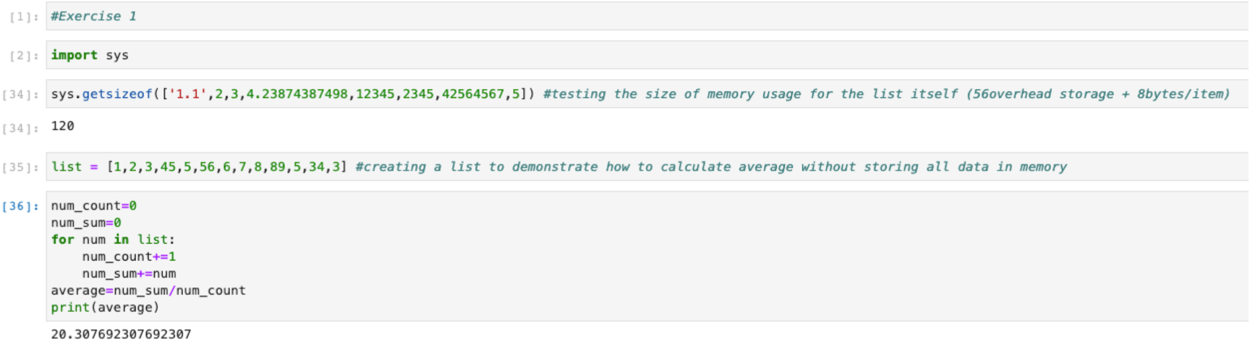
Exercise 1

a) What went wrong was that even though the file was 4GB in txt, when the items in the text file are read and inserted into a list, the storage count will be different. For list's storage, it includes overhead storage for metadata, being 56bytes, and 8bytes for every item added to the list. Therefore, in this case, the weights list will take up storage of (56+8*500,000,000), which is approximately 4,000,000,056 bytes, which can be rounded to (4*10**9/10**9)=4GB; Additionally, for pointers to point to the space in memory for storage, it takes 24bytes/ item with regards to refcount, datatype and real data. Therefore, in total, storing the list will take up (8+24)*500,000,000=16GB storage, which is way greater than the 8GB RAM limit.

b) A way that would work to store the data would be storing it in array: given that array can only store one type of data, it only costs the overhead, 64 bytes, and 8 bytes per value (since they are floats), to store the data—there is no need to occupy memory to specify datatype, refcount and etc. Thus, with the use of array: the storage will be taken up as: 64+8*500,000,000=4000000064bytes < 8GB RAM

c) A way to calculate the average without storing all data in memory is that when forlooping to a new data, summation and number count of the previous data point will be calculated and used to calculate the final average, as shown in Exercise1. Ipynb, and as specified below:

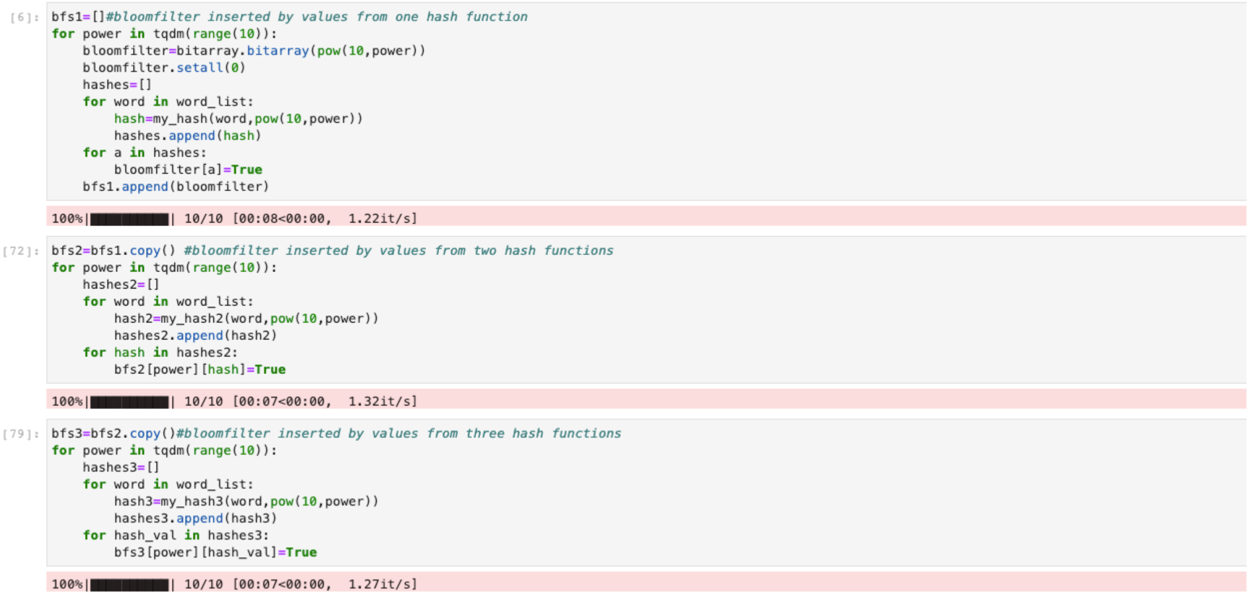


```
[1]: #Exercise 1

[2]: import sys

[34]: sys.getsizeof(['1.1',2,3,4.23874387498,12345,2345,42564567,5]) #testing the size of memory usage for the list itself (56overhead storage + 8bytes/item)

[34]: 120

[35]: list = [1,2,3,45,5,56,6,7,8,89,5,34,3] #creating a list to demonstrate how to calculate average without storing all data in memory

[36]: num_count=0
      num_sum=0
      for num in list:
          num_count+=1
          num_sum+=num
      average=num_sum/num_count
      print(average)

      20.307692307692307
```

Exercise 2: (25 points)

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

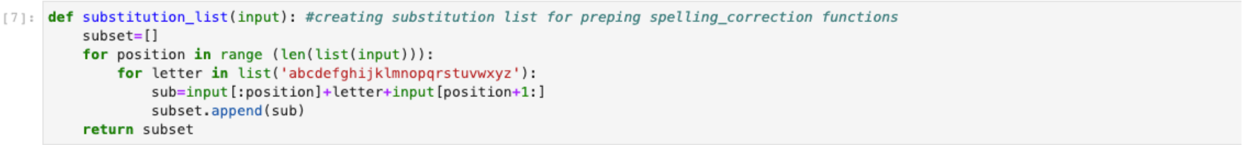
Three lists of lists of bloomfilter using 3 different hash functions are **created** as shown below while the words are **stored** in the respective bloomfilters:



```
[6]: bfs1=[]#bloomfilter inserted by values from one hash function
     for power in tqdm(range(10)):
         bloomfilter=bitarray.bitarray(pow(10,power))
         bloomfilter.setall(0)
         hashes=[]
         for word in word_list:
             hash=my_hash(word,pow(10,power))
             hashes.append(hash)
         for a in hashes:
             bloomfilter[a]=True
         bfs1.append(bloomfilter)

     100%|██████████| 10/10 [00:08<00:00,  1.22it/s]

[72]: bfs2=bfs1.copy() #bloomfilter inserted by values from two hash functions
      for power in tqdm(range(10)):
          hashes2=[]
          for word in word_list:
              hash2=my_hash2(word,pow(10,power))
              hashes2.append(hash2)
          for hash in hashes2:
              bfs2[power][hash]=True

      100%|██████████| 10/10 [00:07<00:00,  1.32it/s]

[79]: bfs3=bfs2.copy()#bloomfilter inserted by values from three hash functions
      for power in tqdm(range(10)):
          hashes3=[]
          for word in word_list:
              hash3=my_hash3(word,pow(10,power))
              hashes3.append(hash3)
          for hash_val in hashes3:
              bfs3[power][hash_val]=True

      100%|██████████| 10/10 [00:07<00:00,  1.27it/s]
```

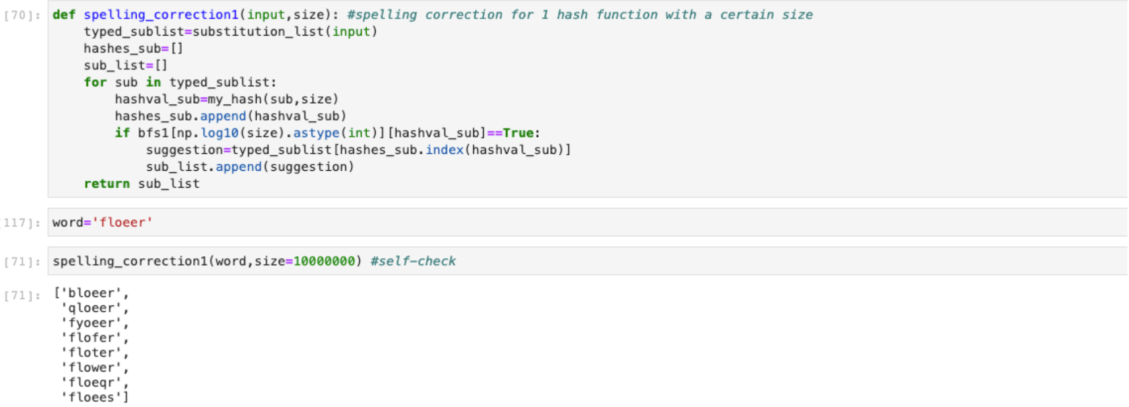
Write a function that suggests spelling corrections using the bloom filter as follows: Try all possible single letter substitutions and see which ones are identified by the filter as possibly words. This algorithm will always find the intended word for this data set, and possibly other words as well.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

1. A function "substitution_list" is generated to output individual list for input words with regards to their single-letter substitutions, as shown:

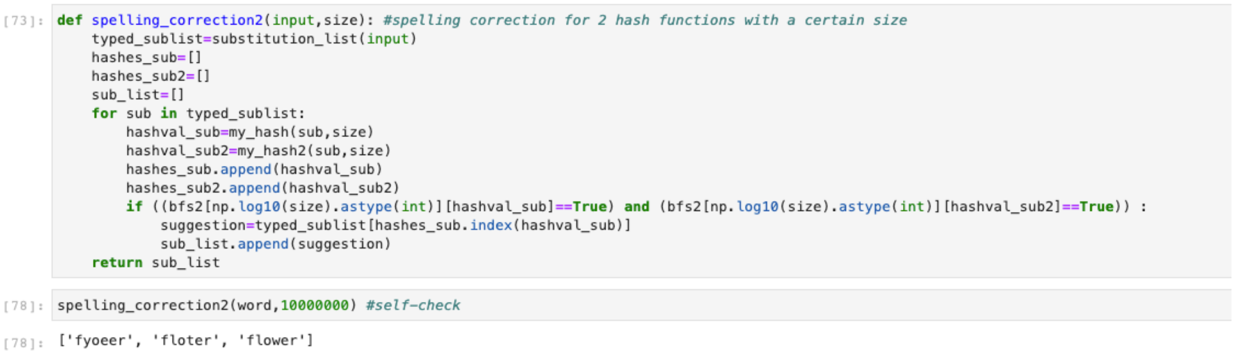


```
[7]: def substitution_list(input): #creating substitution list for preping spelling_correction functions
         subset=[]
         for position in range (len(list(input))):
             for letter in list('abcdefghijklmnopqrstuvwxyz'):
                 sub=input[:position]+letter+input[position+1:]
                 subset.append(sub)
         return subset
```

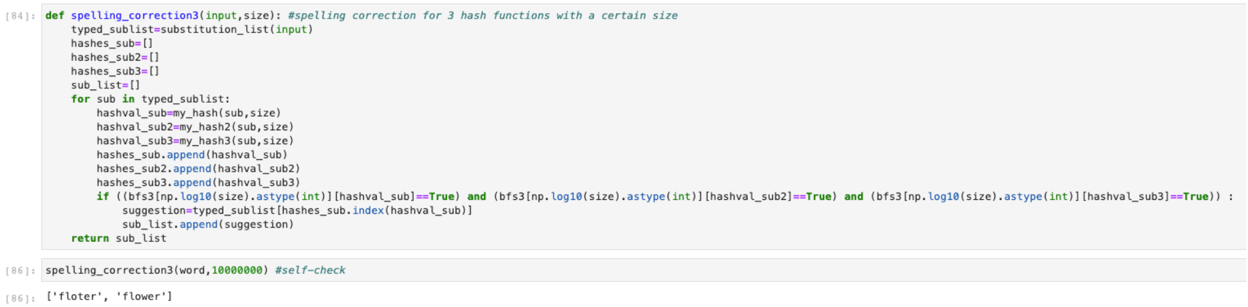
2. Spelling correction functions are created with respects to 3 different hash functions, that is, they will first use the hash function(s) to provide a list of hashed values for the substitution list for each input word, and check if such hashed values matched those in the bloomfilter filled with words from the word list, as shown with tests (while word is identified to be 'floeer':



```
[70]: def spelling_correction1(input,size): #spelling correction for 1 hash function with a certain size
          typed_sublist=substitution_list(input)
          hashes_sub=[]
          sub_list=[]
          for sub in typed_sublist:
              hashval_sub=my_hash(sub,size)
              hashes_sub.append(hashval_sub)
              if bfs1[np.log10(size).astype(int)][hashval_sub]==True:
                  suggestion=typed_sublist[hashes_sub.index(hashval_sub)]
                  sub_list.append(suggestion)
          return sub_list

117]: word='floeer'

[71]: spelling_correction1(word,size=10000000) #self-check

[71]: ['bloeer',
       'qloeer',
       'fyoeer',
       'flofer',
       'floter',
       'flower',
       'floeqr',
       'floees']
```



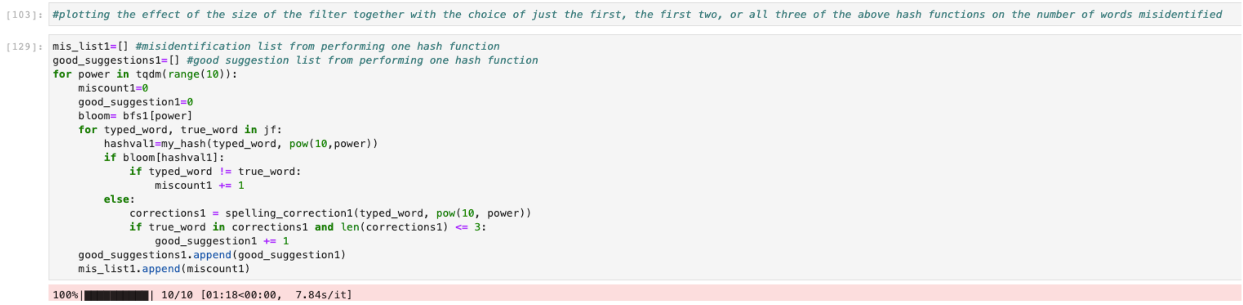
```
[73]: def spelling_correction2(input,size): #spelling correction for 2 hash functions with a certain size
          typed_sublist=substitution_list(input)
          hashes_sub=[]
          hashes_sub2=[]
          sub_list=[]
          for sub in typed_sublist:
              hashval_sub=my_hash(sub,size)
              hashval_sub2=my_hash2(sub,size)
              hashes_sub.append(hashval_sub)
              hashes_sub2.append(hashval_sub2)
              if ((bfs2[np.log10(size).astype(int)][hashval_sub]==True) and (bfs2[np.log10(size).astype(int)][hashval_sub2]==True)) :
                  suggestion=typed_sublist[hashes_sub.index(hashval_sub)]
                  sub_list.append(suggestion)
          return sub_list

[78]: spelling_correction2(word,10000000) #self-check

[78]: ['fyoeer', 'floter', 'flower']
```



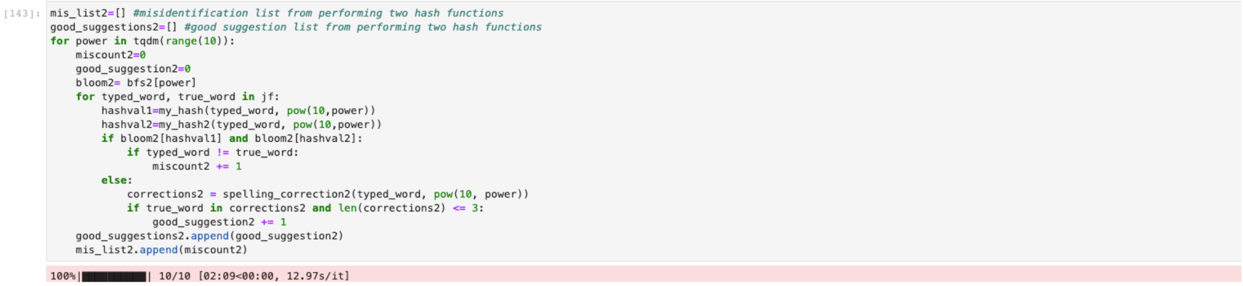
```
[84]: def spelling_correction3(input,size): #spelling correction for 3 hash functions with a certain size
          typed_sublist=substitution_list(input)
          hashes_sub=[]
          hashes_sub2=[]
          hashes_sub3=[]
          sub_list=[]
          for sub in typed_sublist:
              hashval_sub=my_hash(sub,size)
              hashval_sub2=my_hash2(sub,size)
              hashval_sub3=my_hash3(sub,size)
              hashes_sub.append(hashval_sub)
              hashes_sub2.append(hashval_sub2)
              hashes_sub3.append(hashval_sub3)
              if ((bfs3[np.log10(size).astype(int)][hashval_sub]==True) and (bfs3[np.log10(size).astype(int)][hashval_sub2]==True) and (bfs3[np.log10(size).astype(int)][hashval_sub3]==True)) :
                  suggestion=typed_sublist[hashes_sub.index(hashval_sub)]
                  sub_list.append(suggestion)
          return sub_list

[86]: spelling_correction3(word,10000000) #self-check

[86]: ['floter', 'flower']
```

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 typo.json:

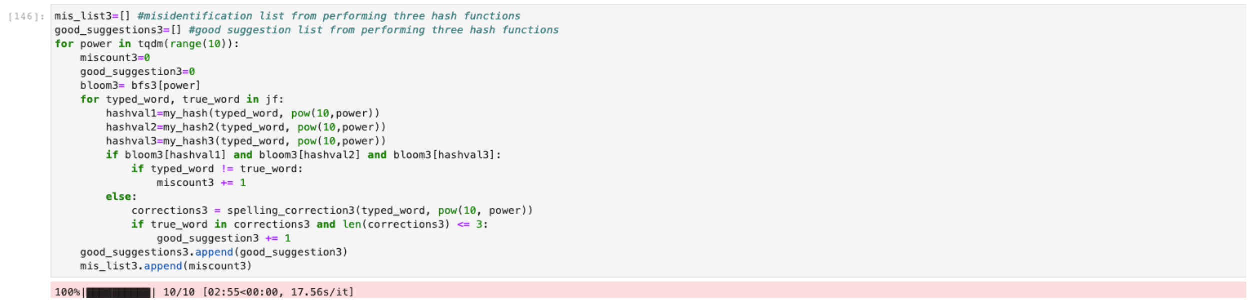
1. Misidentification lists and good_suggestion lists are created for all three hash functions, as shown:



```
[103]: #plotting 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

[129]: mis_list1=[] #misidentification list from performing one hash function
       good_suggestions1=[] #good suggestion list from performing one hash function
       for power in tqdm(range(10)):
           miscount1=0
           good_suggestion1=0
           bloom= bfs1[power]
           for typed_word, true_word in jf:
               hashval1=my_hash(typed_word, pow(10,power))
               if bloom[hashval1]:
                   if typed_word != true_word:
                       miscount1 += 1
               else:
                   corrections1 = spelling_correction1(typed_word, pow(10, power))
                   if true_word in corrections1 and len(corrections1) <= 3:
                       good_suggestion1 += 1
           good_suggestions1.append(good_suggestion1)
           mis_list1.append(miscount1)

       100%|██████████| 10/10 [01:18<00:00,  7.84s/it]
```

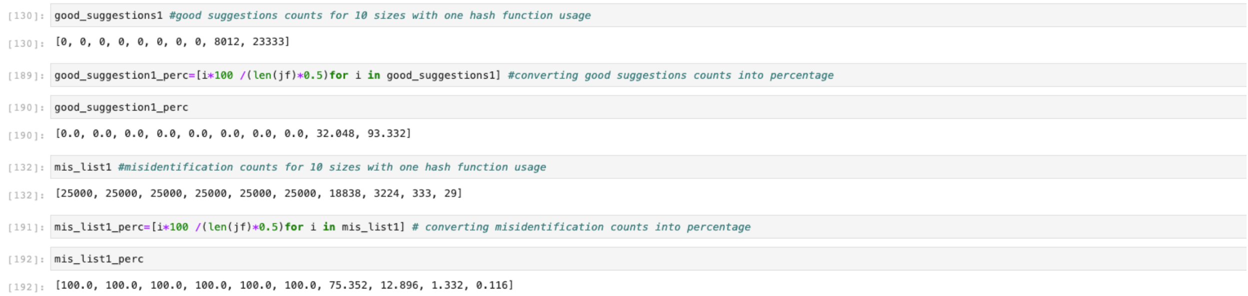


```
[143]: mis_list2=[] #misidentification list from performing two hash functions
       good_suggestions2=[] #good suggestion list from performing two hash functions
       for power in tqdm(range(10)):
           miscount2=0
           good_suggestion2=0
           bloom2= bfs2[power]
           for typed_word, true_word in jf:
               hashval1=my_hash(typed_word, pow(10,power))
               hashval2=my_hash2(typed_word, pow(10,power))
               if bloom2[hashval1] and bloom2[hashval2]:
                   if typed_word != true_word:
                       miscount2 += 1
               else:
                   corrections2 = spelling_correction2(typed_word, pow(10, power))
                   if true_word in corrections2 and len(corrections2) <= 3:
                       good_suggestion2 += 1
           good_suggestions2.append(good_suggestion2)
           mis_list2.append(miscount2)

       100%|██████████| 10/10 [02:09<00:00, 12.97s/it]
```

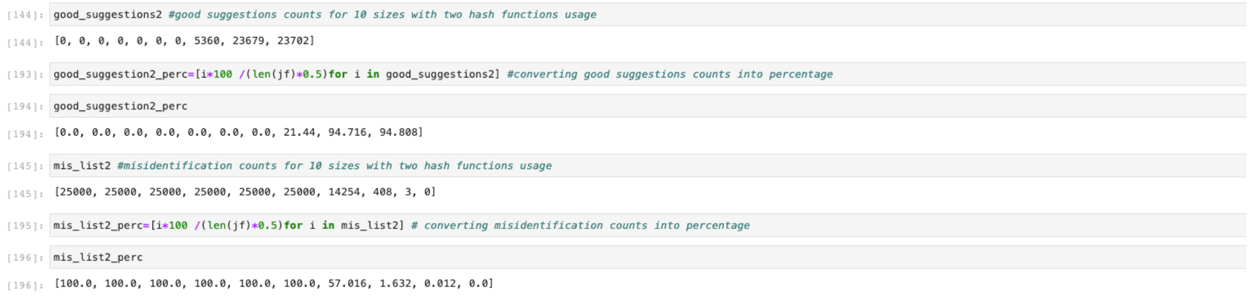


```
[146]: mis_list3=[] #misidentification list from performing three hash functions
       good_suggestions3=[] #good suggestion list from performing three hash functions
       for power in tqdm(range(10)):
           miscount3=0
           good_suggestion3=0
           bloom3= bfs3[power]
           for typed_word, true_word in jf:
               hashval1=my_hash(typed_word, pow(10,power))
               hashval2=my_hash2(typed_word, pow(10,power))
               hashval3=my_hash3(typed_word, pow(10,power))
               if bloom3[hashval1] and bloom3[hashval2] and bloom3[hashval3]:
                   if typed_word != true_word:
                       miscount3 += 1
               else:
                   corrections3 = spelling_correction3(typed_word, pow(10, power))
                   if true_word in corrections3 and len(corrections3) <= 3:
                       good_suggestion3 += 1
           good_suggestions3.append(good_suggestion3)
           mis_list3.append(miscount3)

       100%|██████████| 10/10 [02:55<00:00, 17.56s/it]
```

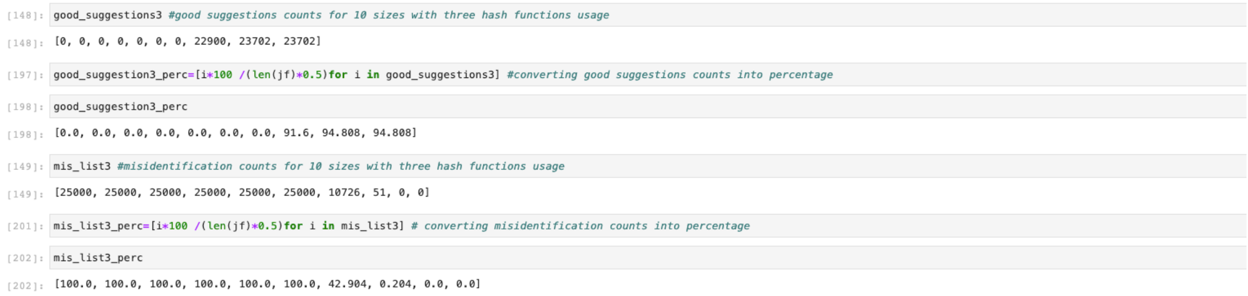
2. The percentage values of the lists are calculated and new lists are created accordingly:



```
[130]: good_suggestions1 #good suggestions counts for 10 sizes with one hash function usage

[130]: [0, 0, 0, 0, 0, 0, 0, 0, 8012, 23333]

[189]: good_suggestion1_perc=[i*100 /(len(jf)*0.5)for i in good_suggestions1] #converting good suggestions counts into percentage

[190]: good_suggestion1_perc

[190]: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 32.048, 93.332]

[132]: mis_list1 #misidentification counts for 10 sizes with one hash function usage

[132]: [25000, 25000, 25000, 25000, 25000, 25000, 18838, 3224, 333, 29]

[191]: mis_list1_perc=[i*100 /(len(jf)*0.5)for i in mis_list1] # converting misidentification counts into percentage

[192]: mis_list1_perc

[192]: [100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 75.352, 12.896, 1.332, 0.116]
```

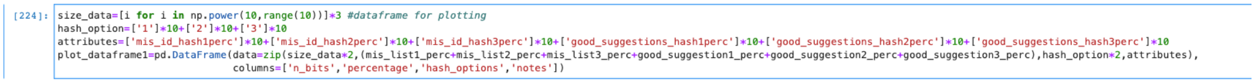


```
[144]: good_suggestions2 #good suggestions counts for 10 sizes with two hash functions usage

[144]: [0, 0, 0, 0, 0, 0, 0, 5360, 23679, 23702]

[193]: good_suggestion2_perc=[i*100 /(len(jf)*0.5)for i in good_suggestions2] #converting good suggestions counts into percentage

[194]: good_suggestion2_perc

[194]: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 21.44, 94.716, 94.808]

[145]: mis_list2 #misidentification counts for 10 sizes with two hash functions usage

[145]: [25000, 25000, 25000, 25000, 25000, 25000, 14254, 408, 3, 0]

[195]: mis_list2_perc=[i*100 /(len(jf)*0.5)for i in mis_list2] # converting misidentification counts into percentage

[196]: mis_list2_perc

[196]: [100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 57.016, 1.632, 0.012, 0.0]
```

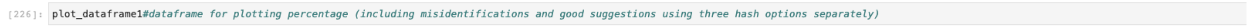


```
[148]: good_suggestions3 #good suggestions counts for 10 sizes with three hash functions usage

[148]: [0, 0, 0, 0, 0, 0, 0, 22900, 23702, 23702]

[197]: good_suggestion3_perc=[i*100 /(len(jf)*0.5)for i in good_suggestions3] #converting good suggestions counts into percentage

[198]: good_suggestion3_perc

[198]: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 91.6, 94.808, 94.808]

[149]: mis_list3 #misidentification counts for 10 sizes with three hash functions usage

[149]: [25000, 25000, 25000, 25000, 25000, 25000, 10726, 51, 0, 0]

[201]: mis_list3_perc=[i*100 /(len(jf)*0.5)for i in mis_list3] # converting misidentification counts into percentage

[202]: mis_list3_perc

[202]: [100.0, 100.0, 100.0, 100.0, 100.0, 100.0, 42.904, 0.204, 0.0, 0.0]
```

3. Dataframe, based on the lists mentioned above is created:

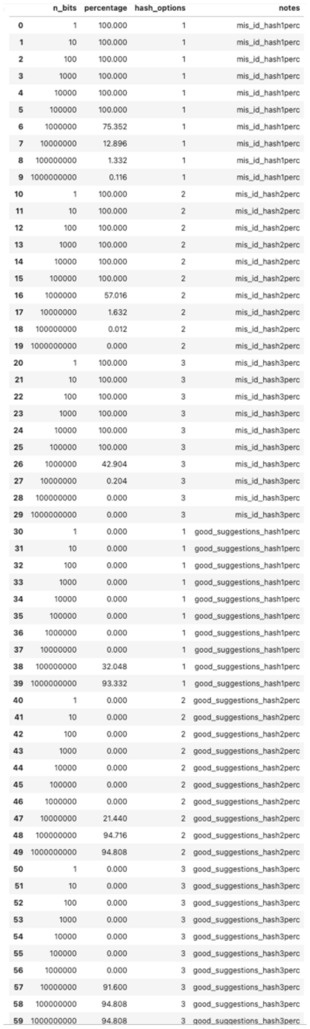


```
[224]: size_data=[i for i in np.power(10,range(10))]*3 #dataframe for plotting
       hash_option=['1']*10+['2']*10+['3']*10
       attributes=['mis_id_hash1perc']*10+['mis_id_hash2perc']*10+['mis_id_hash3perc']*10+['good_suggestions_hash1perc']*10+['good_suggestions_hash2perc']*10+['good_suggestions_hash3perc']*10
       plot_dataframe1=pd.DataFrame(data=zip(size_data*2,(mis_list1_perc+mis_list2_perc+mis_list3_perc+good_suggestion1_perc+good_suggestion2_perc+good_suggestion3_perc),hash_option*2,attributes),
                                   columns=['n_bits','percentage','hash_options','notes'])
```



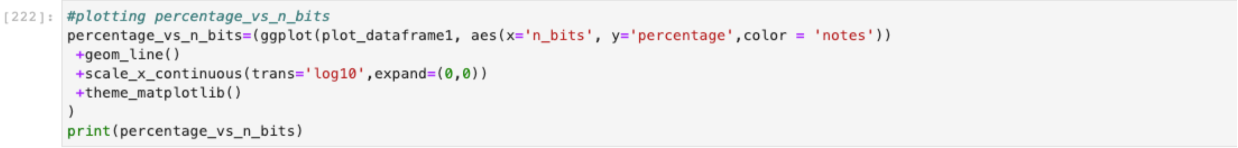
```
[226]: plot_dataframe1#dataframe for plotting percentage (including misidentifications and good suggestions using three hash options separately)
```

With output (values are in the same conlume with annotations due to the convenience of using ggplot color variable:

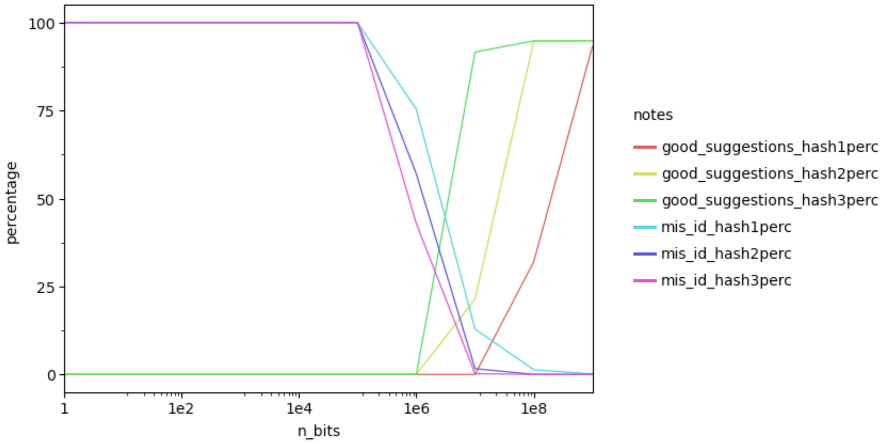


	n_bits	percentage	hash_options	notes
0	1	100.000	1	mis_id_hash1perc
1	10	100.000	1	mis_id_hash1perc
2	100	100.000	1	mis_id_hash1perc
3	1000	100.000	1	mis_id_hash1perc
4	10000	100.000	1	mis_id_hash1perc
5	100000	100.000	1	mis_id_hash1perc
6	1000000	75.352	1	mis_id_hash1perc
7	10000000	12.896	1	mis_id_hash1perc
8	100000000	1.332	1	mis_id_hash1perc
9	1000000000	0.116	1	mis_id_hash1perc
10	1	100.000	2	mis_id_hash2perc
11	10	100.000	2	mis_id_hash2perc
12	100	100.000	2	mis_id_hash2perc
13	1000	100.000	2	mis_id_hash2perc
14	10000	100.000	2	mis_id_hash2perc
15	100000	100.000	2	mis_id_hash2perc
16	1000000	57.016	2	mis_id_hash2perc
17	10000000	1.632	2	mis_id_hash2perc
18	100000000	0.012	2	mis_id_hash2perc
19	1000000000	0.000	2	mis_id_hash2perc
20	1	100.000	3	mis_id_hash3perc
21	10	100.000	3	mis_id_hash3perc
22	100	100.000	3	mis_id_hash3perc
23	1000	100.000	3	mis_id_hash3perc
24	10000	100.000	3	mis_id_hash3perc
25	100000	100.000	3	mis_id_hash3perc
26	1000000	42.904	3	mis_id_hash3perc
27	10000000	0.204	3	mis_id_hash3perc
28	100000000	0.000	3	mis_id_hash3perc
29	1000000000	0.000	3	mis_id_hash3perc
30	1	0.000	1	good_suggestions_hash1perc
31	10	0.000	1	good_suggestions_hash1perc
32	100	0.000	1	good_suggestions_hash1perc
33	1000	0.000	1	good_suggestions_hash1perc
34	10000	0.000	1	good_suggestions_hash1perc
35	100000	0.000	1	good_suggestions_hash1perc
36	1000000	0.000	1	good_suggestions_hash1perc
37	10000000	0.000	1	good_suggestions_hash1perc
38	100000000	32.048	1	good_suggestions_hash1perc
39	1000000000	93.332	1	good_suggestions_hash1perc
40	1	0.000	2	good_suggestions_hash2perc
41	10	0.000	2	good_suggestions_hash2perc
42	100	0.000	2	good_suggestions_hash2perc
43	1000	0.000	2	good_suggestions_hash2perc
44	10000	0.000	2	good_suggestions_hash2perc
45	100000	0.000	2	good_suggestions_hash2perc
46	1000000	0.000	2	good_suggestions_hash2perc
47	10000000	21.440	2	good_suggestions_hash2perc
48	100000000	94.716	2	good_suggestions_hash2perc
49	1000000000	94.808	2	good_suggestions_hash2perc
50	1	0.000	3	good_suggestions_hash3perc
51	10	0.000	3	good_suggestions_hash3perc
52	100	0.000	3	good_suggestions_hash3perc
53	1000	0.000	3	good_suggestions_hash3perc
54	10000	0.000	3	good_suggestions_hash3perc
55	100000	0.000	3	good_suggestions_hash3perc
56	1000000	0.000	3	good_suggestions_hash3perc
57	10000000	91.600	3	good_suggestions_hash3perc
58	100000000	94.808	3	good_suggestions_hash3perc
59	1000000000	94.808	3	good_suggestions_hash3perc

4. Percentages of the misidentifications and good suggestions are plotted as followed:



```
[222]: #plotting percentage_vs_n_bits
       percentage_vs_n_bits=(ggplot(plot_dataframe1, aes(x='n_bits', y='percentage',color = 'notes'))
        +geom_line()
        +scale_x_continuous(trans='log10',expand=(0,0))
        +theme_matplotlib()
       )
       print(percentage_vs_n_bits)
```



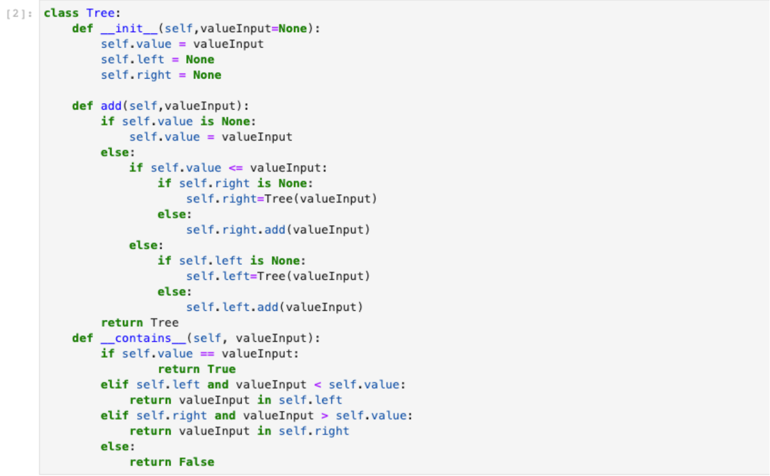
(Annotations for percentage_vs_n_bits:
good_suggestions_hash1perc represents the percentage of good suggestions when using one hash function
good_suggestions_hash2perc represents the percentage of good suggestions when using two hash functions
good_suggestions_hash3perc represents the percentage of good suggestions when using three hash functions
while: mis_id_hash1perc represents the percentage of misidentifications when using one hash function
mis_id_hash2perc represents the percentage of misidentifications when using two hash functions
mis_id_hash3perc represents the percentage of misidentifications when using three hash functions)

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)

Looking at colored lines that represent good suggestions percentage and comparing with table generated for plotting: for one hash function, approximately around 10^9 that good suggestions started to appear 90% of the time, specifically at the percentage of 93.332%; for two hash functions, approximately around 10^8 that good suggestions started to appear 90% of the time, specifically at the percentage of 94.716 % at 10^8 and 94.808% at 10^9; for the use of three hash functions, approximately around 10^7 that good suggestions started to appear 90% of the time, specifically at the percentage of 91.600 % at 10^7, 94.808% at 10^8, and 94.808% at 10^9.

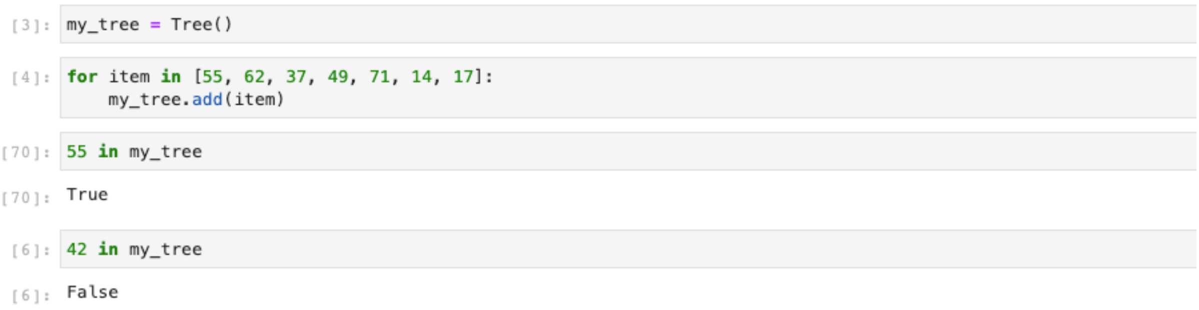
Exercise 3 (25 points)

Based on the Class provided, a function add was added onto it(with the contains function):



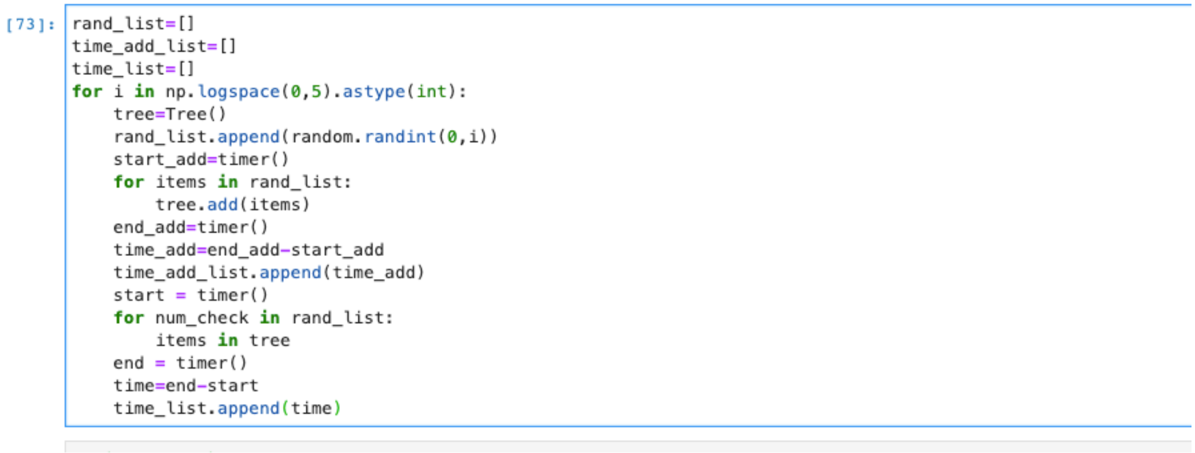
```
[2]: class Tree:
         def __init__(self,valueInput=None):
             self.value = valueInput
             self.left = None
             self.right = None

         def add(self,valueInput):
             if self.value is None:
                 self.value = valueInput
             else:
                 if self.value <= valueInput:
                     if self.right is None:
                         self.right=Tree(valueInput)
                     else:
                         self.right.add(valueInput)
                 else:
                     if self.left is None:
                         self.left=Tree(valueInput)
                     else:
                         self.left.add(valueInput)
             return Tree
         def __contains__(self, valueInput):
             if self.value == valueInput:
                     return True
             elif self.left and valueInput < self.value:
                 return valueInput in self.left
             elif self.right and valueInput > self.value:
                 return valueInput in self.right
             else:
                 return False
```

Check if items are in the tree:



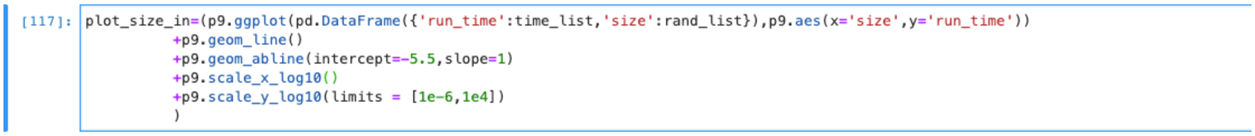
```
[3]: my_tree = Tree()

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

[70]: 55 in my_tree

[70]: True

[6]: 42 in my_tree

[6]: False
```

Random lists within the loop of different sizes, and two lists for running time are created for operation of adding items to the tree **and** check if items are in the tree:

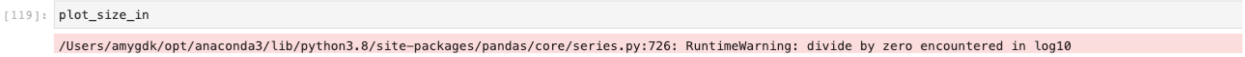


```
[73]: rand_list=[]
      time_add_list=[]
      time_list=[]
      for i in np.logspace(0,5).astype(int):
          tree=Tree()
          rand_list.append(random.randint(0,i))
          start_add=timer()
          for items in rand_list:
              tree.add(items)
          end_add=timer()
          time_add=end_add-start_add
          time_add_list.append(time_add)
          start = timer()
          for num_check in rand_list:
              items in tree
          end = timer()
          time=end-start
          time_list.append(time)
```

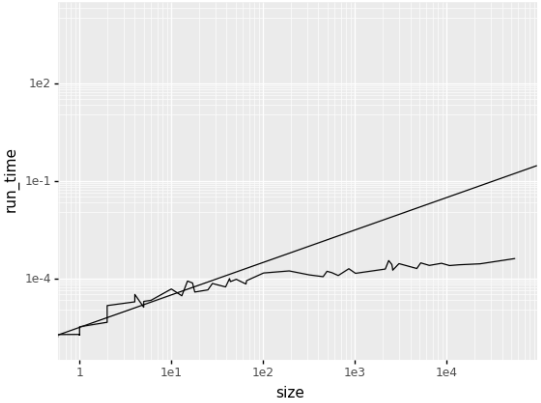
Plotting for contain method running time in different sizes with reference line O(n):



```
[117]: plot_size_in=(p9.ggplot(pd.DataFrame({'run_time':time_list,'size':rand_list}),p9.aes(x='size',y='run_time'))
                +p9.geom_line()
                +p9.geom_abline(intercept=-5.5,slope=1)
                +p9.scale_x_log10()
                +p9.scale_y_log10(limits = [1e-6,1e4])
                )
```

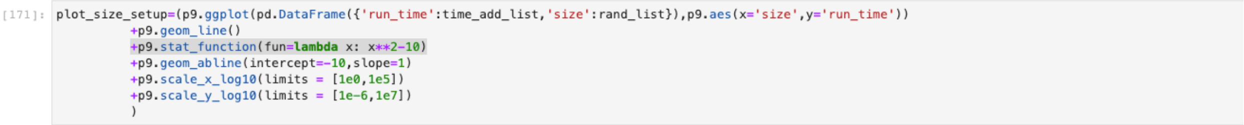


```
[119]: plot_size_in

/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: divide by zero encountered in log10
```



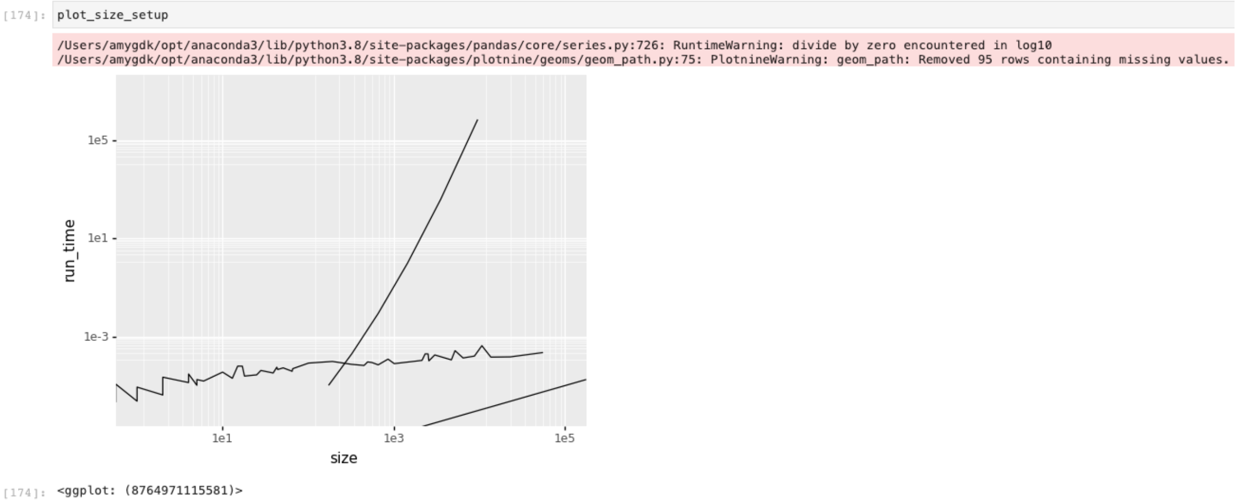
Since the line tends to be horizontal afterwards and under y=x, it can be demonstrated that it follows a O(logn) pattern.

Plotting for add method running time in different sizes with reference lines with O(n) and O(n**2):



```
[171]: plot_size_setup=(p9.ggplot(pd.DataFrame({'run_time':time_add_list,'size':rand_list}),p9.aes(x='size',y='run_time'))
                 +p9.geom_line()
                 +p9.stat_function(fun=lambda x: x**2-10)
                 +p9.geom_abline(intercept=-10,slope=1)
                 +p9.scale_x_log10(limits = [1e0,1e5])
                 +p9.scale_y_log10(limits = [1e-6,1e7])
                 )
```

With output:



```
[174]: plot_size_setup
```

```
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: divide by zero encountered in log10
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 95 rows containing missing values.
```

```
[174]: <ggplot: (8764971115581)>
```

Since the line is in between O(n) and O(n**2), the line follows O(nlogn).

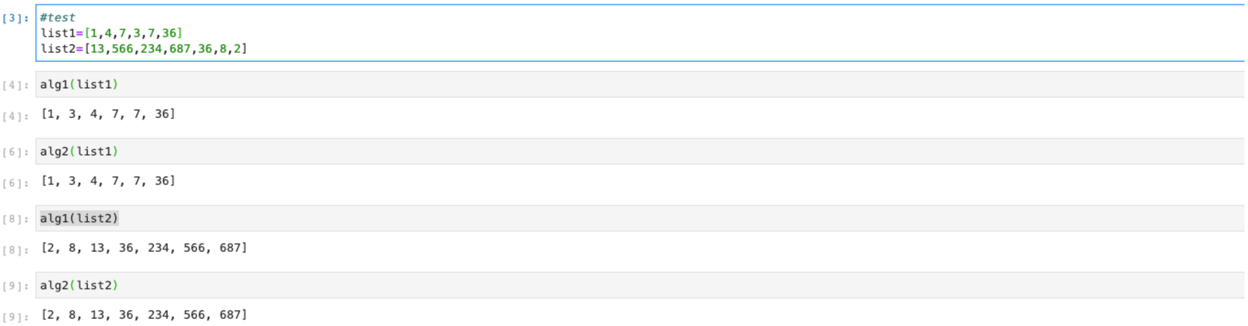
Exercise 4 (35 points)

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)

#test

list1=[1,4,7,3,7,36]

list2=[13,566,234,687,36,8,2]



```
[3]: #test
     list1=[1,4,7,3,7,36]
     list2=[13,566,234,687,36,8,2]

[4]: alg1(list1)

[4]: [1, 3, 4, 7, 7, 36]

[6]: alg2(list1)

[6]: [1, 3, 4, 7, 7, 36]

[8]: alg1(list2)

[8]: [2, 8, 13, 36, 234, 566, 687]

[9]: alg2(list2)

[9]: [2, 8, 13, 36, 234, 566, 687]
```

alg1 and alg2 both sort the values in the lists in ascending orders.

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)

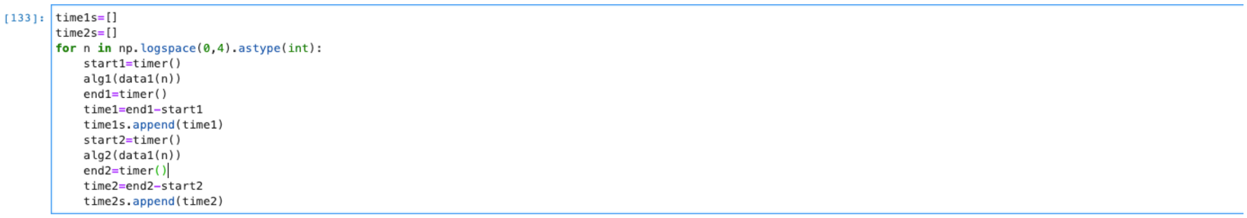
For alg1: first convert inputted values into a list, then set changes to be true. Within the while loop, the changes is set to be false first, and will only turn to be true when position i (looped through the forloop) within the list matches the condition that the value 1 position is smaller than the one after the value at position i, and then the according two values will be inserted into the list-> then changes turns True, goes back up to before the while loop, and while loop is performed again. Therefore, when every datapoint is looped through, the list should be sorted.

For alg2: If the length of list is not sufficient to be sorted (0 or 1), then that list of the datapoint itself will be returned; if it is sufficient, split the list from the middle recursively until the list has one value or is empty, and go to the left of the left of lists and right to the right of lists to take the tops items off. Within the while loop, first if the leftest number is smaller than rightest number, then the leftest number will be inserted in the final list, than the next leftest number will be the new leftest number, however if there isn’t anything on the left(next), then the final list will be outputted with merging the leftest numbers (since the splits are looped, so sorted) combined with the rightest number and the right section of the splits(since the splits are looped, so sorted); however, if the leftest number at the beginning of the loop is not smaller than the rightest number, than the lists will be created from the rightest position of the splits in sorted order, and try every rightest position until there is none, and inserts the these number in a ascending order to the list: the list will then be merged with the leftest number at the beginning and the left section of the splits(since the splits are looped, so sorted).

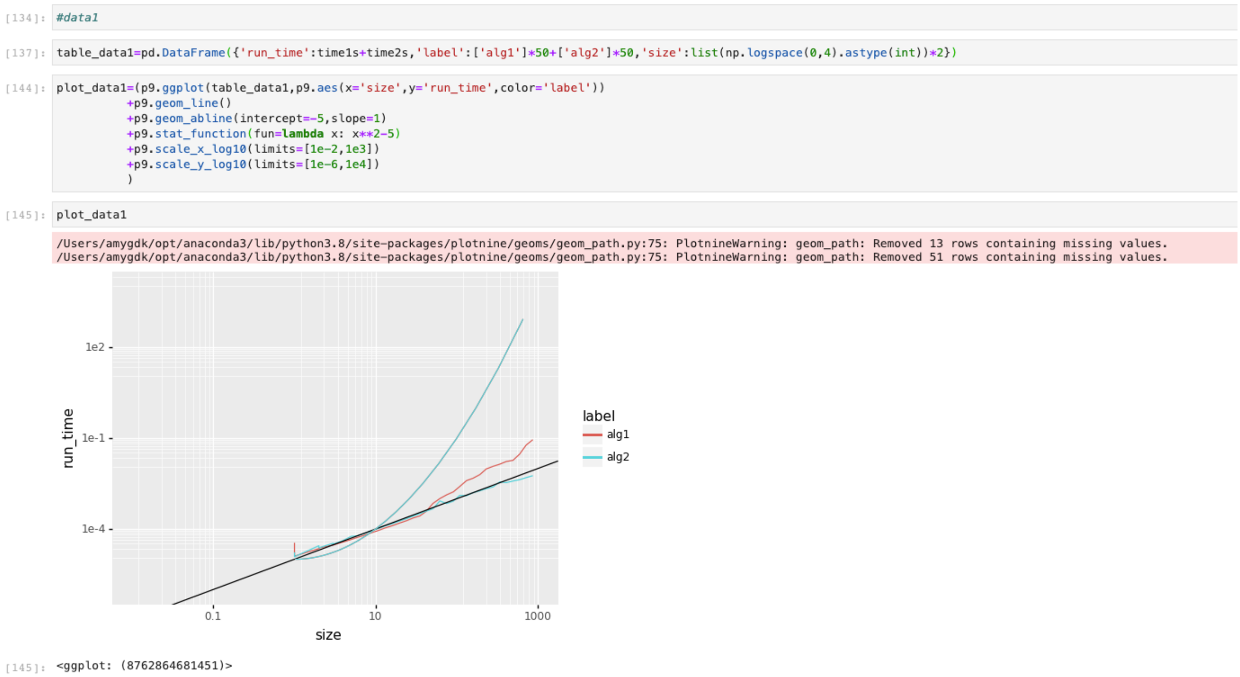
Time the performance (use time.perf_counter) of alg1 and alg2 for various sizes of data n where the data comes from the function below, plot on a log-log graph as a function of n, and describe the apparent big-O scaling of each. (4 points).

Time lists created for add running time and contain running time:

For Data1 function



```
[133]: time1s=[]
       time2s=[]
       for n in np.logspace(0,4).astype(int):
           start1=timer()
           alg1(data1(n))
           end1=timer()
           time1=end1-start1
           time1s.append(time1)
           start2=timer()
           alg2(data1(n))
           end2=timer()
           time2=end2-start2
           time2s.append(time2)
```



```
[134]: #data1
```

```
[137]: table_data1=pd.DataFrame({'run_time':time1s+time2s,'label':['alg1']*50+['alg2']*50,'size':list(np.logspace(0,4).astype(int))*2})
```

```
[144]: plot_data1=(p9.ggplot(table_data1,p9.aes(x='size',y='run_time',color='label'))
                 +p9.geom_line()
                 +p9.geom_abline(intercept=-5,slope=1)
                 +p9.stat_function(fun=lambda x: x**2-5)
                 +p9.scale_x_log10(limits=[1e-2,1e3])
                 +p9.scale_y_log10(limits=[1e-6,1e4])
                 )
```

```
[145]: plot_data1
```

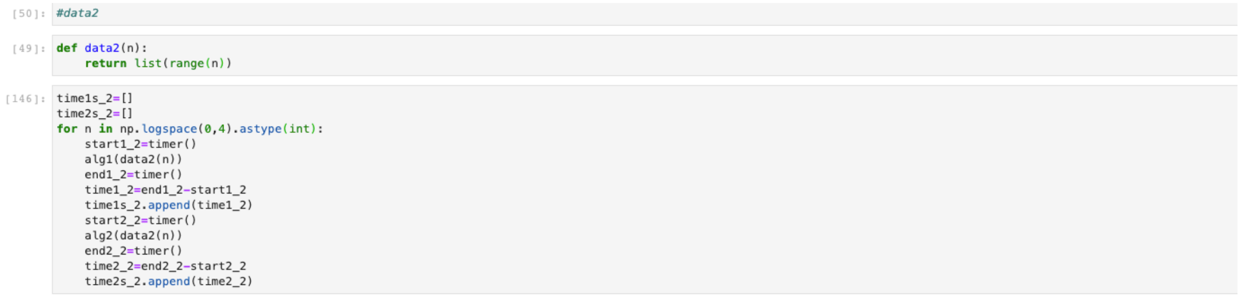
```
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 13 rows containing missing values.
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 51 rows containing missing values.
```

```
[145]: <ggplot: (8762864681451)>
```

Apparent runtime of big O of scaling: alg1: O(nlogn)—since it’s between O(n^2) and O(n); alg2: O(n) (since it overlaps with the reference line y=n)

Scaling performance: for data1 shown above, before the 2 algorithms intersect with y=x^2, they tend to overlap, but afterwards, alg2 has better performance.

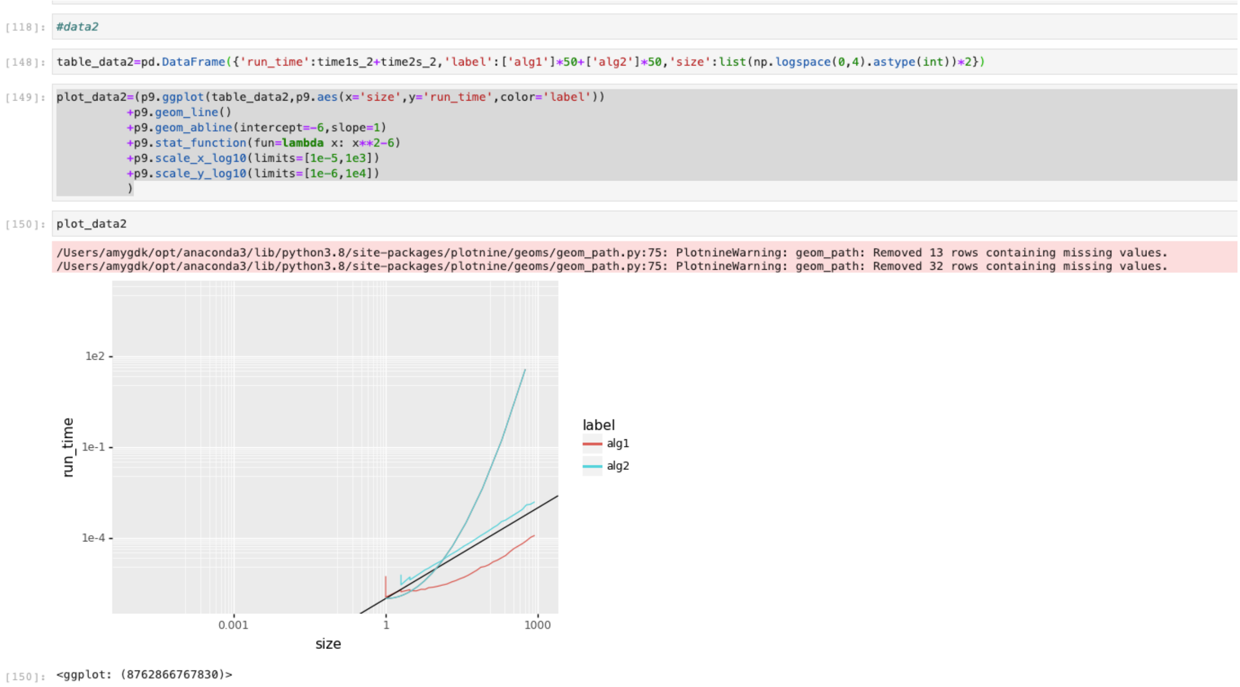
For Data2 function:



```
[50]: #data2
```

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

```
[146]: time1s_2=[]
       time2s_2=[]
       for n in np.logspace(0,4).astype(int):
           start1_2=timer()
           alg1(data2(n))
           end1_2=timer()
           time1_2=end1_2-start1_2
           time1s_2.append(time1_2)
           start2_2=timer()
           alg2(data2(n))
           end2_2=timer()
           time2_2=end2_2-start2_2
           time2s_2.append(time2_2)
```



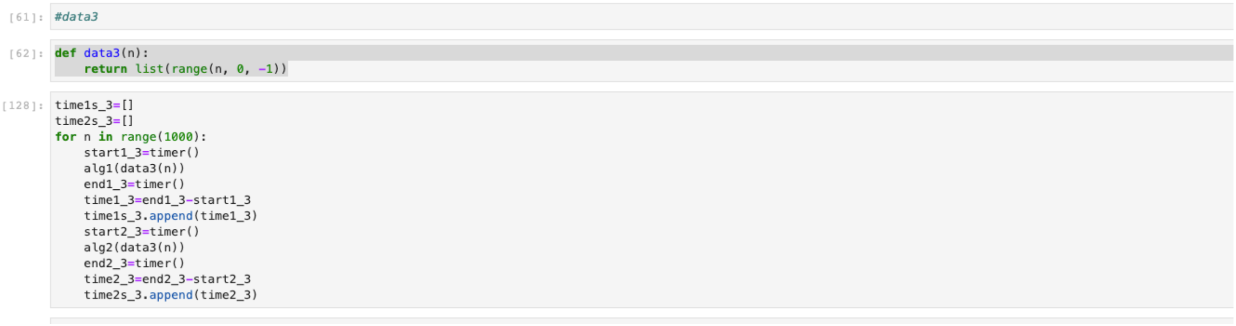
```
[118]: #data2

[148]: table_data2=pd.DataFrame({'run_time':time1s_2+time2s_2,'label':['alg1']*50+['alg2']*50,'size':list(np.logspace(0,4).astype(int))*2})

[149]: plot_data2=(p9.ggplot(table_data2,p9.aes(x='size',y='run_time',color='label'))
                 +p9.geom_line()
                 +p9.geom_abline(intercept=-6,slope=1)
                 +p9.stat_function(fun=lambda x: x**2-6)
                 +p9.scale_x_log10(limits=[1e-5,1e3])
                 +p9.scale_y_log10(limits=[1e-6,1e4])
                 )

[150]: plot_data2

/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 13 rows containing missing values.
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 32 rows containing missing values.
```

```
[150]: <ggplot: (8762866767830)>
```

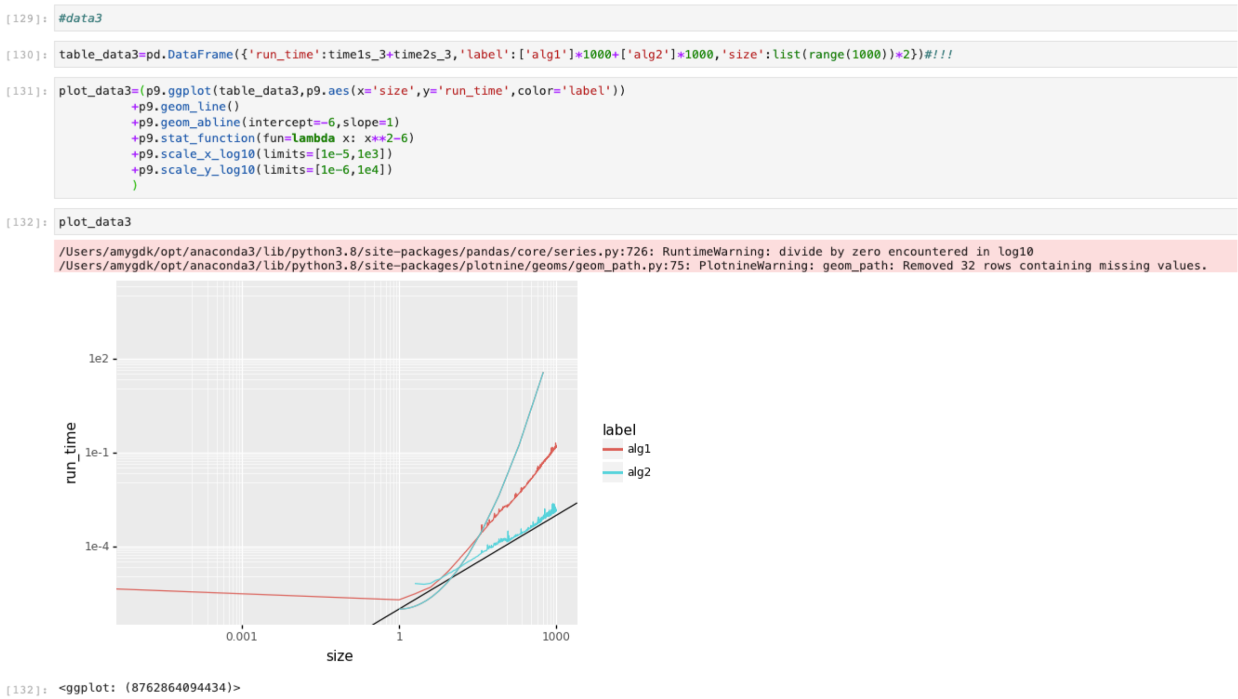
Apparent runtime of big O of scaling: alg1: O(n)—since it’s parallel to O(n); alg2: O(n) for the same reason

Scaling performance: alg1 has better performance since it’s below alg2

For data3 function:



```
[61]: #data3

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

[128]: time1s_3=[]
       time2s_3=[]
       for n in range(1000):
           start1_3=timer()
           alg1(data3(n))
           end1_3=timer()
           time1_3=end1_3-start1_3
           time1s_3.append(time1_3)
           start2_3=timer()
           alg2(data3(n))
           end2_3=timer()
           time2_3=end2_3-start2_3
           time2s_3.append(time2_3)
```



```
[129]: #data3
```

```
[130]: table_data3=pd.DataFrame({'run_time':time1s_3+time2s_3,'label':['alg1']*1000+['alg2']*1000,'size':list(range(1000))*2})#!!!
```

```
[131]: plot_data3=(p9.ggplot(table_data3,p9.aes(x='size',y='run_time',color='label'))
                  +p9.geom_line()
                  +p9.geom_abline(intercept=-6,slope=1)
                  +p9.stat_function(fun=lambda x: x**2-6)
                  +p9.scale_x_log10(limits=[1e-5,1e3])
                  +p9.scale_y_log10(limits=[1e-6,1e4])
                  )
```

```
[132]: plot_data3
```

```
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/pandas/core/series.py:726: RuntimeWarning: divide by zero encountered in log10
/Users/amygdk/opt/anaconda3/lib/python3.8/site-packages/plotnine/geoms/geom_path.py:75: PlotnineWarning: geom_path: Removed 32 rows containing missing values.
```

```
[132]: <ggplot: (8762864094434)>
```

Apparent runtime of big O of scaling: alg1: O(nlogn)— since it's between O(n^2) and O(n); alg2: O(n) (since it overlaps with the reference line y=n)

Scaling performance: before the 2 algorithms intersect with y=x^2, they tend to overlap, but afterwards, alg2 has better performance since it's below alg1.

In all, I will recommend to use alg2 since it has better performance in data1 and data 3 function, as in data 2 function, it particularly focuses on sorted values, which would not be the best function to test better performance and effectiveness.

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

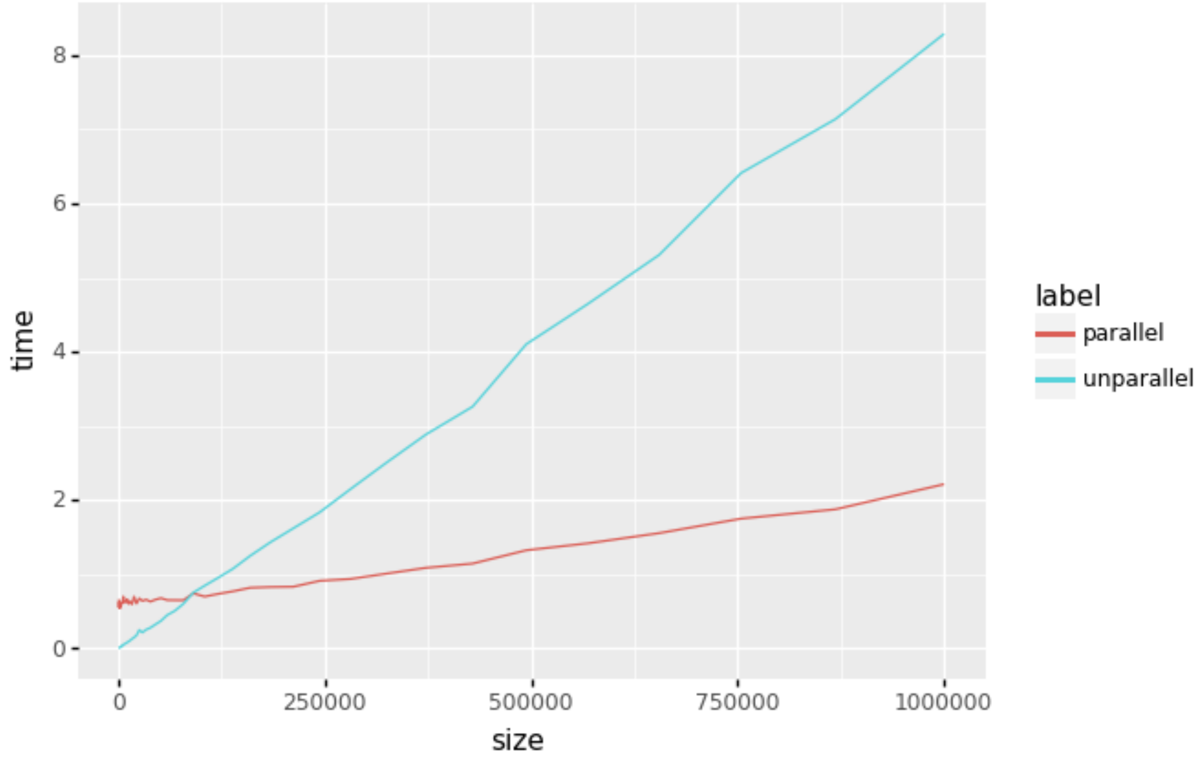
The independent tasks are left and right subsets, that is the multiprocessing tool can process the work that left and right subsets sort simultaneously.

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).

```
import multiprocessing
import numpy as np
import time

from scipy.stats import nakagami

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)

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

if __name__ == "__main__":
    ns = np.logspace(3,6, dtype = int)
    times = []
    time2s=[]
    for n in ns:
        data = data1(n)
        left = data[:len(data)//2]
        right = data[len(data)//2:]
        start_time = time.perf_counter()
        with multiprocessing.Pool(2) as worker:
            results = worker.map(alg2,[left,right])
        stop_time = time.perf_counter()
        times.append(stop_time-start_time)
        start2=time.perf_counter()
        alg2(data1(n))
        end2=time.perf_counter()
        time2=end2-start2
        time2s.append(time2)
```

```
    import plotnine as p9
    import pandas as pd
    plotdata =
pd.DataFrame({'size':list(ns)*2,'time':times+time2s,'label':['parallel']*50+['unparall
el']*50})
    plot=p9.ggplot(plotdata,p9.aes(x='size',y='time',color='label'))+p9.geom_line()
    p9.ggsave(plot,'runtime_amy.png')
```

comparison between performance of data 1 in parallel and unparallel time list:



Discussion of the difference in run time between parallel and unparallel tools usage: the parallel task has speeded up algorithm 2 by approximately 2x: for example, when size = 250000, the runtime for unparallel is approximately 2s while the parallel time is approximately 1s.