Name of the student:	Tanmay Prashant Rane	**Roll No.**	8031
Practical Number:	6	**Date of Practical:**	
Relevant CO's	**At the end of the course students will be able to apply appropriate algorithms for extracting knowledge from given dataset.**		
Sign here to indicate that you have read all the relevant material provided before attempting this practical		**Sign:**	

Practical grading using Rubrics

Indicator	Very Poor	Poor	Average	Good	Excellent
Timeline (2)	More than a session late (0)	NA	NA	NA	Early or on time (2)
Code design (2)	N/A	Very poor code design with no comments and indentation(0.5)	Poor code design with very comments and indentation (1)	Design with good coding standards (1.5)	Accurate design with better coding satndards (2)
Performance (4)	Unable to perform the experiment (0)	Able to partially perform the experiment (1)	Able to perform the experiment for certain use cases (2)	Able to perform the experiment considering most of the use cases (3)	Able to perform the experiment considering all use cases (4)
Postlab (2)	No Execution(0)	N/A	Partially Executed (1)	N/A	Fully Executed (2)

Total Marks (10)	Sign of instructor with date

Practical

Course title: Big Data Analytics
Course term: 2019-2020
Instructor name: Saurabh Kulkarni

Problem Statement: To effectively search a word in huge English dictionary using Bloom Filter

Theory:

A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it's not.

Interesting Properties of Bloom Filters

- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate false negative result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

Working of Bloom Filter

A empty bloom filter is a bit array of m bits, all set to zero, like

0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10

We need k number of hash functions to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions.
We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition of item and checking membership.

Probability of False positivity: Let m be the size of bit array, k be the number of hash functions and n be the number of expected elements to be inserted in the filter, then the probability of false positive p can be calculated as:

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}$$

Size of Bit Array: If expected number of elements n is known and desired false positive probability is p then the size of bit array m can be calculated as :

$$m = -\frac{n \ln P}{(ln 2)^2}$$

Optimum number of hash functions: The number of hash functions k must be a positive integer. If m is size of bit array and n is number of elements to be inserted, then k can be calculated as :

$$k = \frac{m}{n} ln 2$$

Space Efficiency

If we want to store large list of items in a set for purpose of set membership, we can store it in hashmap, tries or simple array or linked list. All these methods require storing item itself, which is not very memory efficient. For example, if we want to store “geeks” in hashmap we have to store actual string “ geeks” as a key value pair {some_key : ”geeks”}.
Bloom filters do not store the data item at all. As we have seen they use bit array which allow hash collision. Without hash collision, it would not be compact.

Choice of Hash Function

The hash function used in bloom filters should be independent and uniformly distributed. They should be fast as possible. Fast simple non cryptographic hashes which are independent enough include murmur, FNV series of hash functions and Jenkins hashes.

Code:

Find words **google** and **apple** in English dictionary present in ubuntu using Bloom Filter written in Python.

code for Bloom Filter

```
//Bloom_Filter.py
# Python 3 program to build Bloom Filter
import math
import mmh3
from bitarray import bitarray

class BloomFilter(object):

	'''
	Class for Bloom filter, using murmur3 hash function
	'''

	def __init__(self, items_count,fp_prob):
		'''
		items_count : int
			Number of items expected to be stored in bloom filter
		fp_prob : float
			False Positive probability in decimal
		'''
		# False posible probability in decimal
		self.fp_prob = fp_prob

		# Size of bit array to use
		self.size = self.get_size(items_count,fp_prob)

		# number of hash functions to use
		self.hash_count = self.get_hash_count(self.size,items_count)

		# Bit array of given size
		self.bit_array = bitarray(self.size)

		# initialize all bits as 0
		self.bit_array.setall(0)

	def add(self, item):
		'''
		Add an item in the filter
		'''
		digests = []
		for i in range(self.hash_count):

			# create digest for given item.
			# i work as seed to mmh3.hash() function
			# With different seed, digest created is different
			digest = mmh3.hash(item,i) % self.size
			digests.append(digest)

			# set the bit True in bit_array
			self.bit_array[digest] = True
```

```
def check(self, item):
        '''
        Check for existence of an item in filter
        '''
        for i in range(self.hash_count):
            digest = mmh3.hash(item,i) % self.size
            if self.bit_array[digest] == False:

                # if any of bit is False then,its not present
                # in filter
                # else there is probability that it exist
                return False
        return True

    @classmethod
    def get_size(self,n,p):
        '''
        Return the size of bit array(m) to used using
        following formula
        m = -(n * lg(p)) / (lg(2)^2)
        n : int
            number of items expected to be stored in filter
        p : float
            False Positive probability in decimal
        '''
        m = -(n * math.log(p))/(math.log(2)**2)
        return int(m)

    @classmethod
    def get_hash_count(self, m, n):
        '''
        Return the hash function(k) to be used using
        following formula
        k = (m/n) * lg(2)

        m : int
            size of bit array
        n : int
            number of items expected to be stored in filter
        '''
        k = (m/n) * math.log(2)
        return int(k)
//MAIN PROGRAM
from bloomfilter import BloomFilter
from random import shuffle

# words to be added
word_present = []
count = 0
f = open("/usr/share/dict/american-english")
for x in f:
   if "'" in x:
      continue
   word_present.append(x)
   count += 1
n = count #no of items to add
p = 0.5 #false positive probability
```

```
bloomf = BloomFilter(n,p)
print("Size of bit array:{}".format(bloomf.size))
print("False positive Probability:{}".format(bloomf.fp_prob))
print("Number of hash functions:{}".format(bloomf.hash_count))

# word not added
word_absent = ['tanmay','rane','prashant']

for item in word_present:
    bloomf.add(item)

shuffle(word_present)
shuffle(word_absent)
test_words = ["apple","google"] + word_absent
shuffle(test_words)
for word in test_words:
        if bloomf.check(word):
                if word in word_absent:
                        print("'{}' is a false positive!".format(word))
                else:
                        print("'{}' is probably present!".format(word))
        else:
                print("'{}' is definitely not present!".format(word))
```

PostLab:

Take password as input from user.Check whether it exists in the dictionary using Bloom filter. If it exists, it indicates it is commonly used password.So give message to user to try another password till he/she enters strong password.

Code for postlab question

```
from bloomfilter import BloomFilter
from random import shuffle

# words to be added
word_present = []
count = 0
f = open("/media/tanmay/Data/SEM-8/BDA/EXP6/passwords")
for x in f:
    if "'" in x:
        continue
    word_present.append(x)
    count += 1
n = count #no of items to add
p = 0.3 #false positive probability

bloomf = BloomFilter(n,p)
print("Size of bit array:{}".format(bloomf.size))
print("False positive Probability:{}".format(bloomf.fp_prob))
print("Number of hash functions:{}".format(bloomf.hash_count))

for item in word_present:
    bloomf.add(item)

shuffle(word_present)
while(True):
    passwd = input("Please Enter a Password")
    if bloomf.check(passwd):
        print("'{}' is probably present please enter a different password!".format(passwd))
    else:
        print("'{}' is definitely not present it's added successfully!".format(passwd))
        break
```