Assignment #5—It all makes sense!

Due: February 22, 2023 at 11:59pm

This assignment consists of two different components:

1. Practice using `make` and CI tools
2. Practice with profiling tools

We expect this assignment to take 1-3 hours depending on your proficiency level with the tools. If you find yourself unproductively stuck or unproductively struggling, ask on Ed and/or go to office hours!

Part I: Practice with Makefiles and CI Tools (2 points)

In this part of the assignment, you will practice writing a `Makefile` that compiles C files. Don't worry, you won't have to write any C code! Instead, we will provide you with the source code for a super secret, super advanced calculator program.

The program is exceptionally complex, and therefore the code consists of 6 (!!) separate files. Compiling 6 files individually is far too much work... We are soliciting your compiling expertise to help us compile the program.

The six files in the program are:

- `main.c`: which implements the core functionality of the calculator, such as prompting the user to enter an operation (+, -, /, *) and calling the appropriate helper function to execute the operation
- `addition.c`: which houses the `add` function
- `subtraction.c`: which houses the `subtract` function
- `multiplication.c`: which houses the `multiply` function
- `division.c`: which houses the `divide` function
- `operations.h`: which defines the prototypes for the `add`, `subtract`, `multiply` and `divide` functions

You can fork then clone these files from [this](#) repository on GitHub. You should make your own **fork** (just like you did in Assignment 4) and then work within that forked repository. For this part of the assignment, you'll be working entirely within the `calculator` directory.

Your task is to (1) write a `Makefile` that will compile these files and allow us to run our program in an executable called `calculator`, then (2) write some tests that will automatically be run by CI. We would like to be able to run the following commands:

- `make`: which should produce an executable called `calculator`

- `make clean`: which should remove any previous binaries (e.g. `.o` files) and should then produce an executable called `calculator`

To test that your `Makefile` works, you should run `make` and then invoke the program using `./calculator` and the appropriate arguments. You should also try running `make clean && make` and then `./calculator` (with arguments). In both cases, you should successfully get the calculator program to run.

Remember that, when you run `make` without specifying a "target", it runs the first rule in the Makefile by default.

Then, add **CI Tests** in the `tests` directory. Our CI file (which you can view at `.github/workflows` in the repository if you want!) will automatically run every executable file in the `calculator/tests` directory– we've provided one example file for you. Add some tests (and/or test files, aka "test suites") to your files, so that when you push to your GitHub repository, you get a green checkmark ✅.

Put your Github username in a file called `github.txt`. We will use this to check that your CI tests are working.

> ℹ️ *Aside: Our GitHub workflow file happens in three steps ("jobs").*
> 1. *("`build`") First, it builds your Calculator app using the Makefile you write.*
> 2. *("`test_setup`") Second, it lists out all the executable files in your `tests` directory, and puts it in a format that GitHub Actions understands (JSON format).*
> 3+. *("`test`") Finally, it takes in the list from step (2) and creates a new test job that runs each of those files!*
>
> *The CI fails (❌) if any step of any job fails. Take a look at the format of the file! It's written in YAML, a structured data file that's fairly fast and easy to read and write. It lists a name of the workflow; when it runs (on push, on pull request, and when manually invoked); what permissions we get access to while running CI (the contents of the repository, but read-only); and what jobs constitute the CI run.*

Part II: Profiling Code (1 points)

For this part of the assignment, we will practice profiling some code. Our newest endeavor to produce cutting edge CS algorithmic research to find the most efficient implementation of sorting! The `sorts.py` file includes three different sort implementations. It's available in the `profiling` directory of the repo you forked and cloned for Part I.

Our issue: we can't figure out which one is fastest! Your task is to help us by using profiling tools to figure out which one is fastest in terms of CPU usage.

First, you should use `cProfile` to figure out which sorting implementation takes the most time. Remember that you can use the `-s tottime` flag to sort by the amount of time. Place your answer (the name of the sorting algorithm) on the first line of a file called `profiling.txt`.

Next you should figure out the bottleneck for each sort implementation using [line-profiler](). You will need to install `line-profiler` if you don't have it already. You can install it using:

```
Unset
pip3 install line-profiler
```

To run the `line-profiler`, you will need to add the `@profile` decorator to each function that you want to profile. Once you've added the decorator(s), you can run:

```
Unset
kernprof -l -v sorts.py
```

Below are the first few lines of the output for the analysis for `quicksort`:

```
Unset
Total time: 0.058354 s
File: sorts.py
Function: quicksort at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           @profile
    23                                           def quicksort(array):
    24     17566       3791.0      0.2      6.5       if len(array) <= 1:
    25     17566       2720.0      0.2      4.7           return array
    26     16566       2670.0      0.2      4.6       pivot = array[0]
[ CONTINUED ]
```

You will want to identify which line takes the longest amount of time in the program as a whole, and which line takes the longest amount of time per execution. (If there is a tie, you can list the first line that has the longest time per execution.) For the sample output above, we see that the line `if len(array) <= 1:` takes the longest amount of time as a percentage of total time (6.5%). However, in terms of time per execution (labeled `Time per Hit`), all three lines take the same amount of time. They each take 0.2 units of time, where a unit of time is 10^{-6} seconds.

For your results for this part, you should include two answers (each on a separate line): the line that takes the longest in terms of percentage of total time, and the line that takes the longest in terms of time per execution.

Your final `profiling.txt` file should look as follows:

```
Unset
<slowest-algorithm>
<line-of-code-from-insertionsort-longest-percent-tottime>
<line-of-code-from-insertionsort-longest-time-per-hit>
<line-of-code-from-quicksort-longest-percent-tottime>
<line-of-code-from-quicksort-longest-time-per-hit>
<line-of-code-from-quicksort_inplace-longest-percent-tottime>
<line-of-code-from-quicksort_inplace-longest-time-per-hit>
```

For instance, if we used the sample results from above (which are wrong as they only look at a subset of the results for `insertionsort`), we could start populating our file as follows:

```
Unset
<slowest-algorithm>
if len(array) <= 1:
if len(array) <= 1:
<line-of-code-quicksort-longest-percent-tottime>
<line-of-code-quicksort-longest-time-per-hit>
<line-of-code-quicksort_inplace-longest-percent-tottime>
<line-of-code-quicksort_inplace-longest-time-per-hit>
```

Feedback Survey (0.5 points)

Once you have completed the assignment, you can earn an additional 0.5 points by completing our anonymous feedback survey. Given this is the first offering of the course, we want to collect as much feedback as possible to improve the course in the future. You can complete the survey [here](). Once you complete the survey, you will receive a completion code which you should place in a text file named `survey.txt`. The survey is anonymous so submitting the completion code is the only way to verify that you completed the survey. *Please do not share this code with anyone, as that would constitute a breach of the honor code.*

Submitting Your Assignment

Once you have finished this assignment, you will need to upload your files to [Gradescope](). Make sure to upload all files to the Assignment 4 submission page. You should also upload `survey.txt` if you completed the survey.

This assignment will be autograded for Part I, Task I (the Makefile) and Part II (Profiling). Part I, Task II (CI tests) will be manually graded. You will thus see that Part I, Task II will not be graded until one of us goes through and checks your tests.

```
Unset
zip -jv assign5_submission.zip ./calculator/Makefile
./profiling/profiling.txt ./github.txt ./survey.txt
./calculator/tests/*
```

All files must have the same name as specified above.