Hashing

The submit form for this lab is here: SUBMIT (https://docs.google.com/forms/d/1uSD5B-6NwefKrCLyPSwNiYjTl3a3Y-AY9esAL9WEnyY/viewform?usp=send_form)

Building a Bloom filter

Individual exercise

Today you're going to build a special type of hash table which:

- doesn't store any values, just keys
- doesn't use any type of collision resolution at all
- doesn't guarantee that you get the right answer

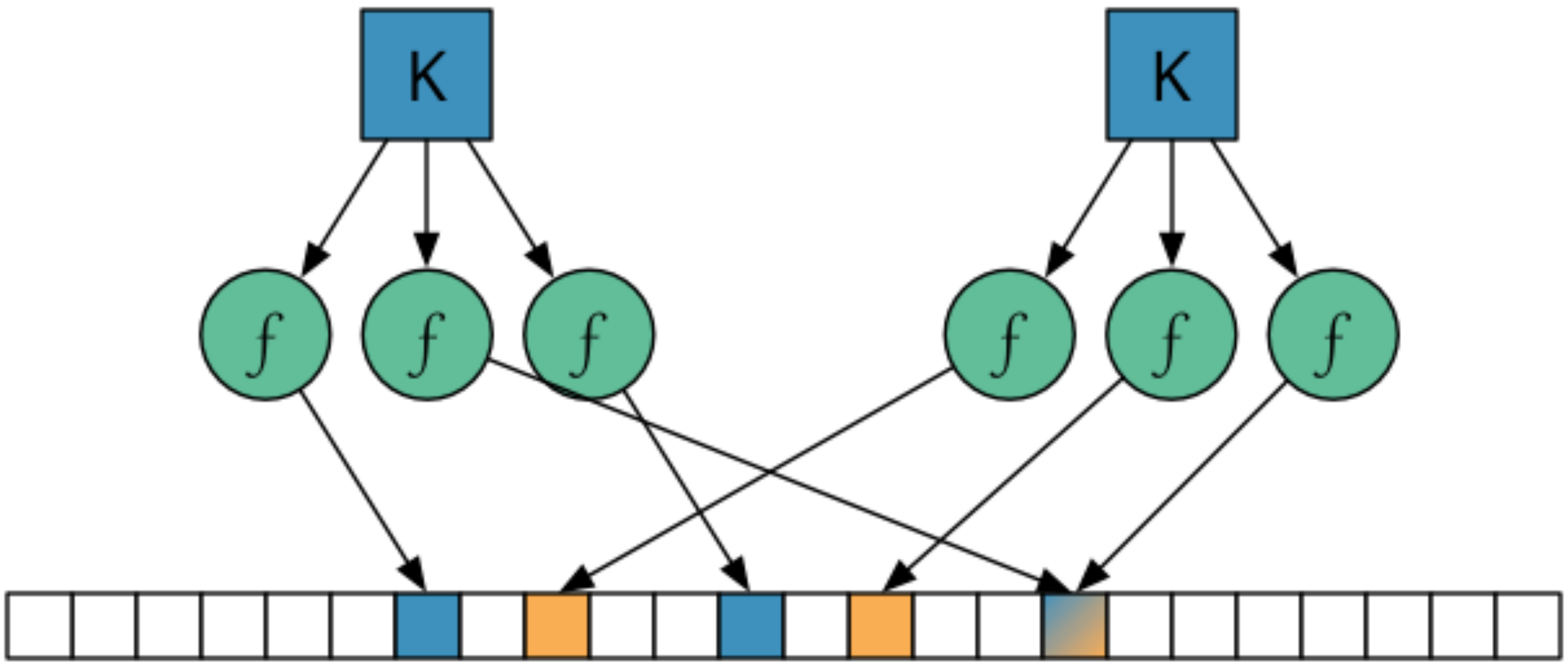
If this sounds strange, it should, but in practice, this device finds applications all over.

The data structure is called a Bloom filter (http://www.eecs.harvard.edu/%7Emichaelm/postscripts/im2005b.pdf), after Burton Howard Bloom. The purpose of a Bloom filter is to tell you, with high likelihood, whether a key has been added to a table or not. This is more or less equivalent to a hash table whose values are only True or False, but it's also sometimes called "set inclusion". Importantly, we will consider the case where it's okay to return wrong answers some small fraction of the time, but only in certain cases. We assume that it's okay (but obviously not preferable) to return "True" if the key is not actually in the table, but it is *not* okay to return "False" if the key is indeed in the table. This is the reason Bloom filters are called filters: they typically find application as a pre-select stage for detecting some behavior. Wikipedia (https://en.wikipedia.org/wiki/Bloom_filter#Examples) has a reasonable handful of examples, but in general they are useful when a large amount of data needs to be checked with an expensive test. The filter drastically reduces the number of checks that need to be made while still guaranteeing that no true positives slip through.

Back to top ↑

Principle of operation

Bloom filters are actually quite simple. We start with a table of bits, all set to zero. When we want to add a key to the table, we use K hash functions to generate K indices into the hash table. For each of these locations we set the bit to one (*without* checking first). To check if something is in the table, we use the exact same procedure, except we read the bit instead, and if *ALL* the bits are set, we say that the key was indeed in the table.



Why does this work? The basic idea is that while you might get unlucky and collide between two keys with one hash function, the chance that you'll collide on K hash functions is much, much smaller. Moreover, notice that we store only K bits per key—we don't store the key OR the value at all! This is the approximate part of Bloom filters: you can't guarantee that if your filter returns "true" that the key is actually in the table. If you got unlucky when checking a key you never added and collided K times with other bits set by keys you did add, then you'll get a false positive.

Despite the drawbacks, Bloom filters have some nice properties. Because an "element" in our table is only one bit, storing K elements per key can actually be a lot better use of space than storing a whole key. Plus, we never actually do a key comparison on lookup, so as long as your hash functions are efficient, your Bloom filter will be efficient.

As we'll see, there are some tradeoffs, but in general, Bloom filters are a pretty efficient way to implement approximate set inclusion, both in space and time.

Back to top ↑

Task 0: Starting your bloom filter implementation

We'll be doing this lab in C. To get you started, we've been nice enough to give you a header file with function prototypes for everything that you need to implment. Download it and create a `bloom.c` file to complement it. You'll be writing your code in there, and once you're done, we'd like you to check in both the `bloom.h` skeleton and your `bloom.c`

You can either clone the cs207 class repo or just copy-paste it from here: bloom.h (https://github.com/iacs-cs207/cs207/blob/master/labs/bloom.h)

Note: if you are not familiar with some of the C features used in the header (like `typedef` or `#define`) it's okay to spend a minute or two on Google to get acquainted.

Back to top ↑

Task 1: Implement a family of hash functions

Finding an arbitrary number of hash functions seems like a daunting task, but there's a clever trick that was discovered which simplifies the problem drastically. It works like this: assume that you already have two reasonable hash functions `h1` and `h2` . (Finding just two hash functions is not that hard). If `h1` and `h2` uniformly spread keys across a given number space, then it's been proven that the family of hash functions defined by `h1(k) + i*h2(k) mod P` for `i=0, 1, 2, ...` also has spreads keys uniformly and, more importantly, is non-interfering for all values of `i` . Here, `P` is the size of your hash table, since hash functions must map keys into an index that exists in your table.

So your first task is to implement the two functions `hash1` and `hash2` . You can choose any hash functions you want, but in order for your filter to work well, I'd suggest you use something non-trivial. You're welcome to use Google for this: we don't mind if you use someone else's algorithm, so long as you adapt it for your purposes here. If you do, please make sure you have appropriate license permission to use it and cite the author in the code.

Note that these functions take a `bloom_filter_t` as an argument. All hash algorithms must, at some point, ensure that a hash function produces a value which is less than the size of their storage space. Some authors choose to build that into their hash function. This was the `mod P` part of the equation above: it makes sure that the hash function always lands somewhere in the table. We leave it up to you whether you want to do your modulo operation within the hash function (i.e.- inside `hash1` and `hash2`) or outside of it (i.e.- inside `bloom_add` and `bloom_check`). If you choose the latter, you can safely ignore the `B` argument to the hash functions.

To show us that your hash functions are working, please evaluate both `hash1` and `hash2` on the integers 0, 1, 2, 3, 13, and 97. If you chose to do the modulus inside the hash function, you can manually create a dummy `bloom_filter_t` with size 100 to test with.

Back to top ↑

Task 2: Get bit operations working

As we described, Bloom filters operate on *bits* in order to save space. This means that when we get a value from the hash function, we'll eventually be using it as the index of a particular bit. This is a little awkward, since C doesn't have native bit types. Instead, we'll be writing a pair of functions to manipulate bits in a packed array.

The goal here is to start with an array of 64-bit unsigned integers and write a function which can set or retrieve exactly one bit in that array. This requires two pieces:

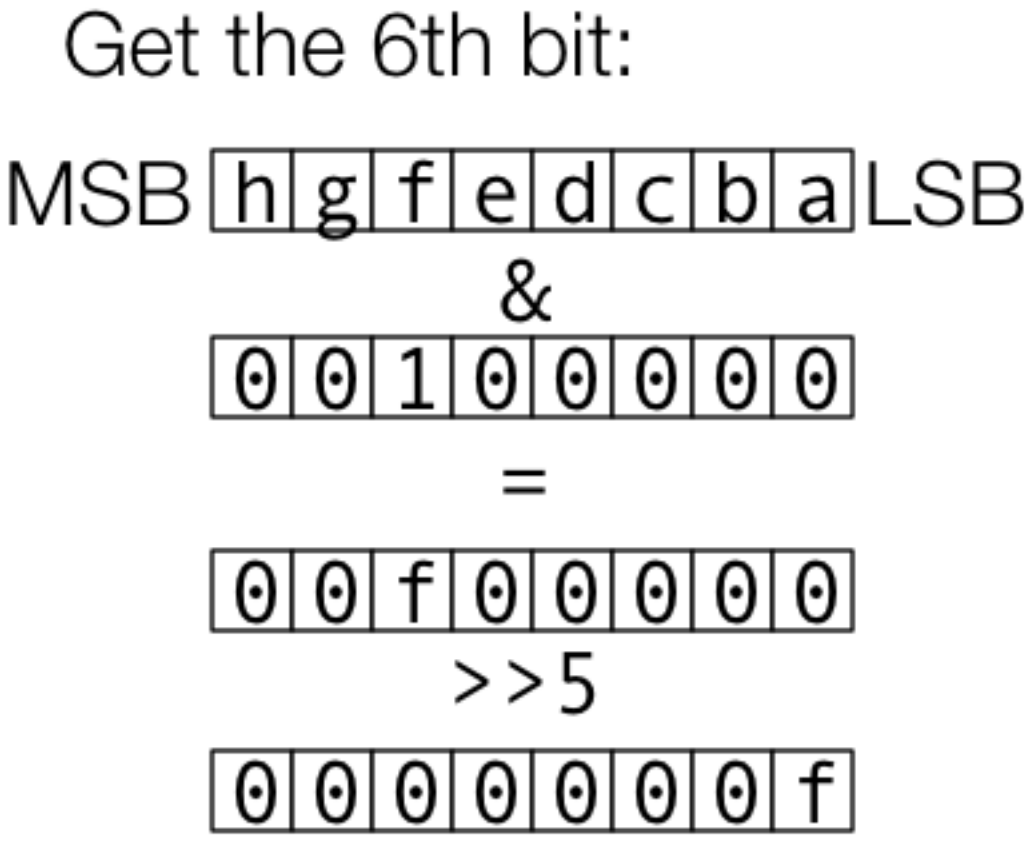
- Find the right integer.
- Get or set the right bit within that integer.

Ultimately, you can implement this any way you want, but we'll give you an efficient method if you want to use it.

Let's say we want to choose the 67th bit out of an array of 70 bits.

First, realize that 70 is not a multiple of 64, so our array actually consists of *two* 64-bit integers. We want the 67th bit, and from above, the first step is to find correct integer. This is as easy as dividing our index by our word size (in this case, 64). Because 64 is also 2^6, we can do this quickly in C using the right-shift operator: `index >> 6`. Notice that we're *integer division*, meaning no fractions: 67/64 = 1. So we want the first integer, and we can get it easily using C indexing: `table[1]`.

Now we want the correct bit inside that integer. This is easy, too: first we take the index modulo 64, which tells us the index of the bit inside the integer. Because we can't index bits in C, we use bitwise operators again. Let's consider the retrieval case (you can figure out the bit setting case on your own). Recall that bitwise AND of an arbitrary integer and a number with all its bits set returns the original integer (this is the mathematical identity of bitwise AND). In C, that might be expressed like this: `a & ((uint64_t)-1) == a`. Likewise, bitwise AND of an arbitrary integer and zero is zero: `a & 0 == 0`. So if we want to "select" and integer, an easy way is to *mask* the number and then *shift* the bit to the least significant bit position. In other words, we want to bitwise-AND our table integer with a number that has zeros in every place except the bit index, and then we want to shift the result down to put that bit in the 1's place.



Task 3: Complete the Bloom filter API

Now that you've gotten the components out of the way, the rest is relatively straightforward.

The first two functions, `bloom_init` and `bloom_destroy` are bookkeeping. Even though C is not syntactically an object-oriented language, you can use it as one if you like. In our case, we're effectively treating `bloom_filter_t` as an object and the `bloom_` functions as its methods. You can think of `bloom_init` and `bloom_destory` as constructor and destructor, then. We don't have syntactic encapsulation, but we still gain the semantic and organizational benefits of the object model.

Since we've given you the data structure, the `size` and `table` elements should be clear from what you know about bloom filters. All you need to do is set the values in the `bloom_filter_t` data structure to reasonable values and allocate/deallocate memory for the table. Just remember that `size` is measured in *bits*. The `count` field will be used to track how full the hash table is, so initialize it to zero.

`bloom_add` and `bloom_check` should add a key and check for a key. From the description before, you should remember that each of these will be evaluating multiple hash functions from the family you designed. Specifically, we ask you to evaluate `N_HASHES` number of hash functions, so you'll need to produce and set `N_HASHES` number of bits in the table for each key.

You should probably write some quick tests to make sure your data structure is working as intended. If something is broken, don't be afraid to use the debugger! That's what it's there for. Recall that you need to compile your program with the `-g` option to get it to work.

Back to top ↑

Task 4: Evaluating your filter

Now comes the fun part: trying it out.

Smoke test:

Let's make sure it works first.

1. Create a Bloom filter with 1000 elements (bits).
2. Add the first 70 positive integers to your filter.
3. Write a quick function that counts the actual number of bits set in your table. (This doesn't need to be clever, a brute force loop calling `get_bit` is fine.)

We use three hash functions and 70 numbers, so if we didn't have any collisions, we would see 210 bits set. In reality, you should expect less than that due to collisions. If you chose rather simplistic hash functions, you might see *substantially* less.

Please report your results on the form.

Understanding the trade-offs

In Bloom filters, there is tension when choosing the number of hash functions to use. With a small number of hash functions, collisions are more likely to directly lead to false positives, but as the number of hash functions grows, the number of bits set grows, which leads to more collisions.

While there is a bunch of neat theory about this area, we'd like you to look at it experimentally.

First, write a function that generates an array of 100 random number between 0 and 1000000. You might find the `rand()` function helpful. To learn about it, either use Google or type `man 3 rand` on a Unix system. Don't worry about the randomness of the function—it will be good enough for us.

Next, we'll write a new function for core of our experiment. The input should be *two* arrays of 100 numbers, just like the ones generated from the previous function. First, have the function create a new Bloom filter with 1000 elements and add all the elements of the first input array to it. Second, create a loop that counts the number of bits, just like your smoke test. Finally, create a loop that checks whether the numbers in the *second* array are in the table. Recall that we never added them, so if our hash table was perfect, there would be a very small number of collisions (equal to the number of expected collisions between two sets of 100 random numbers from 0-1000000). In reality, there will be many more, because of the way Bloom filters work. Count up and print out how many of the numbers in the second array returned true.

If you run this experiment function, you should get two numbers printed, the number of bits set (also called the occupancy), and the number of random numbers which were not added but returned true anyways (we'll say this is a reasonable approximation of the false positive rate).

Now we're going to run five experiments:

1. Change the value of `N_HASHES` to 1.
2. Recompile your program and run your experiment function.
3. Write down the two numbers function printed.
4. Repeat steps 1-3 using values of 2, 3, 4, and 5 for `N_HASHES`.

Go ahead and submit these numbers on the Google form and give a short description of the relationship you see between the number of hash functions and both the occupancy rate and the false positive rate. (Something as simple as "as X goes up, Y goes..." or "X and Y are correlated is fine.)