**LAB 2**

**Daniel Cruz**

**Introduction**

The purpose of this lab is to show how certain sorting algorithms perform in runtime complexity. In this lab, bubble sort and various implementations of quicksort are analyzed experimentally for the number of comparisons. With this knowledge, the lab exemplifies ways to optimize certain well-known algorithms to better fit your needs while reducing excessive operations. A second objective of this lab is to show how to mimic a recursive algorithm using either a loop or a stack.

**Proposed Solution Design and Implementation**

I decided to break the problem into two distinguishable sections: a section for methods for the sorting algorithms and then a section for methods to perform certain tests on the functions. I will explain the implementation of the sorting algorithms first:

Bubble sort: bubble sort algorithm is implemented in the simplest fashion possible. Like all the sorting algorithms to follow, the first thing checked is if the list is empty or if the k value entered is out of bounds. If this is the case ‘None’ is returned. The list is then iterated through, checking if two neighboring elements are arranged from smaller to larger. If a pair is found to be out of place, then they are swapped. This algorithm will continue like this until a full pass of the list is performed with no swaps. Now that the list is sorted, the value at the kth position is returned.

Quicksort: I elected to split the quicksort into two methods. The first method is used for better interfacing to the user, only asking for the list and the k value, which is then checked as before for the empty list or out of bounds k, and then passed into the recursive method of this algorithm. Once the recursive method finishes sorting the list, the kth index of the list is returned. The recursive method is called with the low index and high index to perform the sort range. The first value in the range is chosen as the pivot. Two iterators are set to the low index and high index respectively. The low iterator increments until it reaches out of bounds or a value greater than the pivot is found. The high iterator decrements until it reaches out of bounds or a value less than the pivot is found. If the iterators did not pass each other, then the two values swap places. This continues until the iterators pass each other. Once this happens, the pivot is moved to the center of the section. The range is then split from the pivot index and then both sections are recursively passed on until the indices passed are equal or the low index is greater than the high index.

Modified quicksort: modified quicksort has the same layout as the normal quicksort. The big change is at the recursive call. The k value is passed to the recursive section and the index chosen to split the range is compared to the k value. If k is greater than this value, then the value in search for is in the higher partition and thus only this section is called in the recursion. Otherwise if the k value is less than the split index, recursively call the lower partition as this section contains the sorted kth index. If the kth index is the same as the middle value, then this value is already in place and it can be returned without any more sorting.

Quicksort with stack: This algorithm simulates the recursive calls of quicksort with a stack. The algorithm is very similar to the recursive version except that the two methods are combined. In this case, a stack is created that will take in the high and low indices. Indices with a high index less than the low index is pushed into the stack. This will signify the end of performing the sorting values. Then, while the stack is not empty, the method will perform the same comparison operations as the recursive call in the normal quicksort. At the end of the first pass, the higher partition indices are pushed to the stack and the algorithm continues with sorting the lower partitions. Once the lower partitions are all sorted (the indices for the partition are swapped or equal) then the stack pops the higher partition indices and those continue being sorted. Once the stack is empty, this signifies that all parts of the list have been sorted and the kth index is returned.

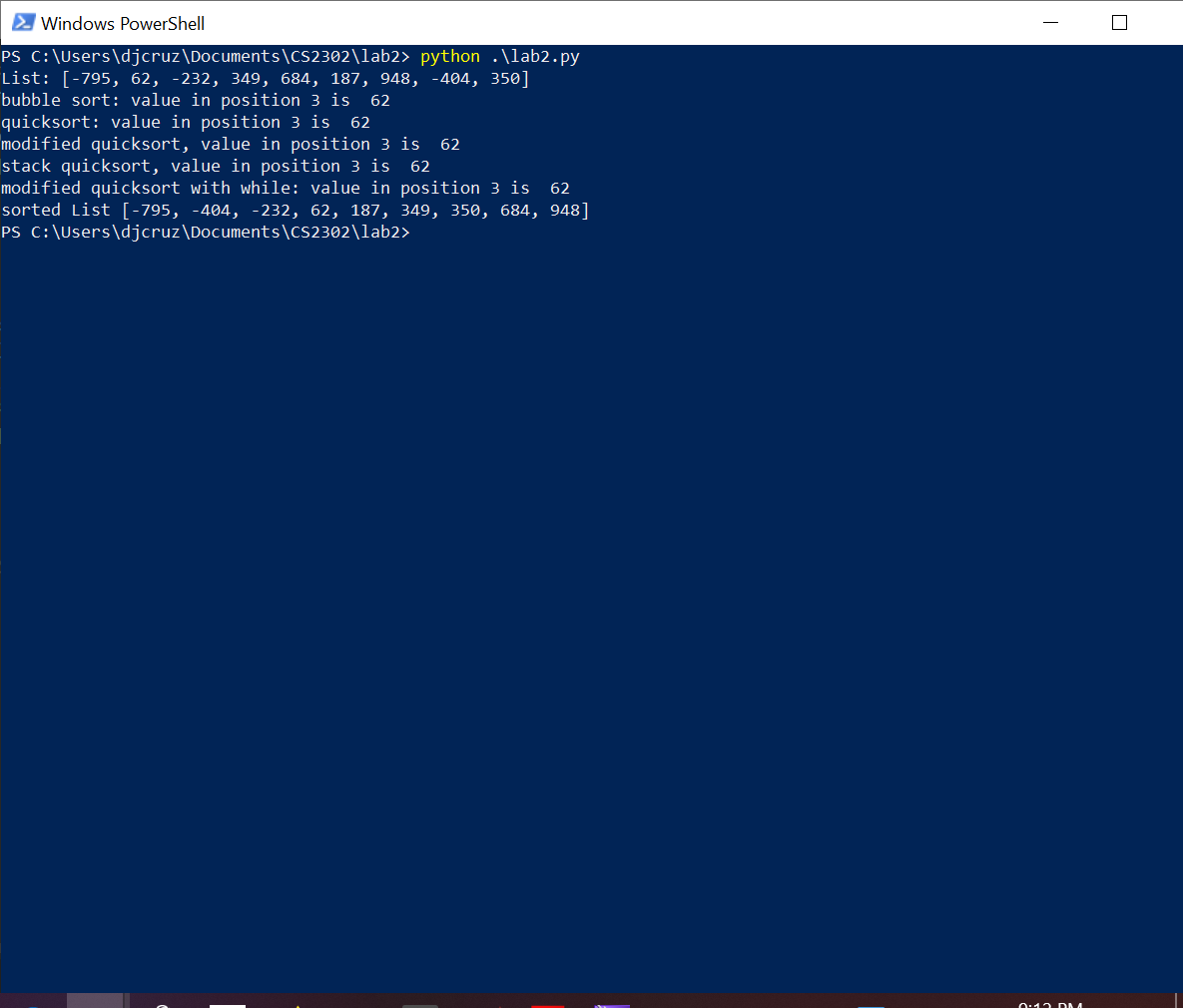
Modified quicksort with a while loop: This method is very similar to the recursive section. Instead of passing the selected partition to the recursive call based on k, the indices are immediately set and the algorithm loops back to performing the quicksort. This algorithm will only end if the pivot index is the same as the k value.

There are two sets of test methods: one set to test if all functions return the correct value and a set to test the comparison amount. The first set creates a list of random length filled with random numbers. Each sorting algorithm is used on the list for each k value in the list and all the functions are compared with each other and then with pythons own list sorting method. Any discrepancies are printed.

In each method, a counter has been added every time two elements in a list are compared. The second set of test methods resets these counters and then calls each method and prints these counters. Another method receives a number that will iterate each sorting algorithm with lists of increasing size up to that inputted number. The number of comparisons in each algorithm for each list length will then be saved to a csv file for analyzing.

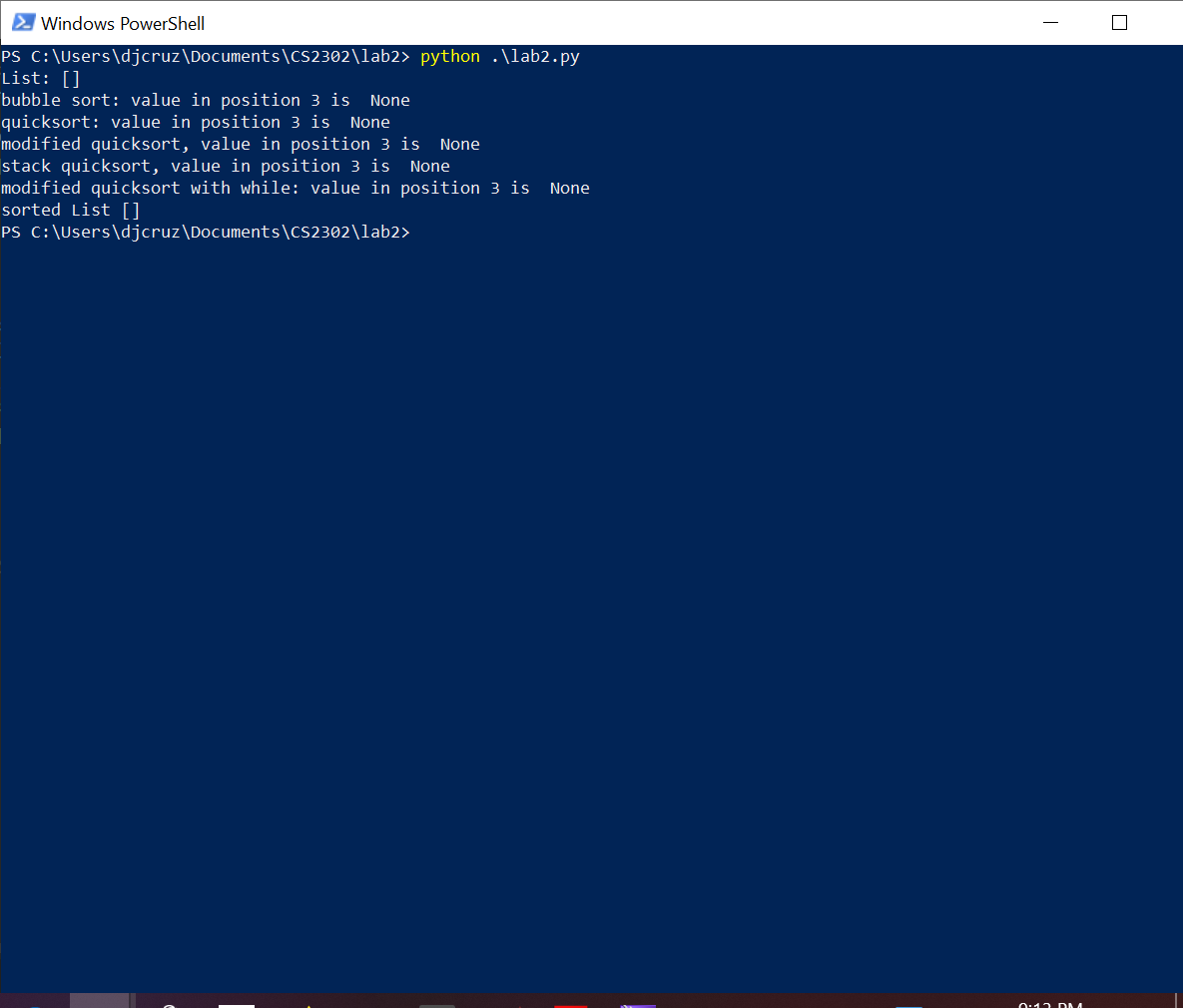
**Experimental Results**

The first result is a test on all the search algorithms to see if they return the correct value:

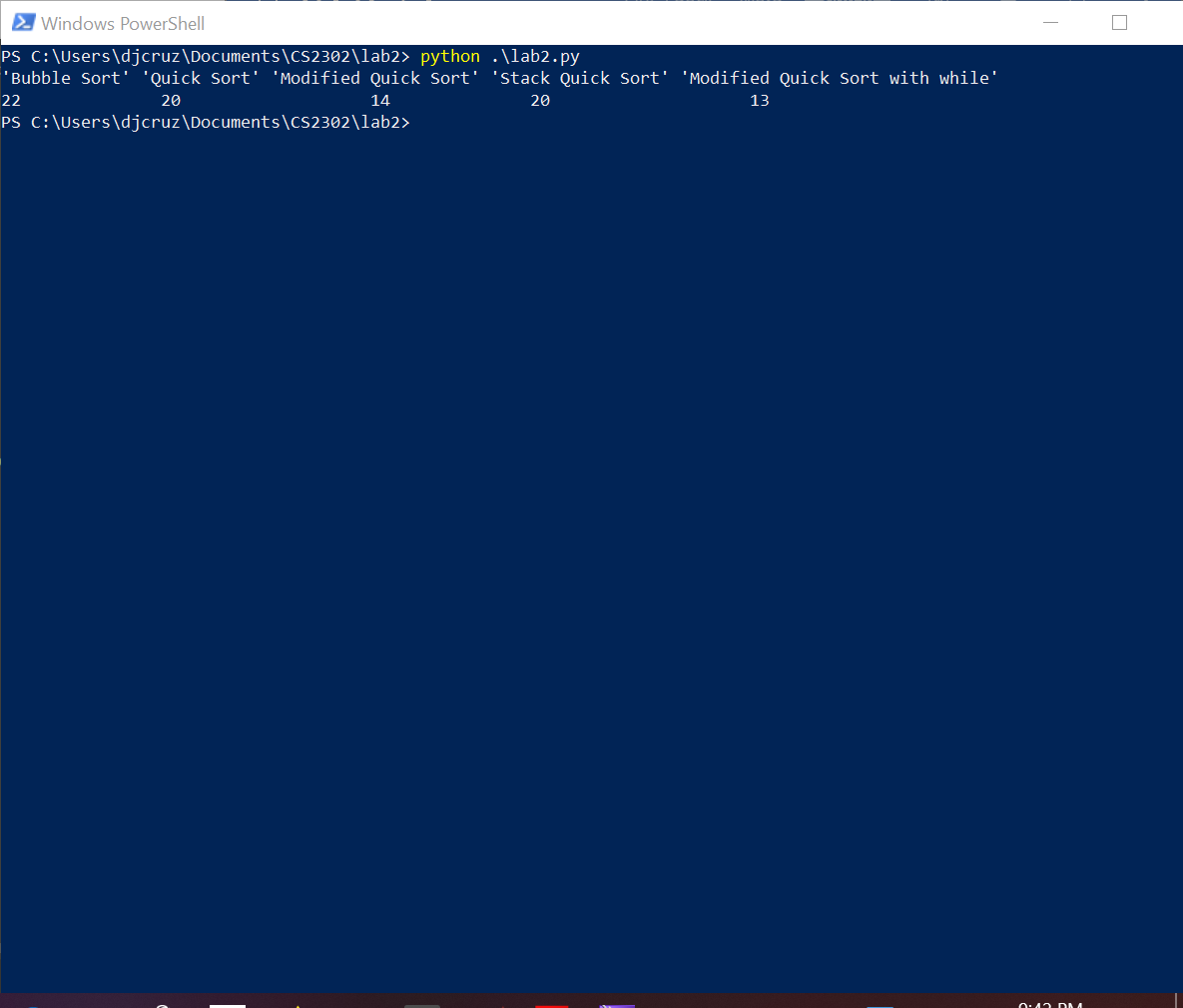


In this test, all algorithms return the same value and this value can be compared with what is in the list sorted by python. In this case everything checks out.

The second test will check for the edge cases. An empty list with an index should return None:



The next test will find the average of all the comparisons in each algorithm. This will be averaged over 10000 randomized lists of randomized length.



As shown, on average the modified quicksorts perform the best while bubble sort performs the worst. To show the growth rate of comparisons, I will graph the number of comparisons from a list of increasing size from 0 to size of 30:

Bubble sort has the fastest comparison growth of all algorithms. Quicksort and stack quicksort performed on par with each other and the modified qucksorts performed the best and on par with themselves.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| N | 'Bubble Sort' | 'Quicksort' | 'Modified Quicksort' | 'Stack Quicksort' | 'Modified Quicksort with while' |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 2 |
| 2 | 2 | 6 | 10 | 6 | 9 |
| 3 | 2 | 7 | 7 | 7 | 6 |
| 4 | 9 | 12 | 13 | 12 | 12 |
| 5 | 20 | 16 | 13 | 16 | 12 |
| 6 | 15 | 19 | 19 | 19 | 18 |
| 7 | 30 | 30 | 11 | 30 | 10 |
| 8 | 42 | 32 | 32 | 32 | 31 |
| 9 | 56 | 42 | 25 | 42 | 24 |

**Conclusion**

The results from this lab proved the theoretical calculations for each of the algorithms. Bubble sort performed considerably worse than the quicksorts. The modified quicksorts were on average better than the normal quicksorts however as expected from how the algorithm works, this was not always a sure case for speedup. This lab helped refresh me on the implementation of certain sorting algorithms. It also allowed me to see an experimental proof of the theoretical runtimes of these algorithms. I struggled the most with this lab when trying to convert the recursive quicksort to a non-recursive version with a stack. However, after using activation records and reviewing the examples shown in class, I was able to form a working version.

**Appendix**

from random import randint

from collections import deque

""" BUBBLE """

#counts comparisons.

bubbleCount = 0

def select\_bubble(L,k):

global bubbleCount

if len(L) == 0 or k > (len(L) - 1):

return None

sorted = False

while(not sorted):

#set to true, will be set to false if a nonsorted section is found.

sorted = True

for i in range(len(L)-1):

bubbleCount += 1

if(L[i] > L[i+1]):

L[i], L[i+1] = L[i+1], L[i]

sorted = False;

return L[k]

""" SELECT\_QUICK """

quickCount = 0

def select\_quick(L,k):

global quickCount

if len(L) == 0 or k > (len(L) - 1):

return None

recurs\_quick(L, 0, len(L)-1)

return L[k]

#recusion function

def recurs\_quick(L,low,high):

global quickCount

#list of 1 is already sorted

if high <= low:

return

else:

lInd = low+1 #left index, low itself is reserved for the pivot

rInd = high #right index

while(lInd <= rInd):

#decrement down the list until value less than pivot is found.

quickCount += 1

while rInd > low and L[rInd] >= L[low]:

quickCount += 1

rInd -= 1

#increment left until value is found greater than pivot

quickCount += 1

while lInd < high+1 and L[lInd] < L[low]:

quickCount += 1

lInd += 1

#as long as indexes dont cross, swap the values

if lInd < rInd:

L[lInd], L[rInd] = L[rInd], L[lInd]

rInd -= 1

lInd += 1

L[rInd], L[low] = L[low], L[rInd] #place pivot in the middle

mid = rInd

recurs\_quick(L, low, mid)

recurs\_quick(L, mid+1, high)

""" SELECT\_MODIFIED\_QUICK """

modifiedQuickCount = 0

def select\_modified\_quick(L,k):

global modifiedQuickCount

modifiedQuickCount += 1

if len(L) == 0 or k > (len(L) - 1):

return None

rec\_modified\_quick(L, 0, len(L)-1, k)

return L[k]

def rec\_modified\_quick(L,low,high, k):

global modifiedQuickCount

#list of 1 is already sorted

if high <= low:

return

else:

lInd = low+1 #left index, low itself is reserved for the pivot

rInd = high #right index

while(lInd <= rInd):

#decrement down the list until value less than pivot is found.

modifiedQuickCount += 1

while rInd > low and L[rInd] >= L[low]:

modifiedQuickCount += 1

rInd -= 1

#increment left until value is found greater than pivot

modifiedQuickCount += 1

while lInd < high+1 and L[lInd] < L[low]:

modifiedQuickCount += 1

lInd += 1

#as long as indexes dont cross, swap the values

if lInd < rInd:

L[lInd], L[rInd] = L[rInd], L[lInd]

rInd -= 1

lInd += 1

L[rInd], L[low] = L[low], L[rInd] #place pivot in the middle

mid = rInd

modifiedQuickCount += 1 #one comparison for first if

if k < mid:

rec\_modified\_quick(L, low, mid, k)

elif k > mid:

modifiedQuickCount += 1 #second comparison for elif

rec\_modified\_quick(L, mid+1, high, k)

else:

modifiedQuickCount += 1 #second comparison for elif

return mid

""" SELECT\_QUICK W/ STACK """

quickStackCount = 0

#Implementing a stack for quicksort

def select\_quickStack(L,k):

global quickStackCount

if len(L) == 0 or k > (len(L) - 1):

return None

stack = deque()

low = 0

high = len(L)-1

#flag values to indicated end of stack

stack.append(0) #low

stack.append(-1)#high

#checks if there are elements in the stack

while stack:

while low < high:

lInd = low+1 #left index, low itself is reserved for the pivot

rInd = high #right index

while(lInd <= rInd):

#decrement down the list until value less than pivot is found.

quickStackCount += 1

while rInd > low and L[rInd] >= L[low]:

quickStackCount += 1

rInd -= 1

#increment left until value is found greater than pivot

quickStackCount += 1

while lInd < high+1 and L[lInd] < L[low]:

quickStackCount += 1

lInd += 1

#as long as indexes dont cross, swap the values

if lInd < rInd:

L[lInd], L[rInd] = L[rInd], L[lInd]

rInd -= 1

lInd += 1

L[rInd], L[low] = L[low], L[rInd] #place pivot in the middle

mid = rInd

#save these values for later to sort this section of the list. (second recusion call)

stack.append(mid+1) #low

stack.append(high) #high

#begin low to mid partitions for sorting (first recusion call)

high = mid

high = stack.pop()

low = stack.pop()

return L[k]

""" SELECT\_MODIFIED\_QUICK W/ WHILE """

whileQuickCount = 0

#select\_modified\_quick with just a while loop

def while\_modified\_quick(L,k):

global whileQuickCount

# whileQuickCount += 1

if len(L) == 0 or k > (len(L) - 1):

return None

low = 0

high = len(L)-1

#function will eventually leave while loop through return statement

while True:

lInd = low+1 #left index, low itself is reserved for the pivot

rInd = high #right index

while(lInd <= rInd):

#decrement down the list until value less than pivot is found.

whileQuickCount += 1

while rInd > low and L[rInd] >= L[low]:

whileQuickCount += 1

rInd -= 1

#increment left until value is found greater than pivot

whileQuickCount += 1

while lInd < high+1 and L[lInd] < L[low]:

whileQuickCount += 1

lInd += 1

#as long as indexes dont cross, swap the values

if lInd < rInd:

L[lInd], L[rInd] = L[rInd], L[lInd]

rInd -= 1

lInd += 1

L[rInd], L[low] = L[low], L[rInd] #place pivot in the middle

mid = rInd

whileQuickCount += 1

if k < mid:

high = mid

elif k > mid:

whileQuickCount += 1

low = mid+1

else:

whileQuickCount += 1

return L[mid]

""" FUNCTIONS FOR TESTING ALL FUNCTIONS """

def randomList(n = randint(0,10)):

L = [None] \* n

return [randint(-1000,1000) for element in L]

def randomListDebug():

for p in range(1000):

R = randomList()

for i in range(len(R)):

quick = select\_quick(R.copy(),i)

modQuick = select\_modified\_quick(R.copy(), i)

bubble = select\_bubble(R.copy(), i)

stackQuick = select\_quickStack(R.copy(), i)

whileQuick = while\_modified\_quick(R.copy(), i)

sortedList = R.copy().sort()

if(quick != modQuick and quick != bubble and modQuick != bubble and quick != stackQuick and quick != whileQuick and quick != sortedList[i]):

print("error, values do not match up or incorrect: ", quick, modQuick, bubble, sortedList[i])

def printOutput():

L = randomList()

k = 3

print("List: " + str(L))

print("bubble sort: value in position",k,"is ",select\_bubble(L.copy(),k))

print("quicksort: value in position",k,"is ",select\_quick(L.copy(),k))

print("modified quicksort, value in position",k,"is ",select\_modified\_quick(L.copy(),k))

print("stack quicksort, value in position",k,"is ",select\_quickStack(L.copy(),k))

print("modified quicksort with while: value in position",k,"is ",while\_modified\_quick(L.copy(),k))

L.sort()

print("sorted List " + str(L))

""" FUNCTIONS FOR TESTING NUMBER OF COMPARISONS """

def resetComparisons():

global bubbleCount

global quickCount

global modifiedQuickCount

global quickStackCount

global whileQuickCount

bubbleCount = 0

quickCount = 0

modifiedQuickCount = 0

quickStackCount = 0

whileQuickCount = 0

def checkComparisons(n=1):

global bubbleCount

global quickCount

global modifiedQuickCount

global quickStackCount

global whileQuickCount

quick = 0

modQuick = 0

bubble = 0

stackQuick = 0

whileQuick = 0

for i in range(n):

resetComparisons()

R = randomList()

select\_quick(R.copy(),0)

quick += quickCount

select\_modified\_quick(R.copy(), 0)

modQuick += modifiedQuickCount

select\_bubble(R.copy(), 0)

bubble += bubbleCount

select\_quickStack(R.copy(), 0)

stackQuick += quickStackCount

while\_modified\_quick(R.copy(), 0)

whileQuick += whileQuickCount

print("'Bubble Sort'", "'Quick Sort'", "'Modified Quick Sort'", "'Stack Quick Sort'", "'Modified Quick Sort with while'")

print(bubble//n, " ", quick//n, " ", modQuick//n, " ", stackQuick//n, " ", whileQuick//n)

#print to file the number of comparisons based on incrementing list size.

def iterateSize(n = 1):

global bubbleCount

global quickCount

global modifiedQuickCount

global quickStackCount

global whileQuickCount

quick = 0

modQuick = 0

bubble = 0

stackQuick = 0

whileQuick = 0

fp = open("comparisons.csv",'w')

fp.write("'Bubble Sort', 'Quick Sort', 'Modified Quick Sort', 'Stack Quick Sort', 'Modified Quick Sort with while'\n")

for i in range(n):

resetComparisons()

R = randomList(i)

select\_quick(R.copy(),0)

quick += quickCount

select\_modified\_quick(R.copy(), 0)

modQuick += modifiedQuickCount

select\_bubble(R.copy(), 0)

bubble += bubbleCount

select\_quickStack(R.copy(), 0)

stackQuick += quickStackCount

while\_modified\_quick(R.copy(), 0)

whileQuick += whileQuickCount

fp.write(str(bubbleCount) + ", " + str(quickCount) + ", " + str(modifiedQuickCount) + ", " + str(quickStackCount) + ", " + str(whileQuickCount) + "\n")

if \_\_name\_\_ == '\_\_main\_\_':

printOutput()

#randomListDebug()

#checkComparisons(10000)

#iterateSize(30)

I [Daniel Cruz] 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