ORIE 5270: Big Data Technologies
Homework 4 – Due: Friday 04/30/2021

Instructions. Submit by **Friday, April 30th** at **11:59pm (midnight) US EDT)**.

Note: Problems 1 has to be submitted via your Cornell Github (detailed instructions are available at the end of each problem). All the other problems should be submitted on Gradescope. **Problems 4 & 5 are optional**; you do not have to submit them to get full score on this homework, but they can be used to boost your score on other homeworks.

Problem 1. For this problem, you will create a Python project in a **private Github repository** (use Cornell's Github website just like in HW1). More specifically:

- Pick a data structure or algorithm of your choice; you may choose to implement anything that we have covered in class or homeworks, *except for the stack data structure and the binary search algorithm.* You may also choose a data structure or algorithm that we haven't covered in class.

  Please contact us on Campuswire if you are unsure about whether the data structure or algorithm you chose is appropriate for this exercise.

- Create a private Github repository containing a Python module with your implementation. In particular, you should:

  1. Follow the project structure for Python projects that was suggested during the lectures on testing and documentation (see page 3 of the March 29th lecture slides).
  2. Include a collection of unit tests using the `unittest` library and following the suggested naming conventions; i.e., running `python -m unittest` from the base folder of your repository should automatically discover and run all your unit tests.
  3. Add documentation (docstrings) to all your Python classes and (public) functions / methods.
  4. Include an `html/` subfolder in the `docs/` section of your project, containing the HTML output of your project's documentation, using a tool like `pdoc3` or `Sphinx` (page 14 of the March 31st lecture slides contains an example using `pdoc3`).

Unit tests: your unit tests should be meaningful, i.e., they should cover nontrivial or "difficult" inputs, corner cases, check that your code raises the correct types of exceptions when applicable, etc. You should aim for having *at least* 5 or 6 unit tests.

Documentation: your docstrings should follow an established style, such as the ones mentioned in class (rEST, numpydoc, or Google style). If you are using a different style, please provide a link to its specification.

To generate HTML output for your documentation using `pdoc`, you can just run:

```
$ pdoc --html --output_dir docs/build <your_package_name>
```

What to submit: for this problem, do not submit anything on Gradescope. Instead, make sure your project is private on Github and add the instructor (`vc333@cornell.edu`) and the TAs (`asa97@cornell.edu` and `ssc255@cornell.edu`) as collaborators using the web interface.

Problem 2. Download the file `stream_example.py` and read through its code. This is a multi-threaded implementation of a program that counts the number of odd elements in a stream. Then, do the following:

- Identify any places where **race conditions** can happen. Add inline comments to those places in the code, explaining why it creates a race condition.
- Add appropriate synchronization mechanisms (locks or semaphores) to prevent the aforementioned race conditions.

What to submit: For this problem, submit your modified version of `stream_example.py` (containing inline comments and appropriate synchronization) on Gradescope.

Problem 3. For this part, you will implement a distributed version of matrix-vector multiplication (the operation $y = Ax$, where $A \in \mathbb{R}^{m \times n}$), using the Python `multiprocessing` library. Download the file `matvec.py` and modify the code appropriately so that it achieves the following:

- Each worker process should receive the input vector x and one or more rows of the array A, and compute a portion of the final result y. This final result should eventually be available in the master process.

  Hint: use a shared array y for storing the results.

- Your code should not introduce any race conditions; i.e., you should not have to use any locks or semaphores for your program to run correctly.

Problem 4. Bonus (25%).

Implement the parallel version of `mergesort` we saw in the `threading` lecture, but now using `multiprocessing`. Use the `timeit` module to investigate the effect of the number of processes used to the runtime of the algorithm, by sorting the same array using $1, 2, 4, 8$ and 16 available processes.

You can also use the parallel version of `merge` from the April 12th lecture at the final step of your algorithm to try and obtain an additional speed-up.

Note: In the `threading` example, we could operate on the input array A directly since threads have shared memory. With `multiprocessing`, the array A can still be passed as an argument but each process needs to communicate the changes back to the master process explicitly. You will have to use an appropriate shared memory construct for this.

Problem 5. Bonus (25%). An example application where parallelization can make a big difference is that of *Monte Carlo simulation*. The objective is to simulate a process, whose mean is known to be equal to a quantity of interest, by taking the *sample mean* of a set of independent trials. The larger the number of trials, the more accurate our estimate will be.

In this problem, you will write a Monte-Carlo simulation to estimate the value of the mathematical constant π. At a high-level, the algorithm works as follows:

1. Generate N random vectors $z^i \in \mathbb{R}^2$ which are sampled from the uniform distribution in the unit square, $[-1, 1] \times [-1, 1]$.
2. For each of those random numbers, output `1` if $\sqrt{(z_1^i)^2 + (z_2^i)^2} \leq 1$, and `0` otherwise.
3. Let N_1 be the number of 1's from the previous step; we output $\hat{\pi} = \frac{4N_1}{N}$ as our estimate of π.

Why does this algorithm work? Observe that we only output 1 in the second step of the algorithm if the sample z^i falls in the unit circle. In other words, the probability that we increment our count N_1 is equal to the probability that z^i falls inside the unit circle. The unit circle has volume $\pi r^2 = \pi$, while the unit square has volume $2 \times 2 = 4$. Therefore, the probability that we output `1` is equal to $\frac{\pi}{4}$, so $\mathbb{E}[N_1] = \frac{N\pi}{4}$, which is why we scale the estimate by 4.

Use a `multiprocessing.ProcessPool` to implement the above algorithm in parallel. In particular, you should:

- Set the number of processes managed by the `ProcessPool` equal to the number of available cores in your system.
- Define a function `trial(m)` that performs m trials of the above algorithm; i.e., it generates m random samples z^i uniformly distributed in the unit square, and counts how many of them fall inside the unit circle.
- Use `ProcessPool.map` to apply the trials in parallel and obtain a list of partial counts; finally, divide the sum by the total number of runs N that you want to simulate.

  Hint: `ProcessPool.map(f, [x1, ..., xn])` returns `[f(x1), ..., f(xn)]`, so you will have to figure out what list to apply the function `trial` to.

Note: The total number of trials N can be hardcoded in your Python script, or passed as an argument from the user. In any case, you may assume that N is a multiple of the number of available cores in your system for simplicity.