Lab 2 – Sorting Algorithms

Jonatan M. Contreras

University of Texas at El Paso

**Introduction**

For this lab, we were asked to implement functions that used classic sorting algorithms to sort a list and then it retrieved the kth element that was requested by the user. The first implementation was to use bubble sort to sort the list before it chose the kth element from the list. The second implementation was to use quicksort to sort the list before it chose the kth element from the list. The third implementation was a variation of quicksort where the algorithm only sorted the part of the list where the kth element existed. Part two implemented the function using quicksort, but it implemented quicksort using a stack, as well as the function using the modified quicksort algorithm but only using a while loop.

**Proposed Solution Design and Implementation**

*Part 1: select\_bubble:*

For the method select\_bubble, I chose to implement bubblesort iteratively in a different function that select\_bubble would call. In that different function, I implemented bubblesort by putting a for loop that loops through the list inside a while loop that has as a condition a Boolean variable that will be changed only if no swaps were made through a loop through the entire list. This is done because only when no swaps are made in the for loop is the list sorted. Once the list is sorted, the while loops breaks and the sorting is complete. Once this was complete, select\_bubble called the bubblesort function to sort the list, and once the sorting was complete, select\_bubble would return the kth element of the list.

*Part 1: select\_quick:*

Similarly to select\_bubble, I implemented quicksort recursively in a different function that select\_quick would call. With the implementation of quicksort however, there also comes the implementation of the function partition, and for my case, I implemented a function for the selection of the pivot called median\_of\_three. The quicksort function first chooses the pivot through the median\_of\_three function. The median\_of\_three function takes the list and compares the first, mid, and last value. It uses this information and selects the median value of these three in order to ensure that the absolute worst pivot is not chosen. Then, if the pivot selected is not the first value of the list, the quicksort function swaps the first value with the pivot value. This is done to better facilitate the implementation of the partition function. Partition was implemented as follows: there is a pointer starting at the leftmost element. Then, a for loop loops through the list, comparing all the elements in the list to the pivot. If the element that is being compared is less than the pivot, the pointer at the leftmost element moves forward, then the element that was less than the pivot is placed at this pointer place. This is slowly building the left part of the list. Wherever the pointer stops is where the correct place for the pivot will be. The algorithm ends by swapping the element at this point with the leftmost element since we know that’s where the pivot is. Partition returns this index so that quicksort can then use it to recursively call quicksort on the sub lists to the left and right of that pivot index. Once the list is sorted, select\_quick returns the kth element.

*Part 1: select\_modified\_quick*

For select\_modified\_quick, I created a modified\_quicksort function that uses the same quicksort algorithm described above. However, the key difference is that it makes a comparison for the recursive calls. If k is equal to the pivot index returned by the partition function, we know that the pivot is in its correct place, and thus we immediately return the element from the list at that index, i.e. L[k]. If k is less than the pivot index returned by the partition function, we only make a recursive call to the left sub list since we know that kth element won’t be in the right sub list, rendering a recursion call to it unnecessary. Similarly, if k is greater than the pivot index returned by the partition function, we only make a recursive call to the right sub list.

*Part 2: select\_quick with a stack*

To implement select\_quick with a stack, I created a separate quicksort function that only used a stack. It was useful for me to think about pushing items onto the stack as a recursive call. The algorithm uses an empty list as a stack since its append and pop functions provide the last-in-first-out structure. It then pushes the left bound and right bound of the list onto the stack. The algorithm then goes into a while loop that checks to see if the stack is empty. If it isn’t, it pops out the two elements in the stack. It then completes a boundary check. If the difference between these two elements is less than 0, then this means that there is no elements in the list to sort. Once this check occurs, it follows the quicksort algorithm used in the previous functions that used quicksort. But instead of recursively calling the algorithm onto the left and right sub lists at the end of the while loop, the algorithm pushes the left bound and right bound of the left sub list onto the stack, then pushes the left and right bound of the right sub list onto the stack. This ensures that when the algorithm pops the two elements, they are appropriately matched as left and right bounds. The algorithm continues doing this until the stack is empty.

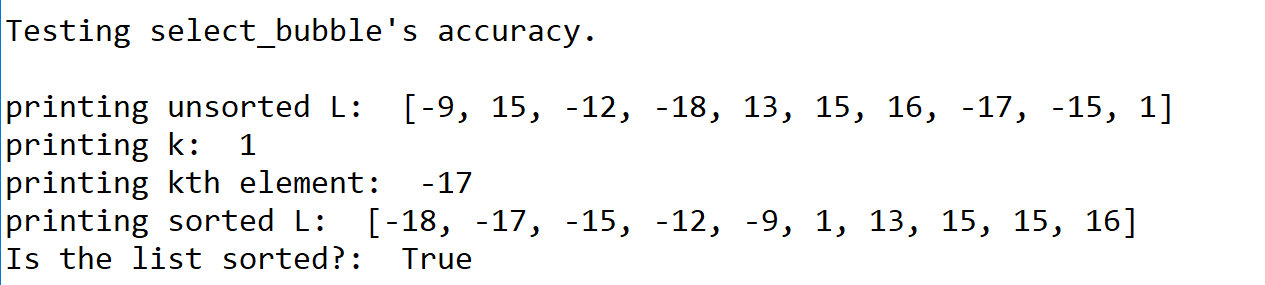
*Part 2: select\_modified\_quick with a while loop*

For this algorithm, I took my original algorithm I implemented for these purposes and used a while loop that checked to see that the left boundary is still less than the right boundary. If so, it performed quicksort with the same checks with regards to k and the pivot index.

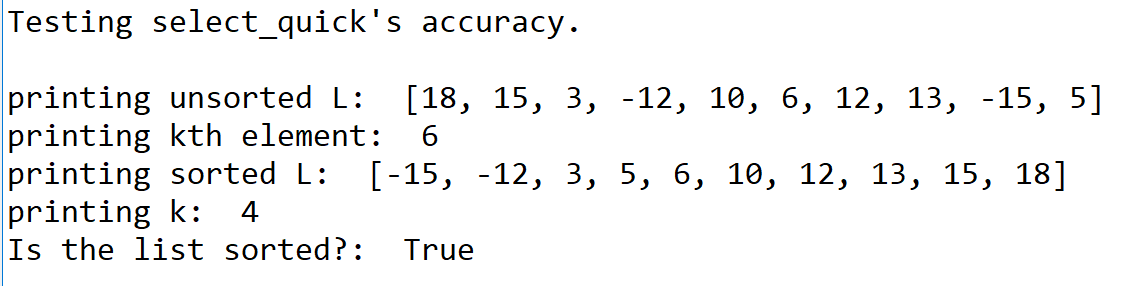
**Experimental Results**

*First Test: Accuracy*

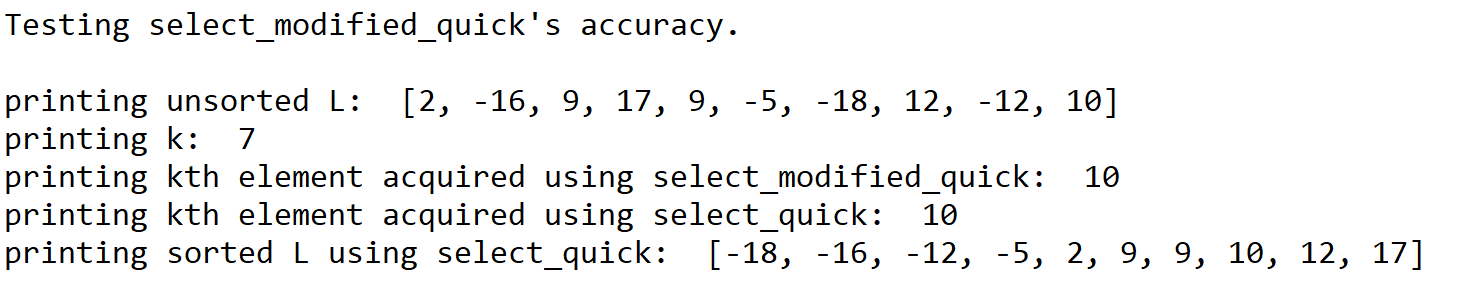
I first tested the sorting algorithms with a small list size of 10 with randomized elements from -20 to 20 in order to check its accuracy in A.) properly sorting the list and B.) returning to correct kth element. This can be confirmed visually with a small list size of 10. The following screenshots are screenshots of accuracy tests I ran with these algorithms. They will have the unsorted randomized list printed, the k that was randomly chosen for that run, the kth element the function returned, and the final sorted list to confirm its sorted nature. Since the modified\_quick functions don’t sort the full list, a comparison kth element is provided using the select\_quick function. A sorted list sorted by select\_quick is also provided for comparisons.



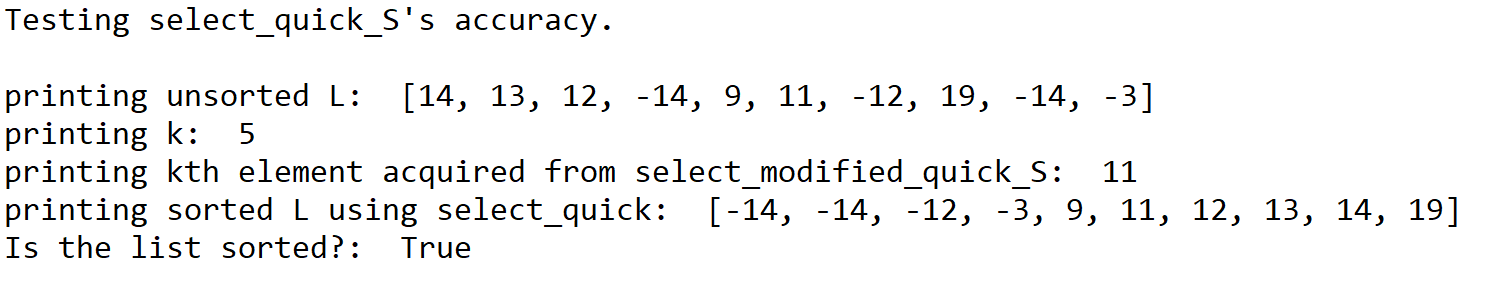
Test of select\_bubble’s accuracy.



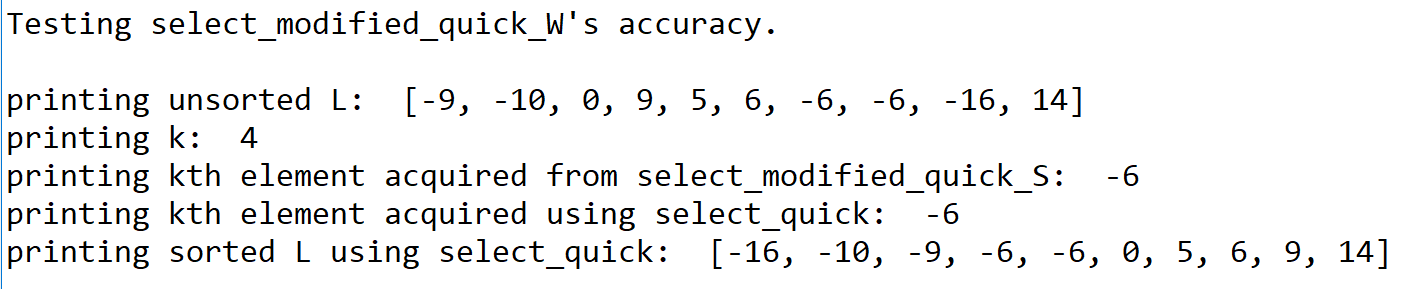
Test of select\_quick’s accuracy.



Test of select\_modified\_quick’s accuracy.



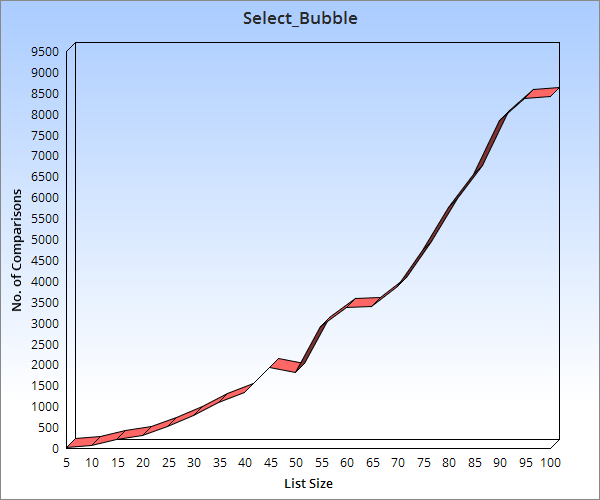
Test of select\_quick\_S’s accuracy.

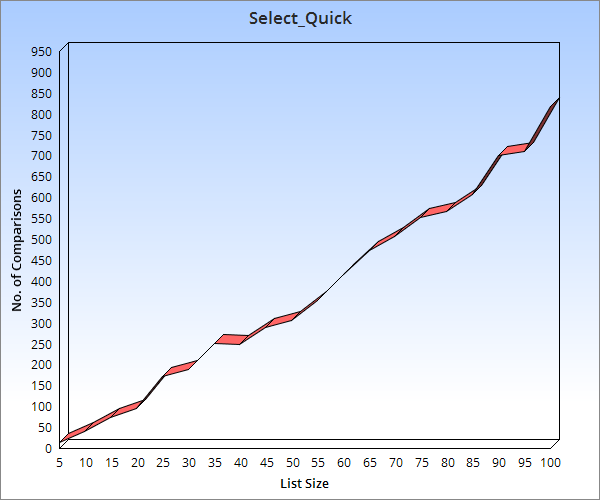


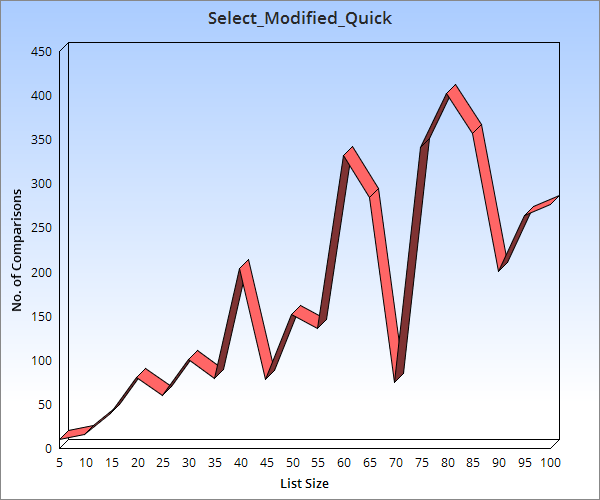
Select\_modified\_quick\_W’s accuracy.

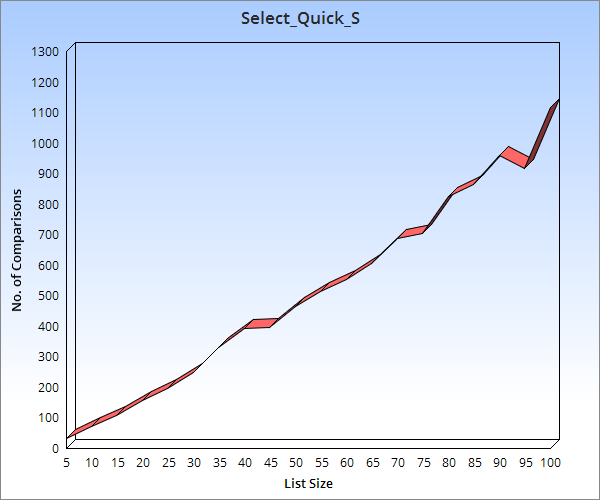
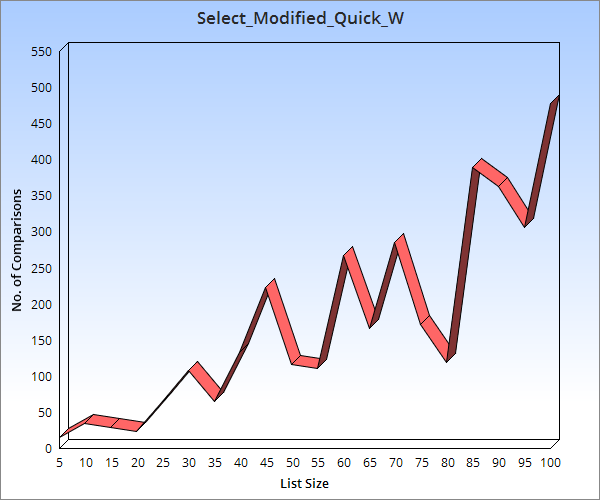
*Second Test: Comparisons Made*

To gather enough data points to illustrate the algorithms’ behavior graphically, I ran the algorithms with list sizes from 5 to 100 with an interval of 5. They are titled by function name.









*Big O and In-Practice Running Times Analysis*

|  |  |
| --- | --- |
| Algorithm | Big O |
| Select\_bubble | N2 |
| Select\_quick | N logN |
| Select\_modified\_quick | logN |
| Select\_quick\_S | N logN |
| Select\_modified\_quick\_W | logN |

The following is a table presenting the Big O running times of the algorithms implemented here.

To demonstrate the magnitude of differences and similarities, I ran the program with extreme list sizes like 1,000 and 10,000. The following is a table that presents the algorithms on the first column, then the following columns will have the various list sizes used for the testing purposes. The values provided will be the number of comparisons the algorithm made during one run.

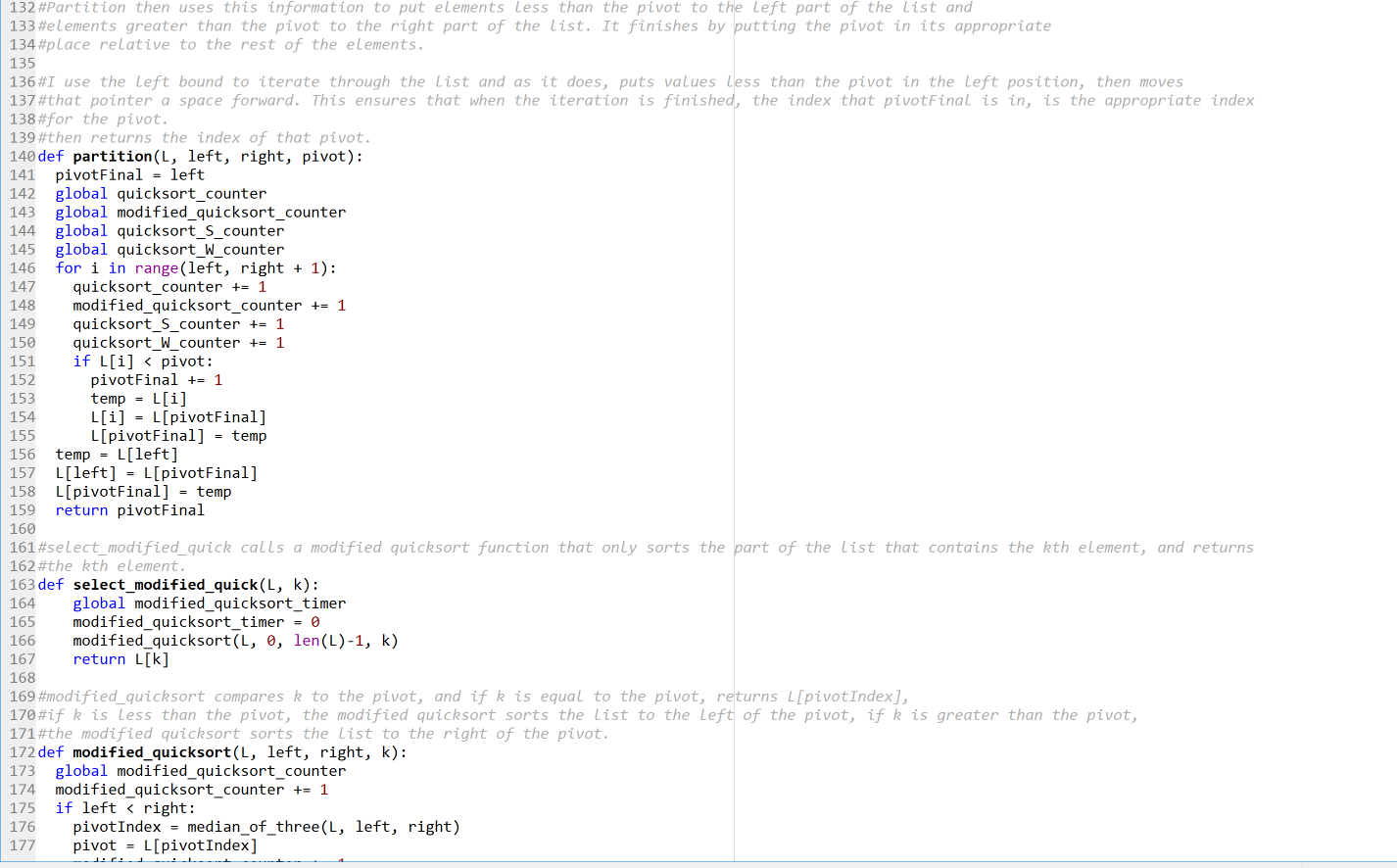
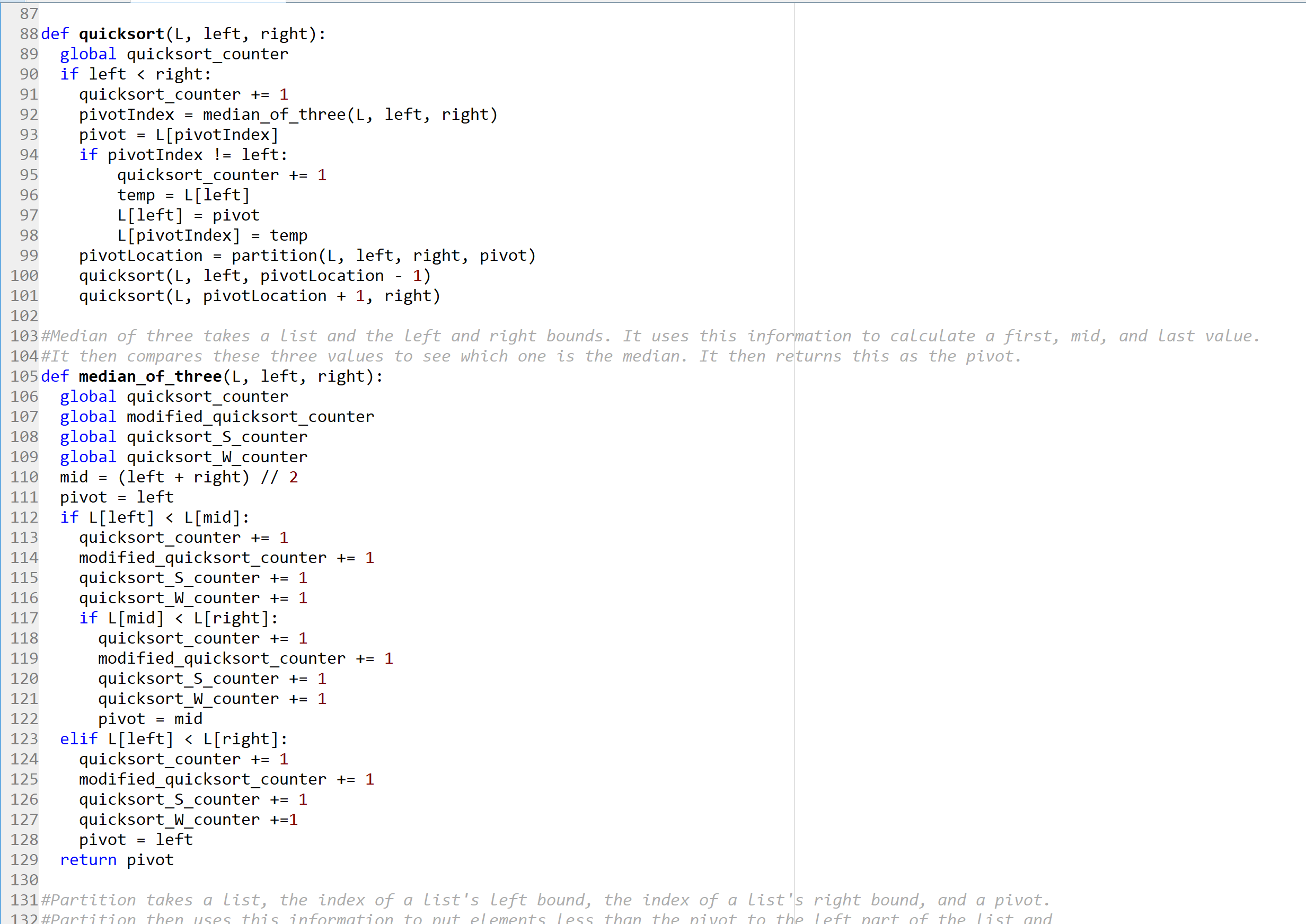
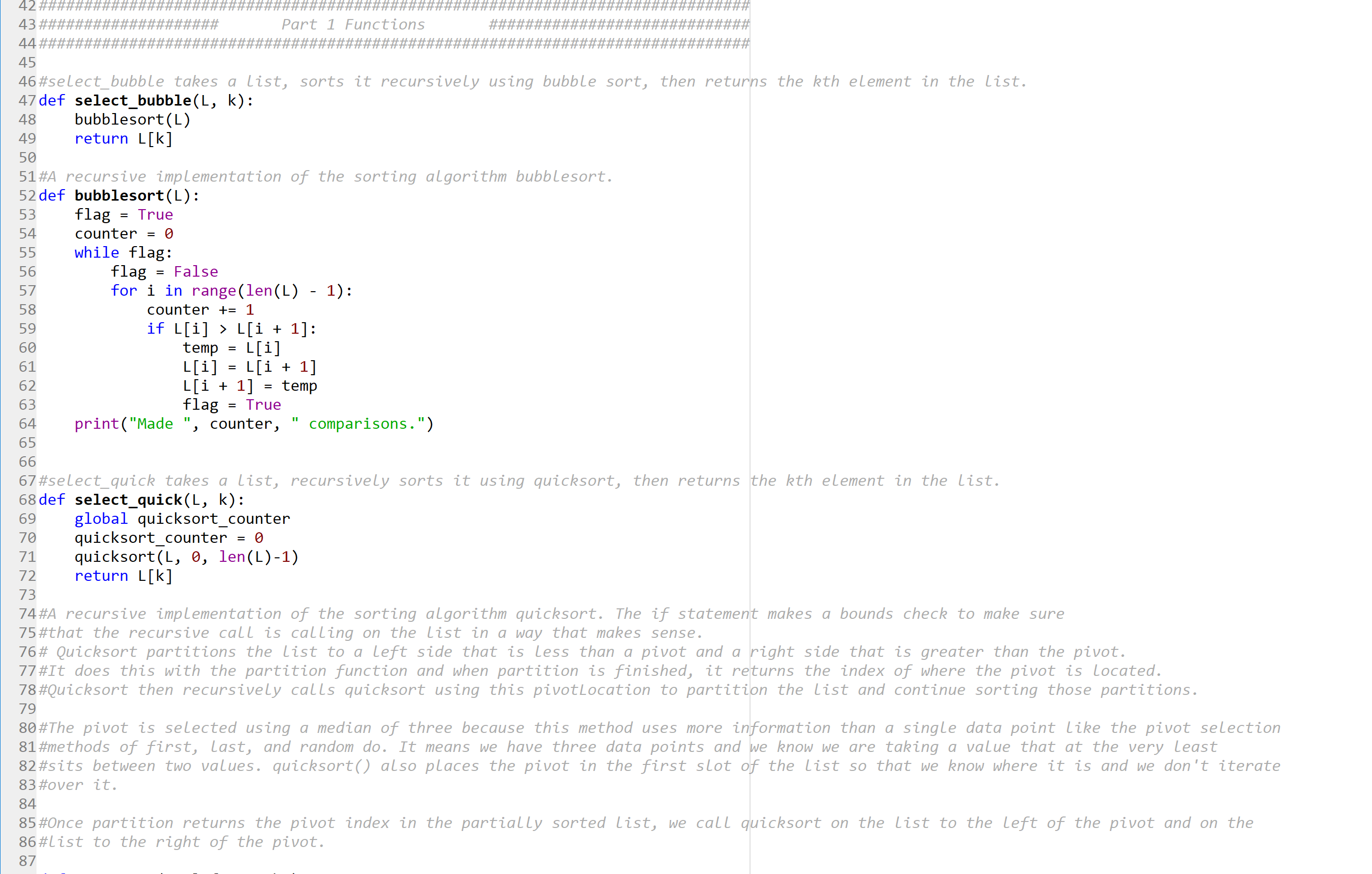
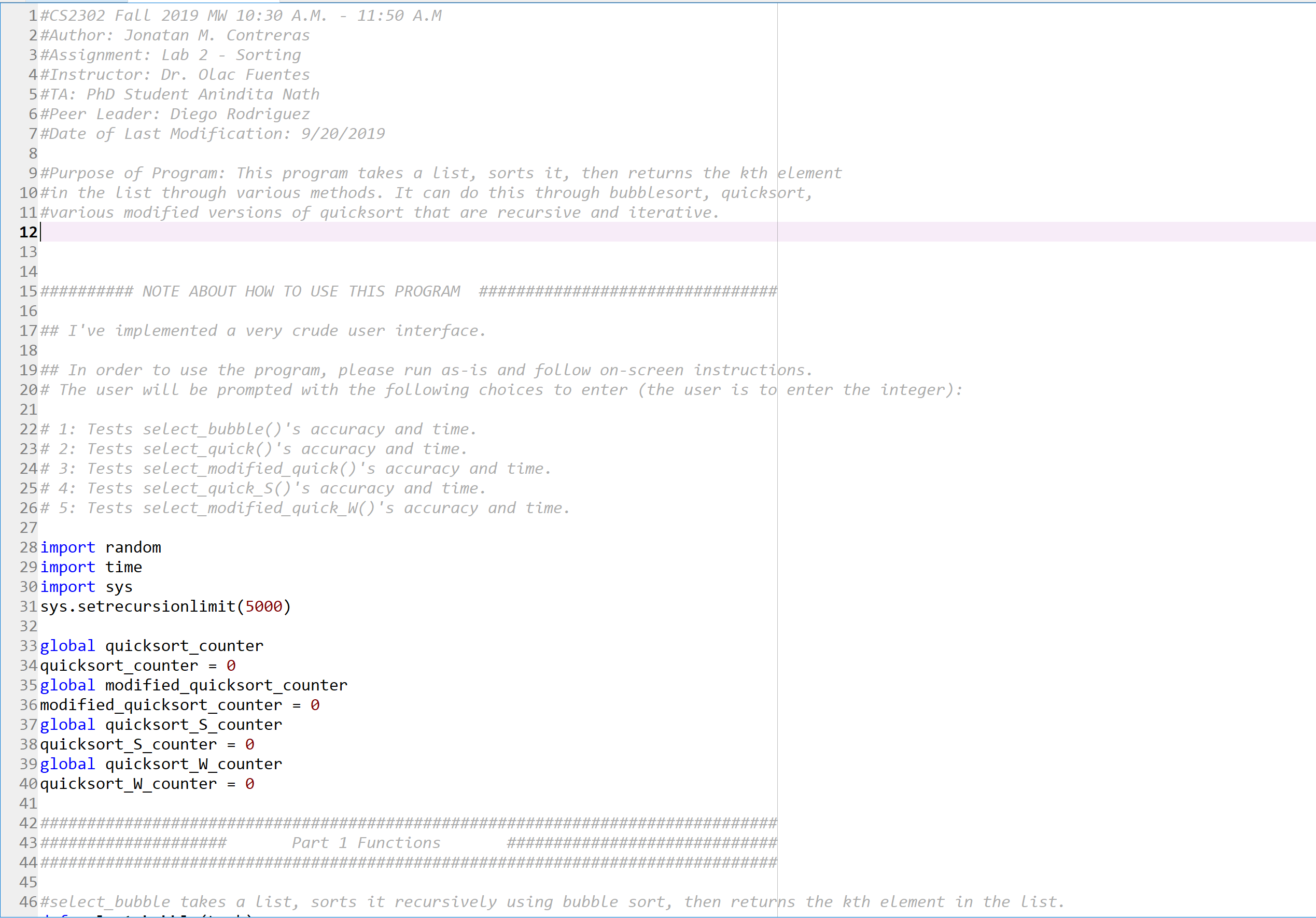
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 10 | 100 | 1,000 | 10,000 |
| Select\_bubble | 90 | 8,118 | 948,051 | 96,990,300 |
| Select\_quick | 43 | 880 | 19,800 | 1,309,072 |
| Select\_modified\_quick | 19 | 389 | 2779 | 34091 |
| Select\_quick\_S | 68 | 1,039 | 23,602 | 1,325,659 |
| Select\_modified\_quick\_W | 30 | 220 | 3394 | 34528 |

My select\_bubble method seems to be performing pretty close to N2. However, my select\_quick methods are not performing as well as they should be. After further analysis, I realized this is due to the way I designed my partition function. There is a for loop in the partition function that causes it to go through more elements than it needs to, causing the time complexity to increase. Another factor that adds to the comparisons is my pivot selection method. The median of threes completes three comparisons for every pivot selection. Thus, it adds three more comparisons per recursive call, causing the comparisons to increase substantially. However, it still has not reached the worst-case scenario of N2.

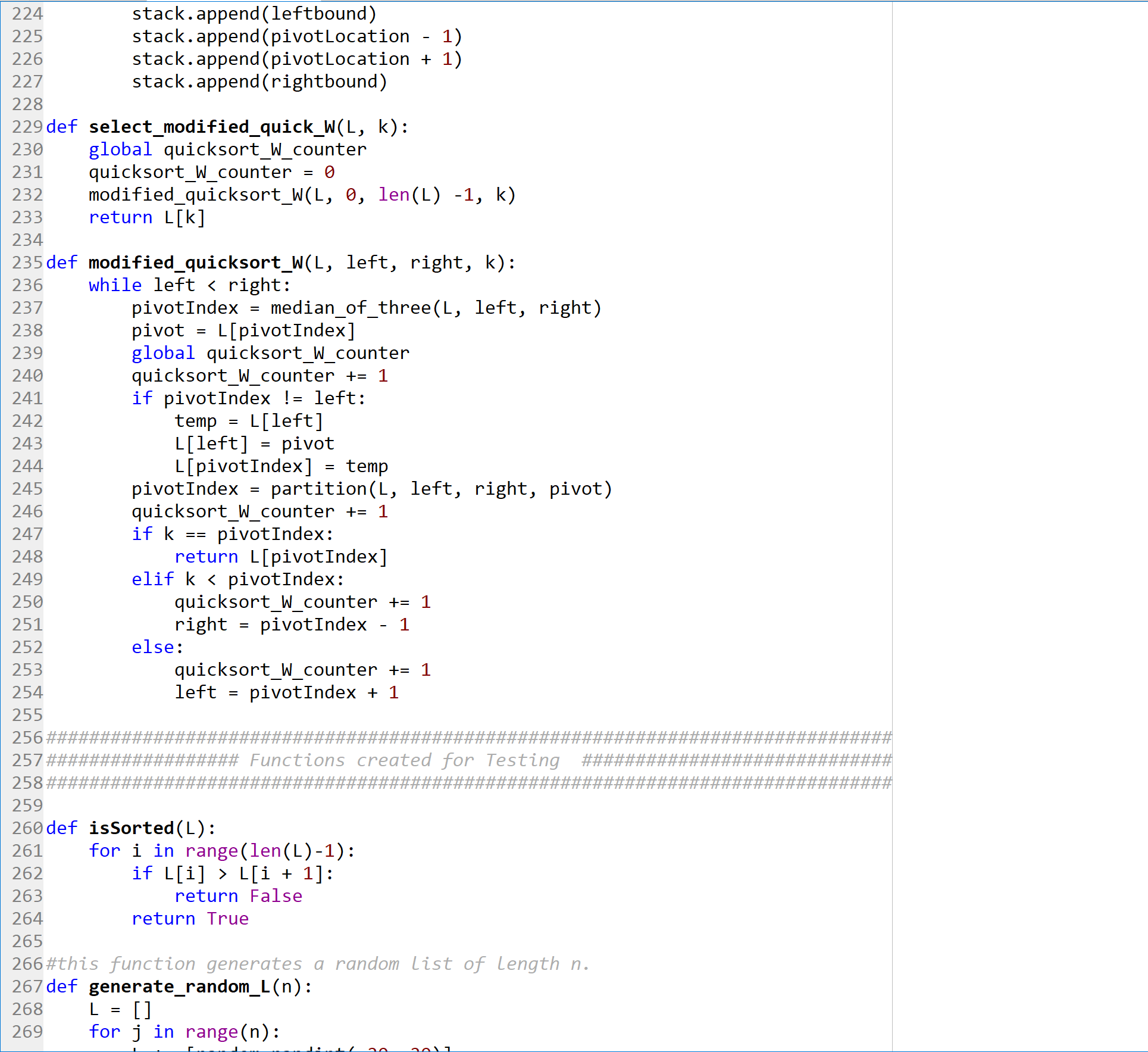
There was also a lot of variance in the results for the modified\_select\_quick functions. This is a combination of two things: first, my partition function and pivot selection function could be better optimized. Second, the way modified\_select\_quick works is that it is also dependent on the k. Thus, if it were to find the kth element quickly, it wouldn’t run for as long. But if it needed to continue dividing the list in two until it found the final kth element, it would run for longer.

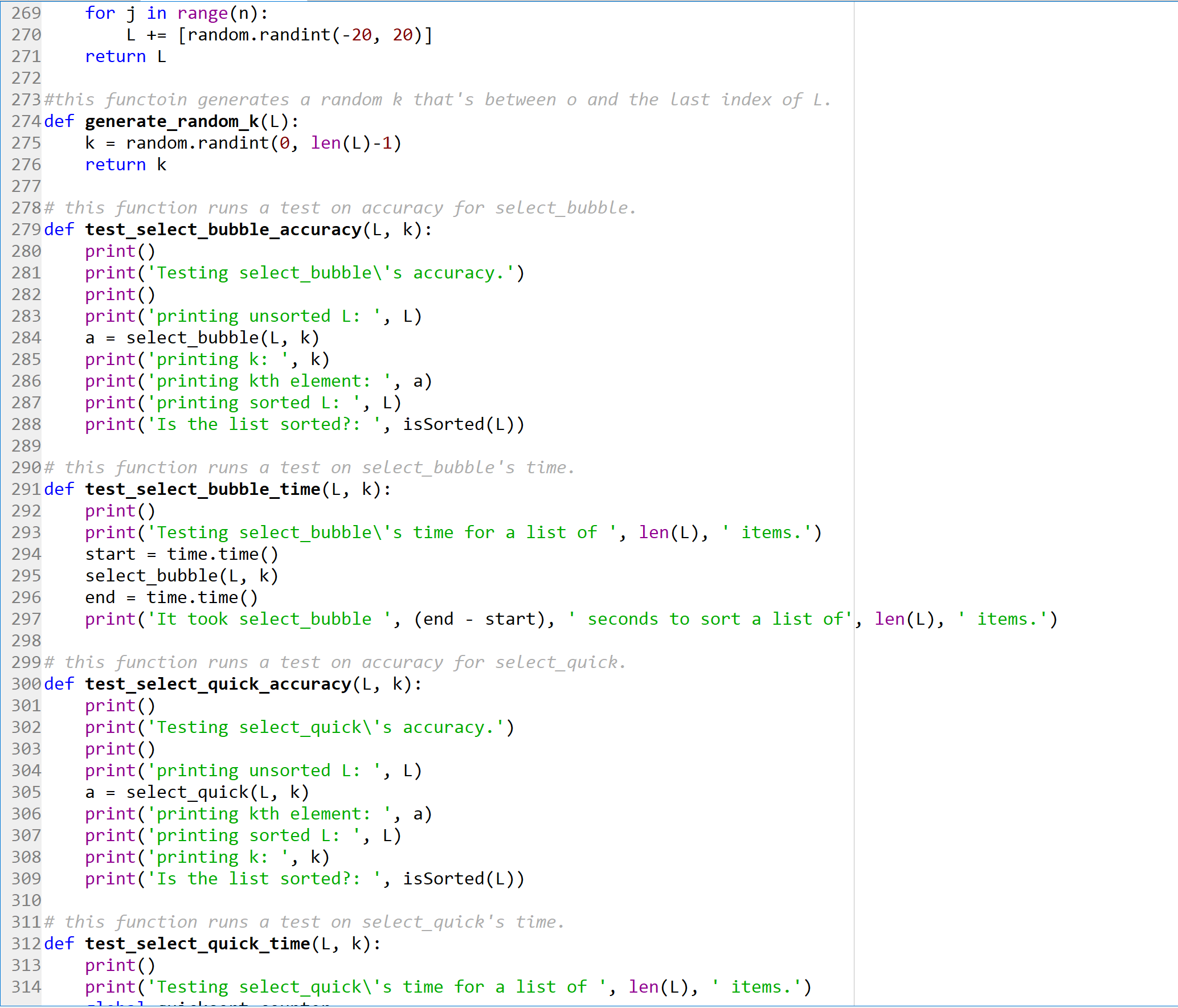
**Conclusion**

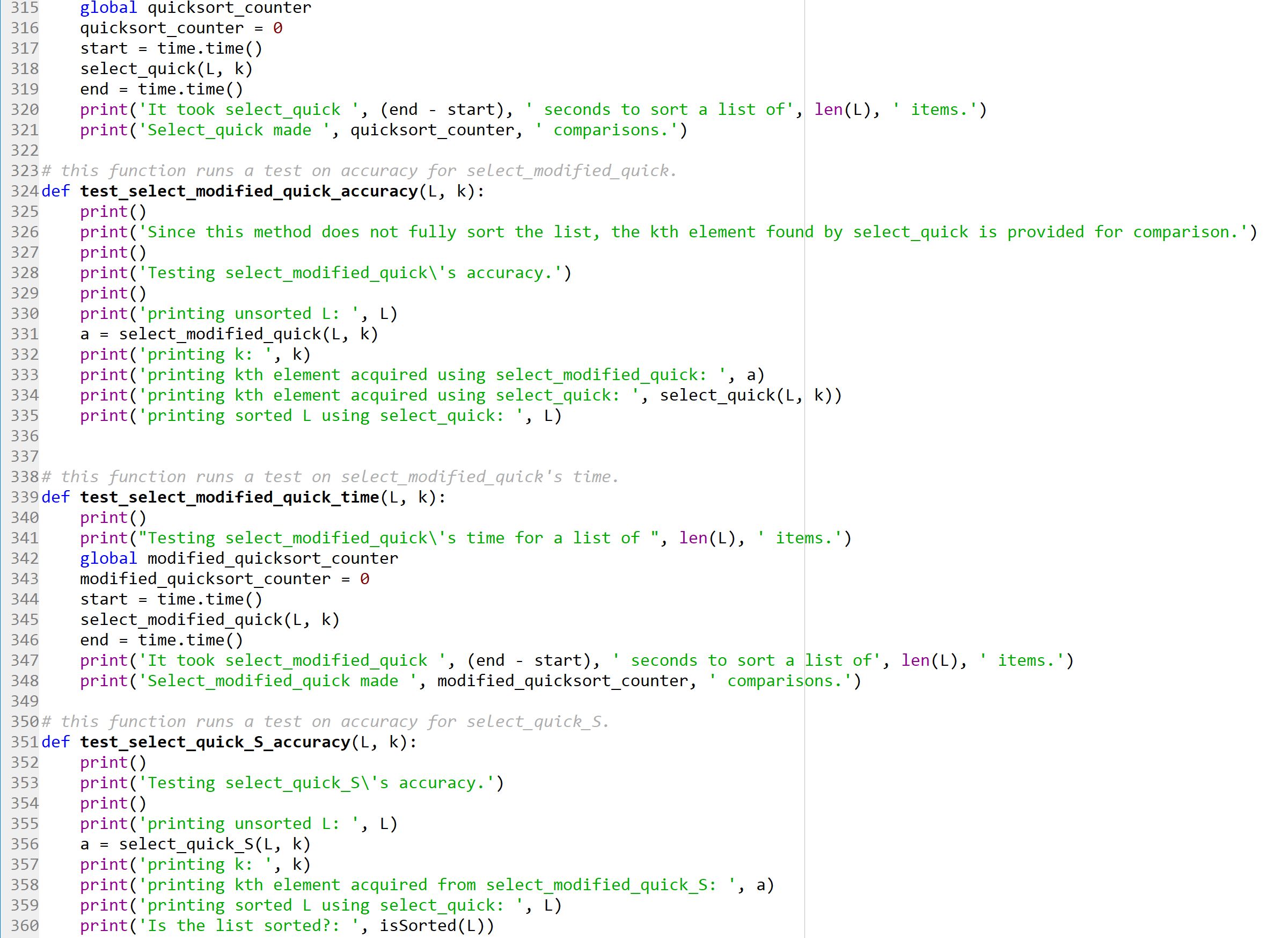
In this lab, I have learned that implementing an algorithm to match its big O notation running time is important. Even though an algorithm is considered the best in running time, it’s important that you make sure that it is at its optimal. Secondly, I learned that there are further optimizations to algorithms, such as the modified select quick algorithm we implemented. Finally, I also learned that it is important to be able to implement a recursive function using a stack, since this is what recursion is doing in the background.

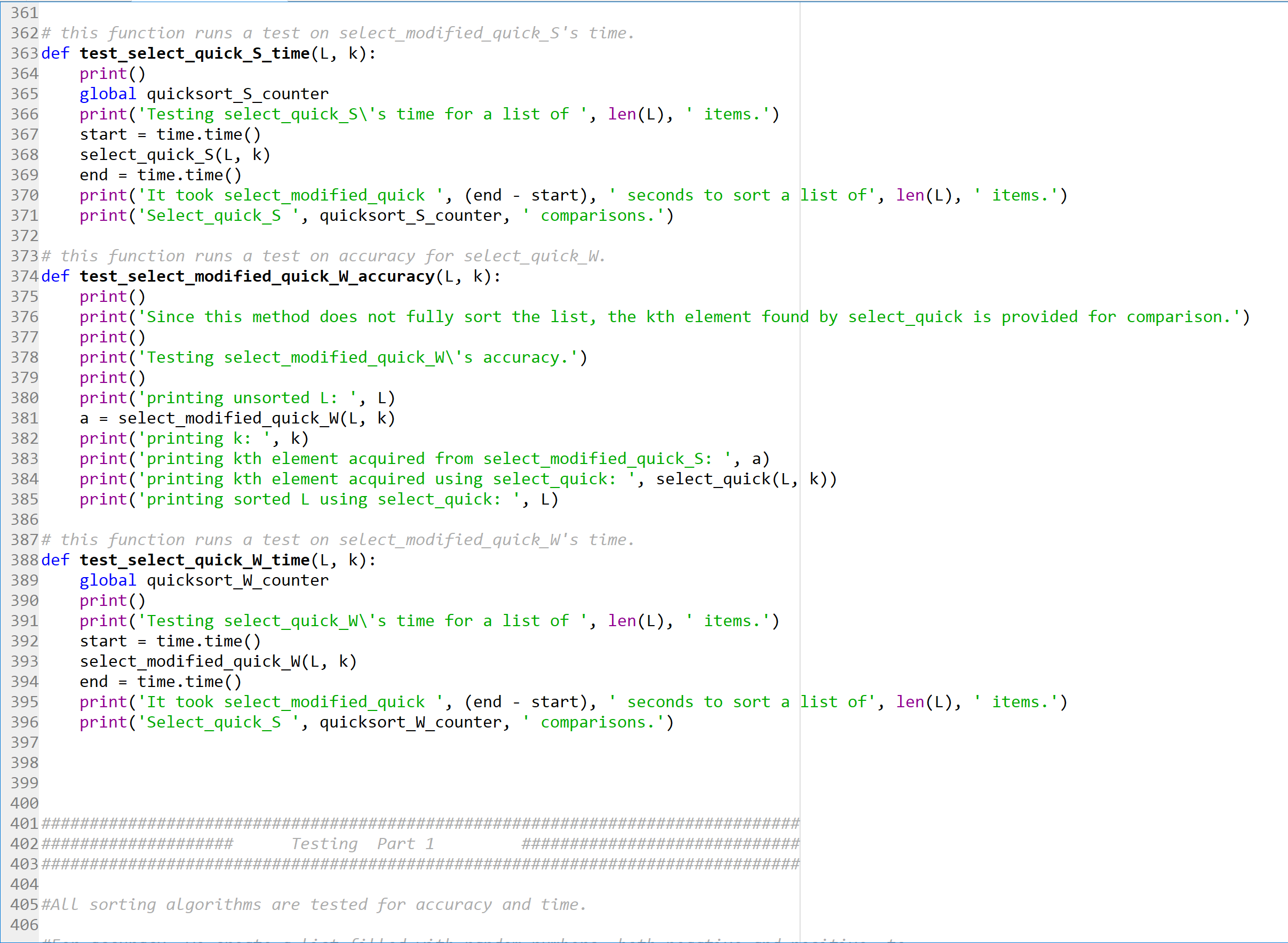
**Appendix – Source Code**



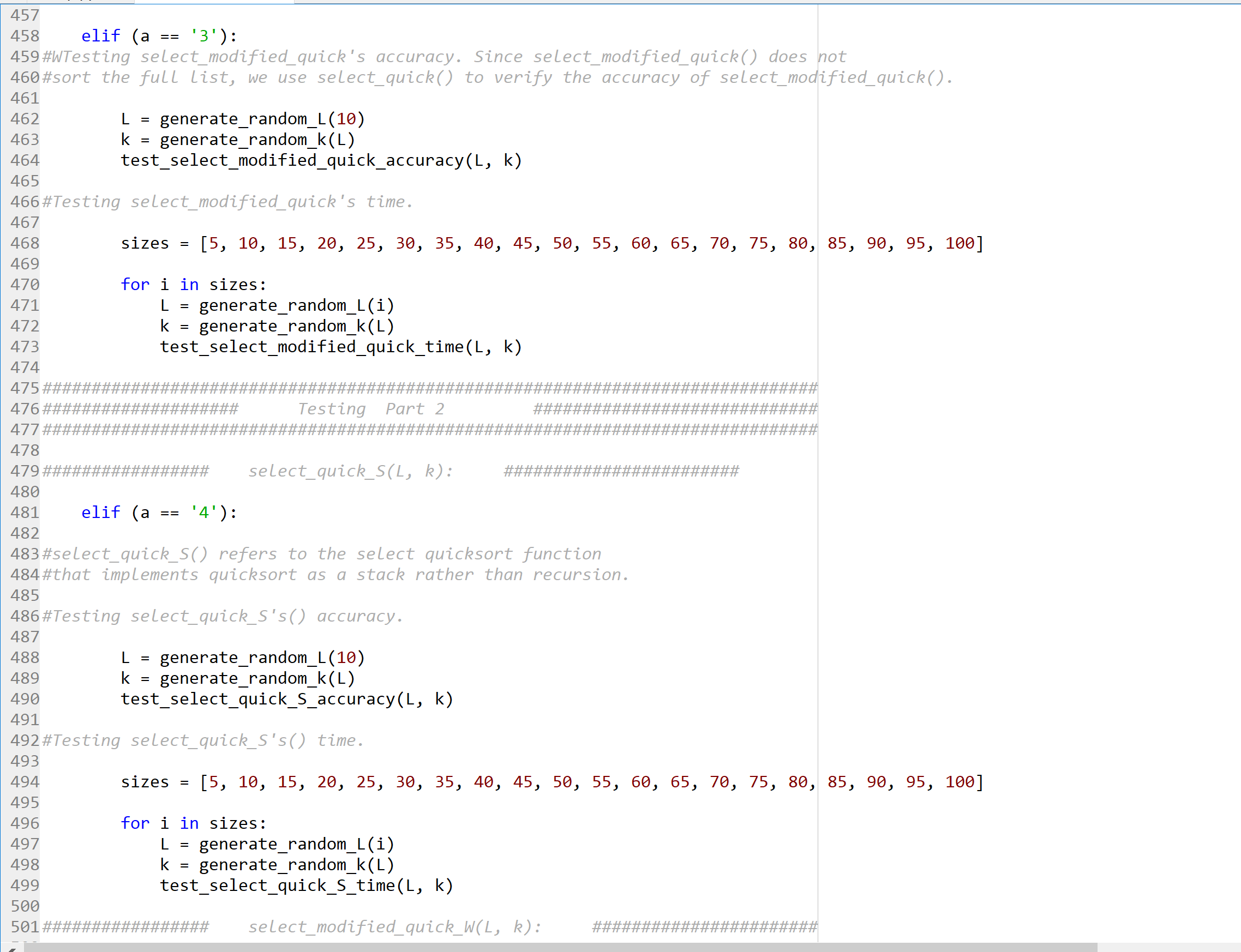


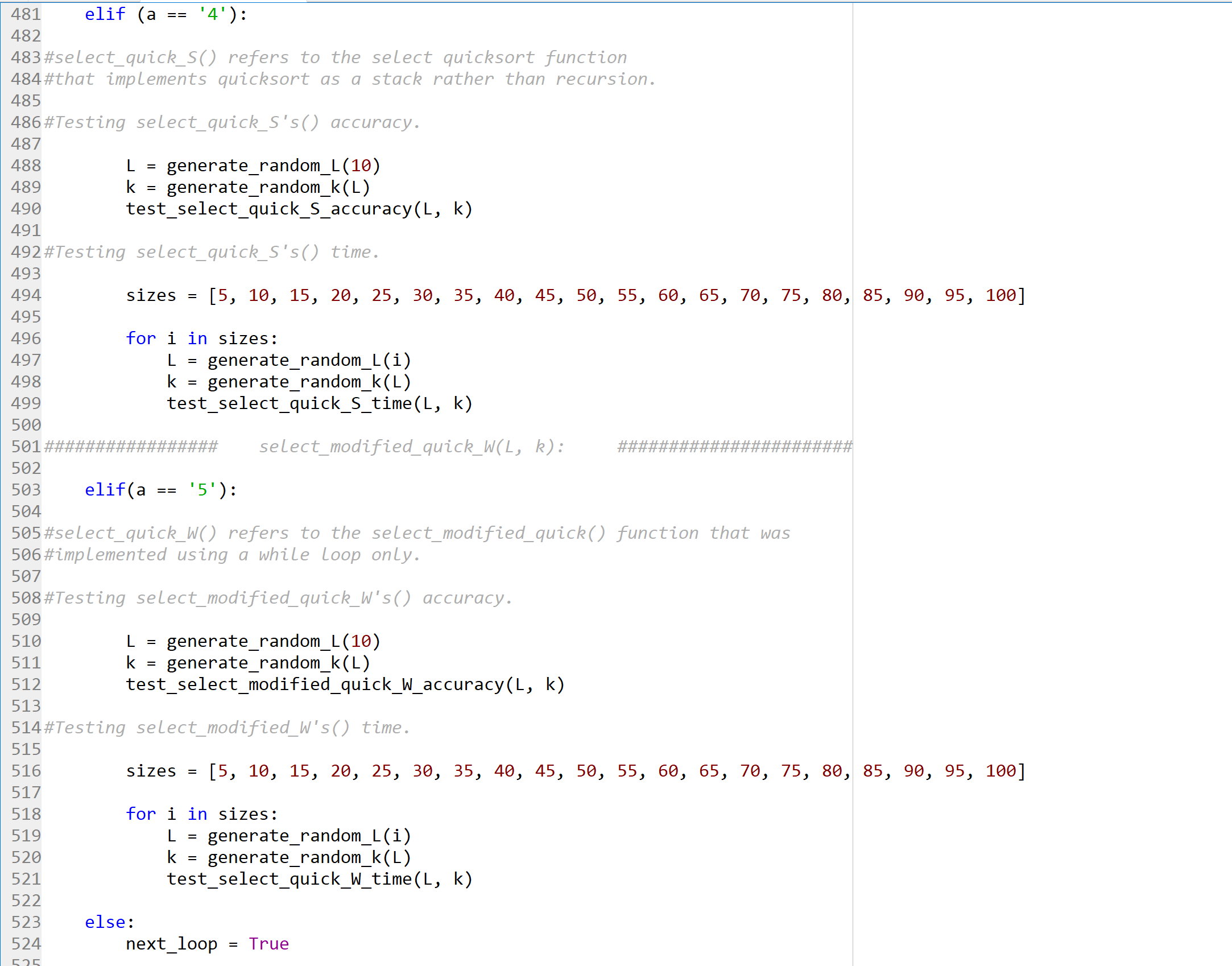












**Academic Honesty Certification**

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.

Jonatan Contreras