BIS 634 01 (FA22): Computational Methods for Informatics : Assignment2

Exercise 1

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 Assume your friend is reasonably computer literate and is not using a vastly outdated computer or installation of Python.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.)

Response

What went wrong?

- Loading a large dataset directly into memory and performing computations on it and saving intermediate results of those computations can quickly fill up the memory.

Suggest a way of storing all the data in memory that would work.

- Method 1: Instead of using list to store the data, we can use array to store the data. Compared to array, list saves the address of the data(pointer), i.e., for each data in the dataset, the list need to store 1 data and 1 pointer, resulting in storage increase and cpu consumption.
- Method 2: Use "read in chunks" method: we can divide the large file into several small files for processing. After processing each small file, release this part of memory.

Strategy for calculating the average that would not require storing all the data in memory.

- Generator functions come in very handy if this is your problem. Many popular python libraries like Keras and TensorFlow have specific functions and classes for generators.

- Pay special attention to any large or nested loops, along with any time you are loading large datasets into your program in one fell swoop.In these cases, the best practice is often to break the work into batches, allowing the memory to be freed in between calls.
- Instead of using list to store all the data, we can define a variable sum (total weight) and a variable count to count the number of data. For each 'for loop', each line of data is added into variable sum, and the variable count increase by one and then the average can be calculated by sum divide by count(sum/count).

Exercise2

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.jsonDownload typos.json 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)

Response

Importing libraries

In [111]:
```
import bitarray
from hashlib import sha3_256, sha256, blake2b
import json
import bitarray
from tqdm import tqdm
import string
import numpy as np
import matplotlib.pyplot as plt
```

In [112]:
```
## Loading the word file

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

Implement a Bloom Filter "from scratch" using a bitarray

In [113]:
```
class BloomFilter(object):

    def __init__(self, size):

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

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

    def add_two(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_three(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_one_hash(self, test_word):
        hash1 = int(self.my_hash(test_word, self.hashsize))
        if self.bit_array[hash1] == False:
            return False
        else:
            return True

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

    def check_three_hash(self, test_word):
        hash1 = int(self.my_hash(test_word, self.hashsize))
        hash2 = int(self.my_hash2(test_word, self.hashsize))
        hash3 = int(self.my_hash3(test_word, 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:]
```

In [114]:

```
# creating a bloom filter
n = int(1e7)
blf = BloomFilter(n)
```

Checking the BloomFilter Function with a short example

In [115]:
```
word_absent = ['bluff','cheater','hate','war','humanity',
                'racism','hurt','nuke','gloomy','facebook',
                'geeksforgeeks','twitter']

word_present = ['abound','abounds','abundance','abundant','accessible',
                    'bloom','blossom','bolster','bonny','bonus','bonuses',
                    'coherent','cohesive','colorful','comely','comfort',
                    'gems','generosity','generous','generously','genial']

# word not added
testing_words = word_absent + word_present[:4]

for item in word_present:
        blf.add_one(item)


for word in testing_words:
        if blf.check_one_hash(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))
```

```
'bluff' is definitely not present!
'cheater' is definitely not present!
'hate' is definitely not present!
'war' is definitely not present!
'humanity' is definitely not present!
'racism' is definitely not present!
'hurt' is definitely not present!
'nuke' is definitely not present!
'gloomy' is definitely not present!
'facebook' is definitely not present!
'geeksforgeeks' is definitely not present!
'twitter' is definitely not present!
'abound' is probably present!
'abounds' is probably present!
'abundance' is probably present!
'abundant' is probably present!
```

Deciding the Number of HashCounts and Creating the Function to add the words

```
In [116]: def hashnumber():
              return(hashcount)

          def addingword(hashcount):
              if hashcount == "1":
                  with open('/Users/mahimakaur/Desktop/words.txt') as f:
                      for line in f:
                          word = line.strip()
                          blf.add_one(word)
              elif hashcount == "2":
                  with open('/Users/mahimakaur/Desktop/words.txt') as f:
                      for line in f:
                          word = line.strip()
                          blf.add_two(word)
              elif hashcount == "3":
                   with open('/Users/mahimakaur/Desktop/words.txt') as f:
                      for line in f:
                          word = line.strip()
                          blf.add_three(word)
```

Function to suggest spelling corrections

In [117]:
```
def suggestions(test_word):
    suggest = []
    alphabet = ['a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p',
                'q','r','s','t','u','v','w','x','y','z']
    result = check_spelling(test_word, hashcount)
    if  result == False:
        for i in range(len(test_word)):
            for a in alphabet:
                suggest_word = replace(test_word, i, a)
                suggest.append(suggest_word)
    return(suggest)


def correct_suggestion(suggestions, hashcount):
    correct_suggest = []
    if hashcount == "1":
        for n in suggestions:
            suggestresult = blf.check_one_hash(n)
            if suggestresult != False:
                correct_suggest.append(n)
    elif hashcount == "2":
        for n in suggestions:
            suggestresult = blf.check_two_hash(n)
            if suggestresult != False:
                correct_suggest.append(n)
    elif hashcount == "3":
        for n in suggestions:
            suggestresult = blf.check_three_hash(n)
            if suggestresult != False:
                correct_suggest.append(n)
    return(correct_suggest)
```

An example to Check the Word

In [118]:
```
hashcount = hashnumber() ## hashcount could be 1, 2, 3
```

In [119]:
```
hashcount = "3" #for the example I have taken 3 hashcount
```

In [120]:
```
addingword(hashcount) ##adding words to the hash
```

In [121]:
```
test_word = "floeer" #testing the word floeer
```

```
In [122]: ## Creating a function to check the spelling

          def check_spelling(test_word, hashcount):
              if hashcount == "1":
                  result = blf.check_one_hash(test_word)
                  return result
              elif hashcount == "2":
                  result = blf.check_two_hash(test_word)
                  return result
              elif hashcount == "3":
                  result = blf.check_three_hash(test_word)
                  return result
```

```
In [123]: ## Checking the spelling and good suggestions for the correct spelling

          check_spelling(test_word, hashcount)
          if check_spelling(test_word, hashcount) == False:
              suggestionlist = suggestions(test_word)
              correct_suggestionlist = correct_suggestion(suggestionlist, hashcount)
              print(correct_suggestionlist)
```

```
['floter', 'flower']
```

Downloading the typos.json file. The file consists of a list of lists of [typed_word, correct_word] pairs; exactly half of the entries are spelled correctly.

```
In [124]: with open('/Users/mahimakaur/Desktop/typos.json', 'r') as f:
              file = f.read()
              text = json.loads(file)
```

```
In [125]: # to find the lenth of the typos.json file

          print(len(text))
```

```
50000
```

Functions for Good Suggestions and Misidentified Words

In [126]:
```python
def goodsuggestion(text, hashcount):
    correction = 0
    misidentified = 0
    good_suggestion = 0
    for i in range(len(text)):
        if text[i][0] == text[i][1]:
            correction += 1
        elif text[i][0] != text[i][1]:
            if check_spelling(text[i][0], hashcount) != False:
                misidentified += 1
            else:
                suggestion = suggestions(text[i][0])
                good = correct_suggestion(suggestion, hashcount)
                for n in good:
                    if (len(good) <= 3) and (n == text[i][1]):
                        good_suggestion += 1
    return correction, misidentified, good_suggestion
```

In [127]:
```python
## Defining the BloomFilter Size

blffiltersize = np.logspace(0, 9, num=20, dtype=int)
```

Function to use one hash function

In [128]:
```python
suggestionlist1 = []
misidentified1 = []
correction1 = []
for n in range(len(blffiltersize)):
    blf = BloomFilter(int(blffiltersize[n]))
    addingword("1")
    response1 = goodsuggestion(text, "1")
    misidentified1.append(response1[1])
    suggestionlist1.append(response1[2])
    correction1.append(response1[0])

goodsuggestionaccuracy1 = []
misidentifiedaccuracy1 =[]

for i in range(len(misidentified1)):
    goodsuggestionaccuracy= (suggestionlist1[i] / 25000)*100
    misidentifiedaccuracy = (misidentified1[i] / 25000)*100
    goodsuggestionaccuracy1.append(goodsuggestionaccuracy)
    misidentifiedaccuracy1.append(misidentifiedaccuracy)
```

Function to use two hash function

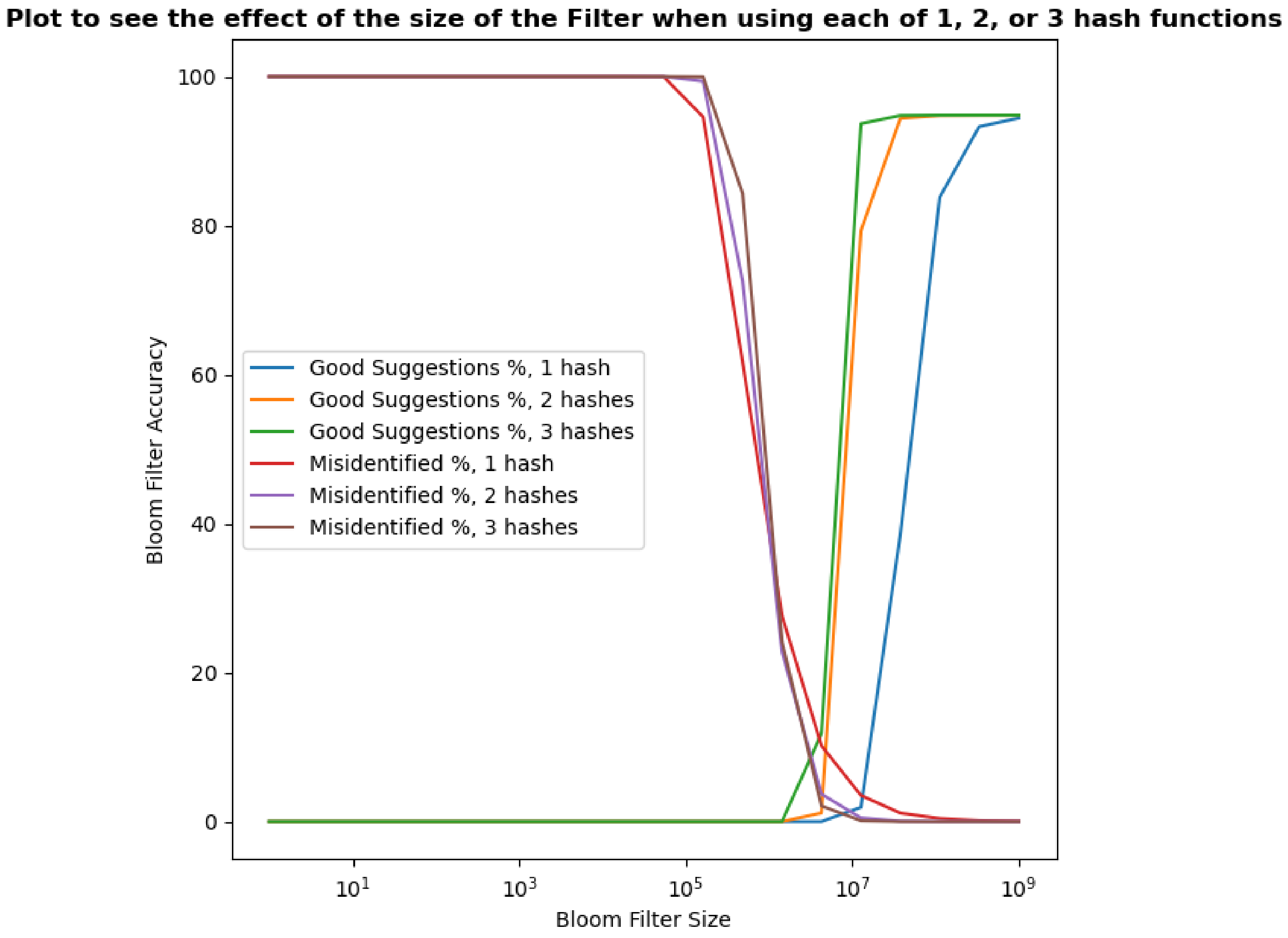
In [129]:
```
suggestionlist2 = []
misidentified2 = []
correction2 = []
for n in range(len(blffiltersize)):
    blf = BloomFilter(int(blffiltersize[n]))
    addingword("2")
    response2 = goodsuggestion(text, "2")
    misidentified2.append(response2[1])
    suggestionlist2.append(response2[2])
    correction2.append(response2[0])

goodsuggestionaccuracy2 = []
misidentifiedaccuracy2 =[]

for i in range(len(misidentified2)):
    goodsuggestionaccuracy = (suggestionlist2[i] / 25000)*100
    misidentifiedaccuracy = (misidentified2[i] / 25000)*100
    goodsuggestionaccuracy2.append(goodsuggestionaccuracy)
    misidentifiedaccuracy2.append(misidentifiedaccuracy)
```

Function to use three hash function

In [130]:
```
suggestionlist3 = []
misidentified3 = []
correction3 = []
for n in range(len(blffiltersize)):
    blf = BloomFilter(int(blffiltersize[n]))
    addingword("3")
    response3 = goodsuggestion(text, "3")
    misidentified3.append(response3[1])
    suggestionlist3.append(response3[2])
    correction3.append(response3[0])

goodsuggestionaccuracy3 = []
misidentifiedaccuracy3 =[]

for i in range(len(misidentified3)):
    goodsuggestionaccuracy = (suggestionlist3[i] / 25000)*100
    misidentifiedaccuracy = (misidentified3[i] / 25000)*100
    goodsuggestionaccuracy3.append(goodsuggestionaccuracy)
    misidentifiedaccuracy3.append(misidentifiedaccuracy)
```

Plotting the effect of the size of the Filter

In [134]:
```
## Graph to show the effect size of the filter

fig = plt.figure(figsize =(7,7))
plt.xscale('log')
plt.plot(blffiltersize, goodsuggestionaccuracy1, label="Good Suggestions %, 1 hash")
plt.plot(blffiltersize, goodsuggestionaccuracy2, label="Good Suggestions %, 2 hashes")
plt.plot(blffiltersize, goodsuggestionaccuracy3, label="Good Suggestions %, 3 hashes")
plt.plot(blffiltersize, misidentifiedaccuracy1, label="Misidentified %, 1 hash")
plt.plot(blffiltersize, misidentifiedaccuracy2, label="Misidentified %, 2 hashes")
plt.plot(blffiltersize, misidentifiedaccuracy3, label="Misidentified %, 3 hashes")
plt.xlabel('Bloom Filter Size')
plt.ylabel("Bloom Filter Accuracy")
plt.title("Plot to see the effect of the size of the Filter when using each of 1, 2, or 3 hash functions", fontweight = "bold")
plt.legend(loc="center left")
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?

- About 7 Bits are necessary for this approach to give suggestions 90% of the time when using each of 1, 2, or 3 hash functions.

Exercise 3 ¶

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).

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.

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)

Response

In [101]:
```
##importing libraries

from tqdm import tqdm #to visualize small progress bar for the process completion
import random
import time
import statistics
import matplotlib.pyplot as plt
```

Extending the Binary Tree

- Self is used to access all the instances defined within a class, including its methods and attributes.
- The insert method compares the value of the node to the parent node and decides whether to add it as a left node or right node.If the node is greater than the parent node, it is inserted as a right node; otherwise,it’s inserted left.
- PrintTree class is used to print the tree.
- Contains allows to use the in operator.

In [102]:
```
class Tree:
    def __init__(self, _value = None):
        self._value = _value
        self.left = None
        self.right = None

# Recursive function to insert a key into a Binary Search Tree

    def add(self, _value):
        if (self._value == None):
            self._value = _value
        else:
            if self._value == _value:
                self._value = _value
## if the given key is more than the root node,recur for the right subtree
            elif self._value < _value:
                if (self.right == None):
                    self.right = Tree(_value)
                else:
                    self.right.add(_value)
## otherwise, recur for the left subtree
            else:
                if (self.left == None):
                    self.left = Tree(_value)
                else:
                    self.left.add(_value)


# Print the tree

    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print(self._value),
        if self.right:
            self.right.PrintTree()

    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
```

In [103]:
```
my_tree = Tree()
for item in [55, 62, 37, 49, 71, 14, 17]:
    my_tree.add(item)
```

In [104]:
```
my_tree.PrintTree()
```

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

Testing whether the given number is in the tree or not.

In [105]:
```
print(f'my_tree.contains(55) = 'f'{55 in my_tree}')
print(f'my_tree.contains(42) = 'f'{42 in my_tree}')
```

```
my_tree.contains(55) = True
my_tree.contains(42) = False
```

Graph of time required for checking if a number is in the tree as a function of n

In [106]:
```
def tree_time(n):
    duration = []
    data_random = [] #contain random n values to be added to the tree
    random_in_data = []

    for i in range(n):
        num = random.randint(1,1000)
        data_random.append(num)

    for i in range(1000):
        num = random.randint(1,1000)
        random_in_data.append(num)

    count = 0

    while(count <= 10):
        my_tree = Tree()
        for num in data_random:
            my_tree.add(num)
        start = time.perf_counter()
        for num in random_in_data:
            num in my_tree
        end = time.perf_counter()
        duration.append(end - start)
        count = count + 1
    return statistics.median(duration)
```

Finding the run time of the tree

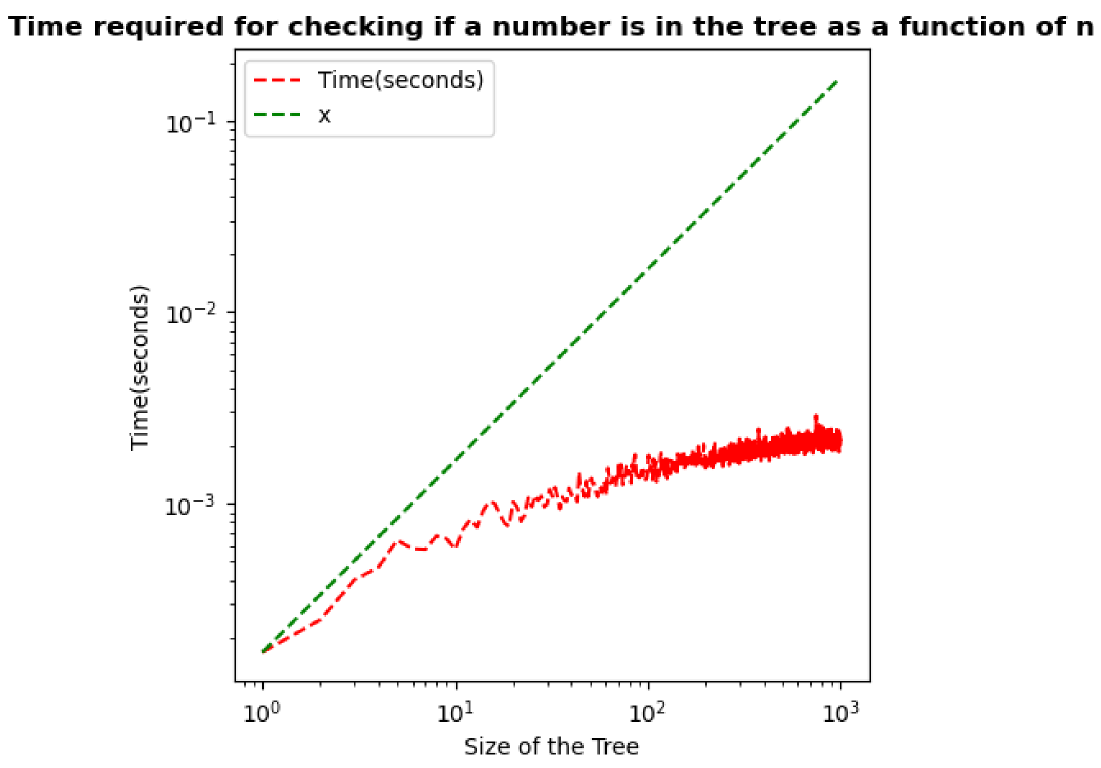
In [107]:
```
numbers = range(1, 1000) #continuous number in the range 1 to 1000
time_run = [tree_time(int(n)) for n in tqdm(numbers)]
starting_pt = time_run[0]
x = []
x2 = []
for n in tqdm(numbers):
    x.append(starting_pt * (n))
for n in tqdm(numbers):
    x2.append(starting_pt * (n**2))
```

```
100%|██████████████████████████████████████████| 999/999 [00:36<00:00, 27.09it/s]
100%|█████████████████████████████████████| 999/999 [00:00<00:00, 3007975.37it/s]
100%|█████████████████████████████████████| 999/999 [00:00<00:00, 1575821.62it/s]
```

Visualizing the time required for checking if a number is in the tree as a function of n

In [108]:
```
numbers = list(numbers)
fig = plt.figure(figsize =(5,5))
plt.loglog(numbers, time_run, 'r--', numbers, x,"g--")
plt.xlabel("Size of the Tree")
plt.ylabel("Time(seconds)")
plt.title("Time required for checking if a number is in the tree as a function of n", fontweight="bold")
plt.legend(["Time(seconds)", "x"])
plt.show
```

Out[108]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```



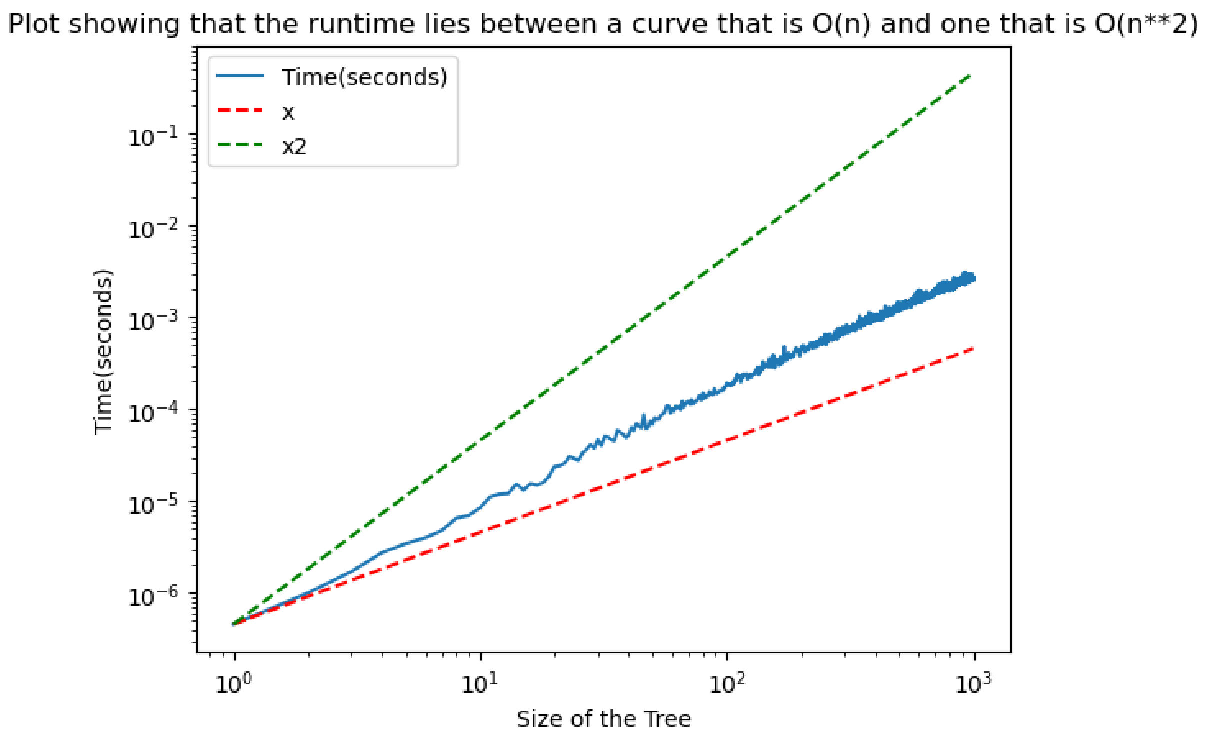
Time to setup the tree

In [109]:
```
def tree_setup_time(n):
    duration1 = []
    random_num = []
    for i in range(n):
        num = random.randint(1,1000)
        random_num.append(num)

    count = 0
    while(count <= 100):
        start = time.perf_counter()
        my_tree = Tree(random_num[0])
        for num in random_num[1:]:
            my_tree.add(num)
        end = time.perf_counter()
        duration1.append(end - start)
        count = count + 1
    return statistics.median(duration1)
```

In [110]:
```
## Checking the runtime for random tree size values

ns = range(1, 1000)
setup_run_time = [tree_setup_time(int(n)) for n in tqdm(ns)]
start_pt = setup_run_time[0]
x = []
x2 = []
for n in tqdm(ns):
    x.append(start_pt * (n))
for n in tqdm(ns):
    x2.append(start_pt * (n**2))
```

```
100%|██████████████████████████████████████| 999/999 [02:14<00:00,  7.41it/s]
100%|█████████████████████████████████| 999/999 [00:00<00:00, 3056243.40it/s]
100%|█████████████████████████████████| 999/999 [00:00<00:00, 1722198.81it/s]
```

```
In [111]: ##Plotting the curve graph

          ns = list(ns)
          plt.plot(ns, setup_run_time)
          plt.loglog(ns, x , "r--", ns, x2, "g--")
          plt.xlabel("Size of the Tree")
          plt.ylabel("Time(seconds)")
          plt.legend(["Time(seconds)","x","x2"])
          plt.title("Plot showing that the runtime lies between a curve that is O(n) and one that is O(n**2)")
          plt.show()
```



Exercise 4

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.

```
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))
```

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

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).

Response

```
In [125]: from time import perf_counter
          import matplotlib.pyplot as plt
          import numpy as np
```

Exploration of the implications of algorithms and how they interact with different data sets.

Algorithm 1

```
In [126]: 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
```

```
In [127]: Ex1 = [1,4,5,3,7,8,0]
          print(alg1(Ex1))

          Ex2 = [11,4,8,7,6,9]
          print(alg1(Ex2))

          Ex3 = [1,2,3,4,5]
          print(alg1(Ex3))
```

```
[0, 1, 3, 4, 5, 7, 8]
[4, 6, 7, 8, 9, 11]
[1, 2, 3, 4, 5]
```

Hypothesis : Alg1 is sorting the list values.

Algorithm 2

In [128]:
```
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)
```

In [129]:
```
Ex1 = [1,4,5,3,7,8,0]
print(alg2(Ex1))

Ex2 = [11,4,8,7,6,9]
print(alg2(Ex2))

Ex3 = [1,2,3,4,5]
print(alg2(Ex3))
```

```
[0, 1, 3, 4, 5, 7, 8]
[4, 6, 7, 8, 9, 11]
[1, 2, 3, 4, 5]
```

Hypothesis : Alg2 is sorting the list values.¶

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.

Algorithm 1

- It swaps interchanges the elements if the element on the left is lesser than the one on the right. After every iteration, the highest element of the list comes to the right and this continues until the list is completely sorted.

- Such sorting is called Bubble sort.
- It is one of the simplest sorting algorithms.
- Process : The two adjacent elements of a list are checked and swapped if they are in wrong order and this process is repeated until we get a sorted list.

Algorithm 2

- The second algorithm is using MergeSort startegy to sort the values.
- Merge sort is a recursive algorithm that continually splits a list in half. If the list is empty or has one item, it is sorted by definition (the base case). If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.
- Once the two halves are sorted, the fundamental operation, called a merge, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted,new list.
- To conclude, This functions divides the lists into 2 equal portions. It then compares which element of the 2 lists is smaller then append that element to the empty list called result which finally appears as the sorted list at the end of function.

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.

In [130]:
```
##Creating a variable storing various sizes of data n

list_values = np.logspace(1, 4, num=50, endpoint=True, base=10.0, dtype=None, axis=0)
print(list_values)
```

```
[   10.            11.51395399    13.25711366    15.26417967
    17.57510625    20.23589648    23.29951811    26.82695795
    30.88843596    35.56480306    40.94915062    47.14866363
    54.28675439    62.50551925    71.9685673     82.86427729
    95.40954763   109.8541142    126.48552169   145.63484775
   167.68329368   193.06977289   222.29964825   255.95479227
   294.70517026   339.32217719   390.69399371   449.8432669
   517.94746792   596.36233166   686.648845     790.60432109
   910.29817799  1048.11313415  1206.79264064  1389.49549437
  1599.85871961  1842.06996933  2120.95088792  2442.05309455
  2811.76869797  3237.45754282  3727.59372031  4291.93426013
  4941.71336132  5689.86602902  6551.2855686   7543.12006335
  8685.11373751 10000.        ]
```

A. Data1

```
In [131]: ## Function for Data1

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
```

Testing Algorithm 1

```
In [132]: y_input1 = []
          j = 0
          for i in list_values:

              data = data1(int(i))

              # Start the stopwatch / counter
              t1_start = perf_counter()

              result = alg1(data)

              t1_stop = perf_counter()

              y_input1.append(t1_stop - t1_start)
              print(y_input1)
              j += 1

          print(y_input1)
```

```
[1.5458001143997535e-05]
[1.5458001143997535e-05, 1.915999746415764e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06, 2.5419976736884564e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06, 2.5419976736884564e-06, 2.91700052912347e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06, 2.5419976736884564e-06, 2.91700052912347e-06, 2.91600008495152e-06]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06, 2.5419976736884564e-06, 2.91700052912347e-06, 2.91600008495152e-06, 2.4125001800712198e-05]
[1.5458001143997535e-05, 1.915999746415764e-06, 1.624997821636498e-06, 1.625001459615305e-06, 1.791999238776043e-06, 2.33399987
3371795e-06, 2.167002094211057e-06, 2.5419976736884564e-06, 2.91700052912347e-06, 2.91600008495152e-06, 2.4125001800712198e-05,
```

Testing Algorithm 2

In [133]:
```
y_input2 = []
j = 0
for i in list_values:

    data = data1(int(i))

    # Start the stopwatch / counter
    t1_start = perf_counter()

    result = alg2(data)

    t1_stop = perf_counter()

    y_input2.append(t1_stop - t1_start)
    print(y_input2)
    j += 1

print(y_input2)
```

```
156, 0.0003280829987488687, 0.000372416998288827]
[7.35829999030102e-05, 2.4708999262657017e-05, 1.5958001313265413e-05, 1.7916998331202194e-05, 3.570800254237838e-05, 2.3917000
36241673e-05, 2.712499917834066e-05, 3.0499999411404133e-05, 3.699999797390774e-05, 4.254200030118227e-05, 4.8999998398358e-05,
5.9333000535843894e-05, 7.558300058008172e-05, 9.28330009628553e-05, 0.00010275000022375025, 0.00011766600073315203, 0.00015124
999845284037, 0.00015825000082259066, 0.00018450000061420724, 0.0015818749998288695, 0.0002687920023163315, 0.00028720900081680
156, 0.0003280829987488687, 0.000372416998288827, 0.0004907499969704077]
[7.35829999030102e-05, 2.4708999262657017e-05, 1.5958001313265413e-05, 1.7916998331202194e-05, 3.570800254237838e-05, 2.3917000
36241673e-05, 2.712499917834066e-05, 3.0499999411404133e-05, 3.699999797390774e-05, 4.254200030118227e-05, 4.8999998398358e-05,
5.9333000535843894e-05, 7.558300058008172e-05, 9.28330009628553e-05, 0.00010275000022375025, 0.00011766600073315203, 0.00015124
999845284037, 0.00015825000082259066, 0.00018450000061420724, 0.0015818749998288695, 0.0002687920023163315, 0.00028720900081680
156, 0.0003280829987488687, 0.000372416998288827, 0.0004907499969704077, 0.0005211250027059577]
[7.35829999030102e-05, 2.4708999262657017e-05, 1.5958001313265413e-05, 1.7916998331202194e-05, 3.570800254237838e-05, 2.3917000
36241673e-05, 2.712499917834066e-05, 3.0499999411404133e-05, 3.699999797390774e-05, 4.254200030118227e-05, 4.8999998398358e-05,
5.9333000535843894e-05, 7.558300058008172e-05, 9.28330009628553e-05, 0.00010275000022375025, 0.00011766600073315203, 0.00015124
999845284037, 0.00015825000082259066, 0.00018450000061420724, 0.0015818749998288695, 0.0002687920023163315, 0.00028720900081680
156, 0.0003280829987488687, 0.000372416998288827, 0.0004907499969704077, 0.0005211250027059577, 0.0005527079993044026]
[7.35829999030102e-05, 2.4708999262657017e-05, 1.5958001313265413e-05, 1.7916998331202194e-05, 3.570800254237838e-05, 2.3917000
36241673e-05, 2.712499917834066e-05, 3.0499999411404133e-05, 3.699999797390774e-05, 4.254200030118227e-05, 4.8999998398358e-05,
5.9333000535843894e-05, 7.558300058008172e-05, 9.28330009628553e-05, 0.00010275000022375025, 0.00011766600073315203, 0.00015124
999845284037, 0.00015825000082259066, 0.00018450000061420724, 0.0015818749998288695, 0.0002687920023163315, 0.00028720900081680
```

B. Data2

In [134]:
```
def data2(n):
    return list(range(n))
```

Testing for Algorithm 1 and Algorithm 2

In [135]:
```
y_inputB = []
j = 0
for i in list_values:

    data = data2(int(i))

    # Start the stopwatch / counter
    t1_start = perf_counter()

    result = alg1(data)

    t1_stop = perf_counter()

    y_inputB.append(t1_stop - t1_start)
    print(y_inputB)
    j += 1

print(y_inputB)

y_inputC = []
j = 0
for i in list_values:

    data = data2(int(i))

    # Start the stopwatch / counter
    t1_start = perf_counter()

    result = alg2(data)

    t1_stop = perf_counter()

    y_inputC.append(t1_stop - t1_start)
    print(y_inputC)
    j += 1

print(y_inputC)
```

```
[6.24999993306119e-05]
[6.24999993306119e-05, 1.4170000213198364e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
734481692e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
734481692e-06, 2.0409970602486283e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
734481692e-06, 2.0409970602486283e-06, 2.125001628883183e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
734481692e-06, 2.0409970602486283e-06, 2.125001628883183e-06, 4.0000013541430235e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
```

```
734481692e-06, 2.0409970602486283e-06, 2.125001628883183e-06, 4.0000013541430235e-06, 3.2079988159239292e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
734481692e-06, 2.0409970602486283e-06, 2.125001628883183e-06, 4.0000013541430235e-06, 3.2079988159239292e-06, 3.084001946263015
3e-06]
[6.24999993306119e-05, 1.4170000213198364e-06, 1.290998625336215e-06, 1.5000005078036338e-06, 1.5840014384593815e-06, 1.7499987
```

Data3

In [136]:
```python
def data3(n):
    return list(range(n, 0, -1))
```

Testing for Algorithm 1 and Algorithm 2

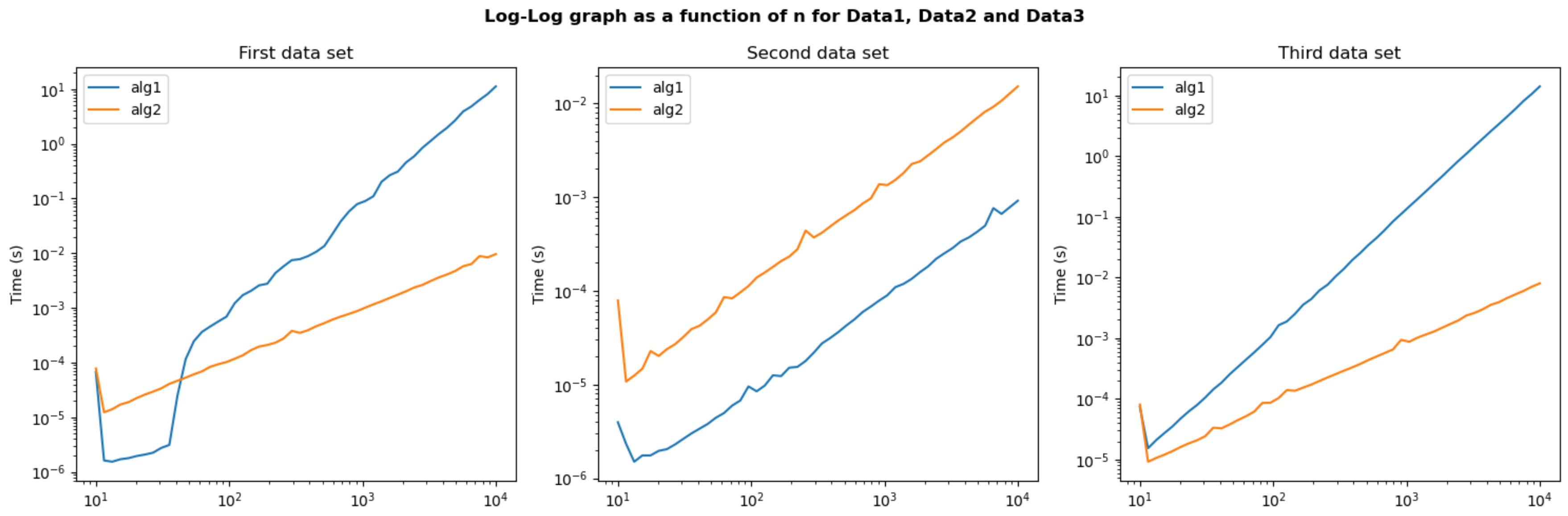
In [137]:
```python
y_inputD = []
j = 0
for i in list_values:

    data = data3(int(i))

    # Start the stopwatch / counter
    t1_start = perf_counter()

    result = alg1(data)

    t1_stop = perf_counter()

    y_inputD.append(t1_stop - t1_start)
    print(y_inputD)
    j += 1

print(y_inputD)

y_inputE = []
j = 0
for i in list_values:

    data = data3(int(i))

    # Start the stopwatch / counter
    t1_start = perf_counter()

    result = alg2(data)

    t1_stop = perf_counter()

    y_inputE.append(t1_stop - t1_start)
    print(y_inputE)
    j += 1

print(y_inputE)
```

```
[7.2750000981614e-05, 1.5250003343680874e-05, 2.0666000636992976e-05, 2.7207999664824456e-05, 3.4583001252030954e-05, 4.7874997
108010575e-05, 6.245900294743478e-05, 7.916599861346185e-05, 0.00010479099728399888, 0.00014287500016507693, 0.0001859580006566
83, 0.0002534160012146458, 0.0003342080017318949, 0.0004402089980430901, 0.0005755409983976278, 0.0010453750001033768, 0.001056
6249984549358, 0.0013470000012603123, 0.0018173749995185062, 0.002659708999999566, 0.003182042000844376, 0.004265999999915948,
0.005854082999576349, 0.0077642079995712265, 0.010365999998612097]
[7.2750000981614e-05, 1.5250003343680874e-05, 2.0666000636992976e-05, 2.7207999664824456e-05, 3.4583001252030954e-05, 4.7874997
108010575e-05, 6.245900294743478e-05, 7.916599861346185e-05, 0.00010479099728399888, 0.00014287500016507693, 0.0001859580006566
83, 0.0002534160012146458, 0.0003342080017318949, 0.0004402089980430901, 0.0005755409983976278, 0.0010453750001033768, 0.001056
6249984549358, 0.0013470000012603123, 0.0018173749995185062, 0.002659708999999566, 0.003182042000844376, 0.004265999999915948,
0.005854082999576349, 0.0077642079995712265, 0.010365999998612097, 0.013941124998382293]
[7.2750000981614e-05, 1.5250003343680874e-05, 2.0666000636992976e-05, 2.7207999664824456e-05, 3.4583001252030954e-05, 4.7874997
108010575e-05, 6.245900294743478e-05, 7.916599861346185e-05, 0.00010479099728399888, 0.00014287500016507693, 0.0001859580006566
83, 0.0002534160012146458, 0.0003342080017318949, 0.0004402089980430901, 0.0005755409983976278, 0.0010453750001033768, 0.001056
6249984549358, 0.0013470000012603123, 0.0018173749995185062, 0.002659708999999566, 0.003182042000844376, 0.004265999999915948,
0.005854082999576349, 0.0077642079995712265, 0.010365999998612097, 0.013941124998382293, 0.018847874998755287]
```

```
[7.2750000981614e-05, 1.5250003343680874e-05, 2.0666000636992976e-05, 2.7207999664824456e-05, 3.4583001252030954e-05, 4.7874997
108010575e-05, 6.245900294743478e-05, 7.916599861346185e-05, 0.00010479099728399888, 0.00014287500016507693, 0.0001859580006566
83, 0.0002534160012146458, 0.0003342080017318949, 0.0004402089980430901, 0.0005755409983976278, 0.0010453750001033768, 0.001056
6249984549358, 0.0013470000012603123, 0.0018173749995185062, 0.002659708999999566, 0.003182042000844376, 0.004265999999915948,
```

Ploting a log-log graph as a function of n for Data1, Data2 and Data3

In [116]:
```
fig, (ax1, ax2, ax3) = plt.subplots(nrows=1, ncols=3, figsize=(15,5))

ax1.loglog(list_values, y_input1,label="alg1")
ax1.loglog(list_values, y_input2,label="alg2")
ax1.set_ylabel('Time (s)')
ax1.set_title('First data set')
ax1.legend()

ax2.loglog(list_values, y_inputB,label="alg1")
ax2.loglog(list_values, y_inputC,label="alg2")
ax2.set_ylabel('Time (s)')
ax2.set_title('Second data set')
ax2.legend()

ax3.loglog(list_values, y_inputD,label="alg1")
ax3.loglog(list_values, y_inputE,label="alg2")
ax3.set_ylabel('Time (s)')
ax3.set_title('Third data set')
ax3.legend()

fig.suptitle('Log-Log graph as a function of n for Data1, Data2 and Data3',fontweight ="bold")
fig.tight_layout()
plt.show()
```



Scaling performance compares across the three data sets. Which algorithm would you recommend to use for arbitrary data and why?

Comparing Alg1 & Alg2 for Data1

- Data1 is a list of random numbers. From the graph we can see that the alg2 takes more time than alg1 on small dataset to sort the values. While on large datasets the alg2 works more efficiently i.e. thats less time to sort the values.

Comparing Alg1 & Alg2 for Data2

- From the graph we can see that the alg2 takes more time than alg1 to sort the values irrespectively of the size of the data points. The data2 is a list of numbers in ascending order. So the list is alreday sorted. Alg1 works better on already sorted list.

Comparing Alg1 & Alg2 for Data3

- From the graph we can see that the alg1 takes more time than alg2 to sort the values irrespectively of the size of the data points. The data3 is in descending order which means that we would take time to sort the values in ascending order as it will have to reverse the order. In this case, alg2 is more efficient than alg1.

Algorithm2 would be recommended to use for arbitrary data. In the first dataset we have a list of random numbers. For the the random number dataset Alg2 takes less time to sort the values.

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

Response

- We know that the alg2 uses MergeSort strategy to sort the values. Merge sort uses the intuition of recursion method.Merge sort is a divide and conquer algorithm. Given an input array, it divides it into halves and each half is then applied the same division method until individual elements are obtained. A pairs of adjacent elements are then merged to form sorted sublists until one fully sorted list is obtained.
- We want merge sort to run on a parallel computing platform.The key to designing parallel algorithms is to find the operations that could be carried out simultaneously i.e we examine a known sequential algorithm and look for possible simultaneous operations.
- We can parallelize the processing of left and right splits which are the independent tasks. We can assume we have an unlimited number of processors. Looking at the merge sort algorithm tree in the sequential algorithm we can try to assign simultaneous operations to separate processors which will work concurrently to do the dividing and merging at each level. After parallelization, use join() method before merging to ensure the left and right sublists are in order.
- We can also use Parallel Quick Sort strategy. Parallel quick sort is very similar to parallel selection. We randomly pick a pivot and sort the input list into two sublists based on their order relationship with the pivot. However, instead of discarding one sublist, we continue this process recursively until the entire list has been sorted.

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).

Note : The code was run in a standalone file. .py file has been uploaded to the repository.

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

## Data as given the assignment

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

## Algorithm which needs to be parallelized

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)
```

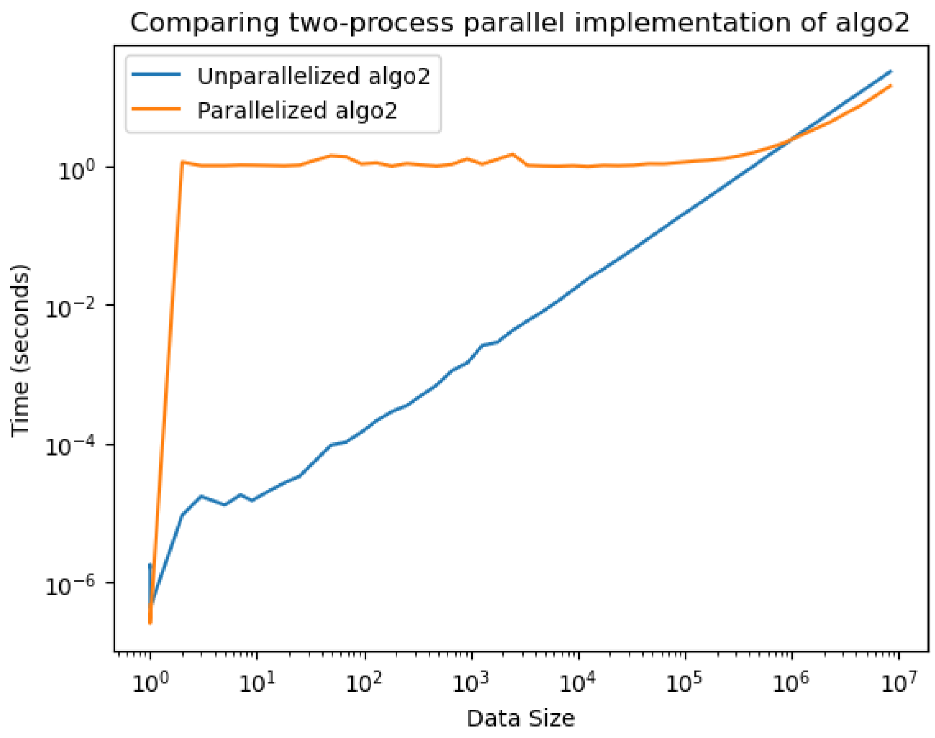
```
def parallel_alg2(data):
    if len(data) <= 1:
        return data
    else:
        split = len(data) // 2
        with multiprocessing.Pool() as p:   ##using multiprocessing to parallelize the two independent tasks
            [left, right] = p.map(alg2, [data[:split], data[split:]])
        left = 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)



if __name__ == '__main__':
    data_var = np.logspace(0, 23, base=2, dtype=int)
    duration = [] #time by the parallelized algo
    algo2d1time = [] #time by the unparallelized algo

    for n in tqdm(data_var):
        data_set = data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1)
        algo2d1_start = perf_counter()
        alg2(data_set)     ## time taken by the unparallelized algo
        algo2d1_stop= perf_counter()
        algo2d1time.append(algo2d1_stop - algo2d1_start)
        start_time = perf_counter()
        parallel_alg2(data_set)    #time taken by the parallelized algo
        stop_time = perf_counter()
        duration.append(stop_time - start_time)
        print(duration[-1],  algo2d1time[-1])
    plt.loglog(data_var, algo2d1time, label = "Unparallelized algo2")
    plt.loglog(data_var, duration, label = "Parallelized algo2")
    plt.title("Comparing two-process parallel implementation of algo2")
    plt.legend()
    plt.show()
```

In [1]:
```
from PIL import Image
```

In [17]:
```
try:
    img  = Image.open("/Users/mahimakaur/Desktop/Computional Methods for Health Informatics/Parallelized.png")
except IOError:
    pass
```

In [18]:
```
img
```

Out[18]:



Findings : From the graph we can see the initially for small dataset the mergesort algorithms works faster and is more efficient. But as the dataset size increases to millions the parallelized algorithms takes less time to sort the values i.e. it works about 1.5 times faster than the usual mergesort.