CS 5112 Algorithms and Data Structures for Applications Fall 2019

Homework 4

Due: December 2, 2019, 11:59 PM

This is a short assignment to leave you more time for the final exam, you will implement Boyer-Moore and a Bloom Filter.

Key instructions:

- 1. Collaboration: You are **required** to work in groups of 2 students on each assignment. Please indicate the name of your collaborator at the top of each assignment and cite any references you used (including articles, books, code, websites, and personal communications). If you're not sure whether to cite a source, err on the side of caution and cite it. Remember NOT TO PLAGIARIZE: **all solutions must be written by members of the group**.
- 2. Partner-finding: You have one week to find your preferred partner yourself and form your own group **on CMS**. If you have not formed a group on CMS by the **partner-finding deadline on 11.29th 11:59PM**, we will run a partner-matching script to assign groups for you.
- 3. Please only use **Python 3** for this assignment
- 4. Please do NOT use any additional imports, only write your code where you see `TODO: YOUR CODE HERE`, and change your return value accordingly.
- 5. Reminder on Late Policy: **Each** student has a total of one slip day that may be used without penalty for homework. We will also drop your lowest homework score. An assignment can be at most one day late without penalty via slip days.
- 6. Please modify and submit the following files: `boyer_moore.py` and `bloom_filter.py`

---

1 Boyer-Moore algorithm

In part 1 of HW4, you will implement the Boyer-Moore algorithm as described in lecture.

TODO In the provided class (`boyer_moore.py`), implement

- `add_next_element`
- `get_majority`

You may assume in your code and do not need to check that each call to `add_next_element` will be a single 1-character string drawn from the upper- and lowercase alphabet. We will check that the values of `counter` and `guess` are correct, although we will not check the value of `guess` when `counter` is zero.

Testing Boyer-Moore

When you run `python3 test_boyer_moore.py`, by default it should terminate when 5 cases failed[1]. You can change this behavior by changing `max_test_failures`, or by passing command-line arguments(e.g. `python3 test_boyer_moore.py --max-test-failures 10`).

For Boyer-Moore, in addition to the final result, we will also grade your code by checking if the counter is correct at each step, and if the guess is correct at each step where the counter is not 0. The test cases are in (`testcases_boyer_moore.txt`). Each line has 4 semicolon-delimited fields, which are:

name	name of the test
symbols	string of symbols
expected guess	symbol, or '!'
expected counter	integer

2 Bloom filter

In part 2 of HW4, you will implement a simple Bloom filter.

TODO In the provided class (`bloom_filter.py`), implement

- `add_elem`
- `check_membership`

Important: Your Bloom filter implementation **must** use the three hash functions that we have provided (`cs5112_hash1`, `cs5112_hash2`, and `cs5112_hash3`).

Do not use Python's built-in hash function! You must use these three.

You are asked to implement a Bloom filter with k=3 hash functions and a m=10 bit array.

The comments in `bloom_filter.py` say more about what we expect for inputs and outputs.

2.1 Testing Bloom filter

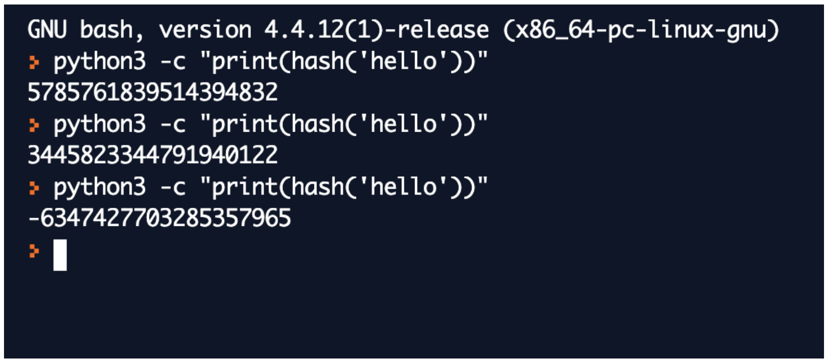
For your Bloom filter, in addition to testing the output of your functions, we will also be inspecting how you're storing data in the underlying array. Therefore, be sure to implement the algorithms to spec.

Run `python3 test_bloom_filter.py` for a very simple test.

Note on Python 3's `hash` function TL;DR: This note shouldn't affect your implementation in `bloom_filter.py` at all, but might be useful when you are debugging.

Note that Python 3's built-in hash is non-deterministic. For example, every time you run `python3 -c "print(hash('hello'))"`, the result will be different. This is great for security reasons, but it can also be a pain when debugging or for research as we want to be able to regenerate previous experimental results in these scenarios.

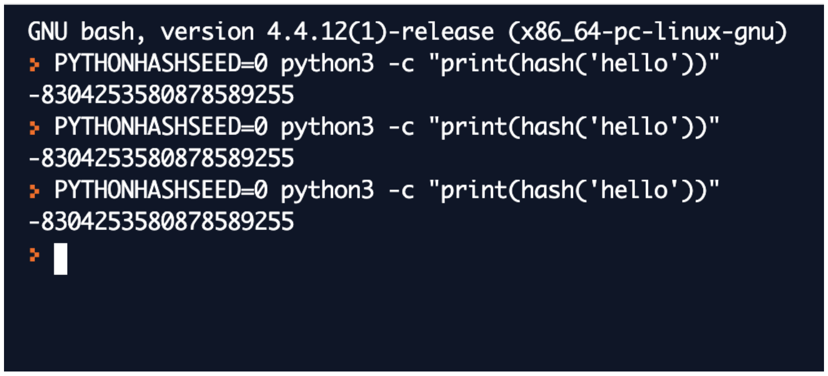
[1] Some cases may pass "by luck" even when you haven't implemented the algorithm yet, because we initialize `guess` as `None` and `counter` as `0`.



```
GNU bash, version 4.4.12(1)-release (x86_64-pc-linux-gnu)
> python3 -c "print(hash('hello'))"
5785761839514394832
> python3 -c "print(hash('hello'))"
3445823344791940122
> python3 -c "print(hash('hello'))"
-6347427703285357965
>
```

Figure 1: Default behavior since Python 3.3, good for security reasons

One way[2] to get past this when you are debugging or testing is by setting the environment variable `PYTHONHASHSEED` to an integer value, like `PYTHONHASHSEED=0 python3 -c "print(hash('hello'))"`.



```
GNU bash, version 4.4.12(1)-release (x86_64-pc-linux-gnu)
> PYTHONHASHSEED=0 python3 -c "print(hash('hello'))"
-8304253580878589255
> PYTHONHASHSEED=0 python3 -c "print(hash('hello'))"
-8304253580878589255
> PYTHONHASHSEED=0 python3 -c "print(hash('hello'))"
-8304253580878589255
>
```

Figure 2: However, determinism might be more desirable when debugging and testing

[2]Admittedly, this approach is intrinsically hacky. In real-world production code, if you actually want a deterministic hash, you should almost always use hashlib.