CMPSC 462: Data Structures and Algorithms (Spring 2022)

Project-2: Developing Searching and Sorting Algorithms

Tyler Thompson
Computer Science

Instructor
Dr. Vinayak Elangovan, Ph.D.
Department of Computer Science

Submitted On: 2/21/2023

TABLE OF CONTENTS

Introduction

Create an application that can correctly and accurately calculate various sorting and searching algorithms time to complete. These algorithms are to be developed from scratch and for algorithms that complete the same task, their time will be noted and compared. Using the time from these functions we will be able to understand and fully explain the time complexity of each algorithm. These will all be performed on a list of randomly generated numbers. For the first set of trails, they will be performed on a list size of 10000 and the results will be calculated. Following this the next set of trails will be performed on a list of size 50000 and the results will once again be calculated.

The sorting algorithms include selection sort, insertion sort, bubble sort, and merge sort and these will all be developed in their own manner. In order to fully understand these concepts selection sort, bubble sort, and insertion sort will be developed iteratively. The final sorting algorithm merge sort will be developed recursively. This is to show possible ways these algorithms can be implemented. It is important to understand that each of these algorithms will be performed on the same unsorted list. A new list shall be created which is a copy of the original, this way the trails can be conducted fair and accurately.

The searching algorithms include finding if the list contains a number, finding the minimum value, finding the maximum value, and determining if the list has distinct numbers. Each of these functions were created iteratively. For finding if the list contains a number both linear search and binary search are used. Record these results and develop an understanding on how they were work.

This application is practical, so all functions were written in respective modules. This is so they may be imported anywhere else and be used in other applications where they may fit. The source code is also provided.

Algorithms

Finding Minimum and Maximum Value

This algorithm was rather simple and in the worst case takes O(n) which is the size of the list. It simply uses a for loop that iterates over the list keeping note of the lowest or highest value depending on which functions is used. It does this by doing a comparison on the current value in the list with the saved value through each iteration. If the value needs to be changed it is set equal to the value that was discovered. This is to be performed on an unsorted list.

Linear Search

This algorithm uses a for loop to iterate over an entire list. It takes in a number that is to be identified as the target value which will be the value to find within the list. As the list is iterated over the target value is compared with the current value until it is found or there is not values left in the list. Depending on what happens a Boolean value is returned. This algorithm runs in O(n) for the worst case since it has to perform over the entire list. If the value is found, then of course the time complexity becomes less depending on where the value was found in the list.

Binary Search

The algorithm used to develop binary search can easily be done recursively but for the matter of this application it was performed iteratively since the input size was so large. Additionally binary search may only be performed on a sorted list. Binary search works by simply taking a list and creating a start and end point. These points are then added up and divided by 2 which determines the middle point of the list. The target value is then compared with the middle point of the list and if it is equal then it returns true. If it is not, then the middle value is compared with the target value to determine if it is less than or greater than it. Depending on what it is, the new start or end is adjusted to be set equal to the middle since the value can only be in a certain range which is between the new start or new end. A new middle value is then calculated and is compared to the target value. This iteration repeats until the value is found or the start is greater than end which means in this case that the value is not contained within the list. This algorithm at worst runs in O(logn) simply because it reduces the input size for each iteration.

Distinct Numbers

The algorithm pertaining to distinct numbers seems complex but is rather easy to implement. This algorithm works iteratively. This simply takes in a list and creates two new temporary empty lists. The list that was passed to the function is then iterated through and with each iteration the current value is appended to a temporary list. A Boolean value is also created and set to true in the beginning of the algorithm. For each value that is appended to the temporary list from the input list, it is checked to see if the temporary list contains the value that is being input. If at all within this algorithm it is determined that the temporary list does contain a value that is being input then the Boolean value is set to false and that value is appended to the other temporary list that was created. By the time the list has been iterated through the Boolean value will determine if the list contained any duplicates and if it did, it will return a list of the duplicates that were found. This algorithm runs in O(n^2) for the worst case since for each current value it is compared to another list that must be checked to determine if it contains it.

Selection Sort

The selection sort algorithm was developed iteratively here and has a time complexity of O(n^2) in all cases. A list is passed and is iterated over for the length of the list. With each iteration the first element at that point is determined and set as the lowest value. This element is then compared with every element in the list and if an element is found while iterating through then the discovered element takes the place as the lowest value. At the end of each iteration the lowest value is then swapped with the current iterations starting point. This process repeats and at the end the list will be sorted. It is important to note that even if the list that is passed is sorted, the same amount of operations will be performed.

Insertion Sort

The insertion sort algorithm was developed iteratively in the case of this application and runs with a time complexity of O(n^2) for the worst case. Insertion sort takes a list and iterates over it comparing each current element with the element in front of it. This is denoted as i + 1 in this case since if the value in front is not less than the current value then the current value is set to i + 1 for the next iteration. While traversing the list if any value is determined to be out of place meaning that the current value is greater than the next value then multiple operations are conducted. These include taking the out of place element which is i + 1 and comparing it with each element before it. A spot will be determined where this element belongs once the condition is not satisfied which in this case means that i + 1 is greater than the value that is being compared. In this case the element at i + 1 is then saved to a temporary value and is deleted. This temporary value is then inserted before the spot that the condition was no longer satisfied through the iteration. If a value is not found that means it is the smallest element and must be inserted to the front of the list. It is important to not that this algorithm can perform much better if the list is partially sorted since less operations will need to be performed.

Bubble Sort

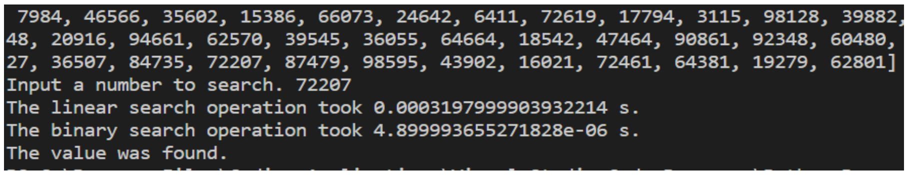
The bubble sort algorithm in this application was developed iteratively and has a time complexity of O(n^2) in the worst case. It takes a list and iterates over the list comparing each element with the element in front of it. A Boolean value is created and set to true at the start of each iteration. If the current value is less than the next element than the Boolean value is set to false meaning that it is not sorted. When this happens the current value and the element in front of it swap places. The same element remains as the current element and is compared with the next value in the list. This keeps happening until the end of the list is reached and once it is complete the last value in the list will be the largest value. Each iteration has one value less to compare since the next value that is placed will end up being in the correct spot. The iteration continues until the Boolean value is true which in this case means the list is sorted.

Merge Sort

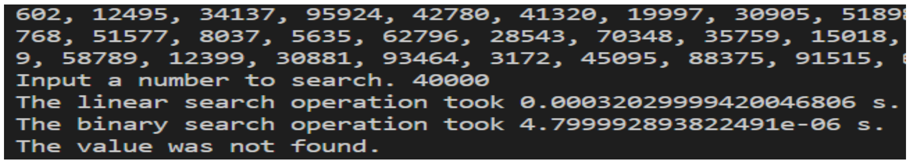
The merge sort algorithm in this application was developed recursively and has a time complexity of O(nlogn) in all cases. This algorithm takes a list and starts out by checking to see if the list size is 1 which would mean it is sorted. If it is not sorted, then a middle point is determined by dividing the list by 2 which is used to create new lists which are sliced from the start to the middle point and from the middle to the end point. This is then repeated until the list size is 1 which once again means it is sorted. When this is the case another algorithm is then used to reconstruct the lists by comparing each created sets of slices and creating a new list from them. This is done by checking that each list is not empty and for each comparison the start index of both list is compared and the lower value is then appended to a new temporary list. The value that was appended from the corresponding list is then deleted from that list. This continues until one of the lists are empty which means that the remaining values in the other list can be appended to the new temporary list since the values would be sorted. This algorithm goes until there are no more lists to piece together which means that the list is sorted. It is important to note that in all cases this algorithm will always have the same time complexity since the function takes an input size and reduces it by half each time it is called. This allows it to run efficiently with large input sizes.

Sample Output

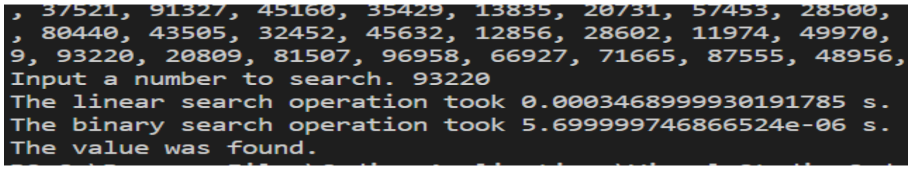
Searching with an input size of 10000 elements



```
 7984, 46566, 35602, 15386, 66073, 24642, 6411, 72619, 17794, 3115, 98128, 39882,
48, 20916, 94661, 62570, 39545, 36055, 64664, 18542, 47464, 90861, 92348, 60480,
27, 36507, 84735, 72207, 87479, 98595, 43902, 16021, 72461, 64381, 19279, 62801]
Input a number to search. 72207
The linear search operation took 0.0003197999903932214 s.
The binary search operation took 4.899993655271828e-06 s.
The value was found.
```

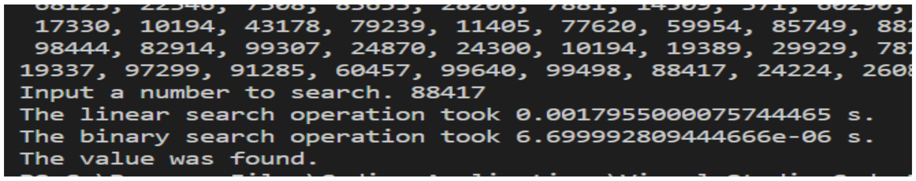


```
 602, 12495, 34137, 95924, 42780, 41320, 19997, 30905, 5189
 768, 51577, 8037, 5635, 62796, 28543, 70348, 35759, 15018,
 9, 58789, 12399, 30881, 93464, 3172, 45095, 88375, 91515,
 Input a number to search. 40000
 The linear search operation took 0.00032029999420046806 s.
 The binary search operation took 4.799992893822491e-06 s.
 The value was not found.
```

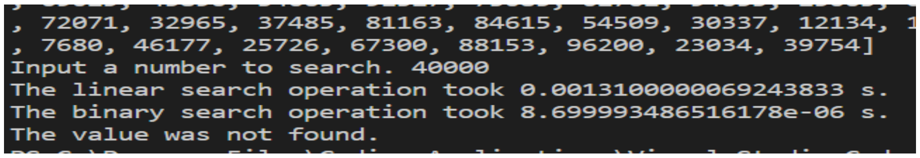


```
, 37521, 91327, 45160, 35429, 13835, 20731, 57453, 28500,
, 80440, 43505, 32452, 45632, 12856, 28602, 11974, 49970,
9, 93220, 20809, 81507, 96958, 66927, 71665, 87555, 48956,
Input a number to search. 93220
The linear search operation took 0.0003468999930191785 s.
The binary search operation took 5.699999746866524e-06 s.
The value was found.
```

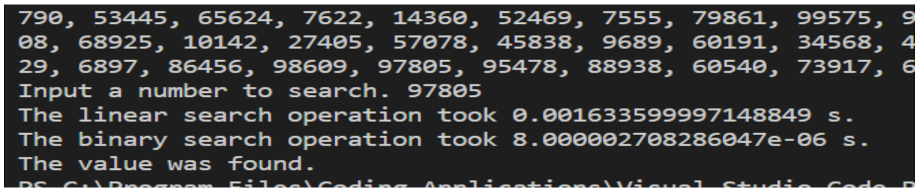
Searching with an input size of 50000 elements



```
  17330, 10194, 43178, 79239, 11405, 77620, 59954, 85749, 88
  98444, 82914, 99307, 24870, 24300, 10194, 19389, 29929, 78
 19337, 97299, 91285, 60457, 99640, 99498, 88417, 24224, 260
 Input a number to search. 88417
 The linear search operation took 0.0017955000075744465 s.
 The binary search operation took 6.699992809444666e-06 s.
 The value was found.
```

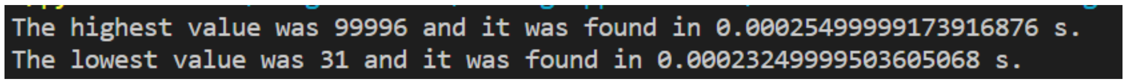


```
, 72071, 32965, 37485, 81163, 84615, 54509, 30337, 12134, 1
, 7680, 46177, 25726, 67300, 88153, 96200, 23034, 39754]
Input a number to search. 40000
The linear search operation took 0.0013100000069243833 s.
The binary search operation took 8.699993486516178e-06 s.
The value was not found.
```

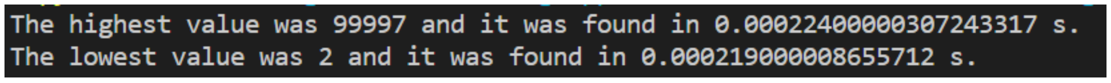


```
 790, 53445, 65624, 7622, 14360, 52469, 7555, 79861, 99575, 9
 08, 68925, 10142, 27405, 57078, 45838, 9689, 60191, 34568, 4
 29, 6897, 86456, 98609, 97805, 95478, 88938, 60540, 73917, 6
 Input a number to search. 97805
 The linear search operation took 0.00163359997148849 s.
 The binary search operation took 8.000002708286047e-06 s.
 The value was found.
 PS C:\Program Files\Coding Applications\Visual Studio Code P
```

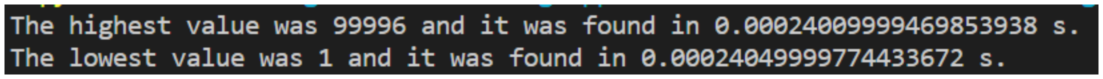
Finding minimum and maximum with 10000 elements



```
The highest value was 99996 and it was found in 0.00025499999173916876 s.
The lowest value was 31 and it was found in 0.00023249999503605068 s.
```

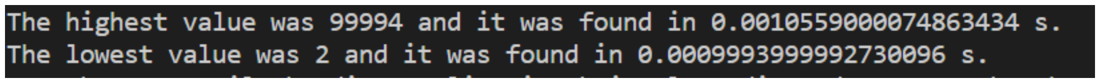


```
The highest value was 99997 and it was found in 0.00022400000307243317 s.
The lowest value was 2 and it was found in 0.000219000008655712 s.
```

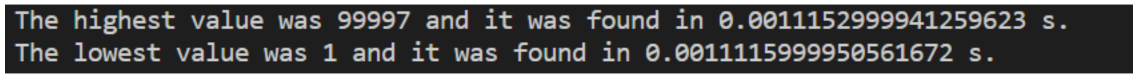


```
The highest value was 99996 and it was found in 0.00024009999469853938 s.
The lowest value was 1 and it was found in 0.00024049999774433672 s.
```

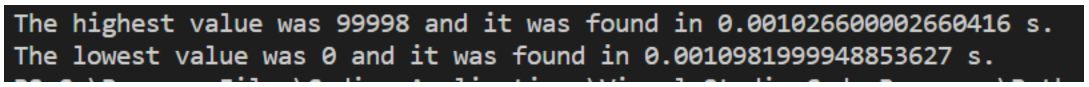
Finding minimum and maximum with 50000 elements



```
The highest value was 99994 and it was found in 0.0010559000074863434 s.
The lowest value was 2 and it was found in 0.0009993999992730096 s.
```

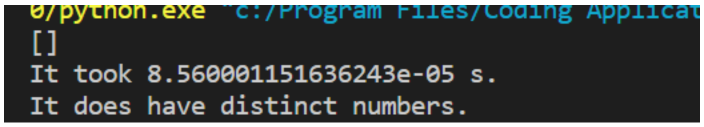


```
The highest value was 99997 and it was found in 0.0011152999941259623 s.
The lowest value was 1 and it was found in 0.0011115999950561672 s.
```

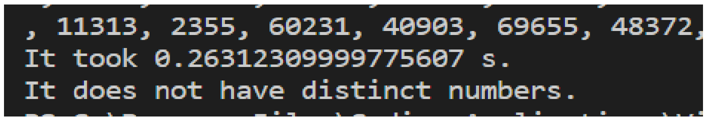


```
The highest value was 99998 and it was found in 0.001026600002660416 s.
The lowest value was 0 and it was found in 0.0010981999948853627 s.
```

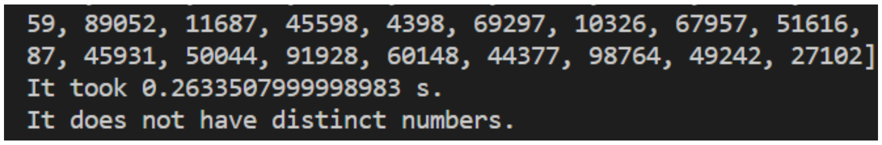
Determine distinct numbers with 10000 elements



```
0/python.exe "c:/Program Files/Coding Applicat
[]
It took 8.560001151636243e-05 s.
It does have distinct numbers.
```

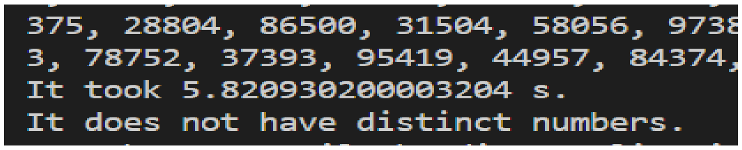


```
, 11313, 2355, 60231, 40903, 69655, 48372,
It took 0.26312309999775607 s.
It does not have distinct numbers.
```

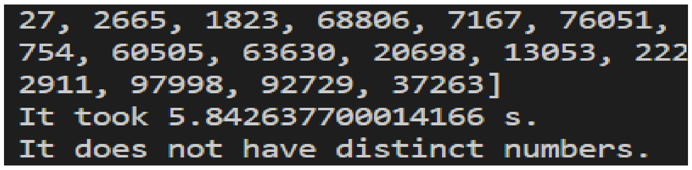


```
59, 89052, 11687, 45598, 4398, 69297, 10326, 67957, 51616,
87, 45931, 50044, 91928, 60148, 44377, 98764, 49242, 27102]
It took 0.2633507999998983 s.
It does not have distinct numbers.
```

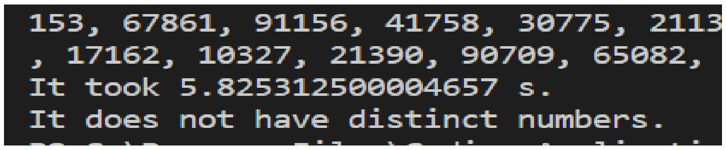
Determine distinct numbers with 50000 elements



```
375, 28804, 86500, 31504, 58056, 9738
3, 78752, 37393, 95419, 44957, 84374,
It took 5.820930200003204 s.
It does not have distinct numbers.
```

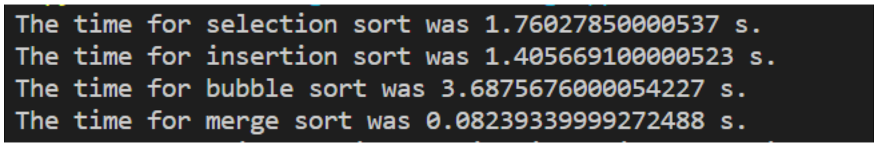


```
27, 2665, 1823, 68806, 7167, 76051,
754, 60505, 63630, 20698, 13053, 222
2911, 97998, 92729, 37263]
It took 5.842637700014166 s.
It does not have distinct numbers.
```

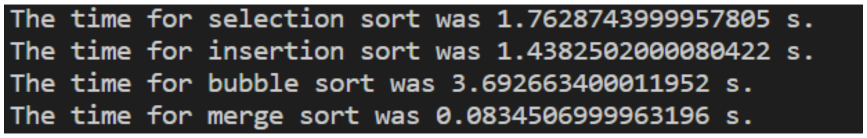


```
153, 67861, 91156, 41758, 30775, 2113
, 17162, 10327, 21390, 90709, 65082,
It took 5.825312500004657 s.
It does not have distinct numbers.
```

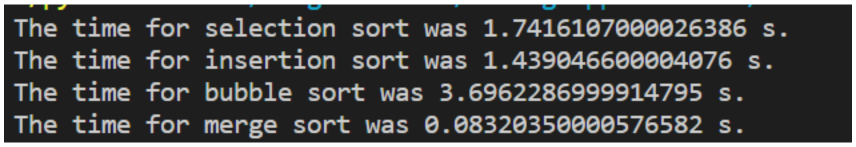
Sorting Operations with 10000 elements



```
The time for selection sort was 1.76027850000537 s.
The time for insertion sort was 1.405669100000523 s.
The time for bubble sort was 3.6875676000054227 s.
The time for merge sort was 0.08239339999272488 s.
```

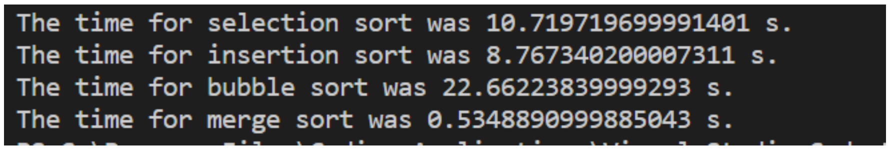


```
The time for selection sort was 1.7628743999957805 s.
The time for insertion sort was 1.4382502000080422 s.
The time for bubble sort was 3.692663400011952 s.
The time for merge sort was 0.0834506999963196 s.
```

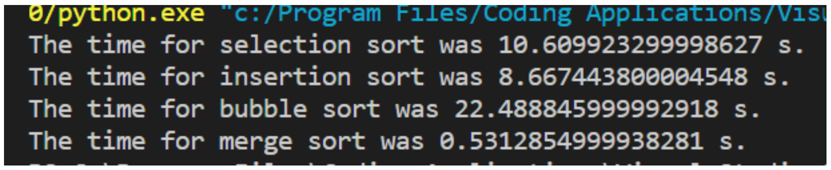


```
The time for selection sort was 1.7416107000026386 s.
The time for insertion sort was 1.439046600004076 s.
The time for bubble sort was 3.6962286999914795 s.
The time for merge sort was 0.08320350000576582 s.
```

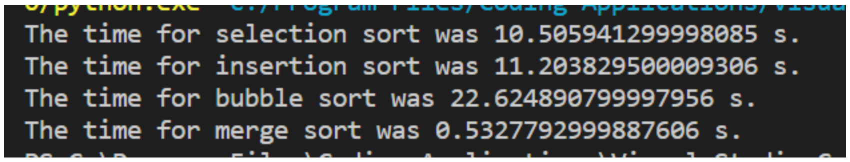
Sorting Operations with 50000 elements



```
The time for selection sort was 10.719719699991401 s.
The time for insertion sort was 8.767340200007311 s.
The time for bubble sort was 22.66223839999293 s.
The time for merge sort was 0.5348890999885043 s.
```



```
0/python.exe "c:/Program Files/Coding Applications/Visu
 The time for selection sort was 10.609923299998627 s.
 The time for insertion sort was 8.667443800004548 s.
 The time for bubble sort was 22.488845999992918 s.
 The time for merge sort was 0.5312854999938281 s.
```



```
The time for selection sort was 10.505941299998085 s.
The time for insertion sort was 11.203829500009306 s.
The time for bubble sort was 22.624890799997956 s.
The time for merge sort was 0.5327792999887606 s.
```

Time Complexity Analysis

Taking each output and understanding the time complexity is straightforward when it comes to this application. For the trails involving 10000 elements we can see that for the most part they all follow a particular trend. This confirms that the results and statements made about the algorithms were indeed true. For example we can see that for each case where the algorithms time complexity is O(n) in the worst case that each trial performed about the same. This is noted well in finding a value with a linear search and also finding the minimum and maximum value. In the case of functions that had a more complex time complexity we can see that the statements also held. Binary search performed extremely well and showed that if the list is sorted it is superior in almost every way. When it came to the sorting we can see that merge sort always came out as the winner and for the most part it was not close. The results were definitely a lot closer when it came to an input size of 10000 elements.

Taking the data and increasing the input size to 50000 only further proved the statements. For the most part searching for a value as well as a minimum and maximum did increase but it was only slightly. This does prove that the input size does increase the run time of the application. Binary search once again showed that when it comes to a sorted list it will almost always out perform the competition. Where we can really see and understand time complexity comes in the form of finding duplicates and searching. All of these algorithms once again have a more complex runtime and when increasing the input size it showed that it was not just a slight increase but rather an exponential runtime increase. Understanding this is important when designing these algorithms and of course there are slight advantages and disadvantages if they are performed recursively or iteratively. The time complexity of each algorithm showed a steady trend that further proved what has been clarified.

Conclusion

The most important part of this application is the knowledge gained from understanding how time complexity concerns itself with input size. Through each trial a trend was clearly developed and shown when it came to the time complexity of each algorithm which can also be seen by simply examining how the algorithm solves the problem. As a programmer it is tempting to jump to the easy solution when solving a problem but what separates great programmers is that they understand how to optimize and efficiently run their code. This of course comes from the understanding of time complexity. When viewing a problem, a programmer must first analyze and develop an algorithm that solves the intended problem all while taking into consideration the time complexity of the algorithm. Only when the most efficient algorithm has been developed is it necessary to begin coding. The calculations involved in time complexity come from understanding how the input size effects run time of the application.

In the case of all the trails that were conducted determining the correct algorithm does not seem to be complicated since a programmer might always want to conclude that which ever algorithm runs the fastest is always the best. This of course is not true, and each algorithm has its own use case but considering time is important in the programming world, time complexity must be fully understood and utilized correctly.

References

Understanding the algorithms

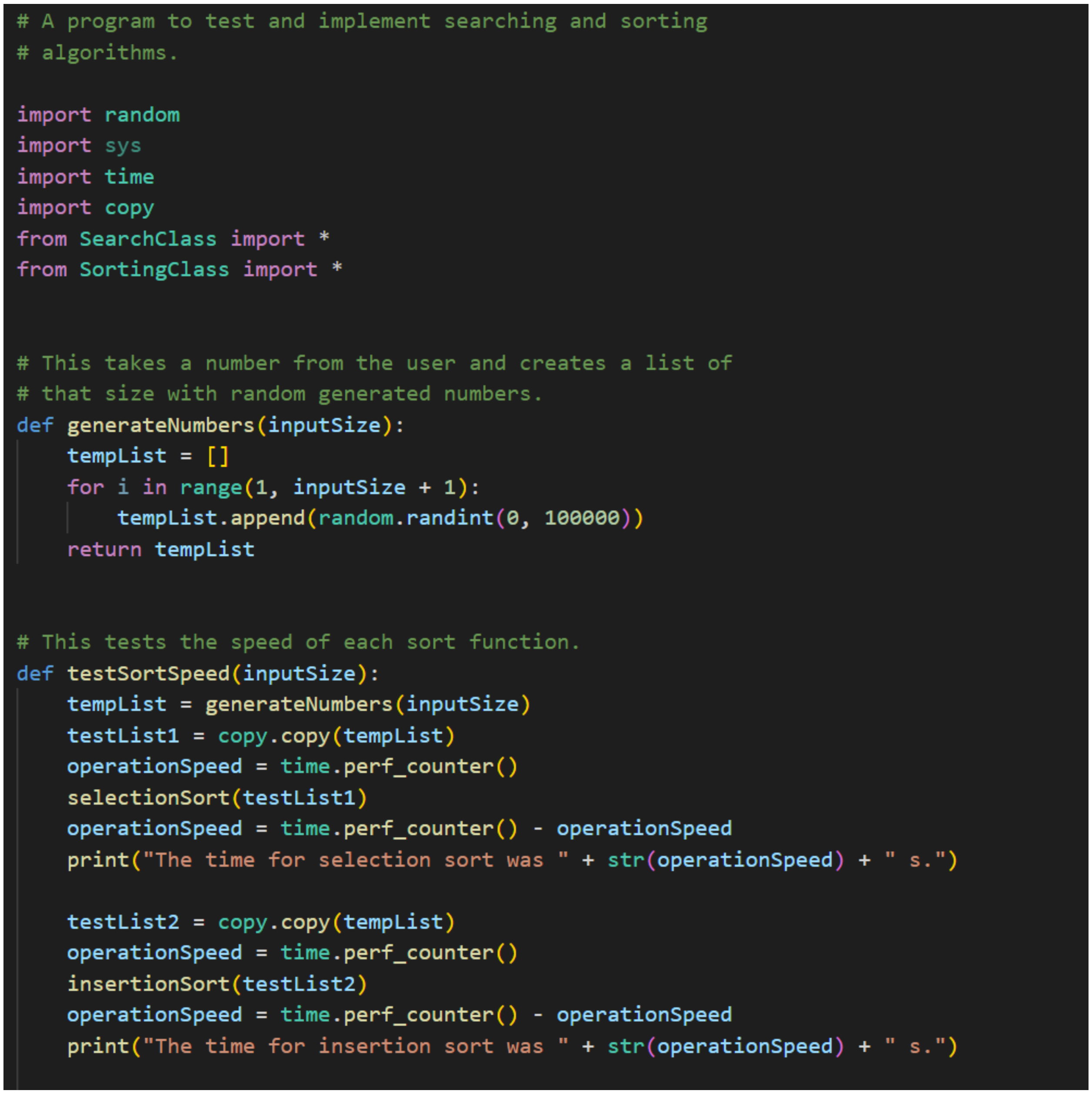
Problem Solving with Algorithms and Data Structures using Python — Problem Solving with Algorithms and Data Structures (runestone.academy)

Python Documentation

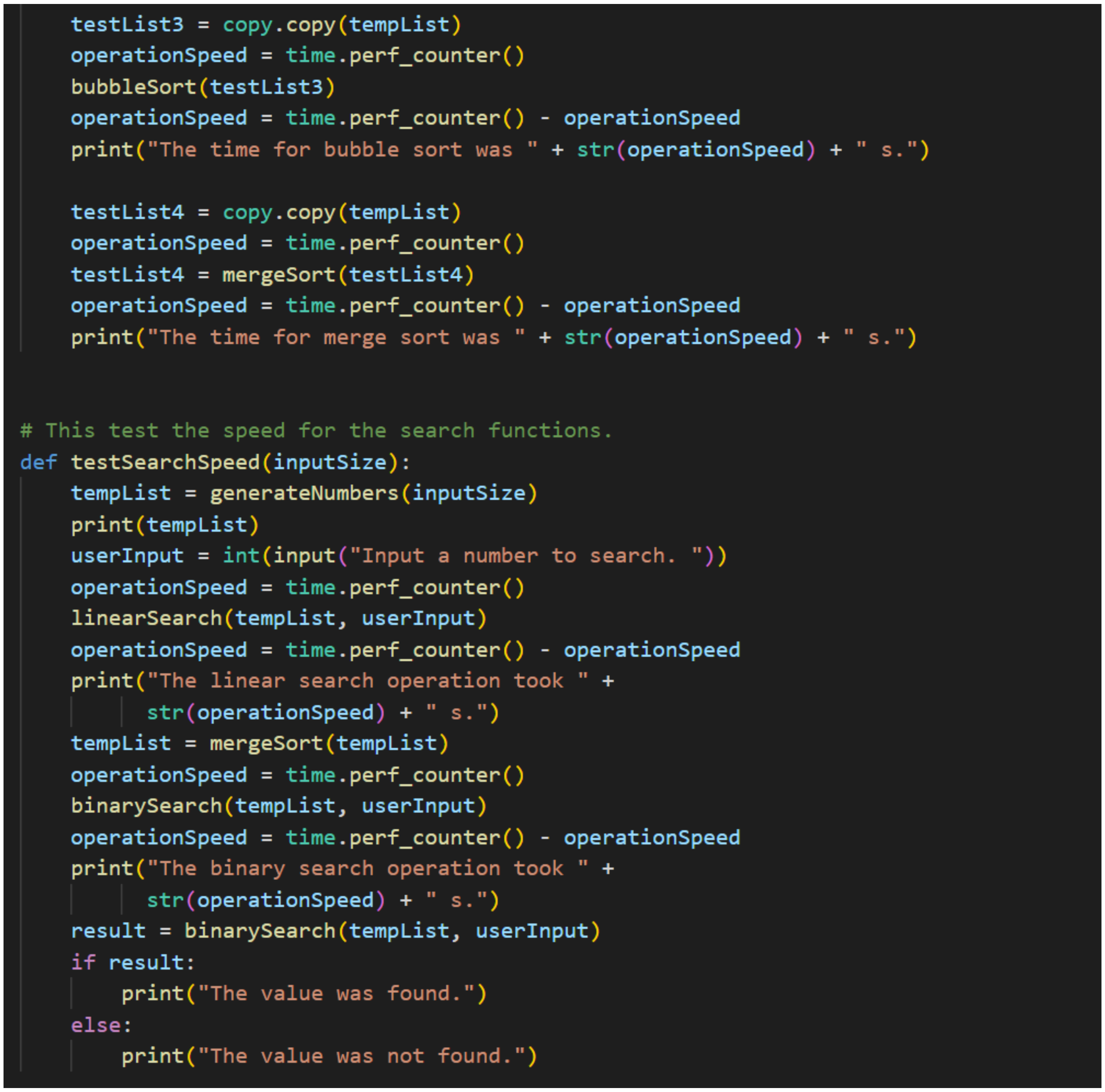
3.11.2 Documentation (python.org)

Code

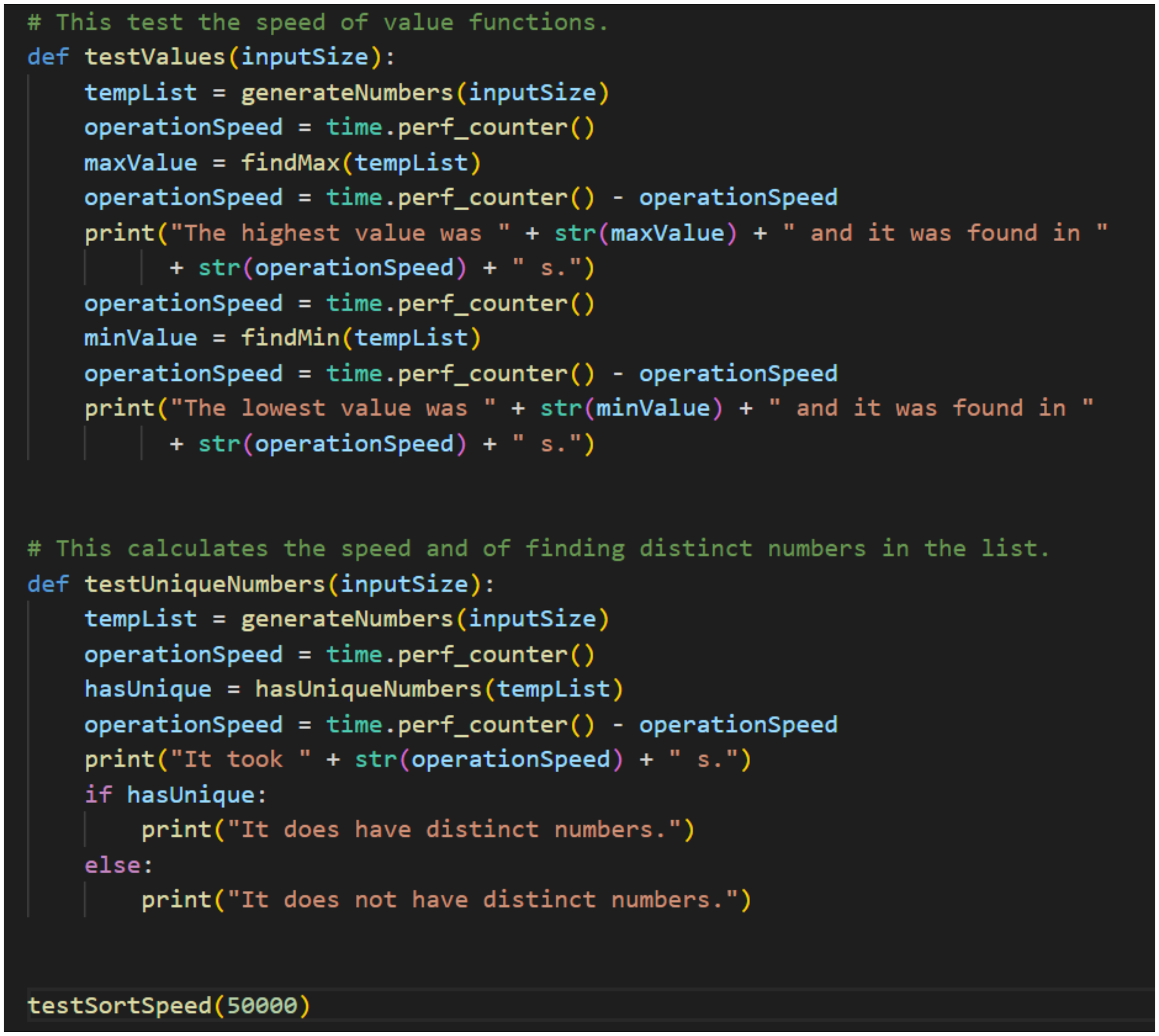
DataResults.py



```python
# A program to test and implement searching and sorting
# algorithms.

import random
import sys
import time
import copy
from SearchClass import *
from SortingClass import *


# This takes a number from the user and creates a list of
# that size with random generated numbers.
def generateNumbers(inputSize):
    tempList = []
    for i in range(1, inputSize + 1):
        tempList.append(random.randint(0, 100000))
    return tempList


# This tests the speed of each sort function.
def testSortSpeed(inputSize):
    tempList = generateNumbers(inputSize)
    testList1 = copy.copy(tempList)
    operationSpeed = time.perf_counter()
    selectionSort(testList1)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The time for selection sort was " + str(operationSpeed) + " s.")

    testList2 = copy.copy(tempList)
    operationSpeed = time.perf_counter()
    insertionSort(testList2)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The time for insertion sort was " + str(operationSpeed) + " s.")
```

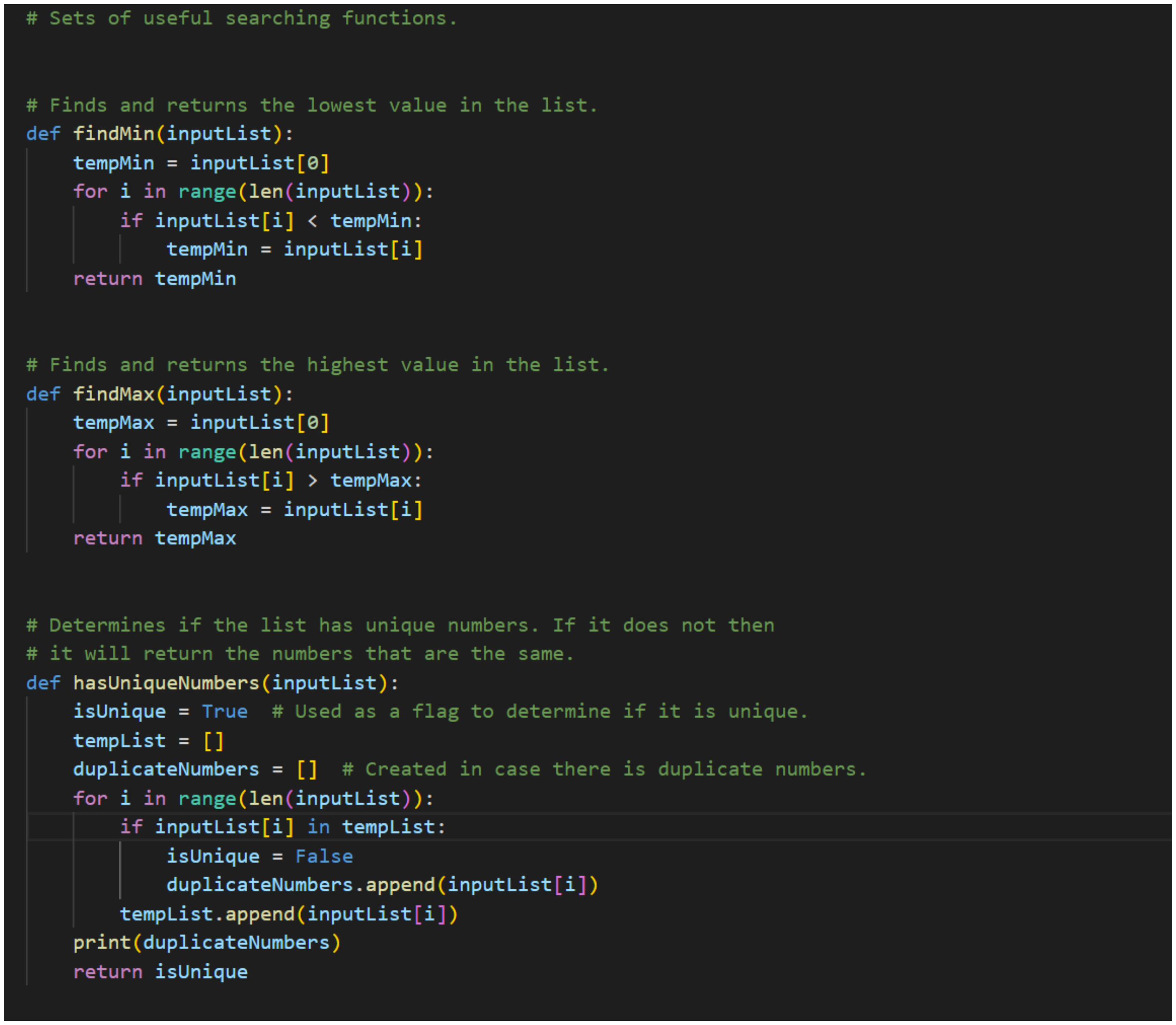


```python
    testList3 = copy.copy(tempList)
    operationSpeed = time.perf_counter()
    bubbleSort(testList3)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The time for bubble sort was " + str(operationSpeed) + " s.")

    testList4 = copy.copy(tempList)
    operationSpeed = time.perf_counter()
    testList4 = mergeSort(testList4)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The time for merge sort was " + str(operationSpeed) + " s.")


# This test the speed for the search functions.
def testSearchSpeed(inputSize):
    tempList = generateNumbers(inputSize)
    print(tempList)
    userInput = int(input("Input a number to search. "))
    operationSpeed = time.perf_counter()
    linearSearch(tempList, userInput)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The linear search operation took " +
            str(operationSpeed) + " s.")
    tempList = mergeSort(tempList)
    operationSpeed = time.perf_counter()
    binarySearch(tempList, userInput)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The binary search operation took " +
            str(operationSpeed) + " s.")
    result = binarySearch(tempList, userInput)
    if result:
        print("The value was found.")
    else:
        print("The value was not found.")
```

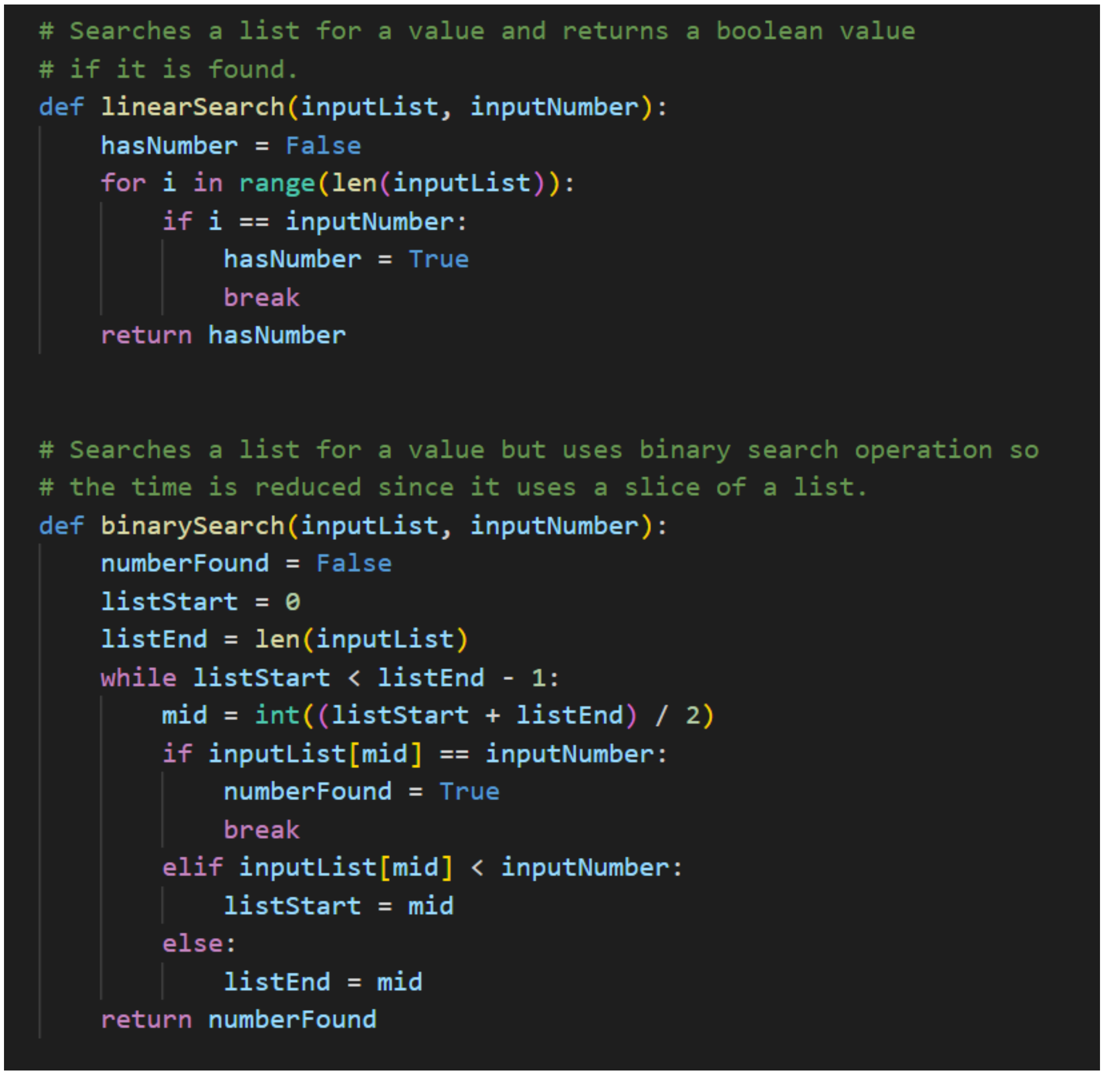


```python
# This test the speed of value functions.
def testValues(inputSize):
    tempList = generateNumbers(inputSize)
    operationSpeed = time.perf_counter()
    maxValue = findMax(tempList)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The highest value was " + str(maxValue) + " and it was found in "
          + str(operationSpeed) + " s.")
    operationSpeed = time.perf_counter()
    minValue = findMin(tempList)
    operationSpeed = time.perf_counter() - operationSpeed
    print("The lowest value was " + str(minValue) + " and it was found in "
          + str(operationSpeed) + " s.")


# This calculates the speed and of finding distinct numbers in the list.
def testUniqueNumbers(inputSize):
    tempList = generateNumbers(inputSize)
    operationSpeed = time.perf_counter()
    hasUnique = hasUniqueNumbers(tempList)
    operationSpeed = time.perf_counter() - operationSpeed
    print("It took " + str(operationSpeed) + " s.")
    if hasUnique:
        print("It does have distinct numbers.")
    else:
        print("It does not have distinct numbers.")


testSortSpeed(50000)
```

SearchClass.py

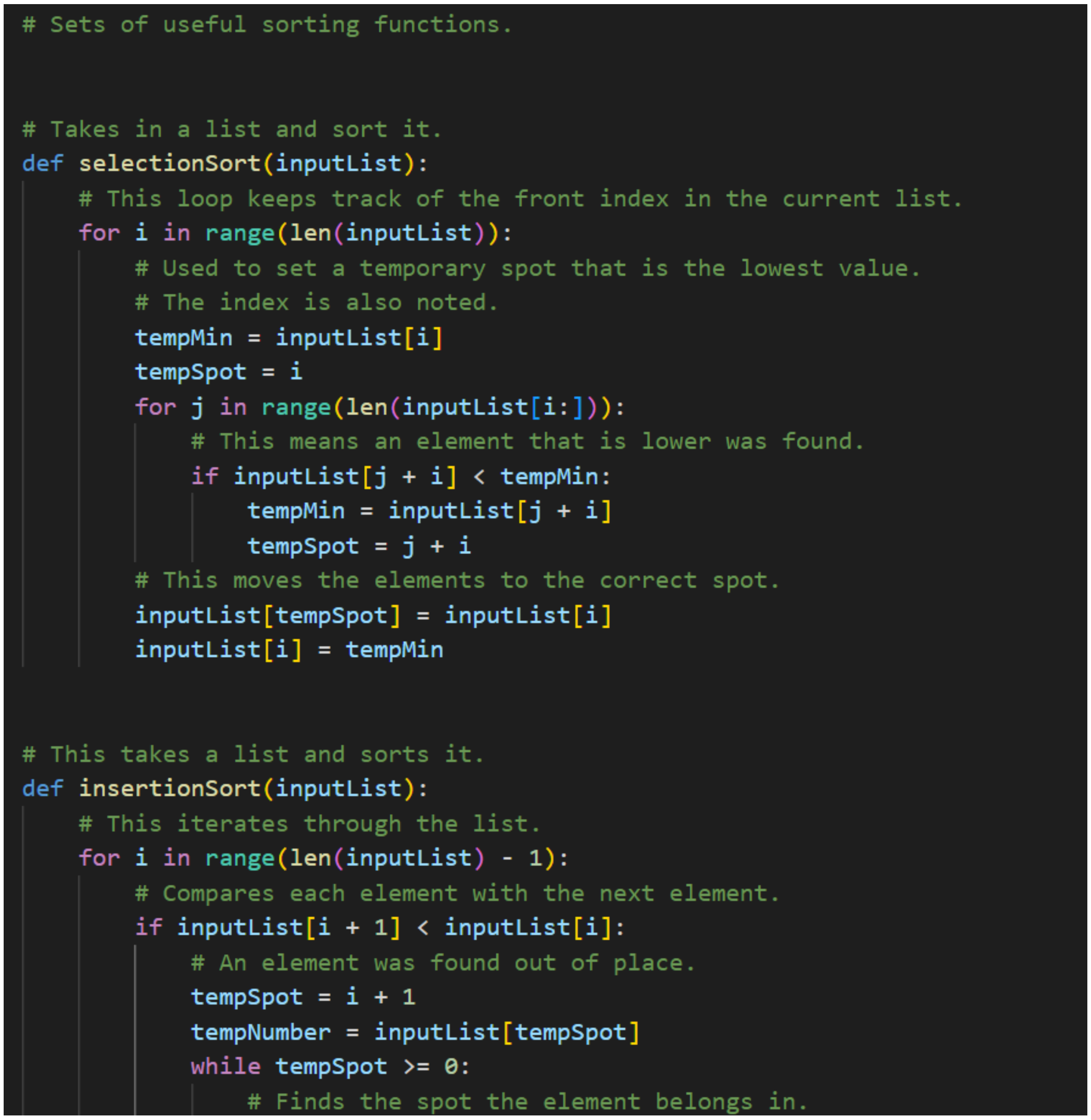


```python
# Sets of useful searching functions.


# Finds and returns the lowest value in the list.
def findMin(inputList):
    tempMin = inputList[0]
    for i in range(len(inputList)):
        if inputList[i] < tempMin:
            tempMin = inputList[i]
    return tempMin


# Finds and returns the highest value in the list.
def findMax(inputList):
    tempMax = inputList[0]
    for i in range(len(inputList)):
        if inputList[i] > tempMax:
            tempMax = inputList[i]
    return tempMax


# Determines if the list has unique numbers. If it does not then
# it will return the numbers that are the same.
def hasUniqueNumbers(inputList):
    isUnique = True  # Used as a flag to determine if it is unique.
    tempList = []
    duplicateNumbers = []  # Created in case there is duplicate numbers.
    for i in range(len(inputList)):
        if inputList[i] in tempList:
            isUnique = False
            duplicateNumbers.append(inputList[i])
        tempList.append(inputList[i])
    print(duplicateNumbers)
    return isUnique
```

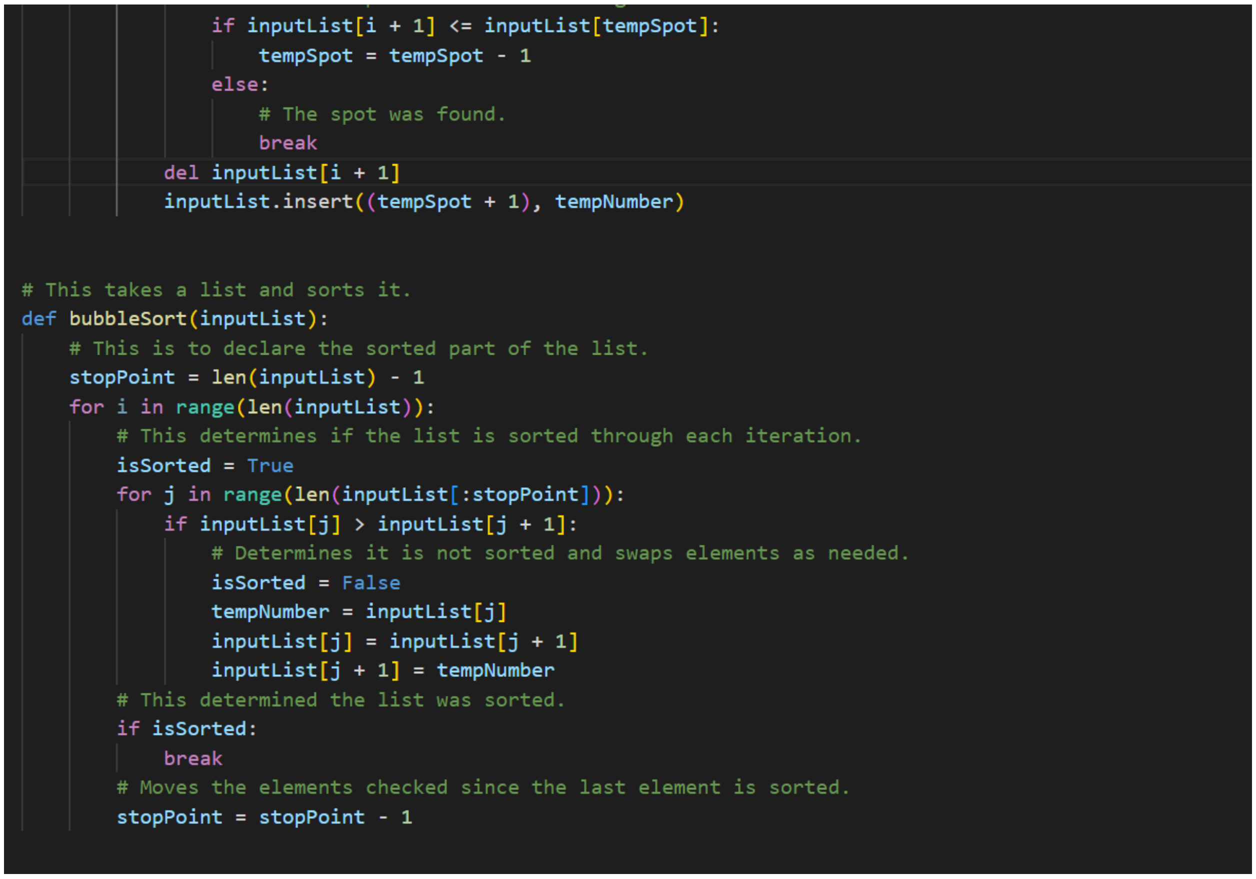


```python
# Searches a list for a value and returns a boolean value
# if it is found.
def linearSearch(inputList, inputNumber):
    hasNumber = False
    for i in range(len(inputList)):
        if i == inputNumber:
            hasNumber = True
            break
    return hasNumber


# Searches a list for a value but uses binary search operation so
# the time is reduced since it uses a slice of a list.
def binarySearch(inputList, inputNumber):
    numberFound = False
    listStart = 0
    listEnd = len(inputList)
    while listStart < listEnd - 1:
        mid = int((listStart + listEnd) / 2)
        if inputList[mid] == inputNumber:
            numberFound = True
            break
        elif inputList[mid] < inputNumber:
            listStart = mid
        else:
            listEnd = mid
    return numberFound
```

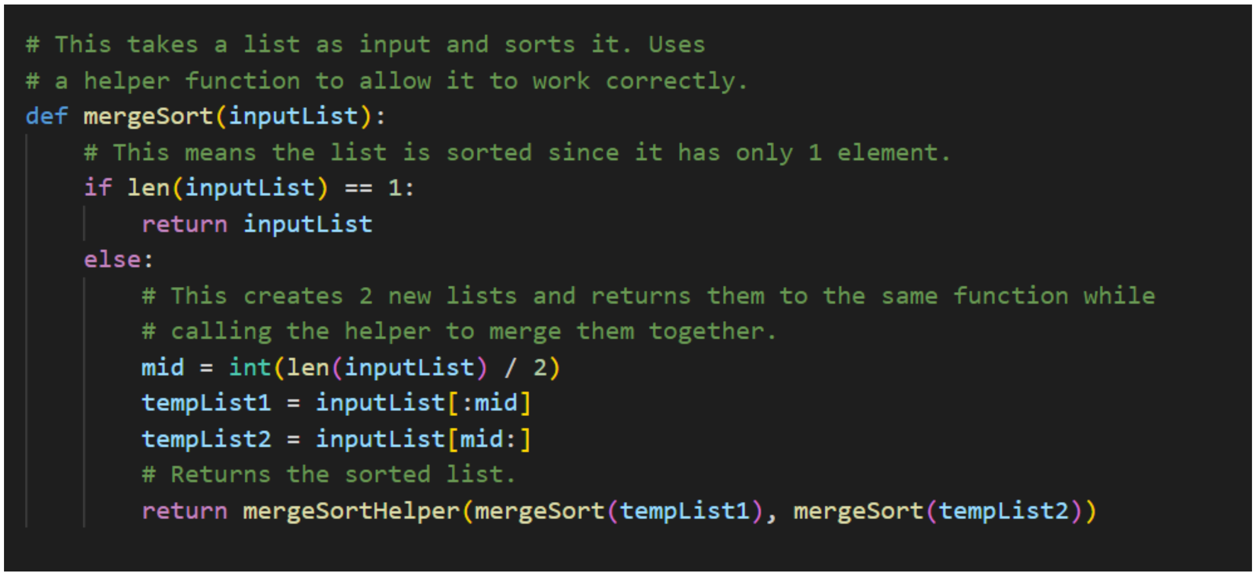
SortingClass.py



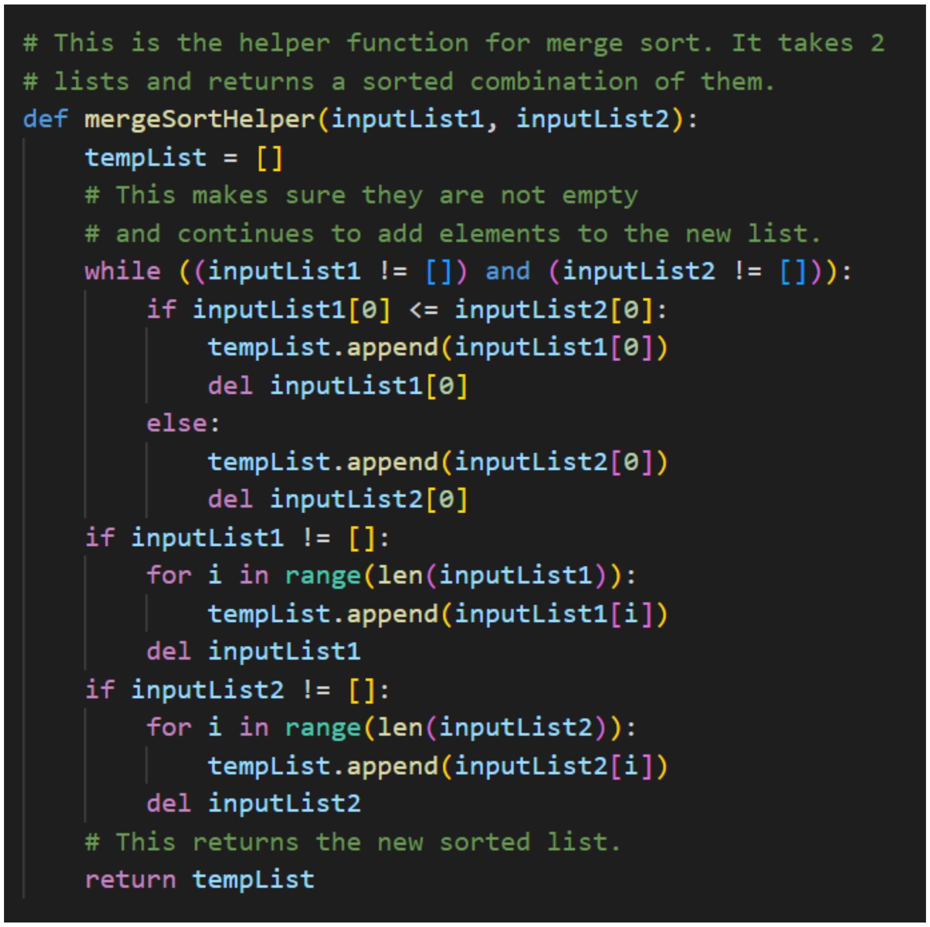
```python
# Sets of useful sorting functions.


# Takes in a list and sort it.
def selectionSort(inputList):
    # This loop keeps track of the front index in the current list.
    for i in range(len(inputList)):
        # Used to set a temporary spot that is the lowest value.
        # The index is also noted.
        tempMin = inputList[i]
        tempSpot = i
        for j in range(len(inputList[i:])):
            # This means an element that is lower was found.
            if inputList[j + i] < tempMin:
                tempMin = inputList[j + i]
                tempSpot = j + i
        # This moves the elements to the correct spot.
        inputList[tempSpot] = inputList[i]
        inputList[i] = tempMin


# This takes a list and sorts it.
def insertionSort(inputList):
    # This iterates through the list.
    for i in range(len(inputList) - 1):
        # Compares each element with the next element.
        if inputList[i + 1] < inputList[i]:
            # An element was found out of place.
            tempSpot = i + 1
            tempNumber = inputList[tempSpot]
            while tempSpot >= 0:
                # Finds the spot the element belongs in.
```



```
            if inputList[i + 1] <= inputList[tempSpot]:
                tempSpot = tempSpot - 1
            else:
                # The spot was found.
                break
        del inputList[i + 1]
        inputList.insert((tempSpot + 1), tempNumber)


# This takes a list and sorts it.
def bubbleSort(inputList):
    # This is to declare the sorted part of the list.
    stopPoint = len(inputList) - 1
    for i in range(len(inputList)):
        # This determines if the list is sorted through each iteration.
        isSorted = True
        for j in range(len(inputList[:stopPoint])):
            if inputList[j] > inputList[j + 1]:
                # Determines it is not sorted and swaps elements as needed.
                isSorted = False
                tempNumber = inputList[j]
                inputList[j] = inputList[j + 1]
                inputList[j + 1] = tempNumber
        # This determined the list was sorted.
        if isSorted:
            break
        # Moves the elements checked since the last element is sorted.
        stopPoint = stopPoint - 1
```



```python
# This takes a list as input and sorts it. Uses
# a helper function to allow it to work correctly.
def mergeSort(inputList):
    # This means the list is sorted since it has only 1 element.
    if len(inputList) == 1:
        return inputList
    else:
        # This creates 2 new lists and returns them to the same function while
        # calling the helper to merge them together.
        mid = int(len(inputList) / 2)
        tempList1 = inputList[:mid]
        tempList2 = inputList[mid:]
        # Returns the sorted list.
        return mergeSortHelper(mergeSort(tempList1), mergeSort(tempList2))
```



```python
# This is the helper function for merge sort. It takes 2
# lists and returns a sorted combination of them.
def mergeSortHelper(inputList1, inputList2):
    tempList = []
    # This makes sure they are not empty
    # and continues to add elements to the new list.
    while ((inputList1 != []) and (inputList2 != [])):
        if inputList1[0] <= inputList2[0]:
            tempList.append(inputList1[0])
            del inputList1[0]
        else:
            tempList.append(inputList2[0])
            del inputList2[0]
    if inputList1 != []:
        for i in range(len(inputList1)):
            tempList.append(inputList1[i])
        del inputList1
    if inputList2 != []:
        for i in range(len(inputList2)):
            tempList.append(inputList2[i])
        del inputList2
    # This returns the new sorted list.
    return tempList
```