Assignment 2

October 13, 2022

Exercise 2:

[ ]:
```
import bitarray
from hashlib import sha3_256, sha256, blake2b
import json
import bitarray
from tqdm import tqdm
import string
import matplotlib.pyplot as plt
```

[ ]:
```
size = int(1e7)
import bitarray
data = bitarray.bitarray(size)
bloomfilter = BloomFilter(size)
```

[ ]:
```
with open('/Users/polina/Desktop/words.txt') as f:
    for line in f:
        word = line.strip()
```

[ ]:
```
class BloomFilter(object):

    def __init__(self, size):


        self.bit_array = bitarray.bitarray(size)

        self.hashsize = size
        self.bit_array.setall(0)

    def add(self, words):


        hash1 = int(self.my_hash(words, self.hashsize))
        self.bit_array[hash1] = True


    def add_2(self, words):
```

```
        hash1 = int(self.my_hash(words, self.hashsize))
        hash2 = int(self.my_hash2(words, self.hashsize))
        self.bit_array[hash1] = True
        self.bit_array[hash2] = True


    def add_3(self, words):

        hash1 = int(self.my_hash(words, self.hashsize))
        hash2 = int(self.my_hash2(words, self.hashsize))
        hash3 = int(self.my_hash3(words, self.hashsize))
        self.bit_array[hash1] = True
        self.bit_array[hash2] = True
        self.bit_array[hash3] = True

    def check(self, test_words):
        hash1 = int(self.my_hash(test_words, self.hashsize))
        if self.bit_array[hash1] == False:
            return False
        else:
            return True

    def check_2(self, test_words):
        hash1 = int(self.my_hash(test_words, self.hashsize))
        hash2 = int(self.my_hash2(test_words, self.hashsize))
        if (self.bit_array[hash1] == False) or (self.bit_array[hash2] == False):
            return False
        else:
            return True

    def check_3(self, test_words):
        hash1 = int(self.my_hash(test_words, self.hashsize))
        hash2 = int(self.my_hash2(test_words, self.hashsize))
        hash3 = int(self.my_hash3(test_words, self.hashsize))
        if (self.bit_array[hash1] == False) or (self.bit_array[hash2] == False)␣
↪or (self.bit_array[hash3] == False):
            return False
        else:
            return True

    def my_hash(self, s, size):
        return int(sha256(s.lower().encode()).hexdigest(), 16) % size

    def my_hash2(self, s, size):
        return int(blake2b(s.lower().encode()).hexdigest(), 16) % size

    def my_hash3(self, s, size):
```

```
        return int(sha3_256(s.lower().encode()).hexdigest(), 16) % size

def replace(s, position, character):
    return s[:position] + character + s[position+1:]
```

[ ]:
```
def hashnumber():
    hashcount = input("hash number = ")
    return(hashcount)

# need to use Python 3.10 for match
#import sys;print(sys.version)
def add_word(hashcount):
    match hashcount:
        case "1":
            with open('/Users/polina/Desktop/words.txt') as f:
                for line in f:
                    word = line.strip()
                    bloomfilter.add(word)
            break
        case "2":
             with open('/Users/polina/Desktop/words.txt') as f:
                for line in f:
                    word = line.strip()
                    bloomfilter.add_2(word)
            break
        case "3":
            with open('/Users/polina/Desktop/words.txt') as f:
                for line in f:
                    word = line.strip()
                    bloomfilter.add_3(word)
```

[ ]:
```
import string
alphabet_list = string.ascii_lowercase
alphabet_list = list(alphabet_list)
print(alphabet_list)
```

[ ]:
```
def all_suggestions(test_words):
    suggest = []
    alphabet = alphabet_list
    result = spell_check(test_words, hashcount)
    if  result == False:
        for i in range(len(test_words)):
            for a in alphabet:
                suggest_word = replace(test_words, i, a)
                suggest.append(suggest_word)
    return(suggest)
```

```
# need to use Python 3.10 for match
def good_suggestion(all_suggestions, hashcount):
    correct_suggestion = []
    match hashcount:
        case "1":
            for n in all_suggestions:
                suggestresult = bloomfilter.check(n)
                if suggestresult == True:
                    correct_suggestion.append(n)
            break
        case "2":
            for n in all_suggestions:
                suggestresult = bloomfilter.check_2(n)
                if suggestresult == True:
                    correct_suggestion.append(n)
            break
        case "3":
            for n in all_suggestions:
                suggestresult = bloomfilter.check_3(n)
                if suggestresult == True:
                    correct_suggestion.append(n)
    return(correct_suggestion)

def spell_check(test_words, hashcount):
    match hashcount:
        case "1":
            result = bloomfilter.check(test_words)
            return result
        break
        case "2":
            result = bloomfilter.check_2(test_words)
            return result
        break
        case "3":
            result = bloomfilter.check_3(test_words)
            return result
```

[ ]:
```
hashcount = hashnumber()
```

[ ]:
```
add_word(hashcount)
```

[ ]:
```
test_words = input(" I asked my implementation to spell-check when using 1e7␣
 ↪bits: ")
```

[ ]:
```
spell_check(test_words, hashcount)
if spell_check(test_words, hashcount) != True:
```

```
    suggestionlist = all_suggestions(test_words)
    good_suggestionlist = good_suggestion(suggestionlist, hashcount)
    print(good_suggestionlist)
```

[ ]:
```
add_word("1")
spell_check("bloeer", "1")
```

[ ]:
```
with open('/Users/polina/Desktop/BIS_634/Assignmnet 2/typos.json', 'r') as f:
    file = f.read()
    text = json.loads(file)
    print(len(text))
```

[ ]:
```
from tqdm import tqdm
import numpy as np

def good_suggest(text, hashcount):
    correct = 0
    misidentified = 0
    good_suggestion = 0
    for i in range(len(text)):
        if text[i][0] == text[i][1]:
            correct += 1
        elif text[i][0] != text[i][1]:
            if spell_check(text[i][0], hashcount) == True:
                misidentified += 1
            else:
                suggestion = all_suggestions(text[i][0])
                good = good_suggestion(suggestion, hashcount)
                for n in good:
                    if (len(good) <= 3) and (n == text[i][1]):
                        good_suggestion += 1
    return correct, misidentified, good_suggestion
```

[ ]:
```
# hash 1
filter_size = np.logspace(0, 9, num=20, dtype=int)
misidentified1 = []
suggestion_list1 = []
correct1 = []
for n in range(len(filter_size)):
    bloomfilter = BloomFilter(int(filter_size[n]))
    add_word("1")
    answer1 = good_suggest(text, "1")
    misidentified1.append(answer1[1])
    suggestion_list1.append(answer1[2])
    correct1.append(answer1[0])

goodsuggestion1 = []
```

```
misidentified1 =[]

for i in range(len(misidentified1)):
    goodsuggestion = (suggestion_list1[i] / 25000)
    misidentified = (misidentified1[i] / 25000)
    goodsuggestion1.append(goodsuggestion)
    misidentified1.append(misidentified)
```

[ ]:
```
# hash 2
misidentified2 = []
suggestion_list2 = []
correct2 = []
for n in range(len(filter_size)):
    bloomfilter = BloomFilter(int(filter_size[n]))
    add_word("2")
    answer2 = good_suggest(text, "2")
    misidentified2.append(answer2[1])
    suggestion_list2.append(answer2[2])
    correct2.append(answer2[0])

goodsuggestion2 = []
misidentified2 =[]

for i in range(len(misidentified2)):
    goodsuggestion = (suggestion_list2[i] / 25000)
    misidentified = (misidentified2[i] / 25000)
    goodsuggestion2.append(goodsuggestion)
    misidentified2.append(misidentified)
```

[ ]:
```
# hash 3
misidentified3 = []
suggestion_list3 = []
correct3 = []
for n in range(len(filter_size)):
    bloomfilter = BloomFilter(int(filter_size[n]))
    add_word("3")
    answer3 = good_suggest(text, "3")
    misidentified3.append(answer3[1])
    suggestion_list3.append(answer3[2])
    correct3.append(answer3[0])

goodsuggestion3 = []
misidentified3 =[]

for i in range(len(misidentified3)):
    goodsuggestion = (suggestion_list3[i] / 25000)
    misidentified = (misidentified3[i] / 25000)
```

```
    goodsuggestion3.append(goodsuggestion)
    misidentified3.append(misidentified)
```

[ ]:
```
plt.xscale('log')
plt.plot(filter_size, goodsuggestion1, label="good suggestions %, 1 hashh")
plt.plot(filter_size, goodsuggestion2, label="good suggestions %, 2 hashes")
plt.plot(filter_size, goodsuggestion3, label="good suggestions %, 3 hashes")
plt.plot(filter_size, misidentified1, label="misidentified %, 1 has")
plt.plot(filter_size, misidentified2, label="misidentified %, 2 hashes")
plt.plot(filter_size, misidentified3, label="misidentified %, 3 hashes")
plt.legend(loc="lower left")
plt.show()
```

Exercise 3:

[ ]:
```
# import libraries
import pandas as pd
import numpy as np
import random
import time
```

[ ]:
```
class Tree:
    def __init__(self):
        self.value = None
        self.left = None
        self.right = None

    def add(self, item):
        if self.value is None:
            self.value = item
            return

        elif item < self.value:
            if self.left:
                self.left.add(item)
            else:
                self.left = Tree()
                self.left.add(item)
            return

        elif item > self.value:
            if self.right:
                self.right.add(item)
            else:
                self.right = Tree()
                self.right.add(item)
        return
```

```
    def __contains__(self, item):
        if self.value == item:
            return True
        elif self.left and item < self.value:
            return item in self.left
        elif self.right and item > self.value:
            return item in self.right
        else:
            return False
```

[ ]:
```
def printTree(tree, list=[]):
    if not tree: return []
    list.append(tree.value)
    if tree.left: printTree(tree.left, list)
    if tree.right: printTree(tree.right, list)
    return list
```

[ ]:
```
my_tree = Tree()
for item in [55, 62, 37, 49, 71, 14, 17]:
     my_tree.add(item)
print(printTree(my_tree, list=[]))
```

[ ]:
```
my_tree.__contains__(55)
```

[ ]:
```
my_tree.__contains__(42)
```

Using various sizes n of trees (populated with random data) and sufficiently many calls to in (each individual call should be very fast, so you may have to run many repeated tests), demonstrate that in is executing in O(log n) times; on a log-log plot, for sufficiently large n, the graph of time required for checking if a number is in the tree as a function of n should be almost horizontal.

[ ]:
```
import random
import time
import math
import statistics
import matplotlib.pyplot as plt
```

O(log n) times

[ ]:
```
test_time = []
n_sample = 1000
repeat = 100
for _ in range(n_sample):
    n = random.randint(0, 10000)
    my_tree = Tree()
    for _ in range(n):
        my_tree.add(random.random())
    start = time.time()
```

```
    for _ in range(repeat):
        my_tree.__contains__(random.random())
    end = time.time()
    test_time.append((end - start, n))
```

[ ]:
```
plt.scatter([x[1] for x in test_time], [x[0] for x in test_time], s=1)
plt.show()
```

This speed is not free. Provide supporting evidence that the time to setup the tree is O(n log n) by timing it for various sized ns and showing that the runtime lies between a curve that is O(n) and one that is O(n**2)

O(n)

[ ]:
```
test_time_2 = []
n_sample = 100
repeat = 10
for _ in range(n_sample):
    n = random.randint(0, 10000)
    secquence = [random.random() for _ in range(n)]
    start = time.time()
    for _ in range(repeat):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    test_time_2.append((end - start, n))
```

[ ]:
```
plt.scatter([x[1] for x in test_time_2], [x[0] for x in test_time_2], s=1)
plt.show()
```

O(n**2)

[ ]:
```
test_time_3 = []
n_sample = 100
repeat = 10
for _ in range(n_sample):
    n = random.randint(0, 200)
    secquence = [random.random() for _ in range(n)]
    secquence = sorted(secquence)
    start = time.time()
    for _ in range(repeat):
        my_tree = Tree()
        for i in secquence:
            my_tree.add(i)
    end = time.time()
    test_time_3.append((end - start, n))
```

[ ]:
```
plt.scatter([x[1] for x in test_time_3], [x[0] for x in test_time_3], s=1)
plt.show()
```

n**2, and n

[ ]:
```
def time_function(n):
    timing = []
    n = [random.randint(1,n) for _ in range(n)]
    for attempt in [i for i in range(100)]:
        start = time.time()
        my_tree = Tree()
        for item in n:
            my_tree.add(item)
        end = time.time()
        timing.append(end - start)
    return min(timing)
```

[ ]:
```
n_size = [1,100,1000,10000]
true_size = [time_function(n) for n in n_size]
size = true_size [0]
quadratic = [size*n**2 for n in n_size]
linear = [size*n for n in n_size]
plt.plot(n_size,true_size)
plt.plot(n_size,quadratic)
plt.plot(n_size,linear)
plt.xscale('log')
plt.yscale('log')
plt.ylabel('time')
plt.xlabel('n_size')
plt.legend(['run time','O(n^2)','O(n)'],loc = 'upper left')
plt.show()
```

[ ]:
```

```

Exercise 4:

[ ]:
```
import matplotlib.pyplot as plt
import multiprocessing
from time import perf_counter
import numpy as np
import random
```

[ ]:
```
def alg1(data):
    data = list(data)
    changes = True
    while changes:
        changes = False
        for i in range(len(data) - 1):
```

```
            if data[i + 1] < data[i]:
                data[i], data[i + 1] = data[i + 1], data[i]
                changes = True
    return data
```

[ ]:
```
def alg2(data):
  if len(data) <= 1:
    return data
  else:
    split = len(data) // 2
    left = iter(alg2(data[:split]))
    right = iter(alg2(data[split:]))
    result = []
    # note: this takes the top items off the left and right piles
    left_top = next(left)
    right_top = next(right)
    while True:
      if left_top < right_top:
        result.append(left_top)
        try:
          left_top = next(left)
        except StopIteration:
          # nothing remains on the left; add the right + return
          return result + [right_top] + list(right)
      else:
        result.append(right_top)
        try:
          right_top = next(right)
        except StopIteration:
          # nothing remains on the right; add the left + return
          return result + [left_top] + list(left)
```

[ ]:
```
test_list = [55, 62, 37, 49, 71, 14, 17]
```

[ ]:
```
print(alg1(test_list))
print(alg2(test_list))
```

Data 1

[ ]:
```
def data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1):
    import numpy
    state = numpy.array([x, y, z], dtype=float)
    result = []
    for _ in range(n):
        x, y, z = state
        state += dt * numpy.array([
            sigma * (y - x),
            x * (rho - z) - y,
```

```
            x * y - beta * z
        ])
        result.append(float(state[0] + 30))
    return result
```

[ ]:
```
from time import perf_counter

def test_time_data(limit, source):
    measure = dict()
    measure['list_size'] = []
    measure['alg1_time'] = []
    measure['alg2_time'] = []

    for n in range(limit):
        measure['list_size'].append(2**n)

        test_list_0 = source(2**n)


        time_alg1_start = perf_counter()
        alg1(test_list_0)
        time_alg1_stop = perf_counter()
        measure['alg1_time'].append(time_alg1_stop-time_alg1_start)
        time_alg1_start = perf_counter()
        alg2(test_list_0)
        time_alg1_stop = perf_counter()
        measure['alg2_time'].append(time_alg1_stop-time_alg1_start)

    return measure
```

[ ]:
```
 measure_dat1 = test_time_data(13, data1)
```

[ ]:
```
data1_data = pd.DataFrame(measure_dat1)
data1_data
```

[ ]:
```
#create log-log plot
fig, ax = plt.subplots()
plt.plot(data1_data.list_size, data1_data.alg1_time, label='Alg1')
plt.plot(data1_data.list_size, data1_data.alg2_time, label='Alg2')
plt.xscale('log', base=2)
plt.yscale('log', base=10)
ax.set_title(' Alg1 and Alg2 -data 1')
fig.legend()
```

Data 2

[ ]:
```
def data2(n):
    return list(range(n))
```

[ ]:
```
measure_dat2 = test_time_data(13, data2)
```

[ ]:
```
data2_data = pd.DataFrame(measure_dat2)
data2_data
```

[ ]:
```
fig, ax = plt.subplots()
plt.plot(data2_data.list_size, data2_data.alg1_time, label='Alg1')
plt.plot(data2_data.list_size, data2_data.alg2_time, label='Alg2')
plt.xscale('log', base=2)
plt.yscale('log', base=10)
ax.set_title('Alg1 and Alg2 - data 2')
fig.legend()
```

Data 3

[ ]:
```
def data3(n):
    return list(range(n, 0, -1))
```

[ ]:
```
measure_dat3 = test_time_data(13, data3)
```

[ ]:
```
data3_data = pd.DataFrame(measure_dat3)
data3_data
```

[ ]:
```
fig, ax = plt.subplots()
plt.plot(data3_data.list_size, data3_data.alg1_time, label='Alg1')
plt.plot(data3_data.list_size, data3_data.alg2_time, label='Alg2')
plt.xscale('log', base=2)
plt.yscale('log', base=10)
ax.set_title('Alg1 and Alg2 - data 3')
fig.legend()
```

Multiprocessing

[ ]:
```
def parallelize_alg2(data):
    if len(data) <= 1:
        return data
    else:
        split = len(data) // 2
        with multiprocessing.Pool() as p:
            [left, right] = p.map(
                alg2, [data[:split], data[split:]])
        left = iter(left)
        right = iter(right)
        # combining the left and right data
        result = []
        left_top = next(left)
        right_top = next(right)
    while True:
```

```
        if left_top < right_top:
            result.append(left_top)
            try:
                left_top = next(left)
            except StopIteration:
            # nothing remains on the left; add the right + return
                return result + [right_top] + list(right)
        else:
            result.append(right_top)
            try:
                right_top = next(right)
            except StopIteration:
        # nothing remains on the right; add the left + return
                return result + [left_top] + list(left)
```

[ ]:
```
from tqdm import tqdm

if __name__ == '__main__':
    data_variant = np.logspace(0, 23, base=2, dtype=int)
    duration = []
    alg2data1time = []

    for n in tqdm(data_variant):
        data_set = data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1)
        alg2data1_start = perf_counter()
        alg2(data_set)     ## time taken by the unparallelized algo
        alg2data1_stop= perf_counter()
        alg2data1time.append(alg2data1_stop - alg2data1_start)
        start_time = perf_counter()
        parallelize_alg2(data_set)    #time taken by the parallelized algo
        stop_time = perf_counter()
        duration.append(stop_time - start_time)
        print(duration[-1],  alg2data1time[-1])
    plt.loglog(data_variant, alg2data1time, label = "mergesort alg2")
    plt.loglog(data_variant, duration, label = "prallelized alg2")
    plt.legend()
    plt.show()
```