**CS 2302 Data Structures**

**Spring 2019**

**Lab Report #2**

Due: Sept. 20, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we looked at bubble sort and quick sort algorithms to select an element at index k from an unsorted list L after being sorted by the implemented sorting algorithms. In this lab I compared the different time complexities and outputs of the slightly different sorting algorithms.

**Proposed Solution Design and Implementation**

**Operation #1:**

Select\_bubble(L,k) is implemented by calling another function bubble\_sort(L, n) which is recursive. I took a regular, non-recursive bubble sort and made it recursive by replacing the outer for loop with a recursive call to itself, getting down to the base case of n < length of the list L.

**Operation #2:**

Select\_quick(L,k) is implemented by calling a function quick\_sort(L, low, high) which is a recursive function. It checks if low < high, and then finds a pivot and does the list reordering in partition(L, low, high), and then calls itself for both the left and right halves of the partitioned list.

**Operation #3:**

Select\_modified\_quick(L,k) is implemented by calling quick\_sort2(L, low, high, k), where k is the index for which we are looking. This modified version checks if k is lower than the pivot index or higher, and only sorts that respective half of the list rather than the entire list. This reduces the recursive calls by a lot, as it only makes a single recursive call each time rather than two recursive calls each time.

**Operation #4:**

I implemented a stack version of quick sort by making a function called select\_quick2(L,k), which calls the function qsort\_stack(L). This stack function creates a stack with the indexes of the first and last elements stored and assigns those values to high and low. After those are popped, the function appends the first and last index of the two halves and runs the partitioning/sorting algorithm again. The first half is sorted, with all the intermediary index values being pushed/popped, and finally once that half is reduced to halves less than a size of 2, the second half of the list is sorted, with their first/last indexes being pushed/popped into a list “stack”. I had to do some research on stacks to implement this method

**Operation #5:**

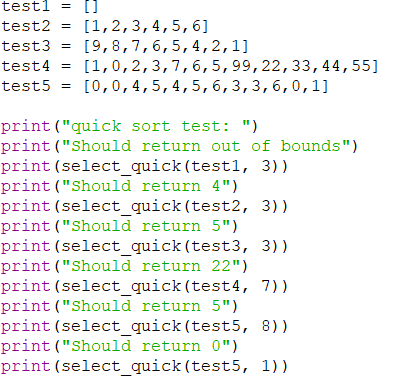
I implemented a modified non-recursive quicksort using a while loop fairly easily. Select\_modified\_quick2(L,k) by calling quick\_sort3(L, low, high, k) which then calls partition(L, low, high) with parameters depending on whether the pivot index is greater than or less than k.

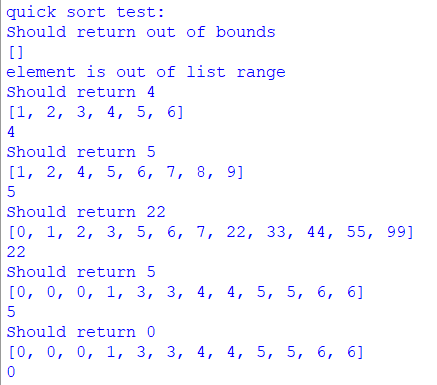
Operation #6:

Partition(L, low, high) first determines a pivot element by comparing the first, last and middle elements to each other and selecting the median value. Then it takes the pivot element value, and then swaps the pivot and first elements. Then it uses a “border” element, compares each element to the pivot value. If it is less, the border index increases and then that element swaps with the element. Once at the high range of the list L, the first element and border element swap, so that the border element is effectively the pivot and all elements to the left of it are of a lesser value.

**Experimental Results**

I began testing part 1, I would see if the functions were sorting lists by printing them and then printing the value at the index we were looking for.

For example:

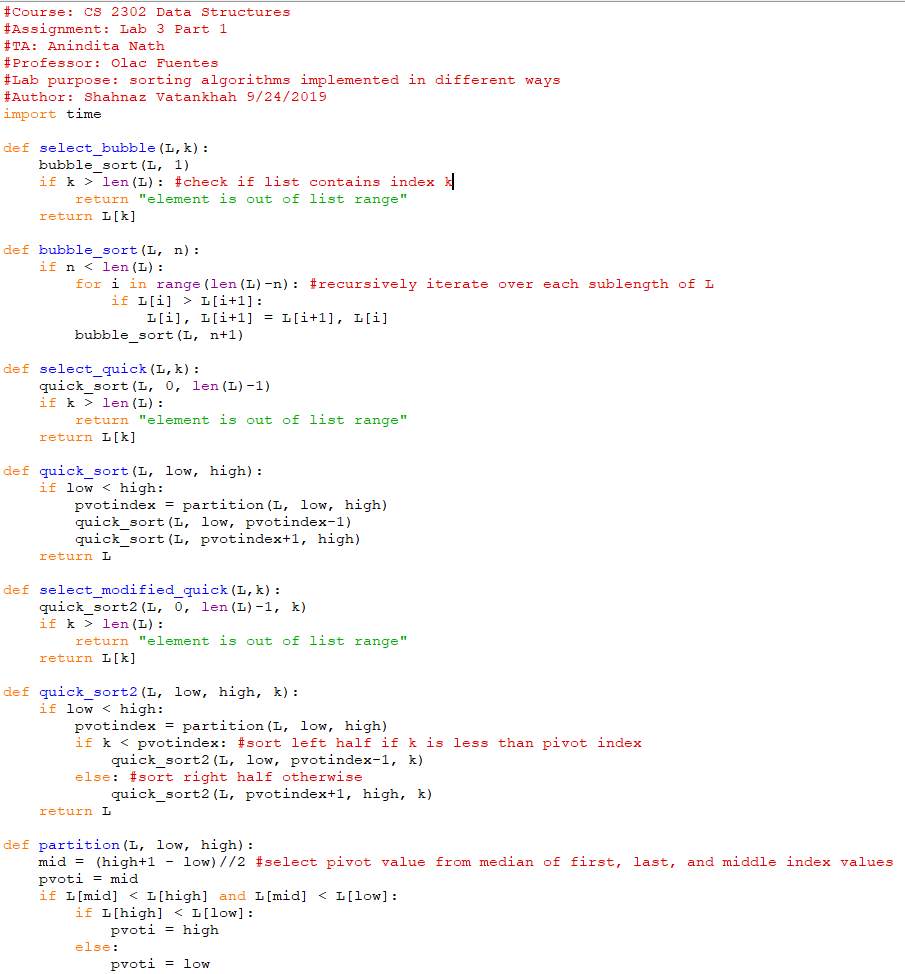
Output:

I did this for every function, and confirmed that all functions output the same index element in the tests. I tested the algorithms with lists of different lengths, even and odd lengths, ordered and unordered lists, lists with repeating values, and an empty list. I also implemented a check to see if the value of k was larger than the list length, and in that case the user is notified that the element is out of the list range. I also added timestamps to calculate the seconds for executing each sorting algorithm.

Runtime analysis:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm O(n)** | **Runtime 1 (sec)** | **Runtime 2** | **Runtime 3** | **Avg Runtime** |
| Bubble sort O() | .0211 | .0203 | .0199 | .0204 |
| Quick sort  O() | .0188 | .0223 | .0201 | .0204 |
| Modified QS O(/ 2) | .0174 | .0071 | .0162 | .0136 |
| Stack QS  O() | .0222 | .0166 | .0212 | .0200 |
| While QS  O(/ 2) | .0189 | .0106 | .0164 | .0153 |

On average, the modified quick sort for part 1 takes the least amount of time to run the same test that all the other algorithms also ran. I determined that the worst case time complexities for bubble and quick sort are both O(), which correlates to my average runtime results for those two algorithms. I figured that the stack implementation of quick sort would take about the same amount of time and have the same time complexity because of the while loop and because it’s pretty much simulating what the recursive stack does. The resulting avg runtime was consistent with this deduction. Then I figured, both of the modified quick sorts, which purposely only sort the side of the list which the index is on, will take the same runtime but cut in half since it is only sorting one side, no . The experimental results are almost around .5 of that of the regular quick sort, closer to .75 actually. It would likely be closer to .5 if n was very large.

**Appendix:**

