Daily Coding Problem #301

Problem

This problem was asked by Triplebyte.

Implement a data structure which carries out the following operations without resizing the underlying array:

- `add(value)`: Add a value to the set of values.
- `check(value)`: Check whether a value is in the set.

The `check` method may return occasional false positives (in other words, incorrectly identifying an element as part of the set), but should always correctly identify a true element.

Solution

While there may be multiple ways of implementing a data structure with these operations, one of the most well-known is called a Bloom filter. A Bloom filter works by hashing each item in multiple ways, so that several values in the array will be set to `True` for any given input. Then, when we want to check whether a given value has been added, we examine each of the locations that it can be hashed to, and only return `True` if all have been set.

To give a simple example, suppose we are dealing with an underlying array of size `100` and have two functions which take in integers as input, defined as follows:

```
def h1(value):
    return ((value + 7) ** 11) % 100

def h2(value):
    return ((value + 11) ** 7) % 100
```

Now suppose we added 3 and 5 to our set. This would involve the following operations:

```
location1 = h1(3) # 0
location2 = h2(3) # 4
array[location1] = array[location2] = True

location1 = h1(5) # 88
location2 = h2(5) # 56
array[location1] = array[location2] = True
```

For most cases this will not cause any problems. However, look what happens when we check 99. Since `h1(99) = 56` and `h2(99) = 0`, both values we check in the array would already be assigned `True`, and we would report this as being in our set.

Although we can reduce the likelihood of this occurring by using more optimal hash functions, and by increasing the initial array size, we cannot get around the fact that a Bloom filter will occasionally return false positives. For this reason it is called a probabilistic data structure.

```
import hashlib

class BloomFilter:
    def __init__(self, n=1000, k=3):
        self.array = [False] * n
        self.hash_algorithms = [
            hashlib.md5,
            hashlib.sha1,
            hashlib.sha256,
            hashlib.sha384,
            hashlib.sha512
        ]
        self.hashes = [self.get_hash(f) for f in self.hash_algorithms[:k]]

    def get_hash(self, f):
        def hash_function(value):
            h = f(str(value).encode('utf-8')).hexdigest()
            return int(h, 16) % len(self.array)

        return hash_function

    def add(self, value):
        for h in self.hashes:
            v = h(value)
            self.array[v] = True

    def check(self, value):
        for h in self.hashes:
            v = h(value)
            if not self.array[v]:
                return False

        return True
```

In the implementation above we maximize the number of hash functions at five, and use built-in cryptographic algorithms that come with Python's `hashlib` library.

Since the number of hashes we must perform for each value is bounded by a constant, and each hash is `O(1)`, the time complexity of each operation will also be constant. The space complexity will be `O(N)`, where `N` is the size of the underlying array.