Comp 202 - Fall 2021
Homework #3
Due date - 23:59 20/11/2021

1 Part 1

The Counting Bloom Filter is a generalized version of the Bloom Filter data structure that you will be implementing in Part 2 of the homework. This is a space efficient probabilistic data structure that is used to check if the number a given element appears in a sequence of elements is smaller than a specified threshold. The Counting Bloom Filter stores an integer array of size m, which is independent of the number of elements in the given sequence. There are two operations that can be performed in a Counting Bloom Filter: add and lookup. While iterating over the elements that are part of the given sequence we add them to the counting bloom filter and after all the sequence has been added we can use lookup to check if a given element was encountered less than a given threshold. In order to add the elements, the Counting Bloom Filter uses k hash functions. When we add an element from the sequence to this data structure, we firstly compute the k hash functions to find the corresponding indices for this element. Then we increment each of the resulting indices in the integer array. When we are performing lookup, we are given a threshold for the number of occurances we want to check for. As in addition, we firstly compute the hash functions on the item that we are given, and then check all the resulting indices. If all of the indices are greater than or equal to the given threshold, then the given item may have occured threshold times or a greater number of times. If one of the values in the resulting indices is less than the threshold then the item has definitely not been encountered in the sequence at least threshold times. As in the case of the Bloom Filter the Counting variant is prone to false positives, meaning that sometimes the result of lookup may indicate that an item has occured more than the input threshold, but that statement may be incorrect. Since we are using an integer array for storing the counts of the elements, if the given sequence is sufficiently large, some of the counts may overflow. Because of that, some Counting Bloom Filter implementations also include a bound, which is the maximum integer that may be present in the array. If that bound is reached the corresponding integer of the array is not incremented anymore. An example Counting Bloom Filter, with k = 3 and m = 10 is given in Figure 1. In this case, we are considering the sequence of elements: a1, a2, a3, a4. Each of the arrows in the figure indicates the index calculated by running one of the hash functions in the example. Suppose after adding the given sequence to the filter, we want to check if another element a5, has occured less than 1 times in the sequence. When computing the hashes of a5, we may get 0, 1, and 2 as the resulting indices for the 4 hash functions. Since all these indices are greater than or equal to 2, the output of the lookup function in this case would be true, meaning that a5 has been encountered at least 1 times in the sequence, which is not correct. In this case we say that a5 is a false positive. That is why the Counting Bloom Filter is called a probabilistic data structure. On the other hand, if the result of the hashes when computed in a5 were 0, 5, and 9 with the corresponding elements in the array: 1, 0, 0, the lookup function with a5 and threshold 1 would return false, meaning that a5 was never encountered in the sequence. In this case the result is correct. In fact, the Counting Bloom Filter can definitely indicate that an element was not encountered at least threshold times in a sequence, which means that it does not have any false negative results. Different implementations of Counting Bloom Filters also allow the delete operation to be performed but we are not going to consider that case in this homework.

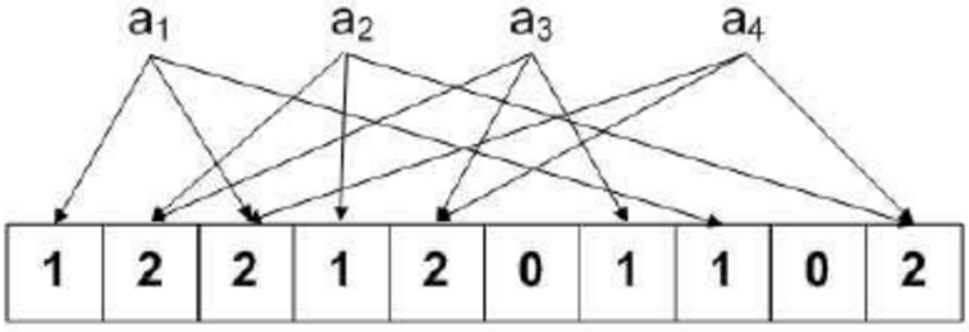

Figure 1: Counting Bloom Filter

You are going to implement a CountingBloomFilter that stores the number of occurences of integers in a sequence. To do that, you are given a CountingBloomFilter class that contains the methods that you should complete. There is also a Main class which adds the integers in a sequence to a Counting Bloom Filter and then queries the data structure for the occurences of some numbers. In order to complete the homework successfully you should correctly implement the following methods which are also marked with TODO in the source code provided to you.

- **void add(int item)** This method takes an integer as input, and adds it to the Counting Bloom Filter data structure. You should call the **calculateIndex** method of each of the hashes in the CountingBloomFilter class and increment the resulting indices in the **counts** array by one.
- **boolean lookup(int item, int threshold)** In this method you should calculate all of the hash functions of the given **item** and check the corresponding elements in the counts array. Then, as described above you should return a boolean indicating whether **item** occurs at least **threshold** times in the sequence.

2 Part 2

The Bloom Filter is bit array of size m that is used to efficiently check if an element has been encountered in a given sequence. The addition of an element to this data structure is performed with the help of k hash different hash functions, similarly to the Counting Bloom Filter case. The allowed operations on the Bloom Filter are again addition and lookup of an element. The lookup of an element is an operation where we find out if the element was added to the data structure or not. When adding an element to the filter all of the hash functions are computed on the given element and the resulting bits in the m bit array are set to 1. By doing this, we can then check if a new input has been previously added to the array by computing the k hash functions on that input and checking the resulting indices in the bit array. If all of the indices are 1, the element might have been present in the initial sequence. This means that there may be cases where the element is not in the array but the corresponding bits are 1. This may happen, for example, if after the addition of multiple elements, the bits that would correspond to another element are set to 1 coincidentally, resulting in false positive results as described in Part 1.

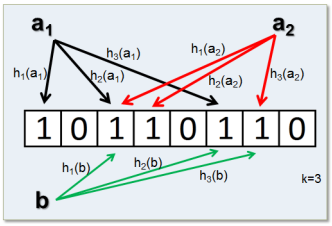

Figure 2: False Positive

Consider the case in Figure 1 depicting a Bloom Filter of size m = 8, with k = 3 hash functions. We are encountering a sequence with elements a1, a2. After each element, we set the corresponding indices in the bit array to 1. In this case, for a1, we set to 1 bits in indices 0, 2, and 5, and for a2 in indices 2, 3, and 6. When we are given a query to check whether another element, b, was present in the sequence we compute its hash functions as 2, 5, and 6. These bits are all 1 in this case, but b was never encountered the sequence of elements. Thus we say that b is a false positive.

On the other hand, if not all of the corresponding indices in the bit array are 1, we can definitely say that the current element was never added, since otherwise those bits would be all 1. The Bloom Filter data structure is useful when you only want to efficiently in space and time check the presence of an item in a sequence, without actually storing the sequence.

In this part, you will implement a Bloom Filter by extending the CountingBloomFilter class that you completed in Part 1 and a hash function that will be used with these data structures. You will have to write a class for the Bloom Filter called BloomFilter. The BloomFilter class should have the same constructors as the CountingBloomFilter class, except the bound of the counts should not be a parameter of neither of the constructors. Signatures and a description of the methods that need to be in the BloomFilter class are given below.

- **void add(int item)** The same function as in the CountingBloomFilter class

- **boolean lookup(int item)** This method should check if the specified **item** has been added to the Bloom Filter.

In the given source code, an interface called **HashFn** is provided, which helps in specifying a hash function that will be used in the bloom filter data structures. The CountingBloomFilter class stores the hash functions in an ArrayList of HashFn objects, so they can be dynamically added and removed. You should create your own hash function and add it to the list of hash functions in the Main class. Your hash function should implement the following method:

- **int calculateIndex(int value, int arrSize)** Given the array size calculate the index of the specified value. Your hash function should perform the following operations:
    1. Xor **value** with the result of doing an unsigned 13 bit shift to it (value) and multiply the result of xor with the hexadecimal value '7feb352d'.
    2. Perform step 1 again using its result as the value.
    3. Xor the result of step 2 with the result of using an unsigned 13 bit shift to it.
    4. The index should fit in the array, so take the modulo of the result of step 3 with the given array size and return it from the function. If the result is negative, multiply it by -1 an take its modulo with the given array size.

Submission Details

Usage of github classroom and submission details:

1. Accept the invitation using this link `https://classroom.github.com/a/6hsxLSR8`.
2. Choose your student ID from the list.
3. A personal repository with the starter code will be created. You can directly edit the code on github's page, clone it to your local storage and edit it there, or use the online IDE VS Code. Ensure that your repositories are private.
4. Make sure your changes are committed and pushed to the repository.
5. Check if you have successfully passed the tests in the "Actions" tab of your repository. Note that these automatic tests do help you understand whether or not your solution works up to some level, but passing them does not guarantee that you will receive a full grade.
6. To make sure your code is received and avoid any potential problems on github's side, submit a copy of java files on Blackboard as well. Submit all files as a single .zip file, named as:

"ID_NAME_SURNAME.zip"

Grading Criteria

You are going to receive 60 points for the correct implementation of Part 1 and 40 points for Part 2.
For your further questions about the homework send an email to: emerkuri20@ku.edu.tr