In [1]:
```
!pip install bitarray
!pip install mmh3
```

```
Requirement already satisfied: bitarray in c:\users\hatef\anaconda3\lib\site-packages (2.5.1)
Requirement already satisfied: mmh3 in c:\users\hatef\anaconda3\lib\site-packages (4.1.0)
```

In [2]:
```
import mmh3
import math
import numpy as np
import sys
from bitarray import bitarray

class BloomFilter:
    """ Implements a Bloom Filter for approximate set membership queries. """

    # Class-level constant for 128-bit maximum integer
    _max_128_int = pow(2, 128) - 1

    def __init__(self, n=10000, delta=0.01):
        """
        Initialize a Bloom Filter.
        n: Maximum number of elements to be inserted.
        delta: Desired false positive rate.
        """
        # m = size of the bit array
        self.m = int((math.log2(1/delta)*n)/math.log(2))

        self.bit_array = bitarray(self.m)
        #print(sys.getsizeof(self.bit_array))

        self.bit_array.setall(0)

        #number of hash functions
        self.k = int(math.log2(1/delta))

    def _hash(self, token, seed):
        """
        Compute the hash of a token using the given seed.
        Maps the hash value to an index in the bit array.
        Use different seed values to simulate different hash functions.
        For example, if you need 10 hash functions, you can use 10 different seeds 1, 2, 3, ..., 10.
        mmh3.hash128("foo", 1)
        mmh3.hash128("foo", 2)
        ...
        """
        x = mmh3.hash128(token, seed, signed=False)/self._max_128_int

        return int(x*(self.m-1))


    def insert(self, x):
        """ Insert an element into the Bloom filter. """
        #Setting hash_i(x) bit of the bit array to 1
        for i in range(0, self.k*2, 2):

            #Passing the token and the seed to hash function
            h = self._hash(x, i)

            #setting the correct bit to 1
            self.bit_array[h] = 1
        return


    def membership(self, x):
        """
        Check if an element is likely to be in the set.
        Note: There can be false positives.
        """
        res = True

        #checking that for every hash function(every seed), the corresponding bits are 1
        #return true if all the bits are set, false otherwise
        for i in range(0, self.k * 2, 2):
            h = self._hash(x, i)
            if self.bit_array[h] != 1:
                res = False

        return res

    def size(self):
        """ Return the size of the bit array (i.e., filter). """
        return self.m
```

In [3]:

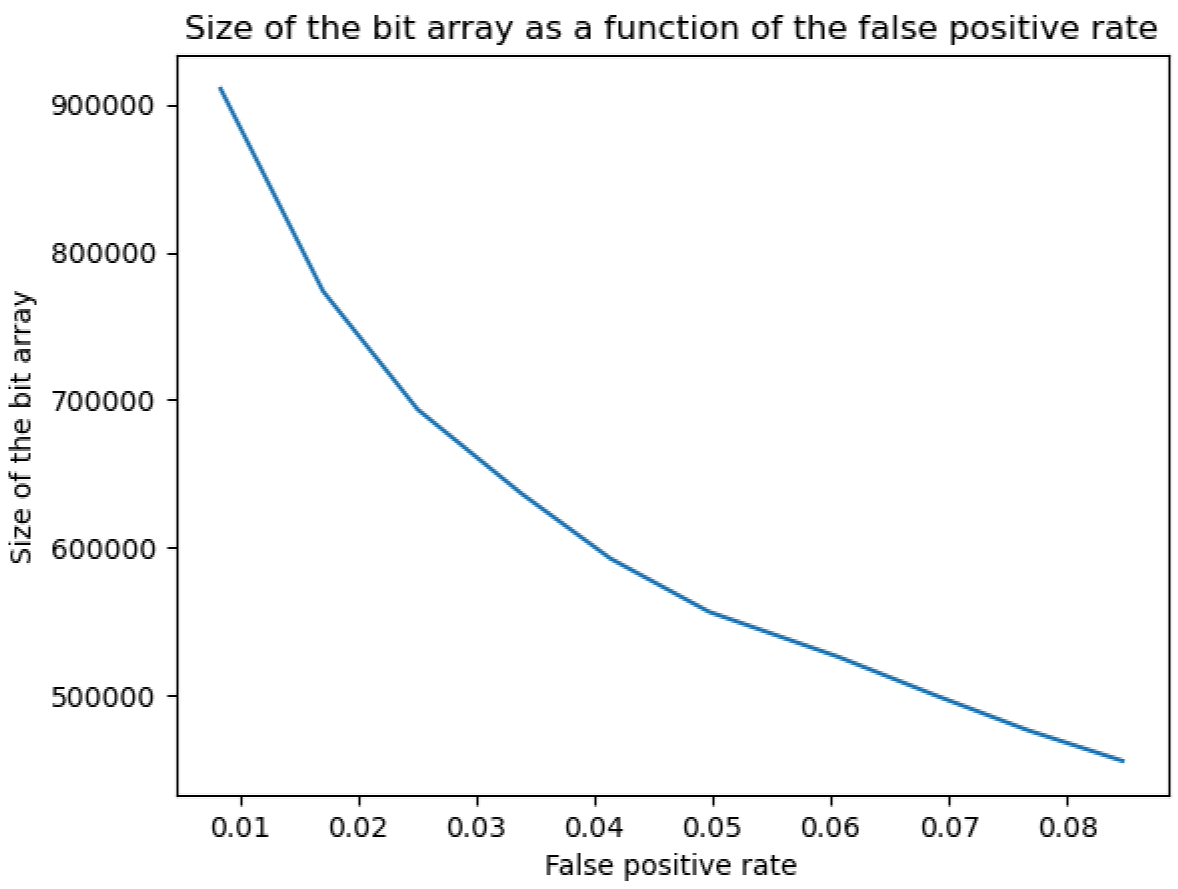
```
# there are about 95k phishing urls to be inserted
n_insertions = 95000
delta = 0.01
filter_size = []
rates = []
```

```python
# try different false positive rates
while delta <= 0.1:
    n_true_positives = 0
    n_positives = 0
    n_queries = 0
    B=BloomFilter(n=95000, delta=delta)

    filter_size.append(B.size())

    with open('phishing.csv', 'r', encoding='utf-8') as f:
        for line in f:
            # insert the url into the bloom filter
            B.insert(line)
            n_true_positives+=1

    with open('urls.csv', 'r', encoding='utf-8') as f:
        for line in f:
            # check if the url is in the bloom filter
            # increment n_positives and n_queries
            n_queries+=1
            if B.membership(line) == True:
                n_positives+=1


    # calculate the false positive rate
    false_positive_rate = (n_positives-n_true_positives)/n_queries
    rates.append(false_positive_rate)
    print(f"False positive rate: {false_positive_rate:.5f}")
    delta += 0.01
```

```
False positive rate: 0.00836
False positive rate: 0.01703
False positive rate: 0.02504
False positive rate: 0.03381
False positive rate: 0.04136
False positive rate: 0.04970
False positive rate: 0.06063
False positive rate: 0.06896
False positive rate: 0.07667
False positive rate: 0.08470
```

Plot the size of the bit array as a function of the false positive rate

In [4]:
```
# plot the size of the bit array as a function of the false positive rate
import matplotlib.pyplot as plt

plt.plot(rates, filter_size)
plt.xlabel('False positive rate')
plt.ylabel('Size of the bit array')
plt.title('Size of the bit array as a function of the false positive rate')
plt.show()
```



In [ ]: