Implementation and Analysis of Search Algorithms with User Input

Denise Becerra | Joshua Cabal | Venkat Amit Kommineni

Krish Viswanadhan Nair | Namrata Patil

California State University Northridge

BANA622 Programming for Business Analytics

Dr. Pouyan Eslami

May 9th, 2024

Table of Contents

1.	Introduction
2.	Methodology
3.	Results
4.	Discussion
5.	Conclusion
6.	Appendices

Implementation and Analysis of Search Algorithms with User Input

Introduction

According to Panos Louridas, author of *Algorithms,* Algorithms are about doing something in a specific way, following some kind of steps or sequence and these steps may describe a selection that determines which steps to follow. These steps can be put into a loop or iteration, where they are executed repeatedly to achieve a certain result (Louridas). To be considered a good algorithm, programmers must consider the time complexity, CPU utilization, scalability, and efficiency of the algorithm's design. Additionally an algorithm must significantly reduce the operational costs associated with its design and implementation. Time complexity Algorithms are deemed important due to their problem solving capabilities, automation, optimization, and innovation. This report will detail the development of python programs that implement recursive and non-recursive versions of linear and binary search algorithms that will allow for a user input for a list and the item the algorithm will be searching for. Furthermore, the performances of these two different approaches to the algorithm's design will be compared against one another across multiple dimensions to gain an understanding of which is the superior option.

Methodology

Handling User Input

The user is first asked for two pieces of information by the script: a target integer and a list of integers to search for the target. This interaction is managed by the function getUserInputs(). By providing an exit option during input collecting, the method makes the user experience more seamless by terminating if the user enters an empty string at any time.

A list comprehension is used to split the integers that the user enters and turn them into a list. It also handles removing any extra spaces. Running search algorithms requires this user input, and the function makes sure all inputs are valid integers by managing exceptions and clearly indicating invalid inputs with error messages.

Using Search Algorithms in Practice

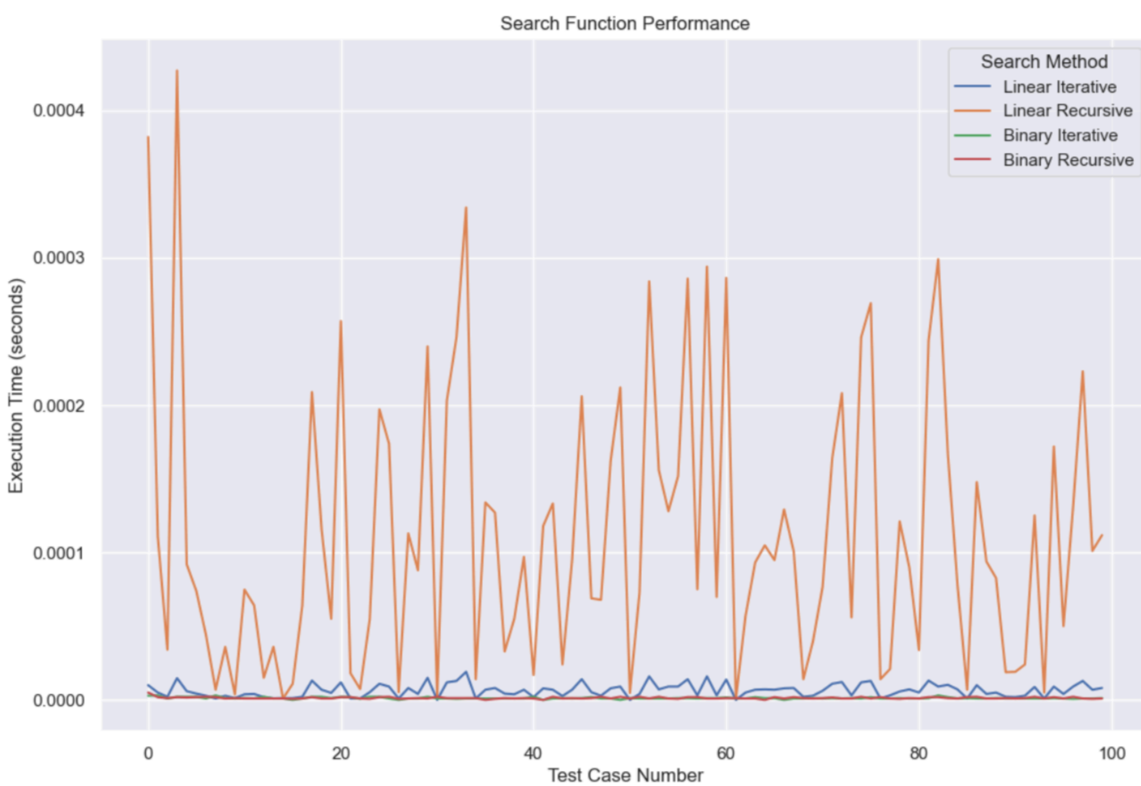
Binary and linear search techniques are implemented both recursively and iteratively in the code:

- Linear Search (Iterative and Recursive): Until the target is located or the list is exhausted, the iterative version (linearSearchIterative()) sequentially goes through each element of the list. Until the target is located or the list is exhausted, the recursive variant, linearSearchRecursive(), calls itself with a decreased list size (excluding the first entry).

- Binary Search (Iterative and Recursive): In the iterative variant (binarySearchIterative()), the boundaries are adjusted depending on comparisons with the center element, therefore repeatedly halving the search space. The binarySearchRecursive() recursive version does the same thing by calling itself with different boundaries depending on whether the target is more or less than the middle element. The list must be sorted for both binary search methods to work, and this is checked using the formula sorted(userList) == userList.

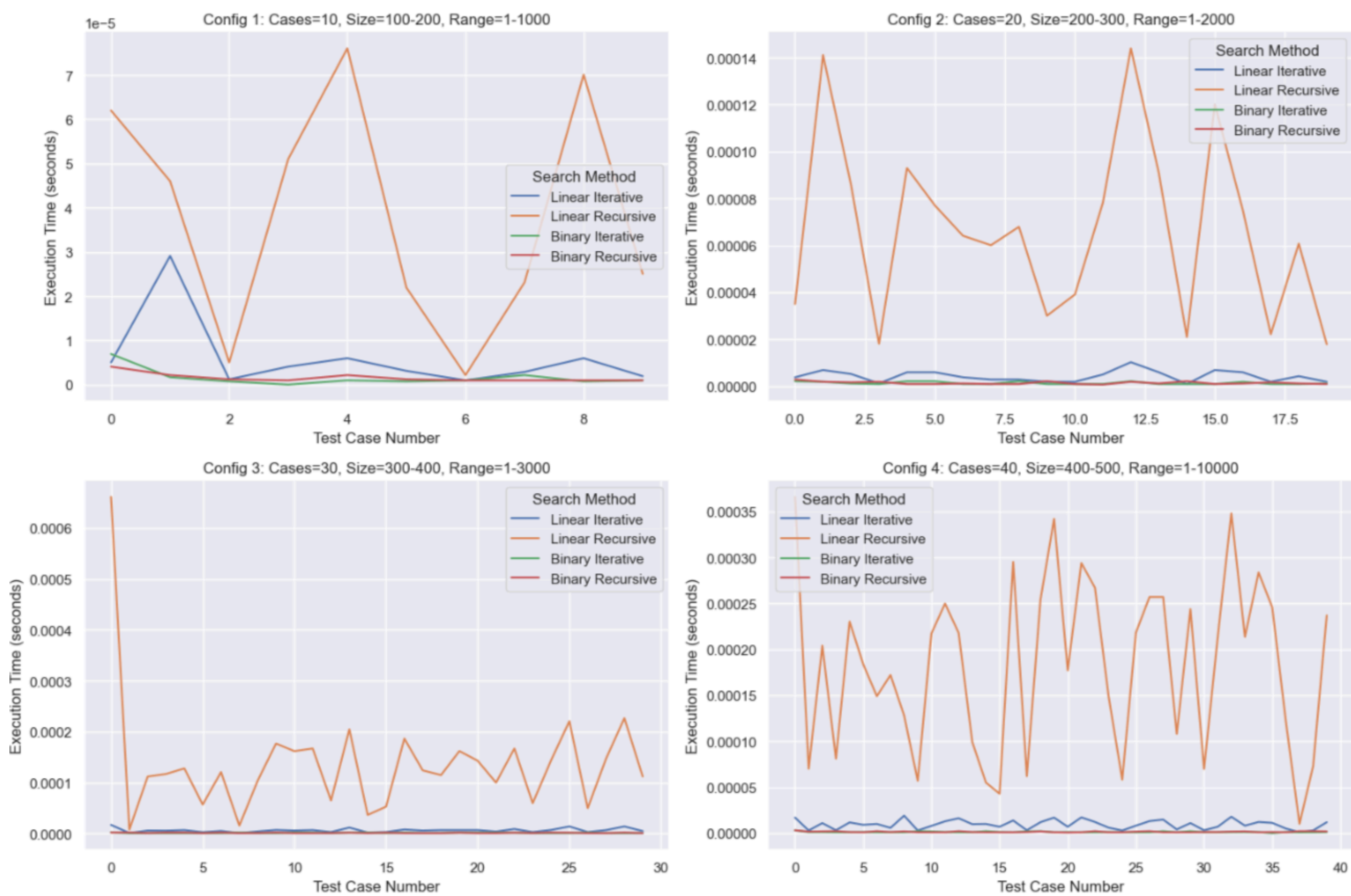
Performance Results

To evaluate the performance of each algorithm, metrics such as execution time and memory usages are taken into consideration.



(Figure 1)

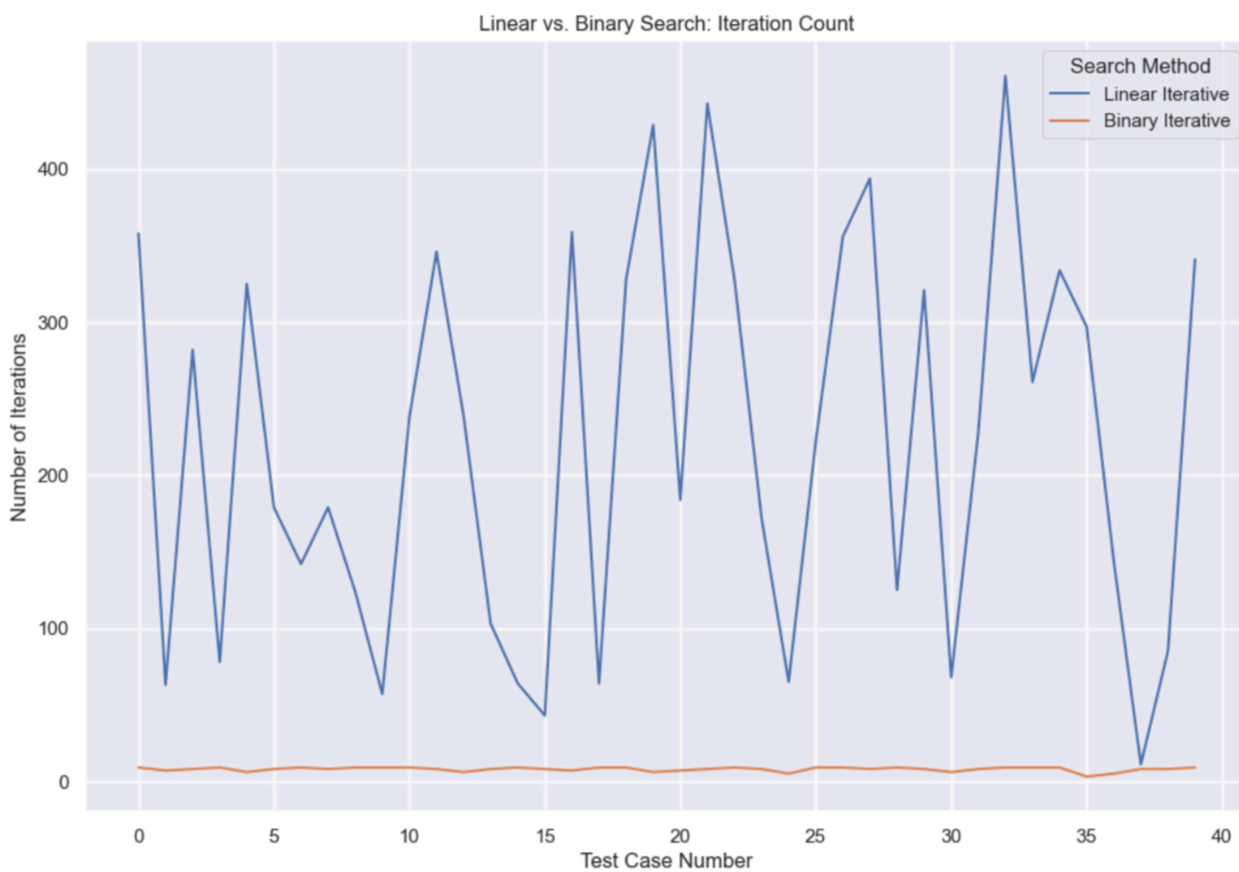
In the graph above it is shown that the linear iterative search function performs significantly worse than the other algorithms when it comes to execution time.



(Figure 2)

Execution Times

When considering each algorithm on a case by case basis with distinct parameters, it can be observed that there is the most amount variation of execution time when there are only ten cases, seen in *Configuration 1*, with some variation in the linear iterative method spiking at the first test case with 3 seconds, then increasing in execution speed as the program continues. Linear recursive methods initially reduce from six seconds to under one second within the first two cases, however, continue to increase and decrease at an exponential rate as the program continues. Both binary iterative and binary recursive maintain lower execution times as the algorithm continues, with execution times of under one second each. As the number of cases and size increases, binary iterative and binary recursive continue to significantly outperform the linear versions of the algorithms, remaining well below the 50 microsecond threshold *(Configuration 4)*.



Linear vs Binary Iterations

Comparison between the linear and binary search algorithms can be seen in the graph above. From this graph it can be determined that for each test case for both linear and binary searches, the binary iterative search method is more efficient as it requires less iterations compared to the linear iterative search algorithm.

Function Calls

When calling the function, the Linear Search Recursive function calls 128 times *(Figure 3)* while the Linear Search Iterative method can function calls only four times *(Figure 4)*. Additionally, Binary Search Iterative and Binary Search Recursive have a function call count of eleven and five times respectively, determining that Linear Search Iterative and Binary Iterative Search have higher efficiency in terms of function calls *(Figures 5 and 6)*.

Memory Usage

As far as memory usage is concerned, Binary Search Iterative methods peaks at 208.89 mebibytes of storage space with increased increments of 2.72 mebibytes from the starting point to the peak memory usage *(Figure 7)*. Comparatively, Linear Search Iterative peak memory output is 205.58 mebibytes with increments of 0.09 mebbytes *(Figure 8)*. Additionally, peak memory values and increments for Linear Search Recursive are as follows: 206.16 mebibytes and 0.58 mebibytes *(Figure 9)*. For Binary Search Recursive, memory usage peaks at 206.03 mebibytes with increments of 0.47

mebibytes *(Figure 10)*. From these results, it can be determined that Linear Search Iterative methods have the lowest memory usage and smallest increments.

Discussion

Algorithm Efficiency and Use Cases

- Linear Search:
  - Pros: Easy to use and efficient with tiny or unsorted datasets.
  - Cons: Low efficiency with an O(n) average time complexity on big datasets.
  - Use Cases: These are best utilized in situations when there is little to no data sorting and infrequent searches.

- Binary Search:
  - Pros: time complexity of O(log n) on sorted lists is highly efficient.
  - Cons: Needs sorted input; maintaining a sorted dataset requires more time or resources.
  - Use Cases: Optimal for large, sorted datasets that are retrieved quickly enough to justify the overhead of maintaining the list sorted.

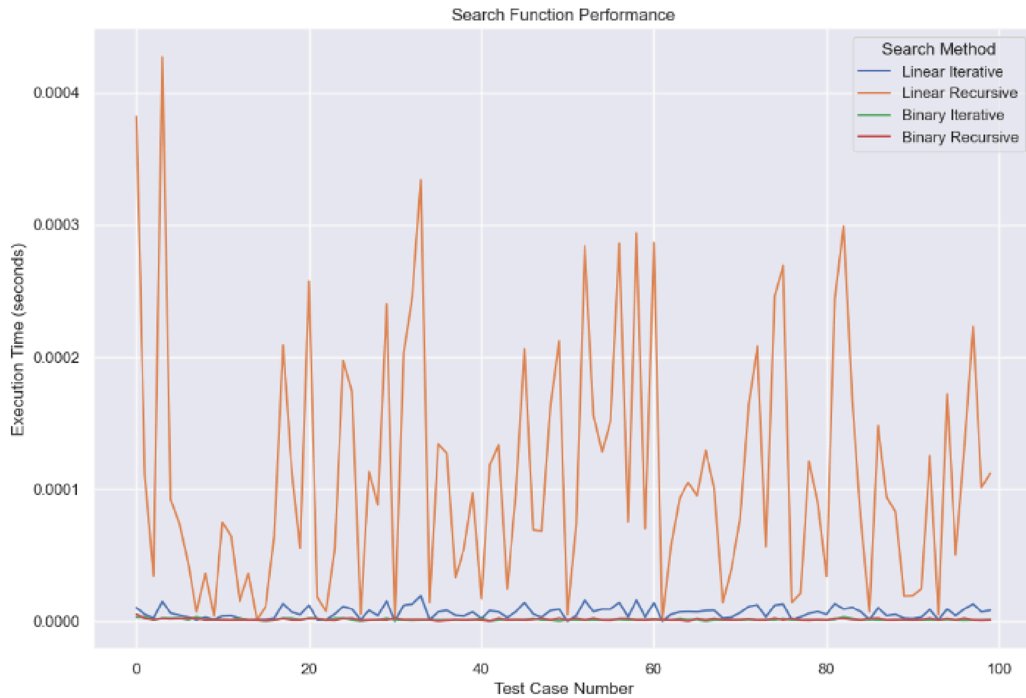
Practical Considerations

Practically speaking, an application's performance can be greatly impacted by the search algorithm selection. For example, situations where the data is not frequently searched and has an unpredictable order may be better suited for linear search. On the other hand, in systems where retrieving data is a routine task, binary search provides significant speed gains that support the necessity of preserving a sorted order.

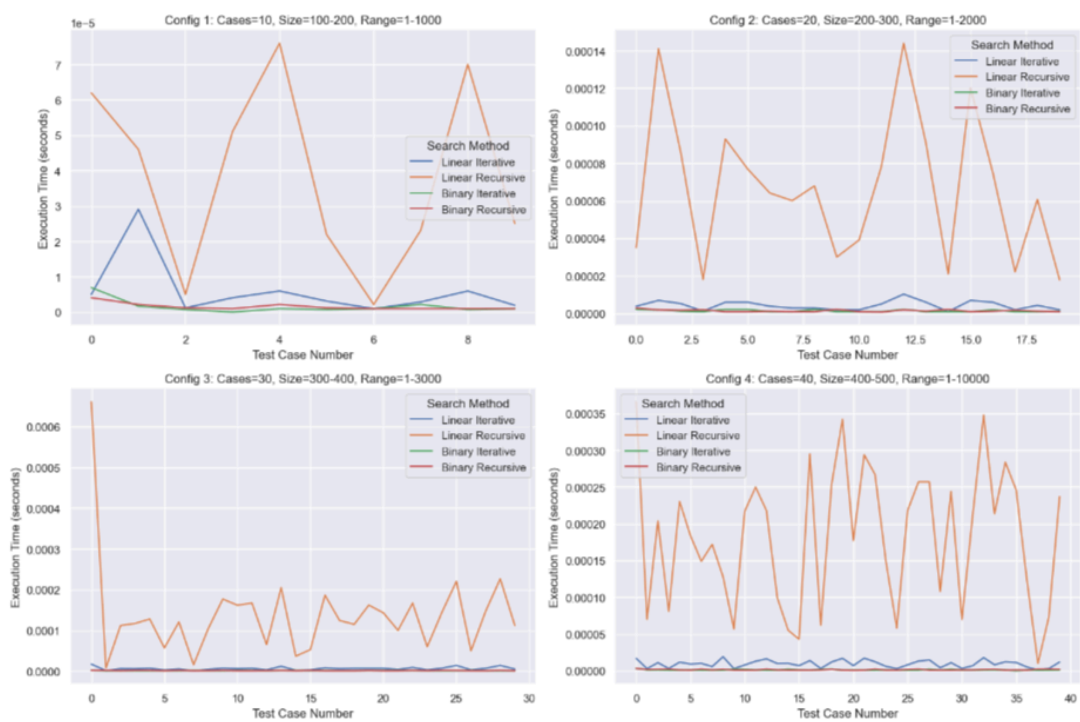
Conclusion

Based on the analysis above, it can be determined that overall Binary Iterative Search algorithms outperform the other approaches when comparing execution times, function calling abilities, and iterations completed. However, it is important to note that considering only memory usage, linear iterative search algorithms are the most efficient.

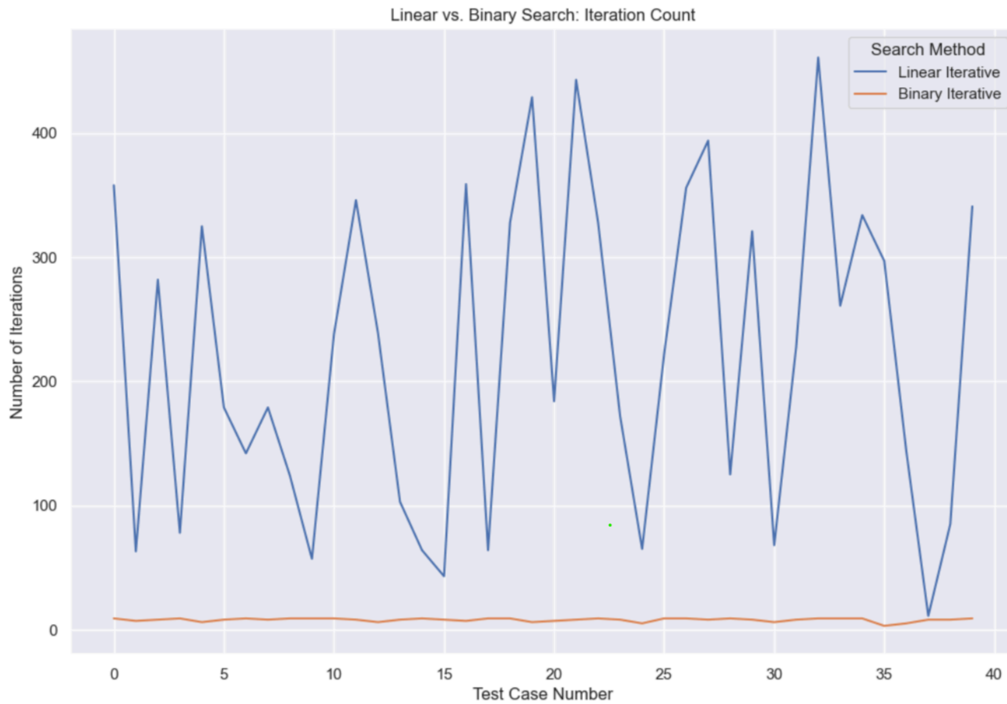
Appendices



(Figure 1)



(Figure 2)



(Figure 3)

```
%prun linearSearchRecursive(test_cases[1][0], test_cases[1][1])
```

```
128 function calls (66 primitive calls) in 0.000 seconds
```

(Figure 4)

```
%prun linearSearchIterative(test_cases[1][0], test_cases[1][1])
```

```
4 function calls in 0.000 seconds
```

(Figure 5)

```
%prun binarySearchRecursive(test_cases[1][0], test_cases[1][1])
```

```
11 function calls (5 primitive calls) in 0.000 seconds
```

(Figure 6)

```
%prun binarySearchIterative(test_cases[1][0], test_cases[1][1])
```

```
5 function calls in 0.000 seconds
```

(Figure 7 + 8)

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    51    136.2 MiB    136.2 MiB          63   def linearSearchRecursive(sourceList, target, index=0):
    52                                             """Finds the target within the `sourceList` using the linear search met
hod recursively.
    53                                             Args:
    54                                                 sourceList (list): A list of `int` values
    55                                                 target (int): The `int` value that will be searched for
    56                                             Retuns:
    57                                                 If the `target` is found, the index of the `sourceList` is returne
d. `False` is returned if otherwise
    58                                             """
    59    136.2 MiB      0.0 MiB          63       if sourceList[0] == target: # target found case
    60    136.2 MiB      0.0 MiB           1           return index
    61    136.2 MiB      0.0 MiB          62       elif len(sourceList) == 1: # target not found case
    62                                                 return False
    63                                             else: # recursion
    64    136.2 MiB      0.0 MiB          62           return linearSearchRecursive(sourceList[1:], target, index + 1)
```

```
%memit linearSearchRecursive(test_cases[1][0], test_cases[1][1])
```

```
peak memory: 206.16 MiB, increment: 0.58 MiB
```

```
    85    144.9 MiB      0.0 MiB           7               midpoint_index = (left + right) // 2
    86    144.9 MiB      0.0 MiB           7               iterations += 1
    87    144.9 MiB      0.0 MiB           7               if target == sourceList[midpoint_index]:
    88    144.9 MiB      0.0 MiB           1                   return midpoint_index, iterations
    89    144.9 MiB      0.0 MiB           6               elif target < sourceList[midpoint_index]:
    90    144.9 MiB      0.0 MiB           5                   right = midpoint_index - 1
    91                                                     else:
    92    144.9 MiB      0.0 MiB           1                   left = midpoint_index + 1
    93
    94                                                 return False, iterations
```

```
%memit binarySearchIterative(test_cases[1][0], test_cases[1][1])
```

```
peak memory: 208.89 MiB, increment: 2.72 MiB
```

```
Filename: /Users/josh/Desktop/Macbook Working Files/Git Repos/622-Final-Project/Task 1/Task1.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    36    145.0 MiB    145.0 MiB           1   def linearSearchIterative(sourceList, target):
    37                                             """Finds the target within the `sourceList` using the linear search met
hod iteratively.
    38                                             Args:
    39                                                 sourceList (list): A list of `int` values
    40                                                 target (int): The `int` value that will be searched for
    41                                             Retuns a `tuple`:
    42                                                 (target or False, iterations). If the `target` is found, the index
of the `sourceList` is returned. `False` is returned if otherwise
    43                                             """
    44    145.0 MiB      0.0 MiB           1       iterations = 0
    45    145.0 MiB      0.0 MiB          63       for index, each in enumerate(sourceList):
    46    145.0 MiB      0.0 MiB          63           iterations += 1
    47    145.0 MiB      0.0 MiB          63           if each == target:
    48    145.0 MiB      0.0 MiB           1               return index, iterations
    49                                             return False, iterations
```

```
%memit linearSearchIterative(test_cases[1][0], test_cases[1][1])
```

```
peak memory: 205.58 MiB, increment: 0.09 MiB
```

(Figure 9)

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    51    136.2 MiB    136.2 MiB          63   def linearSearchRecursive(sourceList, target, index=0):
    52                                             """Finds the target within the `sourceList` using the linear search met
hod recursively.
    53                                             Args:
    54                                                 sourceList (list): A list of `int` values
    55                                                 target (int): The `int` value that will be searched for
    56                                             Retuns:
    57                                                 If the `target` is found, the index of the `sourceList` is returne
d. `False` is returned if otherwise
    58                                             """
    59    136.2 MiB      0.0 MiB          63       if sourceList[0] == target: # target found case
    60    136.2 MiB      0.0 MiB           1           return index
    61    136.2 MiB      0.0 MiB          62       elif len(sourceList) == 1: # target not found case
    62                                                 return False
    63                                             else: # recursion
    64    136.2 MiB      0.0 MiB          62           return linearSearchRecursive(sourceList[1:], target, index + 1)
```

```
%memit linearSearchRecursive(test_cases[1][0], test_cases[1][1])
```

```
peak memory: 206.16 MiB, increment: 0.58 MiB
```

(Figure 10)

```
Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    96     72.6 MiB     72.5 MiB           7   def binarySearchRecursive(sourceList, target, left=0, right=None):
    97                                             """Uses binary search to find the target within a sorted list. Returns
the position of the target, if found, and `False` otherwise. Uses a recursive approach.
    98                                             Args:
    99                                                 sourceList (list): A sorted list of `int` values
   100                                                 target (int): The `int` value that will be searched for
   101                                                 left (int, optional): The starting index of the sublist to search w
ithin. Default is 0.
   102                                                 right (int, optional): The ending index of the sublist to search wi
thin. Default is the last index of the list.
   103                                             """
   104     72.6 MiB      0.0 MiB           7       if right is None:
   105     72.5 MiB      0.0 MiB           1           right = len(sourceList) - 1
   106
   107     72.6 MiB      0.0 MiB           7       if left <= right:
   108     72.6 MiB      0.0 MiB           7           midpoint_index = (left + right) // 2
   109     72.6 MiB      0.0 MiB           7           if sourceList[midpoint_index] == target:
   110     72.6 MiB      0.0 MiB           1               return midpoint_index
   111     72.6 MiB      0.0 MiB           6           elif target < sourceList[midpoint_index]:
   112     72.6 MiB      0.0 MiB           5               return binarySearchRecursive(sourceList, target, left, midpoint
_index - 1)
   113                                                 else:
   114     72.6 MiB      0.0 MiB           1               return binarySearchRecursive(sourceList, target, midpoint_index
+ 1, right)
   115                                             else:
   116                                                 return False
```

```
%memit binarySearchRecursive(test_cases[1][0], test_cases[1][1])
```

```
peak memory: 206.03 MiB, increment: 0.47 MiB
```

(Figure 11)