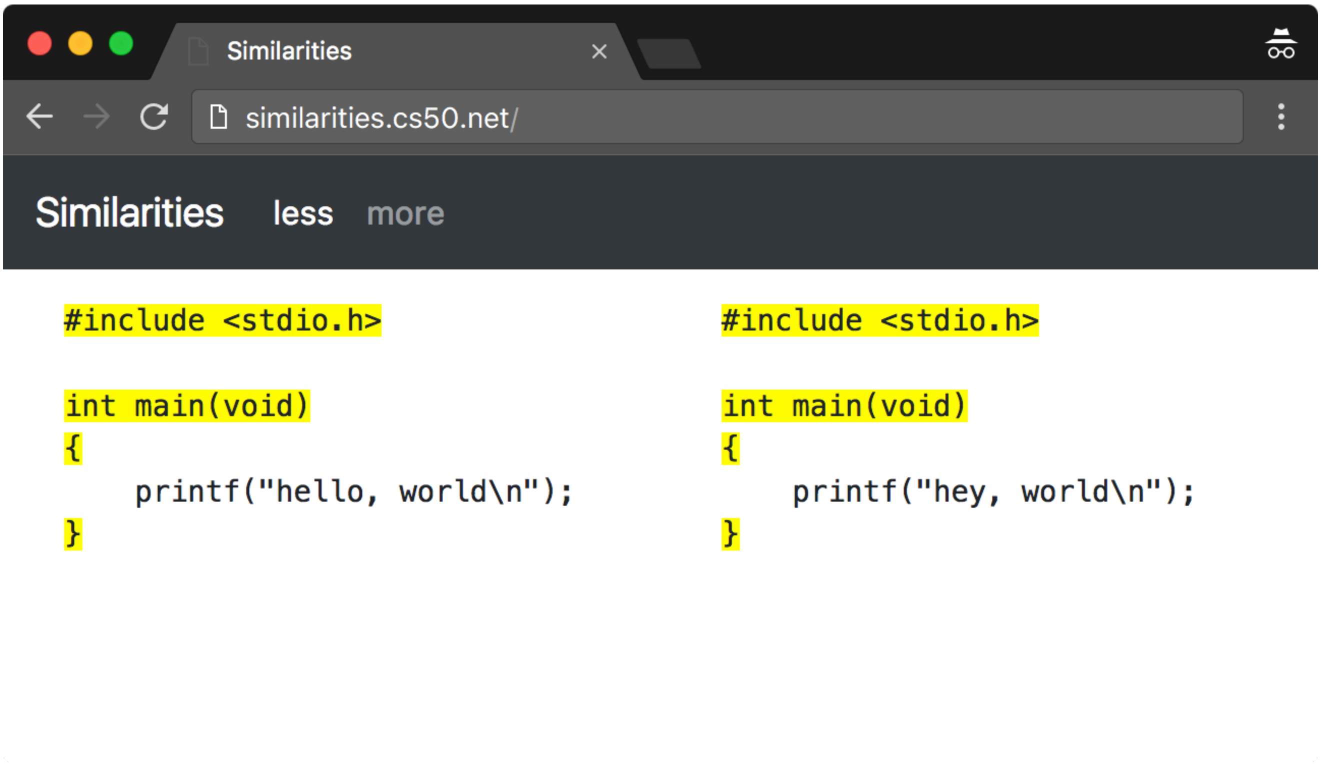
# Table of Contents

# Similarities

## tl;dr

1. Implement a program that compares two files for similarities.

2. Implement a website that highlights similarities across files, a la the below.



```
#include <stdio.h>

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

```
#include <stdio.h>

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

# Background

Determining whether two files are identical is (relatively!) trivial: iterate over the characters in each, checking whether each and every one is identical. But determining whether two files are similar is non-trivial. After all, what does it mean to be similar? Perhaps the files have lines in common. Perhaps the files have sentences in common. Perhaps the files have only substrings in common.

Suffice it to say, the challenge ahead is to determine if two files are similar!

---

# Getting Started

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

1. Execute `cd` to ensure that you're in `~/` (aka your home folder).
2. Execute `mkdir pset7` to make (i.e., create) a directory called `pset7` in your `~/` directory, if you haven't already done so.
3. Execute `cd pset7` to change into (i.e., open) that directory.
4. Execute `wget https://cdn.cs50.net/2018/fall/psets/7/similarities/similarities.zip (https://cdn.cs50.net/2018/fall/psets/7/similarities/similarities.zip)` to download a (compressed) ZIP file with this problem's distribution.
5. Execute `unzip similarities.zip` to uncompress that file.
6. Execute `rm similarities.zip` followed by `yes` or `y` to delete that ZIP file.
7. Execute `ls`. You should see a directory called `similarities`, which was inside of that ZIP file.
8. Execute `cd similarities` to change into that directory.
9. Execute `ls`. You should see this problem's distribution inside: `` ` application.py  compare*  helpers.py  inputs/  requirements.txt  static/  templates/ ``

`

## Understanding

### `inputs/`

Inside of this directory are a whole bunch of sample inputs that you can ultimately compare, among others, for similarities!

### `compare`

Open up `compare`. Suffice it to say that file’s name doesn’t end in `.py`, even though the file contains a program written in Python. But that’s okay! Notice the "shebang" atop the file:

```
#!/usr/bin/env python3
```

That line tells a computer to interpret (i.e., run) the program using `python3` (aka `python` on CS50 IDE), an interpreter that understands Python 3.

Notice how the file defines a function called `main` and calls that function toward the bottom of the file. Defining `main` isn’t strictly necessary in Python, but it is necessary to define functions before you call them. Accordingly, because `main` calls a function called `positive`, and because we wanted to keep the "main" part of this program atop the file, it made sense to implement `main` as a function as well. That way, `main` doesn’t get called until the bottom of the file (after `positive` has been implemented), even though `main` is implemented atop the file.

No need to understand each of the lines in `compare`, but notice, per its comments, what it does overall: it parses its command-line arguments, reads two files into variables as strings, and compares those strings, and then prints a list of similarities. The strings themselves are compared in one of three ways, as specified by a command-line argument: line by line, sentence by sentence, or substring by substring.

## `helpers.py`

Open up `helpers.py`. Ah, the familiar `TODO`. Declared in this file are three functions, each of which is meant to implement a different algorithm: `lines`, `sentences`, and `substrings`. At the moment, each of them returns an empty list. But not for long!

## `application.py`

Open up `application.py`. This file implements a web application that, ultimately, will allow you to run any of those three algorithms on any two text files. No need to understand the entirety of this file, particularly `highlight` and `errorhandler`. But know that `highlight`, given a string, `s`, and a list of other strings, `strings`, highlights (by wrapping them in HTML `span` tags) all instances of the former in the latter. And `errorhandler` ensures that any HTTP errors are displayed on a page of their own.

But do read through `index` and `compare`, the latter of which handles form submissions.

## `templates/layout.html`

Open up `templates/layout.html`. In this file is a template for the web application's overall layout. Odds are you'll recognize a few of the HTML tags therein and notice a few new ones. Notice, in particular, how the template uses Bootstrap, a popular library. In fact, we based this template on their own starter template (http://getbootstrap.com/docs/4.0/getting-started/introduction/).

## `templates/index.html`

Open up `templates/index.html`. Ah, the final `TODO`. Notice how this template "extends" `layout.html`, which is to say that `layout.html` is the "mold" from which `index.html` itself will be made. The `block` defined in `index.html` will effectively get plugged into the placeholder for `block` in `layout.html`.

Ultimately, this file will contain the form via which users will be able to upload two files to your web application for comparison via one of your three algorithms.

## `templates/compare.html`

Open up `templates/compare.html`. We took the liberty of implementing this file for you. Thanks to its use of some CSS (particularly a class called `col-6`), it ensures that users' files, once uploaded and highlighted, will be displayed side by side.

## `templates/error.html`

Open up `templates/error.html`. In this file is a template with which any HTTP errors will be displayed. It happens to use Bootstrap’s Jumbotron (https://getbootstrap.com/docs/4.0/components/jumbotron/) feature.

## `static/styles.css`

Open up `static/styles.css`. In this file are some CSS properties that collectively implement your web application’s user interface. Essentially, they modify some of Bootstrap’s own defaults.

### `requirements.txt`

Open up `requirements.txt` (without changing it, though you can later if you'd like). This file specifies the libraries, one per line, on which all of this functionality depends.

---

# Specification

## `helpers.py`

### `lines`

Implement `lines` in such a way that, given two strings, `a` and `b`, it returns a `list` of the lines that are, identically, in both `a` and `b`. The `list` should not contain any duplicates. Assume that lines in `a` and `b` will be be separated by `\n`, but the strings in the returned `list` should not end in `\n`. If both `a` and `b` contain one or more blank lines (i.e., a `\n` immediately preceded by no other characters), the returned `list` should include an empty string (i.e., `""`).

### `sentences`

Implement `sentences` in such a way that, given two strings, `a` and `b`, it returns a `list` of the *unique* English sentences that are, identically, present in both `a` and `b`. The `list` should not contain any duplicates. Use `sent_tokenize` from the Natural Language Toolkit to "tokenize" (i.e., separate) each string into a `list` of sentences. It can be imported with:

```
from nltk.tokenize import sent_tokenize
```

Per its documentation (http://www.nltk.org/api/nltk.tokenize.html#nltk.tokenize.sent_tokenize), `sent_tokenize`, given a `str` as input, returns a `list` of English sentences therein. It assumes that its input is indeed English text (and not, e.g., code, which might coincidentally have periods too).

### `substrings`

Implement `substrings` in such a way that, given two strings, `a` and `b`, and an integer, `n`, it returns a `list` of all substrings of length `n` that are, identically, present in both `a` and `b`. The `list` should not contain any duplicates.

Recall that a substring of length `n` of some string is just a sequence of `n` characters from that string. For instance, if `n` is `2` and the string is `Yale`, there are three possible substrings of length `2`: `Ya`, `al`, and `le`. Meanwhile, if `n` is `1` and the string is `Harvard`, there are seven possible substrings of length `1`: `H`, `a`, `r`, `v`, `a`, `r`, and `d`. But once we eliminate duplicates, there are only five unique substrings: `H`, `a`, `r`, `v`, and `d`.

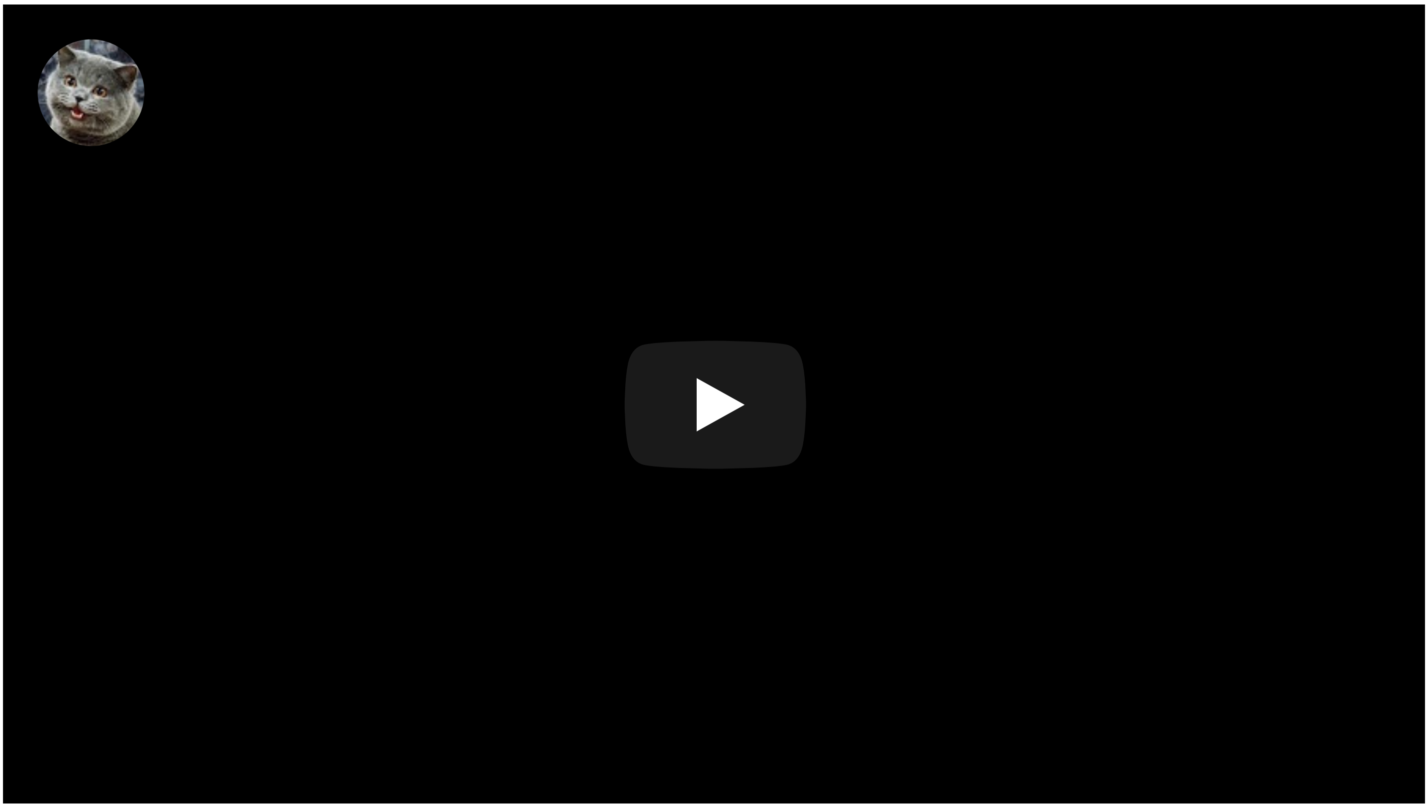
## `templates/index.html`

Implement `templates/index.html` in such a way that it contains an HTML form via which a user can submit:

- a file called `file1`
- a file called `file2`
- a value of `lines`, `sentences`, or `substrings` for an `input` called `algorithm`
- a number called `length`

You're welcome to look at the HTML of the staff's solution as needed, but do try to figure out the right syntax on your own first, as via https://www.google.com/search?q=html+forms! (https://www.google.com/search?q=html+forms!)

# Walkthroughs



# Testing

To test your implementation of `lines`, `sentences`, and/or `substrings` via the command line, execute `compare` as follows, where `FILE1` and `FILE2` are any two text files (e.g., those in `inputs/`):

```
./compare --lines FILE1 FILE2
./compare --sentences FILE1 FILE2
./compare --substrings 1 FILE1 FILE2
./compare --substrings 2 FILE1 FILE2
...
```

To test your implementations via a web app, execute

```
flask run
```

and then visit the outputted URL.

Be sure to test your implementation with the files in `inputs/` (which are also available via a browser (https://cdn.cs50.net/2018/fall/psets/7/similarities/similarities/inputs/)) as well as with some files of your own!

## `check50`

```
check50 cs50/problems/2019/x/similarities
```

## `style50`

```
style50 helpers.py
```

# Staff's Solution

## CLI

```
~cs50/2019/x/pset7/compare
```

## Web

http://similarities.cs50.net/less (http://similarities.cs50.net/less)

# How to Submit

Execute the below from within your `~/pset7/similarities` directory, 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/2019/x/similarities
```