Exercise 1

1.1 Explain what went wrong (5 points).

The reason why the system could not store the data is because after the data is loaded to the system, the file is being stored using 32 bytes: referent count(8 bytes), data structure(8 bytes), data itself(8 bytes) the storage would be 16GB in total, which exceeds the memory capacity.

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

A way to solve this issue is to store the entries as strings and inside a single data structure. Because strings are stored based on 4-bytes and storing in array would eliminate the additional 8 bytes storage of data structure, eventually storing the data only takes about 4 GB.

1.3 Suggest a strategy

In [1]:
```
# with open('weights.txt') as f:
#     sum = 0
#     count = 0
#     for line in f:
#         sum += float(line)
#         count += 1
# print("average =", sum / count )
```

Exercise 2

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

In [2]:
```
#load given functions
with open('words.txt') as f:
    for line in f:
        word = line.strip()
        # do something with the word here
```

In [3]:
```
import bitarray
```

In [4]:
```
from hashlib import sha3_256, sha256, blake2b
def my_hash(s, size):
    return int(sha256(s.lower().encode()).hexdigest(), 16) % size

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

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

In [5]:
```
data = bitarray.bitarray(5)
```

In [6]:
```
data
```

Out[6]:
```
bitarray('01100')
```

In [7]:
```
def bloomFilter(size,my_hash_n):
  data = bitarray.bitarray(size)
  data.setall(0)


  with open('words.txt') as f:
    for line in f:
        word = line.strip()
        temp_i = my_hash_n(word,size)
        data[temp_i] = 1
    return data
```

2.2 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)

In [8]:
```
alphabets = 'abcdefghijklmnopqrstuvwxyz'
def spell_check(word,filter, my_hash_n):
  size = len(filter)
  sug_w = []
  for i in range(len(word)):
    if word[i] in alphabets:
      for letter in alphabets: #for each letter plugged to the word provided
        temp_w = word[:i] + letter + word[i+1:]
        temp_i = my_hash_n(temp_w,size)
        if filter[temp_i] :
          sug_w.append(temp_w)
  return set(sug_w) #return suggested matched word by hash function
```

2.3 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)

In [13]:
```
import json
sizes = [10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8, 10**9]
import numpy as np
mis_clas = np.zeros((3, len(sizes)))
good_rep = np.zeros((3, len(sizes)))
```

In [10]:
```
def return_bits(word, filter, my_hash_n): #check if word is inside the diction
ary
    temp_i = my_hash_n(word, len(filter))
    return filter[temp_i]
```

In [27]:
```
return_bits('flooer',bloomFilter(sizes[0],my_hash),my_hash)
len(spell_check('flooer',bloomFilter(sizes[0],my_hash),my_hash))
```

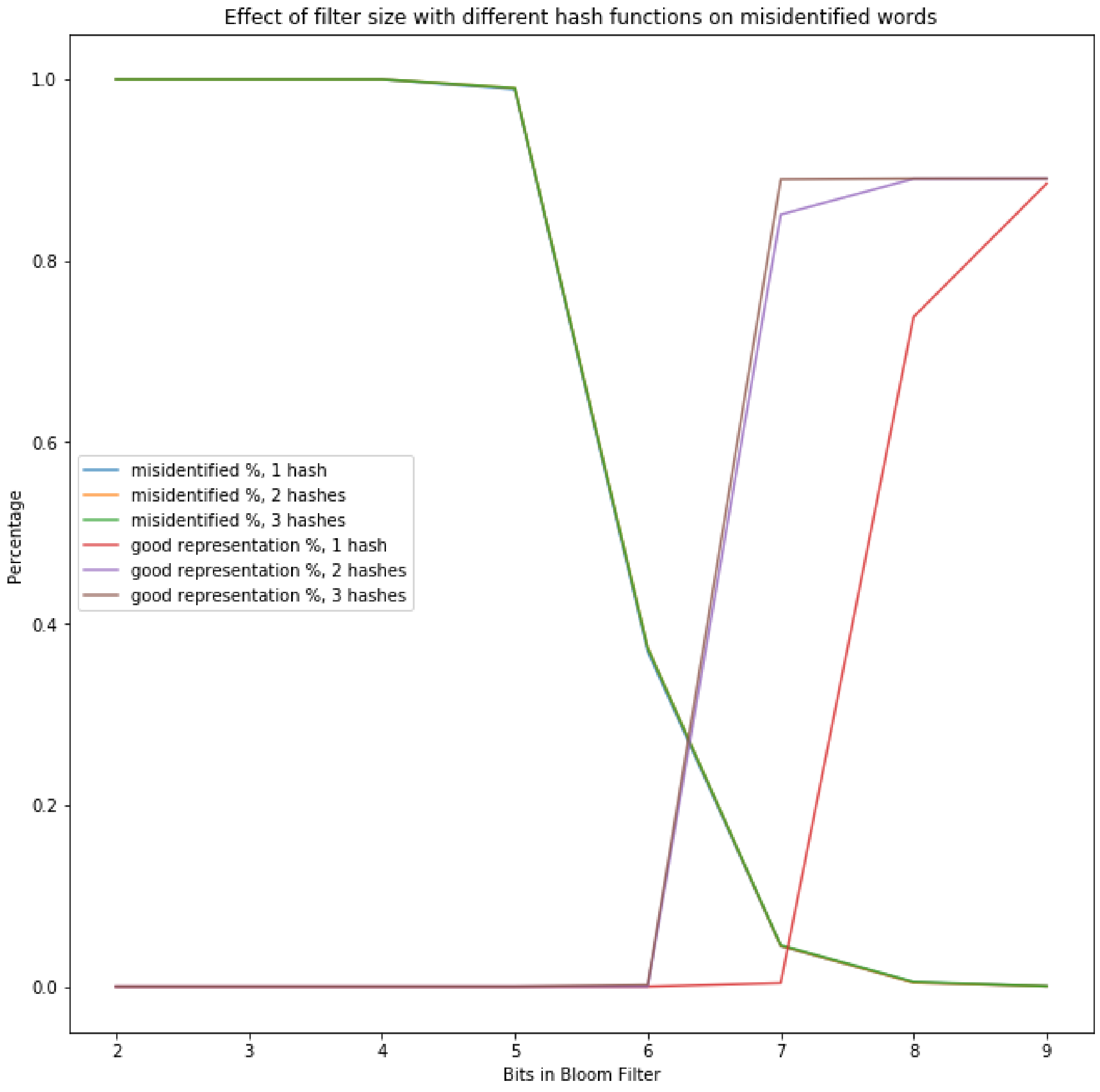
Out[27]:
```
156
```

In [11]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
from time import perf_counter
from multiprocessing import Process
```

In [14]:
```
for i in range(len(sizes)):
    filter1 = bloomFilter(sizes[i], my_hash)
    filter2 = bloomFilter(sizes[i], my_hash2)
    filter3 = bloomFilter(sizes[i], my_hash3)

    with open('typos.json', 'r') as f:
        typos = json.load(f)
        w_count = len(typos)

        for [typed, correct] in typos:

            if return_bits(typed, filter1, my_hash) and typed != correct:
                mis_clas[0][i] += 1
            if return_bits(typed, filter2, my_hash2) and typed != correct:
                mis_clas[1][i] += 1
            if return_bits(typed, filter3, my_hash3) and typed != correct:
                mis_clas[2][i] += 1

            sug_arr1 = spell_check(typed, filter1, my_hash)
            sug_arr2 = sug_arr1.intersection(spell_check(typed, filter2, my_ha
sh2))
            sug_arr3 = sug_arr2.intersection(spell_check(typed, filter3, my_ha
sh3))

            if correct in sug_arr1 and len(sug_arr1) <= 3:
                good_rep[0][i] += 1
            if correct in sug_arr2 and len(sug_arr2) <= 3:
                good_rep[1][i] += 1
            if correct in sug_arr3 and len(sug_arr3) <= 3:
                good_rep[2][i] += 1
```

In [39]:
```
x = np.log10(sizes)
fig1, ax1 = plt.subplots(figsize=(11, 11))
plt.plot(x, mis_clas[0]/(w_count/2), label = 'misidentified %, 1 hash', alpha
= 0.8)
plt.plot(x, mis_clas[1]/(w_count/2), label = 'misidentified %, 2 hashes', alph
a = 0.8)
plt.plot(x, mis_clas[2]/(w_count/2), label = 'misidentified %, 3 hashes', alph
a = 0.8)
plt.plot(x, good_rep[0]/w_count,  label = 'good representation %, 1 hash', alp
ha = 0.8)
plt.plot(x, good_rep[1]/w_count,  label = 'good representation %, 2 hashes', a
lpha = 0.8)
plt.plot(x, good_rep[2]/w_count,  label = 'good representation %, 3 hashes', a
lpha = 0.8)
plt.legend()
plt.xlabel('Bits in Bloom Filter')
plt.ylabel('Percentage')
plt.title('Effect of filter size with different hash functions on misidentifie
d words')
```

Out[39]:
```
Text(0.5, 1.0, 'Effect of filter size with different hash functions on miside
ntified words')
```



In [115]:
```python
#self check

filter1 = bloomFilter(10**7, my_hash)
filter2 = bloomFilter(10**7, my_hash2)
filter3 = bloomFilter(10**7, my_hash3)
sug_arr1 = spell_check('floeer', filter1, my_hash)
sug_arr2 = sug_arr1.intersection(spell_check('floeer', filter2, my_hash2))
sug_arr3 = sug_arr2.intersection(spell_check('floeer', filter3, my_hash3))
```

```
In [116]: sug_arr1
```

```
Out[116]: {'bloeer',
           'floees',
           'floeqr',
           'flofer',
           'floter',
           'flower',
           'fyoeer',
           'qloeer'}
```

```
In [117]: sug_arr2
```

```
Out[117]: {'floter', 'flower'}
```

```
In [118]: sug_arr3
```

```
Out[118]: {'floter', 'flower'}
```

Exercise 3

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

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

3.1 and 3.2 are written together

In [31]:
```
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
        elif self._value < item:
            if self.right:
                self.right.add(item)
            else:
                right_children = Tree()
                right_children._value = item
                self.right = right_children
        elif self._value > item:
            if self.left:
                self.left.add(item)
            else:
                left_children = Tree()
                left_children._value = item
                self.left = left_children


    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 [29]:
```
my_tree = Tree()
for item in [55, 62, 37, 49, 71, 14, 17]:
    my_tree.add(item)
```

In [37]:
```
print(71 in my_tree)
print(42 in my_tree)
```

```
True
False
```

3.3 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).

setup a repeated test for recording the time to setup the trees, based on n nodes; recording the time to search, based on n nodes; plot log-log plot to show the time based on different notes

In [64]:
```
#set up arrays and sizes used for the test
n_sizes = [1,10, 10**2, 10**3, 10**4]
t_setup = np.zeros(len(n_sizes))
t_tests = np.zeros(len(n_sizes))
```

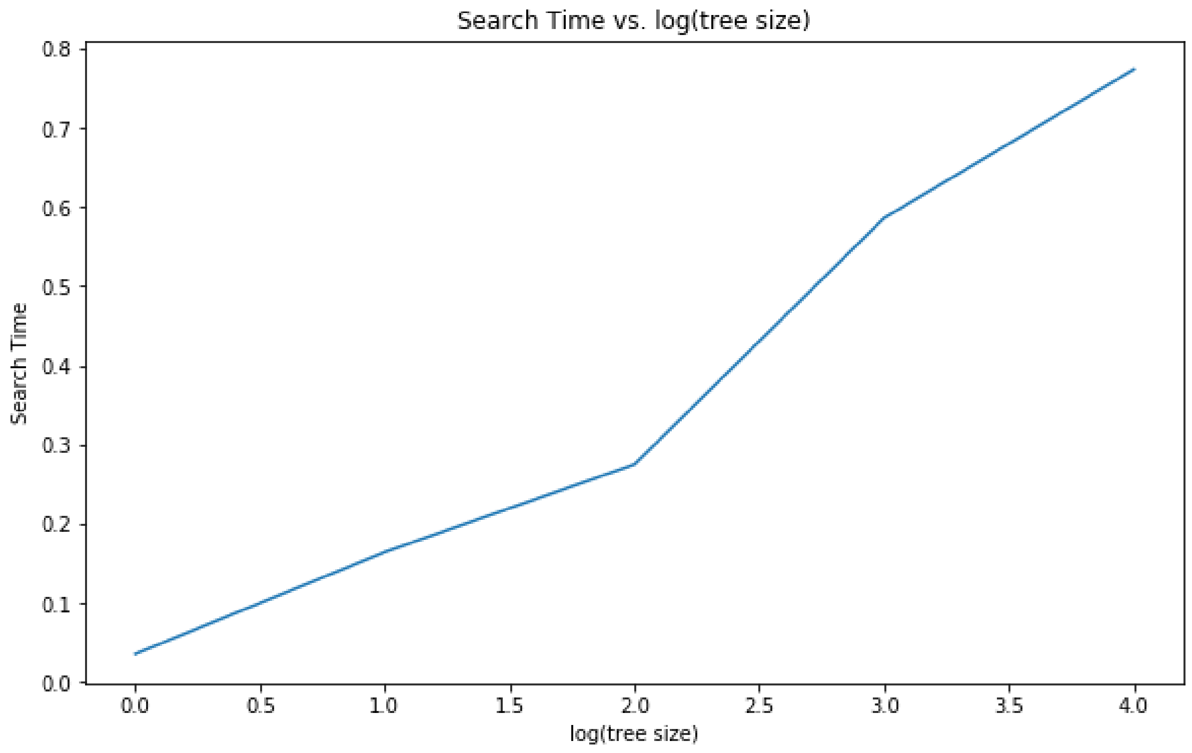
In [65]:
```
for i,n in enumerate(n_sizes):
  data = np.random.randint(100000, size = n)
  tree_test = Tree()

  # record setup time
  checkpoint1 = perf_counter()
  for d in data:
    tree_test.add(d)
  checkpoint2 = perf_counter()
  t_setup[i]=(checkpoint2-checkpoint1)

  #record search time
  data2 = np.random.randint(100000, size = 100000)
  checkpoint3 = perf_counter()
  for t in data2:
    t in tree_test
  checkpoint4 = perf_counter()
  t_tests[i] = (checkpoint4-checkpoint3)
```

In [52]:
```
t_tests
```

Out[52]:
```
array([0.0673578, 0.218272 , 0.4471565, 0.8211947, 1.1618924, 1.8058084,
       1.7053411, 1.8781003])
```

In [66]:
```
fig3, ax3 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(n_sizes), t_tests)
plt.xlabel('log(tree size)')
plt.ylabel('Search Time')
plt.title('Search Time vs. log(tree size)')
```

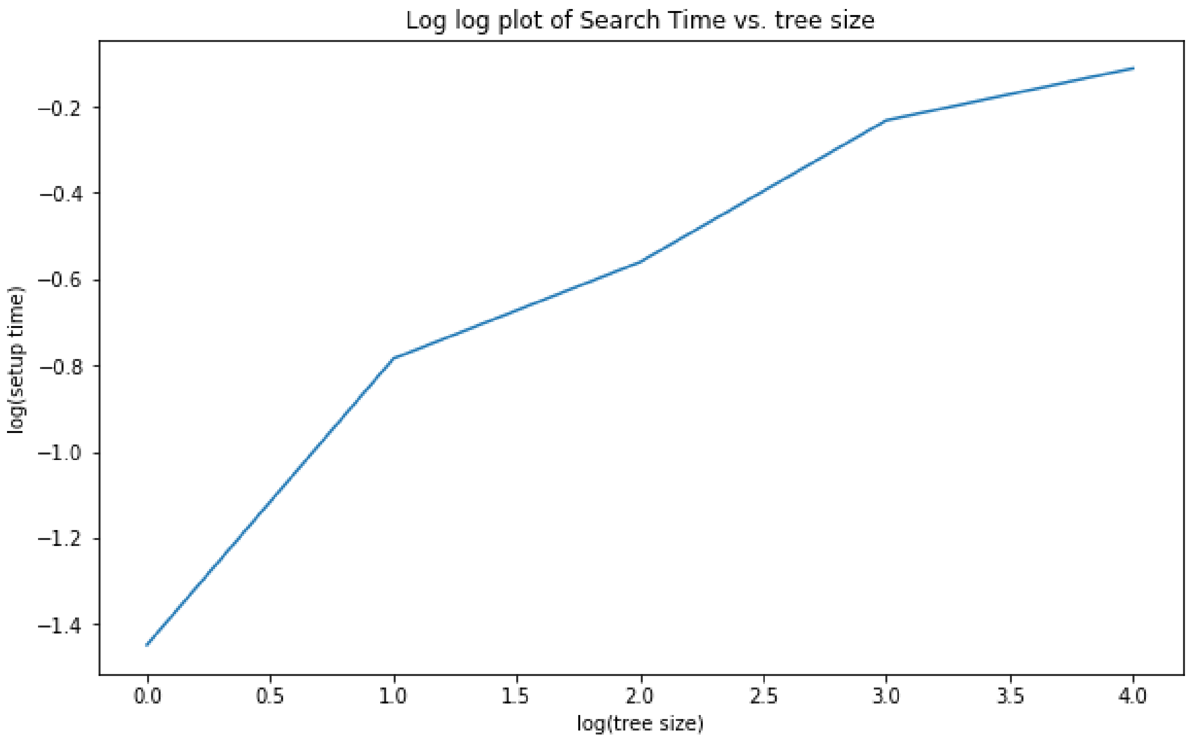
Out[66]:
```
Text(0.5, 1.0, 'Search Time vs. log(tree size)')
```



This graph shows: when n increases, log(n) increasing accordingly, the time for search increase roughly in a slope of 1, almost horizontal. This is indicative of time complexitiy of search is O(log(n)).

```
In [119]: fig4, ax4 = plt.subplots(figsize=(10, 6))
          plt.plot(np.log10(n_sizes), np.log10(t_tests))
          plt.xlabel('log(tree size)')
          plt.ylabel('log(setup time)')
          plt.title('Log log plot of Search Time vs. tree size')
```

```
Out[119]: Text(0.5, 1.0, 'Log log plot of Search Time vs. tree size')
```



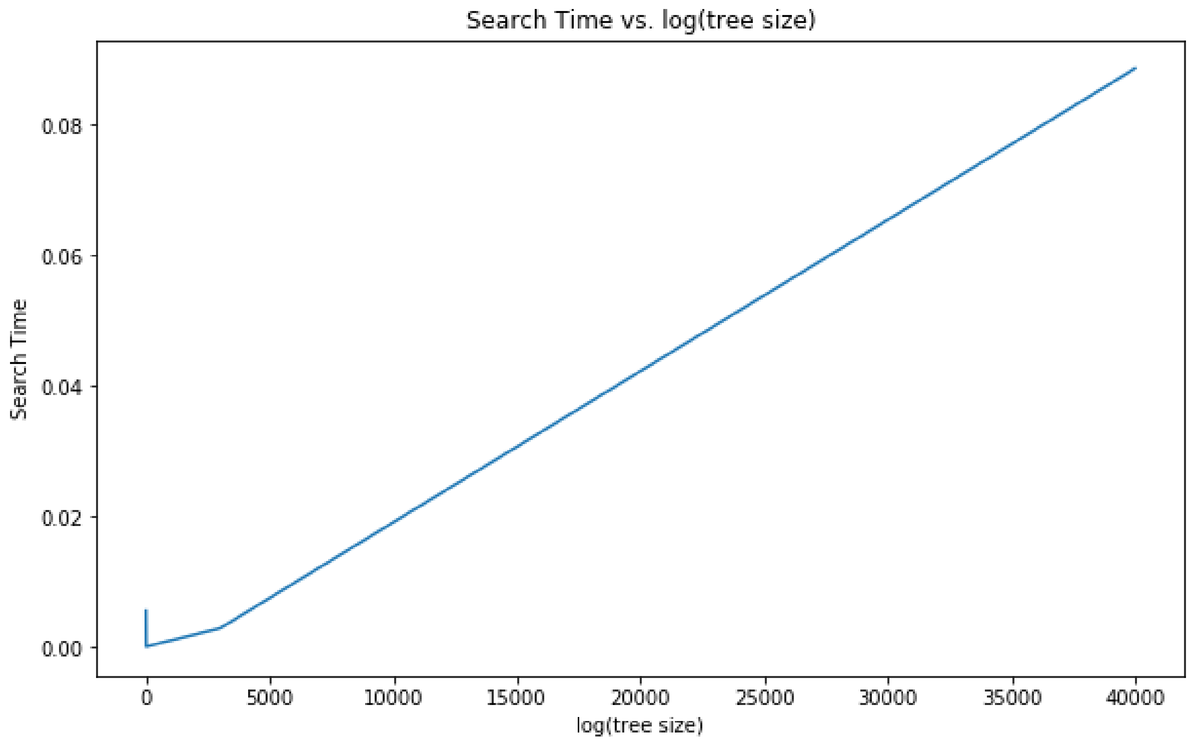
This graph shows: plotting log(n) vs. log(t), where t follows log(n), we are basically plotting log(n) vs. log(log(n)), which is the same as n vs. log(n). So the slope should be following a trend of y = log(x), which is what our grapph are showing. When n gets large, where log(tree size) = 4, we see that the slope is becoming flat and almost horizontal. If n gets large enough we should expect to see a horizontal line.

3.4 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.

In [68]:
```
fig5, ax5 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(n_sizes)*np.array(n_sizes), t_setup)
plt.xlabel('log(tree size)')
plt.ylabel('Search Time')
plt.title('Search Time vs. log(tree size)')
```

Out[68]:
```
Text(0.5, 1.0, 'Search Time vs. log(tree size)')
```



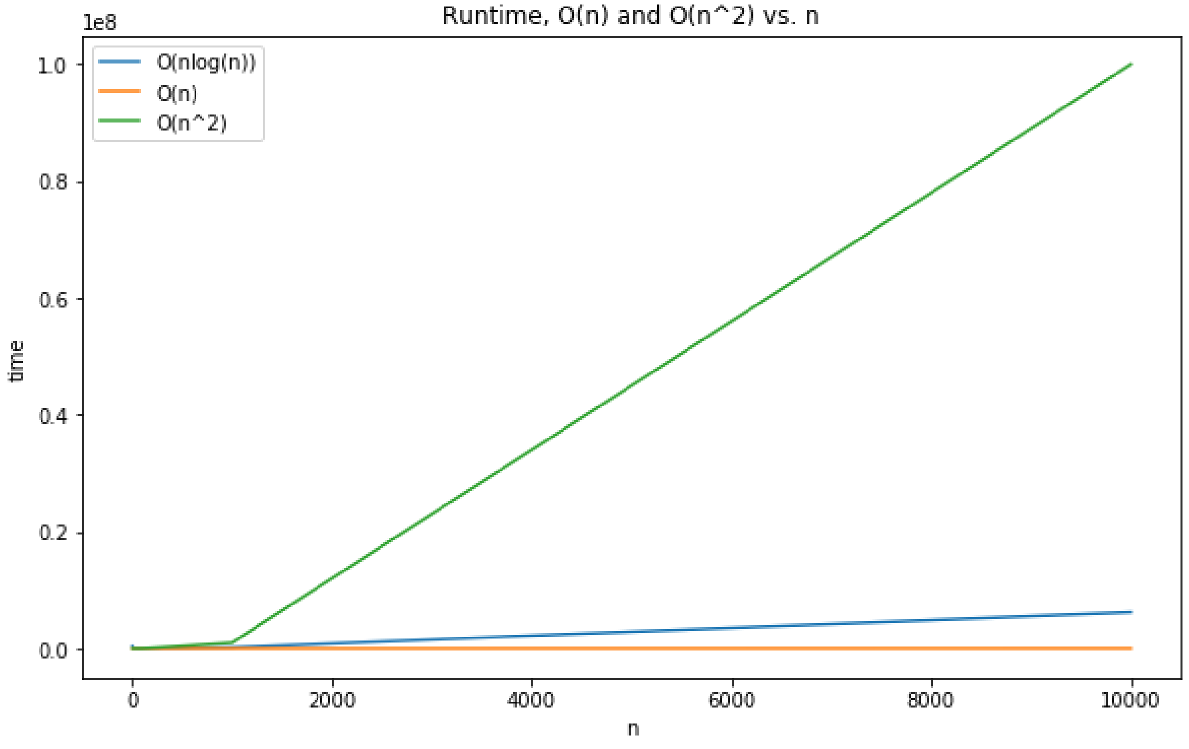
When we plot log(n) vs. t, we see a line with roughly slope of a constant. this indicates t follows time complexity of O(nlog(n))--> plot of log(n) vs. nlog(n)--> 1 vs. n --> y = nx. So this graph provides a evidence of time complexity of O(nlog(n)).

In [72]:
```
arr = np.array(n_sizes)*np.array(n_sizes)
arr
```

Out[72]:
```
array([        1,       100,     10000,   1000000, 100000000])
```

In [76]:
```
fig6, ax6 = plt.subplots(figsize=(10, 6))
plt.plot(n_sizes, np.array(t_setup)*70000000, label = "O(nlog(n))")
plt.plot(n_sizes, n_sizes,  label = "O(n)")
plt.plot(n_sizes, arr,  label = "O(n^2)")
plt.legend()
plt.xlabel('n')
plt.ylabel('time')
plt.title('Runtime, O(n) and O(n^2) vs. n')
```

Out[76]:
```
Text(0.5, 1.0, 'Runtime, O(n) and O(n^2) vs. n')
```



The plot is created after scaling the setup time with a coefficient to visualize it. We can see runtime lies between a curve that is O(n) and one that is O(n**2)

Exercise 4

In [77]:
```
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 [78]:
```
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)
```

4.1 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)

Tests are as follows:

In [80]:
```
print(alg1([55, 62, 37, 49, 71, 14, 17]))
print(alg2([55, 62, 37, 49, 71, 14, 17]))
```

```
[14, 17, 37, 49, 55, 62, 71]
[14, 17, 37, 49, 55, 62, 71]
```

In [81]:
```
print(alg1([6,5,4,1,2,3]))
print(alg2([6,5,4,1,2,3]))
```

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

These two functions are taking numbers in list with any order and returning a list that contains all numbers sorted in ascending order

4.2 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)

In alg1: we used the notation of changes to swtich the positions of numbers. For each entry in the provided list, if the entry is larger than the last entry in the sorted list, we would swap the positions between the last element in the sorted list. This will iterate until no entry violates the sorted order.

In alg2: we first would try to split the list into individual entries; then we proceeded to assigning entries to two different list, left and right list. For each two numbers implemented, we compare if the left number is smaller then the right number: if so, we add each of them to left and right list; if not, we add them to the opposite list. Iterate until all entries are appended and thus, sorted.

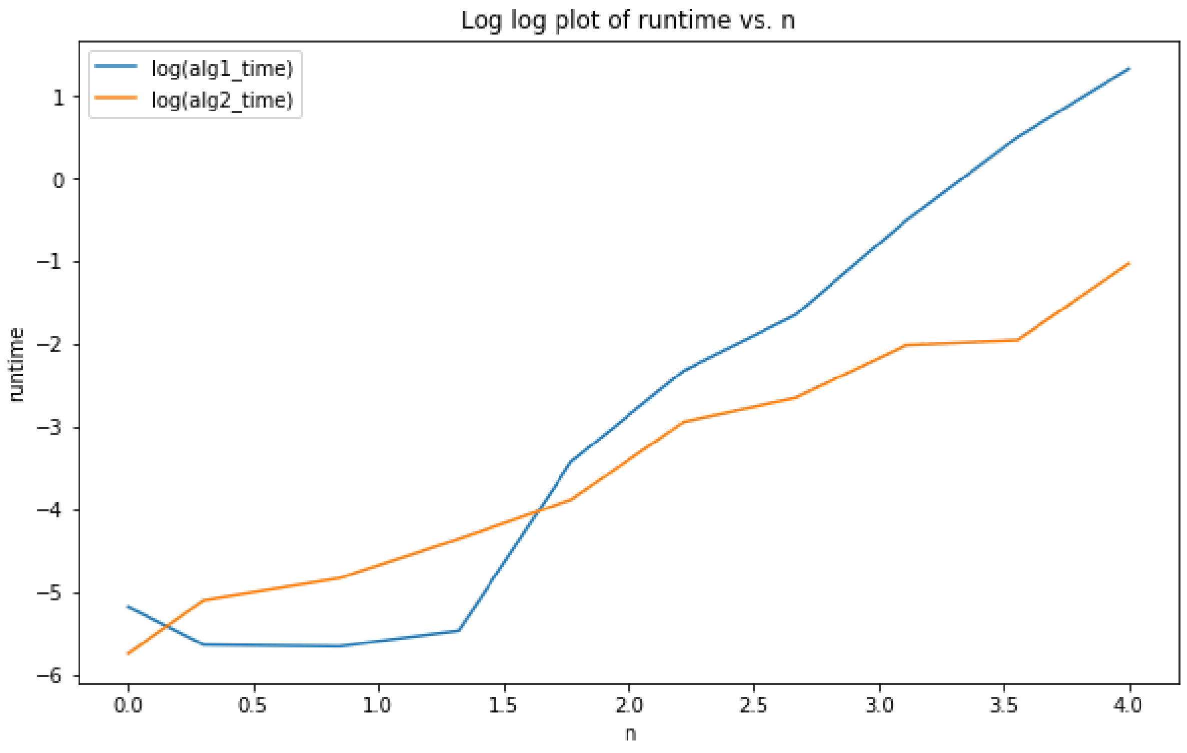
4.3 Time the performance (use time.perf_counter) of alg1 and alg2 for various sizes of data n where the data comes from the function below, plot on a log-log graph as a function of n, and describe the apparent big-O scaling of each. (4 points). Note: Do not include the time spent generating the data 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.)

In [83]:
```python
def data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1):
    state = np.array([x, y, z], dtype=float)
    result = []
    for _ in range(n):
        x, y, z = state
        state += dt * np.array([
            sigma * (y - x),
            x * (rho - z) - y,
            x * y - beta * z
        ])
        result.append(float(state[0] + 30))
    return result
```

In [85]:
```python
# set up all vars
n_sizes2 = [int(n) for n in np.logspace(0, 4, num=10)]
t_a1 = np.zeros(10)
t_a2 = np.zeros(10)

for i,n in enumerate(n_sizes2):
  data = data1(n)
  # record time
  checkpoint1 = perf_counter()
  alg1(data)
  checkpoint2 = perf_counter()
  t_a1[i]=(checkpoint2-checkpoint1)

  #record time
  checkpoint3 = perf_counter()
  alg2(data)
  checkpoint4 = perf_counter()
  t_a2[i] = (checkpoint4-checkpoint3)
```

In [86]:
```
fig7, ax7 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(n_sizes2), np.log10(t_a1), label = "log(alg1_time)")
plt.plot(np.log10(n_sizes2), np.log10(t_a2), label = "log(alg2_time)")
plt.legend()
plt.xlabel('n')
plt.ylabel('runtime')
plt.title('Log log plot of runtime vs. n')
```

Out[86]:
```
Text(0.5, 1.0, 'Log log plot of runtime vs. n')
```



For the first algorithm, the time complexity is O(n^2). This means that when the n is large, we will reach a horizontal line in log log plot with a slope of 2. The above plot shows the trend.

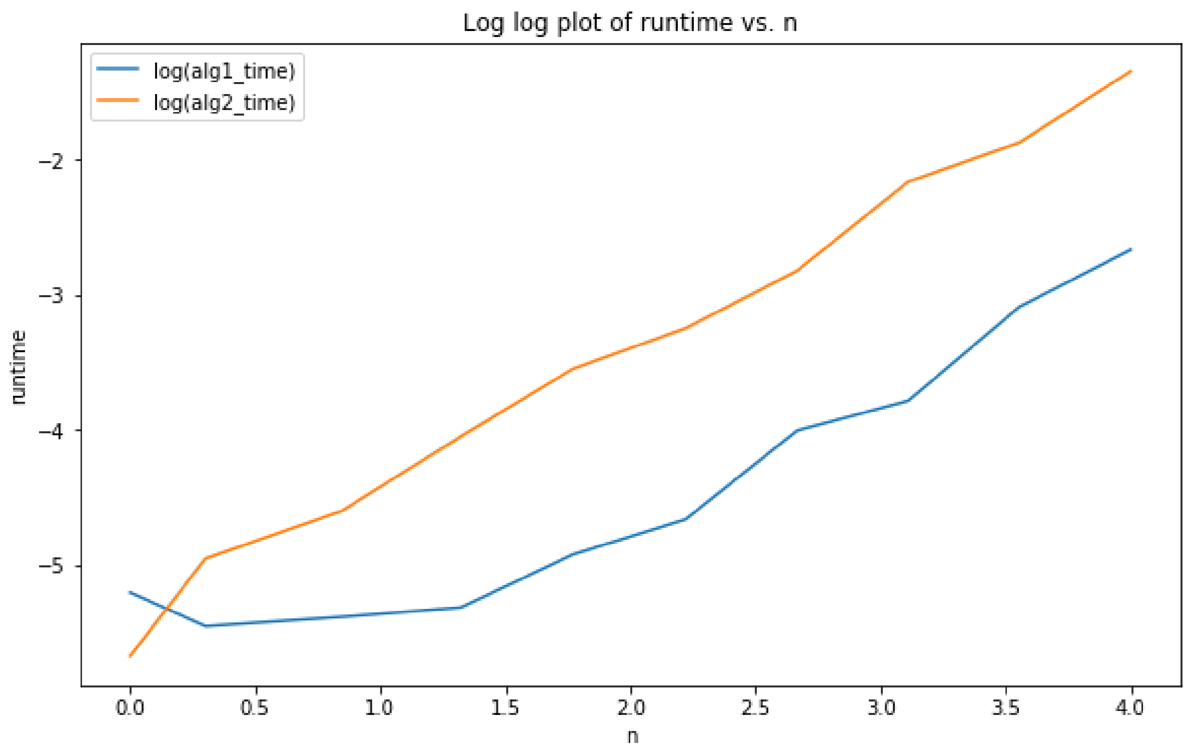
For the second algorithm, the time complexity is O(nlog(n)). When n is large, the log log plot should provide a horizontal line of slope 1. The above plot also shows the trend.

In [87]:
```
# second data function
def data2(n):
    return list(range(n))
```

In [88]:
```python
# set up all vars
t_a1 = np.zeros(10)
t_a2 = np.zeros(10)

for i,n in enumerate(n_sizes2):
  data = data2(n)
  # record time
  checkpoint1 = perf_counter()
  alg1(data)
  checkpoint2 = perf_counter()
  t_a1[i]=(checkpoint2-checkpoint1)

  #record time
  checkpoint3 = perf_counter()
  alg2(data)
  checkpoint4 = perf_counter()
  t_a2[i] = (checkpoint4-checkpoint3)
```

In [89]:
```python
fig8, ax8 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(n_sizes2), np.log10(t_a1), label = "log(alg1_time)")
plt.plot(np.log10(n_sizes2), np.log10(t_a2), label = "log(alg2_time)")
plt.legend()
plt.xlabel('n')
plt.ylabel('runtime')
plt.title('Log log plot of runtime vs. n')
```

Out[89]:
```
Text(0.5, 1.0, 'Log log plot of runtime vs. n')
```



For the first algorithm, given this new dataset provides a sorted list of entries, the time complexity is O(n).

For the second algorithm, given the implemented list is already sorted, the time complexity is also O(n).

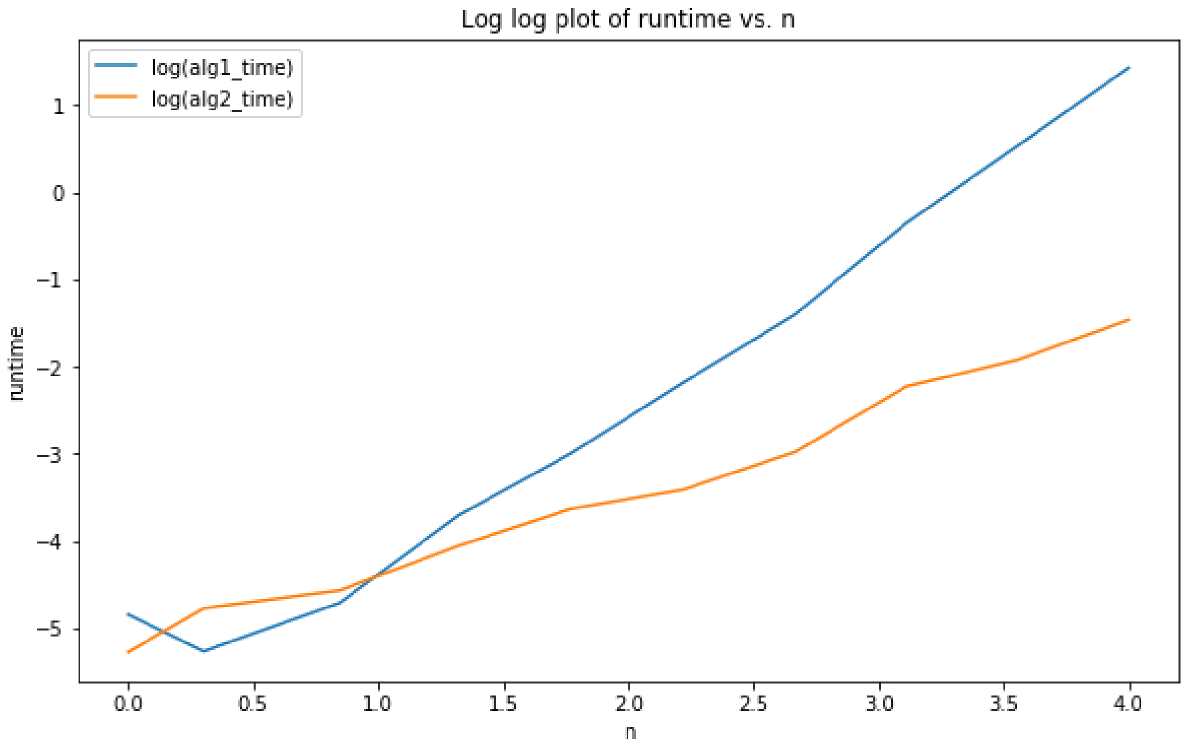
When the n is large enough, the time log-log plot should gives two horizontal lines with same slope. In the above graph we can tell the trends.

In [91]:
```
##third data function
def data3(n):
    return list(range(n, 0, -1))
```

In [92]:
```
# set up all vars
t_a1 = np.zeros(10)
t_a2 = np.zeros(10)

for i,n in enumerate(n_sizes2):
  data = data3(n)
  # record time
  checkpoint1 = perf_counter()
  alg1(data)
  checkpoint2 = perf_counter()
  t_a1[i]=(checkpoint2-checkpoint1)

  #record time
  checkpoint3 = perf_counter()
  alg2(data)
  checkpoint4 = perf_counter()
  t_a2[i] = (checkpoint4-checkpoint3)
```

In [93]:
```
fig9, ax9 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(n_sizes2), np.log10(t_a1), label = "log(alg1_time)")
plt.plot(np.log10(n_sizes2), np.log10(t_a2), label = "log(alg2_time)")
plt.legend()
plt.xlabel('n')
plt.ylabel('runtime')
plt.title('Log log plot of runtime vs. n')
```

Out[93]:
```
Text(0.5, 1.0, 'Log log plot of runtime vs. n')
```



For the first algorithm, the time complexity is O(n^2). Given the new data is sorted in the opposite order, so both algorithm would need to perform swapping continuously. This is the same as using data generate from data1 function.

For the second algorithm, the time complexity is O(nlog(n)), based on similar reasons as for the first algorithm. Because both complexities are the same as using data1 function, the above graph resembles the graph from the first loglog plot generated based on data1 function.

4.4 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)

I think for the second dataset, the scaling would not improve its performance because the dataset is already sorted and its performance is the best. For the first dataset and the thrid dataset, the time would be halfed when used the parallel scaling.

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

In alg2, I will first break the list into two lists of the same size, and then try to split the lists into individual entries; the two individual lists would be running separately through merging and sorting, based on similar procedure described in part 4.3. Then when we have two separately sorted lists, we would try to repeate the same procedure of comparing left and right top element to merge two lists.

This would give us a procedure with time complexity close to: O(nlog(n/2)/2 + n/2) , which would be smaller than O(nlog(n))

4.6 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 implementation is stored in a separate file: exercise4.py

```
In [ ]: '''

        def data1(n, sigma=10, rho=28, beta=8/3, dt=0.01, x=1, y=1, z=1):
            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(data):
          if len(data) <= 1:
            return data
          else:
            split = len(data) // 2
            left = iter(alg2(data[:split]))
            right = iter(alg2(data[split:]))
            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:
                  return result + [right_top] + list(right)
              else:
                result.append(right_top)
                try:
                  right_top = next(right)
                except StopIteration:
                  return result + [left_top] + list(left)

        if __name__ == '__main__':
            runtime = []
            data = []
            i, j = 0, 0
            for n in range(2, 8):
                data = data1(10**n)
                start, end = 0, 10**n - 1
                mid = start + (end - start) // 2
                with multiprocessing.Pool(2) as workers:
                    start = perf_counter()
                    left, right = workers.map(alg2, [data[:mid + 1], data[mid + 1:]])
                    while i < len(left) and j < len(right):
                        if left[i] <= right[j]:
                            data.append(left[i])
                            i += 1
                        else:
                            data.append(right[j])
                            j += 1
                    data += left[i:] + right[j:]
                    end = perf_counter()
                    runtime.append(end - start)
            print(runtime)

        '''
```

By running this py file, I reached an output of:

[0.3487820999871474, 0.37414100000751205, 0.4216407999920193, 0.5100541999854613, 1.840189000009559, 22.92465789997368]

Then I run a normal runtime check on the original unparallel version of data1.

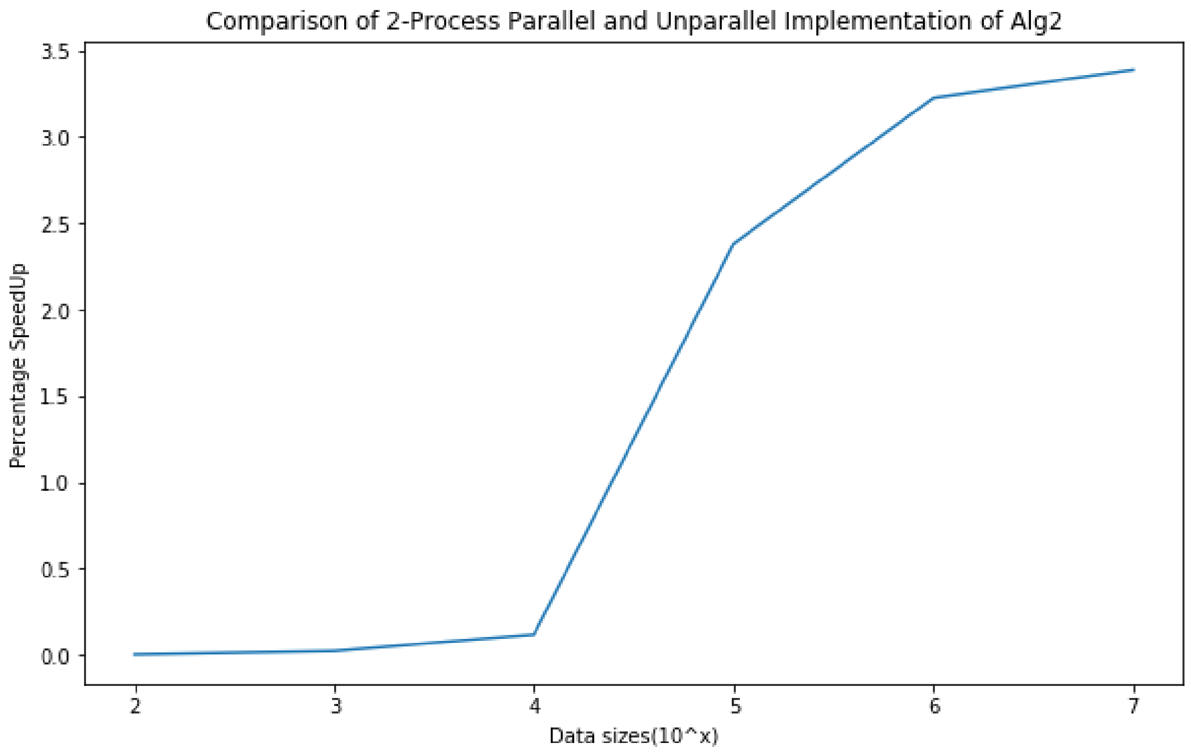
In [108]:
```
data_sizes = [10**2, 10**3, 10**4, 10**5, 10**6, 10**7]
t_a2_un = np.zeros(len(data_sizes))

for i, size in enumerate(data_sizes):
  data = data1(size)
  checkpoint1 = perf_counter()
  alg2(data)
  checkpoint2 = perf_counter()
  t_a2_un[i] = checkpoint2 - checkpoint1
```

In [111]:
```
t_a2_un
```

Out[111]:
```
array([6.11999989e-04, 8.23720000e-03, 4.93230000e-02, 1.21433440e+00,
       5.93389220e+00, 7.76487794e+01])
```

In [114]:
```
#storing the times from the multiprocessed file
t_a2_pa = np.array([0.3487820999871474, 0.37414100000751205, 0.421640799992019
3, 0.5100541999854613, 1.840189000009559, 22.92465789997368])
fig10, ax10 = plt.subplots(figsize=(10, 6))
plt.plot(np.log10(data_sizes), t_a2_un/t_a2_pa)
plt.xlabel('Data sizes(10^x)')
plt.ylabel('Percentage SpeedUp')
plt.title('Comparison of 2-Process Parallel and Unparallel Implementation of A
lg2')
```

Out[114]:
```
Text(0.5, 1.0, 'Comparison of 2-Process Parallel and Unparallel Implementatio
n of Alg2')
```



We can notice that starting from data size of 10^4, the larger the dataset gets, the faster the 2-process procedure gets. When the dataset size hits 10^7, the process hits a highest, 3.5x, boost in runtime.