**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #2**

Due: 09/19/2019

Professor: Olac Fuentes

TA: Anindita Nath

Programmer: Miriam Olague

**Introduction**

` For this lab, we were asked to work with Quick Sort and find the kth element the user enters within the sorted list. For this lab, it is key to remember how Quick Sort works and different ways to sort the lists. The main objective of this lab is to get a better understanding about Quick Sort and find different ways to solve a problem.

**Proposed Solution Design and Implementation**

**Part #1 select\_bubble:**

For this operation, I used a code I created on Java. For this part, I iterated through the entire list, and continuously compared an element to the next one. If the next element was greater than the next one, I swapped the elements. I then printed out the sorted list. After printing the sorted list, I returned the kth position the user entered. On average and worst case scenario, Quicksort takes O(n^2). The number of comparisons that it makes is  *n*(*n* - 1)/2.

**Part #1 select\_quick:**

For this operation, I iterated through the entire list that was between low and high inside of a method called ‘partition’. I compared elements to the pivot to see if they were smaller than the pivot. If they were, I swapped the elements. Inside of ‘select\_quick’, I first compared if low was smaller than high, if they were, I called partition and made recursive calls inside of the method, so partition was called continuously until the list was sorted. I then returned the kth position the user entered. The worst case for this is O(n^2), however, on average, it takes O(n log n) comparisons to sort n. The number of comparisons that it makes on average is O(n^2) and worst case scenario is O(n log n)

**Part #1 select\_modified\_quick:**

For this operation, I created a method called ‘select\_modified\_firstsplit’. Inside of this method, I created empty lists in which I split the given list into two sub lists, one containing smaller elements than the pivot, one that has elements equal to the pivot, and one that has elements greater than the pivot. I made the first element of the list the pivot and iterated through the entire list. I compared all elements to the pivot and appended them to ‘left’, ‘equal’, or ‘right’. I added ‘equal’ to ‘right’. I compared the length of the list ‘left’ with ‘k’ so that way I could determine if k was inside of ‘left’ or inside of ‘right’. If k was greater than the length of left, I returned ‘right’, if not, I returned ‘left’.

Inside of the method called ‘select\_modified\_quick’, I compared low and high, if low was smaller than high, I called the method ‘partition’ (as mentioned before in **Part #1 select\_quick**). This method is almost the same as Part #1 select\_quick, the difference is that it was now given the list that contains ‘k’. I subtracted k minus the length of the list ‘left’ so I could determine the true position of ‘k’. After that, I then returned the element at position k.

The worst case for this is O(n^2), however, on average, it takes O(n log n) comparisons to sort n. The number of comparisons that it makes on average is O(n^2) and worst case scenario is O(n log n)

**Part #2 quick\_stack:**

For this operation, I created a stack that was the size of the list, in which I saved the elements top, low, and high. I then iterated through the list. I kept saving high, low, and called partition. I created an ‘if’ statement, in which I checked if the pivot was greater than low, and if it was, I saved low, and decreased the pivot. In another ‘if’ statement, I did the same but incremented the pivot. I printed the sorted list and returned the kth element. The worst case for this is O(n log n). The worst case scenario for the number of comparisons that it makes is O(n log n).

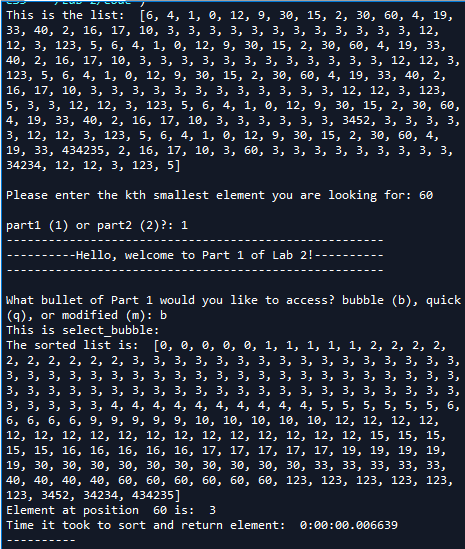
**Part #2 while\_loop:**

For this operation, I was unable to sort the array.

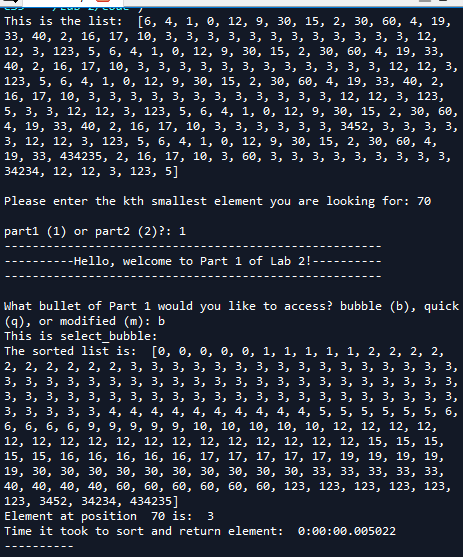
**Experimental Results**

**Part #1 select\_bubble:**

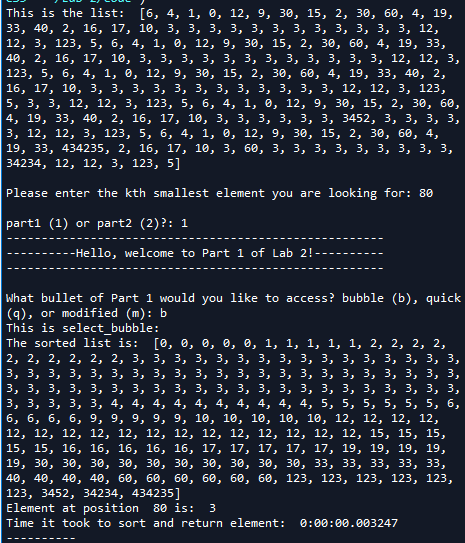
For this operation, I did not have a lot of trouble with since I understood how to do it. I had Quick Sort but in Java. I constantly printed the list to check if it was being sorted inside the for loop.



Case 1: Input = 60



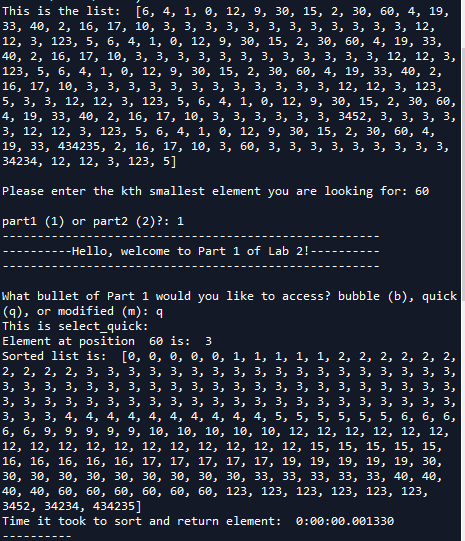
Case 2: Input = 70



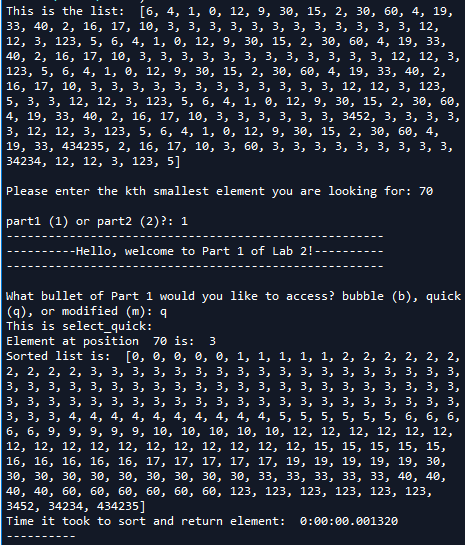
Case 3: Input = 80

**Part #1 select\_quick:**

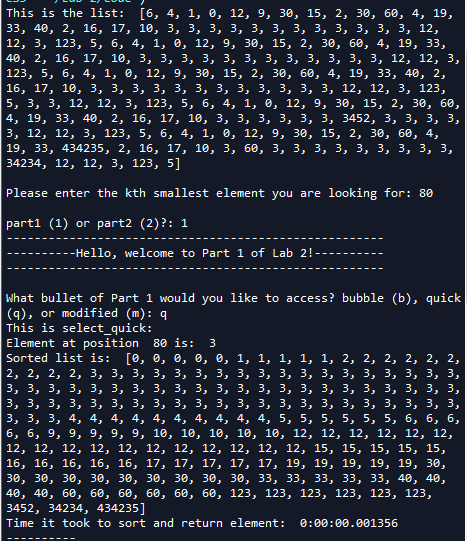
For this operation, I attempted to do Quick Sort inside of only one function. I could not find ‘k’ and return since it was recursive. I also tried splitting the list into ‘left’, ‘right’, and ‘equal’, this did not work since it was also recursive and when I attempted to find ‘k’, it would go out of index. If I tried comparing ‘k’ with the length of list ‘left’, it would not recognize it as an int. I then just created a partition that was called inside of ‘select\_quick’ and made recursive calls within the method so it could sort the list.



Case 1: input=60



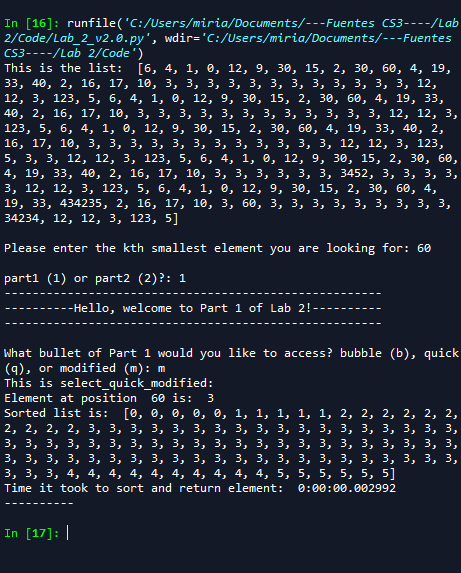
Case 2: Input=70



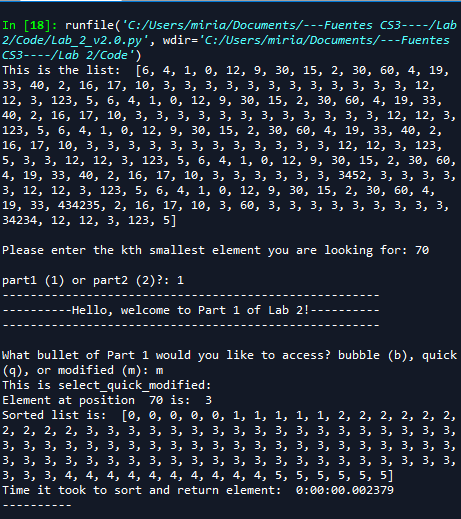
Case 3: Input = 80

**Part #1 select\_modified\_quick:**

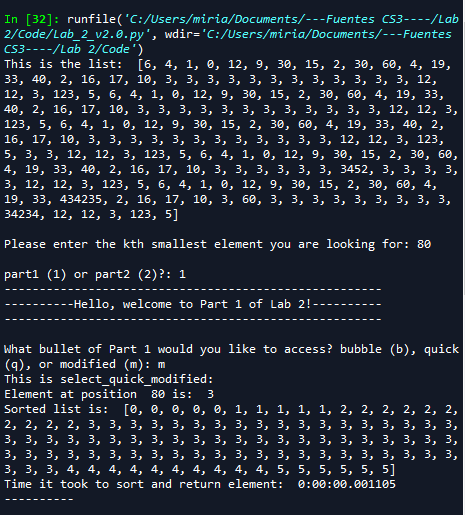
I approached this the same way I approached **Part #1 select\_quick**. It was a lot of trial and error since I did not know how to split the list only once. I tried making a recursive call once, but I had no idea how to stop it. I did not know how to send the list that contained the ‘k’th element. I then thought it would be easier to split it inside of a method and send the sub list that contained the ‘k’th element.



Case 1: Input=60



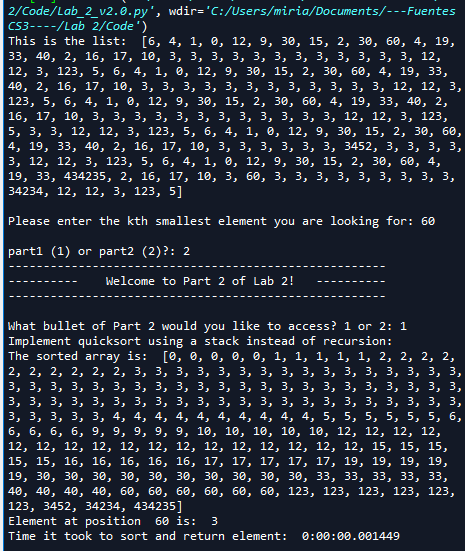
Case 2: Input=70



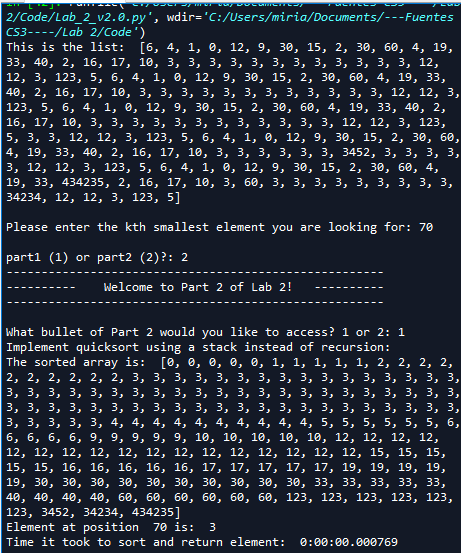
Case 3: Input = 80

Part #2 quick\_stack:

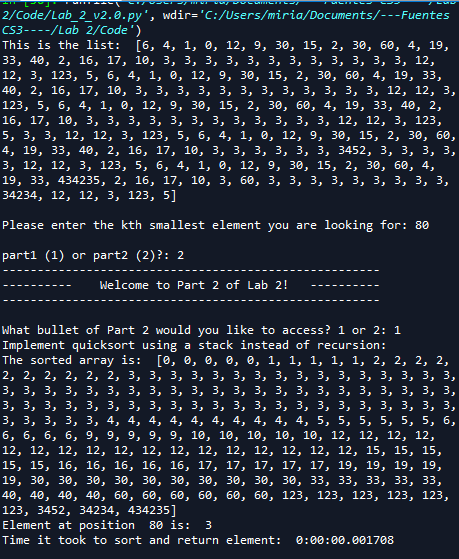
For this operation, I approached this problem the same way I attempted to approach select\_quick. I created three temporary lists in which I determined if the lowest was greater than or smaller than the pivot. I was only able to split the list into two sub lists once and did not know how to proceed from there. I then thought about creating a list in which I could save the greatest element, smallest element, and the pivot. I pre-created the size of the stack so it would not go out of index.



Case 1: Input=60



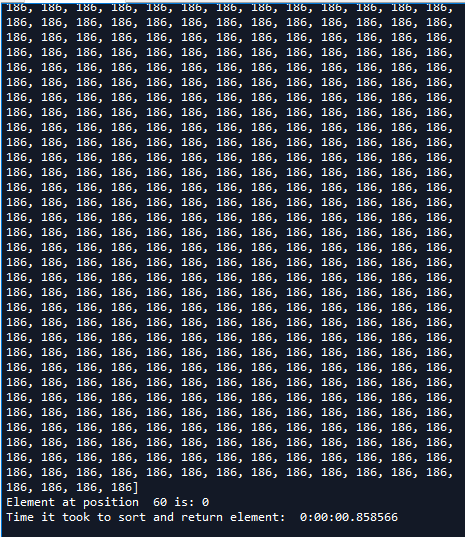
Case 2: Input=70



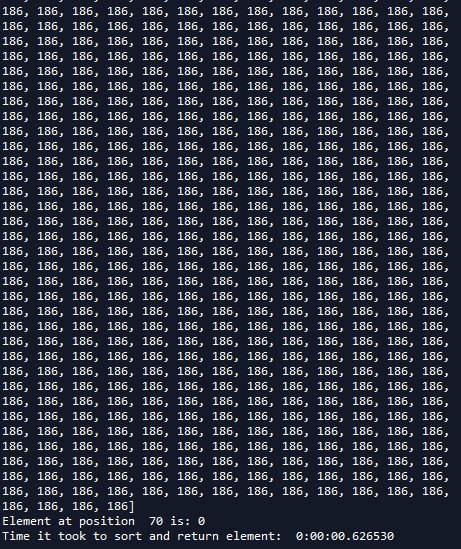
Case 3: Input = 80

**Part #2 while\_loop:**

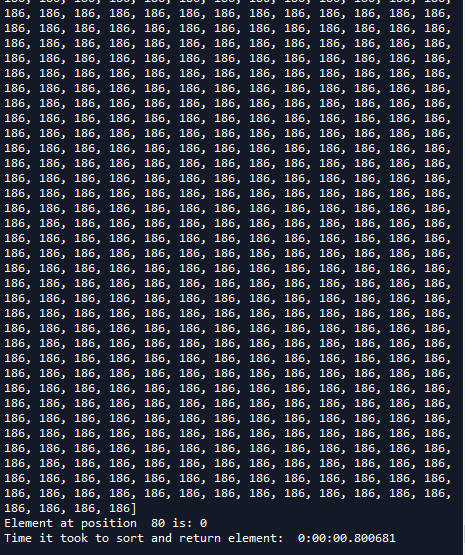
For this operation, as redundant as I sound, I approached this problem the same way I approached select\_quick and quick\_stack. I was only able to divide the list into two sub lists. I also tried to pre-create a list in which I would append the values every time it went inside of the while loop. I tried approaching the problem the same way I approached ‘quick\_stack’ but was unable to replicate everything since I could not make recursive calls (I wanted to call ‘partition’) but that was a recursive function.



Case 1: input = 60

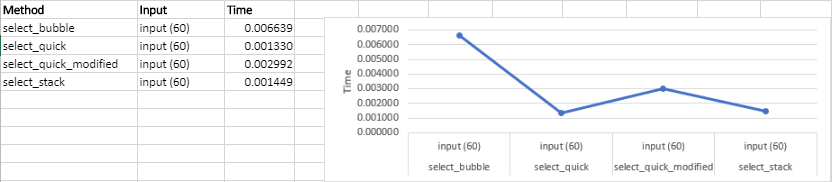


Case 2: input = 70

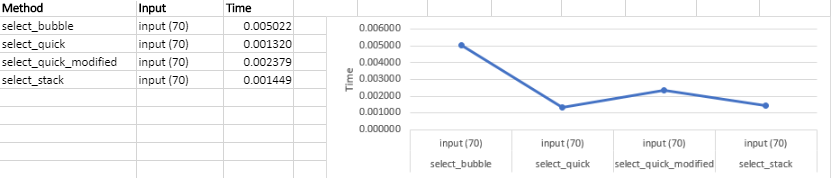


Case 3: Input = 30

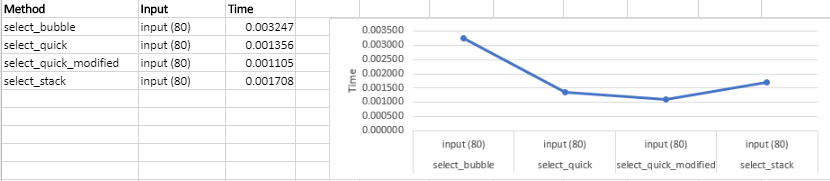
**Input = 60**



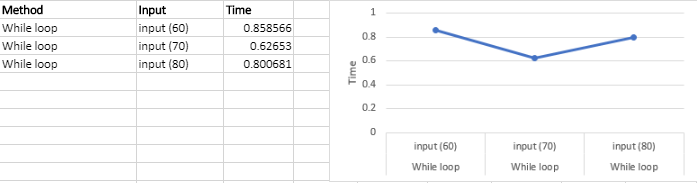
**Input = 70**



**Input = 80**



**While Loop Only**



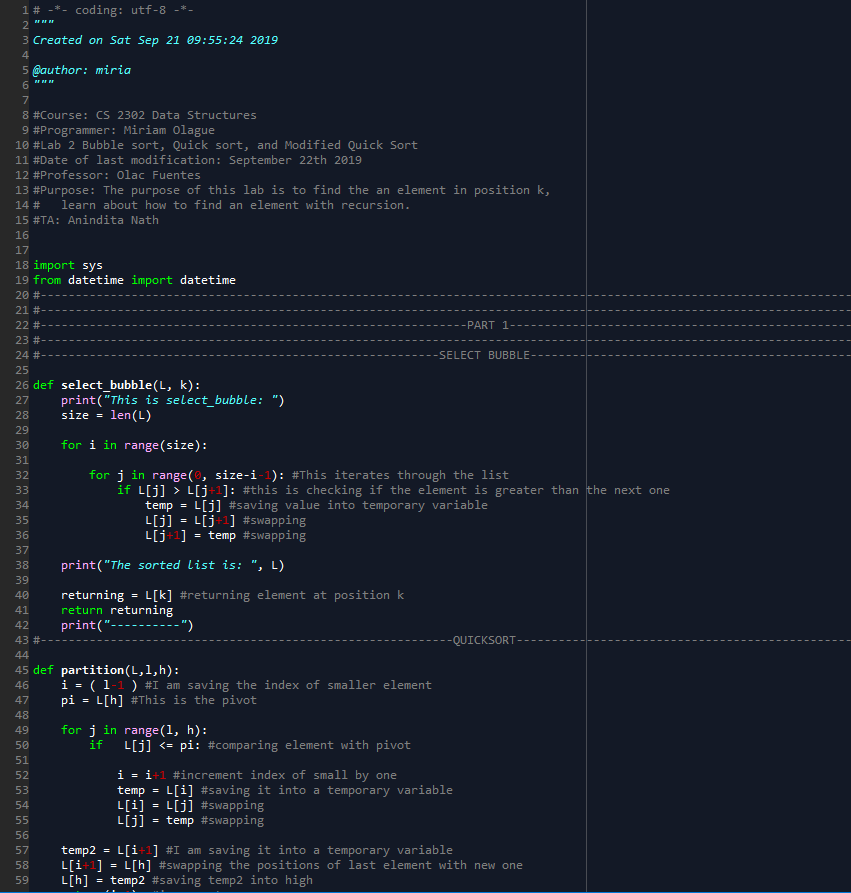
Input = 60; 70; 80

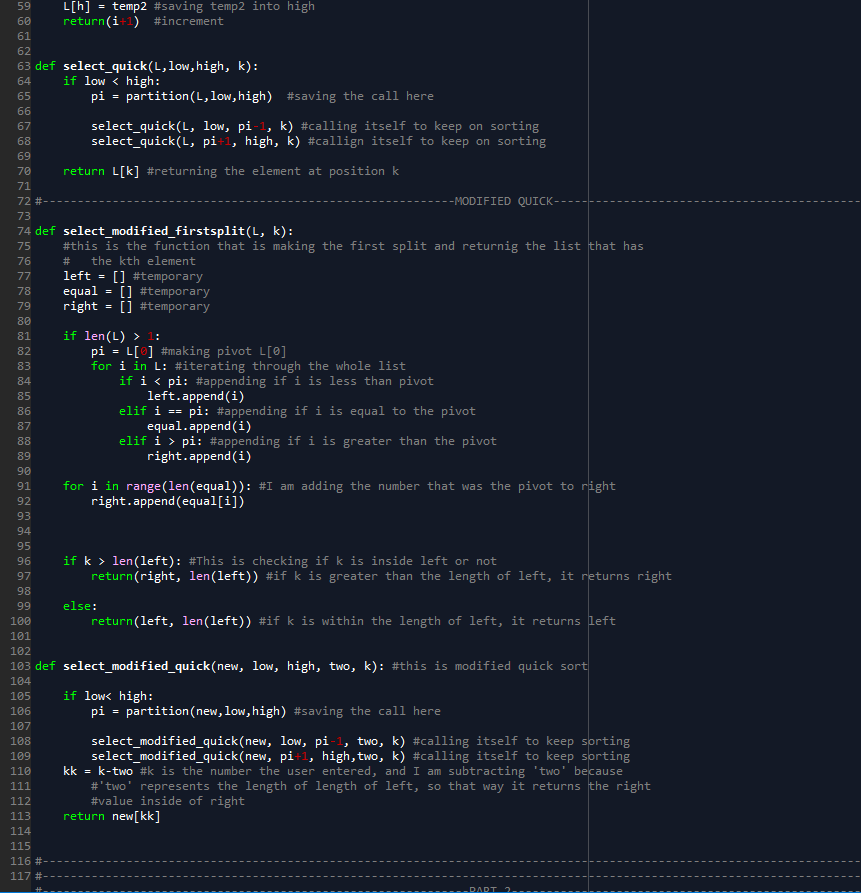
As the results show, there is a visible change between running times between different solutions solving for quick sort. Bubble sort takes more time to sort compared to quick sort. Quick sort is fast but with the modified quick sort it makes even faster since it divided the work into two. Stacks aren’t as efficient as the quick sort that has a partition and the modified version of quick sort. For the while loop, I was not able to solve it, therefore I did not include it in the graph since it made all of the other results appear as zeros; the while loop was producing big running times so I decided to not include it. However, I attached a different graph in which the running time for the while loop is displayed.

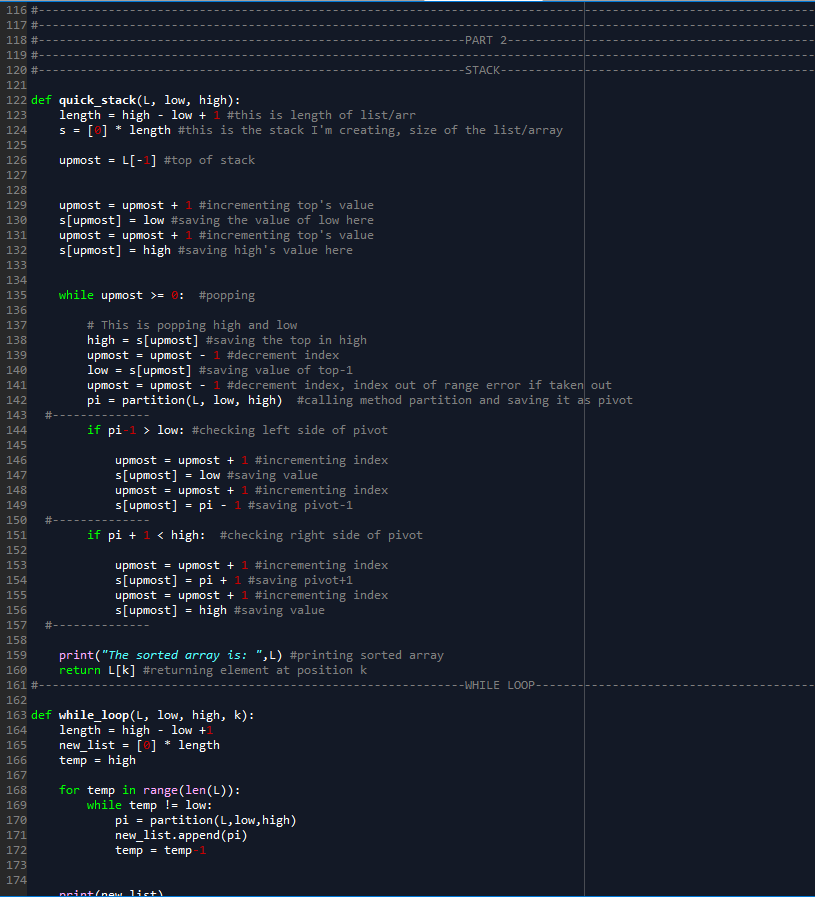
**Conclusion**

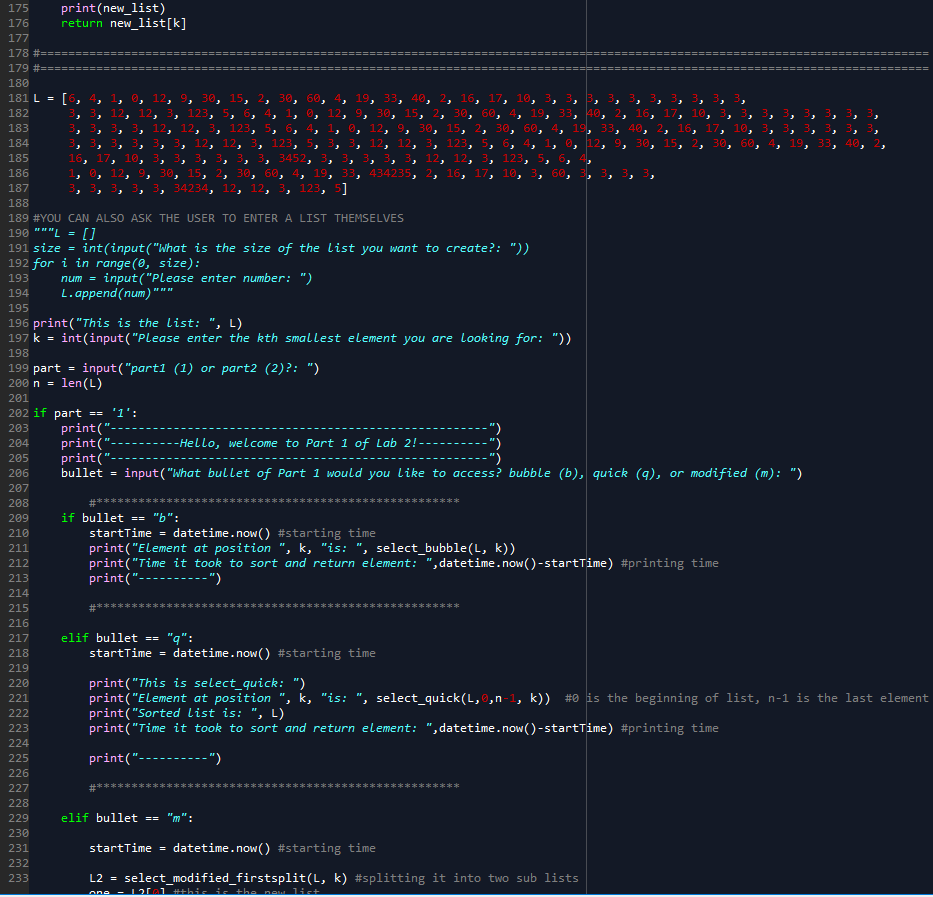
In conclusion, this lab made me think out of the box. It made me think of many ways to solve Quick Sort, not only one solution. My knowledge on stacks improved and was able to organize the data in an orderly manner so it could be cleaner. I also learned how to iterate through lists using stacks. The analytical running times do agree with what I see in practice since quick sort is fast to sort compared to bubble sort.

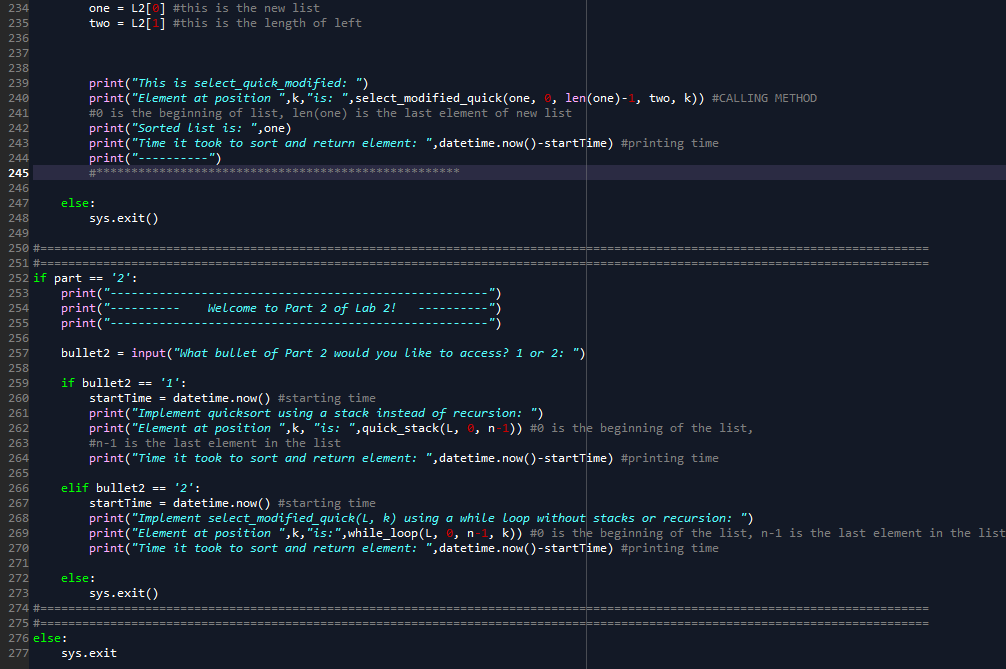
**Appendix**

****

****

****

****

****