Computer Science 352 Spring 2022
Programming Assignment 1
Due 1/28/2022 by 7pm

This first PA for CS 352 has three parts. The first will require you to research, experiment with, and write several bash commands. The second will be a very simple C program. The Third will require you to write python code to test an executable (compiled from C) using the **os** and **subprocess** python modules. For the project, you should end up with the following directory structure:

```
pa1
├── pythagorean
│   └── pythagorean.c
├── sbt
│   └── sbt.py
└── shell
    └── commands.txt
```

Part A - Bash Commands

You should write a bash command for each of the descriptions shown in this section, and put each in the file **commands.txt**. You should format it as follows:

```
(1)
first_command
(2)
second_command
. . . .
```

For part A, you may only use the following commands:

cd ls mkdir touch cat grep cut uniq wc cal find sort head tail tr

You may (and should) also use pipes, redirection, and semicolons. If you are unsure how any of these work, use the man pages. Each command should be a one-liner.

1. A bash command to show the relative paths for every **.py**, **.c**, and **.java** file within the current working directory or any of its subtrees. The python files should display first, then the c files, then the java files. For example, say that you are currently in a directory named **test** with the following structure within it:

```
└── test
    ├── code
    │   ├── something.ml
    │   └── something.py
    ├── other
    │   ├── calculate.c
    │   └── compute.java
    └── scripting.py
```

If you were to run your command, you should see the following result:

```
$ your_command
./scripting.py
./code/something.py
./other/calculate.c
./other/compute.java
```

2. A bash command to search through words within the standard **`/usr/share/dict/words`** unix file. The command should determine how many unique "words" are in the file if only the first four characters of each word on each line is used. For example, lets say that **`/usr/share/dict/words`** had only these four words in it:

   ```
   antelope
   tarantula
   antenna
   taraplaza
   ```

   If we were to run the command, we should get a result of 2.
   If we only include the first 4 letters, there are only two unique words (**ante** and **tara**).

   ```
   $ your_command
         2
   ```

3. A bash command that displays the numbers of the julian-calendar days for 2022 for Sunday. Recall that in a julian calendar, days are numbered 1-365, rather than resetting back to 1 each month. The results should be sorted numerically. In total you should get 52 numbers, but since that is long I will show you a truncated version of the output you should expect:

   ```
   $ your_command
   2
   9
   16
   23
   . . . .
   338
   345
   352
   359
   ```

4. A command to get a list of the unique creation months of the files in the current working directory. No two month names should appear twice, even if there are multiple files with that creation month. You can use **ls -o** to show a list of the files with the creation months (and other things) included. Thus, for example, say we ran an ls -o within a directory and got these results:

```
drwx------   20 userA       52 Dec  9 21:33 somefile.c
drwx------   23 benjamin    53 Dec  9 12:53 runthis.py
drwx------   96 userB      161 Jan  3 16:01 something.js
drwx------   14 userA       25 Dec  7 18:42 numberzz.csv
```

If we were to run our command, we should get:

```
$ your_command

Dec
Jan
```

Part B - Super Simple C Program

In part B, you are tasked with writing a very simple C program. You will be writing much larger and more complex C programs later on in the course, but for now we are starting with something very simple. You should write a C program that uses the pythagorean theorem to determine the length of the hypotenuse of a right triangle, given the length of the other two sides. When using the pythagorean theorem, we generally refer to the hypotenuse as side C, and the other two sides as sides A and B.

The program should read in the lengths of sides A and B as integer values. It should determine the length of side C and store it as a double. It should then print out this value and then exit with a status code of 0. You should write this code in a file named **`pythagorean.c`** located in the directory shown in this PAs overview.

Remember, both for this PA and for every future PA, you should compile using **`gcc`** and with the flags **`-Wall -Werror -std=c11`.** Two examples are shown below, with input values typed by the user highlighted in **red**. Your program should match the input and output formatting exactly.

```
$ gcc -Wall -Werror -std=c11 pythagorean.c
$ ./a.out
Enter side A length:
3
Enter side B length:
4
Length of side C: 5.00
$ ./a.out
Enter side A length:
125
Enter side B length:
81
Length of side C: 148.95
```

Part C - An Executable Tester in Python

In part C (the final part of the assignment) you will be writing a program in python for testing if executables are compiling and running correctly. For this particular PA, you should use it to test out the pythagorean.c program. In future assignments, you can re-use this program to test out other PAs. You are also welcome to add additional features to it in the future, though for the purpose of this PA you should stick to only the features described in the spec.

The program you will write should be named **sbt.py** (Super Basic Tester) and should be located in the sbt directory within the pa1 directory, as indicated in the assignment summary.

This program should accept two input values via standard input. The first will be the path / name of the .c file that it will be testing. The second will be the path / name to a directory containing test cases for the program. Your program should expect that the test directory will have the following structure:

```
─── test_directory
    ├── test-A
    │   ├── input.txt
    │   └── output.txt
    ├── test-B
    │   ├── input.txt
    │   └── output.txt
    . . . .
```

The test directory should contain 0 or more directories within it, with names that correspond to what the “name” of that particular test. Each of these directories should have exactly two files within it named **`input.txt`** and **`output.txt`**. **`input.txt`** should contain whatever content you want to be sent as the standard input to the program being tested for this case. **`output.txt`** should contain the exact test that you expect your program to produce, given the corresponding standard input from **`input.txt`**.

The outline of what this python program should do is as follows:

1. Ask the user for the two input values (.c file name and test directory)
2. Run a command to compile the .c program (`gcc -Wall -Werror -stc=c11 FILE.c`)
3. If the compilation failed for any reason, print "**`failed to compile your code`**" and then exit.
4. If it worked, loop through each of the directories within the test directory folder
5. For each one:
    a. Run the executable and give it the standard input text from input.txt
    b. Store / save what it prints to standard output
    c. Compare the result with the contents of the output.txt file
    d. If the same, print **`"#### Test: DIR_NAME passed! ####"`**
    e. If different, print **`"#### Test: DIR_NAME failed! ####"`** and then print out a comparison of the results.

You should already be familiar with programming in python, however there will be some aspects of this that you might not have done before. In particular, at this point in your coding career, you may not have ever done the following within python:

- Looped through ever directory within a directory
- Run a shell command from python, and checked the result

Both of these are possible though! For the former, I recommend that you use the **`os`** module, in particular the **`os.listdir`** function. For the latter, you should use the **`subprocess`** module, in particular **`subprocess.Popen`** and **`subprocess.check_output`** functions. The subprocess module gives you the ability to run a shell / bash command from python and fetch what the command printed out. You can read more about this module on yht python docs website: https://docs.python.org/3/library/subprocess.html.

Let's walk through an example to see how this program should behave. We will use pythagorean.c as our program to test.

First off, say we have started working on **`pythagorean.c`**, but it is not finished and there is a syntax error in the code. If we try to run the tester on this program that has a syntax error, it should beehive in the following way:

```
$ python3 sbt.py
C source file path: ./pythagorean.c
Test file directory path: ./tests
failed to compile your code
$
```

Now say we spend some more time working on **`pythagorean.c`** so that there are no longer any syntax or compilation problems, though we are still unsure if it actually produces the correct answer. In order to test this, we will create a test directory that contains two test cases:

```
── tests
    ├── area-3-4
    │   ├── input.txt
    │   └── output.txt
    └── area-7-2
        ├── input.txt
        └── output.txt
```

Each of the .txt files have the following contents.

area-3-4/input.txt	area-3-4/output.txt	area-7-2/input.txt	area-7-2/output.txt
3 4	Enter side A length: Enter side B length: Length of side C: 5.00	7 2	Enter side A length: Enter side B length: Length of side C: 7.28

Notice how in the **output.txt** files, the numbers that the user would type as input are not included. Now, we go to test again!

```
$ python3 sbt.py
C source file path: ./pythagorean.c
Test file directory path: ./tests
#### Test: area-3-4 passed! ####
#### Test: area-7-2 failed! ####
#### EXPECTED TO SEE:
Enter side A length:
Enter side B length:
Length of side C: 7.28
#### INSTEAD GOT:
Enter side A length:
Enter side B length:
Length of side C: 7.280110
$
```

Notice that the area-3-4 test case passed! However, the other one failed, due to the expected output having less decimal places printed out. If we went in and fixed the program and then ran again, we should see:

```
$ python3 sbt.py
C source file path: ./pythagorean.c
Test file directory path: ./tests
#### Test: area-3-4 passed! ####
#### Test: area-7-2 passed! ####
$
```

You should come up with a few tests of your own, in addition to the ones shown here in the spec, and ensure your program passes each one.

Submitting Your PA

After you have completed the three parts of this PA, double check that the file structure and file / directory names match what is shown in the project overview on the first page. You are welcome to include your test cases within the sbt or pythagorean directories as well if you want. Also, before you submit, you should ensure that your bash scripts, C program, and SBT compile / run correctly on lectura, not just your own computer.

Once you are ready to submit, zip up the **pa1** directory by running:

```
$ zip -r pa1.zip ./pa1
```

Then, turn this file to the PA 1 dropbox on gradescope.