Brogan Avery
March 10, 2021
Search and Sort Report

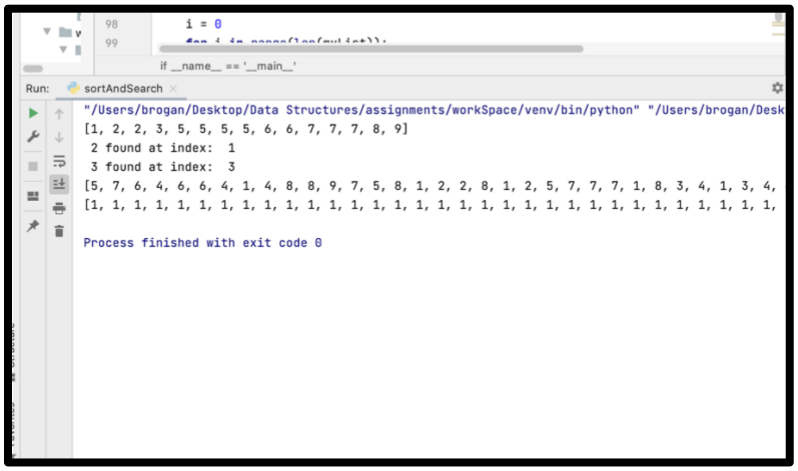
Linear Search

```
"""
Linear Search:
    How it works: In a sorted list, the function begins from the first
element and searches each sequential
    index till the desired value is found or the end of the list is reached.
Then it returns the index where
    the value was found or a message stating the value was not in the list.
    When to use it:  Because this method is based on a sequential search from
beginning index to ending index,
    the list is not required to be sorted before this method is implemented.
    When it cannot be used: The biggest limitation of this method is that it
will take longer than the binary search and this time will be noticeable with
large lists
"""
```

Binary Search

```
"""
Binary Search:
    How it works: In a sorted list, the function finds the middle index by
dividing the length of the list
    in 2 and then going to that index. It then splits the list into two lists
including the values on either side of
    the middle element. Since the list is already sorted, if the number we
are looking for is larger than the
    middle element, the number we are looking for has to be in the list to
the right or if the number we are
    looking for is smaller than the middle element, the number we are looking
for has to be in the list to the left. Or the
    value being searched for could of course not be present. This essentially
cuts the amount of time to traverse a list in half since it
    cut the length of the list in half.
    When to use it: This becomes most advantageous when the list size is very
long.
    When it cannot be used: The list much first be sorted in order to use
this method of searching
"""
```

Bubble Sort



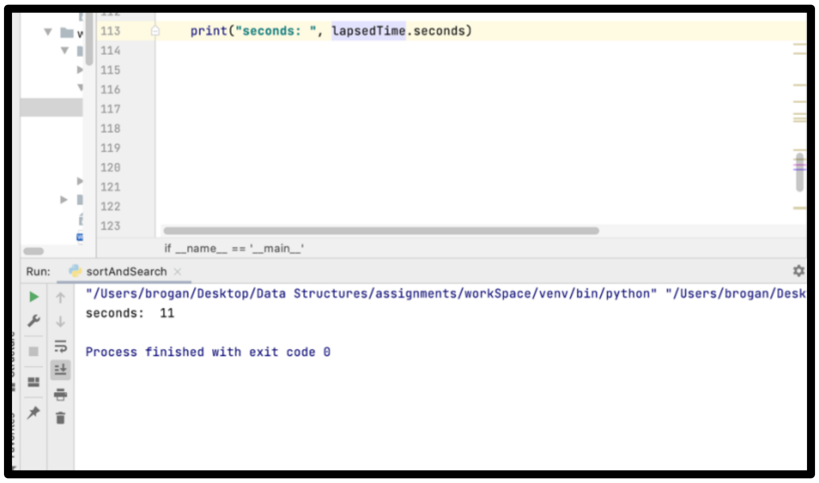
```
i = 0
```

```
if __name__ == '__main__'
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Desk
[1, 2, 2, 3, 5, 5, 5, 5, 6, 6, 7, 7, 7, 8, 9]
 2 found at index:  1
 3 found at index:  3
[5, 7, 6, 4, 6, 6, 4, 1, 4, 8, 8, 9, 7, 5, 8, 1, 2, 2, 8, 1, 2, 5, 7, 7, 7, 1, 8, 3, 4, 1, 3, 4,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

Process finished with exit code 0
```

Number 1 : Bubble Sort algorithm

The last two lines of this output demonstrate a list/array before it was sorted using the bubble sort method and after. The bubble sort method can be thought of as working in two phases (or one outer process and one inner process. The inner process is to compare each value in the list (starting with the first index) to the value right after it. If the value in the first index being compared is larger than the second, the values are swapped. Otherwise, the values are left where they are. This is done for each element in the array. The outer process is that it will keep repeating the first step of comparing each value to the right of it for the entire list over and over in a loop until the list is correctly ordered.



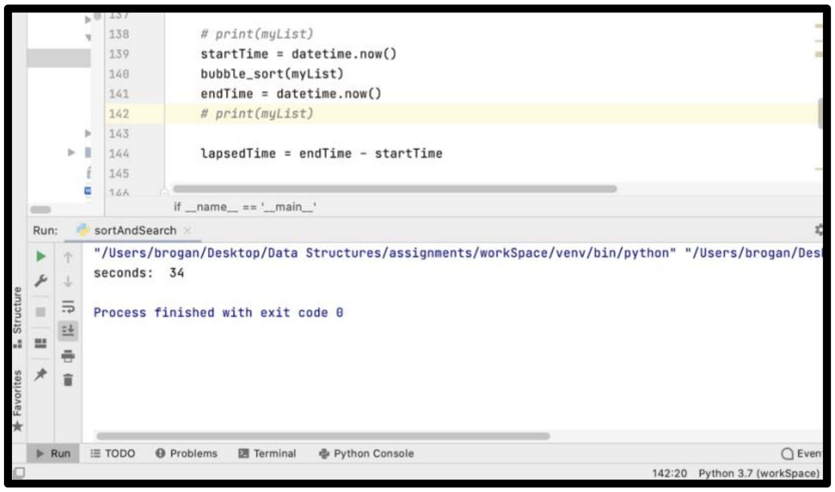
```
    print("seconds: ", lapsedTime.seconds)
```

```
if __name__ == '__main__'
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Desk
seconds:  11

Process finished with exit code 0
```

Number 2 : Bubble Sort Timing 10,000 items

This picture shows the time it took to bubble sort through 10,000 items. It shows how efficient or not this method is compared to the timing of other sort methods. It also shows how when it comes to smaller sized lists, it is not super important to worry about time, but as the list length grows, it becomes more important to consider. Because the first step of the process for bubble sort is to compare every single value with the value to its right and potentially swap them, the time it takes to complete each iteration of the entire array could be quit lengthy. Additionally it has the time of repeating the outer loop until the list is ordered so that would give the time complexity a value of worst-case and average complexity of O(n^2).



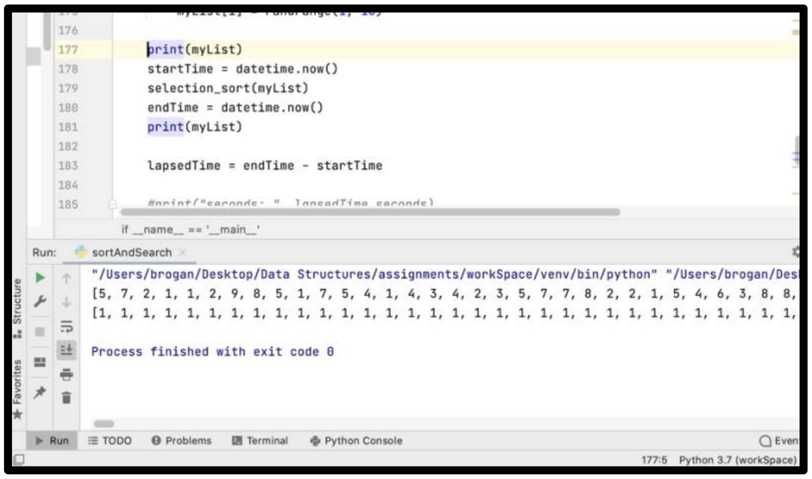
```
        # print(myList)
        startTime = datetime.now()
        bubble_sort(myList)
        endTime = datetime.now()
        # print(myList)

        lapsedTime = endTime - startTime
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
seconds:  34

Process finished with exit code 0
```

Number 3 : Bubble Sort Timing 100,000 (16,000) items

Here, it shows the bubble sort with only around 15 – 20,000 items because my computer is very old and slow. But it demonstrates how very quickly, the more items added, the less efficient the bubble sort method becomes and the longer and longer it takes to run. If it was a program with millions of items in a list (especially on my slow machine) it could literally take hours or days to run.

Selection Sort

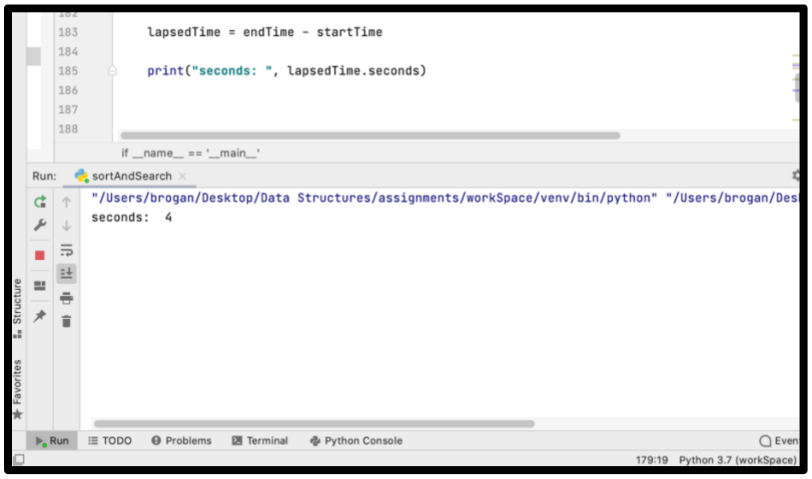


```
    print(myList)
    startTime = datetime.now()
    selection_sort(myList)
    endTime = datetime.now()
    print(myList)

    lapsedTime = endTime - startTime

    #print("seconds: ", lapsedTime.seconds)
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
[5, 7, 2, 1, 1, 2, 9, 8, 5, 1, 7, 5, 4, 1, 4, 3, 4, 2, 3, 5, 7, 7, 8, 2, 2, 1, 5, 4, 6, 3, 8, 8,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

Process finished with exit code 0
```

Number 4 : Selection Sort algorithm

The above image shows the output of the list before and after the implementation of the selection sort method. The selection sort method works in two phases (or one outer process and one inner process. The first of inner step is to loop through the entire list and select the smallest value from it. It them moves that value to the front of the list. It continues doing this over and over again in a loop (finding the smallest number and moving it to the front. However, since the elements that have already been moved to the front are ordered, it only needs to loop through the part of the array that is not sorted. So each loop, it treats the array like its size has been decreased by one and ignoring the already order values.

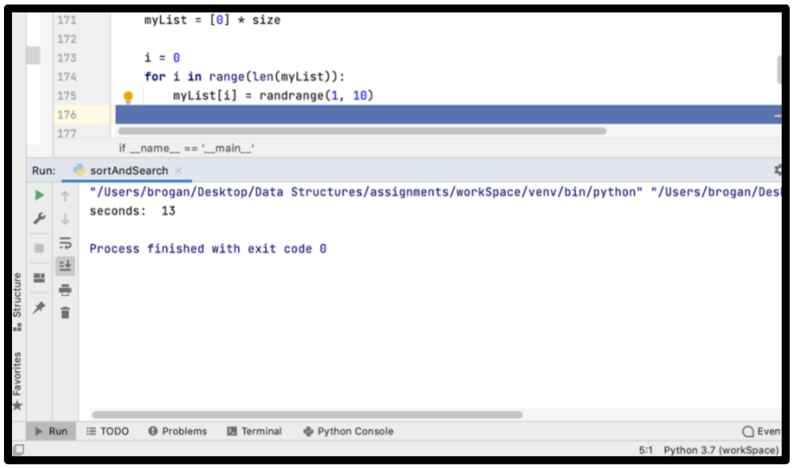


```
    lapsedTime = endTime - startTime

    print("seconds: ", lapsedTime.seconds)
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
seconds:  4
```

Number 5 : : Selection Sort Timing 10,000 items

This picture shows the time it took to do a selection sort through 10,000 items. Compared to the 11 seconds it took to sort the same size array with bubble sort, this method only takes 4. It shows how efficient or not this method is compared to the timing of other sort methods.
In contrast with the bubble sort method which makes a comparison between every single array element and the element to its right, the selection method is only looking to answer the question of “is this value the smallest in the array or is it not”. Therefore, there are less time consuming comparisons to be made which helps explain why this method is done faster. Additionally it has the time of repeating the outer loop until the list is ordered so that would give the time complexity a value of worst-case and average complexity of O(n^2), which is the same as the bubble sort, but the co-efficients play an important role here.



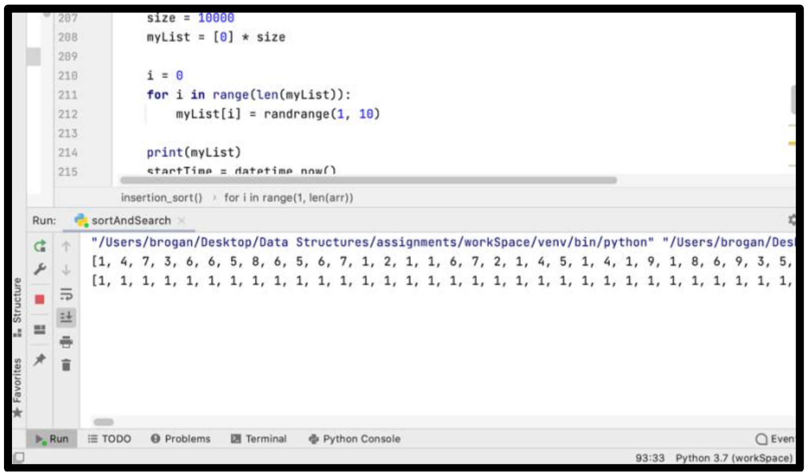
```python
    myList = [0] * size

    i = 0
    for i in range(len(myList)):
        myList[i] = randrange(1, 10)

```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
seconds:  13

Process finished with exit code 0
```

Number 6 : Selection Sort Timing 100,000 (16,000) items

Here, it shows the selection sort with only around 15 – 20,000 items because my computer is very old and slow. When compared to the bubble sort, it is obvious that it cuts the time in over half compared to how long that method would take to sort the same number of items. All though the outer process of looping through the array until it is sorted is the same or similar time complexity for each method, the difference is the inner step. As described earlier, the complexity of comparison is noticeably smaller in this method of sorting.

Insertion Sort

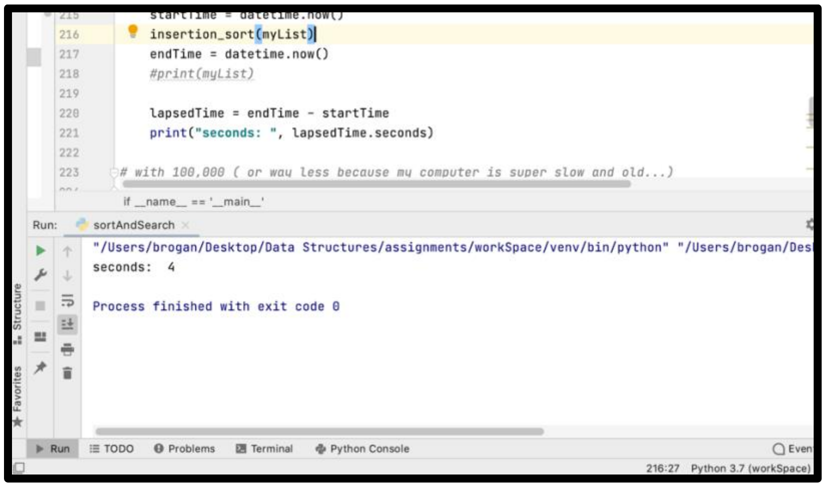


```
    size = 10000
    myList = [0] * size

    i = 0
    for i in range(len(myList)):
        myList[i] = randrange(1, 10)

    print(myList)
    startTime = datetime.now()
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
[1, 4, 7, 3, 6, 6, 5, 8, 6, 5, 6, 7, 1, 2, 1, 1, 6, 7, 2, 1, 4, 5, 1, 4, 1, 9, 1, 8, 6, 9, 3, 5,
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

Number 7 : Insertion Sort algorithm

The above image shows the output of the list before and after the implementation of the insertion sort method. While both of the prior sort methods consisted of two processes, the insertion sort consists of 3. This is because it is more of a hybrid of the above two sort methods. Like the selection sort, it beginnings by selection a value in the array to move to the front of the array (and the end of the part of the array that is already sorted). But instead of selecting this value because it is the smallest in the array, it is simply selecting the next element in the unsorted array and moving it to the sorted one half. Then, it will function similar to the bubble sort where it compares the values of indices that are side by side and swaps them if they are not in the correct order. It will perform this check on each value it moves over next to until it has moved all the way to the left or the value to its left is the same size or smaller.

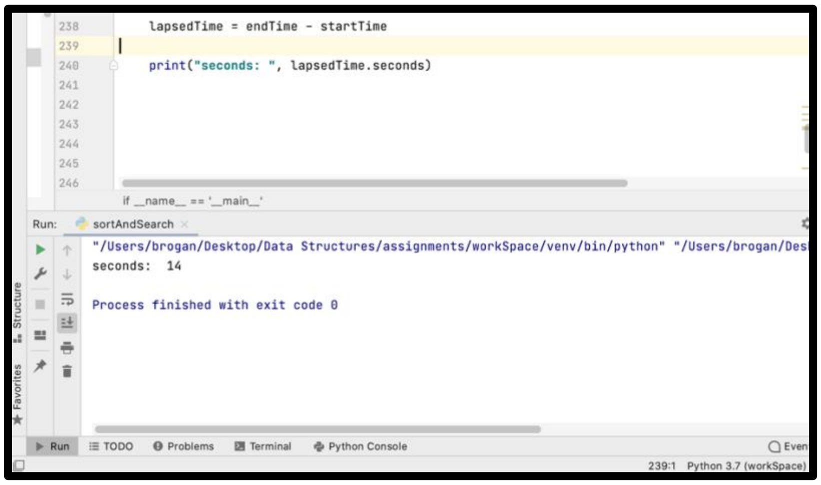


```
    startTime = datetime.now()
    insertion_sort(myList)
    endTime = datetime.now()
    #print(myList)

    lapsedTime = endTime - startTime
    print("seconds: ", lapsedTime.seconds)

# with 100,000 ( or way less because my computer is super slow and old...)
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
seconds:  4

Process finished with exit code 0
```

Number 8 : Insertion Sort Timing 10,000 items

This picture shows the time it took to do an insertion sort through 10,000 items. Compared to the 11 seconds it took to sort the same size array with bubble sort, and the 4 seconds it took to complete the selection sort, this method only takes 4. This demonstrates that in smaller numbers of items, there is not going to be a very noticeable distinction between the selection and insertion methods. However, if one is to actually break down the structure, because this method essentially uses the most efficient parts of the previous to sort methods, it is more efficient than either (but it is more noticeable in larger lists). While the other 2 methods have 2 parts (the inner and the outer) this method can be seen as hanging 3, (2 inner parts and one outer loop). This maximizes the efficiency of bother the outer and inner steps.



```
        lapsedTime = endTime - startTime

        print("seconds: ", lapsedTime.seconds)
```

```
"/Users/brogan/Desktop/Data Structures/assignments/workSpace/venv/bin/python" "/Users/brogan/Des
seconds:  14

Process finished with exit code 0
```

Number 9 : Insertion Sort Timing 100,000 (16,000) items

This shows the insertion sort with the same 16,000 item list that was ran with the previous two methods. Although here it actually shows this method taking 1 second longer than the selection method, that usually would not be the case with larger numbers. However, since I need to run only 16,000 for the bubble sort to meet the requirements of the assignment, the true value of this method of sorting is not displayed. But normally, as previously stated, this method is most efficient when used on very large lists.

While each method completes the same goal, the way of reaching the end solution is done a little differently in each of the three methods of searching discussed. The biggest difference between bubble and selection sort is the first step (discussed above). The commonality that they share is the repetitive outer loop step until the lists can be read through completely and meet the satisfaction of being in order as desired. The final method discussed, the insertion method, is more similar to a combination of the first two methods of sorting which is able to maximize efficiency and minimize the amount of loops (or nested loops) needed to order the list.

As mentioned before, when the list is relatively small, there is likely going to be no important or meaningful difference between any of the methods above to the point where it would actually matter in the long run goal of making a program better for the user. However, if we want to maximize efficiency, we can move on to the selection sort, which even at smaller sized lists is more efficient than lists sorted with the bubble sort method. The true advantage of the insertion sort comes into play when the list size is very large.

The very obvious reason why it is important to test arrays of size 10,000, 1000,000 and even larger that is demonstrated in this report is to show how some methods of soring are more efficient and faster than others. Additionally this testing is important because it gives the developer a realistic expectation of what real run times will be and will help to pin point major issues, if something that should be done in seconds takes 20 minutes to load for example, that’s bad. This report also shows how big a role the difference in machines that every user uses can play. It may be common that a good, new computer can run a list of 100,000 items in 30 seconds, but if your target audience in an app that you are building is all poor college students like in this case, the expectations that you have for the user’s use of the project are not real and this will cause a lot of angry users and loss of money in a real company.