**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #2**

Due: September 24th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to solve the selection problem in which various methods would be used to find the kth smallest element in a presumably unordered list. To do this, we must implement five different methods that use either bubble sort or some variation of quicksort to first sort the list as needed before then returning and printing the desired element. The main objective of this lab is to understand how different approaches to the problem influence their own running times by the number of comparisons they make.

**Proposed Solution Design and Implementation**

**Part 1:**

**Operation #1 (select\_bubble(L,k)):**

For this operation, as with every following operation, I first wrote two separate if statements such that if the given list was empty or the index of the desired element was outside the range of the given list, the method would return None and inform the user of these dilemmas. From there, now that the list and index should both suffice, the method first makes a copy of the list so as not to alter the original before then sorting it with two nested for loops: The outer loop tracks each element such that the inner loop can compare said element to every element. If an element other than the current element is smaller, the two elements are then swapped, and the loops continue where they left off. The main idea behind this list is to iterate through the list *n2* times (thus having a big-O running time of O(*n*2)), where *n* is the length of the given list.

**Operation #2 (select\_quick(L,k)):**

For this operation, after the aforementioned if statements and list copying, the method then calls to a separate method *quicksort(L, first, last)*, which is where the recursion for quicksort can occur. In this method, it checks that the initially given index of 0 is less than that of len(L) - 1 as it will do with the various indices in future recursive calls such that once a recursive call is made referring to only one element, it will stop. Then, assuming the given call is referring to at least two elements, the method will call on a third method, *partition(L, first, last)*, to begin sorting the given section of the original list.

To do this, the *partition()* method tracks the first and last indices, as well as a pivot index found in the middle of the list, and begins comparing the elements of said indices. The main idea here is that if any elements are smaller than the pivot element, they will be placed on the left of the pivot, whereas any elements larger than the pivot will be placed on the right of the pivot. Once this is done, *partition()* returns the last index, which now represents the new index at which the pivot lies.

The *quicksort()* method than takes this index and recursively calls itself twice: The first call will have it repeat this process, but only with the left half of the list (including the former pivot), and the second call will do the same with only the right half of the list (not including the pivot). The main idea here is that by sorting the list in increasingly smaller subsections, the method will recursively call itself *2n - 1* times (thus having a big-O running time of O(*n*)), where *n* is the length of the given list.

**Operation #3 (select\_modified\_quick(L,k)):**

For this operation, I merely recreated the aforementioned *quicksort(L, first, last)* method, only this time, rather than have said method recursively call itself twice on every call, it instead only calls itself once depending on the returned pivot index. Namely, if the pivot index is greater than or equal to the desired index, then a recursive call is made only to the left half of the current section of the list. Conversely, if the pivot index is less than the desired index, a recursive call is made only to the right half of the current section of the list instead. The main idea here is that by *selectively* sorting the list in increasingly smaller subsections, the method will only recursively call itself *n//2* times (thus still having a big-O running time of O(*n*)), where *n* is the length of the given list.

**Part 2:**

**Operation #4 (select\_stack\_quicksort(L, k)):**

For this operation, the idea is to implement quicksort using a stack rather than recursion. To do this, the method calls to a separate *stack\_quicksort(L, first, last)* method, which then creates a stack that stores the first and last indices of a given section of the list rather than use the entire list outright. It then goes about quicksort by recording the indices. Namely, the method pops *last* and *first* from the current stack and uses them as parameters in a call to a new *partition\_index(L, first, last)* method that behaves differently from the previous *partition()* method.

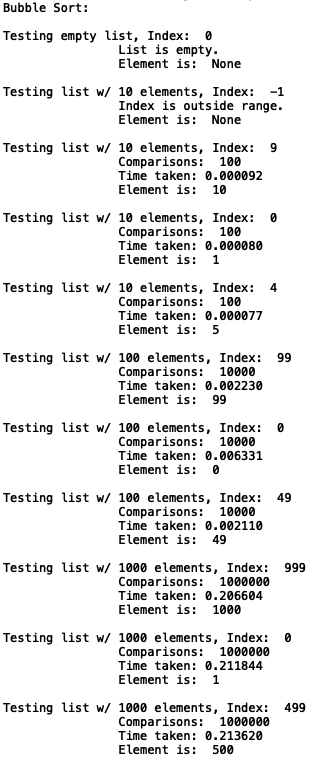
This *partition\_index()* method tracks the pivot index by first assigning it as -1 (an index just outside the range of the stack), then incrementing it whenever an element smaller than the last element is found as it iterates through the stack. Whenever this incrementation happens, if the pivot index is different from the current index, the values of the two indices are swapped, as the pivot index is representative of an element larger than and to the left of the current element. Once it reaches the last value, this method finishes by swapping the last value with the value just after the pivot index, as that is again the leftmost element that is still larger than the last value. The main idea here is that every value less than the current last value of the list has now been sorted on the left side of said value, and the value right after said value has now been made the new pivot point. This method then returns the index of this new pivot point.

The *stack\_quicksort()* method then takes the returned pivot index and considers the following: If the index before the pivot index is greater than the first index formerly in the stack, then the stack should be refilled, or appended, with that original first index as well as this index before the pivot, effectively representing the left half of the list. Conversely, if the index after the pivot index is less than the former last index of the stack, then the stack should be refilled with that index as well as the former last index, effectively representing the right half of the list. The method then repeats this behavior until the stack is inevitably not refilled with any indices, which indicates that the list is now sorted up to the desired index.

**Operation #5 (select\_while\_modified\_quicksort(L,k)):**

For this operation, the idea is to implement Operation #3 without stacks or recursion. To do this, the method now calls a separate *while\_modified\_quicksort(L, first, last)* method, which behaves in much the same way as the *modified\_quicksort()* method, only instead of recursion, there is now a while loop being implemented. Again, the *partition\_index()* method is used to perform quicksort based on the indices rather than on the values, and it is simply the shell of the *while\_modified\_quicksort()* method that cuts down on the number of calls to *partition\_index()*: Namely, when the pivot index is returned, if it is less than the desired index, it is simply called again, but this time only for the left half of the list. Conversely, if the pivot index is greater than the desired index, it is called again with only the right half of the list in its parameters. Lastly, once the pivot index is equal to the desired index, then like *modified\_quicksort()*, we can say that the method has sorted as much as is necessary to return the desired element.

**Experimental Results**



**Operation #1:**

For this operation, as with every other operation, I tested how it operates with an empty list, an index out of range (-1), and three different lists of 10, 100, and 1000 elements, each searching for their first, last, and middle elements. From this, I gathered the number of comparisons made, the amount of time taken, and the returned element of each case.

Case 1:

The list is empty, so nothing is returned.

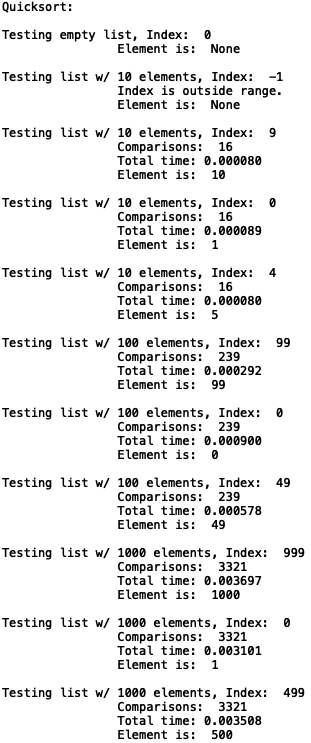
Case 2:

The index is out of range, so nothing is returned.

Case 3:

1. Given a list of 10 elements, on average, the method took 0.000080 milliseconds and 100 comparisons to find the element.
2. Given a list of 100 elements, on average, the method took 0.003667 milliseconds and 10,000 comparisons to find the element.
3. Given a list of 1000 elements, on average, the method took 0.210689 milliseconds and 1,000,000 comparisons to find the element.

Of note is that the number of comparisons is always *n*2, where *n* is the length of the list.

**Operation #2:**

Case 1:

The list is empty, so nothing is returned.

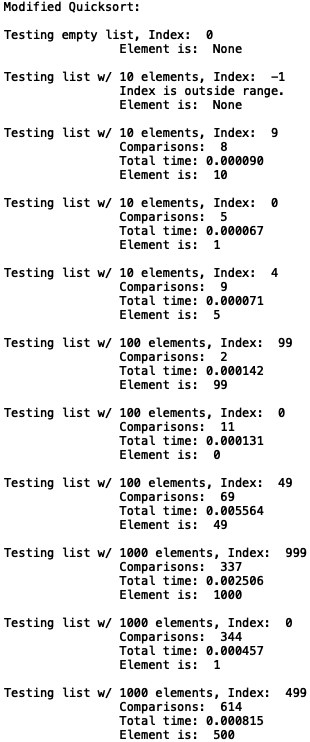
Case 2:

The index is out of range, so nothing is returned.

Case 3:

1. Given a list of 10 elements, on average, the method took 0.000083 milliseconds and 16 comparisons to find the element.
2. Given a list of 100 elements, on average, the method took 0.000590 milliseconds and 239 comparisons to find the element.
3. Given a list of 1000 elements, on average, the method took 0.003435 milliseconds and 3,321 comparisons to find the element.

Of note is that the number of comparisons is always more than my hypothesized *2n - 1* times.

**Operation #3:**

Case 1:

The list is empty, so nothing is returned.

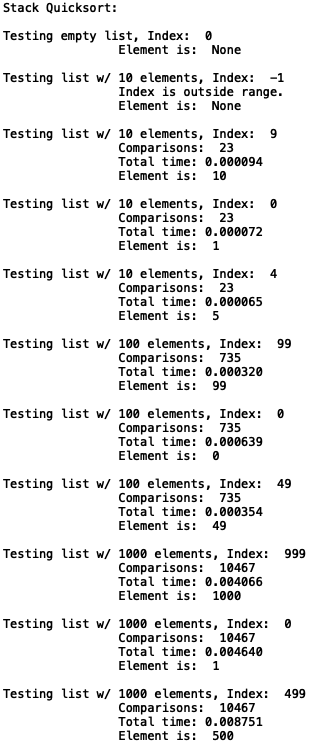
Case 2:

The index is out of range, so nothing is returned.

Case 3:

1. Given a list of 10 elements, on average, the method took 0.000076 milliseconds and 7 comparisons to find the element.
2. Given a list of 100 elements, on average, the method took 0.001946 milliseconds and 27 comparisons to find the element.
3. Given a list of 1000 elements, on average, the method took 0.001259 milliseconds and 432 comparisons to find the element.

Of note is that the number of comparisons is greater than my hypothesized *n//2* times for Case 3a, but is much less than that for Cases 3b and 3c.

**Operation #4:**

Case 1:

The list is empty, so nothing is returned.

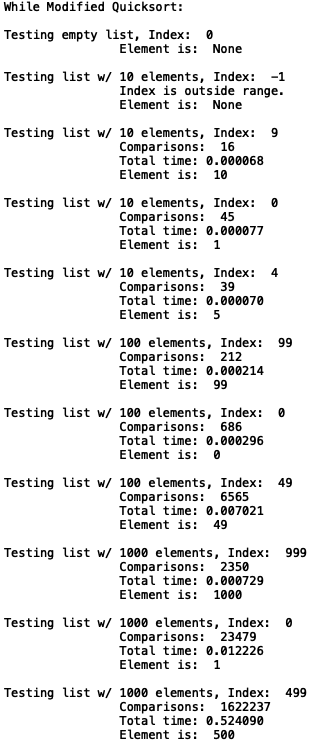
Case 2:

The index is out of range, so nothing is returned.

Case 3:

1. Given a list of 10 elements, on average, the method took 0.000077 milliseconds and 23 comparisons to find the element.
2. Given a list of 100 elements, on average, the method took 0.000438 milliseconds and 735 comparisons to find the element.
3. Given a list of 1000 elements, on average, the method took 0.005819 milliseconds and 10,467 comparisons to find the element.

NOTE: Of note is that the number of comparisons is

**Operation #5:**

Case 1:

The list is empty, so nothing is returned.

Case 2:

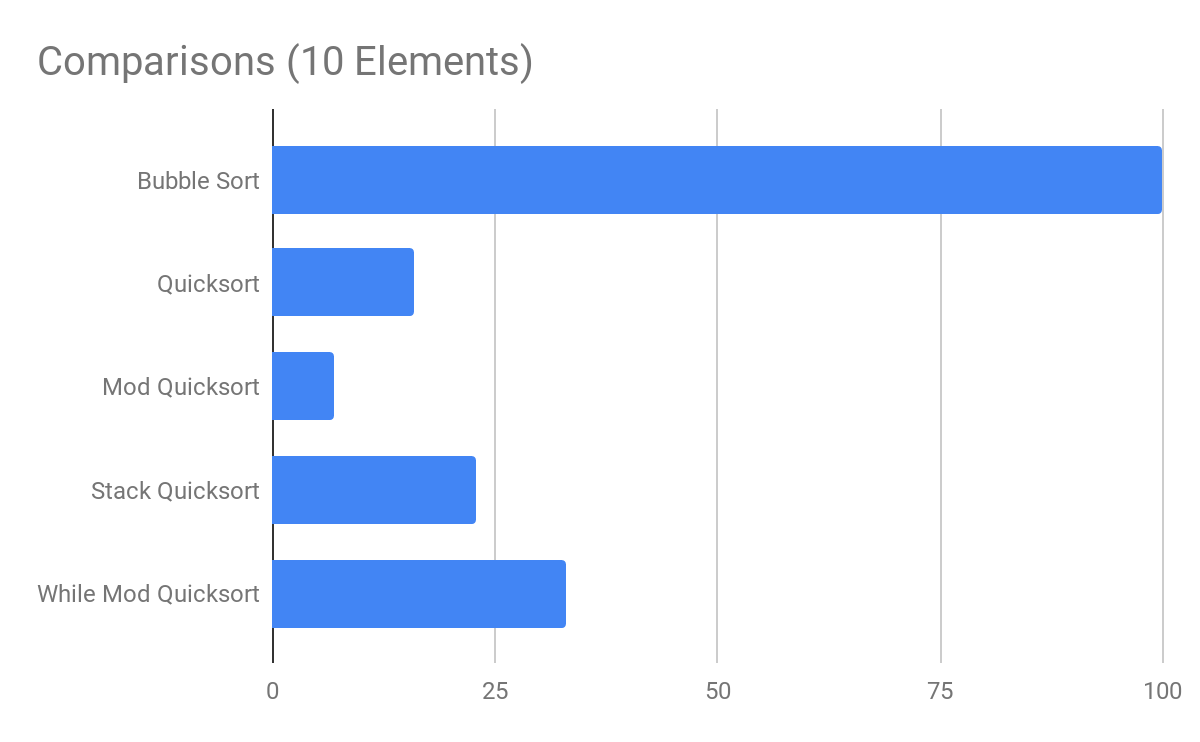
The index is out of range, so nothing is returned.

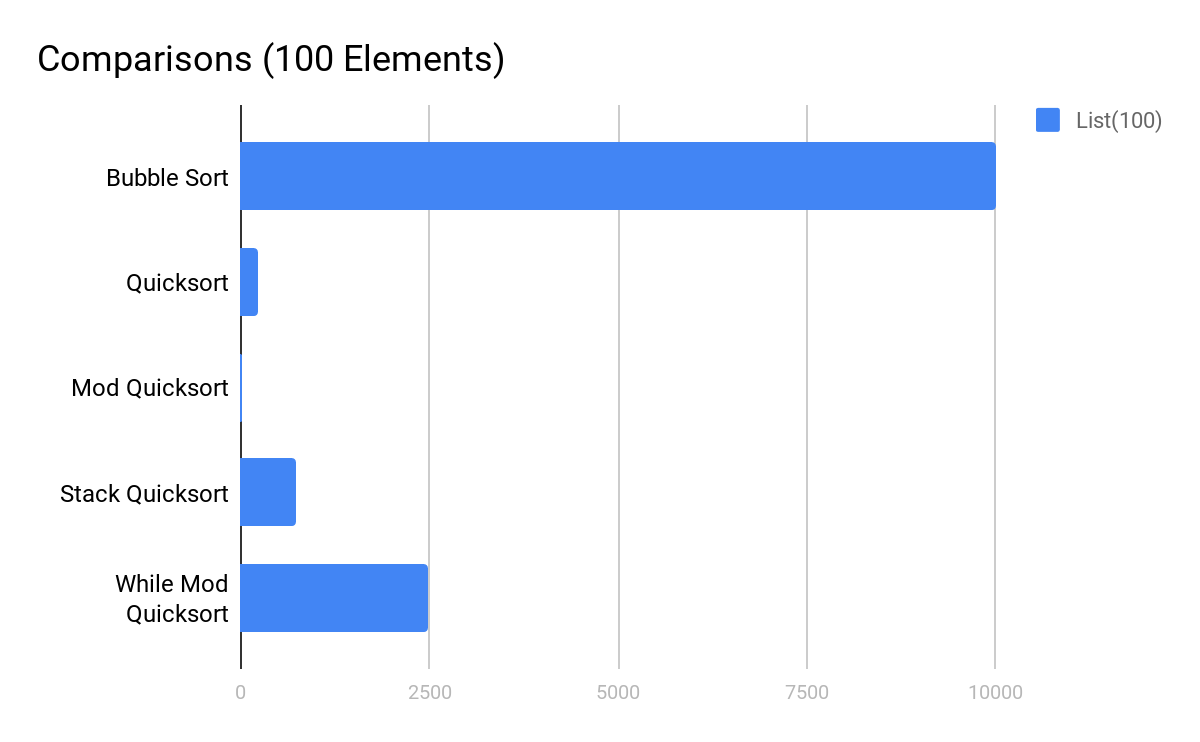
Case 3:

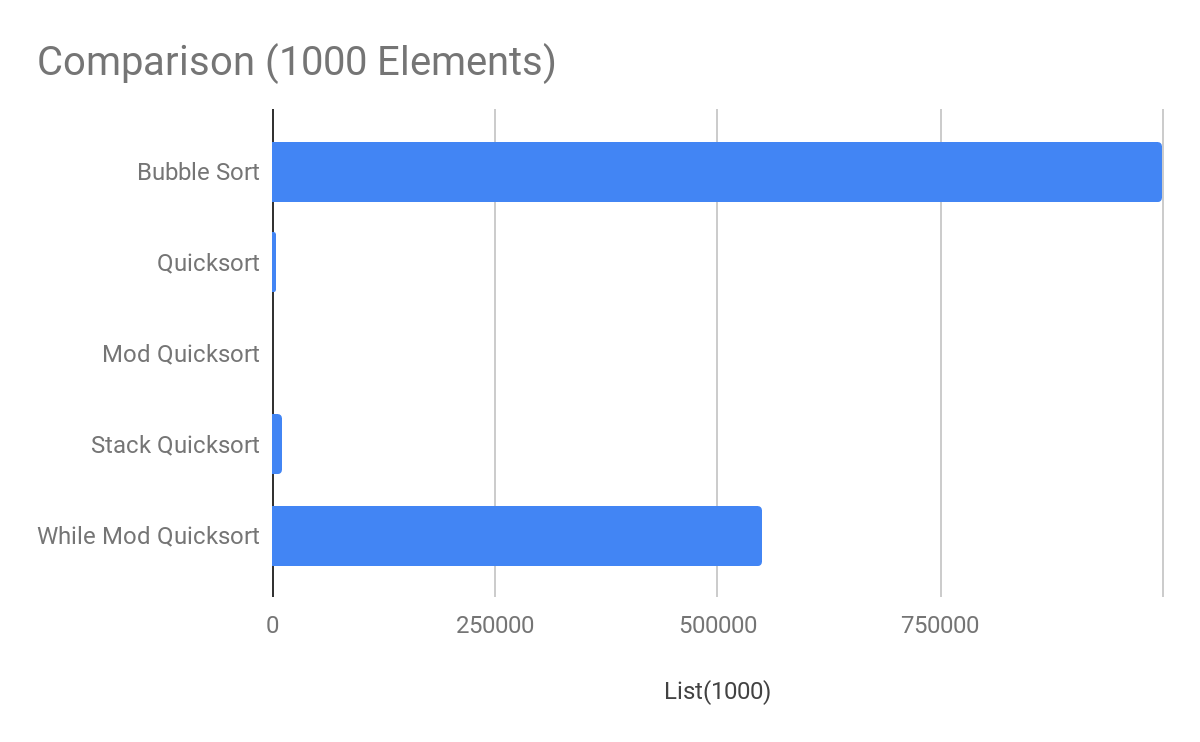
1. Given a list of 10 elements, on average, the method took 0.00070 milliseconds and 33 comparisons to find the element.
2. Given a list of 100 elements, on average, the method took 0.00251 milliseconds and 2,488 comparisons to find the element.
3. Given a list of 1,000 elements, on average, the method took 0.21067 milliseconds and 549,355 comparisons to find the element.

Of note is that the number of comparisons is

**Graphs:**







**Table:**

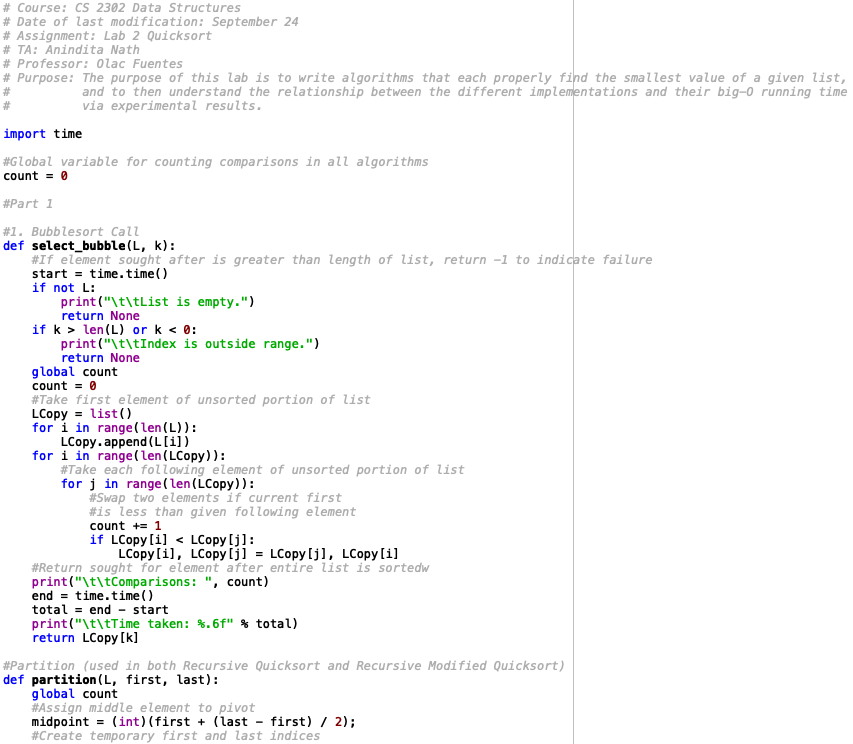
|  |  |  |  |
| --- | --- | --- | --- |
|  | List(10) | List(100) | List(1000) |
| Bubble Sort | 100 | 10000 | 1000000 |
| Quicksort | 16 | 239 | 3321 |
| Mod Quicksort | 7 | 27 | 432 |
| Stack Quicksort | 23 | 735 | 10467 |
| While Mod Quicksort | 33 | 2488 | 549355 |

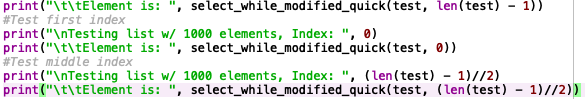
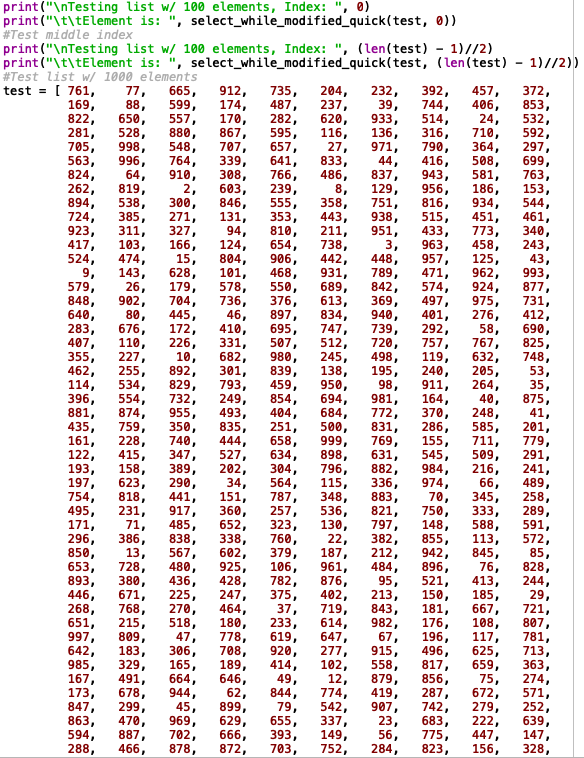
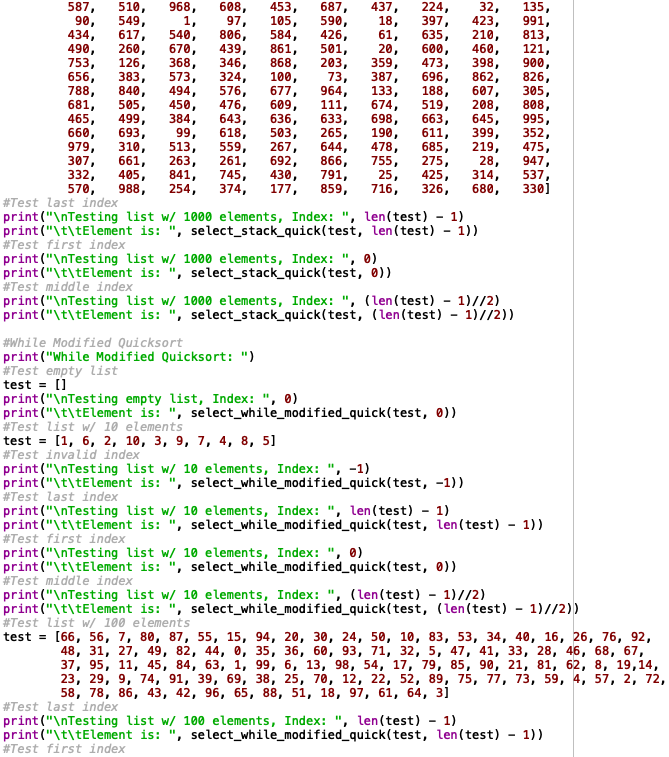
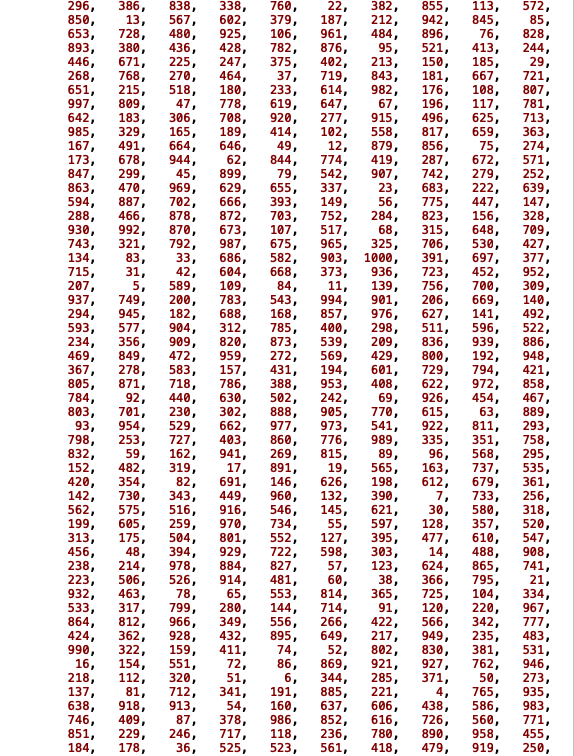
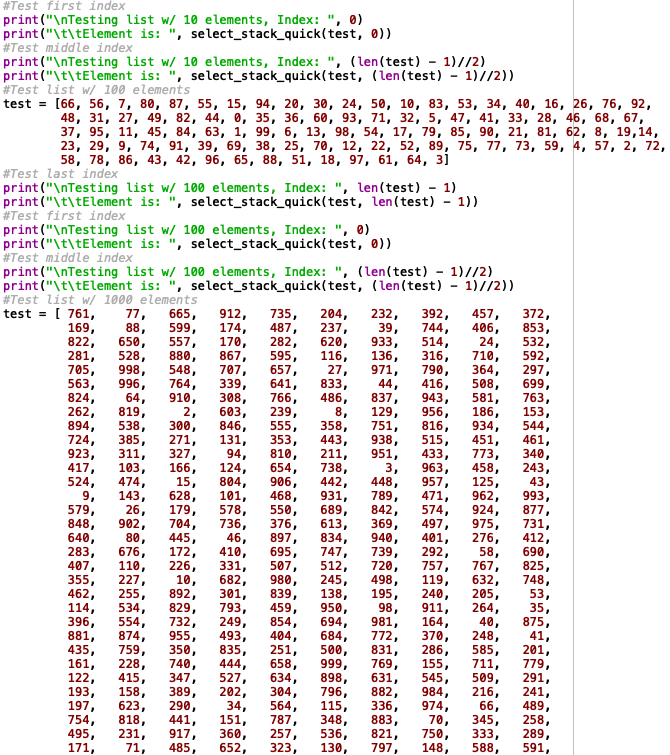
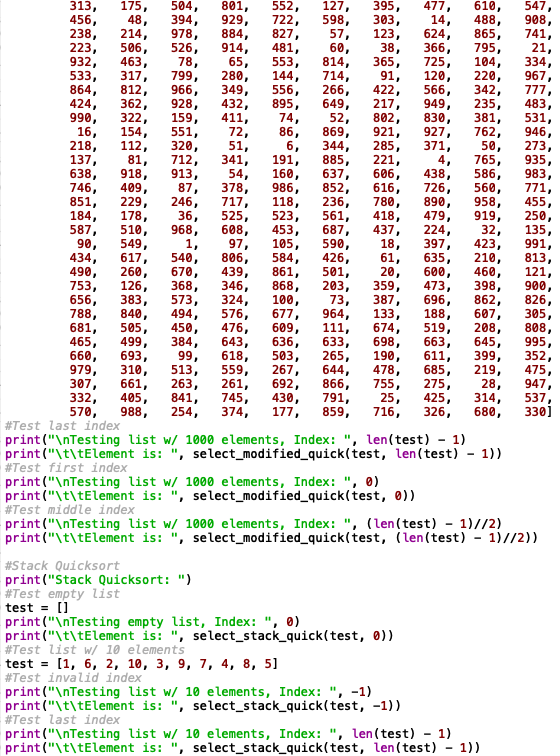
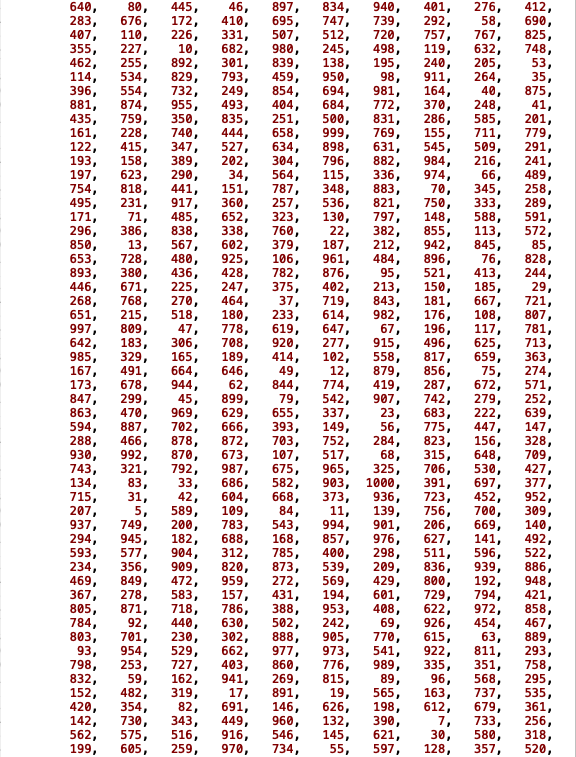
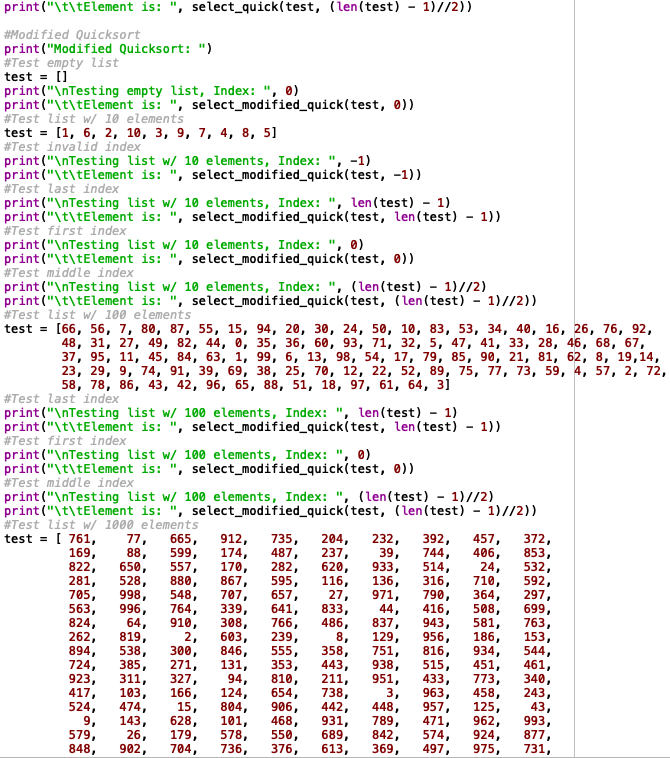
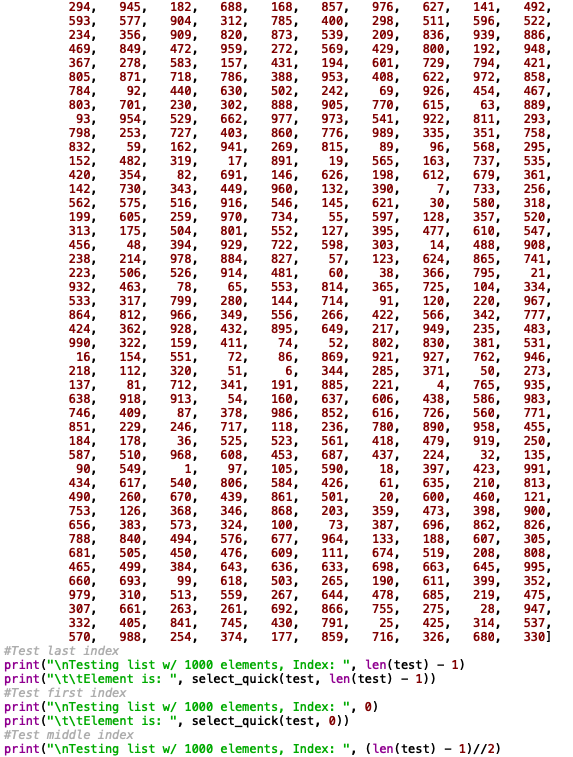
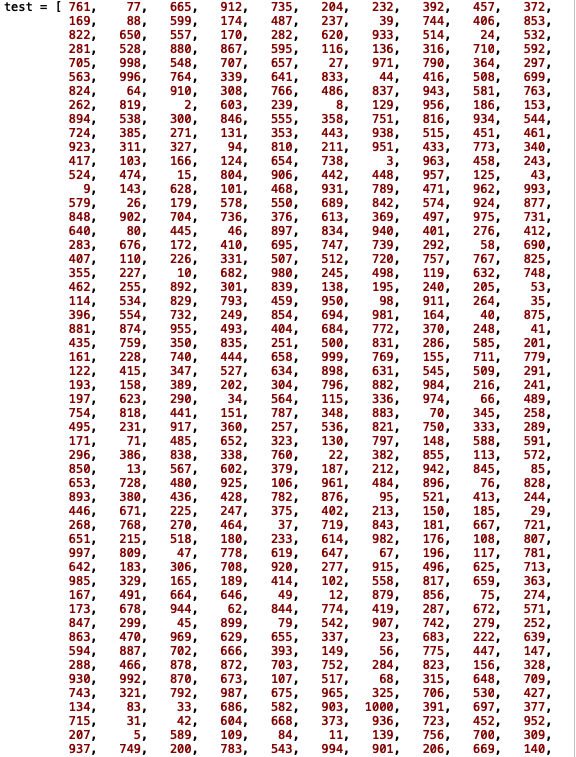
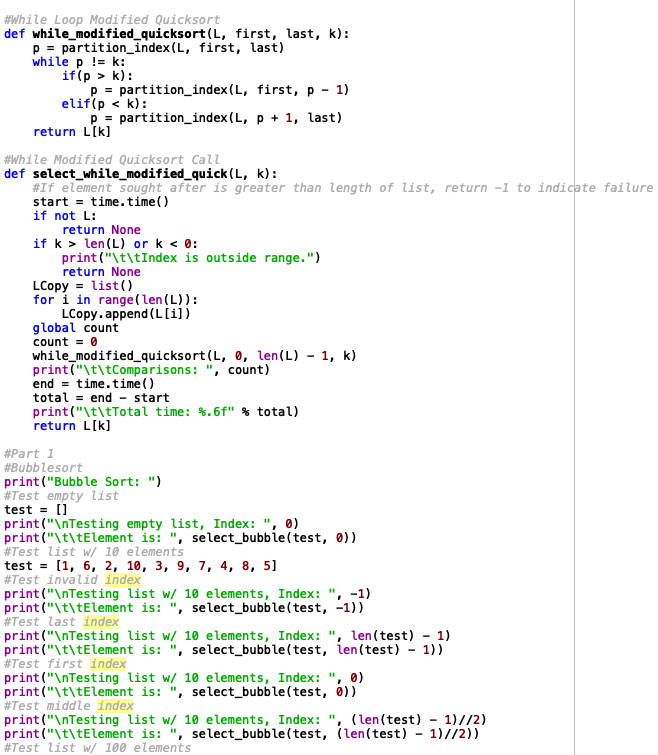
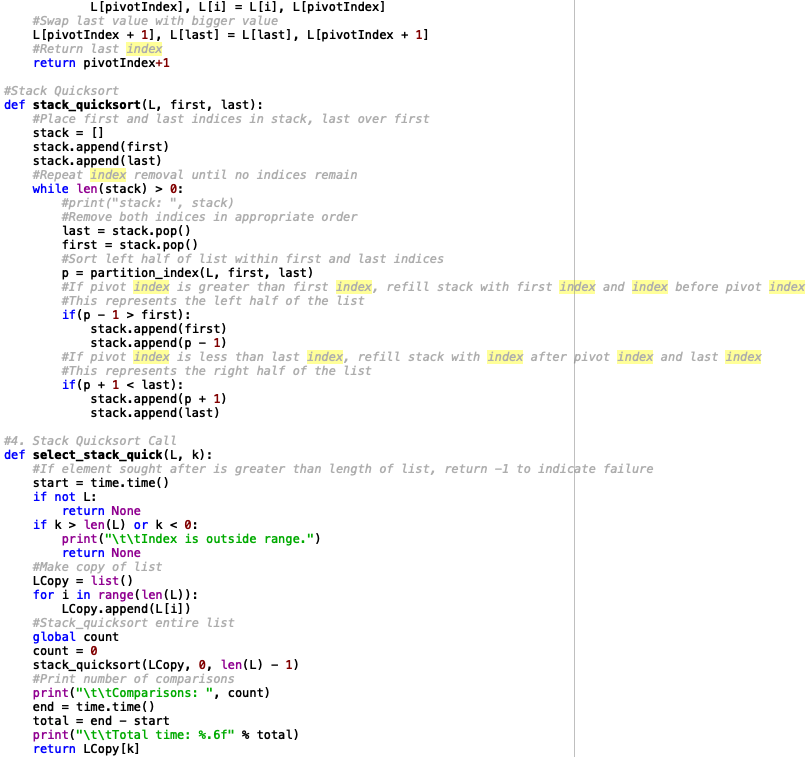
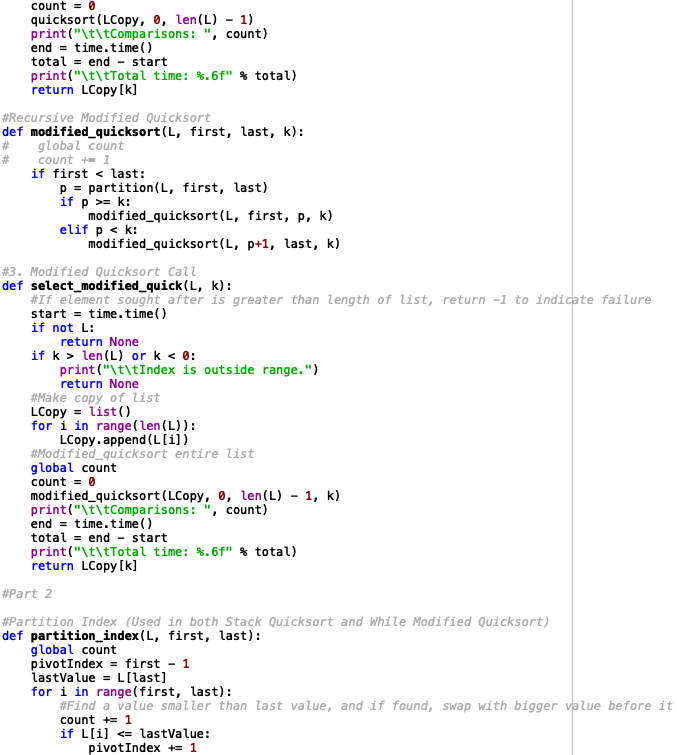
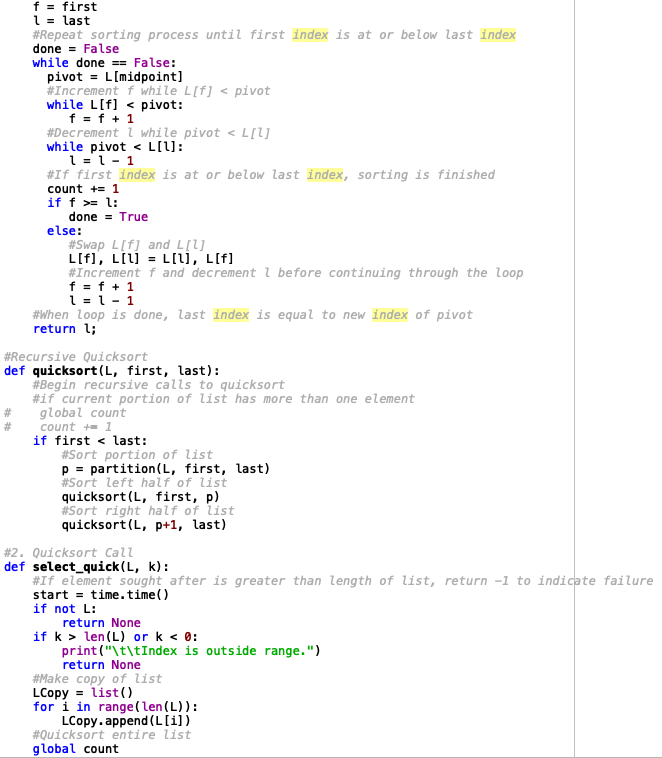
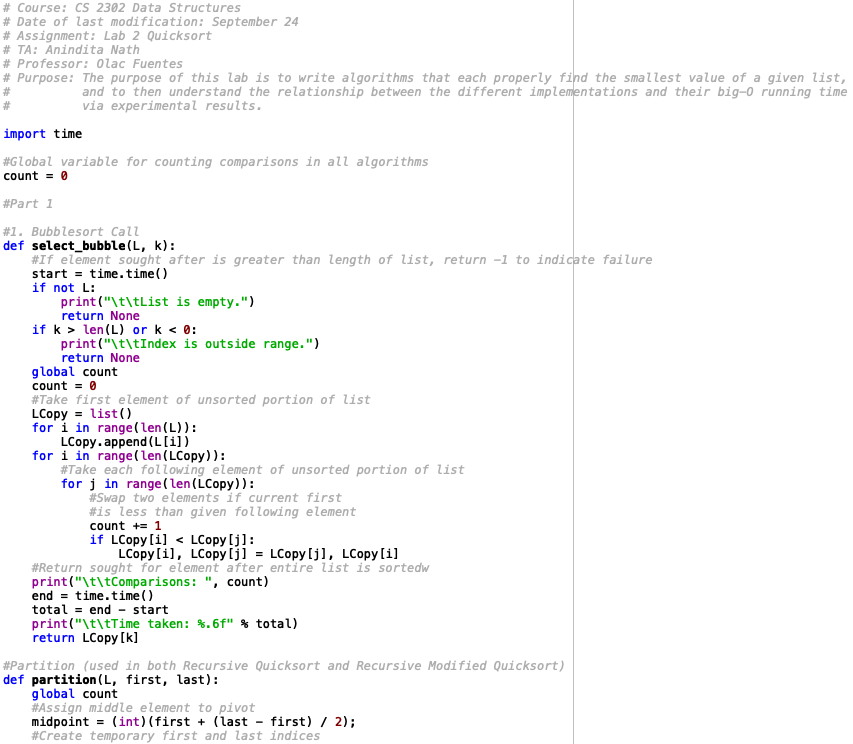
As the results clearly show, the most efficient to least efficient search methods are as follows: Modified Quicksort, Quicksort, Stack Quicksort, While Modified Quicksort, and Bubble Sort. This pattern is only exacerbated the greater the size of the list in question. The reasoning here is that each following method sorts less and less of the list, and therefore does not need to spend as much time to find a given element.

**Conclusion**

From this lab, I learned how to implement quicksort using recursion, stacks, and while loops as well as how to optimize quicksort. For quicksort and its optimization, while the partition method was the same swapping elements within smaller and smaller sections of the original list, the modified method was optimized because it only continued to recursively sort the smaller sections that were relevant to finding the desired element rather than every section. Regarding stacks and the while loop, I learned to implement a different partition method that, rather than track the elements of a subsection of the original list, instead tracks the indices at which these elements become relevant, thus sorting a list by its successive pivots. From all this, it can clearly be seen by my results that, of the various methods attempted, recursion (specifically necessary recursion explicitly) is still the most efficient way to sort a list such that a desired element can be found.

**Appendix**

****



I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class