Assignment 2

Due Friday by 11:59pm **Points** 100

Please put your answers to this assignment in the same GitHub repository that you used for the first assignment.

In the repository, be sure to include runnable source code, and a single consolidated (one file) report (readme) that has answers to all the questions, along with code, and relevant output (e.g. graphs). I would prefer for the consolidated report to be in pdf format, but markdown (md), word (docx), and html are also fine. Failure to provide a single-file report will incur a 5 point penalty.

As always, reasonable discussions about programming issues (or making a report) are encouraged, both in the relevant **forum** and in person with your classmates. Do not, however, share code with your classmates; the work you submit must be your own. (i.e. Learn from each other, but don't cheat.) For questions involving fine details of your implementation, please meet with the TF or professor. Please do not spend hours stuck on an exercise; if you get stuck and have tried to get unstuck, ask the TF or professor a question and move on to something else until you get an answer.

Exercise 1 (15 points)

Your friend says to you, "you have to help me! I'm supposed to present in lab meeting in less than an hour, and I haven't been able to get the simplest thing to work!" After you help them calm down, they explain: through a collaboration with a fitness app, they have a 4GB file of high-precision weights of exactly 500 million people throughout the world. Even with 8GB of RAM, they get a MemoryError when trying to load the file in and find the average weight. They show you their code which worked fine when they tested it on a small number of weights:

```
with open('weights.txt') as f:
    weights = []
    for line in f:
        weights.append(float(line))
print("average =", sum(weights) / len(weights))
```

Aha! You exclaim.

Explain what went wrong (5 points). Suggest a way of storing all the data in memory that

would work (5 points), and suggest a strategy for calculating the average that would not require storing all the data in memory (5 points).

Remember, your friend has to present soon, so keep your answers concise but thorough. Assume your friend is reasonably computer literate and is not using a vastly outdated computer or installation of Python. (i.e. don't blame 32 bit systems.)

Exercise 2: (25 points)

In this exercise, we'll use a bloom filter to identify a correct word from a single-character typo. This is computationally the same problem as mapping a gene with a single nucleotide permutation to multiple reference genomes (this would be too many possibilities to keep all in memory, hence the use of a bloom filter). We'll talk about sequence alignment later in the course, which is a different but related problem.

Download the list of English words from https://github.com/dwyl/english-words/blob/master/words.txt (https://github.com/dwyl/english-words/blob/master/words.txt)

This list may be read in one-at-a-time for processing via e.g.

```
with open('words.txt') as f:
    for line in f:
        word = line.strip()
        # do something with the word here
```

Implement a Bloom Filter "from scratch" using a bitarray (6 points):

```
import bitarray
data = bitarray.bitarray(size)
```

with the following three hash functions

```
from hashlib import sha3_256, sha256, blake2b

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

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

def my_hash3(s):
    return int(sha3_256(s.lower().encode()).hexdigest(), 16) % size
```

and store the words in the bloom filter (2 points).

These hash functions all return integers in [0, size), where size is some integer specified elsewhere.

Write a function that suggests spelling corrections using the bloom filter as follows: Try all possible single letter substitutions and see which ones are identified by the filter as possibly words. This algorithm will always find the intended word for this data set, and possibly other words as well. (8 points)

Plot the effect of the size of the filter together with the choice of just the first, the first two, or all three of the above hash functions on the number of words misidentified from typos.json (https://yale.instructure.com/courses/81264/files/6745069/download?download_frd=1) **as correct and the number of "good suggestions".** (4 points) typos.json consists of a list of lists of [typed_word, correct_word] pairs; exactly half of the entries are spelled correctly. For this exercise, consider a list of spelling suggestions good if there are no more than three suggestions and one of them is the word that was wanted.

Approximately how many bits is necessary for this approach to give good suggestions (as defined above) 90% of the time when using each of 1, 2, or 3 hash functions as above? (5 points)

Notes:

Perfect spelling correction is impossible when considering words in isolation, because many words differ by only one letter (e.g. sequencer and sequences).

Self-check:

I asked my implementation to spell-check "floeer" when using 1e7 bits (about 1.25 MB):

Suggestions from using the first hash: ['bloeer', 'qloeer', 'fyoeer', 'flofer', 'floter', 'flower', 'floeqr', 'floees']

Suggestions from using the first and second hashes: ['fyoeer', 'floter', 'flower']

Suggestions from all three hashes: ['floter', 'flower']

("floter" was a Middle English word that is in our list of words, so both "floter" and "flower" are appropriate suggestions.)

Exercise 3 (25 points)

Starting from the following framework of a Tree:

```
class Tree:
    def __init__(self):
        self._value = None
        self.left = None
        self.right = None
```

extend the above into a binary search tree. In particular, provide an **add** method that inserts a single numeric value at a time according to the rules for a binary search tree (10 points). (Edit: 2022-10-07: consolidated two redundant "5 points" into add being worth "10 points").

When this is done, you should be able to construct the tree from **slides 3** via:

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

Add the following **__contains__** method. This method will allow you to use the **in** operator; e.g. after this change, **55 in my_tree** should be True in the above example, whereas **42 in my_tree** would be False. Test this. (5 points).

```
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
```

__contains__ is an example of what's known as a magic method; such methods allow new data structures you might create to work like built-in data types. For more on magic methods, search the web, or start with **this page (https://python-course.eu/oop/magic-methods.php)** .

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. (5 points).

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). (5 points)

Note: the tree searches are not guaranteed to be able to run in exactly O(log n) time because the tree might not be balanced, but as long as n is large enough and your numbers are generated randomly, you can expect the tree to be nearly balanced.

Do not trivialize this problem; that is, do not use a module that directly provides a binary tree implementation.

Exercise 4 (35 points)

Consider the following two algorithms:

```
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
```

and

```
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)
```

They both take lists of numbers and return a list of numbers. In particular, for the same input, they return the same output; that is, they solve the same data processing problem but they do it in very different ways. Let's explore the implications and how they interact with different data sets.

By trying a few tests, hypothesize what operation these functions perform on the list of values. (Include your tests in your readme file. (3 points)

Explain in your own words how (at a high level... don't go line by line, but provide an intuitive explanation) each of these functions is able to complete the task. (2 points each; 4 points total)

Time the performance (use time.perf_counter) of alg1 and alg2 for various sizes of data n where the data comes from the function below, plot on a log-log graph as a function of n, and describe the apparent big-O scaling of each. (4 points). Note: Do *not* include the time spent generating the data in your timings; only measure the time of alg1 and alg2. Note: since you're plotting on a log axis, you'll want to pick n values that are evenly spaced on a log scale. numpy.logspace can help. (Let n get large but not so large that you can't run the code in a reasonable time.)

```
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
```

Repeat the above but for data coming from the below. (4 points)

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

Repeat the above but for data coming from the below. (4 points)

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

Discuss how the scaling performance compares across the three data sets. (2 points) Which algorithm would you recommend to use for arbitrary data and why? (2 points)

Explain in words how to parallelize alg2; that is, where are there independent tasks whose results can be combined? (2 points)

Using the multiprocessing module, provide a two-process parallel implementation of alg2 (4 points), compare its performance on data from the data1 function for moderate n (3 points), and discuss your findings (3 points).