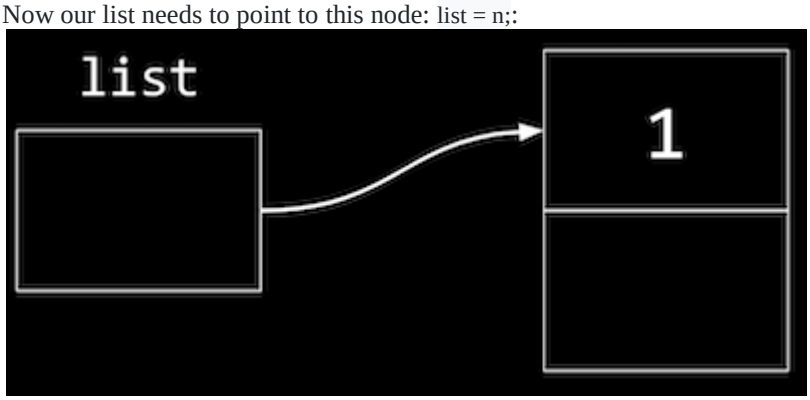
  - We start this struct with `typedef struct node` so that we can refer to a `node` inside our struct.
- We can build a linked list in code starting with our struct. First, we'll want to remember an empty list, so we can use the null pointer: `node *list = NULL;`.
- To add an element, first we'll need to allocate some memory for a node, and set its values:

```
// We use sizeof(node) to get the right amount of memory to allocate, and
// malloc returns a pointer that we save as n
node *n = malloc(sizeof(node));

// We want to make sure malloc succeeded in getting memory for us
if (n != NULL)
{
    // This is equivalent to (*n).number, where we first go to the node pointed
```

```
    // to by n, and then set the number property. In C, we can also use this
    // arrow notation
    n->number = 1;
    // Then we need to make sure the pointer to the next node in our list
    // isn't a garbage value, but the new node won't point to anything (for now)
    n->next = NULL;
}
```

- Now our list needs to point to this node: `list = n;`:



- To add to the list, we'll create a new node the same way by allocating more memory:

```
n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 2;
    n->next = NULL;
}
```

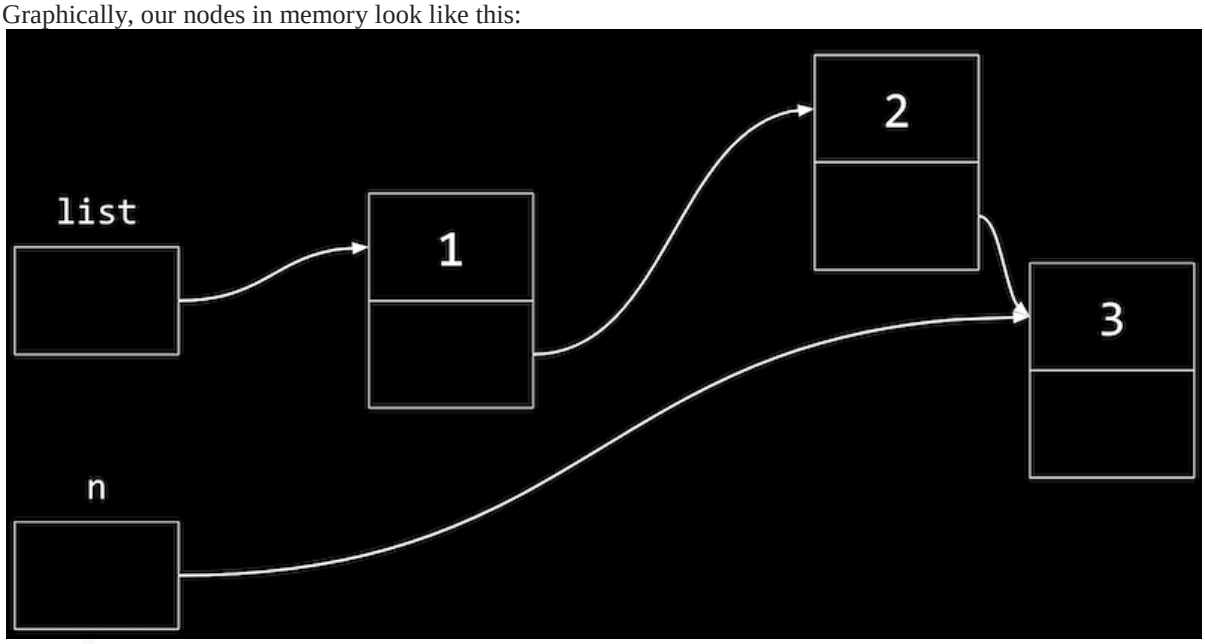
- But now we need to update the pointer in our first node to point to our new `n`:

```
list->next = n;
```

- To add a third node, we'll do the same by following the `next` pointer in our list first, then setting the `next` pointer *there* to point to the new node:

```
n = malloc(sizeof(node));
if (n != NULL)
{
    n->number = 3;
    n->next = NULL;
}
list->next->next = n;
```

- Graphically, our nodes in memory look like this:



  - `n` is a temporary variable, pointing to our new node with value 3.

    - We want the pointer in our node with value 2 to point to the new node as well, so we start from `list` (which points to the node with value 1), follow the `next` pointer to get to our node with value 2, and update the `next` pointer to point to `n`.
- As a result, searching a linked list will also have running time of O(*n*), since we need to look at all elements in order by following each pointer, even if the list is sorted. Inserting into a linked list can have running time of O(1), if we insert new nodes at the beginning of the list.

Implementing arrays

- Let's see how we might implement resizing an array:

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

int main(void)
{
    // Use malloc to allocate enough space for an array with 3 integers
    int *list = malloc(3 * sizeof(int));
    if (list == NULL)
    {
        return 1;
    }

    // Set the values in our array
    list[0] = 1;
    list[1] = 2;
    list[2] = 3;

    // Now if we want to store another value, we can allocate more memory
    int *tmp = malloc(4 * sizeof(int));
    if (tmp == NULL)
    {
        free(list);
        return 1;
    }

    // Copy list of size 3 into list of size 4
    for (int i = 0; i < 3; i++)
    {
        tmp[i] = list[i];
    }

    // Add new number to list of size 4
    tmp[3] = 4;

    // Free original list of size 3
    free(list);

    // Remember new list of size 4
    list = tmp;

    // Print list
    for (int i = 0; i < 4; i++)
    {
        printf("%i\n", list[i]);
    }

    // Free new list
    free(list);
}
```

- Recall that malloc allocates and frees memory from the heap area. It turns out that we can call another library function, realloc, to reallocate some memory that we allocated earlier:
- int *tmp = realloc(list, 4 * sizeof(int));
    - And realloc copies our old array, list, for us into a bigger chunk of memory of the size we pass in. If there happens to be space after our existing chunk of memory, we'll get the same address back, but with the memory after it allocated to our variable as well.

Implementing linked lists

- Let's combine our snippets of code from earlier into a program that implements a linked list:

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

// Represents a node
typedef struct node
{
    int number;
    struct node *next;
}
node;

int main(void)
{
    // List of size 0. We initialize the value to NULL explicitly, so there's
    // no garbage value for our list variable
    node *list = NULL;

    // Allocate memory for a node, n
    node *n = malloc(sizeof(node));
    if (n == NULL)
    {
        return 1;
    }

    // Set the value and pointer in our node
    n->number = 1;
    n->next = NULL;

    // Add node n by pointing list to it, since we only have one node so far
    list = n;

    // Allocate memory for another node, and we can reuse our variable n to
    // point to it, since list points to the first node already
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free(list);
        return 1;
    }

    // Set the values in our new node
    n->number = 2;
    n->next = NULL;

    // Update the pointer in our first node to point to the second node
    list->next = n;

    // Allocate memory for a third node
    n = malloc(sizeof(node));
```

```
    if (n == NULL)
    {
        // Free both of our other nodes
        free(list->next);
        free(list);
        return 1;
    }
    n->number = 3;
    n->next = NULL;

    // Follow the next pointer of the list to the second node, and update
    // the next pointer there to point to n
    list->next->next = n;

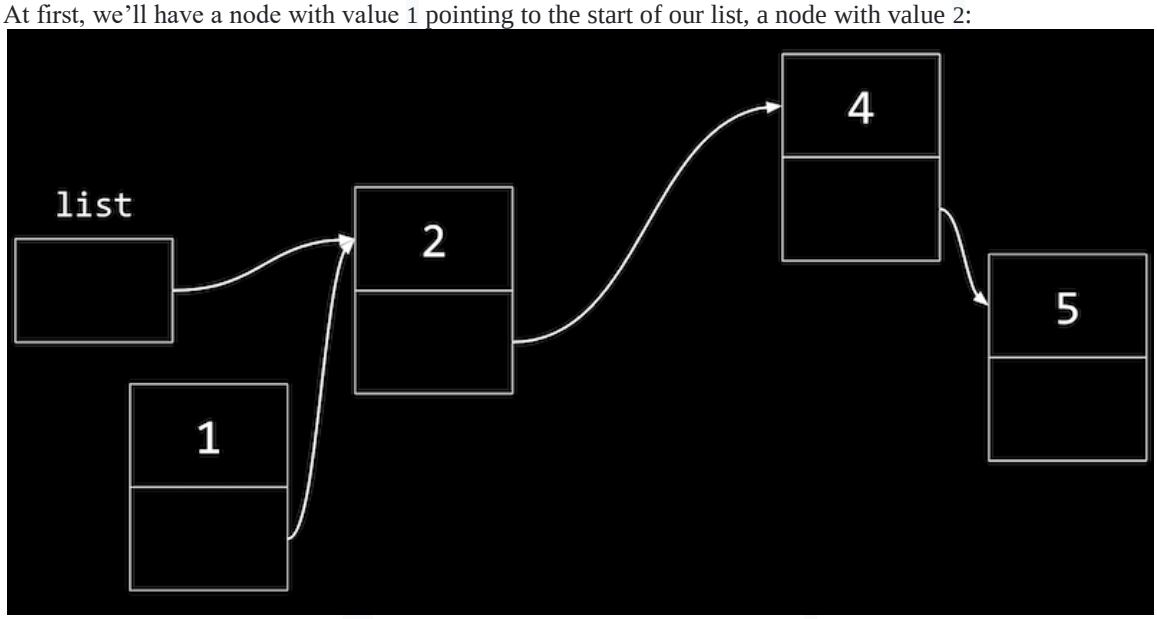
    // Print list using a loop, by using a temporary variable, tmp, to point
    // to list, the first node. Then, every time we go over the loop, we use
    // tmp = tmp->next to update our temporary pointer to the next node. We
    // keep going as long as tmp points to somewhere, stopping when we get to
    // the last node and tmp->next is null.
    for (node *tmp = list; tmp != NULL; tmp = tmp->next)
    {
        printf("%i\n", tmp->number);
    }

    // Free list, by using a while loop and a temporary variable to point
    // to the next node before freeing the current one
    while (list != NULL)
    {
        // We point to the next node first
        node *tmp = list->next;
        // Then, we can free the first node
        free(list);
        // Now we can set the list to point to the next node
        list = tmp;
        // If list is null, when there are no nodes left, our while loop will stop
    }
}
```

If we want to insert a node to the front of our linked list, we would need to carefully update our node to point to the one following it, before updating the list variable. Otherwise, we'll lose the rest of our list:

```
// Here, we're inserting a node into the front of the list, so we want its
// next pointer to point to the original list. Then we can change the list to
// point to n.
n->next = list;
list = n;
```

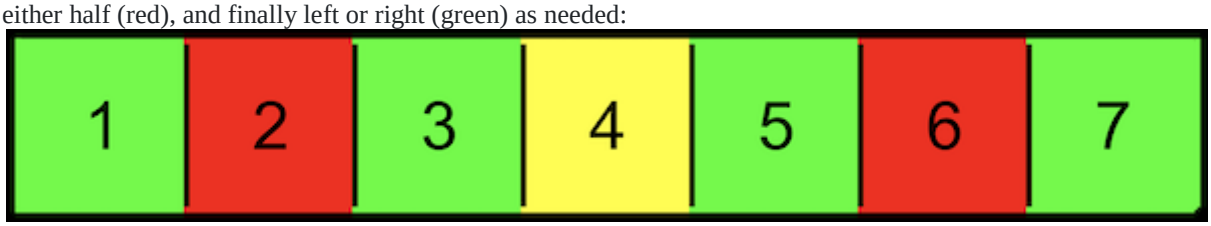
- At first, we'll have a node with value `1` pointing to the start of our list, a node with value `2`:



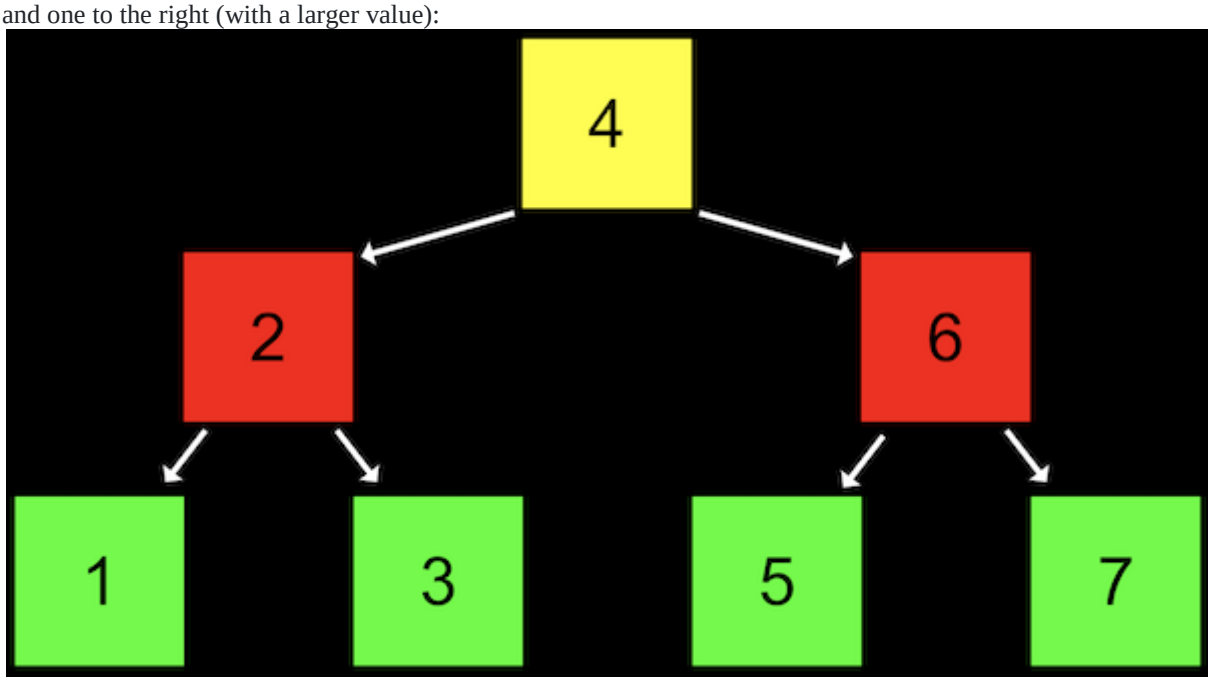
    - Now we can update our `list` variable to point to the node with value `1`, and not lose the rest of our list.
- Similarly, to insert a node in the middle of our list, we change the `next` pointer of the new node first to point to the rest of the list, then update the previous node to point to the new node.
- A linked list demonstrates how we can use pointers to build flexible data structures in memory, though we're only visualizing it in one dimension.

Trees

- With a sorted array, we can use binary search to find an element, starting at the middle (yellow), then the middle of either half (red), and finally left or right (green) as needed:



    - With an array, we can randomly access elements in O(1) time, since we can use arithmetic to go to an element at any index.
- A **tree** is another data structure where each node points to two other nodes, one to the left (with a smaller value) and one to the right (with a larger value):



    - Notice that we now visualize this data structure in two dimensions (even though the nodes in memory can be at any location).
    - And we can implement this with a more complex version of a node in a linked list, where each node has not one but two pointers to other nodes. All the values to the left of a node are smaller, and all the values of nodes to the right are greater, which allows this to be used as a **binary search tree**. And the data structure is itself defined recursively, so we can use recursive functions to work with it.
    - Each node has at most two **children**, or nodes it is pointing to.
    - And like a linked list, we'll want to keep a pointer to just the beginning of the list, but in this case we want to point to the **root**, or top center node of the tree (the 4).
- We can define a node with not one but two pointers:

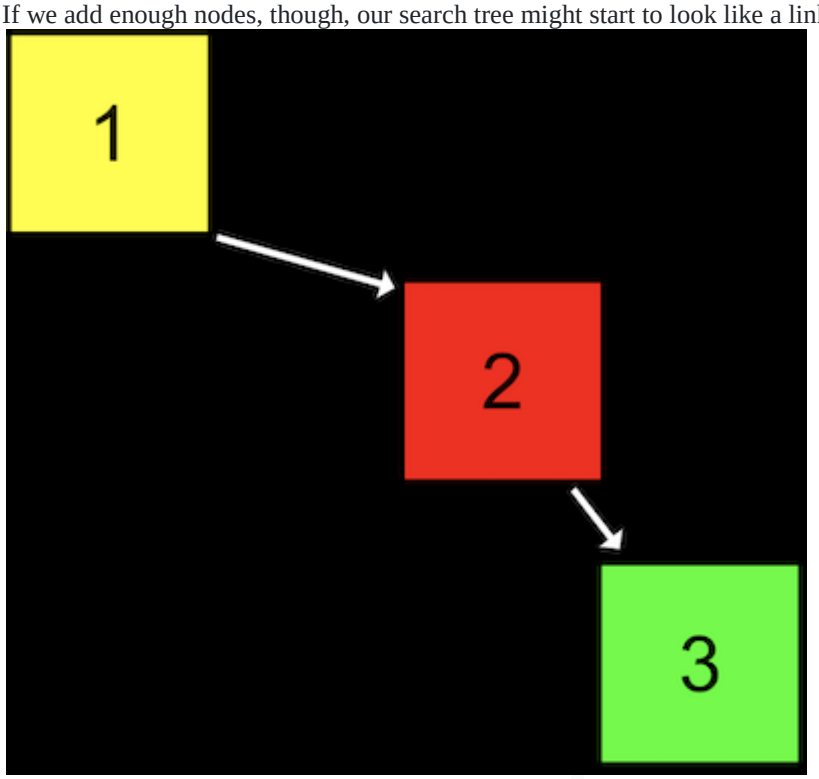
```
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;
```

- And write a function to recursively search a tree:

```
// tree is a pointer to a node that is the root of the tree we're searching in.
// number is the value we're trying to find in the tree.
bool search(node *tree, int number)
{
    // First, we make sure that the tree isn't NULL, if we've reached a node
    // on the bottom, or if our tree is entirely empty
    if (tree == NULL)
    {
        return false;
    }
    // If we're looking for a number that's less than the tree's number,
    // search the left side, using the node on the left as the new root
    else if (number < tree->number)
    {
        return search(tree->left, number);
    }
    // Otherwise, search the right side, using the node on the right as the new root
    else if (number > tree->number)
    {
        return search(tree->right, number);
    }
    // Finally, we've found the number we're looking for, so we can return true.
    // We can simplify this to just "else", since there's no other case possible
    else if (number == tree->number)
    {
        return true;
    }
}
```

- With a binary search tree, we've incurred the cost of even more memory, since each node now needs space for a value and two pointers. Inserting a new value would take O(log *n*) time, since we need to find the nodes that it should go between.

- If we add enough nodes, though, our search tree might start to look like a linked list:

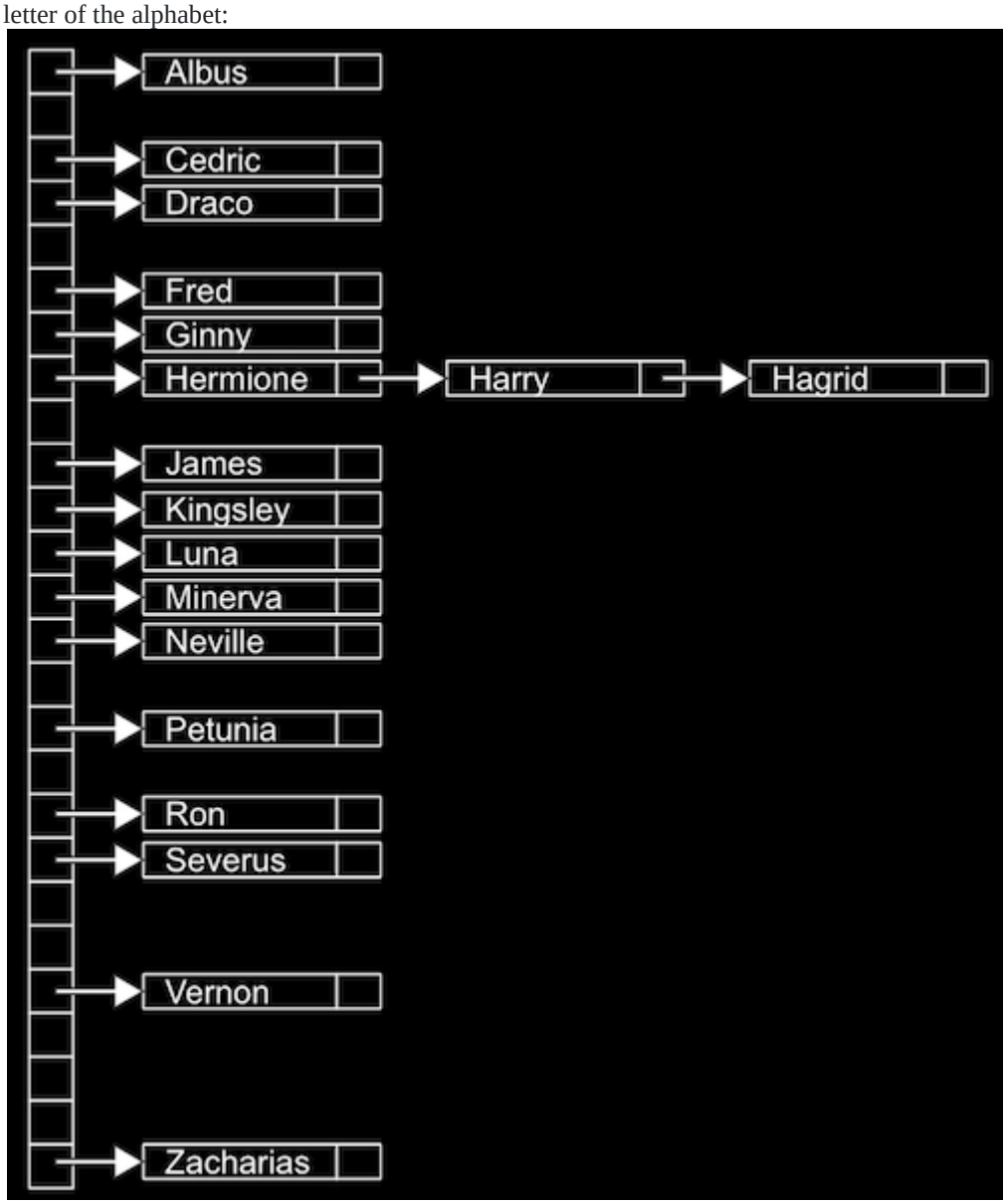


  - We started our tree with a node with value of 1, then added the node with value 2, and finally added the node with value 3. Even though this tree follows the constraints of a binary search tree, it's not as efficient as it could be.
  - We can make the tree balanced, or optimal, by making the node with value 2 the new root node. More advanced courses will cover data structures and algorithms that help us keep trees balanced as nodes are added.

More data structures

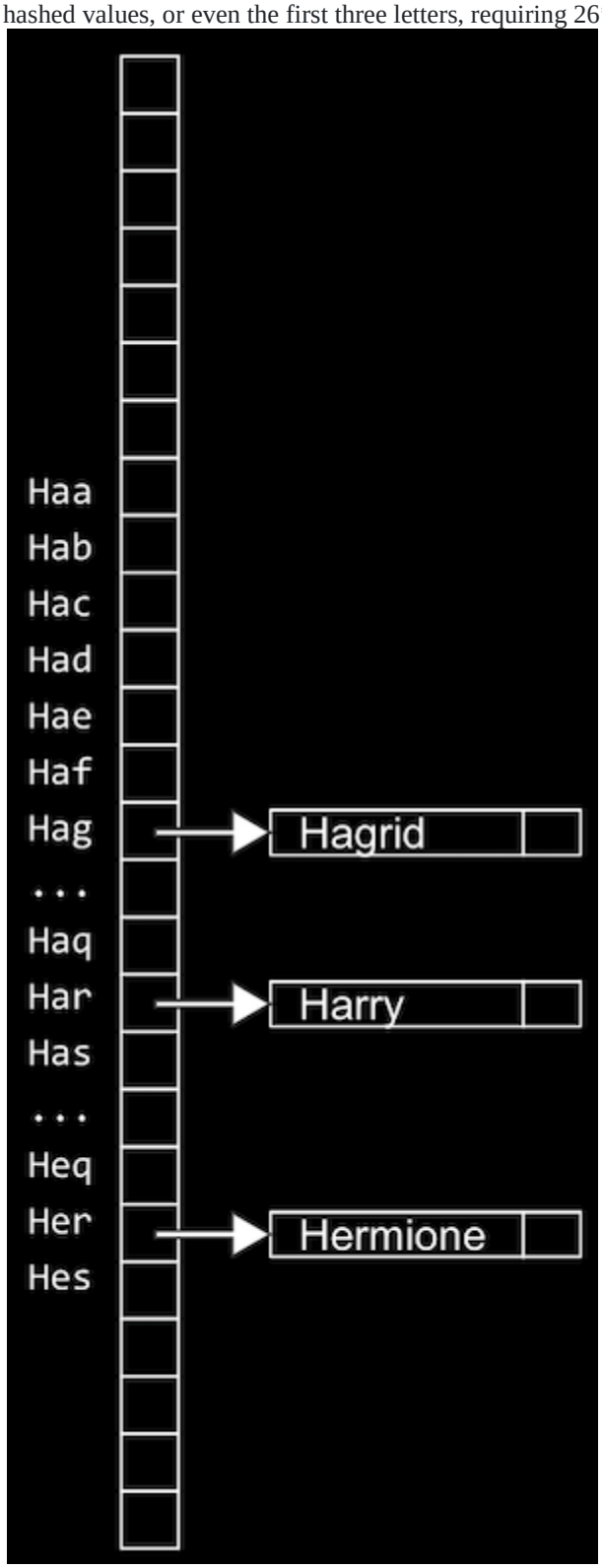
- A data structure with almost a constant time, O(1) search is a **hash table**, which is essentially an array *of* linked lists. Each linked list in the array has elements of a certain category.

- For example, we might have lots of names, and we might sort them into an array with 26 positions, one for each letter of the alphabet:



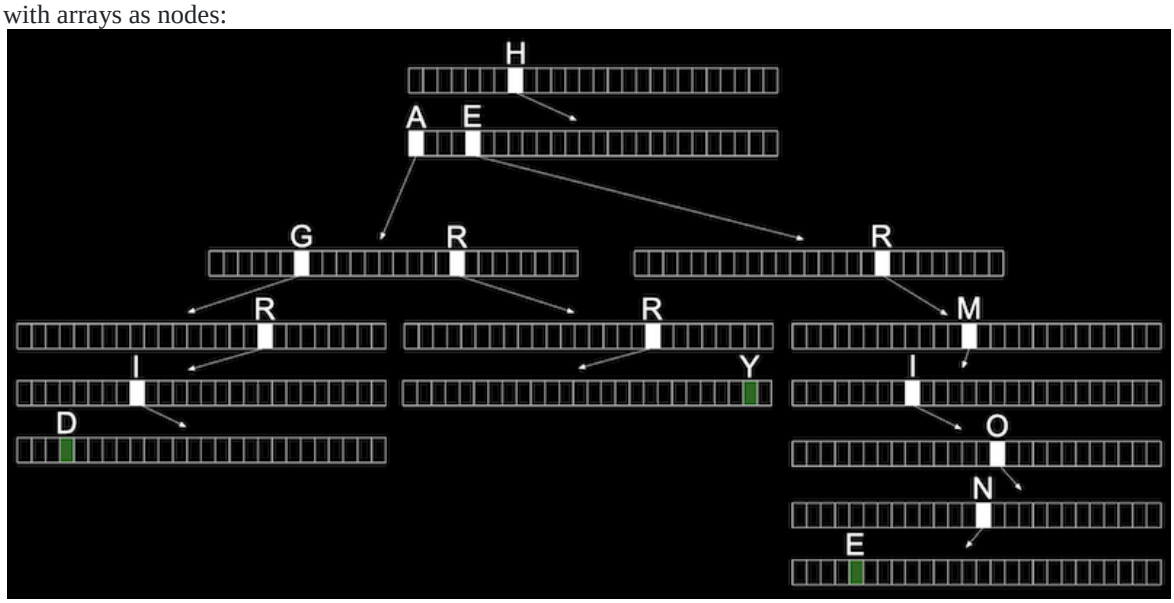
    - Since we have random access with arrays, we can set elements and index into a location, or bucket, in the array quickly.
    - A location might have multiple matching values, but we can add a value to another value since they're nodes in a linked list, as we see with Hermione, Harry, and Hagrid. We don't need to grow the size of our array or move any of our other values.
- This is called a hash table because we use a **hash function**, which takes some input and deterministically maps it to the location it should go in. In our example, the hash function just returns an index corresponding to the first letter of the name, such as 0 for "Albus" and 25 for "Zacharias".
- But in the worst case, all the names might start with the same letter, so we might end up with the equivalent of a single linked list again. We might look at the first two letters, and allocate enough buckets for 26*26 possible

hashed values, or even the first three letters, requiring 26*26*26 buckets:



    - Now, we're using more space in memory, since some of those buckets will be empty, but we're more likely to only need one step to look for a value, reducing our running time for search.
- To sort some standard playing cards, too, we might first start with putting them in piles by suit, of spades, diamonds, hearts, and clubs. Then, we can sort each pile a little more quickly.
- It turns out that the worst case running time for a hash table is O(*n*), since, as *n* gets very large, each bucket will have on the order of *n* values, even if we have hundreds or thousands of buckets. In practice, though, our running time will be faster since we're dividing our values into multiple buckets.
- In problem set 5, we'll be challenged to improve the real world running time of searching for values in our data structures, while also balancing our use of memory.

- We can use another data structure called a **trie** (pronounced like "try", and is short for "retrieval"). A trie is a tree with arrays as nodes:



  - Each array will have each letter, A-Z, stored. For each word, the first letter will point to an array, where the next valid letter will point to another array, and so on, until we reach a boolean value indicating the end of a valid word, marked in green above. If our word isn't in the trie, then one of the arrays won't have a pointer or terminating character for our word.
  - In the trie above, we have the words Hagrid, Harry, and Hermione.
  - Now, even if our data structure has lots of words, the maximum lookup time will be just the length of the word we're looking for. This might be a fixed maximum, so we can have $O(1)$ for searching and insertion.
  - The cost for this, though, is that we need lots of memory to store pointers and boolean values as indicators of valid words, even though lots of them won't be used.
- There are even higher-level constructs, **abstract data structures**, where we use our building blocks of arrays, linked lists, hash tables, and tries to *implement* a solution to some problem.
- For example, one abstract data structure is a **queue**, like a line of people waiting, where the first value we put in are the first values that are removed, or first-in-first-out (FIFO). To add a value we **enqueue** it, and to remove a value we **dequeue** it. This data structure is abstract because it's an idea that we can implement in different ways: with an array that we resize as we add and remove items, or with a linked list where we append values to the end.
- An "opposite" data structure would be a **stack**, where items most recently added are removed first: last-in-first-out (LIFO). At a clothing store, we might take, or **pop**, the top sweater from a stack, and new sweaters would be added, or **pushed**, to the top as well.
- Another example of an abstract data structure is a **dictionary**, where we can map keys to values, such as words to their definitions. We can implement one with a hash table or an array, taking into account the tradeoff between time and space.
- We take a look at "Jack Learns the Facts About Queues and Stacks", an animation about these data structures.

Lab 5: Inheritance

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Simulate the inheritance of blood types for each member of a family.

```
$ ./inheritance
Generation 0, blood type OO
    Generation 1, blood type AO
        Generation 2, blood type OA
        Generation 2, blood type BO
    Generation 1, blood type OB
        Generation 2, blood type AO
        Generation 2, blood type BO
```

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Background

A person's blood type is determined by two alleles (i.e., different forms of a gene). The three possible alleles are A, B, and O, of which each person has two (possibly the same, possibly different). Each of a child's parents randomly passes one of their two blood type alleles to their child. The possible blood type combinations, then, are: OO, OA, OB, AO, AA, AB, BO, BA, and BB.

For example, if one parent has blood type AO and the other parent has blood type BB, then the child's possible blood types would be AB and OB, depending on which allele is received from each parent. Similarly, if one parent has blood type AO and the other OB, then the child's possible blood types would be AO, OB, AB, and OO.

Getting Started

Create a new directory in your IDE called `lab5`. In that directory, execute `wget https://cdn.cs50.net/2020/fall/labs/5/inheritance.c` to download the distribution code for this project.

Understanding

Take a look at the distribution code in `inheritance.c`.

Notice the definition of a type called `person`. Each person has an array of two `parents`, each of which is a pointer to another `person` struct. Each person also has an array of two `alleles`, each of which is a `char` (either `'A'`, `'B'`, or `'O'`).

Now, take a look at the `main` function. The function begins by "seeding" (i.e., providing some initial input to) a random number generator, which we'll use later to generate random alleles. The `main` function then calls the `create_family` function to simulate the creation of `person` structs for a family of 3 generations (i.e. a person, their parents, and their grandparents). We then call `print_family` to print out each of those family members and their blood types. Finally, the function calls `free_family` to free any memory that was previously allocated with `malloc`.

The `create_family` and `free_family` functions are left to you to write!

Implementation Details

Complete the implementation of `inheritance.c`, such that it creates a family of a specified generation size and assigns blood type alleles to each family member. The oldest generation will have alleles assigned randomly to them.

- The `create_family` function takes an integer (`generations`) as input and should allocate (as via `malloc`) one `person` for each member of the family of that number of generations, returning a pointer to the `person` in the youngest generation.
    - For example, `create_family(3)` should return a pointer to a person with two parents, where each parent also has two parents.

    - Each person should have alleles assigned to them. The oldest generation should have alleles randomly chosen (as by calling the random_allele function), and younger generations should inherit one allele (chosen at random) from each parent.
    - Each person should have parents assigned to them. The oldest generation should have both parents set to NULL, and younger generations should have parents be an array of two pointers, each pointing to a different parent.

We've divided the create_family function into a few TODOs for you to complete.

- First, you should allocate memory for a new person. Recall that you can use malloc to allocate memory, and sizeof(person) to get the number of bytes to allocate.
- Next, we've included a condition to check if generations > 1.
    - If generations > 1, then there are more generations that still need to be allocated. Your function should set both parents by recursively calling create_family. (How many generations should be passed as input to each parent?) The function should then set both alleles by randomly choosing one allele from each parent.
    - Otherwise (if generations == 1), then there will be no parent data for this person. Both parents should be set to NULL, and each allele should be generated randomly.
- Finally, your function should return a pointer for the person that was allocated.

The free_family function should accept as input a pointer to a person, free memory for that person, and then recursively free memory for all of their ancestors.

- Since this is a recursive function, you should first handle the base case. If the input to the function is NULL, then there's nothing to free, so your function can return immediately.
- Otherwise, you should recursively free both of the person's parents before freeing the child.

Walkthrough

Hints

- You might find the rand() function useful for randomly assigning alleles. This function returns an integer between 0 and RAND_MAX, or 32767.
    - In particular, to generate a pseudorandom number that is either 0 or 1, you can use the expression rand() % 2.
- Remember, to allocate memory for a particular person, we can use malloc(n), which takes a size as argument and will allocate n bytes of memory.
- Remember, to access a variable via a pointer, we can use arrow notation.
    - For example, if p is a pointer to a person, then a pointer to this person's first parent can be accessed by p->parents[0].

How to Test Your Code

Upon running ./inheritance, your program should adhere to the rules described in the background. The child should have two alleles, one from each parent. The parents should each have two alleles, one from each of their parents.

For example, in the example below, the child in Generation 0 received an O allele from both Generation 1 parents. The first parent received an A from the first grandparent and a O from the second grandparent. Similarly, the second parent received an O and a B from their grandparents.

```
$ ./inheritance
Generation 0, blood type OO
  Generation 1, blood type AO
    Generation 2, blood type OA
    Generation 2, blood type BO
  Generation 1, blood type OB
    Generation 2, blood type AO
    Generation 2, blood type BO
```

Not sure how to solve?

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/labs/2021/x/inheritance
```

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

```
style50 inheritance.c
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/inheritance
```

Problem Set 5

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 5** before beginning this problem set.

1. Submit Speller

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Try out any of David's programs from Week 5.
- 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!

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 and then, in a terminal window, execute each of the below.

- Execute `cd ~/` (or simply `cd` with no arguments) to ensure that you're in your home directory.
- Execute `mkdir pset5` to make (i.e., create) a directory called `pset5`.
- Execute `cd pset5` to change into (i.e., open) that directory.
- Execute `wget http://cdn.cs50.net/2021/spring/psets/5/speller/speller.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip speller.zip` to uncompress that file.
- Execute `rm speller.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `speller`, which was inside of that ZIP file.
- Execute `cd speller` to change into that directory.
- 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.

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, dictionary.h, 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.
- 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 (declared in strings.h) useful! You'll likely also want to ensure that your hash function is case-insensitive, such that foo and FOO have the same hash value.

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/2021/x/speller
```
Note that `check50` will also check for memory leaks, so be sure you've run `valgrind` as well.

style50

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

```
style50 dictionary.c
```

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

If you've worked on this problem in the past, you know that we formerly had a "Big Board" against which students could compare their benchmark times for their spell-checker against other students. After some consideration, afraid we've decided to retire the Big Board for 2021!

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/2021/x/speller
```

Week 6 Python

Lecture 6

Python Basics

- Today we'll learn a new programming language called Python. A newer language than C, it has additional features as well as simplicity, leading to its popularity.
- Source code in Python looks a lot simpler than C. In fact, to print "hello, world", all we need to write is:

```
print("hello, world")
```

  - Notice that, unlike in C, we don't need to specify a newline in the `print` function or use a semicolon to end our line.
  - To write and run this program, we'll use the CS50 IDE, save a new file as `hello.py` with just the line above, and run the command `python hello.py`.
- We can get strings from a user:

```
answer = get_string("What's your name? ")
print("hello, " + answer)
```

  - We also need to import the Python version of the CS50 library, `cs50`, for just the function `get_string`, so our code will look like this:

```
from cs50 import get_string

answer = get_string("What's your name? ")
print("hello, " + answer)
```

  - We create a variable called `answer` without specifying the type, and we can combine, or concatenate, two strings with the `+` operator before we pass it into `print`.
- We can use the syntax for **format strings**, `f"..."`, to plug in variables. For example, we could have written `print(f"hello, {answer}")` to plug in the value of `answer` into our string by surrounding it with curly braces.
- We can create variables with just `counter = 0`. By assigning the value of `0`, we're implicitly setting the type to an integer, so we don't need to specify the type. To increment a variable, we can use `counter = counter + 1` or `counter += 1`.
- Conditions look like:

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

  - Unlike in C, where curly braces are used to indicate blocks of code, the exact indentation of each line is what determines the level of nesting in Python.
  - And instead of `else if`, we just say `elif`.
- Boolean expressions are slightly different, too:

```
while True:
    print("hello, world")
```

  - Both `True` and `False` are capitalized in Python.
- We can write a loop with a variable:

```
i = 0
while i < 3:
```

```
    print("hello, world")
    i += 1
```

- We can also use a `for` loop, where we can do something for each value in a list:

```
for i in [0, 1, 2]:
    print("cough")
```

  - Lists in Python, `[0, 1, 2]`, are like arrays in C.
  - This `for` loop will set the variable `i` to the first element, `0`, run, then to the second element, `1`, run, and so on.
  - And we can use a special function, `range`, to get some number of values, as in `for i in range(3):`. `range(3)` will give us a list up to but not including 3, with the values `0`, `1`, and `2`, that we can then use. `range()` takes other options as well, so we can have lists that start at different values and have different increments between values. By looking at the documentation, for example, we can use `range(0, 101, 2)` to get a range from `0` to `100` (since the second value is exclusive), incrementing by `2` at a time.
  - To print out `i`, too, we can just write `print(i)`.
  - Since there are often multiple ways to write the same code in Python, the most commonly used and accepted ways are called **Pythonic**.
- In Python, there are many built-in data types:
  - `bool`, `True` or `False`
  - `float`, real numbers
  - `int`, integers
  - `str`, strings
- While C is a **strongly typed** language, where we need to specify types, Python is **loosely typed**, where the type is implied by the values.
- Other types in Python include:
  - `range`, sequence of numbers
  - `list`, sequence of mutable values, or values we can change
    - And lists, even though they're like arrays in C, can grow and shrink automatically in Python
  - `tuple`, collection of ordered values like x- and y-coordinates, or longitude and latitude
  - `dict`, dictionaries, collection of key/value pairs, like a hash table
  - `set`, collection of unique values, or values without duplicates
- The CS50 library for Python includes:
  - `get_float`
  - `get_int`
  - `get_string`
- And we can import functions one at a time, or all together:

```
from cs50 import get_float
from cs50 import get_int
from cs50 import get_string
import cs50
from cs50 import get_float, get_int, get_string
```

Examples

- Since Python includes many features as well as libraries of code written by others, we can solve problems at a higher level of abstraction, instead of implementing all the details ourselves.
- We can blur an image with:

```
from PIL import Image, ImageFilter

before = Image.open("bridge.bmp")
after = before.filter(ImageFilter.BoxBlur(1))
after.save("out.bmp")
```

  - In Python, we include other libraries with `import`, and here we'll `import` the `Image` and `ImageFilter` names from the `PIL` library. (Other people have written this library, among others, and made it available for all of us to download and use.)
  - `Image` is a structure that not only has data, but functions that we can access with the `.` syntax, such as with `Image.open`.
  - We open an image called `bridge.bmp`, call a blur filter function, and save it to a file called `out.bmp`.
  - And we can run this with `python blur.py` after saving to a file called `blur.py`.
- We can implement a dictionary with:

```
words = set()

def load(dictionary):
```

```
    file = open(dictionary, "r")
    for line in file:
        words.add(line.rstrip())
    file.close()
    return True

def check(word):
    if word.lower() in words:
        return True
    else:
        return False

def size():
    return len(words)

def unload():
    return True
```

- First, we create a new set called words.
- Notice that we haven't needed a main function. Our Python program will run from top to bottom. Here, we want to define a function, so we use def load(). load will take a parameter, dictionary, and its return value is implied. We open the file with open, and iterate over the lines in the file with just for line in file:. Then, we remove the newline at the end of line, and add it to our set words. Notice that line is a string, but has a .rstrip function we can call.
- Then, for check, we can just ask if word.lower() in words. For size, we can use len to count the number of elements in our set, and finally, for unload, we don't have to do anything, since Python manages memory for us.

- It turns out, even though implementing a program in Python is simpler for us, the running time of our program in Python is slower than our program in C since the language has to do more work for us with general-purpose solutions, like for memory management.
- In addition, Python is also the name of a program called an **interpreter**, which reads in our source code and translates it to code that our CPU can understand, line by line.
- For example, if our pseudocode from week 0 was in Spanish, and we didn't understand Spanish, we would have to slowly translate it, line by line, into English before we could search for a name in a phone book:

```
1  Recoge guía telefónica
2  Abre a la mitad de guía telefónica
3  Ve la página
4  Si la persona está en la página
5     Llama a la persona
6  Si no, si la persona está antes de mitad de guía telefónica
7     Abre a la mitad de la mitad izquierda de la guía telefónica
8     Regresa a la línea 3
9  Si no, si la persona está después de mitad de guía telefónica
10    Abre a la mitad de la mitad derecha de la guía telefónica
11    Regresa a la línea 3
12 De lo contrario
13    Abandona
```

- So, depending on our goals, we'll also have to consider the tradeoff of human time of writing a program that's more efficient, versus the running time of the program.

Input, conditions

- We can get input from the user with the input function:

```
answer = input("What's your name? ")
print(f"hello, {answer}")
```

- We can ask the user for two integers and add them:

```
from cs50 import get_int

# Prompt user for x
x = get_int("x: ")
```

```
# Prompt user for y
y = get_int("y: ")

# Perform addition
print(x + y)
```

- Comments start with # instead of //.

- If we call input ourselves, we get back strings for our values:

```
# Prompt user for x
x = input("x: ")

# Prompt user for y
y = input("y: ")

# Perform addition
print(x + y)
```

- So we need to **cast**, or convert, each value from input into an int before we store it:

```
# Prompt user for x
x = int(input("x: "))

# Prompt user for y
y = int(input("y: "))

# Perform addition
print(x + y)
```

- But if the user didn't type in a number, we'll need to do even more error-checking or our program will crash. So we'll generally want to use a commonly used library to solve problems like this.

- We'll divide values:

```
# Prompt user for x
x = int(input("x: "))

# Prompt user for y
y = int(input("y: "))

# Perform division
print(x / y)
```

- Notice that we get floating-point, decimal values back, even if we divided two integers.

- And we can demonstrate conditions:

```
from cs50 import get_int

x = get_int("x: ")
y = get_int("y: ")

if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x is equal to y")
```

- We can import entire libraries, and use functions inside them as if they were a struct:

```
import cs50

x = cs50.get_int("x: ")
y = cs50.get_int("y: ")
```

- If our program needed to import two different libraries, each with a get_int function, for example, we would need to use this method to **namespace** functions, keeping their names in different spaces to prevent them fromm colliding.

- To compare strings, we can say:

```
from cs50 import get_string
```

```
s = get_string("Do you agree? ")

if s == "Y" or s == "y":
    print("Agreed.")
elif s == "N" or s == "n":
    print("Not agreed.")
```

- Python doesn't have chars, so we check `Y` and other letters as strings. We can also compare strings directly with `==`. Finally, in our Boolean expressions we use `or` and `and` instead of symbols.
- We can also say `if s.lower() in ["y", "yes"]:` to check if our string is in a list, after converting it to lowercase first.

meow

- We can improve versions of `meow`, too:

```
print("meow")
print("meow")
print("meow")
```

- We don't need to declare a `main` function, so we just write the same line of code three times.

- We can define a function that we can reuse:

```
for i in range(3):
    meow()

def meow():
    print("meow")
```

- But this causes an error when we try to run it: `NameError: name 'meow' is not defined`. It turns out that we need to define our function before we use it, so we can either move our definition of `meow` to the top, or define a main function first:

```
def main():
    for i in range(3):
        meow()

def meow():
    print("meow")

main()
```

- Now, by the time we actually call our `main` function, the `meow` function will already have been defined.

- Our functions can take inputs, too:

```
def main():
    meow(3)

def meow(n):
    for i in range(n):
        print("meow")

main()
```

- Our `meow` function takes in a parameter, `n`, and passes it to `range`.

get_positive_int

- We can define a function to get a positive integer:

```
from cs50 import get_int

def main():
    i = get_positive_int()
    print(i)

def get_positive_int():
    while True:
```

```
        n = get_int("Positive Integer: ")
        if n > 0:
            break
    return n

main()
```

- Since there is no do-while loop in Python as there is in C, we have a `while` loop that will go on infinitely, and use `break` to end the loop as soon as `n > 0`. Finally, our function will `return n`, at our original indentation level, outside of the `while` loop.
- Notice that variables in Python are **scoped to functions** by default, meaning that `n` can be initialized within a loop, but still be accessible later in the function.

Mario

- We can print out a row of question marks on the screen:

```
for i in range(4):
    print("?", end="")
print()
```

- When we print each block, we don't want the automatic new line, so we can pass a **named argument**, also known as keyword argument, to the `print` function, which specifies the value for a specific parameter. So far, we've only seen **positional arguments**, where parameters are set based on their position in the function call.
- Here, we say `end=""` to specify that nothing should be printed at the end of our string. `end` is also an **optional argument**, one we don't need to pass in, with a default value of `\n`, which is why `print` usually adds a new line for us.
- Finally, after we print our row with the loop, we can call `print` with no other arguments to get a new line.

- We can also "multiply" a string and print that directly with: `print("?" * 4)`.
- We can implement nested loops:

```
for i in range(3):
    for j in range(3):
        print("#", end="")
    print()
```

Overflow, imprecision

- In Python, trying to cause an integer overflow actually won't work:

```
i = 1
while True:
    print(i)
    i *= 2
```

- We see larger and larger numbers being printed, since Python automatically uses more and more memory to store numbers for us, unlike C where integers are fixed to a certain number of bytes.

- Floating-point imprecision, too, still exists, but can be prevented by libraries that can represent decimal numbers with as many bits as are needed.

Lists, strings

- We can make a list:

```
scores = [72, 73, 33]

print("Average: " + str(sum(scores) / len(scores)))
```

- We can use `sum`, a function built into Python, to add up the values in our list, and divide it by the number of scores, using the `len` function to get the length of the list. Then, we cast the float to a string before we can concatenate and print it.
- We can even add the entire expression into a formatted string for the same effect:
- `print(f"Average: {sum(scores) / len(scores)}")`

- We can add items to a list with:

```
from cs50 import get_int

```

```
scores = []
for i in range(3):
    scores.append(get_int("Score: "))
...
```

- We can iterate over each character in a string:

```
from cs50 import get_string

s = get_string("Before:  ")
print("After: ", end="")
for c in s:
    print(c.upper(), end="")
print()
```

  - Python will iterate over each character in the string for us with just `for c in s`.
- To make a string uppercase, we can also just call `s.upper()`, without having to iterate over each character ourselves.

Command-line arguments, exit codes

- We can take command-line arguments with:

```
from sys import argv

if len(argv) == 2:
    print(f"hello, {argv[1]}")
else:
    print("hello, world")
```

  - We import `argv` from `sys`, or system module, built into Python.
  - Since `argv` is a list, we can get the second item with `argv[1]`, so adding an argument with the command `python argv.py David` will result in `hello, David` printed.
  - Like in C, `argv[0]` would be the name of our program, like `argv.py`.
- We can also let Python iterate over the list for us:

```
from sys import argv

for arg in argv:
    print(arg)
```

- We can return exit codes when our program exits, too:

```
import sys

if len(sys.argv) != 2:
    print("missing command-line argument")
    sys.exit(1)
print(f"hello, {sys.argv[1]}")
sys.exit(0)
```

  - We import the entire `sys` module now, since we're using multiple components of it. Now we can use `sys.argv` and `sys.exit()` to exit our program with a specific code.

Algorithms

- We can implement linear search by just checking each element in a list:

```
import sys

numbers = [4, 6, 8, 2, 7, 5, 0]

if 0 in numbers:
    print("Found")
    sys.exit(0)

print("Not found")
sys.exit(1)
```

  - With `if 0 in numbers:`, we're asking Python to check the list for us.
- A list of strings, too, can be searched with:

```
names = ["Bill", "Charlie", "Fred", "George", "Ginny", "Percy", "Ron"]

if "Ron" in names:
    print("Found")
else:
    print("Not found")
```

- If we have a dictionary, a set of key-value pairs, we can also check for a particular key, and look at the value stored for it:

```
from cs50 import get_string

people = {
    "Brian": "+1-617-495-1000",
    "David": "+1-949-468-2750"
}

name = get_string("Name: ")
if name in people:
    print(f"Number: {people[name]}")
```

  - We first declare a dictionary, `people`, where the keys are strings of each name we want to store, and the value we want to associate with each key is a string of a corresponding phone number.
  - Then, we use `if name in people:` to search the keys of our dictionary for a `name`. If the key exists, then we can get the value with the bracket notation, `people[name]`, much like indexing into an array with C, except here we use a string instead of an integer.
  - Dictionaries, as well as sets, are typically implemented in Python with a data structure like a hash table, so we can have close to constant time lookup. Again, we have the tradeoff of having less control over exactly what happens under the hood, like being able to choose a hash function, with the benefit of having to do less work ourselves.

- Swapping two variables can also be done simply by assigning both values at the same time:

```
x = 1
y = 2

print(f"x is {x}, y is {y}")
x, y = y, x
print(f"x is {x}, y is {y}")
```

  - In Python, we don't have access to pointers, which protects us from making mistakes with memory.

Files

- Let's open a CSV file:

```
import csv

from cs50 import get_string

file = open("phonebook.csv", "a")

name = get_string("Name: ")
number = get_string("Number: ")

writer = csv.writer(file)
writer.writerow([name, number])

file.close()
```

  - It turns out that Python also has a `csv` library that helps us work with CSV files, so after we open the file for appending, we can call `csv.writer` to create a `writer` from the file, which gives additional functionality, like `writer.writerow` to write a list as a row.

- We can use the `with` keyword, which will close the file for us after we're finished:

```
...
with open("phonebook.csv", "a") as file:
    writer = csv.writer(file)
```

```
    writer.writerow((name, number))
```

- We can open another CSV file, tallying the number of times a value appears:

```
import csv

houses = {
    "Gryffindor": 0,
    "Hufflepuff": 0,
    "Ravenclaw": 0,
    "Slytherin": 0
}

with open("Sorting Hat (Responses) - Form Responses 1.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        house = row[1]
        houses[house] += 1

for house in houses:
    print(f"{house}: {houses[house]}")
```

  - We use the `reader` function from the `csv` library, skip the header row with `next(reader)`, and then iterate over each of the rest of the rows.
  - The second item in each row, `row[1]`, is the string of a house, so we can use that to access the value stored in `houses` for that key, and add one to it.
  - Finally, we'll print out the count for each house.

More libraries

- On our own Mac or PC, we can open a terminal after installing Python, and use another library to convert text to speech:

```
import pyttsx3

engine = pyttsx3.init()
engine.say("hello, world")
engine.runAndWait()
```

  - By reading the documentation, we can figure out how to initialize the library, and say a string.
  - We can even pass in a format string with `engine.say(f"hello, {name}")` to say some input.
- We can use another library, `face_recognition`, to find faces in images:

```
# Find faces in picture
# https://github.com/ageitgey/face_recognition/blob/master/examples/find_faces_in_picture.py

from PIL import Image
import face_recognition

# Load the jpg file into a numpy array
image = face_recognition.load_image_file("office.jpg")

# Find all the faces in the image using the default HOG-based model.
# This method is fairly accurate, but not as accurate as the CNN model and not GPU accelerated.
# See also: find_faces_in_picture_cnn.py
face_locations = face_recognition.face_locations(image)

for face_location in face_locations:

    # Print the location of each face in this image
    top, right, bottom, left = face_location

    # You can access the actual face itself like this:
    face_image = image[top:bottom, left:right]
```

```
    pil_image = Image.fromarray(face_image)
    pil_image.show()
```

- With recognize.py, we can write a program that finds a match for a particular face.
- We can create a QR code, or two-dimensional barcode, with another library:

```
import os
import qrcode

img = qrcode.make("https://youtu.be/oHg5SJYRHA0")
img.save("qr.png", "PNG")
os.system("open qr.png")
```

- We can recognize audio input from a microphone:

```
import speech_recognition

# Obtain audio from the microphone
recognizer = speech_recognition.Recognizer()
with speech_recognition.Microphone() as source:
    print("Say something:")
    audio = recognizer.listen(source)

# Recognize speech using Google Speech Recognition
print("You said:")
print(recognizer.recognize_google(audio))
```

  - We're following the documentation of the library to listen to our microphone and convert it to text.

- We can even add additional logic for basic responses:

```
...
words = recognizer.recognize_google(audio)

# Respond to speech
if "hello" in words:
    print("Hello to you too!")
elif "how are you" in words:
    print("I am well, thanks!")
elif "goodbye" in words:
    print("Goodbye to you too!")
else:
    print("Huh?")
```

- Finally, we use another, more sophisticated program to generate deepfakes, or realistic-appearing but computer-generated videos of various personalities.
- By taking advantage of all these libraries that are freely available online, we can easily add advanced functionality to our own applications.

Lab 6: World Cup

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Write a program to run simulations of the FIFA World Cup.

```
$ python tournament.py 2018m.csv
Belgium: 20.9% chance of winning
Brazil: 20.3% chance of winning
Portugal: 14.5% chance of winning
Spain: 13.6% chance of winning
Switzerland: 10.5% chance of winning
Argentina: 6.5% chance of winning
England: 3.7% chance of winning
France: 3.3% chance of winning
Denmark: 2.2% chance of winning
Croatia: 2.0% chance of winning
Colombia: 1.8% chance of winning
Sweden: 0.5% chance of winning
Uruguay: 0.1% chance of winning
Mexico: 0.1% chance of winning
```

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Background

In soccer's World Cup, the knockout round consists of 16 teams. In each round, each team plays another team and the losing teams are eliminated. When only two teams remain, the winner of the final match is the champion.

In soccer, teams are given FIFA Ratings, which are numerical values representing each team's relative skill level. Higher FIFA ratings indicate better previous game results, and given two teams' FIFA ratings, it's possible to estimate the probability that either team wins a game based on their current ratings. The FIFA Ratings from just before the two previous World Cups are available as the May 2018 Men's FIFA Ratings and March 2019 Women's FIFA Ratings.

Using this information, we can simulate the entire tournament by repeatedly simulating rounds until we're left with just one team. And if we want to estimate how likely it is that any given team wins the tournament, we might simulate the tournament many times (e.g. 1000 simulations) and count how many times each team wins a simulated tournament.

Your task in this lab is to do just that using Python!

Getting Started

1. Log into ide.cs50.io using your GitHub account.
2. In your terminal window, run `wget https://cdn.cs50.net/2020/fall/labs/6/lab6.zip` to download a Zip file of the lab distribution code.
3. In your terminal window, run `unzip lab6.zip` to unzip (i.e., decompress) that Zip file.
4. In your terminal window, run `cd lab6` to change directories into your `lab6` directory.

Understanding

Start by taking a look at the `2018m.csv` file. This file contains the 16 teams in the knockout round of the 2018 Men's World Cup and the ratings for each team. Notice that the CSV file has two columns, one called `team` (representing the team's country name) and one called `rating` (representing the team's rating).

The order in which the teams are listed determines which teams will play each other in each round (in the first round, for example, Uruguay will play Portugal and France will play Argentina; in the next round, the winner of the Uruguay-

Portugal match will play the winner of the France-Argentina match). So be sure not to edit the order in which teams appear in this file!

Ultimately, in Python, we can represent each team as a dictionary that contains two values: the team name and the rating. Uruguay, for example, we would want to represent in Python as `{"team": "Uruguay", "rating": 976}`.

Next, take a look at `2019w.csv`, which contains data formatted the same way for the 2019 Women's World Cup.

Now, open `tournament.py` and see that we've already written some code for you. The variable `N` at the top represents how many World Cup simulations to run: in this case, 1000.

The `simulate_game` function accepts two teams as inputs (recall that each team is a dictionary containing the team name and the team's rating), and simulates a game between them. If the first team wins, the function returns `True`; otherwise, the function returns `False`.

The `simulate_round` function accepts a list of teams (in a variable called `teams`) as input, and simulates games between each pair of teams. The function then returns a list of all of the teams that won the round.

In the `main` function, notice that we first ensure that `len(sys.argv)` (the number of command-line arguments) is 2. We'll use command-line arguments to tell Python which team CSV file to use to run the tournament simulation. We've then defined a list called `teams` (which will eventually be a list of teams) and a dictionary called `counts` (which will associate team names with the number of times that team won a simulated tournament). Right now they're both empty, so populating them is left up to you!

Finally, at the end of `main`, we sort the teams in descending order of how many times they won simulations (according to `counts`) and print the estimated probability that each team wins the World Cup.

Populating `teams` and `counts` and writing the `simulate_tournament` function are left up to you!

Implementation Details

Complete the implementation of `tournament.py`, such that it simulates a number of tournaments and outputs each team's probability of winning.

First, in `main`, read the team data from the CSV file into your program's memory, and add each team to the list `teams`.

- The file to use will be provided as a command-line argument. You can access the name of the file, then, with `sys.argv[1]`.
- Recall that you can open a file with `open(filename)`, where `filename` is a variable storing the name of the file.
- Once you have a file `f`, you can use `csv.DictReader(f)` to give you a "reader": an object in Python that you can loop over to read the file one row at a time, treating each row as a dictionary.
- By default, all values read from the file will be strings. So be sure to first convert the team's `rating` to an `int` (you can use the `int` function in Python to do this).
- Ultimately, append each team's dictionary to `teams`. The function call `teams.append(x)` will append `x` to the list `teams`.
- Recall that each team should be a dictionary with a `team` name and a `rating`.

Next, implement the `simulate_tournament` function. This function should accept as input a list of teams and should repeatedly simulate rounds until you're left with one team. The function should the return the name of that team.

- You can call the `simulate_round` function, which simulates a single round, accepting a list of teams as input and returning a list of all of the winners.
- Recall that if `x` is a list, you can use `len(x)` to determine the length of the list.
- You should not assume the number of teams in the tournament, but you may assume it will be a power of 2.

Finally, back in the `main` function, run `N` tournament simulations, and keep track of how many times each team wins in the `counts` dictionary.

- For example, if Uruguay won 2 tournaments and Portugal won 3 tournaments, then your `counts` dictionary should be `{"Uruguay": 2, "Portugal": 3}`.
- You should use your `simulate_tournament` to simulate each tournament and determine the winner.

- Recall that if `counts` is a dictionary, then syntax like `counts[team_name] = x` will associate the key stored in `team_name` with the value stored in `x`.
- You can use the `in` keyword in Python to check if a dictionary has a particular key already. For example, `if "Portugal" in counts:` will check to see if `"Portugal"` already has an existing value in the `counts` dictionary.

Walkthrough

Hints

- When reading in the file, you may find this syntax helpful, with `filename` as the name of your file and `file` as a variable.

```
with open(filename) as file:
    reader = csv.DictReader(file)
```

- In Python, to append to the end of a list, use the `.append()` function.

Testing

Your program should behave per the examples below. Since simulations have randomness within each, your output will likely not perfectly match the examples below.

```
$ python tournament.py 2018m.csv
Belgium: 20.9% chance of winning
Brazil: 20.3% chance of winning
Portugal: 14.5% chance of winning
Spain: 13.6% chance of winning
Switzerland: 10.5% chance of winning
Argentina: 6.5% chance of winning
England: 3.7% chance of winning
France: 3.3% chance of winning
Denmark: 2.2% chance of winning
Croatia: 2.0% chance of winning
Colombia: 1.8% chance of winning
Sweden: 0.5% chance of winning
Uruguay: 0.1% chance of winning
Mexico: 0.1% chance of winning
$ python tournament.py 2019w.csv
Germany: 17.1% chance of winning
United States: 14.8% chance of winning
England: 14.0% chance of winning
France: 9.2% chance of winning
Canada: 8.5% chance of winning
Japan: 7.1% chance of winning
Australia: 6.8% chance of winning
Netherlands: 5.4% chance of winning
Sweden: 3.9% chance of winning
Italy: 3.0% chance of winning
Norway: 2.9% chance of winning
Brazil: 2.9% chance of winning
Spain: 2.2% chance of winning
China PR: 2.1% chance of winning
Nigeria: 0.1% chance of winning
```

- You might be wondering what actually happened at the 2018 and 2019 World Cups! For Men's, France won, defeating Croatia in the final. Belgium defeated England for the third place position. For Women's, the United States won, defeating the Netherlands in the final. England defeated Sweden for the third place position.

Not sure how to solve?

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/labs/2021/x/worldcup
```

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

```
style50 tournament.py
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/worldcup
```

Problem Set 6

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 6** before beginning this problem set.

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 Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Try out any of David's programs from Week 6.

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*.
- 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.

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 Lab 1, except that your program this time should be written in Python.

Usage

Your program should behave per the example below.

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

Testing

While `check50` is available for this problem, you're encouraged to first test your code on your own for each of the following.

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

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/2021/x/sentimental/hello
```

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

```
style50 hello.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/hello
```

Mario-less

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

```
$ python mario.py
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 in Python.
- 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.

```
$ python mario.py
Height: 4
   #
  ##
 ###
####
```

Testing

While `check50` is available for this problem, you're encouraged to first test your code on your own 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.

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/2021/x/sentimental/mario/less
```

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

```
style50 mario.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/mario/less
```

Mario-more

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

```
$ python mario.py
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 in Python.
- 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.

```
$ python mario.py
Height: 4
   # #
  ## ##
 ### ###
#### ####
```

Testing

While `check50` is available for this problem, you're encouraged to first test your code on your own 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.

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/2021/x/sentimental/mario/more
```

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

```
style50 mario.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/mario/more
```

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 in Python.
- 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

While `check50` is available for this problem, you're encouraged to first test your code on your own 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.

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/2021/x/sentimental/cash
```

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

```
style50 cash.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/cash
```

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 in Python.
- 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
```

Hints

- It's possible to use regular expressions to validate user input. You might use Python's `re` module, for example, to check whether the user's input is indeed a sequence of digits of the correct length.

Testing

While `check50` is available for this problem, you're encouraged to first test your code on your own 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`.

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/2021/x/sentimental/credit
```

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

```
style50 credit.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/credit
```

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"`.

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

While `check50` is available for this problem, you're encouraged to first test your code on your own 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+`.

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/2021/x/sentimental/readability
```

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

```
style50 readability.py
```

This problem will be graded only along the axes of correctness and style.

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/2021/x/sentimental/readability
```

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 and then, in a terminal window, execute each of the below.

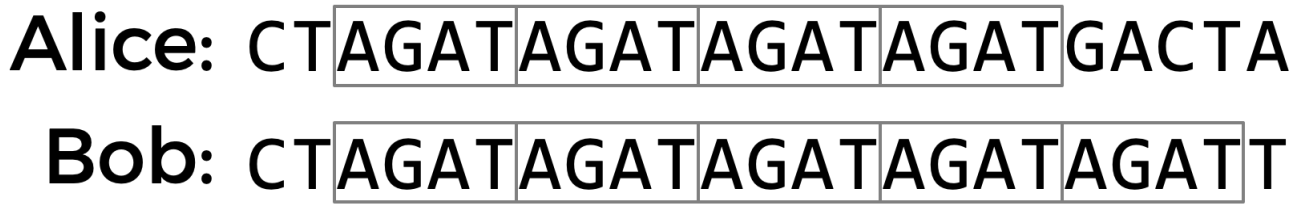
- Navigate to your `pset6` directory that should already exist.
- Execute `wget https://cdn.cs50.net/2020/fall/psets/6/dna/dna.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip dna.zip` to uncompress that file.
- Execute `rm dna.zip` followed by `yes` or `y` 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, 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 AGATs 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 module helpful for reading CSV files into memory. You may want to take advantage of either csv.reader or csv.DictReader.
- The open and read 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 or a dict may prove useful.
- Python strings allow "slicing" (accessing a particular substring within a string). If s is a string, then s[i:j] will return a new string with just the characters of s starting from character i up through (but not including) character j.

- It may be helpful to start by writing a function that, given both a DNA sequence and an STR as inputs, returns the maximum number of times that the STR repeats. You can then use that function in other parts of your program!

Testing

While `check50` is available for this problem, you're encouraged to first test your code on your own 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`.

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/2021/x/dna
```

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

```
style50 dna.py
```

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.

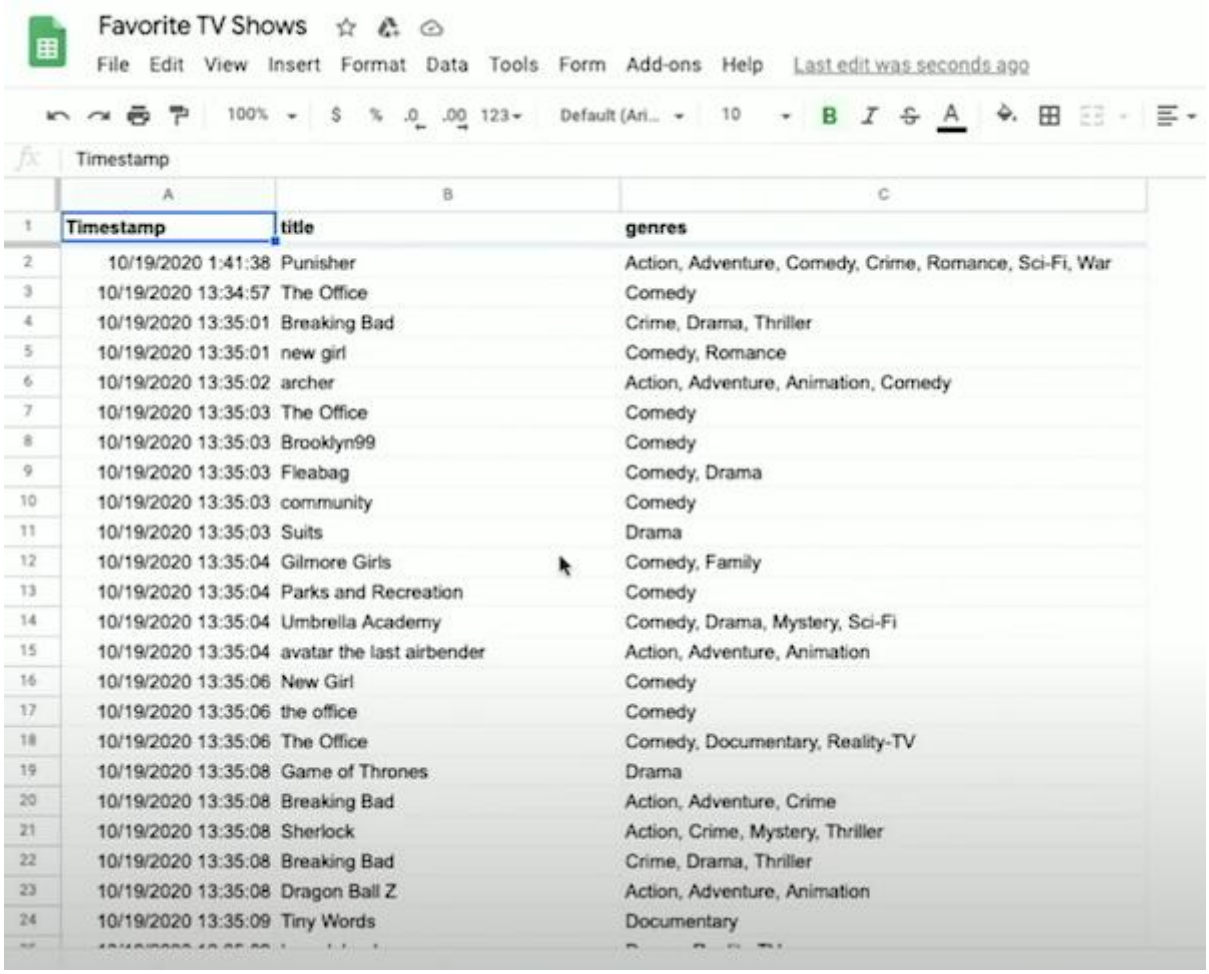
```
submit50 cs50/problems/2021/x/dna
```

Week 7 SQL

Lecture 7

Data processing

- Last week, we collected a survey of Hogwarts house preferences, and tallied the data from a CSV file with Python.
- This week, we'll collect some more data about your favorite TV shows and their genres.
- With hundreds of responses, we can start looking at the responses on Google Sheets, a web-based spreadsheet application, showing our data in rows and columns:



Favorite TV Shows

File Edit View Insert Format Data Tools Form Add-ons Help Last edit was seconds ago

Timestamp

	A	B	C
1	Timestamp	title	genres
2	10/19/2020 1:41:38	Punisher	Action, Adventure, Comedy, Crime, Romance, Sci-Fi, War
3	10/19/2020 13:34:57	The Office	Comedy
4	10/19/2020 13:35:01	Breaking Bad	Crime, Drama, Thriller
5	10/19/2020 13:35:01	new girl	Comedy, Romance
6	10/19/2020 13:35:02	archer	Action, Adventure, Animation, Comedy
7	10/19/2020 13:35:03	The Office	Comedy
8	10/19/2020 13:35:03	Brooklyn99	Comedy
9	10/19/2020 13:35:03	Fleabag	Comedy, Drama
10	10/19/2020 13:35:03	community	Comedy
11	10/19/2020 13:35:03	Suits	Drama
12	10/19/2020 13:35:04	Gilmore Girls	Comedy, Family
13	10/19/2020 13:35:04	Parks and Recreation	Comedy
14	10/19/2020 13:35:04	Umbrella Academy	Comedy, Drama, Mystery, Sci-Fi
15	10/19/2020 13:35:04	avatar the last airbender	Action, Adventure, Animation
16	10/19/2020 13:35:06	New Girl	Comedy
17	10/19/2020 13:35:06	the office	Comedy
18	10/19/2020 13:35:06	The Office	Comedy, Documentary, Reality-TV
19	10/19/2020 13:35:08	Game of Thrones	Drama
20	10/19/2020 13:35:08	Breaking Bad	Action, Adventure, Crime
21	10/19/2020 13:35:08	Sherlock	Action, Crime, Mystery, Thriller
22	10/19/2020 13:35:08	Breaking Bad	Crime, Drama, Thriller
23	10/19/2020 13:35:08	Dragon Ball Z	Action, Adventure, Animation
24	10/19/2020 13:35:09	Tiny Words	Documentary

  - Some responses show a single genre selected, like "Comedy", while others, with multiple genres, show them in one cell still but separated by a comma, like "Crime, Drama".
- With a spreadsheet app like Google Sheets, Apple's Numbers, Microsoft Excel, or others, we can:
  - sort our data
  - store data in rows and columns, where each additional entry is a row, and properties of each entry, like title or genre, is a column
  - decide on the **schema**, or format, of our data in advance by choosing the columns
- A **database** is a file or program that stores data for us.
- A CSV file is a **flat-file database** where the data for each column is separated by commas, and each row is on a new line, saved simply as a file.

    - If some data in a CSV contains a comma itself, then it's usually surrounded by quotes as a string to prevent confusion.
    - Formulas and calculations in spreadsheet programs are built into the programs themselves; a CSV file can only store raw, static values.
- We'll download a CSV file with the data from the spreadsheet with "File > Download", upload it to our IDE by dragging and dropping it into our file tree, and see that it's indeed a text file with comma-separated values matching the spreadsheet's data.

Cleaning

- We'll start by writing `favorites.py`, choosing Python over C as our tool of choice for its libraries and abstraction:

```
import csv

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        print(row[1])
```

    - We're going to open the file and make sure we can print the title of each row, using the `with` keyword in Python that will close our file for us after we leave its scope, based on indentation.
    - `open` uses read mode by default, but to be clear in our code we'll add `r` explicitly.
    - The `csv` library has a `reader` function that will create a `reader` variable we can use.
    - We'll call `next` to skip the first row, since that's the header row, and then use a loop to print the second column in each row, which is the title.
- To improve this, we'll use a `DictReader`, dictionary reader, which creates a dictionary from each row, allowing us to access each column by its name. We also don't need to skip the header row in this case, since the `DictReader` will use it automatically.

```
import csv

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        print(row["title"])
```

    - Since the first row in our CSV has the names of the columns, it can be used to label each column in our data as well.
- Now let's take a look at all the unique titles in our responses:

```
import csv

titles = set()

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        titles.add(row["title"])

for title in titles:
    print(title)
```

    - We'll create a set called `titles`, and add each row's title value to it. Calling `add` on a set will automatically check for duplicates and ensure that there are only unique values.
    - Then, we can iterate over the elements in the set with a `for` loop, printing each of them.
- To sort the titles, we can just change our loop to `for title in sorted(titles)`, which will sort our set before we iterate over it.
- We'll see that our titles are considered different if their capitalization or punctuation is different, so we'll clean up the capitalization by adding title in all uppercase with `titles.add(row["title"].upper())`.
- We'll also have to remove spaces before or after, so we can add `titles.add(row["title"].strip().upper())` which strips the whitespace from the title, and then converts it to uppercase.
- Now, we've **canonicalized**, or standardized, our data, and our list of titles are much cleaner.

Counting

- We can use a dictionary, instead of a set, to count the number of times we've seen each title, with the keys being the titles and the values being an integer counting the number of times we see each of them:

```
import csv

titles = {}

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)

    for row in reader:
        title = row["title"].strip().upper()
        if title not in titles:
            titles[title] = 0
        titles[title] += 1

for title in sorted(titles):
    print(title, titles[title])
```

  - Here, we first check if we've haven't seen the title before (if it's `not in titles`). We set the initial value to 0 if that's the case, and then we can safely increment the value by 1 every time.
  - Finally, we can print out our dictionary's keys and values by passing them as arguments to `print`, which will separate them by a space for us.
- We can sort by the values in the dictionary by changing our loop to:

```
...
def f(title):
    return titles[title]

for title in sorted(titles, key=f, reverse=True):
...
```

  - We define a function, `f`, which just returns the count of a title in the dictionary with `titles[title]`. The `sorted` function, in turn, will use that function as the **key** to sort the dictionary's elements. And we'll also pass in `reverse=True` to sort from largest to smallest, instead of smallest to largest.
  - So now we'll see the most popular shows printed.
- We can actually define our function in the same line, with this syntax:

```
for title in sorted(titles, key=lambda title: titles[title], reverse=True):
```

  - We pass in a **lambda**, or anonymous function, which has no name but takes in some argument or arguments, and returns a value immediately.

Searching

- We can write a program to search for a title and report its popularity:

```
import csv

title = input("Title: ").strip().upper()

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)

    counter = 0
    for row in reader:
        if row["title"].strip().upper() == title:
            counter += 1

print(counter)
```

  - We ask the user for input, and then open our CSV file. Since we're looking for just one title, we can have one `counter` variable that we increment.
  - We check for matches after standardizing both the input and the data as we check each row.
- The running time of this is $O(n)$, since we need to look at every row.

Relational databases

- **Relational databases** are programs that store data, ultimately in files, but with additional data structures that allow us to search and store data more efficiently.
- With another programming language, **SQL** (pronounced like "sequel"), Structured Query Language, we can interact with many relational databases and their **tables**, like spreadsheets, which store data.
- We'll use a common database program called **SQLite**, one of many available programs that support SQL. Other database programs include Oracle Database, MySQL, PostgreSQL, and Microsoft Access.
- SQLite stores our data in a binary file, with 0s and 1s that represent data efficiently. We'll interact with our tables of data through a command-line program, `sqlite3`.
- We'll run some commands in the CS50 IDE to import our CSV file into a table called "shows":

```
~/ $ sqlite3
SQLite version 3.22.0 2018-01-22 18:45:57
Enter ".help" for usage hints.
sqlite> .mode csv
sqlite> .import 'Favorite TV Shows (Responses) - Form Responses 1.csv' shows
```

  - Based on the rows in the CSV file, SQLite will create a table in our database with the data and columns.
  - We'll set SQLite to CSV mode, and use the `.import` command to create a table from our file.
- It turns out that, when working with data, we generally need four types of operations supported by relational databases:
  - CREATE
  - READ
  - UPDATE
  - DELETE

SQL

- In SQL, the commands to perform each of these operations are:
  - CREATE, INSERT
    - For example, to create a new table, we can use: `CREATE TABLE table (column type, ...);` where `table` is the name of our new table, and `column` is the name of a column, followed by its type.
  - SELECT
    - `SELECT column FROM table;`
  - UPDATE
  - DELETE
- We can check the schema of our new table with `.schema`:

```
sqlite> .schema
CREATE TABLE shows(
  "Timestamp" TEXT,
  "title" TEXT,
  "genres" TEXT
);
```

  - We see that `.import` used the `CREATE TABLE ...` command listed to create a table called `shows`, with column names automatically copied from the CSV's header row and types assumed to be text.
- We can select a column with:

```
sqlite> SELECT title FROM shows;
title
...
"Madam Secretary"
"Game of Thrones"
"Still Game"
```

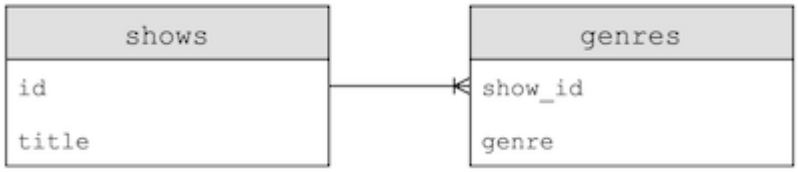
  - Notice that we capitalize SQL keywords by convention, and we'll see titles from our rows printed in the order from the CSV.
  - We can also select multiple columns with `SELECT Timestamp, title FROM shows;` (`Timestamp` was capitalized in the CSV), or all columns with `SELECT * FROM shows;`.
- SQL supports many functions that we can use to count and summarize data:
  - AVG
  - COUNT
  - DISTINCT, for getting distinct values without duplicates
  - LOWER
  - MAX
  - MIN

    - UPPER
    - …
- We can clean up our titles as before, converting them to uppercase and printing only the unique values:
- sqlite> SELECT DISTINCT(UPPER(title)) FROM shows;
- title
- ...
- "GREY'S ANATOMY"
- "SCOOBY DOO"
- "MADAM SECRETARY"
- We can also add more **clauses**, or phrases that modify our query:
    - WHERE, matching results on a strict condition
    - LIKE, matching results on a less strict condition
    - ORDER BY, ordering results in some way
    - LIMIT, limiting the number of results
    - GROUP BY, grouping results in some way
    - …
- Let's filter rows by titles:
- sqlite> SELECT title FROM shows WHERE title = "The Office";
- title
- ...
- "The Office"
- "The Office"
- "The Office"
- But there are other entries we would like to catch, so we can use:
- sqlite> SELECT title FROM shows WHERE title LIKE "%Office%";
- title
- ...
- office
- "The Office"
- "the office "
- "The Office"
    - The % character is a placeholder for zero or more other characters.
- We can order our titles:
- sqlite> SELECT DISTINCT(UPPER(title)) FROM shows ORDER BY UPPER(title);
- ...
- X-FILES
- "ZETA GUNDAM"
- "ZONDAG MET LUBACH"
- We can even group the same titles together, and count the number of times they appear:
- sqlite> SELECT UPPER(title), COUNT(title) FROM shows GROUP BY UPPER(title);
- ...
- "THE OFFICE",23
- ...
- "TOP GEAR",1
- ...
- "TWIN PEAKS",4
- ...
- We can order by the counts:
- sqlite> SELECT UPPER(title), COUNT(title) FROM shows GROUP BY UPPER(title) ORDER BY COUNT(title);
- ...
- "THE OFFICE",23
- FRIENDS,26
- "GAME OF THRONES",33
    - And if we add DESC to the end, we could see the results in descending order.
- With LIMIT 10 added as well, we see the top 10 rows:
- sqlite> SELECT UPPER(title), COUNT(title) FROM shows GROUP BY UPPER(title) ORDER BY COUNT(title) DESC LIMIT 10;
- UPPER(title),COUNT(title)

- "GAME OF THRONES",33
- FRIENDS,26
- "THE OFFICE",23
- ...
- Finally, we'll trim whitespace from each title too, nesting that function:
- sqlite> SELECT UPPER(TRIM(title)), COUNT(title) FROM shows GROUP BY UPPER(TRIM(title)) ORDER BY COUNT(title) DESC LIMIT 10;
- UPPER(title),COUNT(title)
- "GAME OF THRONES",33
- FRIENDS,26
- "THE OFFICE",23
- ...
- Before we finish, we'll want to save our data into a file with `.save shows.db`, which we'll see in our IDE after running that command.
- Notice that our program to find the most popular shows from earlier, which took dozens of lines of code in Python, now only requires one (long) line of SQL.
- We've used SQLite's command-line interface, but there are also graphical programs that support working with SQL queries and viewing results more visually.

Tables

- Our `genres` column has multiple genres in the same field, so we'll use `LIKE` to get all the titles containing some genre:
- sqlite> SELECT title FROM shows WHERE genres LIKE "%Comedy%";
- ...
    - But the genres are still stored as a comma-separated list themselves, which is not as clean. For example, if our genres included both "Music" and "Musical", it would be difficult to select titles with just the "Music" genre.
- We can insert data into a table manually with `INSERT INTO table (column, ...) VALUES(value, ...);`.
    - For example, we can say:
    - sqlite> INSERT INTO shows (Timestamp, title, genres) VALUES("now", "The Muppet Show", "Comedy, Musical");
- We can update a row with `UPDATE table SET column = value WHERE condition;`, like:
- sqlite> UPDATE shows SET genres = "Comedy, Drama, Musical" WHERE title = "The Muppet Show";
- We can even remove all rows that match with: `DELETE FROM table WHERE condition;` :
- sqlite> DELETE FROM shows WHERE title LIKE "Friends";
- Now let's write our own Python program that will use SQL to import our CSV data into tables with the following design:



    - This design will start to **normalize** our data, or reduce redundancy and ensure a single source of truth.
    - Here, for example, we have a table named `shows`, with each show having an `id` and `title`, and another table, `genre`, which uses each's show's `id` to associate a genre with a show. Note that the show's `title` doesn't need to be stored multiple times.
    - We can also now add multiple rows in the `genre` table, to associate a show with more than one genre.
- It turns out that SQL, too, has its own data types to optimize the amount of space used for storing data, which we'll need to specify when creating a table manually:
    - BLOB, for "binary large object", raw binary data that might represent files
    - INTEGER
    - NUMERIC, number-like but not quite a number, like a date or time
    - REAL, for floating-point values
    - TEXT, like strings
- Columns can also have additional attributes:
    - NOT NULL, which specifies that there must be some value
    - UNIQUE, which means that the value for that column must be unique for every row in the table
    - PRIMARY KEY, like the `id` column above that will be used to uniquely identify each row
    - FOREIGN KEY, like the `show_id` column above that refers to a column in some other table

- We'll use the CS50 library's SQL feature to make queries easily, and there are other libraries for Python as well:

```
import csv

from cs50 import SQL

open("shows.db", "w").close()
db = SQL("sqlite:///shows.db")

db.execute("CREATE TABLE shows (id INTEGER, title TEXT, PRIMARY KEY(id))")
db.execute("CREATE TABLE genres (show_id INTEGER, genre TEXT, FOREIGN KEY(show_id) REFERENCES shows(id))")

with open("Favorite TV Shows - Form Responses 1.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        title = row["title"].strip().upper()

        id = db.execute("INSERT INTO shows (title) VALUES(?)", title)

        for genre in row["genres"].split(", "):
            db.execute("INSERT INTO genres (show_id, genre) VALUES(?, ?)", id, genre)
```

  - First, we'll open a `shows.db` file and close it, to make sure that the file is created.
  - Then, we'll create a `db` variable to store our database created by `SQL`, which takes the database file we just created.
  - Next, we'll run SQL commands by writing it as a string, and calling `db.execute` with it. Here, we'll create two tables as we designed above, indicating the names, types, and properties of each column we want in each table.
  - Now, we can read our original CSV file row by row, getting the title and using `db.execute` to run an `INSERT` command for each row. It turns out that we can use the `?` placeholder in a SQL command, and pass in a variable to be substituted. Then, we'll get back an `id` that's automatically created for us for each row, since we declared it to be a primary key.
  - Finally, we'll split the genre string in each row by the comma, and insert each of them into the `genres` table, using the `id` for the show as `show_id`.
- After we run this program, we can see the IDs and titles of each show, as well as the genres where the first column is a show's ID:

```
sqlite> SELECT * FROM shows;
...
511 | MADAM SECRETARY
512 | GAME OF THRONES
513 | STILL GAME
sqlite> SELECT * FROM genres;
...
511 | Drama
512 | Action
512 | Adventure
512 | History
512 | Thriller
512 | War
513 | Comedy
```

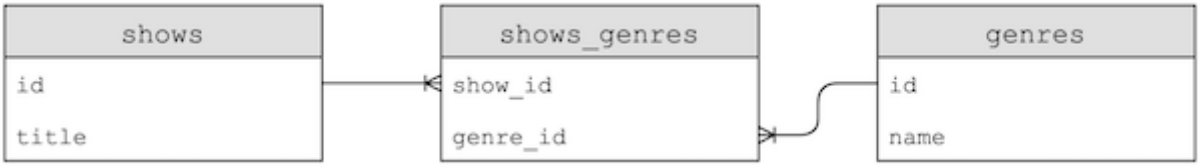
  - Notice that the show with id 512, "GAME OF THRONES", now has five genres associated with it.
- To find all the musicals, for example, we can run:

```
sqlite> SELECT show_id FROM genres WHERE genre = "Musical";
...
422
435
468
```

- And we can nest that query to get titles from the list of show IDs:

```
sqlite> SELECT title FROM shows WHERE id IN (SELECT show_id FROM genres WHERE genre = "Musical");
title
BREAKING BAD
```

- ...
- THE LAWYER
- MY BROTHER, MY BROTHER, AND ME
    - Our first query, in the parentheses, will be executed first, and then used in the outer query.
- We can find all the rows in shows with the title of “THE OFFICE”, and find all the genres associated:
- sqlite> SELECT DISTINCT(genre) FROM genres WHERE show_id IN (SELECT id FROM shows WHERE title = "THE OFFICE") ORDER BY genre;
- genre
- ...
- Comedy
- Documentary
- ...
- We could improve on the design of our tables even more, with a third table:

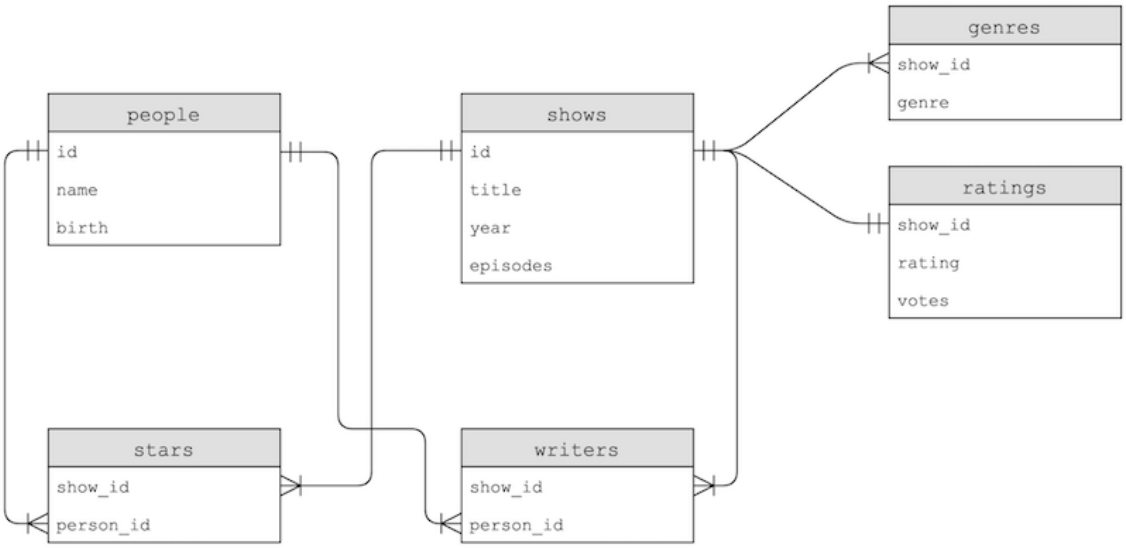


    - Now, each genre’s name will only be stored once, with a new table, a **join table**, called shows_genres that contains foreign keys that link shows to genres. This is a **many-to-many** relationship, where one show can have many genres, and one genre can belong to many shows.
    - If we needed to change a genre’s name, we would only have to change one row now, instead of many.
- It turns out that there are subtypes for a column, too, that we can be even more specific with:
    - INTEGER
        - smallint, with fewer bits
        - integer
        - bigint, with more bits
    - NUMERIC
        - boolean
        - date
        - datetime
        - numeric(scale,precision), with a fixed number of digits
        - time
        - timestamp
    - REAL
        - real
        - double precision, with twice as many bits
    - TEXT
        - char(n), a fixed number of characters
        - varchar(n), a variable number of characters, up to some limit n
        - text, a string with no limit

IMDb

- IMDb, or the Internet Movie Database, has datasets available for download as TSV (tab-separated values) files.

- After we import one such dataset, we'll see tables with the following schema:



    - The `genres` table has some duplication, since the `genre` column is repeated, but the `stars` and `writers` table join rows in the `people` and `shows` table based on their relationship.
- With `SELECT COUNT(*) FROM shows;` we can see that there are more than 150,000 shows in our table, so with a large amount of data we can use **indexes**, which tells our database program to create additional data structures so we can search and sort with logarithmic time:
- `sqlite> CREATE INDEX title_index ON shows (title);`
    - It turns out that these data structures are generally **B-trees**, like binary trees we've seen in C, with nodes organized such that we can search faster than linearly.
    - Creating an index takes some time, but afterwards we can run queries much more quickly.
- With our data spread among different tables, we can use **JOIN** commands to combine them in our queries:
- `sqlite3> SELECT title FROM people`
- `...> JOIN stars ON people.id = stars.person_id`
- `...> JOIN shows ON stars.show_id = shows.id`
- `...> WHERE name = "Steve Carell";`
- ...
- The Morning Show
- LA Times: the Envelope
    - With the `JOIN` syntax, we can virtually combine tables based on their foreign keys, and use their columns as though they were one table.
- After creating some more indexes, our `JOIN` command also runs much more quickly.

Problems

- One problem in SQL is called a **SQL injection attack**, where an someone can inject, or place, their own commands into inputs that we then run on our database.
- Our query for logging a user in might be `rows = db.execute("SELECT * FROM users WHERE username = ? AND password = ?", username, password)`. By using the `?` placeholders, our SQL library will escape the input, or prevent dangerous characters from being interpreted as part of the command.
- In contrast, we might have a SQL query that's a formatted string, such as:
- `f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"`
- If a user types in `malan@harvard.edu'--`, then the query will end up being:
- `f"SELECT * FROM users WHERE username = 'malan@harvard.edu'--' AND password = '{password}'"`
- This query will actually select the row where `username = 'malan@harvard.edu'`, without checking the password, since the `--` turns the rest of the line into a comment in SQL.
- Another problem with databases is **race conditions**, where code in a multi-threaded environment can be comingled, or mixed together, in each thread.
- One example is a popular post getting lots of likes. A server might try to increment the number of likes, asking the database for the current number of likes, adding one, and updating the value in the database:
- `rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);`
- `likes = rows[0]["likes"]`
- `db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);`

    - But for applications with multiple servers, each of them might by trying to add likes at the same time. Two servers, responding to two different users, might get the same starting number of likes since the first line of code runs at the same time on each server. Then, both will set the same new number of likes, even though there should have been two separate increments.
- To solve this problem, SQL supports **transactions**, where we can lock rows in a database, such that a particular set of actions are guaranteed to happen together, with syntax like:
    - BEGIN TRANSACTION
    - COMMIT
    - ROLLBACK
- For example, we can fix our problem above with:

```
db.execute("BEGIN TRANSACTION")
rows = db.execute("SELECT likes FROM posts WHERE id = ?", id);
likes = rows[0]["likes"]
db.execute("UPDATE posts SET likes = ? WHERE id = ?", likes + 1, id);
db.execute("COMMIT")
```

    - The database will ensure that all the queries in between are executed together.
- Another example might be of two roommates and a shared fridge in their dorm. The first roommate comes home, and sees that there is no milk in the fridge. So the first roommate leaves to the store to buy milk, and while they are at the store, the second roommate comes home, sees that there is no milk, and leaves for another store to get milk as well. Later, there will be two jugs of milk in the fridge.
- We can solve this problem by locking the fridge so that our roommate can't check whether there is milk until we've gotten back.

Lab 7: Songs

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Write SQL queries to answer questions about a database of songs.

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Getting Started

Here's how to download this lab into your own CS50 IDE. Log into CS50 IDE 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 `~`).
- Execute `wget https://cdn.cs50.net/2020/fall/labs/7/lab7.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip lab7.zip` to uncompress that file.
- Execute `rm lab7.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `lab7`, which was inside of that ZIP file.
- Execute `cd lab7` to change into that directory.
- Execute `ls`. You should see a `songs.db` file, and some empty `.sql` files as well.

Understanding

Provided to you is a file called `songs.db`, a SQLite database that stores data from Spotify about songs and their artists. This dataset contains the top 100 streamed songs on Spotify in 2018. In a terminal window, run `sqlite3 songs.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 every `artist` has an `id` and a `name`. Notice, too, that every song has a `name`, an `artist_id` (corresponding to the `id` of the artist of the song), as well as values for the danceability, energy, key, loudness, speechiness (presence of spoken words in a track), valence, tempo, and duration of the song (measured in milliseconds).

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.

Implementation Details

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 songs or artists: your queries should be accurate even if the `id` of any particular song 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 songs, for example, then your query should not also output each song's tempo.

1. In `1.sql`, write a SQL query to list the names of all songs in the database.
    - Your query should output a table with a single column for the name of each song.
2. In `2.sql`, write a SQL query to list the names of all songs in increasing order of tempo.
    - Your query should output a table with a single column for the name of each song.
3. In `3.sql`, write a SQL query to list the names of the top 5 longest songs, in descending order of length.
    - Your query should output a table with a single column for the name of each song.
4. In `4.sql`, write a SQL query that lists the names of any songs that have danceability, energy, and valence greater than 0.75.
    - Your query should output a table with a single column for the name of each song.
5. In `5.sql`, write a SQL query that returns the average energy of all the songs.

    - Your query should output a table with a single column and a single row containing the average energy.
6. In `6.sql`, write a SQL query that lists the names of songs that are by Post Malone.
    - Your query should output a table with a single column for the name of each song.
    - You should not make any assumptions about what Post Malone's `artist_id` is.
7. In `7.sql`, write a SQL query that returns the average energy of songs that are by Drake.
    - Your query should output a table with a single column and a single row containing the average energy.
    - You should not make any assumptions about what Drake's `artist_id` is.
8. In `8.sql`, write a SQL query that lists the names of the songs that feature other artists.
    - Songs that feature other artists will include "feat." in the name of the song.
    - Your query should output a table with a single column for the name of each song.

Walkthrough

Hints

- See this SQL keywords reference for some SQL syntax that may be helpful!

Not sure how to solve?

Testing

Execute the below to evaluate the correctness of your code using `check50`.

```
check50 cs50/labs/2021/x/songs
```

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/songs
```

Acknowledgements

Dataset from Kaggle.

Problem Set 7

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 7** before beginning this problem set.

1. Submit Movies
2. Submit Fiftyville

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Advice

- Head to w3schools.com/sql for a handy reference!

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 and then, in a terminal window, execute each of the below.

- Execute `cd ~` (or simply `cd` with no arguments) to ensure that you're in your home directory.
- Execute `mkdir pset7` to make (i.e., create) a directory called `pset7`.
- Execute `cd pset7` to change into (i.e., open) that directory.
- Execute `wget https://cdn.cs50.net/2020/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.

Understanding

Provided to you is a file called `movies.db`, a SQLite database that stores data from IMDb 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 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 each movie's release year.

You're welcome to check your queries' results against IMDb 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 (not including the 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 (not including the 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 (not including the 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.
    - If a person directed more than one movie that received a rating of at least 9.0, they should only appear in your results once.
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

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.

You can also run

```
$ cat filename.sql | sqlite3 movies.db > output.txt
```

to redirect the output of the query to a text file called `output.txt`. (This can be useful for checking how many rows are returned by your query!)

Hints

- See this SQL keywords reference for some SQL syntax that may be helpful!

Testing

While `check50` is available for this problem, you're encouraged to instead test your code on your own for each of the following. 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 9,545 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 50,863 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 6,864 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 18,237 rows.
- Executing `10.sql` results in a table with 1 column and 1,887 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.

Note that row counts do not include header rows that only show column names.

If your query returns a number of rows that is slightly different from the expected output, be sure that you're properly handling duplicates! For queries that ask for a list of names, no one person should be listed twice, but two different people who have the same name should each be listed.

Execute the below to evaluate the correctness of your code using `check50`.

```
check50 cs50/problems/2021/x/movies
```

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/2021/x/movies
```

Acknowledgements

Information courtesy of IMDb (imdb.com). Used with permission.

Fiftyville

Write SQL queries to solve a mystery.

A Mystery in Fiftyville

The CS50 Duck has been stolen! The town of Fiftyville has called upon you to solve the mystery of the stolen duck. Authorities believe that the thief stole the duck and then, shortly afterwards, took a flight out of town with the help of an accomplice. Your goal is to identify:

- Who the thief is,
- What city the thief escaped to, and
- Who the thief's accomplice is who helped them escape

All you know is that the theft **took place on July 28, 2020** and that it **took place on Chamberlin Street**.

How will you go about solving this mystery? The Fiftyville authorities have taken some of the town's records from around the time of the theft and prepared a SQLite database for you, `fiftyville.db`, which contains tables of data from around the town. You can query that table using SQL `SELECT` queries to access the data of interest to you. Using just the information in the database, your task is to solve the mystery.

Getting Started

Here's how to download this problem into your own CS50 IDE. Log into CS50 IDE and then, in a terminal window, execute each of the below.

- Navigate to your `pset7` directory that should already exist.
- Execute `wget https://cdn.cs50.net/2020/fall/psets/7/fiftyville/fiftyville.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip fiftyville.zip` to uncompress that file.
- Execute `rm fiftyville.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `fiftyville`, which was inside of that ZIP file.
- Execute `cd fiftyville` to change into that directory.
- Execute `ls`. You should see a `fiftyville.db` file, a `log.sql` file, and an `answers.txt` file.

Specification

For this problem, equally as important as solving the mystery itself is the process that you use to solve the mystery. In `log.sql`, keep a log of all SQL queries that you run on the database. Above each query, label each with a comment (in SQL, comments are any lines that begin with `--`) describing why you're running the query and/or what information you're hoping to get out of that particular query. You can use comments in the log file to add additional notes about your thought process as you solve the mystery: ultimately, this file should serve as evidence of the process you used to identify the thief!

Once you solve the mystery, complete each of the lines in `answers.txt` by filling in the name of the thief, the city that the thief escaped to, and the name of the thief's accomplice who helped them escape town. (Be sure not to change any of the existing text in the file or to add any other lines to the file!)

Ultimately, you should submit both your `log.sql` and `answers.txt` files.

Walkthrough

Hints

- Execute `sqlite3 fiftyville.db` to begin running queries on the database.
  - While running `sqlite3`, executing `.tables` will list all of the tables in the database.
  - While running `sqlite3`, executing `.schema TABLE_NAME`, where `TABLE_NAME` is the name of a table in the database, will show you the `CREATE TABLE` command used to create the table. This can be helpful for knowing which columns to query!
- You may find it helpful to start with the `crime_scene_reports` table. Start by looking for a crime scene report that matches the date and the location of the crime.
- See this SQL keywords reference for some SQL syntax that may be helpful!

Testing

Execute the below to evaluate the correctness of your code using `check50`.

```
check50 cs50/problems/2021/x/fiftyville
```

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/2021/x/fiftyville
```

Inspired by another case over at SQL City.

Week 8 HTML, CSS, JavaScript

Lecture 8

The internet

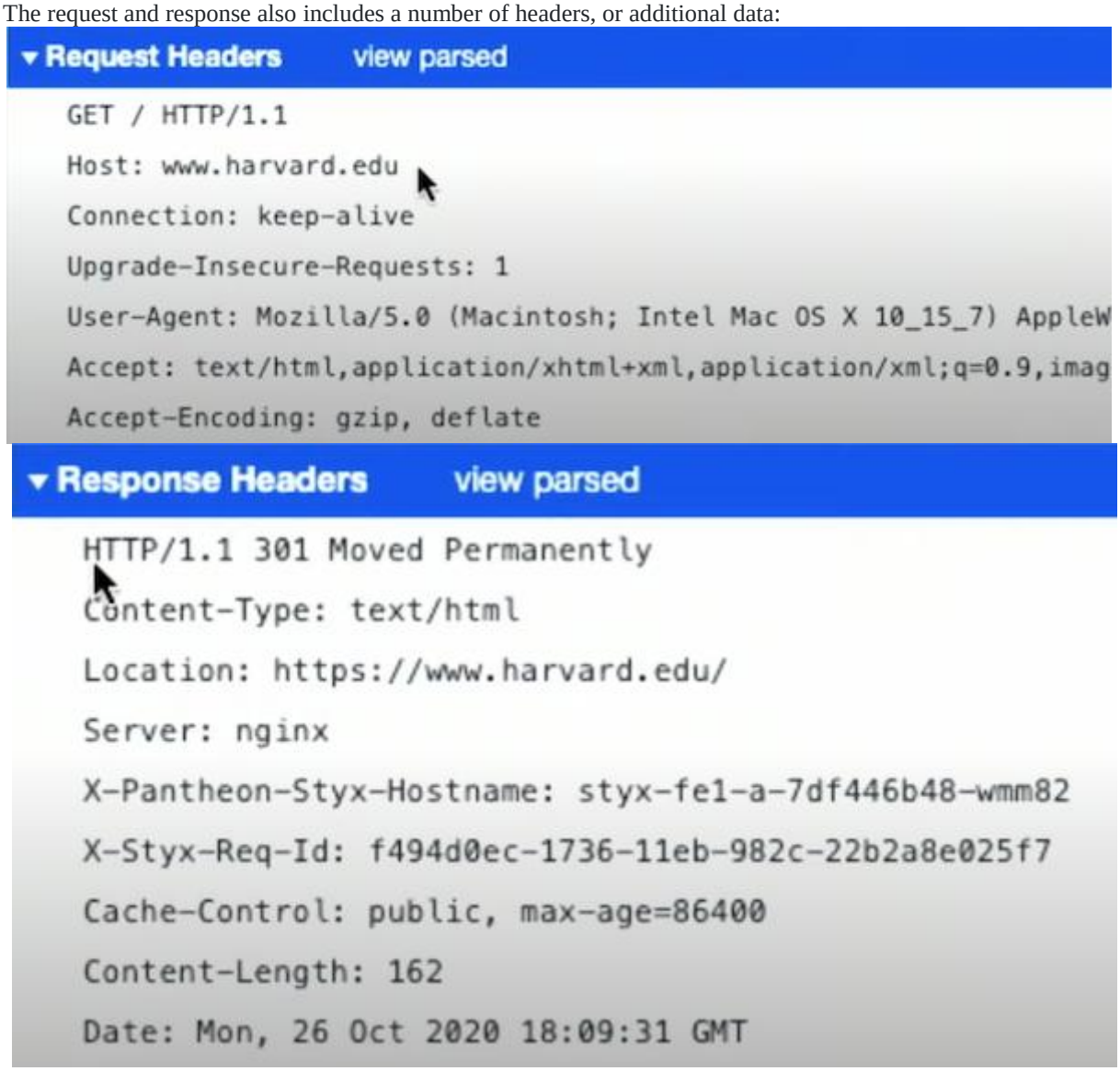
- Today we'll take a look at web programming, using a set of new languages and technologies to create graphical and visual applications for the internet.
- The **internet** is the network of networks of computers communicating with one another, which provides the infrastructure to send zeros and ones. On top of that foundation, we can build applications that send and receive data.
- **Routers** are specialized computers, with CPUs and memory, whose purpose is to relay data across cables or wireless technologies, between other devices on the internet.
- **Protocols** are a set of standard conventions, like a physical handshake, that the world has agreed upon for computers to communicate with. For example, there are certain patterns of zeros and ones, or messages, a computer has to use to tell a router where it wants to send data.
- **TCP/IP** are two protocols for sending data between two computers. In the real world, we might write an address on an envelope in order to send a letter to someone, along with our own address for a letter in return. The digital version of an envelope, or a message with from and to addresses, is called a **packet**.
- **IP** stands for internet protocol, a protocol supported by modern computers' software, which includes a standard way for computers to address each other. **IP addresses** are unique addresses for computers connected to the internet, such that a packet sent from one computer to another will be passed along routers until it reaches its destination.
    - Routers have, in their memory, a table mapping IP addresses to cables each connected to other routers, so they know where to forward packets to. It turns out that there are protocols for routers to communicate and figure out these paths as well.
- **DNS**, domain name system, is another technology that translates domain names like cs50.harvard.edu to IP addresses. DNS is generally provided as a service by the nearest internet service provider, or ISP.
- Finally, **TCP**, transmission control protocol, is one final protocol that allows a single server, at the same IP address, to provide multiple services through the use of a **port number**, a small integer added to the IP address. For example, HTTP, HTTPS, email, and even Zoom has their own port numbers for those programs to use to communicate over the network.
- TCP also provides a mechanism for resending packets if a packet is somehow lost and not received. It turns out that, on the internet, there are multiple paths for a packet to be sent since there are lots of routers that are interconnected. So a web browser, making a request for a cat, might see its packet sent through one path of routers, and the responding server might see its response packets sent through another.
    - A large amount of data, such as a picture, will be broken into smaller chunks so that the packets are all of a similar size. This way, routers along the internet can send everyone's packets along more fairly and easily. **Net neutrality** refers to the idea that these public routers treat packets equally, as opposed to allowing packets from certain companies or of certain types to be prioritized.
    - When there are multiple packets for a single response, TCP will also specify that each of them be labeled, as with "1 of 2" or "2 of 2", so they can be combined or re-sent as needed.
- With all of these technologies and protocols, we're able to send data from one computer to another, and can abstract the internet away, to build applications on top.

Web development

- The web is one application running on top of the internet, allowing us to get web pages. Other applications like Zoom provide video conferencing, and email is another application as well.
- **HTTP**, or Hypertext Transfer Protocol, governs how web browsers and web servers communicate within TCP/IP packets.
- Two commands supported by HTTP include **GET** and **POST**. GET allows a browser to ask for a page or file, and POST allows a browser to send data *to* the server.
- A **URL**, or web address, might look like https://www.example.com/.

    - https is the protocol being used, and in this case HTTPS is the secure version of HTTP, ensuring that the contents of packets between the browser and server are encrypted.
    - example.com is the domain name, where .com is the top-level domain, conventionally indicating the "type" of website, like a commercial website for .com, or an organization for .org. Now there are hundreds of top-level domains, and they vary in restrictions on who can use them, but many of them allow anyone to register for a domain.
    - www is the hostname that, by convention, indicates to us that this is a "world wide web" service. It's not required, so today many websites aren't configured to include it.
    - Finally, the / at the end is a request for the default file, like index.html, that the web server will respond with.
- An HTTP request will start with:
- GET / HTTP/1.1
- Host: www.example.com
- ...
    - The GET indicates that the request is for some file, and / indicates the default file. A request could be more specific, and start with GET /index.html.
    - There are different versions of the HTTP protocol, so HTTP/1.1 indicates that the browser is using version 1.1.
    - Host: www.example.com indicates that the request is for www.example.com, since the same web server might be hosting multiple websites and domains.
- A response will start with:
- HTTP/1.1 200 OK
- Content-Type: text/html
- ...
    - The web server will respond with the version of HTTP, followed by a status code, which is 200 OK here, indicating that the request was valid.
    - Then, the web server indicates the type of content in its response, which might be text, image, or other format.
    - Finally, the rest of the packet or packets will include the content.
- We can see a redirect in a browser by typing in a URL, like http://www.harvard.edu, and looking at the address bar after the page has loaded, which will show https://www.harvard.edu. Browsers include developer tools, which allow us to see what's happening. In Chrome's menu, for example, we can go to View > Developer > Developer Tools, which will open a panel on the screen. In the Network tab, we can see that there were many requests, for text, images, and other pieces of data that were downloaded separately for the single web pages.
- The first request actually returned a status code of 301 Moved Permanently, redirecting our browser from http://... to https://...:

- The request and response also includes a number of headers, or additional data:



  - Note that the response includes a `Location:` header for the browser to redirect us to.
- Other HTTP status codes include:
  - 200 OK
  - 301 Moved Permanently
  - 304 Not Modified
    - This allows the browser to use its cache, or local copy, of some resource like an image, instead of having the server send it back again.
  - 307 Temporary Redirect
  - 401 Unauthorized
  - 403 Forbidden
  - 404 Not Found
  - 418 I'm a Teapot
  - 500 Internal Server Error
    - Buggy code on a server might result in this status code.
  - 503 Service Unavailable
  - …
- We can use a command-line tool, `curl`, to connect to a URL. We can run:
- curl -I http://safetyschool.org
- HTTP/1.1 301 Moved Permanently
- Server: Sun-ONE-Web-Server/6.1
- Date: Wed, 26 Oct 2020 18:17:05 GMT
- Content-length: 122
- Content-type: text/html
- Location: http://www.yale.edu
- Connection: close

    - It turns out that safetyschool.org redirects to yale.edu!
    - And harvardsucks.org is a website with another prank on Harvard!
- Finally, an HTTP request can include inputs to servers, like the string q=cats after the ?:

```
GET /search?q=cats HTTP/1.1
Host: www.google.com
...
```

    - This uses a standard format for passing input, like command-line arguments, to web servers.

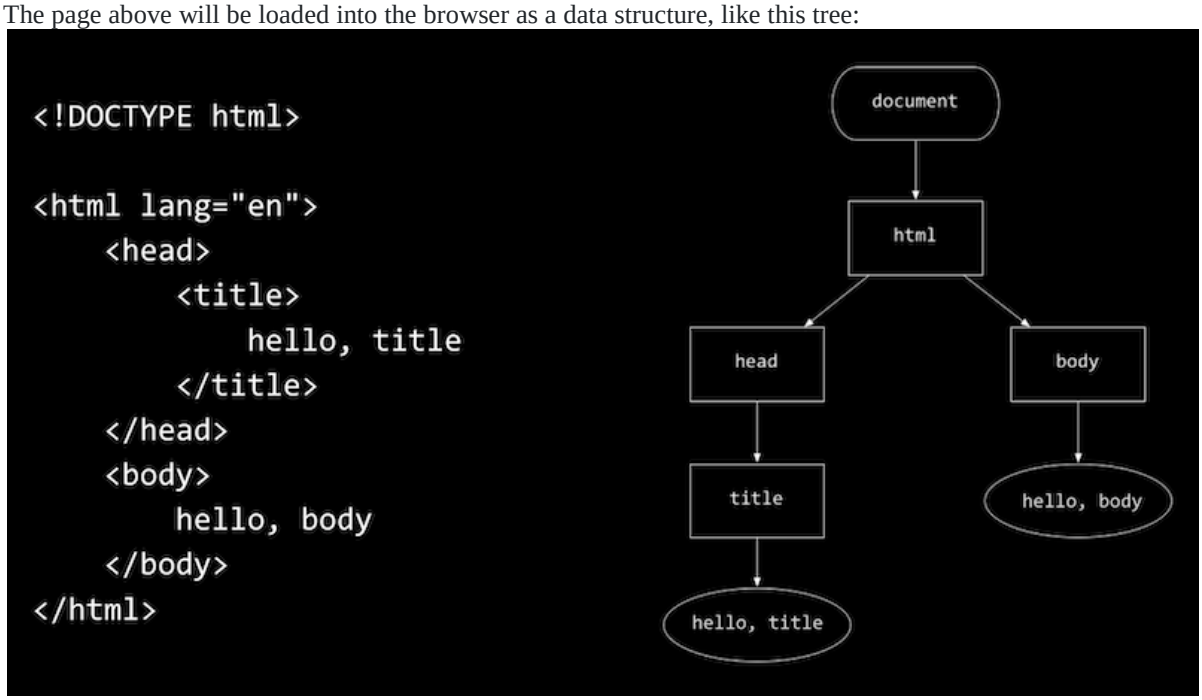
HTML

- Now that we can use the internet and HTTP to send and receive messages, it's time to see what's in the content for web pages. **HTML**, Hypertext Markup Language, is not a programming language, but rather used to format web pages and tell the browser how to display pages, using tags and attributes.
- A simple page in HTML might look like this:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>
      hello, title
    </title>
  </head>
  <body>
    hello, body
  </body>
</html>
```

    - The first line is a declaration that the page follows the HTML standard.
    - Next is a **tag**, a word in brackets like <html> and </html>. The first is a start or open tag, and the second is a close tag. In this case, the tags indicate the start and end of the HTML page. The start tag here has an **attribute** as well, lang="en" which specifies that the language of the page will be in English, to help the browser translate the page if needed.
    - Within the <html> tag are two more tags, <head> and <body>, which are both like children nodes in a tree. And within <head> is the <title> tag, the contents of which we see in a tab or window's title in a browser. Within <body> is the contents of the page itself, which we'll see in the main view of a browser as well.
- The page above will be loaded into the browser as a data structure, like this tree:



    - Note that there is a hierarchy mapping each tag and its children. Rectangular nodes are tags, while oval ones are text.
- We can save the code above as an HTML on our local computers, which would work in a browser, but just for us. With the CS50 IDE, we can create an HTML file, and actually make it available over the internet.

- We'll create hello.html with the code above, and start a web server installed on the CS50 IDE with http-server, a program that will listen for HTTP requests and respond with pages or other content.
- The CS50 IDE itself is already running on some web server, using ports 80 and 443, so our own web server within the IDE will have to use a different port, 8080 by default. We'll see a long URL, ending in cs50.ws, and if we open that URL we'll see a listing of files, including hello.html.
- Back in the terminal of our IDE, we'll see new rows of text printed by our web server, a log of requests that it's getting.
- We'll take a look at paragraphs.html.
    - With the <p> tag, we can indicate that each section of text should be a paragraph.
    - After we save this file, we'll need to refresh the index in the web browser, and then open paragraphs.html.
- We can add headings with tags that start with h, and have levels of 1 through 6 in headings.html.
- We take a look at list.html, table.html, and image.html as well, to add lists, tables, and images.
    - We can use the <ul> tag to create an unordered list, like bullet points, and <ol> for an ordered list with numbers.
    - Tables start with a <table> tag and have <tr> tags as rows, and <td> tags for individual cells.
    - For image.html, we can upload an image to the CS50 IDE, to include it in our page, as well as use the alt attribute to add alternative text for accessibility.
- By looking for documentation or other online resources, we can learn the tags that exist in HTML, and how to use them.
- We can create links in link.html with the <a>, or anchor, tag. The href attribute is for a hypertext reference, or simply where the link should take us, and within the tag is the text that should appear as the link.
    - We could set the href to https://www.yale.edu, but leave Harvard within the tag, which might prank users or even trick them into visiting a fake version of some website. **Phishing** is an act of tricking users, a form of social engineering that includes misleading links.
- In search.html, we can create a more complex form that takes user input and sends it to Google's search engine:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>search</title>
  </head>
  <body>
    <form action="https://www.google.com/search" method="get">
      <input name="q" type="search">
      <input type="submit" value="Search">
    </form>
  </body>
</html>
```

    - First, we have a <form> tag that has an action of Google's search URL, with a method of GET.
    - Inside the form, we have one <input>, with the name q, and another <input> with the type of submit. When the second input, a button, is clicked, the form will append the text in the first input to the action URL, ending it with search?q=....
    - So when we open search.html in our browser, we can use the form to search via Google.
    - A form can also use a POST method, which doesn't include the form's data in the URL, but elsewhere in the request.

CSS

- We can improve the aesthetics of our pages with **CSS**, Cascading Style Sheets, another language that tells our browser how to display tags on a page. CSS uses **properties**, or key-value pairs, like color: red; to tags with selectors.
- In HTML, we have some options for including CSS. We can add a <style> tag within the <head> tag, with styles direcly inside, or we can link to a styles.css file with a <link> tag within the <head> tag.
- We can also include CSS directly in each tag:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>css</title>
  </head>
```

```
    <body>
      <header style="font-size: large; text-align: center;">
        John Harvard
      </header>
      <main style="font-size: medium; text-align: center;">
        Welcome to my home page!
      </main>
      <footer style="font-size: small; text-align: center;">
        Copyright &#169; John Harvard
      </footer>
    </body>
</html>
```

- `<header>`, `<main>`, and `<footer>` tags are like `<p>` tags, indicating the sections that the text on our page are in.
- For each tag, we can add a `style` attribute, with the value being a list of CSS key-value properties, separated by semicolons. Here, we're setting the `font-size` for each tag, and aligning the text in the center.
- Note that we can use `&#169;`, an **HTML entity**, as a code to include some symbol in our web page.

- We can align all the text at once:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <title>css</title>
  </head>
  <body style="text-align: center;">
    <header style="font-size: large;">
      John Harvard
    </header>
    <main style="font-size: medium;">
      Welcome to my home page!
    </main>
    <footer style="font-size: small;">
      Copyright &#169; John Harvard
    </footer>
  </body>
</html>
```

- Here, the style applied to the `<body>` tag cascades, or applies, to its children, so all the sections inside will have centered text as well.

- To factor out, or separate our CSS from HTML, we can include styles in the `<head>` tag:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <style>

      header
      {
        font-size: large;
        text-align: center;
      }

      main
      {
        font-size: medium;
        text-align: center;
      }

      footer
      {
        font-size: small;
```

```
            text-align: center;
        }

    </style>
    <title>css</title>
</head>
<body>
    <header>
        John Harvard
    </header>
    <main>
        Welcome to my home page!
    </main>
    <footer>
        Copyright &#169; John Harvard
    </footer>
</body>
</html>
```

- For each *type* of tag, we've used a CSS **type selector** to style it.

- We can also use a more specific **class selector**:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            .centered
            {
                text-align: center;
            }

            .large
            {
                font-size: large;
            }

            .medium
            {
                font-size: medium;
            }

            .small
            {
                font-size: small;
            }

        </style>
        <title>css</title>
    </head>
    <body>
        <header class="centered large">
            John Harvard
        </header>
        <main class="centered medium">
            Welcome to my home page!
        </main>
        <footer class="centered small">
            Copyright &#169; John Harvard
        </footer>
```

```
    </body>
</html>
```

- We can define our own CSS class with a `.` followed by a keyword we choose, so here we've created `.large`, `.medium`, and `.small`, each with some property for the font size.
- Then, on any number of tags in our page's HTML, we can add one or more of these classes with `class="centered large"`, reusing these styles.
- We can remove the redundancy for `centered`, and apply it to just the `<body>` tag as well.

- Finally, we can take all of the CSS for the properties and move them to another file with the `<link>` tag:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <link href="styles.css" rel="stylesheet">
        <title>css</title>
    </head>
    <body>
        <header class="centered large">
            John Harvard
        </header>
        <main class="centered medium">
            Welcome to my home page!
        </main>
        <footer class="centered small">
            Copyright &#169; John Harvard
        </footer>
    </body>
</html>
```

- Now, one person can work on the HTML and another can work on the CSS, more independently.

- With CSS, we'll also rely on references and other resources to look up how to use properties as we need them.
- We can use **pseudoselectors**, which selects certain states:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <style>

            #harvard
            {
                color: #ff0000;
            }

            #yale
            {
                color: #0000ff;
            }

            a
            {
                text-decoration: none;
            }

            a:hover
            {
                text-decoration: underline;
            }

        </style>
        <title>link</title>
    </head>
```

```
    <body>
        Visit <a href="https://www.harvard.edu/" id="harvard" >Harvard</a> or <a href="https://www.yale.edu/" id="yale" >Yale</a>.
    </body>
</html>
```

- Here, we're using `a:hover` to set properties on `<a>` tags when the user hovers over them.
- We also have an `id` attribute on each `<a>` tag, to set different colors on each with **ID selectors** that start with a # in CSS.

JavaScript

- To write code that can run in users' browsers, or on the client, we'll use a new language, **JavaScript**.
- The syntax of JavaScript is similar to that of C and Python for basic constructs:

```
let counter = 0;
counter = counter + 1;
counter += 1;
counter++;
if (x < y)
{

}
if (x < y)
{

}
else
{

}
if (x < y)
{

}
else if (x > y)
{

}
else
{

}
while (true)
{

}
for (let i = 0; i < 3; i++)
{

}
```

- Notice that JavaScript is loosely typed as well, with `let` being the keyword to declare variables of any type.

- With JavaScript, we can change the HTML in the browser in real-time. We can use `<script>` tags to include our code directly, or from a `.js` file.
- We'll create another form:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

```

```
            function greet()
            {
               alert('hello, body');
            }

        </script>
        <title>hello</title>
    </head>
    <body>
        <form onsubmit="greet(); return false;">
            <input id="name" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

- Here, we won't add an `action` to our form, since this will stay on the same page. Instead, we'll have an `onsubmit` attribute that will call a function we've defined in JavaScript, and use `return false;` to prevent the form from actually being submitted anywhere.
- Now, if we load that page, we'll see `hello, body` being shown when we submit the form.

- Since our input tag, or **element**, has an ID of `name`, we can use it in our script:

```
<script>

    function greet()
    {
        let name = document.querySelector('#name').value;
        alert('hello, ' + name);
    }

</script>
```

- `document` is a global variable that comes with JavaScript in the browser, and `querySelector` is another function we can use to select a node in the **DOM**, Document Object Model, or the tree structure of the HTML page. After we select the element with the ID `name`, we get the `value` inside the input, and add it to our alert.
- Note that JavaScript uses single quotes for strings by convention, though double quotes can be used as well as long as they match for each string.

- We can add more attributes to our form, to change placeholder text, change the button's text, disable autocomplete, or autofocus the input:

```
<form>
    <input autocomplete="off" autofocus id="name" placeholder="Name" type="text">
    <input type="submit">
</form>
```

- We can also listen to **events** in JavaScript, which occur when something happens on the page. For example, we can listen to the `submit` event on our form, and call the `greet` function:

```
<script>

    function greet()
    {
        let name = document.querySelector('#name').value;
        alert('hello, ' + name);
    }

    function listen() {
        document.querySelector('form').addEventListener('submit', greet);
    }

    document.addEventListener('DOMContentLoaded', listen);

</script>
```

- Here, in `listen` we pass the function `greet` by name, and not call it yet. The event listener will call it for us when the event happens.

    - We need to first listen to the `DOMContentLoaded` event, since the browser reads our HTML file from top to bottom, and `form` wouldn't exist until it's read the entire file and loaded the content. So by listening to this event, and calling our `listen` function, we know `form` will exist.
- We can also use **anonymous functions** in JavaScript:

```
<script>

    document.addEventListener('DOMContentLoaded', function() {
        document.querySelector('form').addEventListener('submit', function() {
            let name = document.querySelector('#name').value;
            alert('hello, ' + name);
        });
    });

</script>
```

    - We can pass in a lambda function with the `function()` syntax, so here we've passed in both listeneres directly to `addEventListener`.
- In addition to `submit`, there are many other events we can listen to:
    - blur
    - change
    - click
    - drag
    - focus
    - keyup
    - load
    - mousedown
    - mouseover
    - mouseup
    - submit
    - touchmove
    - unload
    - ...
- For example, we can listen to the `keyup` event, and change the DOM as soon as we release a key:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <script>

            document.addEventListener('DOMContentLoaded', function() {
                let input = document.querySelector('input');
                input.addEventListener('keyup', function(event) {
                    let name = document.querySelector('#name');
                    if (input.value) {
                        name.innerHTML = `hello, ${input.value}`;
                    }
                    else {
                        name.innerHTML = 'hello, whoever you are';
                    }
                });
            });

        </script>
        <title>hello</title>
    </head>
    <body>
        <form>
            <input autocomplete="off" autofocus placeholder="Name" type="text">
        </form>
        <p id="name"></p>
    </body>
</html>
```

    - Notice that we can substitute strings in JavaScript as well, with the `${input.value}` inside a string surrounded by backticks, `.
- We can programmatically change style, too:

```
<!DOCTYPE html>

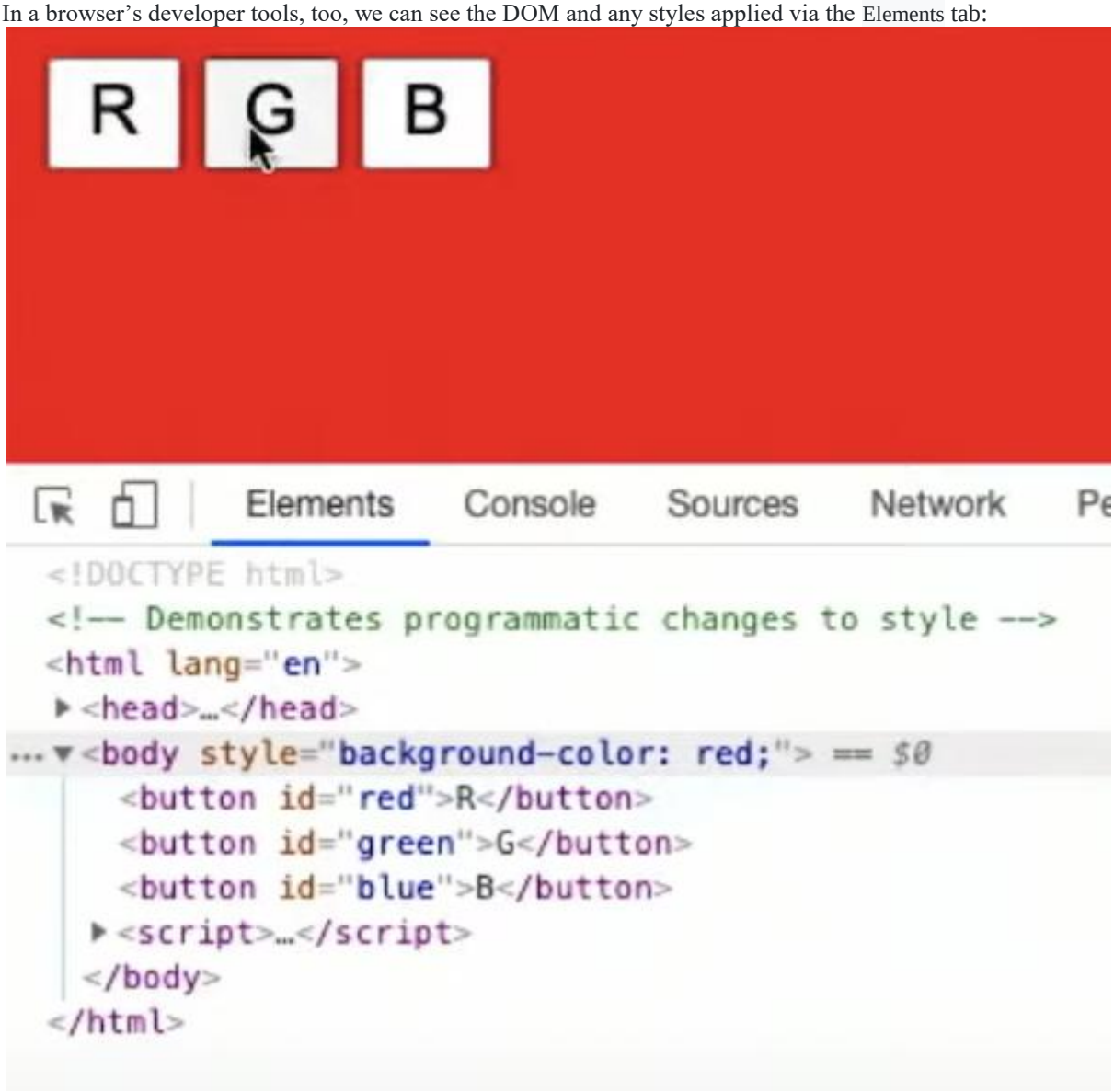
<html lang="en">
  <head>
    <title>background</title>
  </head>
  <body>
    <button id="red">R</button>
    <button id="green">G</button>
    <button id="blue">B</button>
    <script>

      let body = document.querySelector('body');
      document.querySelector('#red').onclick = function() {
        body.style.backgroundColor = 'red';
      };
      document.querySelector('#green').onclick = function() {
        body.style.backgroundColor = 'green';
      };
      document.querySelector('#blue').onclick = function() {
        body.style.backgroundColor = 'blue';
      };

    </script>
  </body>
</html>
```

    - After selecting an element, we can use the `style` property to set values for CSS properties as well. Here, we have three buttons, each of which has an `onclick` listener that changes the background color of the `<body>` element.
    - Notice here that our `<script>` tag is at the end of our HTML file, so we don't need to listen to the `DOMContentLoaded` event, since the rest of the DOM will already have been read by the browser.

- In a browser's developer tools, too, we can see the DOM and any styles applied via the `Elements` tab:



  - We can even use this to change a page in our browser after it's loaded, by clicking on some element and editing the HTML. But these changes will only be made in our browser, not in our original HTML file or on some web page elsewhere.
- In `size.html`, we can set the font size with a dropdown via JavaScript, and in `blink.html` we can make an element "blink", alternating between visible and hidden.
- With `geolocation.html`, we can ask the browser for a user's GPS coordinates, and with `autocomplete.html`, we can autocomplete something we typed in, with words from a dictionary file.
- Finally, we can use Python to write code that connect to other devices on a local network, like a light bulb, via an **API**, application programming interface. Our light bulb's API in particular accepts request at certain URLs:

```
import os
import requests

USERNAME = os.getenv("USERNAME")
IP = os.getenv("IP")

URL = f"http://{IP}/api/{USERNAME}/lights/1/state"

requests.put(URL, json={"on": False})
```

  - With this code, we can use the PUT method to send a message to our light bulb, turning it off.
  - We use environment variables, values stored elsewhere on our computer, for our username and IP address.
- Now, with a little more logic, we can make our light bulb blink:

```
import os
import requests
```

```
import time

USERNAME = os.getenv("USERNAME")
IP = os.getenv("IP")
URL = f"http://{IP}/api/{USERNAME}/lights/1/state"

while True:
    requests.put(URL, json={"bri": 254, "on": True})
    time.sleep(1)
    requests.put(URL, json={"on": False})
    time.sleep(1)
```

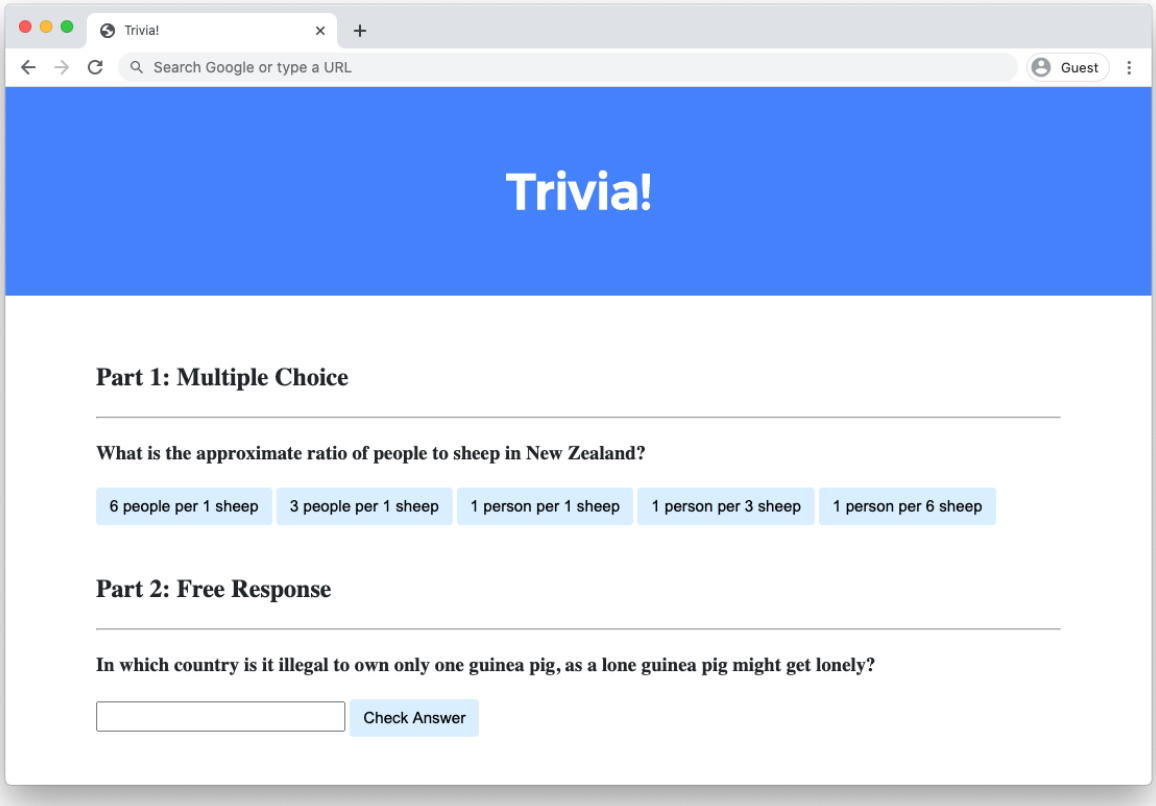
- We'll put together HTML, CSS, JavaScript, Python, and SQL next time!

Lab 8: Trivia

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Write a webpage that lets users answer trivia questions.



When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Getting Started

Here's how to download this lab into your own CS50 IDE. Log into CS50 IDE 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 `~`).
- Execute `wget https://cdn.cs50.net/2020/fall/labs/8/lab8.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip lab8.zip` to uncompress that file.
- Execute `rm lab8.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `lab8`, which was inside of that ZIP file.
- Execute `cd lab8` to change into that directory.
- Execute `ls`. You should see an `index.html` and a `styles.css` file.

Implementation Details

Design a webpage using HTML, CSS, and JavaScript to let users answer trivia questions.

- In `index.html`, add beneath "Part 1" a multiple-choice trivia question of your choosing with HTML.
  - You should use an `h3` heading for the text of your question.

    - You should have one `button` for each of the possible answer choices. There should be at least three answer choices, of which exactly one should be correct.
- Using JavaScript, add logic so that the buttons change colors when a user clicks on them.
    - If a user clicks on a button with an incorrect answer, the button should turn red and text should appear beneath the question that says "Incorrect".
    - If a user clicks on a button with the correct answer, the button should turn green and text should appear beneath the question that says "Correct!".
- In `index.html`, add beneath "Part 2" a text-based free response question of your choosing with HTML.
    - You should use an `h3` heading for the text of your question.
    - You should use an `input` field to let the user type a response.
    - You should use a `button` to let the user confirm their answer.
- Using JavaScript, add logic so that the text field changes color when a user confirms their answer.
    - If the user types an incorrect answer and presses the confirmation button, the text field should turn red and text should appear beneath the question that says "Incorrect".
    - If the user types the correct answer and presses the confirmation button, the input field should turn green and text should appear beneath the question that says "Correct!".

Optionally, you may also:

- Edit `styles.css` to change the CSS of your webpage!
- Add additional trivia questions to your trivia quiz if you would like!

Walkthrough

Hints

- Use `document.querySelector` to query for a single HTML element.
- Use `document.querySelectorAll` to query for multiple HTML elements that match a query. The function returns an array of all matching elements.

Not sure how to solve?

Testing

No `check50` for this lab, as implementations will vary based on your questions! But be sure to test both incorrect and correct responses for each of your questions to ensure that your webpage responds appropriately.

Run `http-server` in your terminal while in your `lab8` directory to start a web server that serves your webpage.

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/trivia
```

Problem Set 8

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 8** before beginning this problem set.

1. Submit Homepage

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Homepage

Build a simple homepage using HTML, CSS, and JavaScript.

Background

The internet has enabled incredible things: we can use a search engine to research anything imaginable, communicate with friends and family members around the globe, play games, take courses, and so much more. But it turns out that nearly all pages we may visit are built on three core languages, each of which serves a slightly different purpose:

1. HTML, or *HyperText Markup Language*, which is used to describe the content of websites;
2. CSS, *Cascading Style Sheets*, which is used to describe the aesthetics of websites; and
3. JavaScript, which is used to make websites interactive and dynamic.

Create a simple homepage that introduces yourself, your favorite hobby or extracurricular, or anything else of interest to you.

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 and then, in a terminal window, execute each of the below.

- Execute `cd ~` (or simply `cd` with no arguments) to ensure that you're in your home directory.
- Execute `mkdir pset8` to make (i.e., create) a directory called `pset8`.
- Execute `cd pset8` to change into (i.e., open) that directory.
- Execute `wget http://cdn.cs50.net/2020/fall/psets/8/homepage/homepage.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip homepage.zip` to uncompress that file.
- Execute `rm homepage.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `homepage`, which was inside of that ZIP file.
- Execute `cd homepage` to change into that directory.
- Execute `ls`. You should see this problem's distribution, including `index.html` and `styles.css`.
- You can immediately start a server to view the site by running

```
$ http-server
```

in the terminal window and clicking on the link that appears.

Specification

Implement in your `homepage` directory a website that must:

- Contain at least four different `.html` pages, at least one of which is `index.html` (the main page of your website), and it should be possible to get from any page on your website to any other page by following one or more hyperlinks.
- Use at least ten (10) distinct HTML tags besides `<html>`, `<head>`, `<body>`, and `<title>`. Using some tag (e.g., `<p>`) multiple times still counts as just one (1) of those ten!
- Integrate one or more features from Bootstrap into your site. Bootstrap is a popular library (that comes with lots of CSS classes and more) via which you can beautify your site. See Bootstrap's documentation to get started. In particular, you might find some of Bootstrap's components of interest. To add Bootstrap to your site, it suffices to include

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/css/bootstrap.min.css" integrity="sha384-TX8t27EcRE3e/ihU7zmQxVncDAy5uIKz4rEkgIXeMed4M0jlfIDPvg6uqKI2xXr2" crossorigin="anonymous">
<script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-DfXdz2htPH0lsSSs5nCTpuj/zy4C+OGpamoFVy38MVBnE+IbbVYUew+OrCXaRkfj" crossorigin="anonymous"></script>
<script src="https://cdn.jsdelivr.net/npm/bootstrap@4.5.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ho+j7jyWK8fNQe+A12Hb8AhRq26LrZ/JpcUGGOn+Y7RsweNrtN/tE3MoK7ZeZDyx" crossorigin="anonymous"></script>
```

in your pages' `<head>`, below which you can also include

```
<link href="styles.css" rel="stylesheet">
```

to link your own CSS.

- Have at least one stylesheet file of your own creation, `styles.css`, which uses at least five (5) different CSS selectors (e.g. tag (`example`), class (`.example`), or ID (`#example`)), and within which you use a total of at least five (5) different CSS properties, such as `font-size`, or `margin`; and
- Integrate one or more features of JavaScript into your site to make your site more interactive. For example, you can use JavaScript to add alerts, to have an effect at a recurring interval, or to add interactivity to buttons, dropdowns, or forms. Feel free to be creative!
- Ensure that your site looks nice on browsers both on mobile devices as well as laptops and desktops.

Testing

If you want to view how your site looks while you work on it, there are two options:

1. Within CS50 IDE, navigate to your `homepage` directory (remember how?) and then execute
2. `$ http-server`
3. Within CS50 IDE, right-click (or Ctrl+click, on a Mac) on the `homepage` directory in the file tree at left. From the options that appear, select **Serve**, which should open a new tab in your browser (it may take a second or two) with your site therein.

Recall also that by opening Developer Tools in Google Chrome, you can *simulate* visiting your page on a mobile device by clicking the phone-shaped icon to the left of **Elements** in the developer tools window, or, once the Developer Tools tab has already been opened, by typing `Ctrl+Shift+M` on a PC or `Cmd+Shift+M` on a Mac, rather than needing to visit your site on a mobile device separately!

Assessment

No `check50` for this assignment! Instead, your site's correctness will be assessed based on whether you meet the requirements of the specification as outlined above, and whether your HTML is well-formed and valid. To ensure that your pages are, you can use this Markup Validation Service, copying and pasting your HTML directly into the provided text box. Take care to eliminate any warnings or errors suggested by the validator before submitting!

Consider also:

- whether the aesthetics of your site are such that it is intuitive and straightforward for a user to navigate;
- whether your CSS has been factored out into a separate CSS file(s); and
- whether you have avoided repetition and redundancy by "cascading" style properties from parent tags.

Afraid `style50` does not support HTML files, and so it is incumbent upon you to indent and align your HTML tags cleanly. Know also that you can create an HTML comment with:

```
<!-- Comment goes here -->
```

but commenting your HTML code is not as imperative as it is when commenting code in, say, C or Python. You can also comment your CSS, in CSS files, with:

```
/* Comment goes here */
```

Hints

For fairly comprehensive guides on the languages introduced in this problem, check out these tutorials:

- HTML
- CSS
- JavaScript

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/2021/x/homepage
```

Week 9 Flask

Lecture 9

Web programming

- Today we'll create more advanced web applications by writing code that runs on the server.
- Last week, we used `http-server` in the CS50 IDE as a **web server**, a program that listens for connections and requests, and responds with web pages or other resources.
- An HTTP request has headers, like:

```
GET / HTTP/1.1
...
```

  - These headers can ask for some file or page, or send data from the browser back to the server.
- While `http-server` only responds with static pages, we can use other web servers that parses, or analyzes request headers, like `GET /search?q=cats HTTP/1.1`, to return pages dynamically.

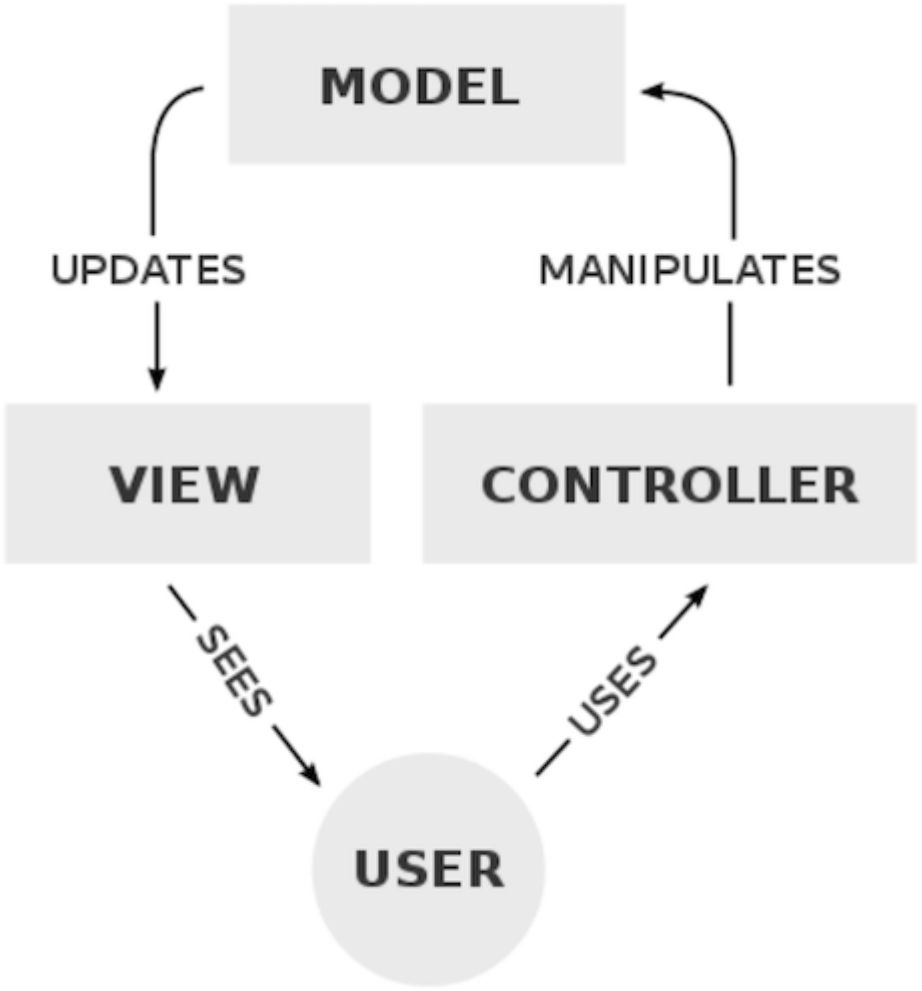
Flask

- We'll use Python and a library called **Flask** to write our own web server, implementing additional features. Flask is also a **framework**, where the library of code also comes with a set of conventions for how it should be used. For example, like other libraries, Flask includes functions we can use to parse requests individually, but as a framework, also requires our program's code to be organized in a certain way:

```
application.py
requirements.txt
static/
templates/
```

  - `application.py` will have the Python code for our web server.
  - `requirements.txt` includes a list of required libraries for our application.
  - `static/` is a directory of static files, like CSS and JavaScript files.
  - `templates/` is a directory for files that will be used to create our final HTML.
- There are many web server frameworks for each of the popular languages, and Flask will be a representative one that we use today.

- Flask also implements a particular **design pattern**, or way that our program and code is organized. For Flask, the design pattern is generally **MVC**, or Model–view–controller:



    - The controller is our logic and code that manages our application overall, given user input. In Flask, this will be our Python code.
    - The view is the user interface, like the HTML and CSS that the user will see and interact with.
    - The model is our application's data, such as a SQL database or CSV file.
- The simplest Flask application might look like this:

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def index():
    return "hello, world"
```

    - First, we'll import `Flask` from the `flask` library, which happens to use a capital letter for its main name.
    - Then, we'll create an `app` variable by giving our file's name to the `Flask` variable.
    - Next, we'll label a function for the `/` route, or URL with `@app.route`. The `@` symbol in Python is called a decorator, which applies one function to another.
    - We'll call the function `index`, since it should respond to a request for `/`, the default page. And our function will just respond with a string for now.
- In the CS50 IDE, we can go to the directory with our application code, and type `flask run` to start it. We'll see a URL, and we can open it to see `hello, world`.
- We'll update our code to actually return HTML with the `render_template` function, which finds a file given and returns its contents:

```
from flask import Flask, render_template

```

```
app = Flask(__name__)


@app.route("/")
def index():
    return render_template("index.html")
```

- We'll need to create a `templates/` directory, and create an `index.html` file with some content inside it.
- Now, typing `flask run` will return that HTML file when we visit our server's URL.

- We'll pass in an argument to `render_template` in our controller code:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route("/")
def index():
    return render_template("index.html", name=request.args.get("name", "world"))
```

- It turns out that we can give `render_template` any named argument, like `name`, and it will substitute that in our template, or our HTML file with placeholders.
  - In `index.html`, we'll replace `hello, world` with `hello,` to tell Flask where to substitute the `name` variable:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        hello, {{ name }}
    </body>
</html>
```

- We can use the `request` variable from the Flask library to get a parameter from the HTTP request, in this case also `name`, and fall back to a default of `world` if one wasn't provided.
- Now, when we restart our server after making these changes, and visit the default page with a URL like `/?name=David`, we'll see that same input returned back to us in the HTML generated by our server.

- We can presume that Google's search query, at `/search?q=cats`, is also parsed by some code for the `q` parameter and passed along to some database to get all the results that are relevant. Those results are then used to generate the final HTML page.

Forms

- We'll move our original template into `greet.html`, so it will greet the user with their name. In `index.html`, we'll create a form:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        <form action="/greet" method="get">
            <input name="name" type="text">
            <input type="submit">
        </form>
    </body>
</html>
```

- We'll send the form to the `/greet` route, and have an input for the `name` parameter and one for the submit button.
- In our `applications.py` controller, we'll also need to add a function for the `/greet` route, which is almost exactly what we had for `/` before:

```
@app.route("/")
def index():
```

```
    return render_template("index.html")


@app.route("/greet")
def greet():
    return render_template("greet.html", name=request.args.get("name", "world"))
```

  - Our form at index.html will be static since it can be the same every time.
- Now, we can run our server, see our form at the default page, and use it to generate another page.

POST

- Our form above used the GET method, which includes our form's data in the URL.
- We'll change the method in our HTML: <form action="/greet" method="post">. Our controller will also need to be changed to accept the POST method, and look for the parameter somewhere else:

```
@app.route("/greet", methods=["POST"])
def greet():
    return render_template("greet.html", name=request.form.get("name", "world"))
```

  - While request.args is for parameters in a GET request, we have to use request.form in Flask for parameters in a POST request.
- Now, when we restart our application after making these changes, we can see that the form takes us to /greet, but the contents aren't included in the URL anymore.

Layouts

- In index.html and greet.html, we have some repeated HTML code. With just HTML, we aren't able to share code between files, but with Flask templates (and other web frameworks), we can factor out such common content.
- We'll create another template, layout.html:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <title>hello</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

  - Flask supports Jinja, a templating language, which uses the {% %} syntax to include placeholder blocks, or other chunks of code. Here we've named our block body since it contains the HTML that should go in the <body> element.
- In index.html, we'll use the layout.html blueprint and only define the body block with:

```
{% extends "layout.html" %}

{% block body %}

    <form action="/greet" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        <input type="submit">
    </form>

{% endblock %}
```

- Similarly, in greet.html, we define the body block with just the greeting:

```
{% extends "layout.html" %}

{% block body %}

    hello, {{ name }}

{% endblock %}
```

- Now, if we restart our server, and view the source of our HTML after opening our server's URL, we see a complete page with our form inside our HTML file, generated by Flask.
- We can even reuse the same route to support both GET and POST methods:

```
@app.route("/", methods=["GET", "POST"])
def index():
    if request.method == "POST":
        return render_template("greet.html", name=request.form.get("name", "world"))
    return render_template("index.html")
```

  - First, we check if the `method` of the `request` is a POST request. If so, we'll look for the `name` parameter and return HTML from the `greet.html` template. Otherwise, we'll return HTML from the `index.html`, which has our form.
  - We'll also need to change the form's `action` to the default `/` route.

Frosh IMs

- One of David's first web applications was for students on campus to register for "frosh IMs", intramural sports.
- We'll use a `layout.html` similar to what we had before:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>froshims</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

  - A `<meta>` tag in `<head>` allows us to add more metadata to our page. In this case, we're adding a `content` attribute for the `viewport` metadata, in order to tell the browser to automamtically scale our page's size and fonts to the device.
- In our `application.py`, we'll return our `index.html` template for the default `/` route:

```
from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Dodgeball",
    "Flag Football",
    "Soccer",
    "Volleyball",
    "Ultimate Frisbee"
]

@app.route("/")
def index():
    return render_template("index.html")
```

- Our `index.html` template will look like this:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Register</h1>

    <form action="/register" method="post">

        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        <select name="sport">
            <option disabled selected value="">Sport</option>
            <option value="Dodgeball">Dodgeball</option>
```

```
            <option value="Flag Football">Flag Football</option>
            <option value="Soccer">Soccer</option>
            <option value="Volleyball">Volleyball</option>
            <option value="Ultimate Frisbee">Ultimate Frisbee</option>
        </select>
        <input type="submit" value="Register">

    </form>
{% endblock %}
```

- We'll have a form like before, and have a `<select>` menu with options for each sport.

- In our `application.py`, we'll allow POST for our `/register` route:

```
@app.route("/register", methods=["POST"])
def register():

  if not request.form.get("name") or not request.form.get("sport"):
      return render_template("failure.html")

  return render_template("success.html")
```

- We'll check that our form's values are valid, and then return a template depending on the results, even though we aren't actually doing anything with the data yet.

- But a user can change the form's HTML in their browser, and send a request that contains some other sport as the selected option!
- We'll check that the value for `sport` is valid by creating a list in `application.py`:

```
from flask import Flask, render_template, request

app = Flask(__name__)

SPORTS = [
    "Dodgeball",
    "Flag Football",
    "Soccer",
    "Volleyball",
    "Ultimate Frisbee"
]

@app.route("/")
def index():
    return render_template("index.html", sports=SPORTS)

...
```

- Then, we'll pass that list into the `index.html` template.

- In our template, we can even use loops to generate a list of options from the list of strings passed in as `sports`:

```
...
<select name="sport">
    <option disabled selected value="">Sport</option>
    {% for sport in sports %}
        <option value="{{ sport }}">{{ sport }}</option>
    {% endfor %}
</select>
...
```

- Finally, we can check that the `sport` sent in the POST request is in the list `SPORTS` in `application.py`:

```
...
@app.route("/register", methods=["POST"])
def register():

    if not request.form.get("name") or request.form.get("sport") not in SPORTS:
        return render_template("failure.html")

    return render_template("success.html")
```

- We can change the select menu in our form to be checkboxes, to allow for multiple sports:

```
{% extends "layout.html" %}

{% block body %}
    <h1>Register</h1>

    <form action="/register" method="post">

        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        {% for sport in sports %}
            <input name="sport" type="checkbox" value="{{ sport }}"> {{ sport }}
        {% endfor %}
        <input type="submit" value="Register">

    </form>
{% endblock %}
```

    - In our `register` function, we can call `request.form.getlist` to get the list of checked options.
- We can also use radio buttons, which will allow only one option to be chosen at a time.

Storing data

- Let's store our registered students, or registrants, in a dictionary in the memory of our web server:

```
from flask import Flask, redirect, render_template, request

app = Flask(__name__)

REGISTRANTS = {}

...

@app.route("/register", methods=["POST"])
def register():

    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")

    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    REGISTRANTS[name] = sport

    return redirect("/registrants")
```

    - We'll create a dictionary called `REGISTRANTS`, and in `register` we'll first check the `name` and `sport`, returning a different error message in each case. Then, we can safely store the name and sport in our `REGISTRANTS` dictionary, and redirect to another route that will display registered students.
    - The error message template, meanwhile, will just display the message:

```
{% extends "layout.html" %}

{% block body %}
    {{ message }}
{% endblock %}
```

- Let's add the `/registrants` route and template to show the registered students:

```
@app.route("/registrants")
def registrants():
    return render_template("registrants.html", registrants=REGISTRANTS)
```

- In our route, we'll pass in the `REGISTRANTS` dictionary to the template as a parameter called `registrants`:
- ```
  {% extends "layout.html" %}

  {% block body %}
    <h1>Registrants</h1>
    <table>
      <thead>
        <tr>
          <th>Name</th>
          <th>Sport</th>
        </tr>
      </thead>
      <tbody>
        {% for name in registrants %}
          <tr>
            <td>{{ name }}</td>
            <td>{{ registrants[name] }}</td>
          </tr>
        {% endfor %}
      </tbody>
    </table>
  {% endblock %}
  ```
- Our template will have a table, with a heading row and row for each key and value stored in `registrants`.

- If our web server stops running, we'll lose the data stored, so we'll use a SQLite database with the SQL library from `cs50`:

```
from cs50 import SQL
from flask import Flask, redirect, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///froshims.db")

...
```

- In the IDE's terminal, we can run `sqlite3 froshims.db` to open the database, and use the `.schema` command to see the table with columns of `id`, `name`, and `sport`, which was created in advance.

- Now, in our routes, we can insert and select rows with SQL:

```
@app.route("/register", methods=["POST"])
def register():

    name = request.form.get("name")
    if not name:
        return render_template("error.html", message="Missing name")
    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    db.execute("INSERT INTO registrants (name, sport) VALUES(?, ?)", name, sport)

    return redirect("/registrants")


@app.route("/registrants")
def registrants():
    registrants = db.execute("SELECT * FROM registrants")
    return render_template("registrants.html", registrants=registrants)
```

- Once we've validated the request, we can use `INSERT INTO` to add a row, and similarly, in `registrants()`, we can `SELECT` all rows and pass them to the template as a list of rows.
```

- Our registrants.html template will also need to be adjusted, since each row returned from db.execute is a dictionary. So we can use registrant.name and registrant.sport to access the value of each key in each row:

```
<tbody>
    {% for registrant in registrants %}
        <tr>
            <td>{{ registrant.name }}</td>
            <td>{{ registrant.sport }}</td>
            <td>
                <form action="/deregister" method="post">
                    <input name="id" type="hidden" value="{{ registrant.id }}">
                    <input type="submit" value="Deregister">
                </form>
            </td>
        </tr>
    {% endfor %}
</tbody>
```

- We can even email users with another library, flask_mail:

```
import os
import re

from flask import Flask, render_template, request
from flask_mail import Mail, Message

app = Flask(__name__)
app.config["MAIL_DEFAULT_SENDER"] = os.getenv("MAIL_DEFAULT_SENDER")
app.config["MAIL_PASSWORD"] = os.getenv("MAIL_PASSWORD")
app.config["MAIL_PORT"] = 587
app.config["MAIL_SERVER"] = "smtp.gmail.com"
app.config["MAIL_USE_TLS"] = True
app.config["MAIL_USERNAME"] = os.getenv("MAIL_USERNAME")
mail = Mail(app)
```

    - We've set some sensitive variables outside of our code, in the IDE's environment, so we can avoid including them in our code.
    - It turns out that we can provide configuration details like a username and password and mail server, in this case Gmail's, to the Mail variable, which will send mail for us.
- Finally, in our register route, we can send an email to the user:

```
@app.route("/register", methods=["POST"])
def register():

    email = request.form.get("email")
    if not email:
        return render_template("error.html", message="Missing email")
    sport = request.form.get("sport")
    if not sport:
        return render_template("error.html", message="Missing sport")
    if sport not in SPORTS:
        return render_template("error.html", message="Invalid sport")

    message = Message("You are registered!", recipients=[email])
    mail.send(message)

    return render_template("success.html")
```

    - In our form, we'll also need to ask for an email instead of a name:
    - <input autocomplete="off" name="email" placeholder="Email" type="email">
- Now, if we restart our server and use the form to provide an email, we'll see that we indeed get one sent to us!

Sessions

- **Sessions** are how web servers remembers information about each user, which enables features like allowing users to stay logged in.
- It turns out that servers can send another header in a response, called `Set-Cookie`:

```
HTTP/1.1 200 OK
Content-Type: text/html
Set-Cookie: session=value
...
```

  - **Cookies** are small pieces of data from a web server that the browser saves for us. In many cases, they are large random numbers or strings used to uniquely identify and track a user between visits.
  - In this case, the server is asking our browser to set a cookie for that server, called `session` to a value of `value`.
- Then, when the browser makes another request to the same server, it'll send back cookies that the same server has set before:

```
GET / HTTP/1.1
Host: gmail.com
Cookie: session=value
```

- In the real world, amusement parks might give you a hand stamp so you can return after leaving. Similarly, our browser is presenting our cookies back to the web server, so it can remember who we are.
- Advertising companies might set cookies from a number of websites, in order to track users across all of them. In Incognito mode, by contrast, the browser doesn't send any cookies set from before.
- In Flask, we can use the `flask_session` library to manage this for us:

```
from flask import Flask, redirect, render_template, request, session
from flask_session import Session

app = Flask(__name__)
app.config["SESSION_PERMANENT"] = False
app.config["SESSION_TYPE"] = "filesystem"
Session(app)


@app.route("/")
def index():
    if not session.get("name"):
        return redirect("/login")
    return render_template("index.html")


@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        session["name"] = request.form.get("name")
        return redirect("/")
    return render_template("login.html")


@app.route("/logout")
def logout():
    session["name"] = None
    return redirect("/")
```

  - We'll configure the session library to use the IDE's filesystem, and use `session` like a dictionary to store a user's name. It turns out that Flask will use HTTP cookies for us, to maintain this `session` variable for each user visiting our web server. Each visitor will get their own `session` variable, even though it appears to be global in our code.
  - For our default `/` route, we'll redirect to `/login` if there's no `name` set in `session` for the user yet, and otherwise show a default `index.html` template.
  - For our `/login` route, we'll set `name` in `session` to the form's value sent via POST, and then redirect to the default route. If we visited the route via GET, we'll render the login form at `login.html`.
  - For the `/logout` route, we can clear the value for `name` in `session` by setting it to `None`, and redirect to `/` again.

    - We'll also generally need a requirements.txt that includes the names of libraries we want to use, so they can be installed for our application, but the ones we use here have been preinstalled in the IDE.
- In our login.html, we'll have a form with just a name:

```
{% extends "layout.html" %}

{% block body %}

    <form action="/login" method="post">
        <input autocomplete="off" autofocus name="name" placeholder="Name" type="text">
        <input type="submit" value="Log In">
    </form>

{% endblock %}
```

- And in our index.html, we can check if session.name exists, and show different content:

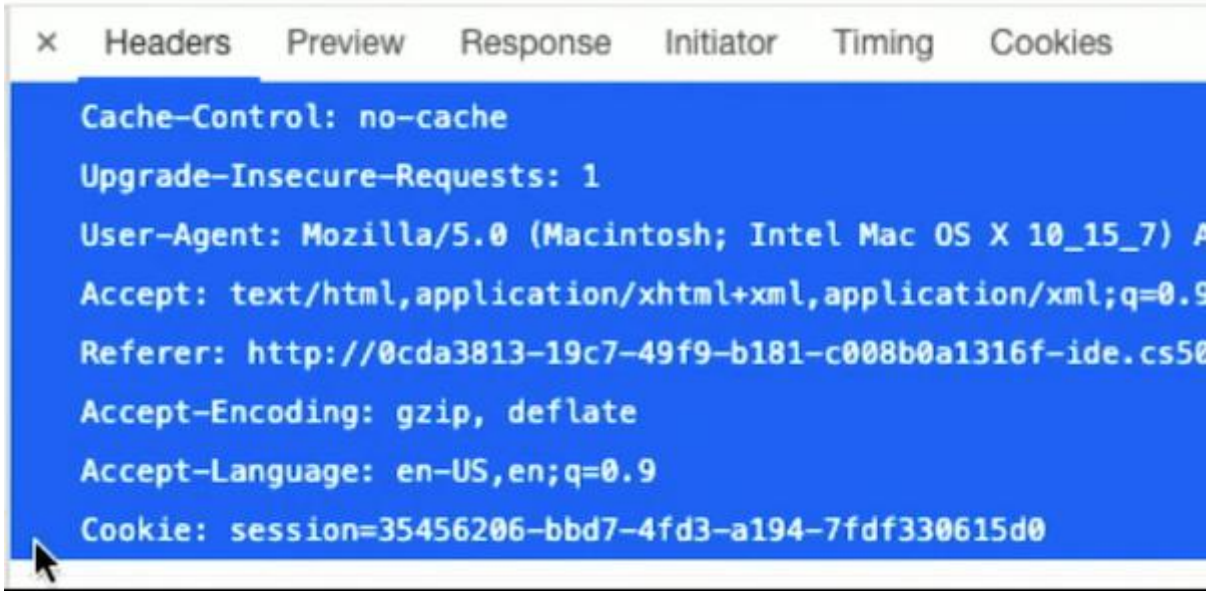
```
{% extends "layout.html" %}

{% block body %}

    {% if session.name %}
        You are logged in as {{ session.name }}. <a href="/logout">Log out</a>.
    {% else %}
        You are not logged in. <a href="/login">Log in</a>.
    {% endif %}

{% endblock %}
```

- When we restart our server, go to its URL, and log in, we can see in the Network tab that our browser is indeed sending a Cookie: header in the request:



store, shows

- We'll look through an example, store:
    - application.py initializes and configures our application to use a database and sessions. In index(), the default route renders a list of books stored in the database.
    - templates/books.html shows the list of books, as well as a form that allows us to click "Add to Cart" for each of them.
    - The /cart route, in turn, stores an id from a POST request in the session variable in a list. If the request used a GET method, however, /cart would show a list of books with ids matching the list of ids stored in session.
- So, "shopping carts" on websites can be implemented with cookies and session variables stored on the server.
- When we view the source generated by our default route, we see that each book has its own <form> element, each with a different id input that's hidden and generated. This id comes from the SQLite database on our server, and is sent back to the /cart route.

- We'll look at another example, `shows`, where we can use both JavaScript on the **front-end**, or side that the user sees, and Python on the **back-end**, or server side.
- In `application.py` here, we'll open a database, `shows.db`:

```
from cs50 import SQL
from flask import Flask, render_template, request

app = Flask(__name__)

db = SQL("sqlite:///shows.db")


@app.route("/")
def index():
    return render_template("index.html")


@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + request.args.get("q") + "%")
    return render_template("search.html", shows=shows)
```

  - The default / route will show a form, where we can type in some search term.
  - The form will use the GET method to send the search query to `/search`, which in turn will use the database to find a list of shows that match. Finally, a `search.html` template will show the list of shows.
- With JavaScript, we can show a partial list of results as we type. First, we'll use a function called `jsonify` to return our shows in the JSON format, a standard format that JavaScript can use.

```
@app.route("/search")
def search():
    shows = db.execute("SELECT * FROM shows WHERE title LIKE ?", "%" + request.args.get("q") + "%")
    return jsonify(shows)
```

  - Now we can submit a search query, and see that we get back a list of dictionaries:

- Then, our `index.html` template can convert this list to elements in the DOM:

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta name="viewport" content="initial-scale=1, width=device-width">
        <title>shows</title>
    </head>
    <body>

        <input autocomplete="off" autofocus placeholder="Query" type="search">

        <ul></ul>

```

```
        <script crossorigin="anonymous" integrity="sha256-9/aliU8dGd2tb6OSsuzixeV4y/faTqgFtohetphbbj0="
    src="https://code.jquery.com/jquery-3.5.1.min.js"></script>
        <script>

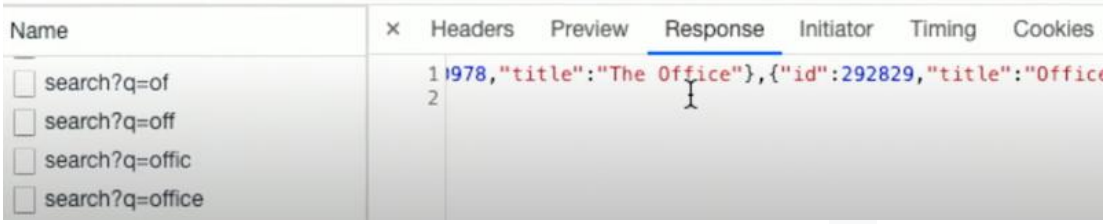
            let input = document.querySelector('input');
            input.addEventListener('keyup', function() {
               $.get('/search?q=' + input.value, function(shows) {
                 let html = '';
                 for (let id in shows)
                 {
                    let title = shows[id].title;
                    html += '<li>' + title + '</li>';
                 }

                 document.querySelector('ul').innerHTML = html;
               });
            });

        </script>

    </body>
</html>
```

- We'll use another library, JQuery, to make requests more easily.
- We'll listen to changes in the `input` element, and use `$.get`, which calls a JQuery library function to make a GET request with the input's value. Then, the response will be passed to an anonymous function as the variable `shows`, which will set the DOM with generated `<li>` elements based on the response's data.
- `$.get` is an **AJAX** call, which allows for JavaScript to make additional HTTP requests after the page has loaded, to get more data. If we open the Network tab again, we can indeed see that each key we pressed made another request, with a response:



- Since the network request might be slow, the anonymous function we pass to `$.get` is a **callback** function, which is only called after we get a response from the server. In the meantime, the browser can run other JavaScript code.

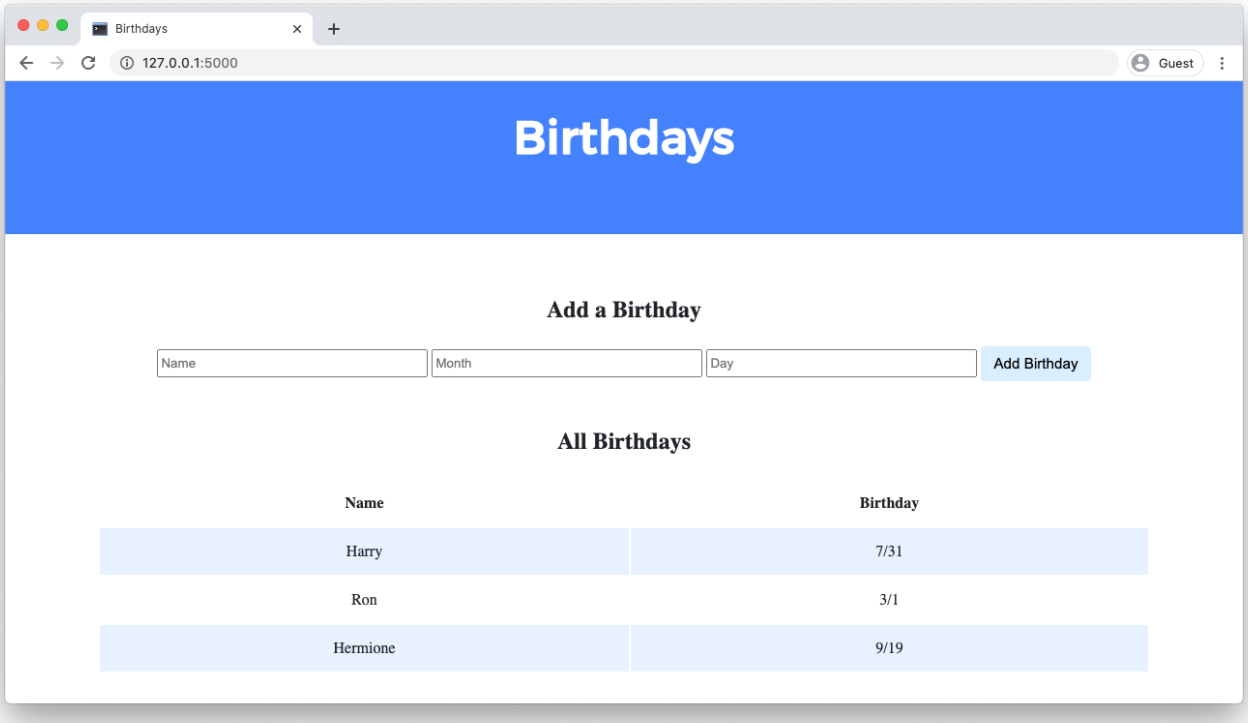
- That's it for today!

Lab 9: Birthdays

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Create a web application to keep track of friends' birthdays.



When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

Getting Started

Here's how to download this lab into your own CS50 IDE. Log into CS50 IDE 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 `~`).
- Execute `wget https://cdn.cs50.net/2020/fall/labs/9/lab9.zip` to download a (compressed) ZIP file with this problem's distribution.
- Execute `unzip lab9.zip` to uncompress that file.
- Execute `rm lab9.zip` followed by `yes` or `y` to delete that ZIP file.
- Execute `ls`. You should see a directory called `lab9`, which was inside of that ZIP file.
- Execute `cd lab9` to change into that directory.
- Execute `ls`. You should see an `application.py` file, a `birthdays.db` file, a `static` directory, and a `templates` directory.

Understanding

In `application.py`, you'll find the start of a Flask web application. The application has one route (`/`) that accepts both `POST` requests (after the `if`) and `GET` requests (after the `else`). Currently, when the `/` route is requested via `GET`, the `index.html` template is rendered. When the `/` route is requested via `POST`, the user is redirected back to `/` via `GET`.

`birthdays.db` is a SQLite database with one table, `birthdays`, that has four columns: `id`, `name`, `month`, and `day`. There are a few rows already in this table, though ultimately your web application will support the ability to insert rows into this table!

In the static directory is a styles.css file containing the CSS code for this web application. No need to edit this file, though you're welcome to if you'd like!

In the templates directory is an index.html file that will be rendered when the user views your web application.

Implementation Details

Complete the implementation of a web application to let users store and keep track of birthdays.

- When the / route is requested via GET, your web application should display, in a table, all of the people in your database along with their birthdays.
  - First, in application.py, add logic in your GET request handling to query the birthdays.db database for all birthdays. Pass all of that data to your index.html template.
  - Then, in index.html, add logic to render each birthday as a row in the table. Each row should have two columns: one column for the person's name and another column for the person's birthday.
- When the / route is requested via POST, your web application should add a new birthday to your database and then re-render the index page.
  - First, in index.html, add an HTML form. The form should let users type in a name, a birthday month, and a birthday day. Be sure the form submits to / (its "action") with a method of post.
  - Then, in application.py, add logic in your POST request handling to INSERT a new row into the birthdays table based on the data supplied by the user.

Optionally, you may also:

- Add the ability to delete and/or edit birthday entries.
- Add any additional features of your choosing!

Walkthrough

Hints

- Recall that you can call db.execute to execute SQL queries within application.py.
  - If you call db.execute to run a SELECT query, recall that the function will return to you a list of dictionaries, where each dictionary represents one row returned by your query.
- You'll likely find it helpful to pass in additional data to render_template() in your index function so that access birthday data inside of your index.html template.
- Recall that the tr tag can be used to create a table row and the td tag can be used to create a table data cell.
- Recall that, with Jinja, you can create a for loop inside your index.html file.
- In application.py, you can obtain the data POSTed by the user's form submission via request.form.get(field) where field is a string representing the name attribute of an input from your form.
  - For example, if in index.html, you had an <input name="foo" type="text">, you could use request.form.get("foo") in application.py to extract the user's input.

Not sure how to solve?

Testing

No check50 for this lab! But be sure to test your web application by adding some birthdays and ensuring that the data appears in your table as expected.

Run flask run in your terminal while in your lab9 directory to start a web server that serves your Flask application.

How to Submit

Execute the below to submit your work.

```
submit50 cs50/labs/2021/x/birthdays
```

Problem Set 9

Collaboration on problem sets is not permitted except to the extent that you may ask classmates and others for help so long as that help does not reduce to another doing your work for you, per the course's policy on **academic honesty**.

The staff conducts random audits of submissions to CS50x. Students found to be in violation of this policy will be removed from the course. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate permanently revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

What to Do

Be sure you have completed **Lab 9** before beginning this problem set.

1. Submit Finance

When to Do It

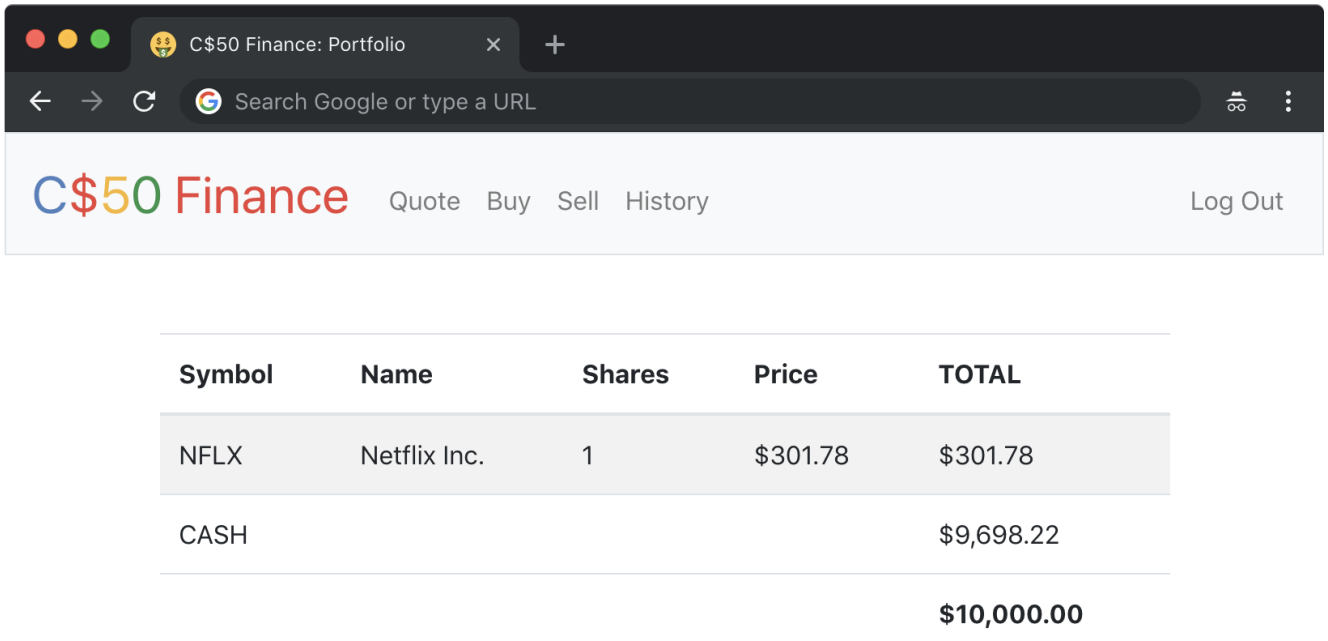
By Sat, Jan 1, 2022, 7:59 AM GMT+3.

C$50 Finance

This problem set's distribution code has recently changed. If you downloaded the distribution code prior to Fri, Apr 9, 2021, 7:00 AM GMT+3, run the following terminal commands in your `finance` directory to download the latest version of the distribution code.

```
$ rm helpers.py
$ wget https://cdn.cs50.net/2020/fall/psets/9/finance/finance/helpers.py
```

Implement a website via which users can "buy" and "sell" stocks, a la the below.



Background

If you're not quite sure what it means to buy and sell stocks (i.e., shares of a company), head here for a tutorial.

You're about to implement C$50 Finance, a web app via which you can manage portfolios of stocks. Not only will this tool allow you to check real stocks' actual prices and portfolios' values, it will also let you buy (okay, "buy") and sell (okay, "sell") stocks by querying IEX for stocks' prices.

Indeed, IEX lets you download stock quotes via their API (application programming interface) using URLs like https://cloud.iexapis.com/stable/stock/nflx/quote?token=API_KEY. Notice how Netflix's symbol (NFLX) is embedded in this URL; that's how IEX knows whose data to return. That link won't actually return any data because IEX requires you to use an API key (more about that in a bit), but if it did, you'd see a response in JSON (JavaScript Object Notation) format like this:


```
{
  "symbol": "NFLX",
  "companyName": "Netflix, Inc.",
  "primaryExchange": "NASDAQ",
  "calculationPrice": "close",
  "open": 317.49,
  "openTime": 1564752600327,
  "close": 318.83,
  "closeTime": 1564776000616,
  "high": 319.41,
  "low": 311.8,
  "latestPrice": 318.83,
  "latestSource": "Close",
  "latestTime": "August 2, 2019",
  "latestUpdate": 1564776000616,
  "latestVolume": 6232279,
  "iexRealtimePrice": null,
  "iexRealtimeSize": null,
  "iexLastUpdated": null,
  "delayedPrice": 318.83,
  "delayedPriceTime": 1564776000616,
  "extendedPrice": 319.37,
  "extendedChange": 0.54,
  "extendedChangePercent": 0.00169,
  "extendedPriceTime": 1564876784244,
  "previousClose": 319.5,
  "previousVolume": 6563156,
  "change": -0.67,
  "changePercent": -0.0021,
  "volume": 6232279,
  "iexMarketPercent": null,
  "iexVolume": null,
  "avgTotalVolume": 7998833,
  "iexBidPrice": null,
  "iexBidSize": null,
  "iexAskPrice": null,
  "iexAskSize": null,
  "marketCap": 139594933050,
  "peRatio": 120.77,
  "week52High": 386.79,
  "week52Low": 231.23,
  "ytdChange": 0.18907500000000002,
  "lastTradeTime": 1564776000616
}
```


Notice how, between the curly braces, there's a comma-separated list of key-value pairs, with a colon separating each key from its value.

Let's turn our attention now to this problem's distribution code!

Distribution

Downloading

```
$ wget http://cdn.cs50.net/2020/fall/psets/9/finance/finance.zip
$ unzip finance.zip
$ rm finance.zip
$ cd finance
$ ls
application.py  helpers.py       static/
finance.db      requirements.txt  templates/
```

Configuring

Before getting started on this assignment, we'll need to register for an API key in order to be able to query IEX's data. To do so, follow these steps:

- Visit iexcloud.io/cloud-login#/register/.
- Select the "Individual" account type, then enter your email address and a password, and click "Create account".
- Once registered, scroll down to "Get started for free" and click "Select Start" to choose the free plan.
- Once you've confirmed your account via a confirmation email, visit https://iexcloud.io/console/tokens.
- Copy the key that appears under the *Token* column (it should begin with `pk_`).
- In a terminal window within CS50 IDE, execute:

```
$ export API_KEY=value
```

where `value` is that (pasted) value, without any space immediately before or after the `=`. You also may wish to paste that value in a text document somewhere, in case you need it again later.

Running

. Start Flask's built-in web server (within `finance/`):

```
$ flask run
```

Visit the URL outputted by `flask` to see the distribution code in action. You won't be able to log in or register, though, just yet!

Via CS50's file browser, double-click `finance.db` in order to open it with phpLiteAdmin. Notice how `finance.db` comes with a table called `users`. Take a look at its structure (i.e., schema). Notice how, by default, new users will receive $10,000 in cash. But there aren't (yet!) any users (i.e., rows) therein to browse.

Here on out, if you'd prefer a command line, you're welcome to use `sqlite3` instead of phpLiteAdmin.

Understanding

application.py

Open up `application.py`. Atop the file are a bunch of imports, among them CS50's SQL module and a few helper functions. More on those soon.

After configuring Flask, notice how this file disables caching of responses (provided you're in debugging mode, which you are by default on CS50 IDE), lest you make a change to some file but your browser not notice. Notice next how it configures Jinja with a custom "filter," `usd`, a function (defined in `helpers.py`) that will make it easier to format values as US dollars (USD). It then further configures Flask to store sessions on the local filesystem (i.e., disk) as opposed to storing them inside of (digitally signed) cookies, which is Flask's default. The file then configures CS50's SQL module to use `finance.db`, a SQLite database whose contents we'll soon see!

Thereafter are a whole bunch of routes, only two of which are fully implemented: `login` and `logout`. Read through the implementation of `login` first. Notice how it uses `db.execute` (from CS50's library) to query `finance.db`. And notice how it uses `check_password_hash` to compare hashes of users' passwords. Finally, notice how `login` "remembers" that a user is logged in by storing his or her `user_id`, an INTEGER, in `session`. That way, any of this file's routes can check which user, if any, is logged in. Meanwhile, notice how `logout` simply clears `session`, effectively logging a user out.

Notice how most routes are "decorated" with @login_required (a function defined in helpers.py too). That decorator ensures that, if a user tries to visit any of those routes, he or she will first be redirected to login so as to log in.

Notice too how most routes support GET and POST. Even so, most of them (for now!) simply return an "apology," since they're not yet implemented.

helpers.py

Next take a look at helpers.py. Ah, there's the implementation of apology. Notice how it ultimately renders a template, apology.html. It also happens to define within itself another function, escape, that it simply uses to replace special characters in apologies. By defining escape inside of apology, we've scoped the former to the latter alone; no other functions will be able (or need) to call it.

Next in the file is login_required. No worries if this one's a bit cryptic, but if you've ever wondered how a function can return another function, here's an example!

Thereafter is lookup, a function that, given a symbol (e.g., NFLX), returns a stock quote for a company in the form of a dict with three keys: name, whose value is a str, the name of the company; price, whose value is a float; and symbol, whose value is a str, a canonicalized (uppercase) version of a stock's symbol, irrespective of how that symbol was capitalized when passed into lookup.

Last in the file is usd, a short function that simply formats a float as USD (e.g., 1234.56 is formatted as $1,234.56).

requirements.txt

Next take a quick look at requirements.txt. That file simply prescribes the packages on which this app will depend.

static/

Glance too at static/, inside of which is styles.css. That's where some initial CSS lives. You're welcome to alter it as you see fit.

templates/

Now look in templates/. In login.html is, essentially, just an HTML form, stylized with Bootstrap In apology.html, meanwhile, is a template for an apology. Recall that apology in helpers.py took two arguments: message, which was passed to render_template as the value of bottom, and, optionally, code, which was passed to render_template as the value of top. Notice in apology.html how those values are ultimately used! And here's why 0:-)

Last up is layout.html. It's a bit bigger than usual, but that's mostly because it comes with a fancy, mobile-friendly "navbar" (navigation bar), also based on Bootstrap. Notice how it defines a block, main, inside of which templates (including apology.html and login.html) shall go. It also includes support for Flask's message flashing so that you can relay messages from one route to another for the user to see.

Specification

register

Complete the implementation of register in such a way that it allows a user to register for an account via a form.

- Require that a user input a username, implemented as a text field whose name is username. Render an apology if the user's input is blank or the username already exists.
- Require that a user input a password, implemented as a text field whose name is password, and then that same password again, implemented as a text field whose name is confirmation. Render an apology if either input is blank or the passwords do not match.
- Submit the user's input via POST to /register.
- INSERT the new user into users, storing a hash of the user's password, not the password itself. Hash the user's password with generate_password_hash Odds are you'll want to create a new template (e.g., register.html) that's quite similar to login.html.
- Once the user is registered, you may either automatically log in the user or bring the user to a page where they can log in themselves.

Once you've implemented register correctly, you should be able to register for an account and log in (since login and logout already work)! And you should be able to see your rows via sqlite3 or phpLiteAdmin.

quote

Complete the implementation of `quote` in such a way that it allows a user to look up a stock's current price.

- Require that a user input a stock's symbol, implemented as a text field whose `name` is `symbol`.
- Submit the user's input via `POST` to `/quote`.
- Odds are you'll want to create two new templates (e.g., `quote.html` and `quoted.html`). When a user visits `/quote` via `GET`, render one of those templates, inside of which should be an HTML form that submits to `/quote` via `POST`. In response to a `POST`, `quote` can render that second template, embedding within it one or more values from `lookup`.

buy

Complete the implementation of `buy` in such a way that it enables a user to buy stocks.

- Require that a user input a stock's symbol, implemented as a text field whose `name` is `symbol`. Render an apology if the input is blank or the symbol does not exist (as per the return value of `lookup`).
- Require that a user input a number of shares, implemented as a text field whose `name` is `shares`. Render an apology if the input is not a positive integer.
- Submit the user's input via `POST` to `/buy`.
- Odds are you'll want to call `lookup` to look up a stock's current price.
- Odds are you'll want to `SELECT` how much cash the user currently has in `users`.
- Add one or more new tables to `finance.db` via which to keep track of the purchase. Store enough information so that you know who bought what at what price and when.
    - Use appropriate SQLite types.
    - Define `UNIQUE` indexes on any fields that should be unique.
    - Define (non-`UNIQUE`) indexes on any fields via which you will search (as via `SELECT` with `WHERE`).
- Render an apology, without completing a purchase, if the user cannot afford the number of shares at the current price.
- When a purchase is complete, redirect the user back to the `index` page.
- You don't need to worry about race conditions (or use transactions).

Once you've implemented `buy` correctly, you should be able to see users' purchases in your new table(s) via `sqlite3` or phpLiteAdmin.

index

Complete the implementation of `index` in such a way that it displays an HTML table summarizing, for the user currently logged in, which stocks the user owns, the numbers of shares owned, the current price of each stock, and the total value of each holding (i.e., shares times price). Also display the user's current cash balance along with a grand total (i.e., stocks' total value plus cash).

- Odds are you'll want to execute multiple `SELECT`s. Depending on how you implement your table(s), you might find `GROUP BY` `HAVING` `SUM` and/or `WHERE` of interest.
- Odds are you'll want to call `lookup` for each stock.

sell

Complete the implementation of `sell` in such a way that it enables a user to sell shares of a stock (that he or she owns).

- Require that a user input a stock's symbol, implemented as a `select` menu whose `name` is `symbol`. Render an apology if the user fails to select a stock or if (somehow, once submitted) the user does not own any shares of that stock.
- Require that a user input a number of shares, implemented as a text field whose `name` is `shares`. Render an apology if the input is not a positive integer or if the user does not own that many shares of the stock.
- Submit the user's input via `POST` to `/sell`.
- When a sale is complete, redirect the user back to the `index` page.
- You don't need to worry about race conditions (or use transactions).

history

Complete the implementation of `history` in such a way that it displays an HTML table summarizing all of a user's transactions ever, listing row by row each and every buy and every sell.

- For each row, make clear whether a stock was bought or sold and include the stock's symbol, the (purchase or sale) price, the number of shares bought or sold, and the date and time at which the transaction occurred.

- You might need to alter the table you created for `buy` or supplement it with an additional table. Try to minimize redundancies.

personal touch

Implement at least one personal touch of your choice:

- Allow users to change their passwords.
- Allow users to add additional cash to their account.
- Allow users to buy more shares or sell shares of stocks they already own via `index` itself, without having to type stocks' symbols manually.
- Require users' passwords to have some number of letters, numbers, and/or symbols.
- Implement some other feature of comparable scope.

Walkthrough

Testing

To test your code with `check50`, execute the below.

```
$ check50 cs50/problems/2021/x/finance
```

Be aware that `check50` will test your entire program as a whole. If you run it **before** completing all required functions, it may report errors on functions that are actually correct but depend on other functions.

Be sure to test your web app manually too, as by

- inputting alphabetical strings into forms when only numbers are expected,
- inputting zero or negative numbers into forms when only positive numbers are expected,
- inputting floating-point values into forms when only integers are expected,
- trying to spend more cash than a user has,
- trying to sell more shares than a user has,
- inputting an invalid stock symbol, and
- including potentially dangerous characters like `'` and `;` in SQL queries.

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

```
style50 *.py
```

Staff's Solution

You're welcome to stylize your own app differently, but here's what the staff's solution looks like!

https://finance.cs50.net/

Feel free to register for an account and play around. Do **not** use a password that you use on other sites.

It is **reasonable** to look at the staff's HTML and CSS.

Hints

- To format a value as a US dollar value (with cents listed to two decimal places), you can use the `usd` filter in your Jinja templates (printing values as `{{ value | usd }}` instead of `{{ value }}`.
- Within `cs50.SQL` is an `execute` method whose first argument should be a `str` of SQL. If that `str` contains question mark parameters to which values should be bound, those values can be provided as additional named parameters to `execute`. See the implementation of `login` for one such example. The return value of `execute` is as follows:
  - If `str` is a `SELECT`, then `execute` returns a `list` of zero or more `dict` objects, inside of which are keys and values representing a table's fields and cells, respectively.
  - If `str` is an `INSERT`, and the table into which data was inserted contains an autoincrementing `PRIMARY KEY`, then `execute` returns the value of the newly inserted row's primary key.
  - If `str` is a `DELETE` or an `UPDATE`, then `execute` returns the number of rows deleted or updated by `str`.
- Recall that `cs50.SQL` will log to your terminal window any queries that you execute via `execute` (so that you can confirm whether they're as intended).

- Be sure to use question mark-bound parameters (i.e., a paramstyle of named) when calling CS50's execute method, a la WHERE ?. Do **not** use f-strings, format or + (i.e., concatenation), lest you risk a SQL injection attack.
- If (and only if) already comfortable with SQL, you're welcome to use SQLAlchemy Core or Flask-SQLAlchemy (i.e., SQLAlchemy ORM) instead of cs50.SQL.
- You're welcome to add additional static files to static/.
- Odds are you'll want to consult Jinja's documentation when implementing your templates.
- It is **reasonable** to ask others to try out (and try to trigger errors in) your site.
- You're welcome to alter the aesthetics of the sites, as via
    - https://bootswatch.com/,
    - https://getbootstrap.com/docs/4.1/content/,
    - https://getbootstrap.com/docs/4.1/components/, and/or
    - https://memegen.link/.
- You may find Flask's documentation and Jinja's documentation helpful!

FAQs

ImportError: No module named 'application'

By default, flask looks for a file called application.py in your current working directory (because we've configured the value of FLASK_APP, an environment variable, to be application.py). If seeing this error, odds are you've run flask in the wrong directory!

OSError: [Errno 98] Address already in use

If, upon running flask, you see this error, odds are you (still) have flask running in another tab. Be sure to kill that other process, as with ctrl-c, before starting flask again. If you haven't any such other tab, execute fuser -k 8080/tcp to kill any processes that are (still) listening on TCP port 8080.

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/2021/x/finance
```

Week 10 Ethics

Lecture 10

The End

- Big thanks to the American Repertory Theater for hosting lectures this term, providing amazing props, lighting, and sounds to our stage.
- CS50's team, too, has been making everything possible off-stage, including our teaching fellows and course assistants, at both Harvard and Yale.
- And even David makes mistakes and can be unsure about answers for some questions, so rest assured that learning will continue well beyond this course.
- Don't forget that, ultimately:

  > what ultimately matters in this course is not so much where you end up relative to your classmates but where you end up relative to yourself when you began

- We've learned some first principles:
  - computational thinking
  - using algorithms to solve problems, given some inputs from which to produce outputs
  - axes of correctness, design, and style to evaluate our code
  - abstraction, using layering to simplify problems, as with functions in code
  - precision, as by considering all the possible edge cases in our instructions
- With these basic building blocks, we can learn to use tools in the future, beyond C and Python, to solve even more problems.
- We ask a volunteer to give a series of instructions for how to draw a cube and a snowman, and since everyone else interpreted each instruction slightly differently, the final drawings all ended up very different.

Ethics

- We might think of ethics as whether we *should* do something, or *how* we go about doing it, even when we have the ability to.
- For example, we can now use code to send lots of emails, creating more spam. We can collect passwords for our website with a form, and if a user uses that same password for another site, we end up with access to their account unless we store the passwords securely.
- JavaScript, too, can be used to log users' actions on our sites, like items they've added to their cart. But logging all actions, over time, can cause worry for users' privacy.
- Before Facebook, the website, there was another website, Facemash, where code was used to scrape, or download, images of Harvard students and use them without prior permission.
- With some colleagues from the Department of Philosophy, Meica Magnani and Susan Kennedy, we discuss some frameworks for making decisions more rigorously.
- The Embedded EthiCS program at Harvard integrates tools for ethical reasoning into computer science courses, to help ensure that future computer scientists will be creating and using technology ethically.
- The transcript for this section of lecture, as well as readings for the related lab, have been posted separately.

Looking Forward

- Even without additional courses in computer science, we hope that you're now equipped to use technology to solve problems in your own domain.
- When we face new problems, we can rely on one or more of these skills:
  - asking questions
  - finding answers
  - reading documentation
  - teaching yourself new languages
- CS50 IDE, too, can be used for future projects, but it comes with tools for the course.
- More industry-standard tools, for our own Mac or PC, include a terminal and other command-line tools:

    - https://developer.apple.com/xcode/
    - https://docs.microsoft.com/en-us/windows/wsl/about
    - …
- Brian has a workshop on Git, a version-control software used to manage different versions of code and enable collaboration with others.
- One of the most popular IDEs, VS Code is open-source and freely available, with a text editor at its core and lots of other features that can be added. There are many alternatives as well.
- Web hosts include:
    - https://pages.github.com/
    - https://www.netlify.com/
    - …
- Web app hosts include:
    - https://www.heroku.com/platform
    - https://aws.amazon.com/education/awseducate/
    - https://azure.microsoft.com/en-us/free/students/
    - https://edu.google.com/programs/students/
    - …
- And news sources for technology and programming include:
    - https://www.reddit.com/r/learnprogramming/
    - https://www.reddit.com/r/programming/
    - https://stackoverflow.com/
    - https://serverfault.com/
    - https://techcrunch.com/
    - https://news.ycombinator.com/
    - …
- CS50 has many communities as well:
    - https://discord.gg/cs50
    - https://www.facebook.com/groups/cs50
    - https://www.facebook.com/cs50
    - https://gitter.im/cs50/x
    - https://github.com/cs50
    - https://www.instagram.com/cs50/
    - https://www.linkedin.com/groups/7437240/
    - https://www.linkedin.com/school/CS50/
    - https://www.quora.com/topic/CS50
    - https://www.reddit.com/r/cs50
    - https://cs50x.slack.com/
    - https://www.snapchat.com/add/cs50
    - https://soundcloud.com/cs50
    - http://cs50.stackexchange.com/
    - https://twitter.com/cs50
    - http://www.youtube.com/cs50

Quiz Show

- We host a quiz show with the audience, with the following questions:
    - What are the steps for compiling source code into machine code?
        - Preprocessing, compiling, assembling, linking
    - What is the runtime of binary search?
        - $O(\log n)$
    - Which of these animals was the first to be mentioned in a CS50 lecture?
        - Cat
    - Every time you `malloc` memory, you should also be sure to…
        - `free`
    - What is a race condition?
        - When two things happen at the same time and produce an unexpected result
    - Does zooming in on a photo let you “enhance” it to generate more detail?
        - No, a photo only has a certain amount of detail
    - Which of the following is not a characteristic of a good hash function?
        - Randomness
    - What does FIFO stand for?
        - First in, first out

    - Which of the following would represent pink using RGB values?
        - `#ffd0e0`
    - In C, which of the following lines of code allocates enough memory for a copy of the string `s`?
        - `malloc(strlen(s) + 1)`
    - How should you organize your clothes to be cool?
        - queue
    - What is a segmentation fault?
        - When a program tries to access memory that it shouldn't
    - Which of the following types of overflow can result from recursion without a base case?
        - stack overflow
    - In the town of Fiftyville, what were the names of the three people who witnessed the rubber duck robbery?
        - Ruth, Eugene, and Raymond
    - Which of these command-line programs check your code for memory leaks?
        - `valgrind`
    - Which of the following exists in C, but not Python?
        - do-while loops
    - What HTTP request method should you use when sending private information, like a password?
        - POST
    - What data structure allows for constant-time lookup for words in a dictionary?
        - trie
    - What is a cookie?
        - both of the above
    - What's your comfort level now?
        - I'm among those more comfortable
- Thanks for joining us at CS50!

Lab 10: Ethics

You are welcome to collaborate with one or two classmates on this lab, though it is expected that every student in any such group contribute equally to the lab.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use check50 and submit50! See **cs50.ly/github** for instructions if you haven't already!

Getting Started

1. Watch Week 10's lecture if you haven't already.
2. Read How to Fix Fake News, by Regina Rini.
3. Optionally read Escape the echo chamber, by C Thi Nguyen.
4. Optionally read this excerpt from Democracy and the Digital Public Sphere, by Joshua Cohen and Archon Fung.

What To Do

1. Open this lab in Google Docs.
2. Make a copy of the quiz in your own Google account by choosing **File → Make a Copy**.
3. Answer each of the questions in the quiz document by filling in the blanks marked **TODO**.

When to Do It

By Sat, Jan 1, 2022, 7:59 AM GMT+3.

How to Submit

1. Download your completed lab as a PDF by choosing **File → Download → PDF Document**, and save it to your computer.
2. Go to https://submit.cs50.io/upload/cs50/labs/2021/x/ethics.
3. Click "Choose File" and choose your .pdf file. Click **Submit**.

That's it! Once your submission uploads, you should be redirected to your submission page.

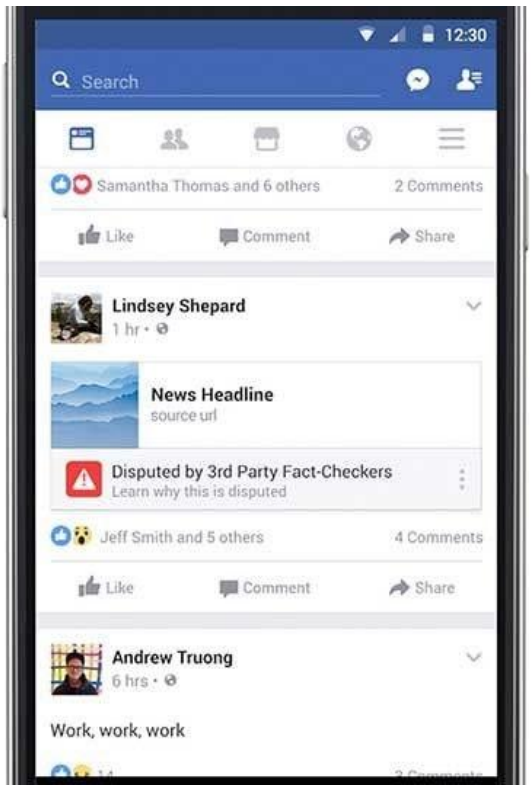
Facebook and Fake News

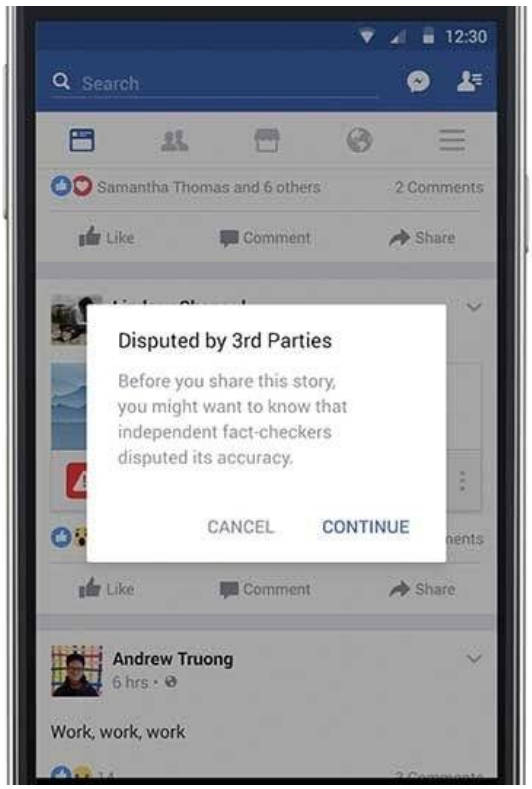
Consider each of the below proposals, and fill in each of the below TODOs.

Proposals

Proposal 1

Facebook relies on an algorithm as well as individual users' reports to identify content that is potentially "fake news." Once the content has been identified, it is sent to third party fact-checkers for verification. If the content is verified as fake news, it is publicly flagged with a warning that the content is disputed by fact-checkers.





Proposal 2

However, a designer at Facebook believes that there needs to be a different approach to content regulation. Their proposal is that content deemed problematic by third-party fact-checkers should be prevented from being shared on the platform altogether.

Questions

1. Which form of content regulation, Proposal 1 or Proposal 2, do you feel is better? Why?

a. TODO

2. Which form of content regulation, Proposal 1 or Proposal 2, do you think best preserves or promotes the five rights and opportunities necessary for a democratic public sphere?

. Which proposal best preserves or promotes Rights? Why?

i. TODO

a. Which proposal best preserves or promotes Opportunity for Expression? Why?

. TODO

b. Which proposal best preserves or promotes Access? Why?

. TODO

c. Which proposal best preserves or promotes Diversity? Why?

. TODO

d. Which proposal best preserves or promotes Communicative Power? Why?

. TODO

Final Project

The climax of this course is its final project. The final project is your opportunity to take your newfound savvy with programming out for a spin and develop your very own piece of software. So long as your project draws upon this course's lessons, the nature of your project is entirely up to you. You may implement your project in any language(s). You are welcome to utilize infrastructure other than the CS50 IDE. All that we ask is that you build something of interest to you, that you solve an actual problem, that you impact your community, or that you change the world. Strive to create something that outlives this course.

Inasmuch as software development is rarely a one-person effort, you are allowed an opportunity to collaborate with one or two classmates for this final project. Needless to say, it is expected that every student in any such group contribute equally to the design and implementation of that group's project. Moreover, it is expected that the scope of a two- or three-person group's project be, respectively, twice or thrice that of a typical one-person project. A one-person project, mind you, should entail more time and effort than is required by each of the course's problem sets.

Note that CS50's staff audits submissions to CS50x including this final project. Students found to be in violation of **the Academic Honesty policy** will be removed from the course and deemed ineligible for a certificate. Students who have already completed CS50x, if found to be in violation, will have their CS50 Certificate (and edX Certificate, if applicable) revoked.

GitHub now requires that you use SSH or a personal access token instead of a password to log in, but you can still use `check50` and `submit50`! See **cs50.ly/github** for instructions if you haven't already!

Ideas

- a web-based application using JavaScript, Python, and SQL
- an iOS app using Swift
- a game using Lua with LÖVE
- an Android app using Java
- a Chrome extension using JavaScript
- a command-line program using C
- a hardware-based application for which you program some device
- …

Getting Started

Creating an entire project may seem daunting. Here are some questions that you should think about as you start:

- What will your software do? What features will it have? How will it be executed?
- What new skills will you need to acquire? What topics will you need to research?
- If working with one or two classmates, who will do what?
- In the world of software, most everything takes longer to implement than you expect. And so it's not uncommon to accomplish less in a fixed amount of time than you hope. What might you consider to be a good outcome for your project? A better outcome? The best outcome?

Consider making goal milestones to keep you on track.

If using the CS50 IDE, create a directory called `~/project` to store your project source code and other files. You are welcome to develop your project outside of the CS50 IDE.

How to Submit

You must complete all three steps!

Step 1 of 3

Create a short video (that's no more than 3 minutes in length) in which you present your project to the world, as with slides, screenshots, voiceover, and/or live action. Your video should somehow include your project's title, your name, your city and country, and any other details that you'd like to convey to viewers. See https://www.howtogeek.com/205742/how-to-record-your-windows-mac-linux-android-or-ios-screen/ for tips on how to make a "screencast," though you're welcome to use an actual camera. Upload your video to YouTube (or, if blocked in your country, a similar site) and take note of its URL; it's fine to flag it as "unlisted," but don't flag it as "private."

Submit this form!

Step 2 of 3

Create a README.md text file in your ~/project folder that explains your project. This file should include your Project Title, the URL of your video (created in step 1 above) and a description of your project. You may use the below as a template.

```
# YOUR PROJECT TITLE
#### Video Demo:  <URL HERE>
#### Description:
TODO
```

If unfamiliar with Markdown syntax, you might find GitHub's Basic Writing and Formatting Syntax helpful.

Your README.md file should be minimally multiple paragraphs in length, and should explain what your project is, what each of the files you wrote for the project contains and does, and if you debated certain design choices, explaining why you made them. Ensure you allocate sufficient time and energy to writing a README.md that you are proud of and that documents your project thoroughly. Be proud of it!

Execute the submit50 command below from within your ~/project directory (or from whichever directory contains README.md file and your project's code, which must also be submitted), 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/2021/x/project
```

Trouble Submitting?

If you encounter issues because your project is too large, try to ZIP all of the contents of that directory (except for README.md) and then submit that instead. If still too large, try removing certain configuration files, reducing the size of your submission below 100MB, or try to upload directly using GitHub's web interface by visiting github.com/me50/USERNAME (where USERNAME is your own GitHub username) and manually dragging and dropping folders, ensuring that when uploading you are doing so to your cs50/problems/2021/x/project branch, otherwise the system will not be able to check it!

Step 3 of 3

Be sure to visit your gradebook at cs50.me/cs50x a few minutes after you submit. It's only by loading your Gradebook that the system can check to see whether you have completed the course, and that is also what triggers the (instant) generation of your free CS50 Certificate and the (within 30 days) generation of the Verified Certificate from edX, if you've completed all of the other assignments. Be sure to claim your free certificate (by following the link at the top of your gradebook) before 1 January 2022.

That's it! Your project should be graded within a few minutes. If you don't see any results in your gradebook, best to resubmit (running the above submit50 command) with only your README.md file this time. No need to resubmit your form.

This was CS50x!