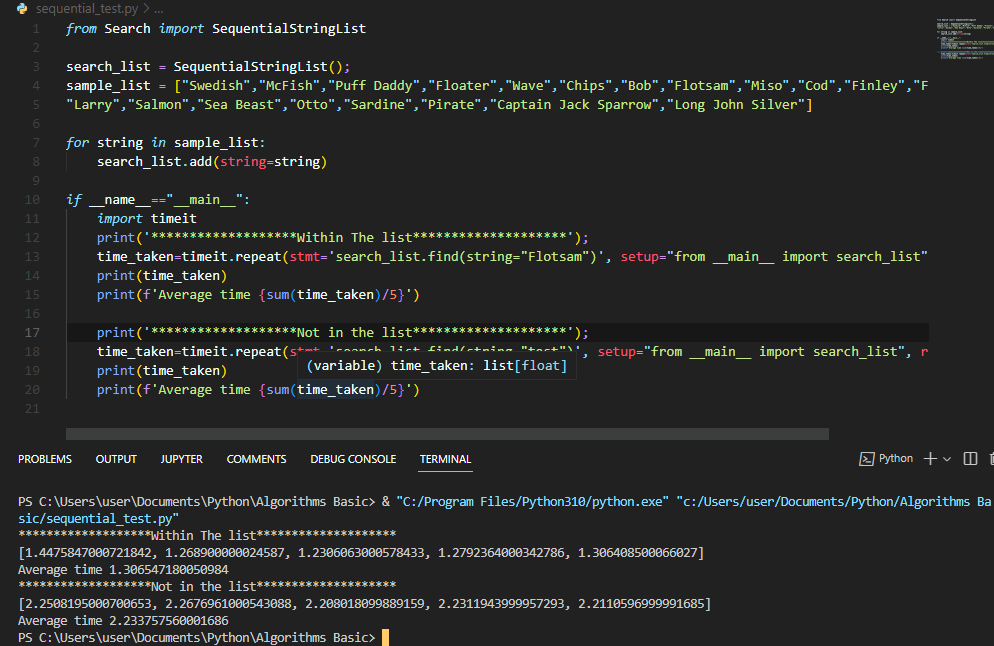
# **ASSIGNMENT 3 SUBMISSION**

## **QUESTION ONE: SEARCHING**

### **SEQUENTIAL SEARCH**

**TEST Run RESULTS**

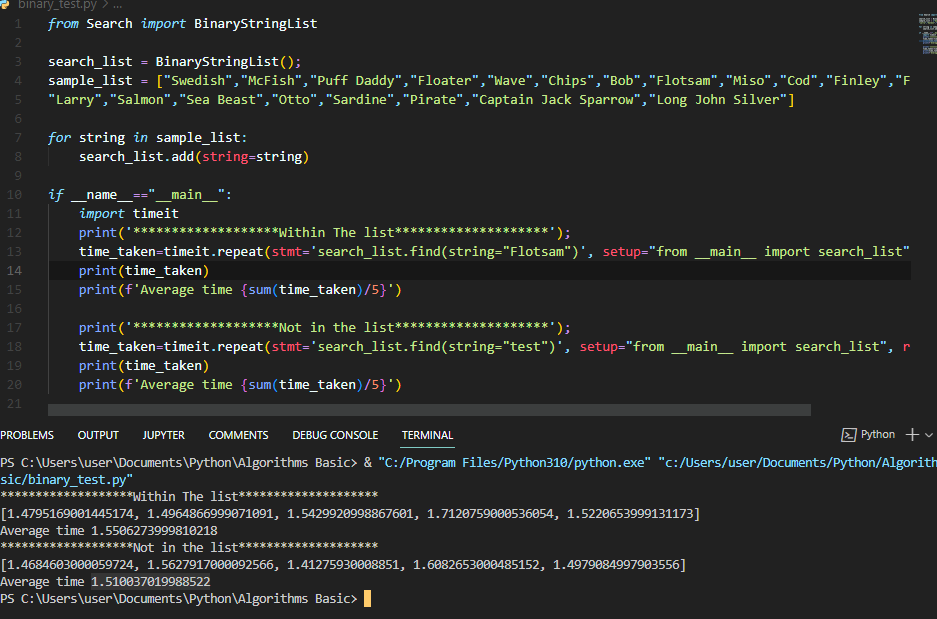
|  |  |  |
| --- | --- | --- |
| **Runs** | **With a word Not in The List Execution time** | **With a word Not in The List Execution time** |
|  | 1.4475847000721842 | 2.2508195000700653 |
|  | 1.268900000024587 | 2.2676961000543088 |
|  | 1.2306063000578433 | 2.208018099889159 |
|  | 1.2792364000342786 | 2.2311943999957293 |
|  | 1.306408500066027 | 2.2110596999991685 |
| **Average** | **1.306547180050984** | **2.233757560001686** |



### **BINARY SEARCH**

**TEST RUN RESULTS**

|  |  |  |
| --- | --- | --- |
| **Runs** | **With a word Not in The List Execution time** | **With a word Not in The List Execution time** |
|  | 1.4795169001445174 | 1.4684603000059724 |
|  | 1.4964866999071091 | 1.5627917000092566 |
|  | 1.5429920998867601 | 1.41275930008851 |
|  | 1.7120759000536054 | 1.6082653000485152 |
|  | 1.5220653999131173 | 1.4979084997903556 |
| **Average** | **1.5506273999810218** | **1.510037019988522** |



### **BINARY SEARCH**

**Sequential search**

In Sequential search loops though the values with in a list and compares the current one to be searched.

In the worst case we’ll need to scan through the whole array which being of will give us a worst case time complexity of.

According to the test results; Sequential search’s time for execution is almost twice as much when the string to search for is not found thus making the speed to slow down with larger data sets.

**Binary search**

In binary Search, We are using the divide and conquer method to find the string in the array.

We first divide the array, then check If the other half that satisfies the comparison requirement.

Recurrence for binary search is . This is because we are always considering have of the input array and throwing out the other half.

By using Divide and Conquer Master Theorem, we get,

Therefor the worst case time complexity is

This makes Binary search for efficient and fast for large data sets.

## **QUESTION TWO: SORTING**

### **BUBBLE SORT**

**TEST RUN RESULTS**

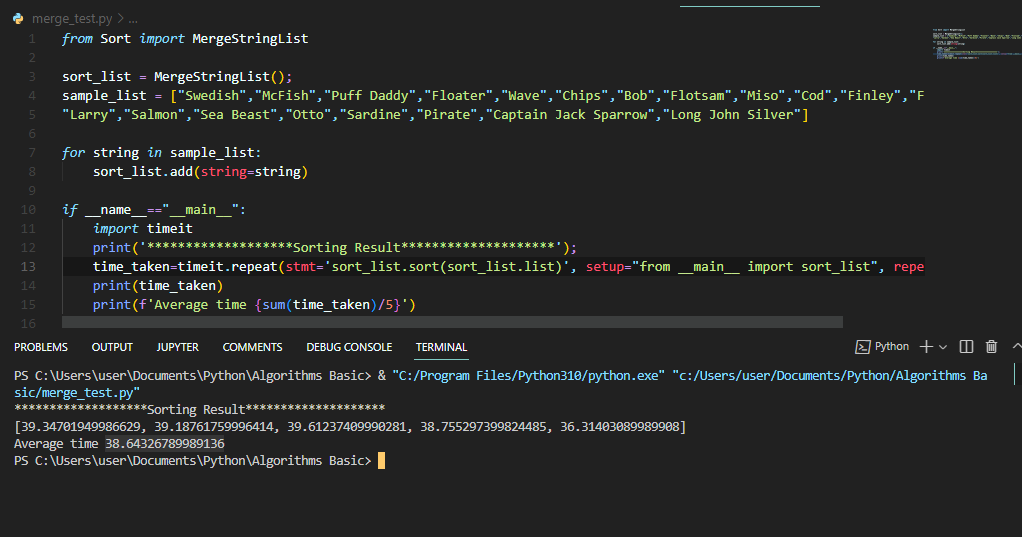
|  |  |
| --- | --- |
| **Runs** | **Execution Time** |
|  | 40.012543299933895 |
|  | 38.39682430005632 |
|  | 42.157030700007454 |
|  | 41.77343209995888 |
|  | 41.37923590000719 |
| **Average** | **40.74381325999275** |



### **MERGE SORT**

**TEST RUN RESULTS**

|  |  |
| --- | --- |
| **Runs** | **Execution Time** |
|  | 39.34701949986629 |
|  | 39.18761759996414 |
|  | 39.61237409990281 |
|  | 38.755297399824485 |
|  | 36.31403089989908 |
| **Average** | **38.64326789989136** |



### **QUICK SORT**

**TEST RUN RESULTS**

|  |  |
| --- | --- |
| **Runs** | **Execution Time** |
|  | 53.06188349984586 |
|  | 51.59682149998844 |
|  | 53.35377090005204 |
|  | 49.0553822000511 |
|  | 51.810373899992555 |
| **Average** | **51.775646399986** |



### **ANALYSIS**

**BUBBLE SORT**

Being the simplest sorting algorithm, Bubble sort works by iterating the input list from first to last and compare each pair of elements and swap them if needed.

It continues with the iterations until no more swaps are needed.

The only advantage bubble sort has over the other algorithm is that it can detect whether the input list is already sorted or not.

This algorithm takes Quadratic time of) even in the best case. This makes it amongst the most inefficient sorting algorithms while compared to other implementation.

**MERGE SORT**

In Merge Sort, The input is divided in to tow parts that are solved recursively.

After solving the sub problems, they are merged by scanning the resultant sub problems.

Therefore with the assumption that T(n) is the complexity merge sort with n elements, then the recurrence is defined as below.

This makes Quicksort, amongst the algorithms tested to be the quickest in processing

**QUICK SORT**

Quick Sort is an example of divide-and-conquer technique that uses recursive calls for sorting elements.

*Divide:* The array is partitioned into non empty subarrays and .

Conquer: the two sub arrays and are sorted recursively using quick sort.

If we assume that is the complexity of Quick Sort and all the elements are also unique.

The Recurrence of depends of two partitions.

If the pivot is the smallest element then exactly) items will be in the left part and) in the right part.

Since each element has a probability of that is equal of selecting the pivot then the probability of selecting element would be .

Best Case: Each partition splits array into halves results to.

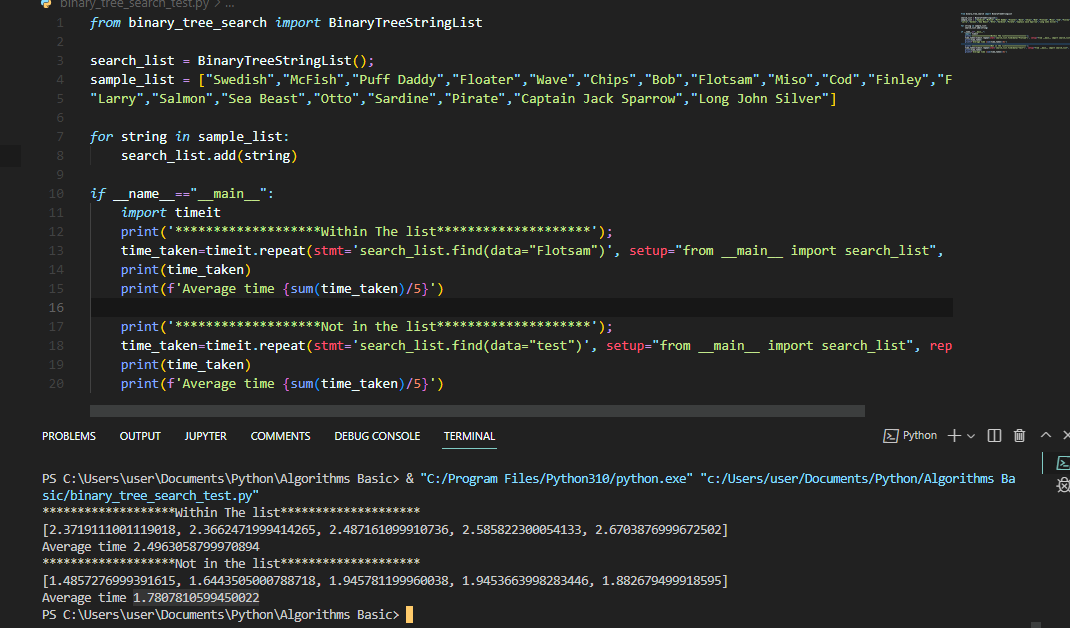
Worst Case: Each partition gives unbalanced splits results to.

According to the test runs, Quick sort is the slowest in execution time.

## **QUESTION THREE: BINARY TREE**

**TEST RUN RESULTS**

|  |  |  |
| --- | --- | --- |
| **Runs** | **With a word Not in The List Execution time** | **With a word Not in The List Execution time** |
|  | 2.3719111001119018 | 1.4857276999391615 |
|  | 2.3662471999414265 | 1.6443505000788718 |
|  | 2.487161099910736 | 1.945781199960038 |
|  | 2.585822300054133 | 1.9453663998283446 |
|  | 2.6703876999672502 | 1.882679499918595 |
| **Average** | **2.4963058799970894** | **1.7807810599450022** |



In a Binary Search Tree, To find an item, we just start from the root and keep moving left or right using the BST property.

With this we navigate the tree according the value of the input that is being searched for until we reach the node that that the input is or none if not found.

With this, the worst case time complexity is dependent on the number of nodes therefore assuming that;

## **SOURCE CODE**

### **QUESTION ONE: Search Algorithm**

**Search.py**

class SequentialStringList():

    def \_\_init\_\_(self) -> None:

*self*.list: list = []

    def add(self, string: str) -> list:

*self*.list.append(string)

*return* *self*.list

    def find(self, string: str):

*for* i *in* range(len(*self*.list)):

*if* *self*.list[i] == string:

*return* *self*.list[i]

*return* None

class BinaryStringList():

    def \_\_init\_\_(self) -> None:

*self*.list: list = []

    def add(self, string: str) -> list:

*self*.list.append(string)

*return* *self*.list

    def find(self, string: str):

        low: int = 0;

        high: int = len(*self*.list)-1

*while* low < high:

            mid = (low + high)//2

*if* *self*.list[mid] == string:

*return* string

*if* string > *self*.list[mid]:

                low = mid + 1

*else*:

                high = mid - 1

*return* None

**sequential\_test.py**

*from* Search *import* SequentialStringList

search\_list = SequentialStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    search\_list.add(string=string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Within The list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(string="Flotsam")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Not in the list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(string="test")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

**binary\_test.py**

*from* Search *import* BinaryStringList

search\_list = BinaryStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    search\_list.add(string=string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Within The list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(string="Flotsam")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Not in the list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(string="test")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

### **QUESTION TWO: Sorting Algorithms**

**Sort.py**

class BubbleStringList():

    def \_\_init\_\_(self) -> None:

*self*.list: list = []

    def add(self, string: str) -> list:

*self*.list.append(string)

*return* *self*.list

    def sort(self):

*for* i *in* range(len(*self*.list)):

*for* j *in* range(len(*self*.list) - 1, i, -1):

*if* *self*.list[j] < *self*.list[j - 1]:

*self*.swap(j, j - 1)

    def swap(self, x, y):

        temp = *self*.list[x]

*self*.list[x] = *self*.list[y]

*self*.list[y] = temp

class MergeStringList():

    def \_\_init\_\_(self) -> None:

*self*.list: list = []

    def add(self, string: str) -> list:

*self*.list.append(string)

*return* *self*.list

    def sort(self, listToSort: list) -> list:

*if* len(listToSort) > 1:

            mid = len(listToSort)//2

            left = listToSort[:mid]

            right = listToSort[mid:]

*self*.sort(left)

*self*.sort(right)

            i = j = k = 0

*while* i< len(left) and j < len(right):

*if* left[i] < right[i]:

                    listToSort[k] = left[i]

                    i += 1

*else*:

                    listToSort[k] = right[j]

                    j += 1

                k += 1

*while* i< len(left):

                listToSort[k] = left[i]

                i += 1

                k += 1

*self*.list = listToSort

class QuickStringList():

    def \_\_init\_\_(self) -> None:

*self*.list: list = []

    def add(self, string: str) -> list:

*self*.list.append(string)

*return* *self*.list

    def swap(self,listToSort: list, x: int, y: int):

        temp = listToSort[x]

        listToSort[x] = listToSort[y]

        listToSort[y] = temp

    def sort(self, listToSort: list, low: int, high: int):

*if* low < high:

            pivot = *self*.divide(listToSort, low, high)

*self*.sort(listToSort, low, pivot - 1)

*self*.sort(listToSort, pivot + 1, high)

*self*.list = listToSort

    def divide(self, listToSort: list, low, high) -> int:

        pivot = low

*self*.swap(listToSort, pivot, high)

*for* i *in* range(low, high):

*if* listToSort[i] <= listToSort[high]:

*self*.swap(listToSort, i, low)

                low += 1

*self*.swap(listToSort, low, high)

*return* low

**bubble\_test.py**

*from* Sort *import* BubbleStringList

sort\_list = BubbleStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    sort\_list.add(string=string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Sorting Result\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='sort\_list.sort()', setup="from \_\_main\_\_ import sort\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

**merge\_test.py**

*from* Sort *import* MergeStringList

sort\_list = MergeStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    sort\_list.add(string=string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Sorting Result\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='sort\_list.sort(sort\_list.list)', setup="from \_\_main\_\_ import sort\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

**quick\_test.py**

*from* Sort *import* QuickStringList

sort\_list = QuickStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    sort\_list.add(string=string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Sorting Result\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='sort\_list.sort(sort\_list.list, 0, len(sort\_list.list)-1)', setup="from \_\_main\_\_ import sort\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

### **QUESTION THREE: Binary Tree**

**Binary\_tree.py**

class Node():

    def \_\_init\_\_(self) -> None:

*self*.data = None

*self*.left = None

*self*.right = None

    def setData(self, data):

*self*.data = data

    def getData(self):

*return* *self*.data

    def setLeft(self, data):

*self*.left = data

    def getLeft(self):

*return* *self*.left

    def setRight(self, data):

*self*.right = data

    def getRight(self):

*return* *self*.right

**binary\_search\_tree.py**

*from* binary\_tree *import* Node

class BinaryTreeStringList:

    def \_\_init\_\_(self) -> None:

*self*.root: Node = None

*self*.connections: int = 0

*self*.stringList: list = []

    def find(self, data):

*'''*

*Worst case time complexity is O(n) since we can navigate the whole tree*

*'''*

        currentNode: Node = *self*.root

*while* currentNode:

*if* data == currentNode.getData():

*return* currentNode.getData()

*if* data < currentNode.getData():

                currentNode = currentNode.getLeft()

*else*:

                currentNode = currentNode.getRight()

*self*.connections += 1

*return* None

    def all(self):

        currentNode: Node = *self*.root

*self*.stringList: list = []

*self*.allNodes(*self*.root)

*return* *self*.stringList

    def allNodes(self, currentNode: Node):

*'''*

*Worst case time complexity is O(n) since we can navigate the whole tree*

*'''*

*while* currentNode:

*self*.stringList.append(currentNode.getData())

*if* not currentNode.getLeft() is None:

*return*(*self*.allNodes(currentNode.getLeft()))

*else*:

*return*(*self*.allNodes(currentNode.getRight()))

    def add(self, data):

*self*.insertNode(*self*.root, data)

    def insertNode(self, root: Node, data: str):

        node: Node = Node()

        node.setData(data)

*if* *self*.root is None:

*self*.root = node

*else*:

*if* root.getData() > node.getData():

*if* root.getLeft() is None:

                    root.setLeft(node)

*else*:

*return*(*self*.insertNode(root.left, data))

*else*:

*if* root.getRight() is None:

                    root.setRight(node)

*else*:

*return*(*self*.insertNode(root.right, data))

**binary\_search\_tree\_test.py**

*from* binary\_seach\_tree *import* BinaryTreeStringList

search\_list = BinaryTreeStringList();

sample\_list = ["Swedish","McFish","Puff Daddy","Floater","Wave","Chips","Bob","Flotsam","Miso","Cod","Finley","Finneus",

"Larry","Salmon","Sea Beast","Otto","Sardine","Pirate","Captain Jack Sparrow","Long John Silver"]

*for* string *in* sample\_list:

    search\_list.add(string)

*if* \_\_name\_\_=="\_\_main\_\_":

*import* timeit

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Within The list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(data="Flotsam")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

    print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Not in the list\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*');

    time\_taken=timeit.repeat(stmt='search\_list.find(data="test")', setup="from \_\_main\_\_ import search\_list", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')