Programming Assignment 1

Department of Computer Science, University of Wisconsin – Whitewater
Algorithms in the Real World (CS 738)

Instructions For Submissions

- **This assignment is to be complete individually.** Submission is via Canvas as a single zip file. Submit code and a brief report.
- **Your choice of programming languages are limited to C++ or Java.** Make sure you cite everything. If you use online code resources, please make sure that it is not a verbatim copy; you really should avoid this in general.

---

1 Overview

We are going to compare Bloom Filter and Count-Min Sketch (CMS) against C++/Java's in-built hashset/hashmap. The basic idea is to generate a data and add those numbers to the Bloom Filter (or CMS) and in-built hashset/hashmap. Then, generate a test data to note how many numbers are found in Bloom Filter (but not in hashset), which gives us a measure of the false positive rate. Likewise, for CMS, we will note the performance of the CMS against a hashmap and try to ascertain if it is close to the theoretical expectations or not.

In this assignment, most of the code is given to you and you need to use it. Your main task is to run multiple experiments and write a detailed report. Let's start with what is already provided.

1.1 Universal Hashing

The constructor requires a prime p and table-size m. Remember that p should be more than the universe size (and therefore any key) and more than m (else you won't be using the entire hash table). Then, the constructor generates a random integer $a \in [1, p)$ and a random integer $b \in [0, p)$. The `hash` function returns the hash value for key as $\big((a \cdot key + b) \mod p\big) \mod m$

1.2 Prime Generation

The class contains a method `getPrime` that returns a random prime in $[LB, UB]$, where the range from LB to UB should be sufficiently large (failing which the code will get stuck in an infinite loop or throw an exception when $UB < LB$). Since primes are pretty common, we generate a number n randomly in $[LB, UB]$ and then use a standard deterministic $O(\sqrt{n})$ algorithm to test if n is prime.

1.3 Bit Array

A bit-array implementation is provided with the following functions:

- The constructor asks for the size of the bit-array.

- `access` returns a true or false depending on whether the bit at *index* is 1 or 0.
- `set` changes the bit at *index* to a 1.
- `clear` changes the bit at *index* to a 0. You should have no use of this function.
- `size` returns the size of the bit-array.

We can use `BitSet` in Java or `vector<bool>` in C++ for the same purpose. I used a custom implementation for naming consistency.

2 Implement Bloom Filter

Your `BloomFilter` class ideally will need the following class variables:

- β: the number of hash functions
- *ba*: a `BitArray` object. In C++, you can also use a pointer, but remember to delete the pointer in the `BloomFilter` destructor to prevent a memory leak.
- *hashes*: an `UniverSalHashing` array in Java or `UniverSalHashing` vector in C++. This will store β-many universal hash-functions for the Bloom Filter

The functions of the class are:

- a constructor that will accept the size of the Bloom-Filter, the size of the universe, and the number of hash functions.

  Set the class variable β appropriately. Then, allocate the space for *ba* and for *hashes*.

  Finally, create the appropriate universal-hash functions and populate *hashes* with each hash function. To this end, first generate a prime in the range $[1.25 * universe, 1.5 * universe]$. (Feel free to change this if you want, but remember that the prime must be greater than the universe for universal hashing.) Using this prime, create a universal hash function (using the constructor of `UniversalHashing`) and add it to *hashes*.
- `insert` accepts an integer and adds it
- `search` accepts an integer and returns true if it is found, else returns false.

Check the code in `TestBloomFilter` to understand the order of parameters in each function.

3 Test Bloom Filter

Initially the code will generate the data and write it to a file (at `DATA_PATH`). Then, it will read the data from the file and populate the Bloom Filter with it. Following this, it will search for every number in the universe and produce an output. If you wish to carry out multiple experiments for the same data, then you should comment the `generateBloomData()` call in the main after one complete execution.

You will carry out multiple comparison tests. Primarily, code tests the following:

- the average insertion times for the Bloom Filter and in-built hash set,
- the average search times for the Bloom Filter and in-built hash set,

- the false positive rate of the Bloom Filter, and
- the false negative rate of the Bloom Filter – this is just to make sure nothing is broken.

Your task will be to change the following parameters and record the output for the report:

- n (data size),
- *universe* (the range in which the data is generated and the number of searches),
- α (load factor, i.e., the Bloom Filter uses α bits for every number inserted), and
- β (the number of hash functions, which by default is optimized for the load factor).

Here are some results that I obtained:

Java Bloom Filter Test

```
Data size = 1000000
Universe size = 4000000
Load factor = 16
Number of hash functions = 12

% of Successful Searches = 22.11855% (884742 out of 4000000)
False Positive Rate = 9.528925%
False Negative Rate = 0.0%

Average Insertion Time: Hash = 3.11E-4; BF = 0.001603
Average Search Time: Hash = 5.475E-5; BF = 7.25E-4
```

C++ Bloom Filter Test

```
Data size = 1000000
Universe size = 4000000
Load factor = 16
Number of hash functions = 12

% of Successful Searches = 22.1257% (885028 out of 4000000)
False Positive Rate = 9.8937%
False Negative Rate = 0%

Average Insertion Time: Hash = 0.0011883; BF = 0.0638376
Average Search Time: Hash = 0.000642959; BF = 0.0375375
```

4 Implement CMS

Your `CMS` class ideally will need the following class variables:

- β: the number of hash functions (i.e., the number of rows)
- m: the length of each row (i.e., the number of columns)

- *table*: a two-dimensional integer table
- *hashes*: an `UniverSalHashing` array in Java or `UniverSalHashing` vector in C++. This will store all the universal hash-functions for the Bloom Filter.

The functions of the class are:

- a constructor that will accept the number of rows, the number of columns, and the size of the universe.

  Set the class variables β and m appropriately. Then, allocate the space for *table* and *hashes*.

  Finally, populate *hashes* like you did for the Bloom Filter implementation. I generate a prime in the range $[20 * universe, 25 * universe]$ for this implementation. You will need to modify this depending on the size of the universe but ensure that the prime should be more than the size of the universe for Universal Hashing.
- A function `increment` that accepts an integer and increments its frequency
- A function `frequency` that accepts an integer and returns its estimated frequency

Check the code in `TestCMS` to understand the order of parameters in each function.

5 Test CMS

Initially, the code will prompt you for generating the data. You can keep generating as long as you want.[1] Once the data has been generated, the code will treat the data as a stream and populate CMS (as well as a hash map) with it. Next, it will compute the frequency of every number in the universe and produce an output. If you wish to carry out multiple experiments for the same data, then you should comment the `generateCMSData()` call in the main after one complete execution.

You will carry out multiple comparison tests. Primarily, code tests the following:

- the average overestimate of the CMS, i.e., the average of $(estimated - actual)$,
- the number of underestimates – this is just to make sure nothing is broken,
- % of cases where $(estimate - actual \geq \epsilon \cdot n)$,
- % of cases where $(estimate - actual \geq n/B)$,
- the average difference between *actual* and $\max(0, estimate - n/B)$, and
- the number of k-heavy hitters for a pre-defined k.

Your task will be to change the following parameters and record the output for the report:

- n (data size),
- *universe* (the range in which the data is generated and the number of searches),

[1] I have gone upto 2 billion data items in Java and about 500 million in C++, but it gets excruciatingly slow after that. I suspect it is (much) slower in C+, because the usage of the `BigInt` library found here: `https://faheel.github.io/BigInt/`. I am inclined to believe that this library isn't the best when it comes to performance. So feel free to make suggestions on a better resource to use.

- ϵ and δ (ϵ is the acceptable error and δ is the acceptable, error probability. Both must be < 1. Specifically, $\Pr(estimate - actual \geq \epsilon \cdot n) \leq delta$),
- B (the number of columns in the table, with default as $\lceil \frac{e}{\epsilon} \rceil$),
- β (the number of hash functions, with default as $\lceil \ln \frac{1}{\delta} \rceil$), and
- k for finding k-heavy hitters.

Here are some results that I obtained:

Java CMS Test

```
Number of data points (n): 75000000
Universe: 1000
Number of columns (B): 272
Number of rows (hash functions): 4
Epsilon: 0.01
Delta: 0.01
n/B: 275735
n*epsilon: 750000

Number of underestimates: 0
Average of actual: 75000.0
Average of (estimate - actual): 88500.62
% of cases where (estimate - actual >= n*epsilon): 0.0%
% of cases where (estimate >= actual + n/B): 1.4%
Average difference when we compute actual as (estimate - n/B): 67618.124

k for k-heavy hitters: 10000
Number of actual heavy-hitters: 751
Number of estimated heavy-hitters: 964
```

C++ CMS Test

```
Number of data points (n): 75000000
Universe: 1000
Number of columns (B): 272
Number of rows (hash functions): 4
Epsilon: 0.01
Delta: 0.01
n/B: 275735
n*epsilon: 750000

Number of underestimates: 0
Average of actual: 75000
Average of (estimate - actual): 90418.5
% of cases where (estimate - actual >= n*epsilon): 0%
% of cases where (estimate >= actual + n/B): 0.9%
Average difference when we compute actual as (estimate - n/B): 67611.7
```

```
k for k-heavy hitters: 10000
Number of actual heavy-hitters: 766
Number of estimated heavy-hitters: 973
```

6 Report

Conduct at least 4 experiments each for Bloom Filters and CMS. For example, you can study how changing the parameters alter the false positive rate of a Bloom Filter. Likewise, you can study how the estimates in a CMS get affected by the parameters.

I want you to independently design experiments, and so, I won't provide any more suggestions. However, you must ensure that two experiments are not extremely similar to each other. You must ensure that experiments make sense; for example, if you want to study the effect of hash functions on false-positive rates, ideally everything else should remain fixed.

The report will be graded on a curve; typically, you will be graded based on:

- how well thought out your experiments are - the motivation, the hypothesis, and the result
- the breadth of your experiments - how many things have you studied

I will suggest the following structure in the report. For each experiment, note the following:

- the motivation - why are you doing this experiment?
- the hypothesis - what do you think will happen?
- the outcome - you can just copy paste the output
- the conclusion - did the hypothesis and outcome agree? Why or why not?

  It is okay for the hypothesis and outcome to disagree, even if you can't explain it. So, "I have no idea why they disagree" is a perfectly reasonable response (within acceptable bounds).

Please be precise. Spend 1 or 2 sentences for each of the above bullet points.

7 Caveat Lector

This project is not designed to showcase the best performance of Bloom Filter and CMS. The motivation is to engage you in designing experiments, aided by a basic functional implementation. So, don't get put off by the high false positive rate and/or the terrible estimates by the CMS.

The main problem with the implementation is the choice of hash functions. Universal hashing is not truly random in a sense; for limited universe sizes, the outcome is often not the best. Moreover universal hashing is somewhat slow (more so because of the use of big integers in this project). It is better to choose more practical hash functions, such as MD5, SHA-256, Murmur3, and others.

Moreover, generating random numbers doesn't give a good representation of real-world data. So, that makes the experiments biased as well. Typically CMS like data structures don't work well for random data, because if we look into potential applications such as heavy hitters, we expect the data to be skewed. In random data, numbers are evenly distributed for the most part, which makes chances of finding heavy hitters less likely.

Food for thought: you can choose better hash functions and redesign the project with better data for potential project options.