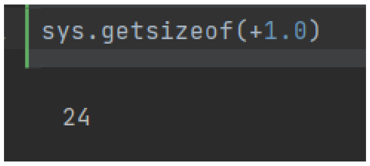
Assignment 2

Instructions: All python libraries used are listed on the top of each Jupyter notebook file.

Answers:

Exercise 1:

1. My friend used **list**[] to store data and turned the data type in the .txt file to **float** data type in Python**. The container list takes up 56 bytes plus 8 bytes per item, and item storage should be counted separately**. I test in the .txt file that each digit, decimal point and sign takes up 1 byte, and the line break symbol takes up 2 bytes. For example, +1.0\n in the .txt file takes up 6 bytes. But +1.0 in python takes up 24 bytes. Therefore, if the .txt file contains *n* data taking up 4GB, assume each data takes up 16 bytes, then we have $16 * n \approx 4GB$. The list containing these *n* float data will be $56 + n * 8 + n * 24 = 56 + 32 * n$ bytes, which is more than 8GB, leading to Out of RAM's 8GB Memory.



```
sys.getsizeof(+1.0)

24
```

Also, the append() operation will generate a new list[] larger than the original list, which also takes up more memory space. In details, if the list a=[1,2,3,4] is full, a.append(5) operation will request a new list with 2 times space of the original list, which means it creates a new list with 8 spaces, [1,2,3,4,5, , ,]. So if the weight list size is large, the append() operation will create a new list that is much larger, leading to Out of Memory.

2. To store all the data in the memory, we can turn the data type of float64 to float32 or float16 using numpy. Also, instead of using append() in the for-loop, we can generate the list by weights=[np.float32(line) for line in f].

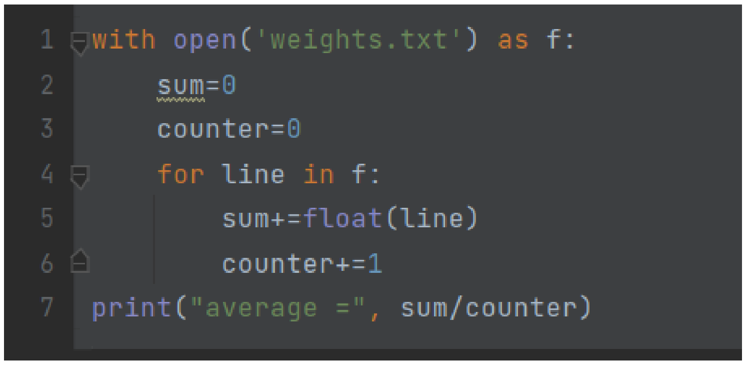
What's more, we can store the data in an array rather than a list to save memory.

3. We don't need to store all the data and calculate the mean in one time. We can process the data in batches or one by one. For example, this dataset of N samples can be divided to $N = 100 * n_1 + n_2$, where n_1=N//100, n_2 = N%100.We can store 100 data

at a time, calculate and store their mean in another list: meanlist=[], and repeat doing so. Then the mean can be calculated as

$$mean = \frac{100 * sum(meanlist[0{:}n_1])}{N} + \frac{n_2 * meanlist[-1]}{N}$$

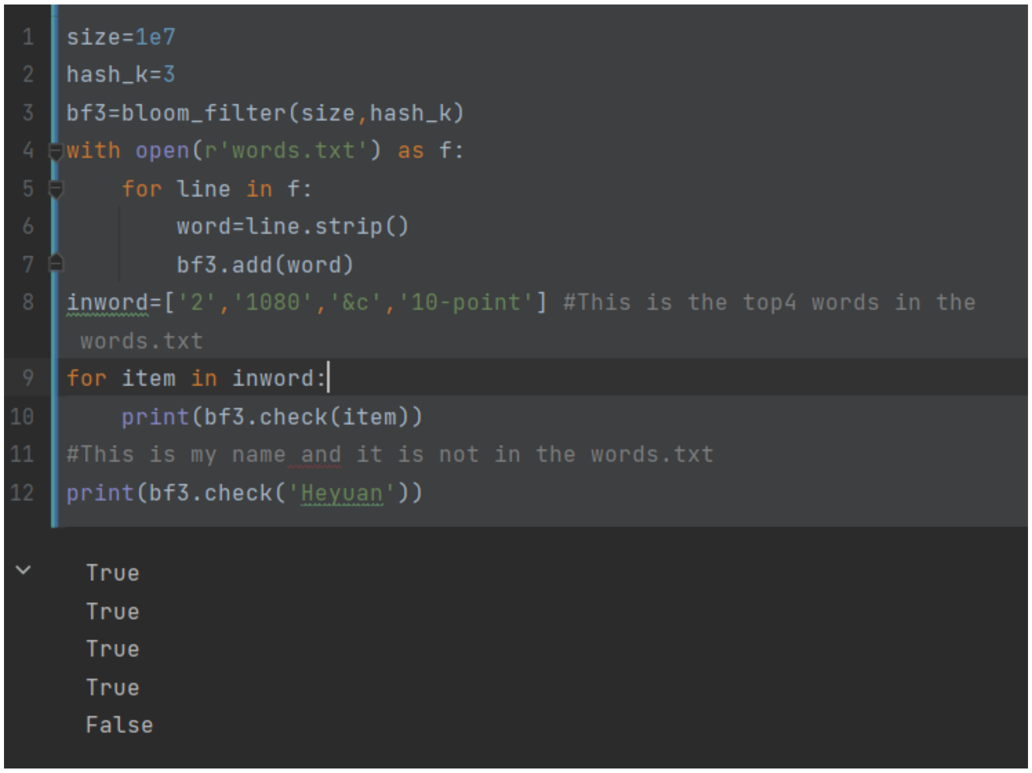
To process data one by one, we can put the calculation method in the reading file loop by setting 2 variables, sum and counter, as follows.



```python
with open('weights.txt') as f:
    sum=0
    counter=0
    for line in f:
        sum+=float(line)
        counter+=1
print("average =", sum/counter)
```

Exercise 2:

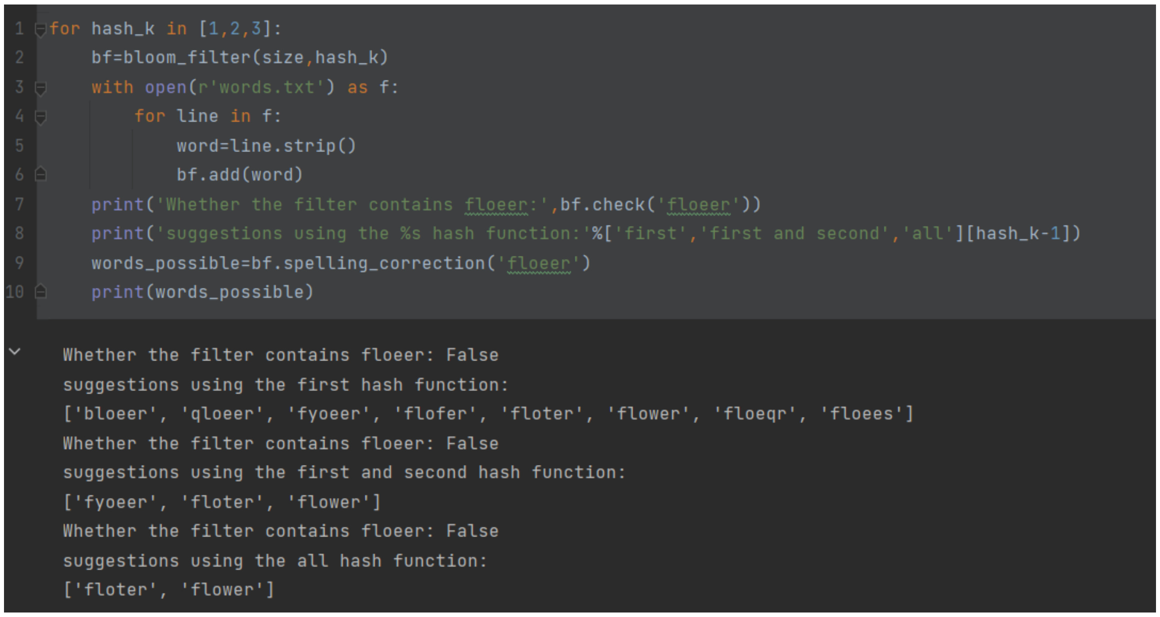
1. A Bloom Filter class is implemented using bitarray for initialization, with the three given hash functions. There are 2 main methods in the class: Add() is to insert words in the filter. Check() is to show whether the word exists in the filter. Words in the words.txt are stored in the bloom filter. The bloom filter **bf3** is tested as below. We need to input the bloom filter size (here we use size=1e7) and hash functions (here we use hash_k=3 to use all three functions) for initialization.



```python
size=1e7
hash_k=3
bf3=bloom_filter(size,hash_k)
with open(r'words.txt') as f:
    for line in f:
        word=line.strip()
        bf3.add(word)
inword=['2','1080','&c','10-point'] #This is the top4 words in the
 words.txt
for item in inword:
    print(bf3.check(item))
#This is my name and it is not in the words.txt
print(bf3.check('Heyuan'))
```

```
True
True
True
True
False
```

2. A function spelling_correction(self, word) is defined to give suggestions on possible words in the given dataset. Its code is in the appendix. The self-check test is the same with the given answer.

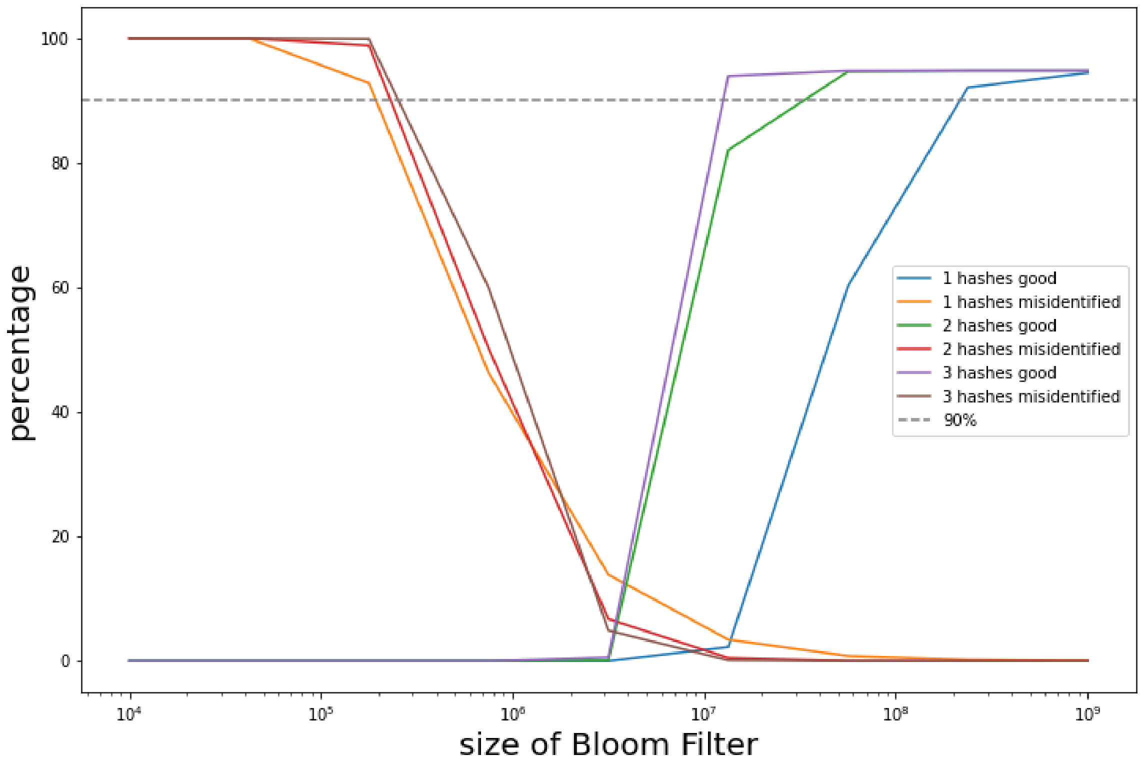


```python
for hash_k in [1,2,3]:
    bf=bloom_filter(size,hash_k)
    with open(r'words.txt') as f:
        for line in f:
            word=line.strip()
            bf.add(word)
    print('Whether the filter contains floeer:',bf.check('floeer'))
    print('suggestions using the %s hash function:'%['first','first and second','all'][hash_k-1])
    words_possible=bf.spelling_correction('floeer')
    print(words_possible)
```

```
Whether the filter contains floeer: False
suggestions using the first hash function:
['bloeer', 'qloeer', 'fyoeer', 'flofer', 'floter', 'flower', 'floeqr', 'floees']
Whether the filter contains floeer: False
suggestions using the first and second hash function:
['fyoeer', 'floter', 'flower']
Whether the filter contains floeer: False
suggestions using the all hash function:
['floter', 'flower']
```

3. The code to calculate the good_rate of making good_suggestions and mis_rate of misidentification is in the jupyter code cell in the appendix.

The results are plotted in the figure, with size n ranges equally from logspace(10^4, 10^9, 9).



4. The highest good_rate achieved is 94.808% with 3 hash functions and size n larger than 10^8.
   The experiment's results are listed below, with size n ranges equally from logspace(10^4, 10^9, 9). n= [1e4, 4.21e4, 1.77e5, 7.49e5, 3.16e6, 1.33e7, 5.62e7, 2.37e8, 1e9].

   When hash_num=3, the good_rate is [0.0, 0.0, 0.0, 0.0, 0.588, 93.948, 94.796, 94.808, 94.808].

   When hash_num=2, the good_rate is [0.0, 0.0, 0.0, 0.0, 0.076, 82.06, 94.676, 94.796, 94.808].

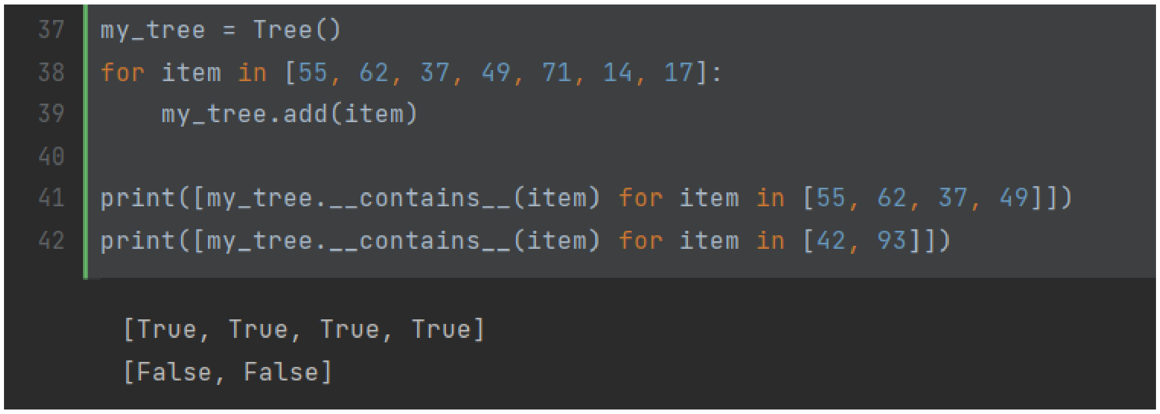
   When hash_num=1, the good_rate is [0.0, 0.0, 0.0, 0.0, 0.0, 2.252, 60.224, 92.104, 94.472].

   Using the first hash function, it needs approximately 2.37*e8 bits to give good suggestions 90% of the time.
   Using the first and second hash functions, it needs approximately 5.62*e7 bits to give good suggestions 90% of the time.

Using the three hash functions, it needs approximately 1.33*e7 bits to give good suggestions 90% of the time.

Exercise 3:

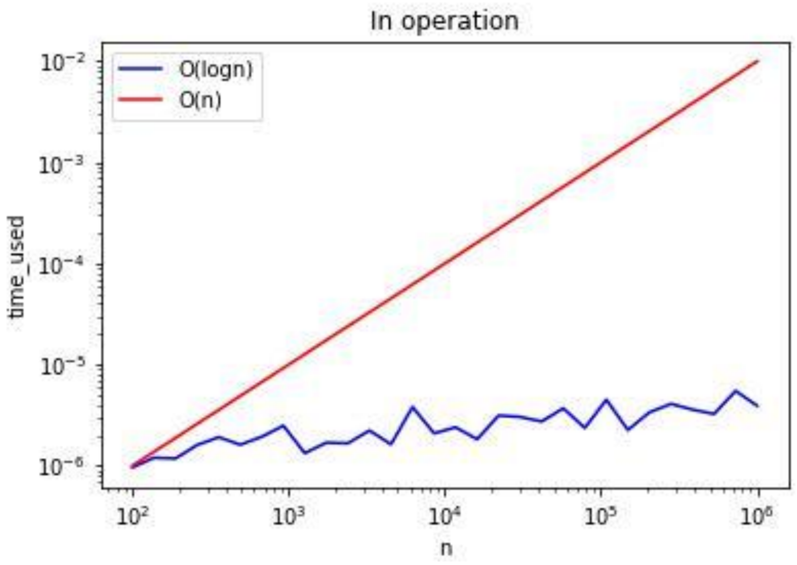
1. The Tree class is extended into a binary search tree. An **add** method is defined and the __contains__ method is tested as below.



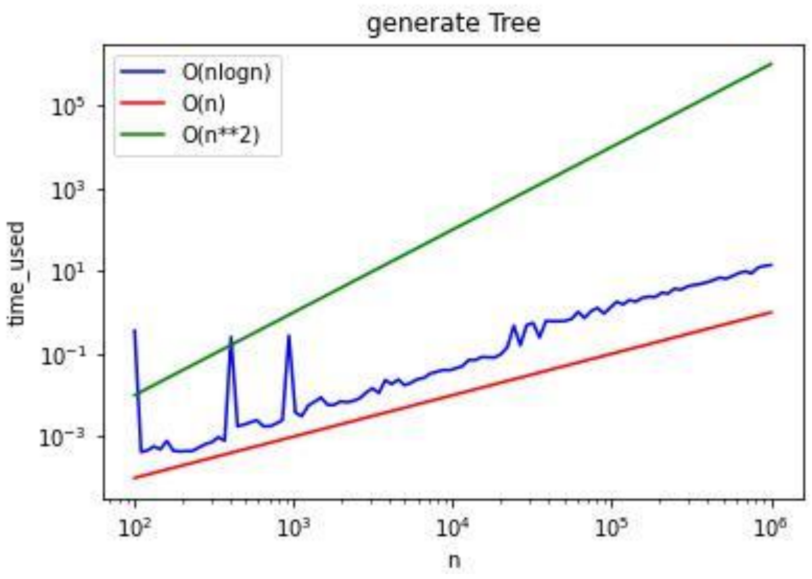
```
my_tree = Tree()
for item in [55, 62, 37, 49, 71, 14, 17]:
    my_tree.add(item)

print([my_tree.__contains__(item) for item in [55, 62, 37, 49]])
print([my_tree.__contains__(item) for item in [42, 93]])
```

```
[True, True, True, True]
[False, False]
```

2. Various sizes, *n*, of trees are used for timing the **in** operation. The log-log plot is shown below. *N* is generated by np.logspace(2,6,30), so it ranges from 100 to 10^6. I fixed the *findsample* list with 1000 randomly selected integer samples and calculated the mean time of **in** execution. The blue line shows that **in** is executing in O(log n) times.

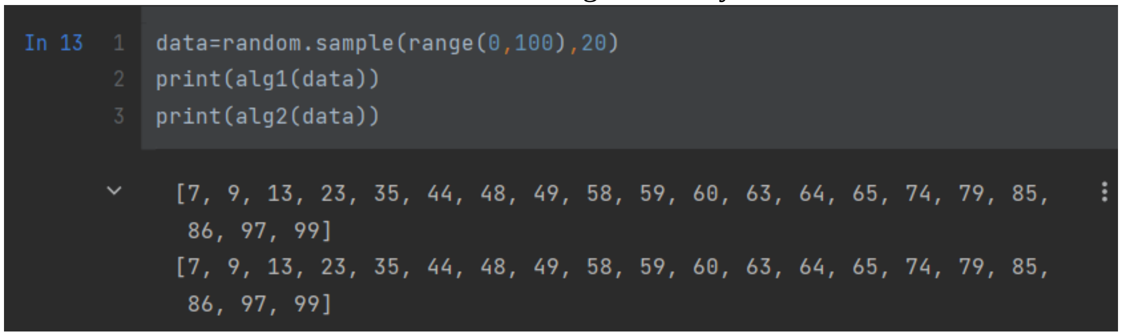


3. The time to setup the tree is O(n logn), shown by the blue line in the log-log plot below. *N* is generated by np.logspace(2,6,100), so it ranges from 100 to 10^6. The green line is O(n^2) and the red line is O(n).



Exercise 4:

1. These functions sort the data in ascending order. My tests are as follows:



```
In 13
data=random.sample(range(0,100),20)
print(alg1(data))
print(alg2(data))
```

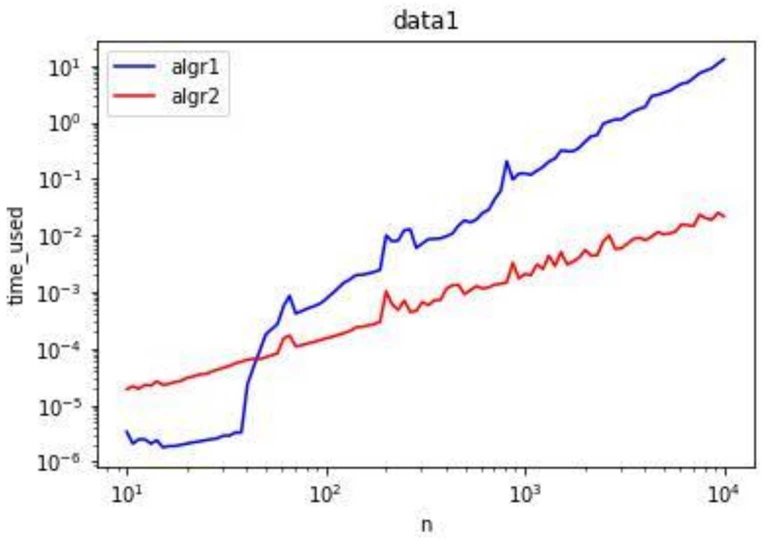
```
[7, 9, 13, 23, 35, 44, 48, 49, 58, 59, 60, 63, 64, 65, 74, 79, 85,
 86, 97, 99]
[7, 9, 13, 23, 35, 44, 48, 49, 58, 59, 60, 63, 64, 65, 74, 79, 85,
 86, 97, 99]
```

   First, I generate a list of 20 data randomly, ranging from 0 to 99. Then I apply these two functions to the data and print the results.

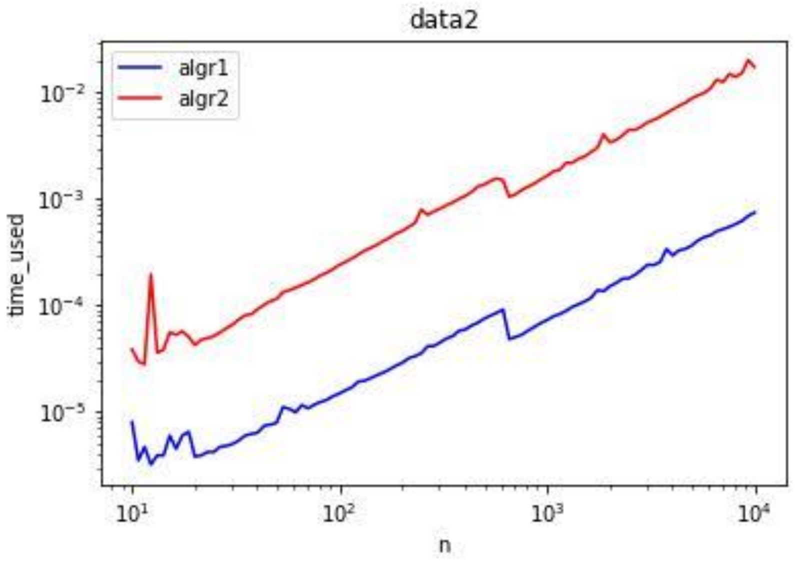
2. For algorithm1 (alg1), it traverses the list to find whether there is a value, data[i], bigger than the next value, data[i+1]. If this condition happens, it swaps the positions of the two values. It keeps finding and swapping values until there is no value bigger than its next value (by using the flag ‘changes’).

   For algorithm2 (alg2), it keeps splitting the list in half until all sub-lists contain only one element. Then it compares 2 elements and merge them into one list with the smaller one on the left and the bigger one on the right. Repeat this step so all single elements are sorted and merged to a larger list, i.e., 2-element list. Merge 2-element lists by comparing their top elements, appending the smaller value to the merged result list and comparing the next element in the smaller value’s list. After comparing all elements, a 4-element list in ascending order is merged. Repeat this kind of merging step and all sub-lists will be merged to the original size list in ascending order.

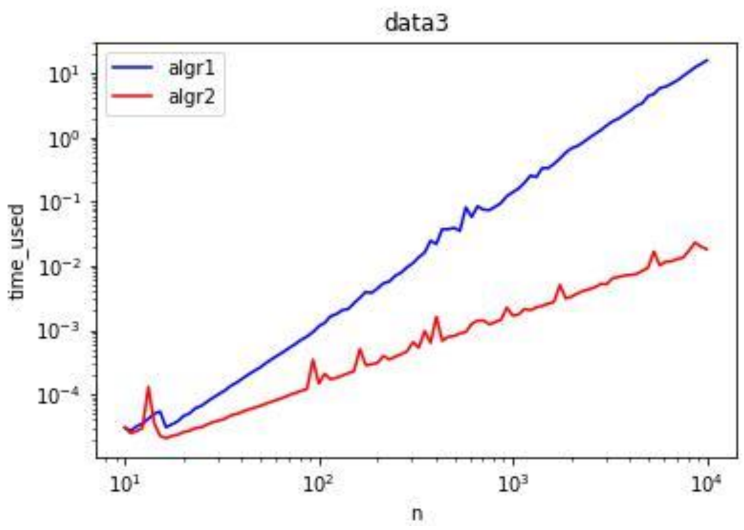
3. The log-log graph of data1(n) is plotted. The big-O scaling of alg1 is O(n^2) and alg2 is O(n logn).



4. The log-log graph of data2(n) is plotted. The big-O scaling of alg1 is O(n) and alg2 is O(n logn).



5. The log-log graph of data3(n) is plotted. The big-O scaling of alg1 is O(n^2) and alg2 is O(n logn).



6. Data2 is originally in ascending order and Data3 is in descending order. Data 1 is a mixture of them, i.e., fluctuation.

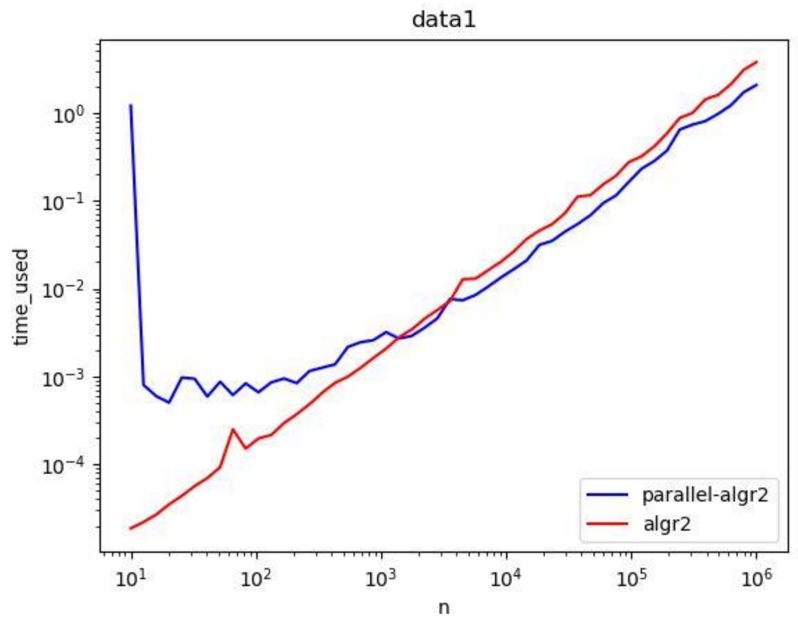
   For algorithm1, it performs best on data2 with the scaling O(n) and worst on data3 with the scaling O(n^2). It performs better on data1 than algorithm2 when the size of *n* is mall. But when *n* becomes larger, it performs worse than algorithm2.

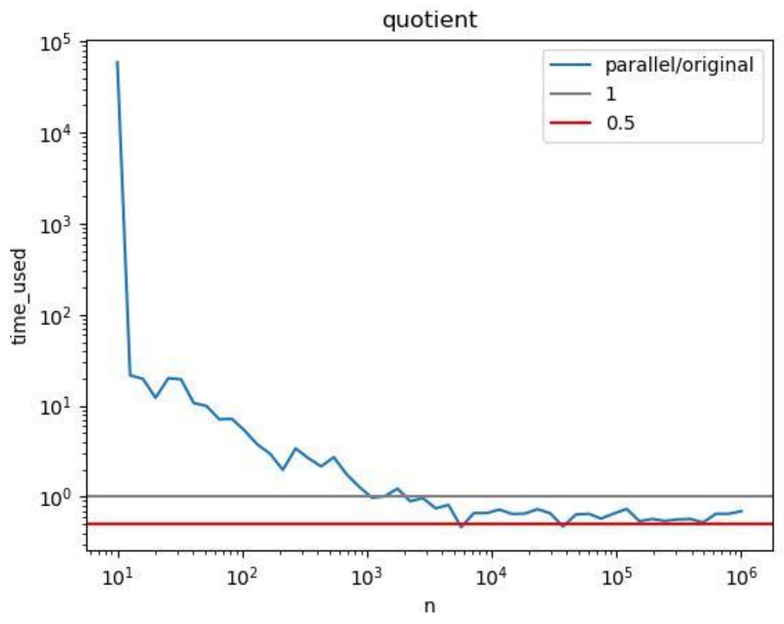
   For algorithm2, it performs the same on the three datasets with the scaling O(n logn).

7. For arbitrary data, I would recommend algorithm2 because it has more stable performance under normal conditions. Random data lists fluctuate so algorithm1 couldn't always reach its best performance to surpass algorithm2.

8. Algorithm2's every forward 'split, compare and merge' step within each sub-list in the same hierarchy is independent from each other. Therefore, they can be parallelized and their sorted results can be combined for further merging.

9. The two-process parallel of algorithm2 is dividing data from the data1 function to 2 equal parts, conducting sorting them using algorithm2 by 2 threads simultaneously and combine their results by one merging step.
   The multiprocessing module's Pool is used for two-process parallel implementation on data generated by data1(n), where *n* ranges from 10 to 10^6. The time used by algorithm2 and parallelized algorithm2 is plotted in the first figure and their quotient is plotted in the second figure.





The gray line in the quotient figure is equal to 1 and the red line is equal to 0.5. We can see that when *n* is bigger than 10^3, the parallelized algorithm is faster than the original algorithm2. When *n* is smaller than 10^3, the time taken to parallelize the algorithm is not worthwhile because launching the multiprocessing.Pool also takes some time.

Choose a moderate n=10^6, the parallelized algorithm uses 2.25283 seconds and the original algorithm2 uses 3.43008 seconds. So we get a 1.52x speedup.

Appendix

Code for Exercise2:

```
from bitarray import bitarray
from hashlib import sha3_256,sha256, blake2b
from string import ascii_letters,ascii_lowercase
import matplotlib.pyplot as plt
import numpy as np
import json

class bloom_filter:
    '''
    initialization: input bloom filter size and the number of hash functions.
    add():insert words in the filter
    check():determine whether the word is in the filter
    '''
    def __init__(self,size,hash_k=3) -> None:
        self.size=int(size)
        self.hash_k=int(hash_k)
        self.__data=bitarray(self.size)
        self.__hash_list=[self.__my_hash1,self.__my_hash2,self.__my_hash3][:self.hash_k]

        self.__data.setall(0)

    def __my_hash1(self,s):
        return int( sha256(s. lower() .encode()). hexdigest(), 16) % self.size
    def __my_hash2(self,s):
        return int( blake2b(s. lower() .encode()). hexdigest(), 16) % self.size
    def __my_hash3(self,s):
        return int( sha3_256(s. lower() .encode()). hexdigest(), 16) % self.size

    def add(self,word):
        p=[hash_func(word) for hash_func in self.__hash_list]
        for i in p:
            self.__data[i]=1

    def check(self,word):
        p=[hash_func(word) for hash_func in self.__hash_list]
        in_bf=0
        for i in p:
            if self.__data[i]==1:
                in_bf+=1
        if in_bf==self.hash_k:
            return True
        else:
            return False

    def show(self):
        print(self.__data)

    def spelling_suggest(self,word):
        '''return word suggestions by letter substitution'''
        words_possible=[]
        for letter in range(len(word)):
            front=word[:letter]
            later=word[letter+1:]
            words_substite=[f'{front}{i}{later}' for i in
ascii_lowercase.replace(word[letter],'')]
            for w in words_substite:
                if self.check(w):
                    words_possible.append(w)
        return words_possible

    def spelling_correction(self,word):
        suggest_words=self.spelling_suggest(word)
        if self.check(word):
            suggest_words=[word]+suggest_words
        return suggest_words
```

```
size=1e7
hash_k=3
bf3=bloom_filter(size,hash_k)
with open(r'words.txt') as f:
    for line in f:
        word=line.strip()
        bf3.add(word)
inword=['2','1080','&c','10-point'] #This is the top4 words in the words.txt
for item in inword:
    print(bf3.check(item))
#This is my name and it is not in the words.txt
print(bf3.check('Heyuan'))
```

```
for hash_k in [1,2,3]:
    bf=bloom_filter(size,hash_k)
    with open(r'words.txt') as f:
        for line in f:
            word=line.strip()
            bf.add(word)
    print('Whether the filter contains floeer:',bf.check('floeer'))
    print('suggestions using the %s hash function:'%['first','first and second','all'][hash_k-
1])
    words_possible=bf.spelling_correction('floeer')
    print(words_possible)
```

```
'''plot good_rate and mis_rate in one plot'''
with open(r'typos.json') as f:
    data=json.loads(f.read())

N=np.logspace(4,9,9)
plt.figure(figsize=(12,8))
plt.axes(xscale='log')
plt.xlabel('size of Bloom Filter',fontsize=20)
plt.ylabel('percentage',fontsize=20)
good_rate_N_k=[]
mis_rate_N_k=[]
for hash_k in [1,2,3]:
    good_rate_N=[]
    mis_rate_N=[]
    for size in N:
        bf=bloom_filter(size,hash_k)
        with open(r'words.txt') as f:
            for line in f:
                word=line.strip()
                bf.add(word)
        good_suggestion=0
        misidentified=0
        for pair in data:
            if bf.check(pair[0]) and pair[0]!=pair[1]:
                misidentified+=1
            suggest_words=bf.spelling_suggest(pair[0])
            if pair[1] in suggest_words:
                if len(suggest_words)<4:
                    good_suggestion+=1
        good_rate=good_suggestion/len(data)*2
        mis_rate=misidentified/len(data)*2
        good_rate_N.append(good_rate*100)
        mis_rate_N.append(mis_rate*100)
    plt.plot(N, good_rate_N, label='%d hashes good'%hash_k)
    plt.plot(N, mis_rate_N,label='%d hashes misidentified'%hash_k)
    good_rate_N_k.append(good_rate_N)
    mis_rate_N_k.append(mis_rate_N)

plt.axhline(y=90,color='gray',linestyle='dashed',label='90%')
plt.legend()
plt.savefig('good_mis_rate.jpg')
plt.show()
print(good_rate_N_k)
print(mis_rate_N_k)
print(N)
```

Code for Exercise3:

```
import random
import time
import matplotlib.pyplot as plt
import numpy as np

class Tree: #This is a TreeNode initialization acutally
    def __init__(self,value=None):
        self._value = value
        self.left = None
        self.right = None

    def add(self, data):
        if self._value!=None:
            if data < self._value:
                if self.left is None:
                    self.left = Tree(data)
                else:
                    self.left.add(data)
            elif data > self._value:
                if self.right is None:
                    self.right = Tree(data)
                else:
                    self.right.add(data)
        else:
            self._value = data

    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

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

print([my_tree.__contains__(item) for item in [55, 62, 37, 49]])
print([my_tree.__contains__(item) for item in [42, 93]])
```

```
N=np.logspace(2,6,30,endpoint=True)
'Randomly sample from integers ranging from 0 to 10^6'
Treelist=[random.sample(range(0,1000000), int(n)) for n in N]
'Fix test sample'
Findsample=random.sample(range(0,10000000), 1000)
print(Treelist[0],Findsample[0])

timeused_N=[]
for i in range(len(N)):
    tree_generate=Tree()
    for item in Treelist[i]:
        tree_generate.add(item)
    starttime=time.perf_counter()
    for item in Findsample:
        tree_generate.__contains__(item)
    endtime=time.perf_counter()
    timeused=endtime-starttime
    timeused_N.append(timeused/1000)
# print(timeused_N)

plt.axes(xscale='log',yscale='log')
plt.plot(N,timeused_N,color='b',label='O(logn)')
plt.plot(N,N*1e-8,color='r',label='O(n)')
plt.xlabel('n')
plt.ylabel('time_used')
plt.title('In operation')
plt.legend()
plt.savefig('containtree.jpg')
plt.show()
```

```
N=np.logspace(2,6,100,endpoint=True)
Treelist=[random.sample(range(0,1000000), int(n)) for n in N]
timegenerate_N=[]
for i in range(0,len(N)):
    starttime=time.perf_counter()
    tree_generate=Tree()
    for item in Treelist[i]:
        tree_generate.add(item)
    endtime=time.perf_counter()
    timeused=endtime-starttime
    timegenerate_N.append(timeused)
plt.axes(xscale='log',yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N,timegenerate_N,color='b',label='O(nlogn)')
plt.plot(N,N*1e-6,color='r',label='O(n)')
plt.plot(N,N**2*1e-6,color='g',label='O(n**2)')
plt.title('generate Tree')
plt.legend()
plt.savefig('generateTime.jpg')
plt.show()
```

Code for Exercise4:

Exercise4.ipynb

```
import sys
import numpy as np
import random
import time
import matplotlib.pyplot as plt

def alg1(data):
  data = list(data)
  changes = True
  while changes:
    changes = False
    for i in range(len(data) - 1):
      if data[i + 1] < data[i]:
        data[i], data[i + 1] = data[i + 1], data[i]
        changes = True
  return data

def alg2(data):
  if len(data) <= 1:
    return data
  else:
    split = len(data) // 2
    left = iter(alg2(data[:split]))
    right = iter(alg2(data[split:]))
    result = []
    # note: this takes the top items off the left and right piles
    left_top = next(left)
    right_top = next(right)
    while True:
      if left_top < right_top:
        result.append(left_top)
        try:
          left_top = next(left)
        except StopIteration:
          # nothing remains on the left; add the right + return
          return result + [right_top] + list(right)
      else:
        result.append(right_top)
        try:
          right_top = next(right)
        except StopIteration:
          # nothing remains on the right; add the left + return
          return result + [left_top] + list(left)
```

```
data=random.sample(range(0,100),20)
print(alg1(data))
print(alg2(data))
```

```
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
```

```
N=np.logspace(1,4,100,endpoint=True)
timeused_1=[]
timeused_2=[]
for n in N:
    data_gen=data1(int(n))
    startime=time.perf_counter()
    alg1(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_1.append(timeused)

    startime=time.perf_counter()
    alg2(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_2.append(timeused)

plt.axes(xscale='log',yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N,timeused_1,color='b',label='algr1')
plt.plot(N,timeused_2,color='r',label='algr2')
plt.title('data1')
plt.legend()
plt.savefig('alg12ondata1.jpg')
plt.show()
```

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

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

```
N=np.logspace(1,4,100,endpoint=True)
timeused_1=[]
timeused_2=[]
for n in N:
    data_gen=data2(int(n))
    startime=time.perf_counter()
    alg1(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_1.append(timeused)

    startime=time.perf_counter()
    alg2(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_2.append(timeused)

plt.axes(xscale='log',yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N,timeused_1,color='b',label='algr1')
plt.plot(N,timeused_2,color='r',label='algr2')
plt.title('data2')
plt.legend()
plt.savefig('alg12ondata2.jpg')
plt.show()
```

```
N=np.logspace(1,4,100,endpoint=True)
timeused_1=[]
timeused_2=[]
for n in N:
    data_gen=data3(int(n))
    startime=time.perf_counter()
    alg1(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_1.append(timeused)

    startime=time.perf_counter()
    alg2(data_gen)
    endtime=time.perf_counter()
    timeused=endtime-startime
    timeused_2.append(timeused)

plt.axes(xscale='log',yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N,timeused_1,color='b',label='algr1')
plt.plot(N,timeused_2,color='r',label='algr2')
plt.title('data3')
plt.legend()
plt.savefig('alg12ondata3.jpg')
plt.show()
```

Parallelization code:

exercise4.py

```
import numpy as np
from multiprocessing import Process, Pool
import random
import time
import matplotlib.pyplot as plt

def alg1(data):
  data = list(data)
  changes = True
  while changes:
    changes = False
    for i in range(len(data) - 1):
      if data[i + 1] < data[i]:
        data[i], data[i + 1] = data[i + 1], data[i]
        changes = True
  return data

def alg2(data):
  if len(data) <= 1:
    return data
  else:
    split = len(data) // 2
    left = iter(alg2(data[:split]))
    right = iter(alg2(data[split:]))
    result = []
    # note: this takes the top items off the left and right piles
    left_top = next(left)
    right_top = next(right)
    while True:
      if left_top < right_top:
        result.append(left_top)
        try:
          left_top = next(left)
        except StopIteration:
          # nothing remains on the left; add the right + return
          return result + [right_top] + list(right)
      else:
        result.append(right_top)
        try:
          right_top = next(right)
        except StopIteration:
          # nothing remains on the right; add the left + return
          return result + [left_top] + list(left)

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
```

```python
if __name__ == "__main__":
    N = np.logspace(1, 6, 50, endpoint=True)
    timeused_1 = []
    timeused_2 = []
    pool = Pool(2)
    for n in N:
        data_gen = data1(int(n))

        startime = time.perf_counter()
        split = len(data_gen) // 2
        data_gen1 = data_gen[:split]
        data_gen2 = data_gen[split:]
        r1, r2 = pool.map(alg2,[data_gen1,data_gen2])
        iter1 = iter(r1)
        iter2 = iter(r2)
        result = []
        iter1_top = next(iter1)
        iter2_top = next(iter2)
        while True:
            if iter1_top < iter2_top:
                result.append(iter1_top)
                try:
                    iter1_top = next(iter1)
                except StopIteration:
                    result = result + [iter2_top] + list(iter2)
                    break
            else:
                result.append(iter2_top)
                try:
                    iter2_top = next(iter2)
                except StopIteration:
                    result = result + [iter1_top] + list(iter1)
                    break
        endtime = time.perf_counter()
        timeused = endtime - startime
        timeused_1.append(timeused)

        startime = time.perf_counter()
        alg2(data_gen)
        endtime = time.perf_counter()
        timeused = endtime - startime
        timeused_2.append(timeused)

    pool.close()
    pool.join()
```

```
plt.axes(xscale='log', yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N, timeused_1, color='b', label='parallel-algr2')
plt.plot(N, timeused_2, color='r', label='algr2')
plt.title('data1')
plt.legend()
plt.savefig('parallelondata1.jpg')
plt.show()

quotient = [a/b for a, b in zip(timeused_1, timeused_2)]
plt.axes(xscale='log', yscale='log')
plt.xlabel('n')
plt.ylabel('time_used')
plt.plot(N,quotient,label='parallel/original')
plt.axhline(1,color='gray',label='1')
plt.axhline(0.5, color='red',label='0.5')
plt.title('quotient')
plt.legend()
plt.savefig('quotient.jpg')
plt.show()

print('n=10^6,paralleled time is %f, original time is'%timeused_1[-1], timeused_2[-1])
```