QUESTION 1 - Memory Allocation

Tahir Manuel D Mello
BIS634 Assignment 2

Explain what went wrong (5 points).

Answer:
My friend tried to load in too much data in one operation.
The operation ran out of memory.

A number of factors could have caused this:

- Some of the RAM is being used by the OS, other programs and other Python overheads.
- Not all 8 GB of RAM is available to be used.
- There could be a limit on how much of the RAM a single thread of the CPU can use. (Since this is not multi-threaded)

Suggest a way of storing all the data in memory that would work (5 points).

Answer:
The data can be divided into batches.
Each batch can be stored sequentially and unneeded memory can be cleared between each batch.

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

Answer:
A running sum of the data could be taken. This would require only a single float to be stored.
Every iteration, the incoming number on the line is added to the 'sum' float.
This way, we can still calculate the sum without storing every line in memory.

```
with open('weights.txt') as f:
    sum = float(0)
    for line in f:
        sum = sum + float(line)
print("average =", sum / len(weights))
```

QUESTION 2 - Bloom Filter

In [1]:
```
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
```

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

In [2]:
```
import bitarray
import string


class BloomFilter3:

    def __init__(self, size):
        self._data = bitarray.bitarray(size)
        self._data.setall(0)


    def build(self, word):
        index = [0] * 3
        index[0] = my_hash(word)
        index[1] = my_hash2(word)
        index[2] = my_hash3(word)

        for i in index:
            self._data[i] = 1


    def search(self, word):
        index = [0] * 3
        index[0] = my_hash(word)
        index[1] = my_hash2(word)
        index[2] = my_hash3(word)

        answer = 0

        for i in index:
            answer = answer + self._data[i]

        if answer == len(index):
            return word


    def spellcheck(self, word):

        results = []

        for i in range(0 , len(word) - 1):
            for character in string.ascii_lowercase:
                word_temp = list(word)
                word_temp[i] = character
                word_temp = ''.join(word_temp)

                check = self.search(word_temp)

                if check is not None:
                    results.append(check)

        return results
```

Store the words in the bloom filter (2 points).

In [3]:
```
size = 10_000_000
bloom_filter = BloomFilter3(size)

with open('words.txt') as f:
    for line in f:
        word = line.strip()
        #print(word)

        bloom_filter.build(word)
```

Write a function that suggests spelling corrections using the bloom filter (Code in above Class) (8 points)

In [4]:
```
bloom_filter.spellcheck('floeer')
```

Out[4]:
```
['floter', 'flower']
```

In [5]:
```
bloom_filter.spellcheck('pitato')
```

Out[5]:
```
['potato']
```

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 as correct and the number of "good suggestions". (4 points)

In [6]:
```
import json

f = open('typos.json')

typos_data = json.load(f)

f.close()

typos_data_misspelt = []

for item in typos_data:
    if item[0] != item[1]:
        typos_data_misspelt.append(item)
```

In [7]:
```
import bitarray
import string
import numpy as np

class BloomFilter1:

    def __init__(self, size):
        self._data = bitarray.bitarray(size)
        self._data.setall(0)


    def build(self, word):
        index = [0] * 1
        index[0] = my_hash(word)

        for i in index:
            self._data[i] = 1


    def search(self, word):
        index = [0] * 1
        index[0] = my_hash(word)

        answer = 0
```

```
        for i in index:
            answer = answer + self._data[i]

        if answer == len(index):
            return word


    def spellcheck(self, word):

        results = []

        for i in range(0 , len(word) - 1):
            for character in string.ascii_lowercase:
                word_temp = list(word)
                word_temp[i] = character
                word_temp = ''.join(word_temp)

                check = self.search(word_temp)

                if check is not None:
                    results.append(check)

        return results



class BloomFilter2:

    def __init__(self, size):
        self._data = bitarray.bitarray(size)
        self._data.setall(0)


    def build(self, word):
        index = [0] * 2
        index[0] = my_hash(word)
        index[1] = my_hash2(word)

        for i in index:
            self._data[i] = 1


    def search(self, word):
        index = [0] * 2
        index[0] = my_hash(word)
        index[1] = my_hash2(word)

        answer = 0

        for i in index:
            answer = answer + self._data[i]

        if answer == len(index):
            return word


    def spellcheck(self, word):

        results = []

        for i in range(0 , len(word) - 1):
            for character in string.ascii_lowercase:
                word_temp = list(word)
                word_temp[i] = character
                word_temp = ''.join(word_temp)

                check = self.search(word_temp)

                if check is not None:
                    results.append(check)
```

```
        return results
```

In [8]:
```
n_range = []

for i in range(0,10):
    n_range.append(10**i)
```

In [9]:
```
success_counts_1 = []

for size in n_range:

    bloom_filter_1 = BloomFilter1(size)

    with open('words.txt') as f:
        for line in f:
            word = line.strip()
            #print(word)

            bloom_filter_1.build(word)

    success = 0

    for item in typos_data_misspelt:
        answer = bloom_filter_1.spellcheck(item[0])

        if len(answer) <= 3 and (item[1] in answer):
            success = success + 1

    success_counts_1.append(success)
```

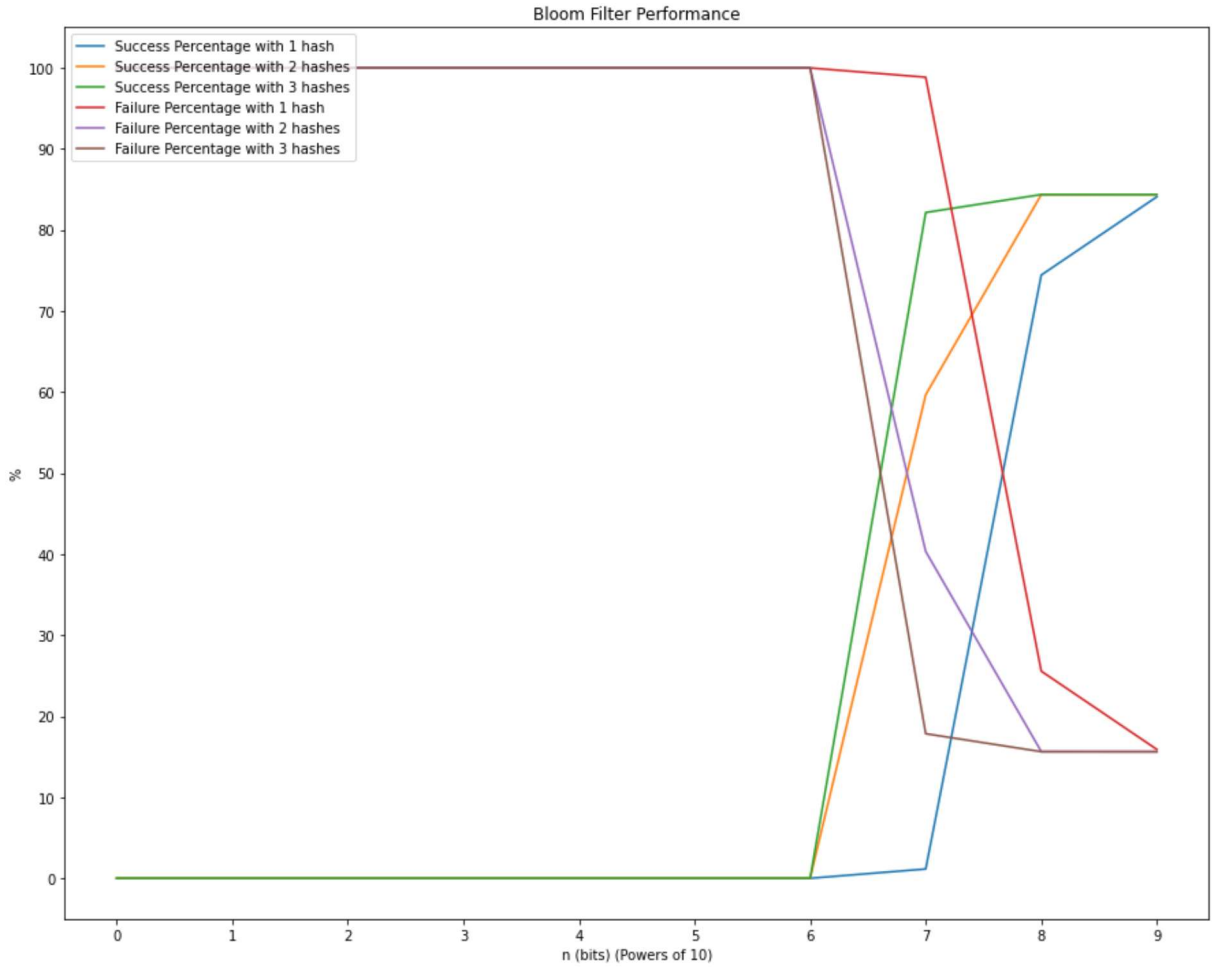
In [10]:
```
success_counts_2 = []

for size in n_range:

    bloom_filter_2 = BloomFilter2(size)

    with open('words.txt') as f:
        for line in f:
            word = line.strip()
            #print(word)

            bloom_filter_2.build(word)

    success = 0

    for item in typos_data_misspelt:
        answer = bloom_filter_2.spellcheck(item[0])

        if len(answer) <= 3 and (item[1] in answer):
            success = success + 1

    success_counts_2.append(success)
```

In [11]:
```
success_counts_3 = []

for size in n_range:

    bloom_filter_3 = BloomFilter3(size)

    with open('words.txt') as f:
        for line in f:
            word = line.strip()
            #print(word)

            bloom_filter_3.build(word)

    success = 0
```

```
    for item in typos_data_misspelt:
        answer = bloom_filter_3.spellcheck(item[0])

        if len(answer) <= 3 and (item[1] in answer):
            success = success + 1

    success_counts_3.append(success)
```

In [28]:

```
import matplotlib.pyplot as plt
import numpy as np
import math

success_percentages_1 = np.array(success_counts_1)*100/len(typos_data_misspelt)
success_percentages_2 = np.array(success_counts_2)*100/len(typos_data_misspelt)
success_percentages_3 = np.array(success_counts_3)*100/len(typos_data_misspelt)

failure_percentages_1 = 100 - success_percentages_1
failure_percentages_2 = 100 - success_percentages_2
failure_percentages_3 = 100 - success_percentages_3

x_axis = np.log10(n_range)

plt.figure(figsize=(15, 12))
plt.plot(x_axis, success_percentages_1, label = 'Success Percentage with 1 hash')
plt.plot(x_axis, success_percentages_2, label = 'Success Percentage with 2 hashes')
plt.plot(x_axis, success_percentages_3, label = 'Success Percentage with 3 hashes')
plt.plot(x_axis, failure_percentages_1, label = 'Failure Percentage with 1 hash')
plt.plot(x_axis, failure_percentages_2, label = 'Failure Percentage with 2 hashes')
plt.plot(x_axis, failure_percentages_3, label = 'Failure Percentage with 3 hashes')


plt.title('Bloom Filter Performance')
plt.ticklabel_format(style='sci', axis='x')
plt.ylabel('%')
plt.xlabel('n (bits) (Powers of 10)')
plt.legend(loc="upper left")

plt.xticks(np.arange(min(x_axis), max(x_axis)+1, 1.0))
plt.yticks(np.arange(min(success_percentages_1), max(failure_percentages_1)+10, 10))
plt.show()
```



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)

Cutoff changed to 80% since no bloomfilter was returning abouve 90%.
This could have been due to a difference in definition of the metric used for performance

In [35]:
```
cutoff_1 = np.where(success_percentages_1  >= 80)
cutoff_1

print("The number of bits necessary for good performance as defined about for 1 hash is 10 ^",
```

```
The number of bits necessary for good performance as defined about for 1 hash is 10 ^ 9
```

In [37]:
```
cutoff_2 = np.where(success_percentages_2  >= 80)
cutoff_2

print("The number of bits necessary for good performance as defined about for 2 hashes is 10 ^
```

```
The number of bits necessary for good performance as defined about for 2 hashes is 10 ^ 8
```

In [38]:
```
cutoff_3 = np.where(success_percentages_3  >= 80)
cutoff_3

print("The number of bits necessary for good performance as defined about for 2 hashes is 10 ^
```

```
The number of bits necessary for good performance as defined about for 2 hashes is 10 ^ 7
```

QUESTION 3 - Binary Search Tree

Provide an add method that inserts a single numeric value at a time according to the rules for a binary search tree (10 points)

In [1]:

```
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

        else:
            if item < self._value:
                if self.left is not None:
                        self.left.add_to_tree(item, self.left)
                else:
                    self.left = Tree()
                    self.left._value = item

            else:
                if self.right is not None:
                        self.right.add_to_tree(item, self.right)
                else:
                    self.right = Tree()
                    self.right._value = item


    def add_to_tree(self, item, tree):
        if item < tree._value:
            if tree.left is not None:
                tree.left.add_to_tree(item, tree.left)
            else:
                tree.left = Tree()
                tree.left.add(item)

        else:
            if tree.right is not None:
                    tree.right.add_to_tree(item, tree.right)
            else:
                tree.right = Tree()
                tree.right.add(item)


    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(self):

        if self.left:
            self.left.printtree()
        print( self._value),

        if self.right:
             self.right.printtree()
```

In [2]:
```
my_tree = Tree()

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

In [3]:
```
my_tree.printtree()
```

```
14
17
37
49
55
62
71
```

Add the following *contains* method and test (5 points)

In [4]:
```
my_tree.__contains__(55)
```

Out[4]:
```
True
```

In [5]:
```
my_tree.__contains__(42)
```

Out[5]:
```
False
```

In [6]:
```
my_tree.__contains__(37)
```

Out[6]:
```
True
```

In [7]:
```
my_tree.__contains__(237)
```

Out[7]:
```
False
```

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 visibly starting to flatten. (5 points).

In [8]:
```
n_range = []

for i in range(0,7):
    n_range.append(10**i)
```

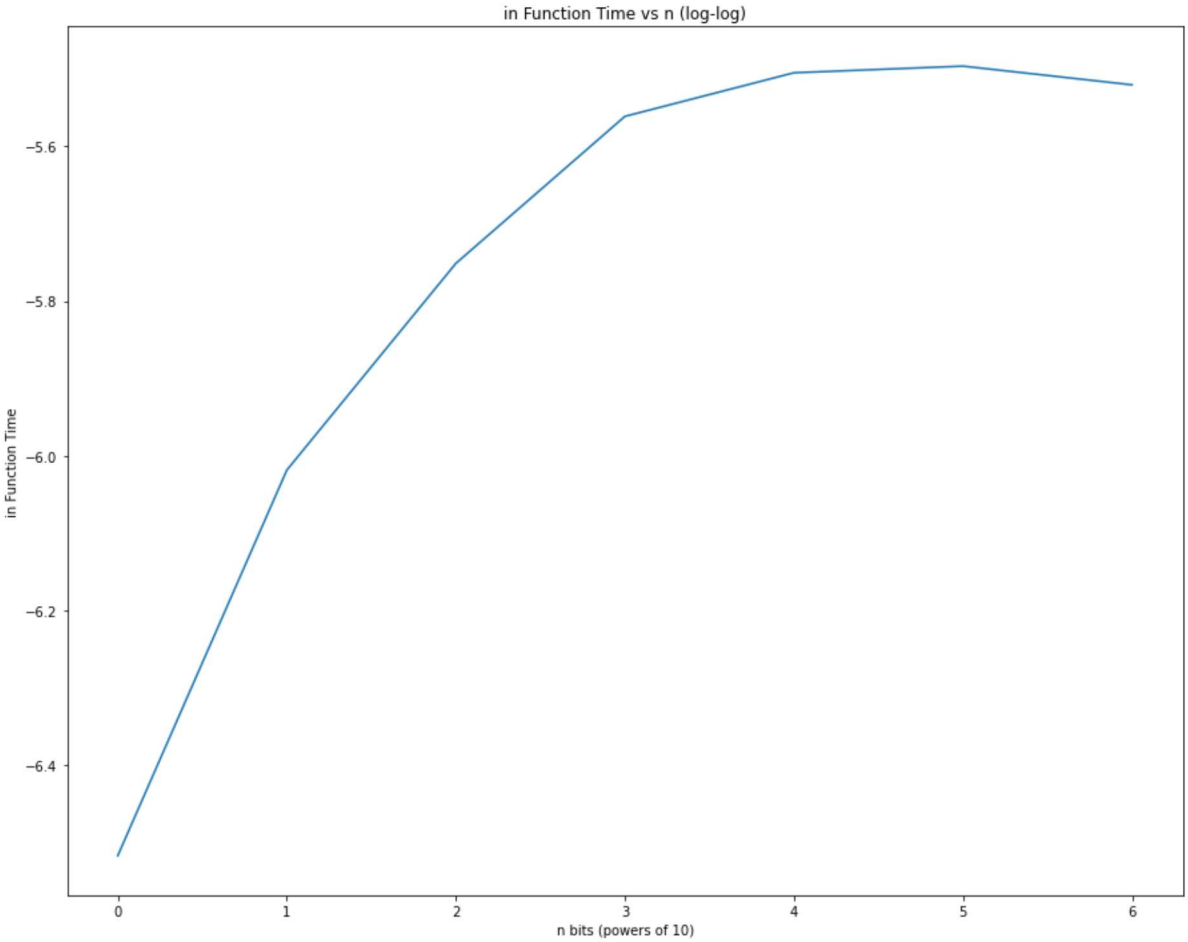
In [9]:
```
import numpy as np
import time

time_contains_set = []
```

```
time_builds_set = []

searching_set = np.random.randint(-1000, 1000, 1000)

for n in n_range:

    print(n)
    input_number = np.random.randint(-1000, 1000, n)




    t_start = time.perf_counter()

    my_tree = Tree()

    for item in input_number:
        my_tree.add(item)

    t_end = time.perf_counter()

    timer_builds = t_end - t_start

    time_builds_set.append(timer_builds)




    timer_contains = 0

    for item in searching_set:

        t_start = time.perf_counter()
        my_tree.__contains__(item)
        t_end = time.perf_counter()

        timer_contains = timer_contains + (t_end-t_start)


    time_contains_set.append(timer_contains/1000)
```

```
1
10
100
1000
10000
100000
1000000
```

In [18]:

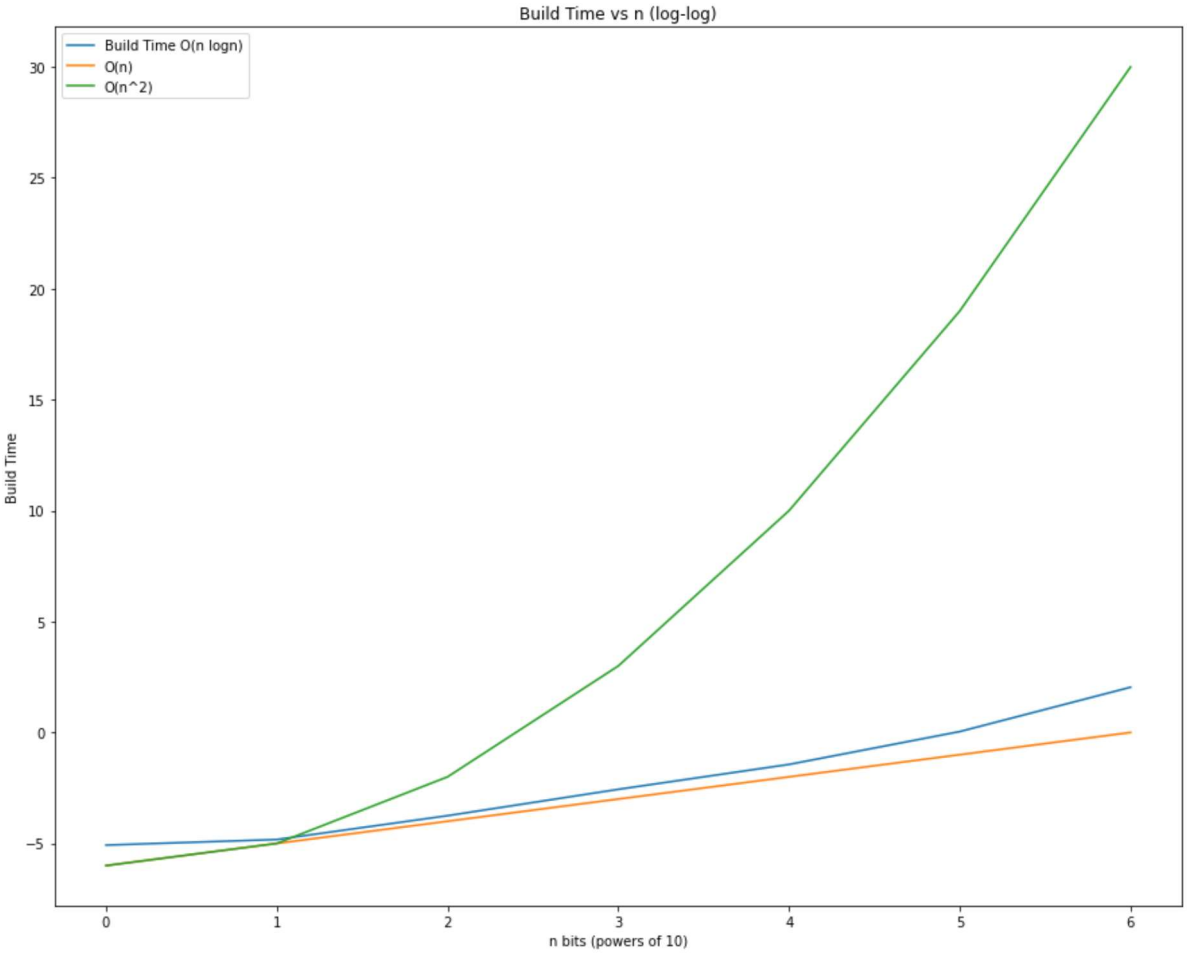
```
import matplotlib.pyplot as plt

time_set_log_1 = np.log10(time_contains_set)
n_log = np.log10(n_range)

plt.figure(figsize=(15, 12))

plt.plot(n_log, time_set_log_1)

plt.title('in Function Time vs n (log-log)')
plt.xlabel('n bits (powers of 10)')
plt.ylabel('in Function Time')

plt.show()
```



As can be seen, in is executing in O(logn) time.

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(n2). (5 points)**

In [31]:
```
import matplotlib.pyplot as plt

time_set_log_2 = np.log10(time_builds_set)
n_log_s = np.square(n_log)

plt.figure(figsize=(15, 12))

plt.plot(n_log, time_set_log_2, label = 'Build Time O(n logn)')
plt.plot(n_log, n_log-6, label = 'O(n)')
plt.plot(n_log, n_log_s-6, label = 'O(n^2)')

plt.title('Build Time vs n (log-log)')
plt.xlabel('n bits (powers of 10)')
plt.ylabel('Build Time')
plt.legend()
plt.show()
```



As can be seen, since the curve lines between O(n) and 0(n^2), the runtime is at O(n logn) to setup the tree.

QUESTION 4 - Sorting Algorithm

In [1]:
```
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)
```

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)

In [2]:
```
alg1([12,-56,78,98,12023])
```

Out[2]:
```
[-56, 12, 78, 98, 12023]
```

In [3]:
```
alg1([12,-56.78 ,78.23, 9348, 12023])
```

Out[3]:
```
[-56.78, 12, 78.23, 9348, 12023]
```

In [4]:
```
alg2([43.78,76,29,70,12])
```

Out[4]:
```
[12, 29, 43.78, 70, 76]
```

In [5]:
```
alg2([12,-56.78 ,78.23, 9348, 12023])
```

Out[5]:
```
[-56.78, 12, 78.23, 9348, 12023]
```

Hypothesis: Both these functions sort a list of input numbers into ascending order.

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)

Algorithm 1:
The algorithm moves through the list starting from the first element.
If the next element is smaller than the current element, it exchanges the element.
It continues to the end of the element with this checking and swapping operation. This is one iteration.
It then restarts the process and completes all iterations needed to sort the list.

Algorithm 2:
This algorithm divides the list into smaller sublists until there are only two elements in the left most sublist.
It then sorts these two elements in the sublist. It will then breakdown and sort the sublist to the right of this.
The algorithm sorts and merges the two sorted sublists together.
This is done for all sublists of all sizes sequentially until the entire list has been broken down, sorted and remerged.

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)

Repeat the above for data2 and data3 (4 + 4 = 8 points each)

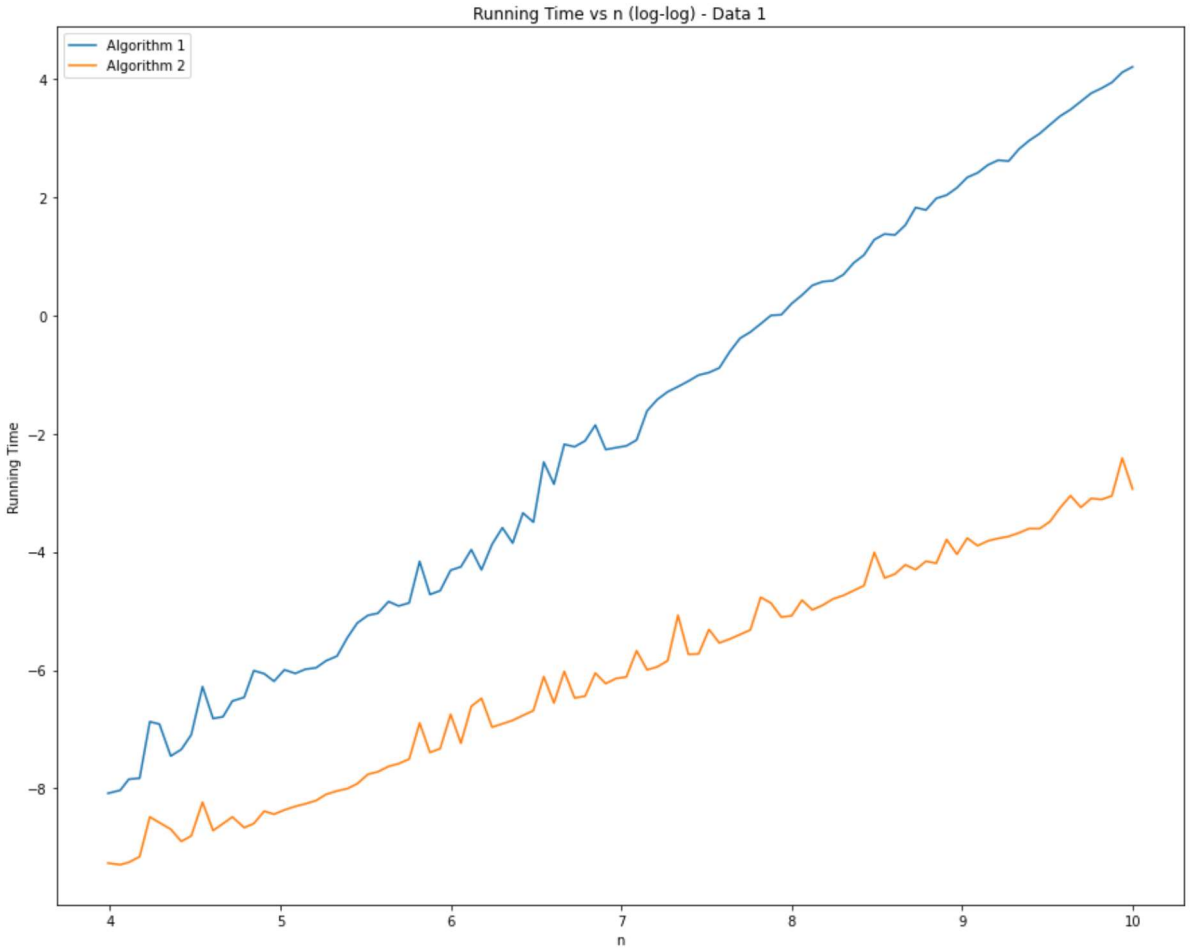
In [4]:
```
import numpy as np
import time
import matplotlib.pyplot as plt
import numpy as np
import math

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

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

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

In [5]:
```
n_log = np.logspace(4, 10, num = 100, base = math.e, dtype = int)
```

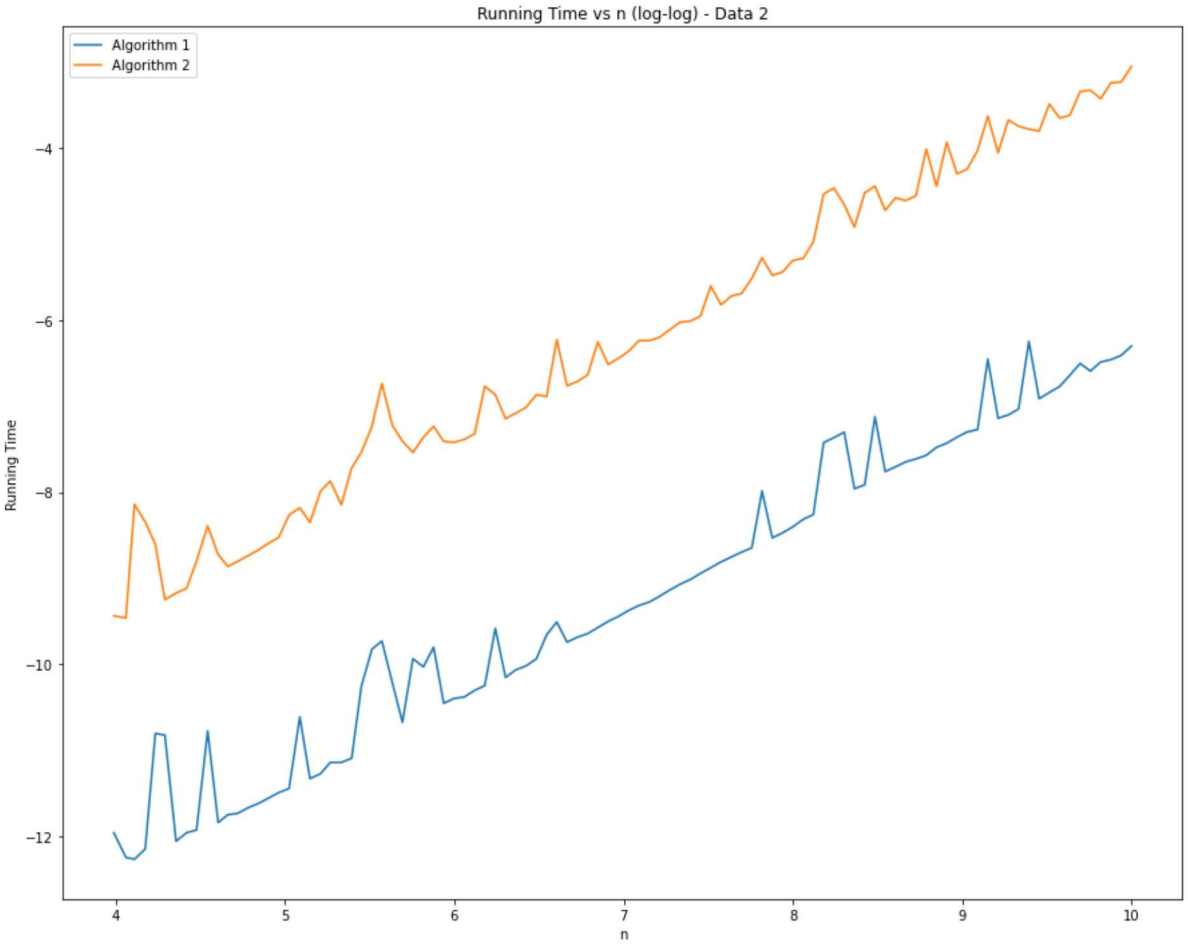
In [7]:
```
timer1_1 = []
timer2_1 = []

for i in n_log:
    data_in = data1(i)

    t_start1 = time.perf_counter()
    alg1(data_in)
    t_end1 = time.perf_counter()

    timer1_1.append(t_end1 - t_start1)

    t_start2 = time.perf_counter()
    alg2(data_in)
    t_end2 = time.perf_counter()

    timer2_1.append(t_end2 - t_start2)


timer1_1_log = np.log(timer1_1)
timer2_1_log = np.log(timer2_1)
```

In [20]:
```
plt.figure(figsize=(15, 12))

plt.plot(np.log(n_log), timer1_1_log, label = "Algorithm 1")
plt.plot(np.log(n_log), timer2_1_log, label = "Algorithm 2")

plt.title('Running Time vs n (log-log) - Data 1')
plt.xlabel('n')
plt.ylabel('Running Time')
plt.legend()
plt.show()
```



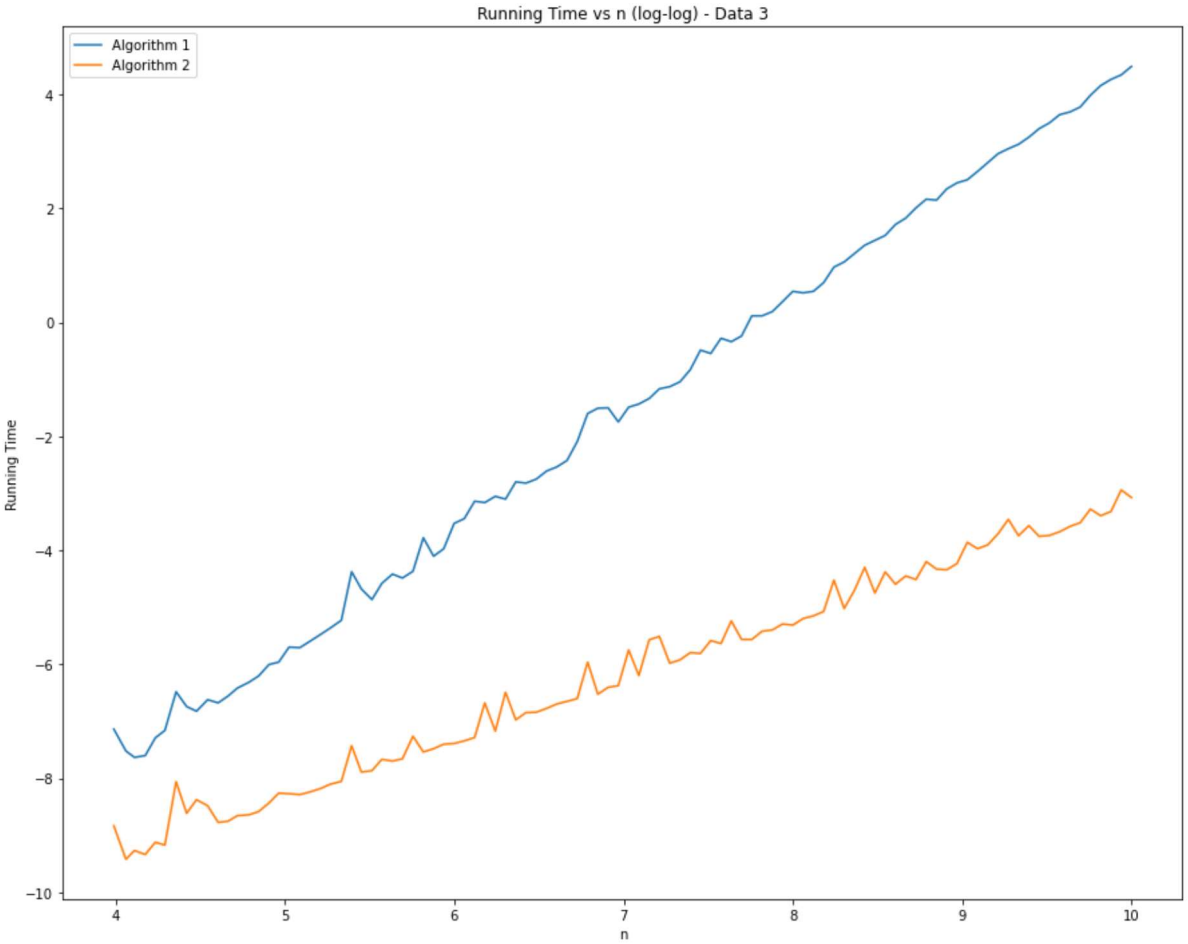
In [9]:
```
timer1_2 = []
timer2_2 = []

for i in n_log:
    data_in = data2(i)

    t_start1 = time.perf_counter()
    alg1(data_in)
    t_end1 = time.perf_counter()

    timer1_2.append(t_end1 - t_start1)

    t_start2 = time.perf_counter()
    alg2(data_in)
    t_end2 = time.perf_counter()

    timer2_2.append(t_end2 - t_start2)


timer1_2_log = np.log(timer1_2)
timer2_2_log = np.log(timer2_2)
```

In [21]:
```
plt.figure(figsize=(15, 12))

plt.plot(np.log(n_log), timer1_2_log, label = "Algorithm 1")
plt.plot(np.log(n_log), timer2_2_log, label = "Algorithm 2")

plt.title('Running Time vs n (log-log) - Data 2')
plt.xlabel('n')
plt.ylabel('Running Time')
plt.legend()
plt.show()
```



In [12]:
```
timer1_3 = []
timer2_3 = []

for i in n_log:
    data_in = data3(i)

    t_start1 = time.perf_counter()
    alg1(data_in)
    t_end1 = time.perf_counter()

    timer1_3.append(t_end1 - t_start1)

    t_start2 = time.perf_counter()
    alg2(data_in)
    t_end2 = time.perf_counter()

    timer2_3.append(t_end2 - t_start2)


timer1_3_log = np.log(timer1_3)
timer2_3_log = np.log(timer2_3)
```

In [22]:
```
plt.figure(figsize=(15, 12))

plt.plot(np.log(n_log), timer1_3_log, label = "Algorithm 1")
plt.plot(np.log(n_log), timer2_3_log, label = "Algorithm 2")

plt.title('Running Time vs n (log-log) - Data 3')
plt.xlabel('n')
plt.ylabel('Running Time')
plt.legend()
plt.show()
```



The algorithms should follow O(n^2) for algorithm 1 and O(nlogn) for algorithm 2.
This would be more apparent in the graphs if larger values of n were used.
Algorithm 1 would perform similar to Algorithm 2 with smaller datasets.
Algorithm 2 is much faster for larger datasets.

Discuss how the scaling performance compares across the three data sets. (2 points)

Data 2 returns sorted data.
Hence, algorithm 1 is faster here since it does only one iteration and then returns the list whereas algorithm 2 has to setup, break down, sort and then return.

All the other datasets are unsorted.
Hence, algorithm 2 performs better on them due to better time complexity.

Which algorithm would you recommend to use for arbitrary data and why? (2 points)

I would use algorithm 2 for arbitrary data. It will perform faster in general on any type of data.
Algorithm 1 would work fine with small data. However, it's performance will quickly drop (runtime increase quickly) with increasing data size.
Algorithm 2 will work consistently for all dataset sizes.

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

Algorithm 2 performs 3 main tasks: breaking down sublists, sorting sublists and merging

sublists.

Breaking down into sublists and then sorting each smaller sublist can be done parallely.
Merging the sublists into larger sublists cannot be done until the smaller sublists are sorted so it cannot be done parallely.
This can be repeated for every level of breakdown and rebuilding of sublists.

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).

The following code was not run on Jupyter notebook but the code and results are attached here for convenience.
It was run via command line in a .py file.

In [ ]:
```
import time
import numpy as np
import matplotlib.pyplot as plt
import multiprocessing
import math


def alg2(data):
    if len(data) <= 1:
        return data
    else:
        split = len(data) // 2
        left = iter(alg2(data[:split])) # left data
        right = iter(alg2(data[split:])) # right data
        result = []
    # note: this takes the top items off the left and right piles
        left_top = next(left)
        right_top = next(right)
    # combining the left and right data
    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)


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


def alg2_parallel(data):
    if len(data) <= 1:
        return data

    else:
        split = len(data) // 2
        with multiprocessing.Pool() as m:
            [left, right] = m.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)



if __name__ == '__main__':

    #n = 20_000
    n = 2**23
    data_in = data1(n)

    t_start_1 = time.perf_counter()
    alg2_parallel(data_in)
    t_end_1 = time.perf_counter()

    time_parallel= t_end_1 - t_start_1


    t_start_2 = time.perf_counter()
    alg2(data_in)
    t_end_2 = time.perf_counter()

    time_alg2 = t_end_2 - t_start_2


    print(time_parallel,  time_alg2)
```

Output for n = 20_000: 1.4426058999961242 and 0.0757389000209514
Output for n = 2^23: 29.96704730001511 and 52.81814590000431

The benefits of parallelization are apparent at higher values.
owever, for smaller and moderate values of n, the time for alg2 was lesser than the parallelization. This is because of the time needed to setup the multiprocessing.