CMPS 2200 Recitation 01

Name (Team Member 1):___________________________________
Name (Team Member 2):___________________________________

In this recitation, we will investigate asymptotic complexity. Additionally, we will get familiar with the various technologies we'll use for recitations and assignments for this semester.

To complete this recitation, follow the instructions in this document. Some of your answers will go in this file, and others will require you to edit `main.py`.

Setup

- Login to Github.
- Click on the assignment link posted on canvas and accept the assignment.
- Click on your personal github repository for the assignment (e.g., https://github.com/CMPS-2200/recitation-01-your_username).
- Clone the repository to your local device
- Complete the lab task
- Add, commit, and push your completed lab back up to GitHub.
  - You will need to issue `git add` for all files that you have modified, e.g., `main.py`, `README.md`, and any others that you modify as well.
  - For example, on the command line, in the same directory as your cloned lab:

```
$ git add main.py
$ git commit -m "Implement Required Functions"
$ git push origin main
```

Running and testing your code

- You can run the tests using `pytest`. To install `pytest`, on your terminal:
  - `pip3 install pytest`
  - You may also have to install other python modules such as `tabulate` or other imported modules as you work through these recitations.
- It's usually best to run only one test at a time. To run tests, from the command-line, you can run
  - `pytest -s main.py` will run all tests
  - `pytest -s main.py::test_one` will just run `test_one`
- If you want to run your whole program, make sure to use `python3`. `python` still defaults to python version 2.

Turning in your work

- You may work with a partner to complete this recitation.
- Only one team member needs to push your completed lab to github.
- In the README.md file, include the names of the team members.

Comparing search algorithms

We'll compare the running times of `linear_search` and `binary_search` empirically.

☐ 1. In `main.py`, the implementation of `linear_search` is already complete. Your task is to implement `binary_search`. Implement a recursive solution using the helper function `_binary_search`.

☐ 2. Test that your function is correct by calling from the command-line `pytest main.py::test_binary_search`

□ 3. Write at least two additional test cases in `test_binary_search` and confirm they pass.

□ 4. Describe the worst case input value of `key` for `linear_search`? for `binary_search`?

enter your answer in answers.md

□ 5. Describe the best case input value of `key` for `linear_search`? for `binary_search`?

enter your answer in answers.md

□ 6. Complete the `time_search` function to compute the running time of a search function. Note that this is an example of a "higher order" function, since one of its parameters is another function.

□ 7. Complete the `compare_search` function to compare the running times of linear search and binary search. Confirm the implementation by running `pytest main.py::test_compare_search`, which contains some simple checks.

□ 8. Call `print_results(compare_search())` and paste the results here:

enter your answer in answers.md

□ 9. The theoretical worst-case running time of linear search is $O(n)$ and binary search is $O(log_2(n))$. Do these theoretical running times match your empirical results? Why or why not?

enter your answer in answers.md

□ 10. Binary search assumes the input list is already sorted. Assume it takes $\Theta(n^2)$ time to sort a list of length n. Suppose you know ahead of time that you will search the same list k times.

- What is worst-case complexity of searching a list of n elements k times using linear search? **enter your answer in answers.md**
- For binary search? **enter your answer in answers.md**
- For what values of k is it more efficient to first sort and then use binary search versus just using linear search without sorting? **enter your answer in answers.md**