Programming Assignment:
Bloom Filters in Python

Testing for membership is an important problem in computing. Whether in deduplication in storage systems or networking devices or in such mundane things as implementing sets efficiently. In 1970, Burton Howard Bloom invented a probabilistic data structure that can test for membership in $O(1)$ time, albeit at the cost of storage.

A Bloom filter has at its core a large array of bits. (We will implement it using a Boolean numpy array.) In storage systems, I have seen Bloom filters with 2GB worth of bits used. Initially, all values of the bit array are set to zero / false. When an element is inserted, we calculate a number of different hashes $h_1, h_2, h_3, \ldots, h_n$ of the element and flip those elements to one / true. When we want to check for membership of a potential object x and calculate the values of these different hashes. If all of them are true, i.e. if

$$\text{array}[h_1(x)] = \text{array}[h_2(x)] = \ldots = \text{array}[h_n(x)] = \text{true}$$

then we decide that x is present. If however one of the bits is not true, then we know for sure that x is not present.

Implementation

There are several ways to implement bit-arrays in native Python, but for convenience, we are going to use Numpy arrays. Numpy and its sibling Scipy were developed to use Python for fast, numerical calculation. One of its strong points is having a true array. The native Python list uses up a lot of space to store the type of each object in a list and processing a list can be slow because Python queries the type information in order to select the right operation.

First, you need to install numpy unless you already have it installed, e.g. by using anaconda. If you have a modern Python version, this can be done via

```
pip3 install numpy
```

We can now import numpy

```
import numpy as np
```

and define our Bloom filter using

```
self.array = np.zeros(LENGTH, dtype=bool)
```

This will create a boolean array. We access its elements with the usual bracket notation, as in

```
self.array[100] = True
```

We also need a number of hash functions. For this exercise, we only insert integers. We can think of them as the identifiers of records. Python has a good library of cryptographically secure hash functions, but for our purposes, we do not need their security. Python also has a native hash function that it uses for internal purposes, but this is not useful. Instead, we are going to use an old and now disreputable technique of using the middle bits of a square. Also, the hash function need to return an index into the array. We choose array length that are power of twos and use a mask of one bits in order to make sure that the result of the hash is an index. Here is an example:

```
def hasher2(ix):
    return (  (17377*ix**2>>4)&MASK,
                ((17377*(ix+5)**2)>>8)&MASK,
                ((10607*(ix+7)**2)>>2)&MASK  )
```

I am using an array of length 1024. Indices should be 10 bits long, therefore I am masking my hash results with a mask `MASK = 0x3ff` because this mask in binary is `b0011 1111 1111`. For my hash function, I take the key, square it, multiply it be a moderately large prime and then shift to the right. This function will generate a tuple of three hash values between 0 and 1023.

Here is the definition of my Bloom filter:

```
LENGTH = 1024
MASK = 0x3ff

def hasher2(ix):
    return (17377*ix**2>>4)&MASK, ((17377*(ix+5)**2)>>8)&MASK,
((10607*(ix+7)**2)>>2)&MASK

class Bloom:
    def __init__(self, LENGTH, MASK, hasher):
        self.length = LENGTH
        self.mask = MASK
        self.array = np.zeros(LENGTH, dtype=bool)
        self.hasher = hasher
```

I can test the Bloom filter by inserting 100 values into it (which sets up to 300 bits to True) and then taking different numbers and see whether the Bloom filter correctly shows that they have not been presented.

```
my_bloom = Bloom(LENGTH, MASK, hasher2)
for i in random.sample(range(my_bloom.length), 100 ):
    my_bloom.insert(i)
for i in range(30):
    x= random.randint(0,10000000)
    print(x, my_bloom.look_up(x))
```

When I insert into a Bloom filter, occasionally, I will set a bit to True that was already True. Therefore, the number of bits flipped is not equal to the product of the number of hash functions and the number of integers inserted. Because I have almost 300 bits set, my chance of a random bit being set is less than $p = \frac{300}{1024}$. That all of a collection of three random bits are set happens with probability $p^3 = 0.0251457$. If I test 30 numbers, then I expect $30p^3 = 0.754371$ false positives, and in this experiment, I indeed found one false positive, where the number was not inserted, but the Bloom filter said that it was probably inserted. If you repeat the experiment, your numbers will differ.

Bloom filters are therefore probabilistic. There is a chance that an element that is not present is diagnosed with being present. We can control this by the size of the array and by the number of hashes. Relatively small number of hashes suffice to reduce the number of false positives to something acceptable.

Your Task:

Finish the implementation of the Bloom filter. Change the length to 4096 and therefore the mask to `0xfff`. Change the number of hash functions to 5 (using a prime number table to find factors and trying out some shifts). Then insert 200 random integers into the Bloom filter. Finally, generate 100000 random integers and see whether the Bloom filter falsely indicates that they are not contained in it. Obviously, you just need to count the number of times that your Bloom filter says True (as we know that it should not). Estimate the false positive rate from this. If you want to, you can calculate the probabilities exactly as well.

Your results will depend on the quality of the hash functions, and frankly, these are not very good hash functions. However, they are fast, and as you will see, they work reasonably well.

Deliverables:

(1) Your complete Python code.
(2) A summary of your findings, i.e. a false positive number.

You need to implement two methods in Bloom, namely

```
def insert(self, x):
def look_up(self, x):
```

My implementation has two and four lines of code. You also need to redefine the hasher2 function.

If you know Python, this exercise might take 15 minutes.