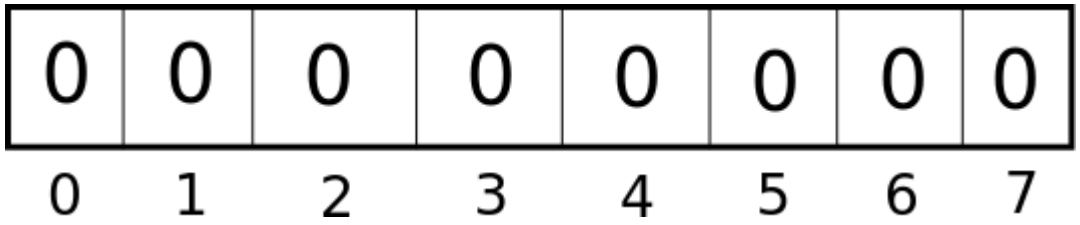
Bloom Filters

There is another common data structure, closely related to Hashtables, which is often used for sets (but not maps!) known as a Bloom Filter. Like Treaps, Bloom Filters are both probablistic and very easy to implement. By implementing a Bloom Filter, you'll gain a better understanding of how Hashtables work, you'll learn how to call cryptographic hash functions from within Python, and you'll learn a new data structure which is commonly used (for example, Google keeps a set of web sites which may introduce malware to a system, so that it can warn you before you go to such a page; it does this using a Bloom Filter).

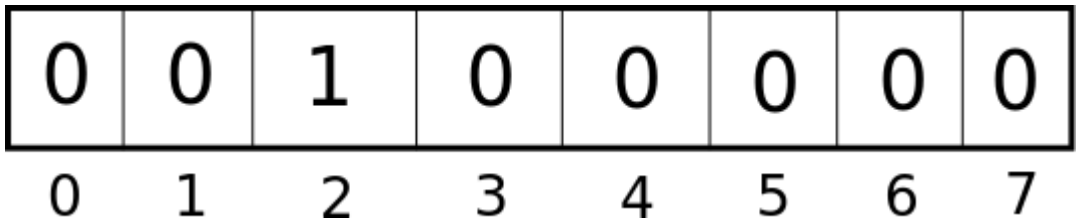
Suppose you wanted to implement a set of strings (such as, like Google, URLs) using a Hashtable. You could do this! You would hash your string, and then insert that key into your array at the resulting index. If there were collisions, you would handle those using either separate chaining or open addressing. Then, to make your contains method work, you would again hash the string, and see if the linked list at that index contains your key (or, if doing open addressing, move along the array until you find your key or see an empty location). That would work great! The problem is, if this is a big set, that is a large array, in which you would store a lot of space-eating strings.

How Bloom Filters Work

Bloom Filters solve this problem by using up *much* less space, but at the expense of not always being correct. In other words, Bloom Filters will tell you either "No, that element is definitely not in the set," or "Yes, that element is probably in the set, but I might be wrong." It does this by NOT storing the strings themselves, but instead by storing whether or not an element has been hashed to a given index. In other words, a Bloom Filter is just an array of bits! Here's an empty Bloom Filter of size 8:

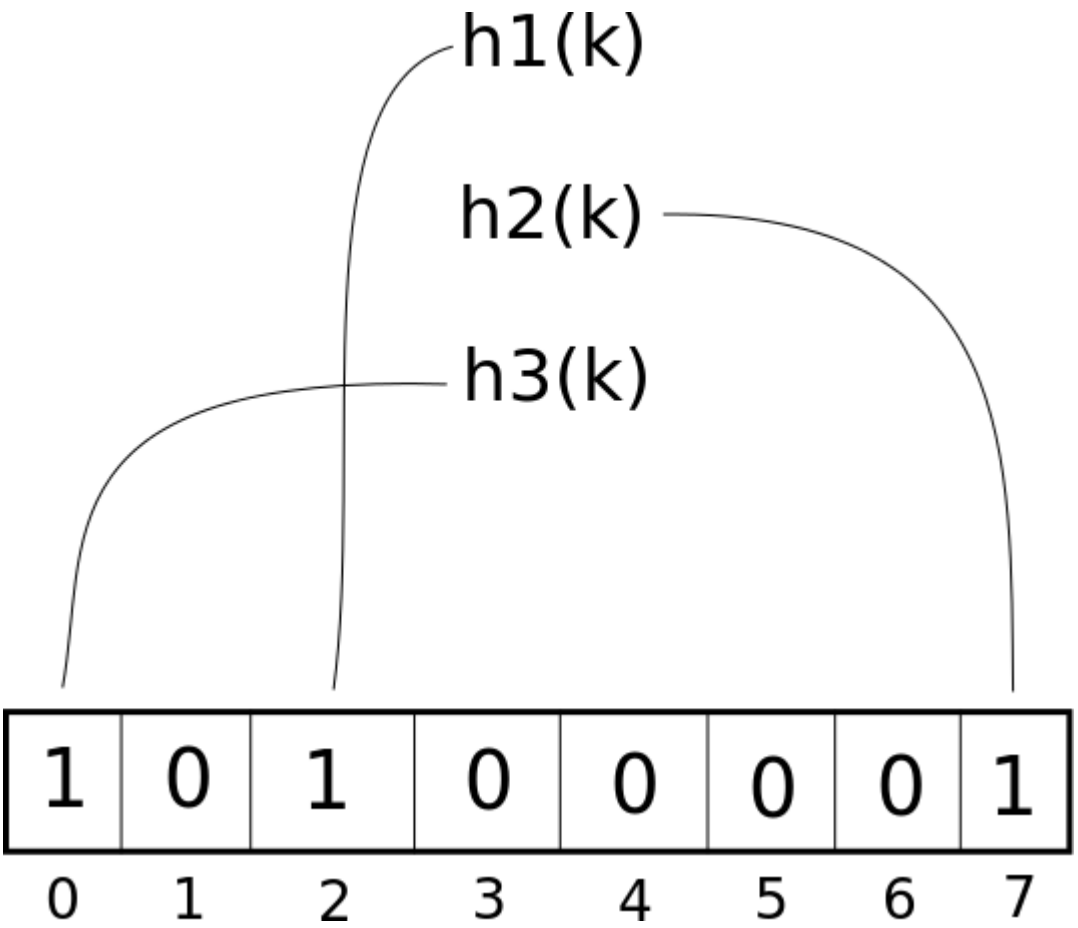


Suppose we then insert a single key, "hello", which hashes to index 2. Well, we would turn that False at index 2 to a True:



Now, if we wanted to know if "hello" was in the set, we would again perform the hash, get index 2, and then see if that element was a True. If it was a False, that element had definitely not been added to the set. If it was True, it either had been added to the set, or some other key had collided at that index, turning it True.

To decrease the chances of that happening, we hash each key several times, using different hash functions, and turn all of those indices to True. For example, suppose we start with an empty Bloom Filter, and use three hash functions:



Here, we've used three hash functions, each of which have given us a different index. So, now suppose we do a contains() on the same key: we'll get the same three indices, and see that all three indices store a True. Therefore, the element is probably in the set. On the other hand, say we do a contains() on a different element. All three hashes would have to collide before we erroneously reported that this element was in the set. Could it happen? Sure. But it won't happen often, as long as the array is long enough.

How long do we need to set our array? Analysis (https://en.wikipedia.org/wiki/Bloom_filter#Probability_of_false_positives) shows that to store n elements you need $-1.44 \log_2 p$ spaces in your array, where p is the false positive rate you want to achieve. For instance, if we want 99% accuracy, we should have $-1.44 \cdot \log_2 .01 \approx 10$ times the number of bits as we have elements to store.

Cryptographic hash functions in Python

So, we're going to need a bunch of hash functions. One (non-cryptographic) hash function for strings is built in, and you merely need to call `hash(k)`. But what about the other two?

Though our hash functions here don't need to be cryptographically secure, it doesn't hurt, either. So what about MD5? Or SHA512? Well, the python module hashlib (https://docs.python.org/3/library/hashlib.html) provides all of these for us, so we don't have to implement them ourselves. Below is an example of how this can be done for the MD5 hash and the string "hello":

```
aKey = "hello"
hashObject = hashlib.md5() #Builds an MD5 object
byteString = aKey.encode() #.encode() turns a string into an array of ints
hashObject.update(byteString) #This adds the string to the input to the hash function
hex=hashObject.hexdigest() #This actually computes the hash, and outputs the result
                           #as a hex string
number = int(hex, 16) #This casts the hex string to an actual integer
                      #The 16 is the base of the number in the input string
#number is now the hash of "hello"
```

For the other two hashes, please use MD5 and SHA512.

The assignment

Step 1 (85/100): Implement a Set of strings with a Bloom Filter. This means you'll make a file called `bloomFilter.py` with a class called `BloomFilter`, which has a constructor, which accepts an int representing the size of the array as an argument. This constructor should create a list of Falses, which is that input size. Additionally, it has a method `add(self,k)` which adds the string `k` to the set, and a method `__contains__(self,k)` which returns true or false, representing whether `k` is in the Set.

Step 2 (100/100): To save even more space, Bloom Filters are usually not actually implemented with an array of booleans, each of which needlessly take up (at least) an entire byte. Instead, this entire bit array is implemented as a single integer which, when printed out in binary, displays the sequence of Trues and Falses as 1s and 0s.

To do this, you'll need to be able to change and check individual bits in a number. You can do this with bitwise operators (https://wiki.python.org/moin/BitwiseOperators). As an example, suppose we have a variable `aNum`, and we'd like to set the third bit. First we create a number with the third bit set: `thirdBit=1<<2` (shift a 1 two bits over, so it's 100), then do a bitwise or: `ans = aNum | thirdBit`. `ans` will have its third bit set to 1. Reading a bit requires a similar operation, but with a bitwise and.

Replace your list of booleans with an int, and use bitwise operators to change or check bits when needed.

Name this file `bloomFilterBits.py`.