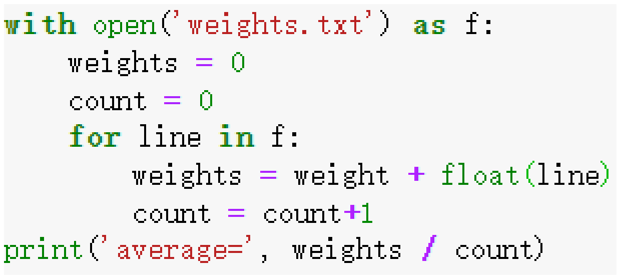
Exercise 1

Python requires far more than 4GB of RAM to process a 4GB list. This is because lists have many attributes, not just the values of the stored elements. A list will record the value, the data type, the reference number, and the position of the element in the list. Also, it is possible to creating backups of lists during the computing process. In addition, it is also possible that other software is taking up RAM. These are the possible reasons why RAM is running out.

I think the friend can use a computer with larger RAM or use a cloud server. And don't use list, use other data structures like array which only record values to store data.

For calculating the average, the code is modified to reduce RAM consumption by not using weights as variable to store value

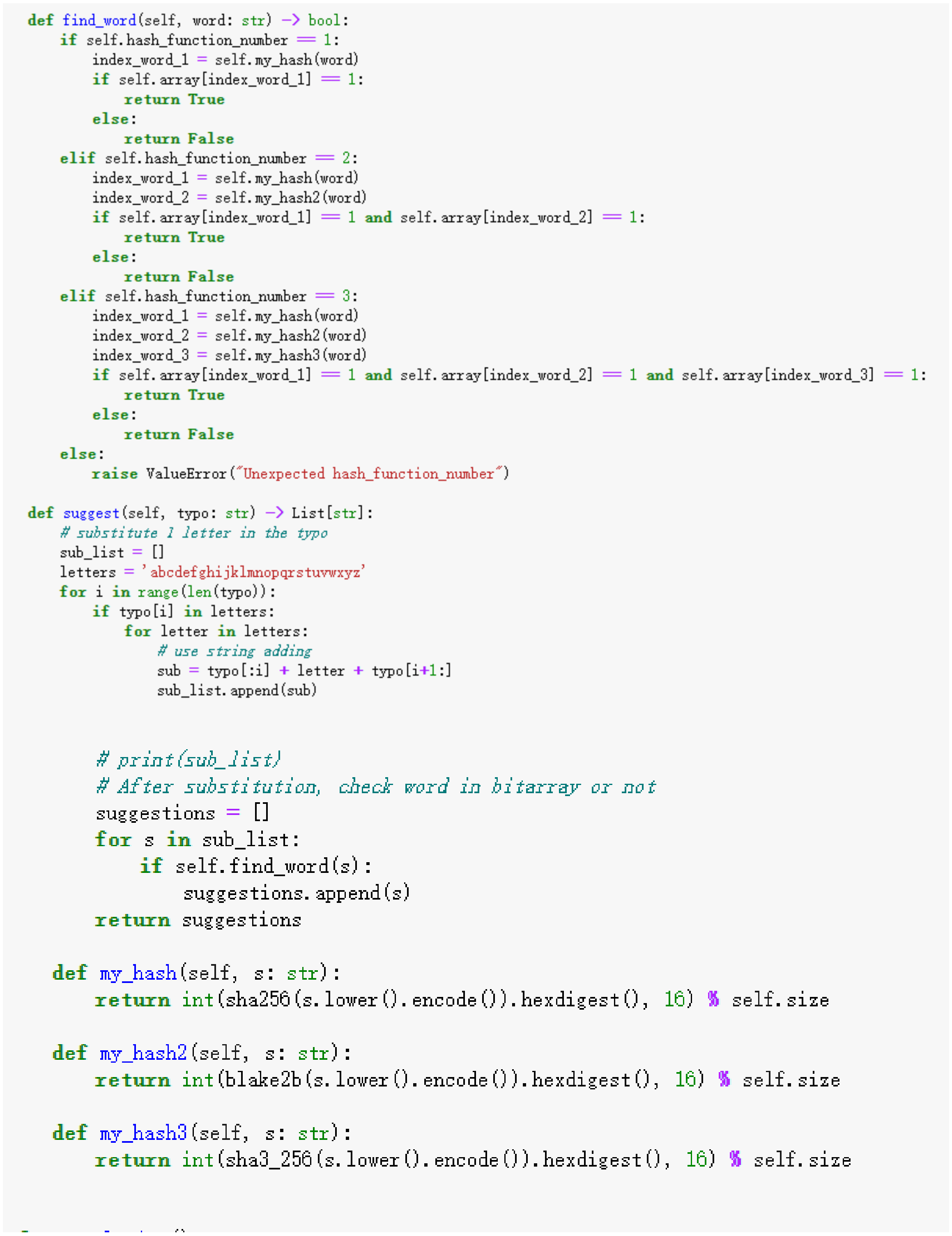


```
with open('weights.txt') as f:
    weights = 0
    count = 0
    for line in f:
        weights = weight + float(line)
        count = count+1
print('average=', weights / count)
```

Exercise 2：

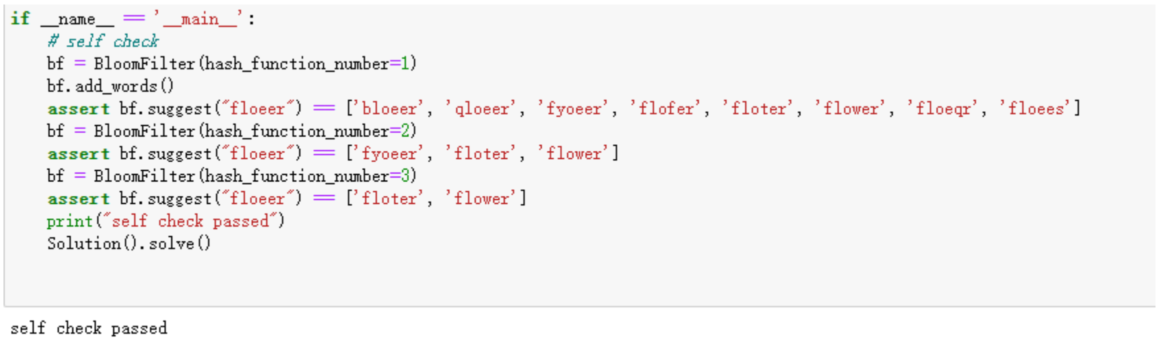
The Bloom filter is implemented as code below.

```
class BloomFilter:
    def __init__(self, size=10**7, hash_function_number=3):
        self.size = size
        self.hash_function_number = hash_function_number
        self.array = self.add_words()

    def add_words(self):
        array = bitarray.bitarray(self.size)
        array.setall(0)
        if self.hash_function_number == 1:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
        elif self.hash_function_number == 2:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
                    index_2 = self.my_hash2(word)
                    array[index_2] = 1
        elif self.hash_function_number == 3:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
                    index_2 = self.my_hash2(word)
                    array[index_2] = 1
                    index_3 = self.my_hash3(word)
                    array[index_3] = 1
        else:
            raise ValueError("Unexpected hash_function_number")
        return array
```



```
    def find_word(self, word: str) -> bool:
        if self.hash_function_number == 1:
            index_word_1 = self.my_hash(word)
            if self.array[index_word_1] == 1:
                return True
            else:
                return False
        elif self.hash_function_number == 2:
            index_word_1 = self.my_hash(word)
            index_word_2 = self.my_hash2(word)
            if self.array[index_word_1] == 1 and self.array[index_word_2] == 1:
                return True
            else:
                return False
        elif self.hash_function_number == 3:
            index_word_1 = self.my_hash(word)
            index_word_2 = self.my_hash2(word)
            index_word_3 = self.my_hash3(word)
            if self.array[index_word_1] == 1 and self.array[index_word_2] == 1 and self.array[index_word_3] == 1:
                return True
            else:
                return False
        else:
            raise ValueError("Unexpected hash_function_number")

    def suggest(self, typo: str) -> List[str]:
        # substitute 1 letter in the typo
        sub_list = []
        letters = 'abcdefghijklmnopqrstuvwxyz'
        for i in range(len(typo)):
            if typo[i] in letters:
                for letter in letters:
                    # use string adding
                    sub = typo[:i] + letter + typo[i+1:]
                    sub_list.append(sub)


        # print(sub_list)
        # After substitution, check word in bitarray or not
        suggestions = []
        for s in sub_list:
            if self.find_word(s):
                suggestions.append(s)
        return suggestions

    def my_hash(self, s: str):
        return int(sha256(s.lower().encode()).hexdigest(), 16) % self.size

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

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

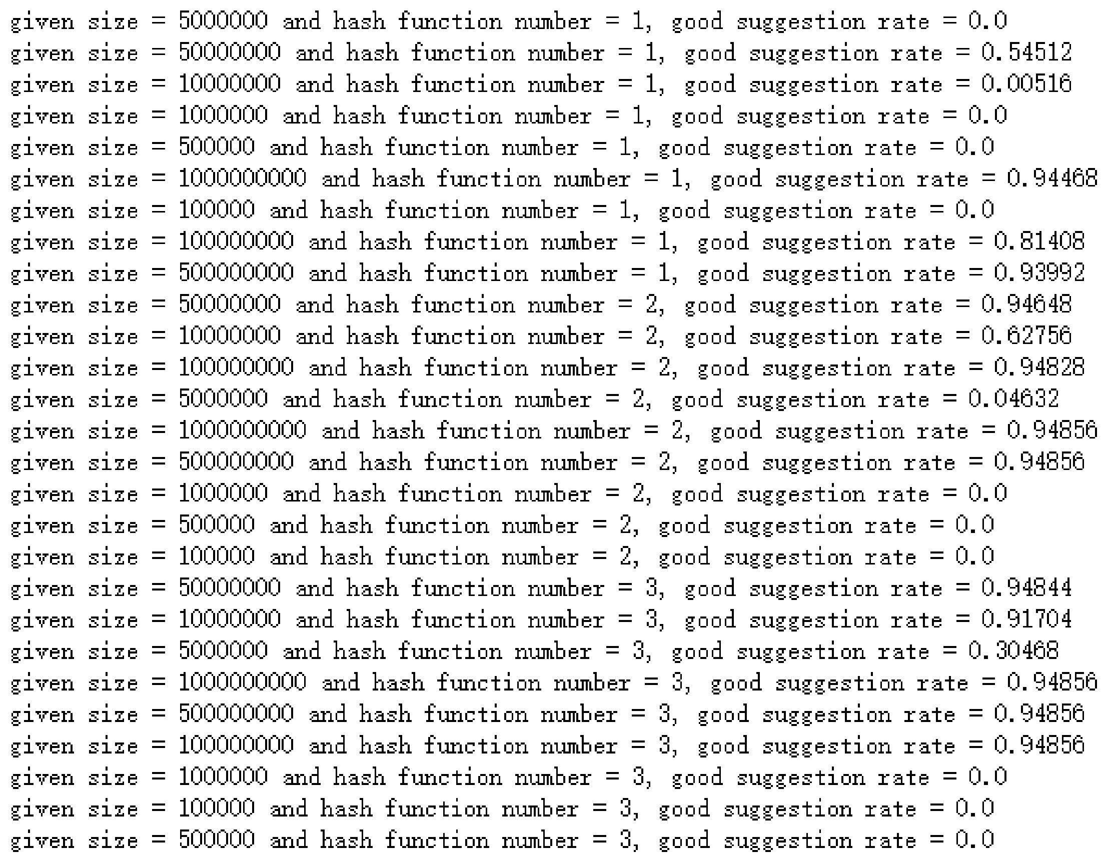
See detailed solution in code appendix. My solution is tested by following code and result is passed.



```python
if __name__ == '__main__':
    # self check
    bf = BloomFilter(hash_function_number=1)
    bf.add_words()
    assert bf.suggest("floeer") == ['bloeer', 'qloeer', 'fyoeer', 'flofer', 'floter', 'flower', 'floeqr', 'floees']
    bf = BloomFilter(hash_function_number=2)
    assert bf.suggest("floeer") == ['fyoeer', 'floter', 'flower']
    bf = BloomFilter(hash_function_number=3)
    assert bf.suggest("floeer") == ['floter', 'flower']
    print("self check passed")
    Solution().solve()
```

```
self check passed
```

The accuracy of each of the three hash-set at different data amounts is as follows. The order of magnitude to reach 90% accuracy is about 500,000,000, 50,000,000, and 100,000,000 respectively. The highest accuracy achievable is 94.86%



```
given size = 5000000 and hash function number = 1, good suggestion rate = 0.0
given size = 50000000 and hash function number = 1, good suggestion rate = 0.54512
given size = 10000000 and hash function number = 1, good suggestion rate = 0.00516
given size = 1000000 and hash function number = 1, good suggestion rate = 0.0
given size = 500000 and hash function number = 1, good suggestion rate = 0.0
given size = 1000000000 and hash function number = 1, good suggestion rate = 0.94468
given size = 100000 and hash function number = 1, good suggestion rate = 0.0
given size = 100000000 and hash function number = 1, good suggestion rate = 0.81408
given size = 500000000 and hash function number = 1, good suggestion rate = 0.93992
given size = 50000000 and hash function number = 2, good suggestion rate = 0.94648
given size = 10000000 and hash function number = 2, good suggestion rate = 0.62756
given size = 100000000 and hash function number = 2, good suggestion rate = 0.94828
given size = 5000000 and hash function number = 2, good suggestion rate = 0.04632
given size = 1000000000 and hash function number = 2, good suggestion rate = 0.94856
given size = 500000000 and hash function number = 2, good suggestion rate = 0.94856
given size = 1000000 and hash function number = 2, good suggestion rate = 0.0
given size = 500000 and hash function number = 2, good suggestion rate = 0.0
given size = 100000 and hash function number = 2, good suggestion rate = 0.0
given size = 50000000 and hash function number = 3, good suggestion rate = 0.94844
given size = 10000000 and hash function number = 3, good suggestion rate = 0.91704
given size = 5000000 and hash function number = 3, good suggestion rate = 0.30468
given size = 1000000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 500000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 100000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 1000000 and hash function number = 3, good suggestion rate = 0.0
given size = 100000 and hash function number = 3, good suggestion rate = 0.0
given size = 500000 and hash function number = 3, good suggestion rate = 0.0
```

Exercise 3：

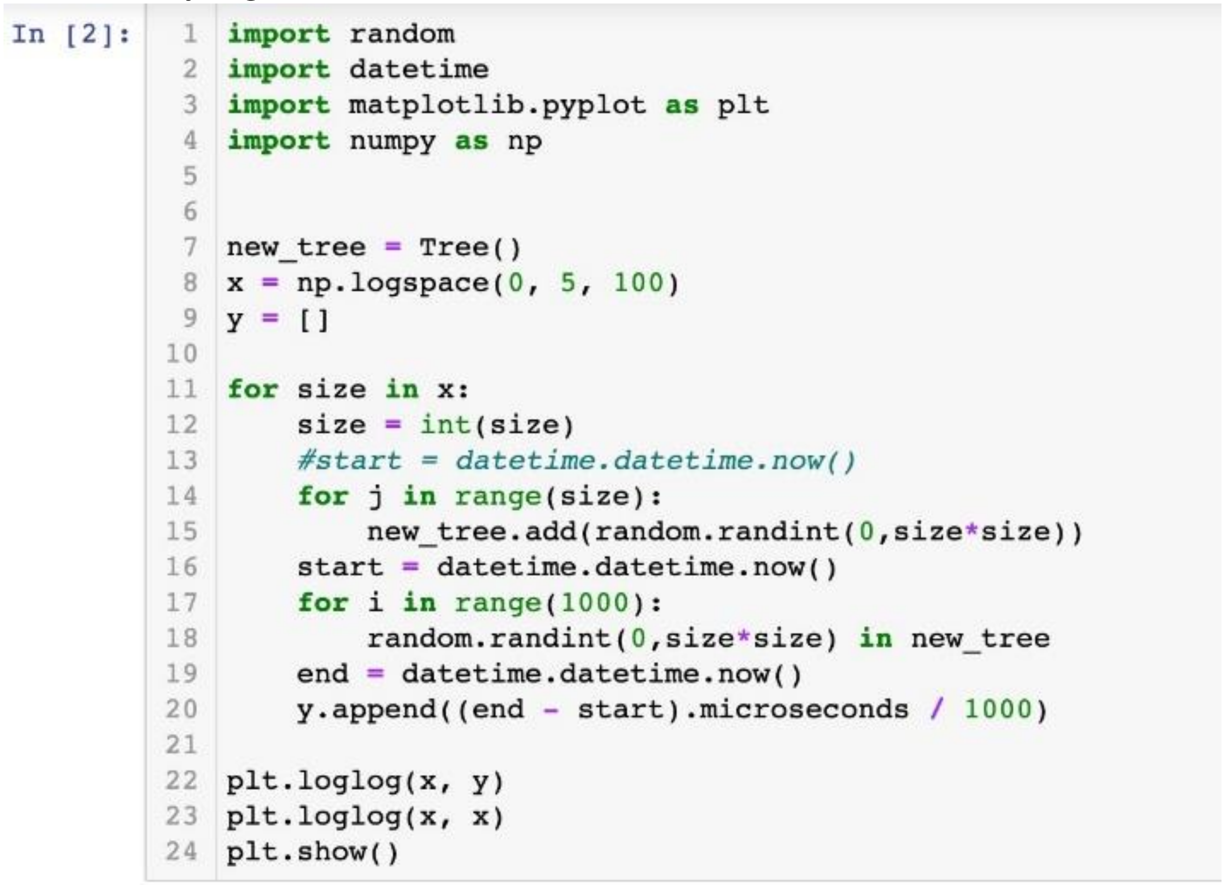
First I expand the code to a binary tree and test the tree by using the given code.

```
class Tree:
    def __init__(self,value=None):
        self._value = value
        self.left = None
        self.right = None

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

    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(37 in my_tree)
print(55 in my_tree)
```

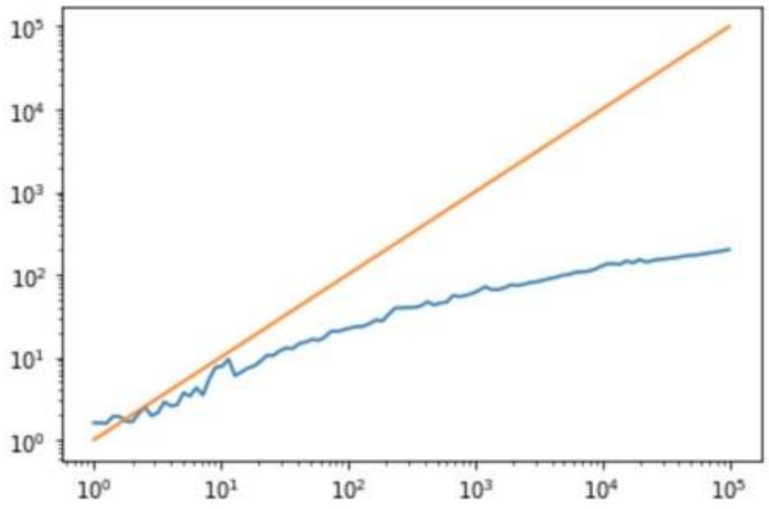
```
True
True
```

For sufficiently large in, n starts to flatten



In [2]:

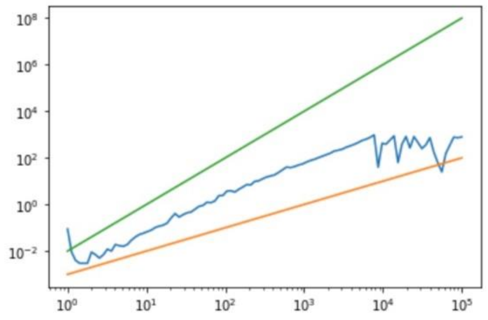
```python
import random
import datetime
import matplotlib.pyplot as plt
import numpy as np


new_tree = Tree()
x = np.logspace(0, 5, 100)
y = []

for size in x:
    size = int(size)
    #start = datetime.datetime.now()
    for j in range(size):
        new_tree.add(random.randint(0,size*size))
    start = datetime.datetime.now()
    for i in range(1000):
        random.randint(0,size*size) in new_tree
    end = datetime.datetime.now()
    y.append((end - start).microseconds / 1000)

plt.loglog(x, y)
plt.loglog(x, x)
plt.show()
```



The curve of function lies between O(n) and O(n**2), this is the evidence that the time to set up the tree is O(n log n)

```
In [3]:
new_tree = Tree()
x = np.logspace(0, 5, 100)
y = []

for size in x:
    size = int(size)
    #start = datetime.datetime.now()

    start = datetime.datetime.now()
    for j in range(size):
        new_tree.add(random.randint(0,size*size))
    end = datetime.datetime.now()

    y.append((end - start).microseconds / 1000)
```

```
In [9]:
plt.loglog(x, y)
plt.loglog(x, x/1000)
plt.loglog(x, [a**2/100 for a in x])

plt.show()
```



Exercise 4:

Both algorithms sort the series in ascending order

```
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 = [1,22,333,4444,55555]
alg1(data)
```

```
[1, 22, 333, 4444, 55555]
```

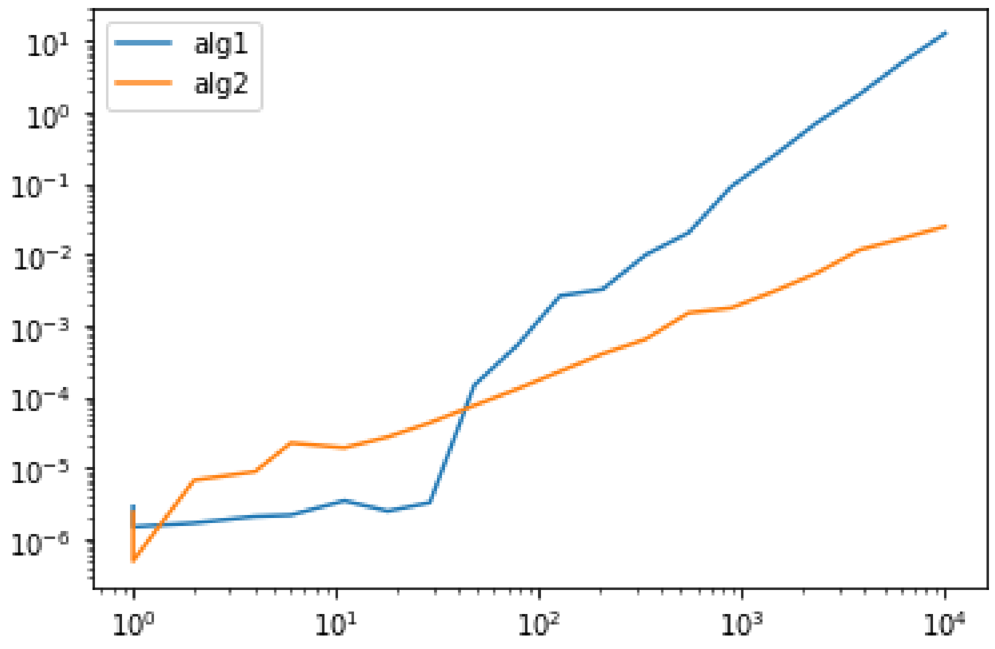
```
data = [66,6,666,666666,66666666666666,66666,6666666666]
alg2(data)
```

```
[6, 66, 666, 66666, 666666, 6666666666, 66666666666666]
```

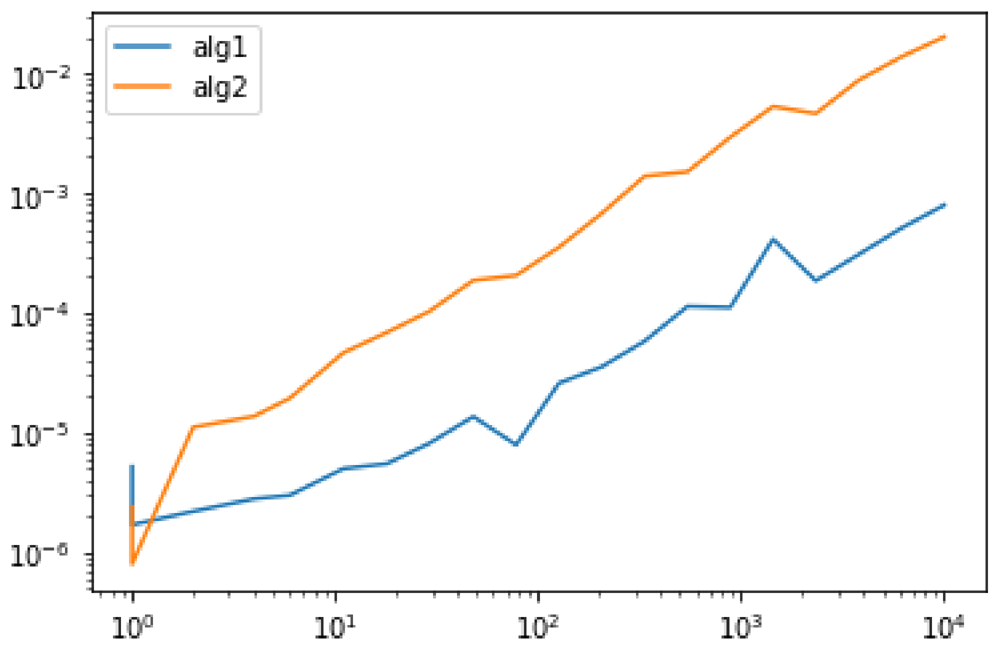
Alg1 will change the position of a number when it is smaller than the previous one until the whole sequence is sorted.

Alg2 splits the entire array until there is only one smallest number in the list. Then two of the lists are compared and the smaller one is merged into the larger one. For the two lists that have been sorted, their minimum values are compared, and then the smaller one and the rest of the numbers are merged into the new list. By recursion, the whole array is sorted.

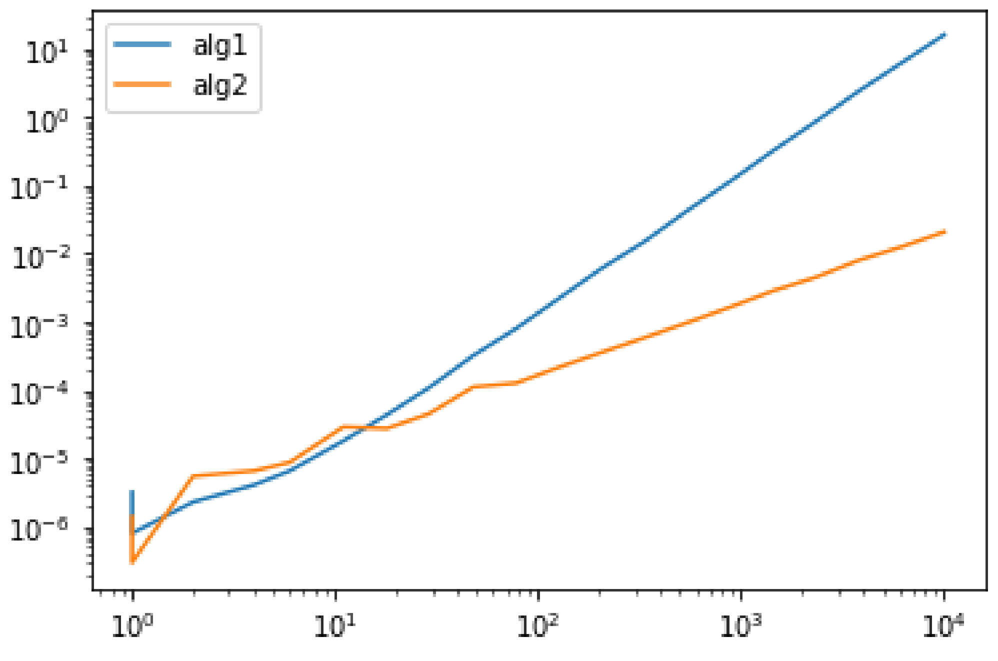
For data set 1, Algorithm 1 performs better when the data volume is small, but as the data volume becomes larger, Algorithm 2 outperforms Algorithm 1



For data set 2, Algorithm 1 performs better when data is sorted



For data set 3, Algorithm 2 performs better



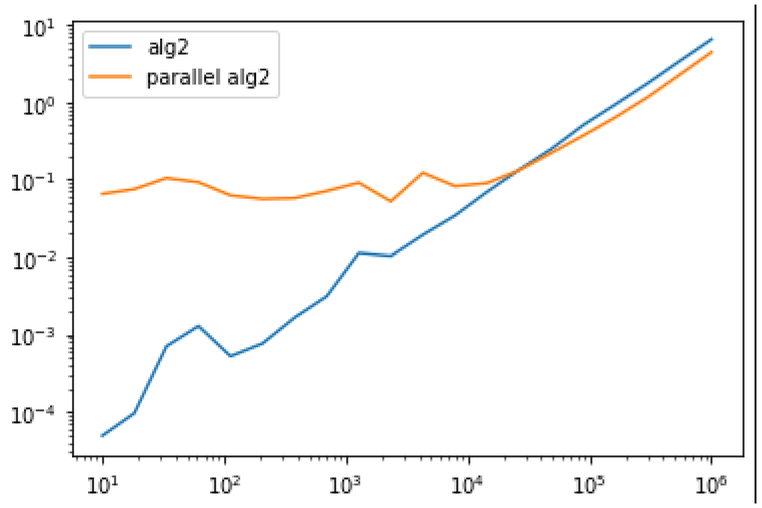
Only on the ordered dataset, Algorithm 1 performs better. And on any data set, Algorithm 2 has stable performance, so for any other data set, I would prefer Algorithm 2.

Explanation of parallelization of alg2:

To parallelize alg2, we use the map function under multiprocessing. Pool (). After that, the data is divided into two parts and the two parts are sorted at the same time using parallel programming. Finally, we merge the two sorted sub lists together.

Implementation of merge sort in parallel:

Please see implementation in code appendix



The result of parallelized alg2 is shown above.

When the size is small, it takes some time to initialize. So, the parallel version is slower than the original version. But when running with more data, the parallel version is roughly twice as fast as the original version. However, with further expansion of the data volume, the performance of the two methods tends to average out with no significant difference.

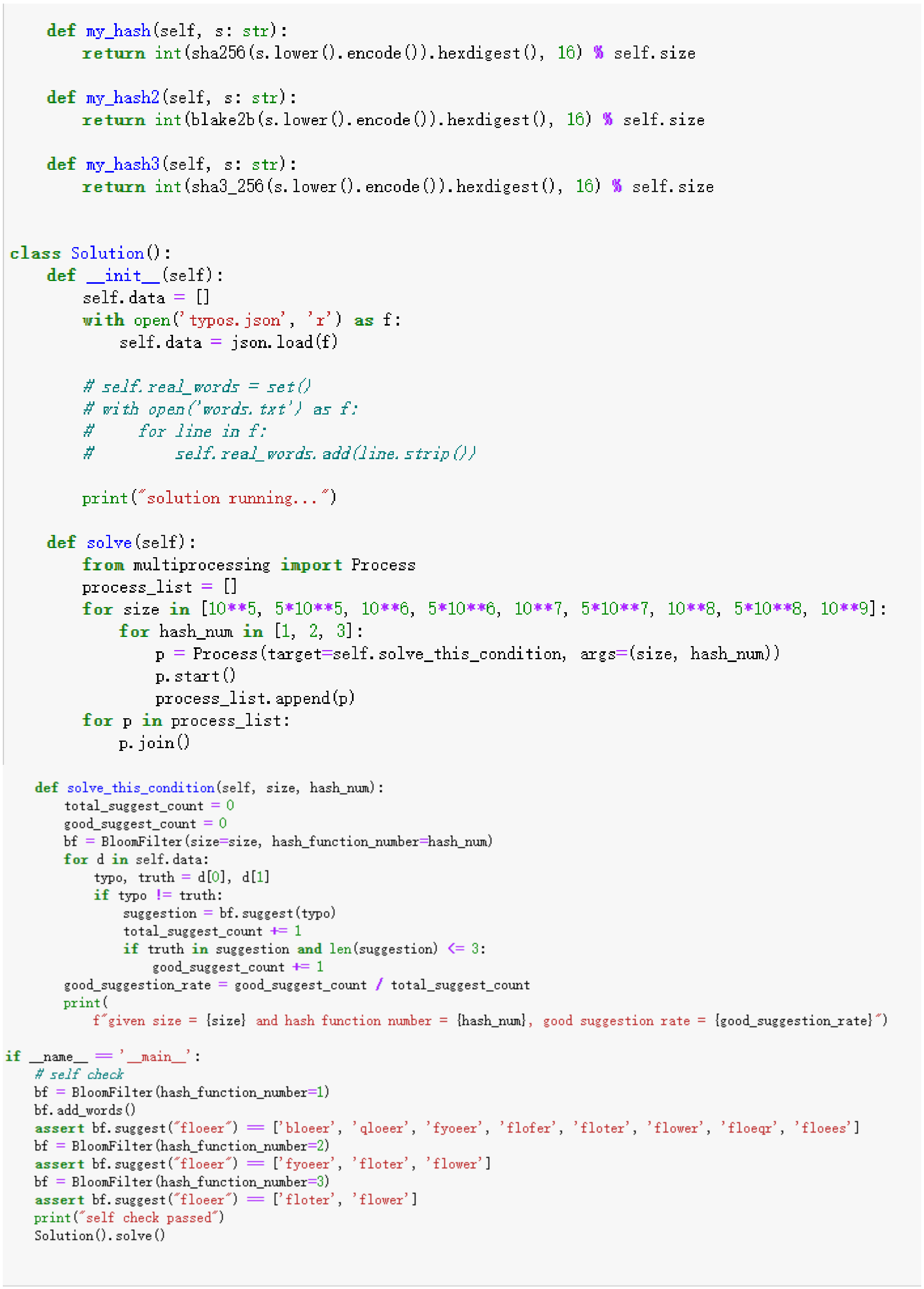
Code:

Exercise 1:

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

Exercise 2:

```python
import json
import math
from typing import List

import bitarray
from hashlib import sha3_256, sha256, blake2b

class BloomFilter:
    def __init__(self, size=10**7, hash_function_number=3):
        self.size = size
        self.hash_function_number = hash_function_number
        self.array = self.add_words()

    def add_words(self):
        array = bitarray.bitarray(self.size)
        array.setall(0)
        if self.hash_function_number == 1:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
        elif self.hash_function_number == 2:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
                    index_2 = self.my_hash2(word)
                    array[index_2] = 1
        elif self.hash_function_number == 3:
            with open("words.txt") as f:
                for line in f:
                    word = line.strip()
                    index_1 = self.my_hash(word)
                    array[index_1] = 1
                    index_2 = self.my_hash2(word)
                    array[index_2] = 1
                    index_3 = self.my_hash3(word)
                    array[index_3] = 1
        else:
            raise ValueError("Unexpected hash_function_number")
        return array
    def find_word(self, word: str) -> bool:
        if self.hash_function_number == 1:
            index_word_1 = self.my_hash(word)
            if self.array[index_word_1] == 1:
                return True
            else:
                return False
        elif self.hash_function_number == 2:
            index_word_1 = self.my_hash(word)
            index_word_2 = self.my_hash2(word)
            if self.array[index_word_1] == 1 and self.array[index_word_2] == 1:
                return True
            else:
                return False
        elif self.hash_function_number == 3:
            index_word_1 = self.my_hash(word)
            index_word_2 = self.my_hash2(word)
            index_word_3 = self.my_hash3(word)
            if self.array[index_word_1] == 1 and self.array[index_word_2] == 1 and self.array[index_word_3] == 1:
                return True
            else:
                return False
        else:
            raise ValueError("Unexpected hash_function_number")

    def suggest(self, typo: str) -> List[str]:
        # substitute 1 letter in the typo
        sub_list = []
        letters = 'abcdefghijklmnopqrstuvwxyz'
        for i in range(len(typo)):
            if typo[i] in letters:
                for letter in letters:
                    # use string adding
                    sub = typo[:i] + letter + typo[i+1:]
                    sub_list.append(sub)

        # print(sub_list)
        # After substitution, check word in bitarray or not
        suggestions = []
        for s in sub_list:
            if self.find_word(s):
                suggestions.append(s)
        return suggestions
```



```python
    def my_hash(self, s: str):
        return int(sha256(s.lower().encode()).hexdigest(), 16) % self.size

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

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


class Solution():
    def __init__(self):
        self.data = []
        with open('typos.json', 'r') as f:
            self.data = json.load(f)

        # self.real_words = set()
        # with open('words.txt') as f:
        #     for line in f:
        #         self.real_words.add(line.strip())

        print("solution running...")

    def solve(self):
        from multiprocessing import Process
        process_list = []
        for size in [10**5, 5*10**5, 10**6, 5*10**6, 10**7, 5*10**7, 10**8, 5*10**8, 10**9]:
            for hash_num in [1, 2, 3]:
                p = Process(target=self.solve_this_condition, args=(size, hash_num))
                p.start()
                process_list.append(p)
        for p in process_list:
            p.join()

    def solve_this_condition(self, size, hash_num):
        total_suggest_count = 0
        good_suggest_count = 0
        bf = BloomFilter(size=size, hash_function_number=hash_num)
        for d in self.data:
            typo, truth = d[0], d[1]
            if typo != truth:
                suggestion = bf.suggest(typo)
                total_suggest_count += 1
                if truth in suggestion and len(suggestion) <= 3:
                    good_suggest_count += 1
        good_suggestion_rate = good_suggest_count / total_suggest_count
        print(
            f"given size = {size} and hash function number = {hash_num}, good suggestion rate = {good_suggestion_rate}")

if __name__ == '__main__':
    # self check
    bf = BloomFilter(hash_function_number=1)
    bf.add_words()
    assert bf.suggest("floeer") == ['bloeer', 'qloeer', 'fyoeer', 'flofer', 'floter', 'flower', 'floeqr', 'floees']
    bf = BloomFilter(hash_function_number=2)
    assert bf.suggest("floeer") == ['fyoeer', 'floter', 'flower']
    bf = BloomFilter(hash_function_number=3)
    assert bf.suggest("floeer") == ['floter', 'flower']
    print("self check passed")
    Solution().solve()
```

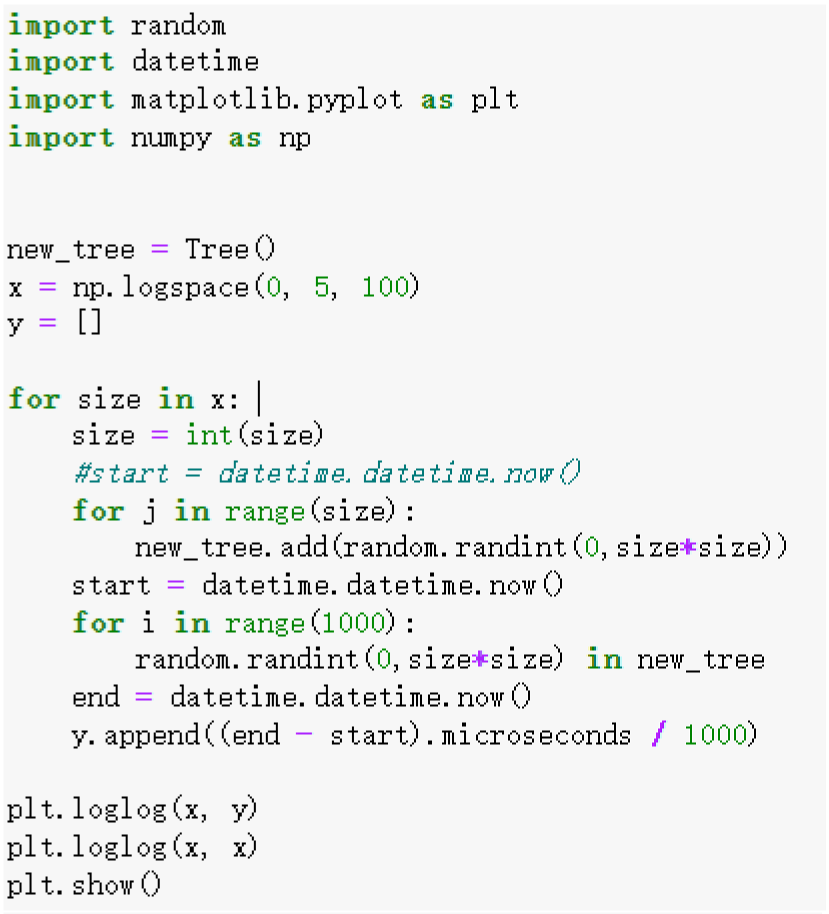
```
self check passed
solution running...
given size = 5000000 and hash function number = 1, good suggestion rate = 0.0
given size = 50000000 and hash function number = 1, good suggestion rate = 0.54512
given size = 10000000 and hash function number = 1, good suggestion rate = 0.00516
given size = 1000000 and hash function number = 1, good suggestion rate = 0.0
given size = 500000 and hash function number = 1, good suggestion rate = 0.0
given size = 1000000000 and hash function number = 1, good suggestion rate = 0.94468
given size = 100000 and hash function number = 1, good suggestion rate = 0.0
given size = 100000000 and hash function number = 1, good suggestion rate = 0.81408
given size = 500000000 and hash function number = 1, good suggestion rate = 0.93992
given size = 50000000 and hash function number = 2, good suggestion rate = 0.94648
given size = 10000000 and hash function number = 2, good suggestion rate = 0.62756
given size = 100000000 and hash function number = 2, good suggestion rate = 0.94828
given size = 5000000 and hash function number = 2, good suggestion rate = 0.04632
given size = 1000000000 and hash function number = 2, good suggestion rate = 0.94856
given size = 500000000 and hash function number = 2, good suggestion rate = 0.94856
given size = 1000000 and hash function number = 2, good suggestion rate = 0.0
given size = 500000 and hash function number = 2, good suggestion rate = 0.0
given size = 100000 and hash function number = 2, good suggestion rate = 0.0
given size = 50000000 and hash function number = 3, good suggestion rate = 0.94844
given size = 10000000 and hash function number = 3, good suggestion rate = 0.91704
given size = 5000000 and hash function number = 3, good suggestion rate = 0.30468
given size = 1000000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 500000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 100000000 and hash function number = 3, good suggestion rate = 0.94856
given size = 1000000 and hash function number = 3, good suggestion rate = 0.0
given size = 100000 and hash function number = 3, good suggestion rate = 0.0
given size = 500000 and hash function number = 3, good suggestion rate = 0.0
```

Exercise 3:

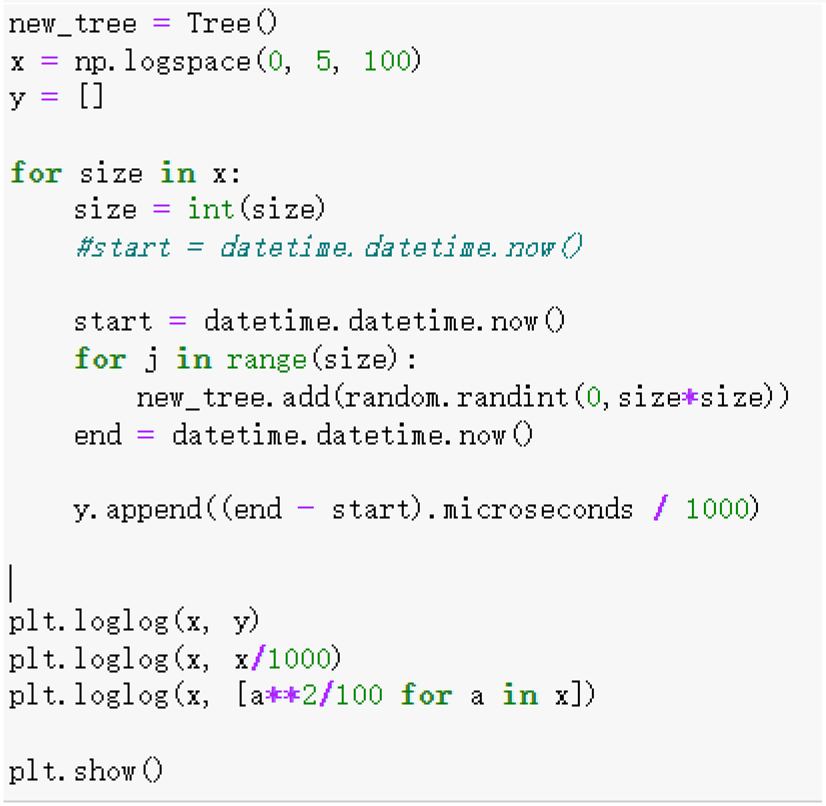
```
class Tree:
    def __init__(self,value=None):
        self._value = value
        self.left = None
        self.right = None

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

    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(37 in my_tree)
print(55 in my_tree)
```

```
True
True
```



```python
import random
import datetime
import matplotlib.pyplot as plt
import numpy as np


new_tree = Tree()
x = np.logspace(0, 5, 100)
y = []

for size in x:
    size = int(size)
    #start = datetime.datetime.now()
    for j in range(size):
        new_tree.add(random.randint(0,size*size))
    start = datetime.datetime.now()
    for i in range(1000):
        random.randint(0,size*size) in new_tree
    end = datetime.datetime.now()
    y.append((end - start).microseconds / 1000)

plt.loglog(x, y)
plt.loglog(x, x)
plt.show()
```



```python
new_tree = Tree()
x = np.logspace(0, 5, 100)
y = []

for size in x:
    size = int(size)
    #start = datetime.datetime.now()

    start = datetime.datetime.now()
    for j in range(size):
        new_tree.add(random.randint(0,size*size))
    end = datetime.datetime.now()

    y.append((end - start).microseconds / 1000)


plt.loglog(x, y)
plt.loglog(x, x/1000)
plt.loglog(x, [a**2/100 for a in x])

plt.show()
```

Exercise 4:

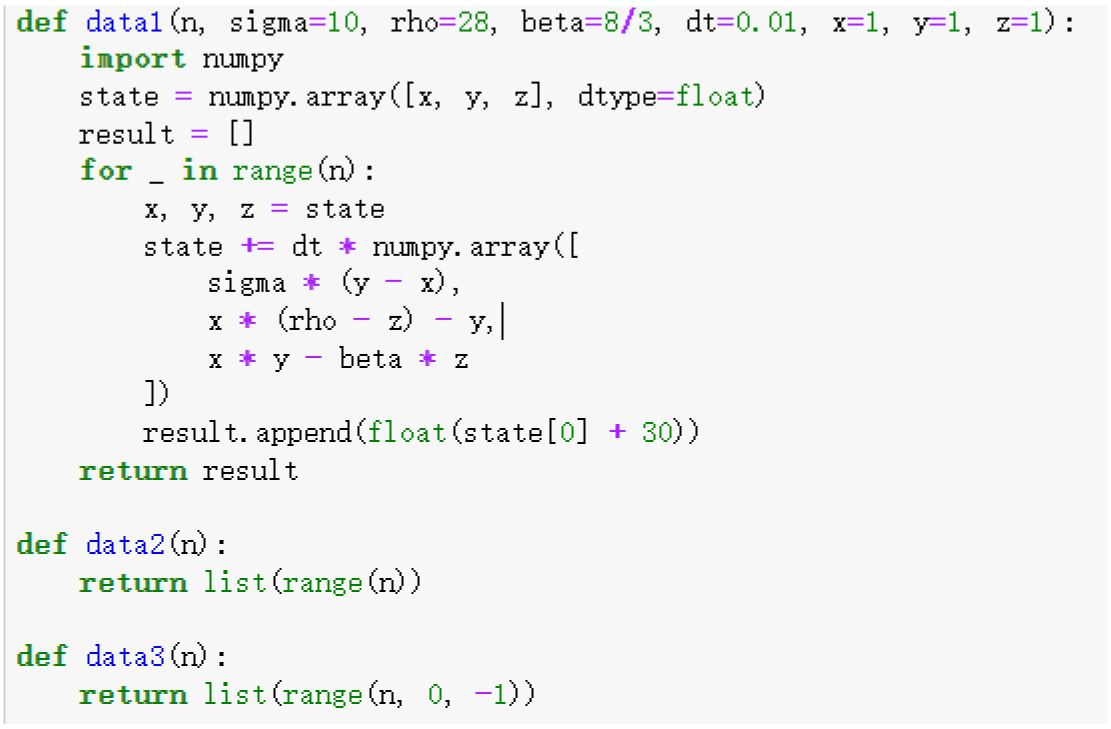
```python
def alg1(data):
    data = list(data)
    changes = True
    while changes:
        changes = False
        for i in range(len(data) - 1):
            if data[i + 1] < data[i]:
                data[i], data[i + 1] = data[i + 1], data[i]
                changes = True
    return data

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)
```

```python
data = [1,22,333,4444,55555]
alg1(data)
```

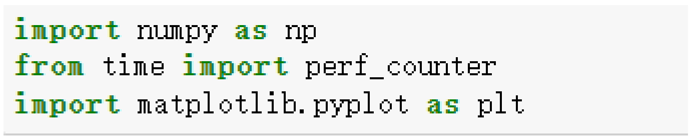
```
[1, 22, 333, 4444, 55555]
```

```python
data = [66,6,666,666666,66666666666666,66666,6666666666]
alg2(data)
```

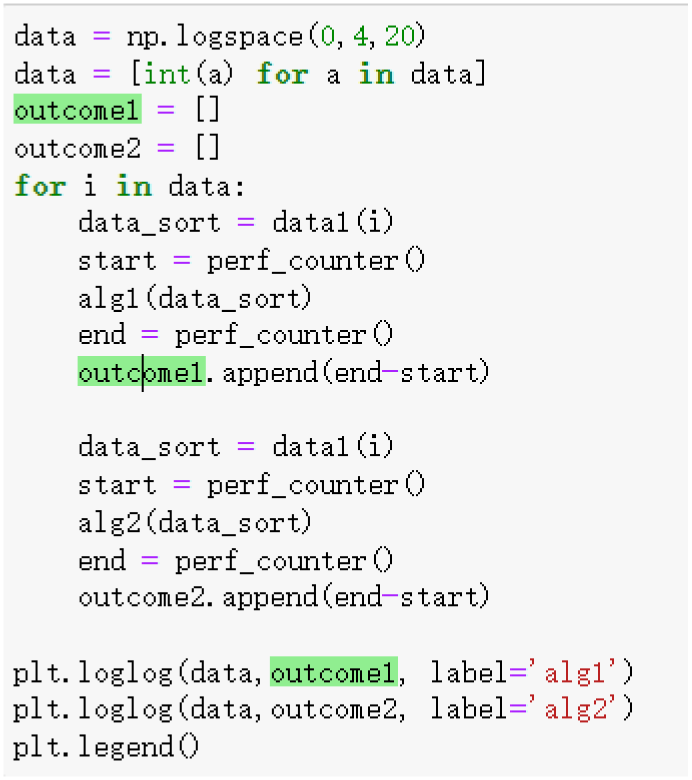
```
[6, 66, 666, 66666, 666666, 6666666666, 66666666666666]
```



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

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

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



```python
import numpy as np
from time import perf_counter
import matplotlib.pyplot as plt
```



```python
data = np.logspace(0,4,20)
data = [int(a) for a in data]
outcome1 = []
outcome2 = []
for i in data:
    data_sort = data1(i)
    start = perf_counter()
    alg1(data_sort)
    end = perf_counter()
    outcome1.append(end-start)

    data_sort = data1(i)
    start = perf_counter()
    alg2(data_sort)
    end = perf_counter()
    outcome2.append(end-start)

plt.loglog(data,outcome1, label='alg1')
plt.loglog(data,outcome2, label='alg2')
plt.legend()
```

```
data = np.logspace(0,4,20)
data = [int(a) for a in data]
outcome1 = []
outcome2 = []
for i in data:
    data_sort = data2(i)
    start = perf_counter()
    alg1(data_sort)
    end = perf_counter()
    outcome1.append(end-start)

    data_sort = data2(i)
    start = perf_counter()
    alg2(data_sort)
    end = perf_counter()
    outcome2.append(end-start)

plt.loglog(data,outcome1, label='alg1')
plt.loglog(data,outcome2, label='alg2')
plt.legend()
```

```
data = np.logspace(0,4,20)
data = [int(a) for a in data]
outcome1 = []
outcome2 = []
for i in data:
    data_sort = data3(i)
    start = perf_counter()
    alg1(data_sort)
    end = perf_counter()
    outcome1.append(end-start)

    data_sort = data3(i)
    start = perf_counter()
    alg2(data_sort)
    end = perf_counter()
    outcome2.append(end-start)

plt.loglog(data,outcome1, label='alg1')
plt.loglog(data,outcome2, label='alg2')
plt.legend()
```

```python
import multiprocessing
from multiprocessing import Process
import time
import matplotlib.pyplot as plt
import numpy as np

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 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
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 worker(data, id, return_dict):
    res = alg2(data)
    return_dict[id] = res
```

```python
def new_alg2(data):

    split = len(data) // 2
    left = data[:split]
    right = data[split:]

    # never used queue for storing large amount of data!
    # q = Queue()
    # use manger instead
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    process_list = []

    p1 = Process(target=worker, args=(left,'left', return_dict))
    process_list.append(p1)
    p1.start()

    p2 = Process(target=worker, args=(right, 'right', return_dict))
    process_list.append(p2)
    p2.start()

    # wait for processes to finished
    p1.join()
    p2.join()

    # combined the result from the processes
    left = iter(list(return_dict['left']))
    right = iter(list(return_dict['right']))
    left_top = next(left)
    right_top = next(right)
    result = []
    while True:
        if left_top < right_top:
            result.append(left_top)
            try:
                left_top = next(left)
            except StopIteration:
                # nothing remains on the left; add the right + return
                return result + [right_top] + list(right)
        else:
            result.append(right_top)
            try:
                right_top = next(right)
            except StopIteration:
                # nothing remains on the right; add the left + return
                return result + [left_top] + list(left)
```

```
if __name__ == '__main__':
    data = np.logspace(1,6,20)
    # data = data1(1000000)
    y1 = []
    y2 = []
    for s in data:
        s = int(s)
        x = data1(s)
        start_time = time.time()
        alg2(x)
        end_time = time.time()
        y1.append(end_time - start_time)

        x = data1(s)
        start_time = time.time()
        new_alg2(x)
        end_time = time.time()
        y2.append(end_time - start_time)

plt.loglog(data, y1, label='alg2')
plt.loglog(data, y2, label='parallel alg2')
plt.legend()
plt.show()
plt.savefig('pbnb.png')
```