1 Overview

A **bloom filter** is a *probabilistic data structure* that can be used to predict, with different likelihoods of success, whether or not an element is in a set or not. Bloom filters are interesting because, while they can have false positives, they **cannot** have false negatives.

This means that, when attempting to determine if an element is in a set, a bloom filter can only tell us either:

- the element **might be** in the set, or
- the element **definitely is not** in the set

2 The Algorithm

The below section will describe how the bloom filter algorithm is implemented – but first, a quick refresher!

### 2.1 Hashing (and HashSet) Refresher

Recall that hashed data structures (like HashSet and HashMap) are particularly good at look-ups, as they can perform this much quicker than linear data structures (like a Queue or ArrayList). This is due to their use of **hash codes**.

A **hash code** is an integer generated from a **hash function**. A hash function accepts some piece of data and generates a number that is relatively unique to that data. A good hash function will **always** return the same number when fed the same data, and a good hash function will rarely generate the same number for different data.

**Every non-primitive type in Java has a **hashCode()** method** (which it inherits from **Object** if it’s not overridden). For example:

- "cat dog bird frog".hashCode() returns 674136573
- "bat dog bird frog".hashCode() returns -679173124

At a high level, a HashSet is backed by a fixed length array, and uses an element’s .hashCode() to determine which index it stores the element in. We deal with collisions (when different elements hash to the same index) by either storing multiple elements in each index (chaining) or placing the element the next available empty index (probing).

Regardless, when checking to see if a HashSet contains an element, we use its hash code to determine where it **should** be in the array, and then check all the elements at that index or adjacent to that index as needed to determine if it has already been inserted into the set. The **bloom filter** seeks to improve on this even further.

### 2.2 The Bloom Filter

The bloom filter works similar to HashSet described above, but instead of using one function to hash its data, it will use **multiple**. A bloom filter consists of a **bit array** (represented by a fixed-length boolean array in our implementation), which it uses to track the elements that have been inserted into its set.
We will define the following with respect to our bloom filter:

- an integer $m$ representing the *length* of our bit array
- an integer $k$ representing the number of hash functions we will use when inserting elements into our set.

When inserting an element into our set, we update our bloom filter as follows:

1. run the data through all $k$ of our hash functions.
2. for each integer returned from the $k$ different hash functions, mod the result by $m$ and set that index in the bit array to *true*. Some of these indices may already be *true* from inserting a previous element (which is ok).

Thus, determining if an element is in our set using the bloom filter works as follows:

1. if *all* the hashed indices calculated from feeding the element to the $k$ hash functions are *true* in the bit array, that element *might be* in the set.
2. if *any* of the hashed indices calculated from the $k$ functions are *false*, that element is definitely not in the set.
3. if an element is determined to *might be* in the set but actually *is not* in the set, that is a false positive.

### 2.3 Filter Example

Below is an example bloom filter with:

- an $m$ of **10** (length of the bit array)
- a $k$ of **3** (number of hash functions)

To start, we can visualize our *bit array* as so:

```
bitArray = { F F F F F F F F F }
```

First, let’s say we insert "dog" into our set. Per our bloom filter’s $k$ value, that means we run "dog" through **three** hash algorithms (which mod the resulting integers by $m$ to get a valid index), and flip those indices in our *bit array*.

Imagine the three indices calculated for "dog" are 4, 5, and 6, like so:

```
bitArray = { F F F T T T F F F }
```

Next, we insert "cat" into our set; the resulting indices from our hash functions are **2**, **3**, and **4**, as seen on the next page:
Suppose we want to check and see if "goose" is in our set. We follow a similar process: feed "goose" to our hash functions and check the respective indices:

Lastly, let’s suppose we check if "ground hog" is in our set; this scenario results in a false positive:

Note that, based on the hash algorithms used, the hashed indices may or may not be sequential (more on this below).
3 The Provided Code

Provided to you are four classes total. However, you will only need to modify one:

- **BloomFilter**: where you will implement your bloom filter operations as described below.

You will need to read, use but not modify the following classes:

- **BFHash**: an interface for a family of String hash functions used by a BloomFilter (contains just one method).
- **SimpleHash**: simple hash algorithm that generates a very quick but potentially less unique hash code – handy for testing.
- **AdvancedHash**: a smarter and more complicated hash algorithm that is a bit more involved but generates more unique hash codes (the specifics of the algorithm aren’t important, but read the comments if you’re curious!)

4 Your Task

Your task in this lab is to finish implementing the functionality of the bloom filter, including a static functions which tests the false positivity rate. Complete the following tasks in the order they are listed. Once you have completed the Bloom Filter implementation, you will experiment and record some observations.

4.1 Finish the BloomFilter Constructor

Though you don’t have the full picture of these classes and their relationships just yet, take a look at BloomFilter’s constructor. There’s one line of code missing here – figure out what it is and implement it! *(Hint: you don’t need to add any instance variables here)*

4.2 Add to Filter

Implement the addToFilter method. Some example code is given to help you get started.

4.3 Contains

Next, implement the contains method. Pay attention to comments and reread Section 2.3 as needed.

4.4 Run Lab Example

With the above completed, it’s time to test your implementation of BloomFilter thus far. In the runLabExample method, replicate the example from Section 2.3 – this example used hash indices generated from SimpleHash, so your results should match exactly.

Use print statements each step of the way to verify your progress. Some code is already provided for you but may need correcting, and you’ll need to fill in the rest!

4.5 Check False Positivity Rate

Lastly you will implement checkFalsePosRate. This static function determines the false positive rate using a BloomFilter with the provided BFHash, m, and k for a given n value.

The the process for calculating the false positive rate are described below:

- first, add n unique elements to the set/bloom filter.
- next, perform lookups for n unique elements that are not in the set.
- the false positive rate is the ratio of lookups which yield true for n
The false positive rate should be returned as a decimal – for example, a false positive rate of 23% would return 0.23. Lastly, you must use the provided generateUniverse function in your solution.

Once implemented, experiment with different m, k, and n values, and compare results between the two provided hash functions. For example, try a bloom filter with:

- m = 100,000
- n = 10,000
- k = 7

and compare your rates using both the simple and advanced hashes – you should see a considerable improvement!

### 4.6 Observations and Responses

For the final portion of this lab, you will respond to two short answer prompts, writing your comments in the provided observations.txt.

Complete and respond to the following:

1. As you have no doubt seen by now, the AdvancedHash yields a far more efficient distribution in most Bloom Filters, resulting in fewer false positives. Try experimenting further with the AdvancedHash – pick fixed m and n values, and experiment with several values for k.

   What do you observe? Write your observations, comments, and potential explanations in the .txt file.

2. Lastly, in the .txt file describe two practical applications of a BloomFilter – what are some scenarios where a BloomFilter would be ideal and why would it improve upon a more traditional data structure like a Map or Set? You can use hypothetical or real-world examples (hint: don’t be afraid to Google around for some leads!)

### 5 Pre-Lab Questions

After reading this document and provided code, answer the following before we meet (we will discuss in-lab):

1. What does the result of BloomFilter’s toString show? Notice that its returned String doesn’t represent the entirety of bitArray.

2. Again, an object implementing the BFHash interface represents a family of hash functions – in other words a BFHash object will provide us with the k different hash functions that our bloom filter needs.

   But, BFHash only defines one method: getHashCode. How can this method generate k different hash results?

3. Review the generateUniverse method – you can try calling it a few times in the main method (using a small size) and printing the results.

   Why does this function use a while loop instead of a for loop, i.e.: for (int i = 0; i < size; i++)?

### 6 Submission

See the top of this document for the lab’s due date and time. When submitting your code, include only the files listed below:

- BloomFilter.java
- Lab09Tester.java
- observations.txt