Overview

Bloom filters are probabilistic space-efficient data structures. They make a trade-off between space and false-positive ratio when assesing membership of an element to the dataset represented by it. As they were originally conceived, they only support two operations, insertion and search. Other operations such as removing (an element) can be implemented in ingenious ways but these are oftentimes not straightforward. ¶

These properties make bloom filters useful when the number of objects to be stored in a set is very large and we don't care about a small chance of false positives. The classical example of this is to check for misspelled words: a bloom filter representing a dictionary takes a lot less space than storing a list of all the words and it will never give you a false negative, only a few false positives.

The false positive ratio can be controlled by changing a few parameters: m, the size of the bloom filter; n, the number of items to be stored; and k, the number of hash functions that our filter uses. It is theoretically estimated to be equal to the quantity given by (1 - (1 - 1 / m) ^ (kn)) ^ k

Implementation

In the cell below, you can see my implementation of a bloom filter. I used the fast and robust mmh3 hash function, with a different seed for each of the hash functions that one may decide to use.

In [6]:
```
# Here, I create the class bloom_fiter. Calling bloom_filter(m, k)
#will create a bloom filter of size m that uses k hash functions.

import bitarray, mmh3

class bloom_filter:
    def __init__(self, size, hashes):
        self.size = size
        self.hashes = hashes
        self.bit_array = bitarray.bitarray(size)
        self.bit_array.setall(0)

    def add(self, k):
        for i in range(self.hashes):
            index = mmh3.hash(k, i) % self.size
            self.bit_array[index] = 1

    def search(self, k):
        for i in range(self.hashes):
            index = mmh3.hash(k, i) % self.size
            if self.bit_array[index] == 0:
                return False
        return True
```

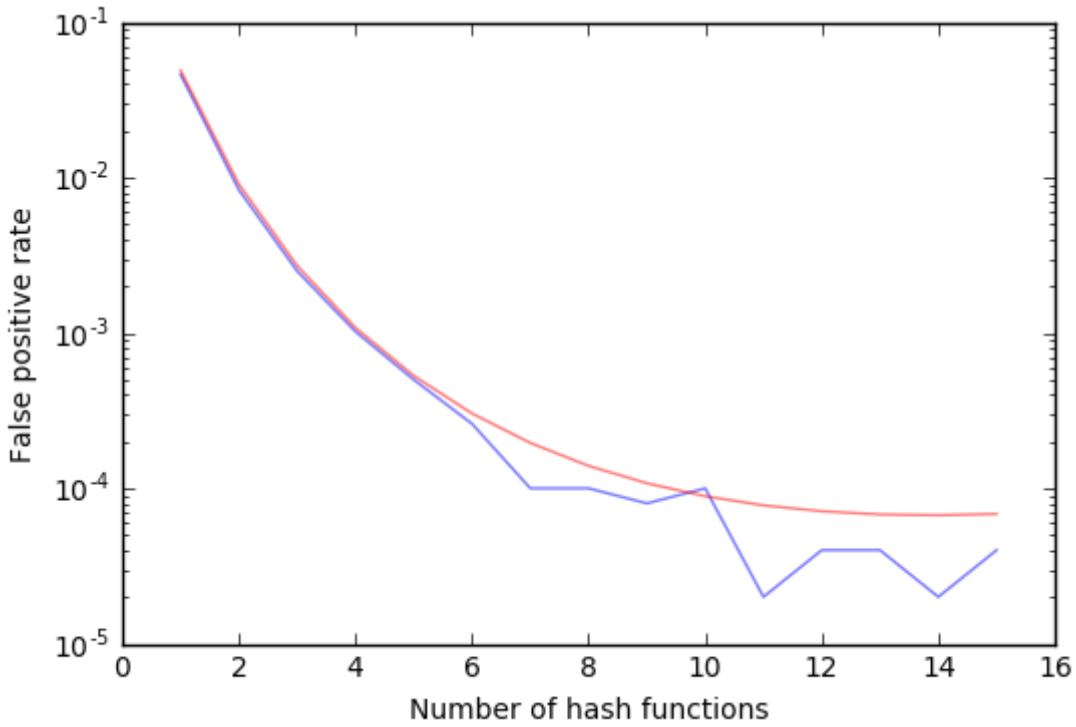
In [12]:
```
# In this section, I create a dataset and a test dataset.
# All the values in dataset go to a bloom filter of size 1,000,000.
# Then, for each k = 1, 2, ..., 15 I compute the false positive ratio by
# assesing membership of each element in the test dataset to
# the (original) dataset and to the bloom filter.

import numpy, random

dataset = numpy.random.randint(0, 1000000, 50000)
test_dataset = numpy.random.randint(0, 1000000, 50000)

false_positive_ratios = []

for hashes in range(1,16):
    false_positives = 0
    bloom = bloom_filter(1000000,hashes)
    for i in dataset:
        bloom.add(i)
    for i in test_dataset:
        if bloom.search(i) == True and not i in dataset:
            false_positives +=1
    false_positives /= len(test_dataset)
    false_positive_ratios.append(false_positives)

print(false_positive_ratios)
```

```
[0.04594, 0.00832, 0.0025, 0.00102, 0.0005, 0.00026, 0.0001, 0.0001, 8e-05,
 0.0001, 2e-05, 4e-05, 4e-05, 2e-05, 4e-05]
```

In [30]:
```
# In this section I compute the expected false positive rate for each of
# the choices of k.

import math

theoretical_false_positive_ratios = []
for i in range(1, 16):
    fpr = (1 - (1 - 1 / 1000000) ** (i * 50000)) ** i
    theoretical_false_positive_ratios.append(fpr)

print(theoretical_false_positive_ratios)
```

```
[0.048770599281404814, 0.009055925617231994, 0.00270258490585659, 0.001079685
5937625928, 0.0005295645215611054, 0.0003031293019801278, 0.00019587003914131
652, 0.0001395538235340495 2, 0.00010774363132422082, 8.894276884918259e-05,
 7.768060981865322e-05, 7.116993848635345e-05, 6.792410876252639e-05, 6.71374
0582097493e-05, 6.838911116490114e-05]
```

In [33]:
```
#Finally, I produce a graph with the theoretical false positive rate in red
# and the computed false positive rate in blue.

import matplotlib.pyplot as plt
fig = plt.figure()
ax = plt.gca()
ax.plot(range(1,16), false_positive_ratios, c='blue', alpha=0.5)
ax.plot(range(1,16), theoretical_false_positive_ratios, c='red', alpha=0.5)
ax.set_yscale('log')
plt.xlabel("Number of hash functions")
plt.ylabel("False positive rate")

show()
```



The graph above shows the relation between k, the number of hash functions and the false positive rate. In red, you can see the theoretical relationship while in blue, you can see the relationship computed by my code above.

One of the wonders of my implementation (and bloom filters in general) is that the access time does not depend on the number of items stored, instead, it only depends linearly on the number of hash functions used (k); if the hash functions take a constant and small time to be computed, which is the case here, then our access time is bound to be pretty low.

As for memory size, since the bloom filter is entirely represented by a bitarray of size m and this is invariant, the memory size occupied does not change as one scales the number of items to be stored.

In [ ]: