**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #2**

Due: September 20, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

Lab 2: Runtime Complexity focuses on developing five sorting algorithms, calculating their theoretical runtime complexity, and determining if the program’s results match the analytical running times. The five sorting methods include a recursive Bubble Sort, QuickSort, a modified QuickSort, an iterative modified QuickSort, and QuickSort implemented with a stack. For each of the sorting methods, the user also identifies the kth element that they want returned. The modified QuickSort varies from the regular QuickSort because only one recursive call is made depending on the kth element that the user wants to find. Additionally, the iterative modified QuickSort acts like the modified QuickSort, however, it does not use a stack or recursion. Lastly, the QuickSort with a stack algorithm mimics how activation records are stored to perform the QuickSort function. Since a major part of this lab focuses on comparing the analytical runtime and number of comparisons, the theoretical runtime for each algorithm is calculated and compared with the collected results.

**Proposed Solution Design and Implementation**

NOTE: The report will continue to reference n, which stands for the number of elements in the designate.

**Part 1:**

Bubble Sort

In order to meet the requirements of a function definition with only the list and kth index as parameters, I decided to create an inner and outer recursive method. The outer recursive method contains the designated parameters and calls the inner recursive method, passing in the list, k, and the length of the list. After the inner recursive function finishes sorting the list, it returns the element at the kth index. The inner Bubble Sort algorithm repeatedly starts at the first element of the list and iterates through the list until the end index is reached. As the list is being iterated through, elements are swapped so that the right element is greater than or equal to the left element. This causes the largest element in the array segment to be in the rightmost position. Thus, for the next recursive call, the end index will be decreased by one. My method also includes two base cases. If the end index eventually reaches 0, then the list is sorted, and the kth element is returned. I also realized that Bubble Sort depends on swaps. If a list has been completely iterated through and no values have been swapped, then the algorithm stops. I thought this was a very important base case, since additional swaps would be unnecessary. As for the expected runtime, in the worst case and average case, the algorithm’s recursive relation is T(n) = T(n - 1) + n which is equal to T(n) = (n(n+1)/2). This is because the recursive calls continue to decrement the end index by one resulting in O(n^2). However, in the best case, the algorithm will only iterate through the list once, if it is sorted. Thus, the recursive relation would be T(n) = n, since no recursive calls would be made. This would yield a best-case runtime of O(n).

QuickSort

Similar to Bubble Sort, QuickSort has an inner and outer recursive function. The outer function receives the list and k, which is then passed to the inner recursive function. The inner recursive function additionally receives the low and high index of the array segment to be sorted. Initially, zero and length of the array minus one are passed in. Before making any recursive calls, the algorithm calls the partition function. This function uses a pivot and places all elements less than the pivot in the left sub-list and all elements greater than or equal to the pivot in the right sub-list. The parameters for this function include the list, the low index, and the high index. In order to pick a central pivot, I decided to select the middle most element and swap this element with the high element. In order to sort the elements as described above, the low pointer is iterated until it reaches an element that is greater than or equal to the pivot and the high pointer is decremented until it reaches an element that is less than or equal to the pivot. If the low pointer is still less than or equal to the right pointer, then elements are swapped and the pointers are updated. As soon as the low pointer is greater than the right, the loop is exited. In order to have the pivot in its correct sorted place, the pivot is swapped with the element denoted by the low pointer. This low pointer, which signifies the division of the list, is returned to the recursive QuickSort algorithm. The algorithm takes this divider and makes two recursive calls. The first call is on the left sub-list and the second call is on the right sub-list. Once the algorithm partitions the left and right sub-lists completely, and the base cases are met (when the low index is greater than or equal to the high index), the list is sorted. The outer recursive function will then take the sorted list and return the kth element. As for runtime complexity, the partition method iterates through the entire sub-list of n size. Thus, g(n) = n for the recursive function. Since two recursive methods are made, a = 2. As for T(f(n)) it could actually be one of two values: T(n/2) or T(n -1). If the pivot selects the smallest or largest element, each time, then the index returned will be the last or first index of the segment. This means that one recursive call will continue to evaluate the list truncated by one element and the other recursive call will reach the base case. Thus, the recurrence relation would be T(n) = T(n - 1) + T(1) + n yielding T(n) = (n^2)/2 + 3n/2 – 1 in the worst case. The average and best case would be T(n) = 2T(n/2) + n, yielding T(n) = nlogn + n. Thus, the worst case would be O(n^2). The average and best case would be O(nlogn).

Modified QuickSort

The modified QuickSort is similar to QuickSort because it has an outer function which receives the list and k. This outer function passes the values, as well as 0 and the length of the list minus one, to the inner recursive function. The inner recursive function acts similar to binary search. Based on the index returned from the partition and k, the algorithm will either partition and search the left sub-list or right sub-list. If k matches the index returned from the partition method, then the algorithm returns the element at index k. If k is less than the index returned, then a new recursive call is made with the high index updated as the returned index minus one. If k is greater than the index returned, then a new recursive call is made with the low index updated as the returned index plus one. The left or right sub-list will continue to be partitioned until the kth element is found. As for the runtime complexity, due to the partition method, g(n) = n. Similar to the original QuickSort, if the pivot is repeatedly the smallest or largest element, each recursive call will evaluate n – 1 element. Thus, the recurrence relation, in the worst case would be T(n) = T(n - 1) + n, which yields T(n) = ((n(n+1)) / 2). The average case would continue to halve the list creating an average recurrence relation of T(n) = T(n/2) + n, which yields T(n) = 2n - 1. The best case scenario would consist of k matching the index returned from the partition method, in which T(n) = n. Thus, no further recursive calls would be made, yielding T(n) = n. In all, the big O-notation would be O(n^2) for the worst case and O(n) for the average case and best case.

**Part2:**

Iterative Modified QuickSort

The iterative modified QuickSort, like the other methods, receives the list and k. This function acts very similar to binary search’s iterative function. The main difference, however, is that the “middle” index is based on the index returned from the partition. Since the method is iterative, the main loop of the algorithm will continue until the kth value matches the index returned from the partition. Initially, the partition method is called, and the index returned is assigned to the variable middle. This middle value is then compared with k. If the middle equals k, then the element at middle is returned. If the value at middle is greater than k, then the high index is assigned to middle minus one. If the value at the middle is less than k, then the low index is assigned to middle plus one. Essentially, the high and low index are adjusted accordingly for the next partition method to be called. As for the runtime complexity of the algorithm, due to the partition, g(n) = n. In the worst case, the pivot will repeatedly be the lowest or highest element of the sub-list and thus would continue to decrease the number of elements to be sorted by n – 1. Since each iteration of the while loop performs a partition, n number of comparisons will be made. This would yield T(n) = n + (n - 1) + (n - 2) + … . This equates to T(n) = (n(n + 1)/2) and consequently O(n^2). In the average case, the algorithm iterates through n elements in the partition. As the algorithm continues, the number of elements iterated through halves each time until k equals the pivot returned. Thus, T(n) = n + (n/2) + (n/4) + (n/8) + (n/16) + ……. yielding T(n) = 2n - 1and consequently O(n). In the best case, the kth element would be the index returned from the partition, and thus only do n comparisons. This would yield T(n) = n and consequently O(n).

QuickSort with Stack

The QuickSort with stack function is similar to how the recursive QuickSort is implemented with activation records. For this function the list and k are passed as parameters. Initially, a stack is created with a Record object. The Record class contains attributes of a low and high index, mimicking the parameters of the recursive method. The algorithm first starts by creating a Record object with low assigned to 0 and high assigned to the last index of the list to be sorted. This Record object is popped to a stack. Then, the algorithm first checks if the current record has a low attribute greater than or equal to the value assigned to the high attribute. If low is greater than or equal to high, then the current record is removed from the stack. This comparison mimics the base case of the recursive algorithm. If the value assigned to low is less than the value assigned to high, then the list is partitioned. Based off the value returned, two Record objects are created and pushed to the stack. The Record objects’ low and high values are assigned based on the sub-list and right sub-list. The function continues to add Record objects and check their low and high values until the stack is empty. With the stack completely empty, the kth element is returned from the now sorted list. As for the runtime complexity, due to the partition function, g(n) = n. In the worst case, the pivot would repeatedly be the lowest or highest element of the sub-list, causing the sub-list to decrease by n – 1. This would equate to T(n) = n + (n - 1) + (n - 2) + … which simplifies to T(n) = (n(n + 1)/2) and O(n^2). The average and best case would be that the partition algorithm continues to return the middle most index, resulting in the segment to be halved. Thus, for each partition called, two more partitions will later be called, each on (n/2). Then, for the (n/2), they will be further halved, resulting in four (n/4) to be partitioned. Drawing this tree, yields T(n) = nlogn + n which simplifies to O(nlogn). It is important to note that when describing the tree above, the nodes do not represent the recursive calls, but rather Record objects created, that will later initiate a partition.

**Shared Design**

Both Part 1 and Part 2 include different sorting algorithms that will, in general, sort the list and return the kth element. As for the rest of the program, I divided it into two parts: one that would allow users to enter their own lists and one that would run automated tests. For the automated tests, I created five tests for lists of size 100, 500, 1000, 2000, and 3000. Each test calculates the performance time and number of comparisons for each of the functions described above. Additionally, for each list size, three sub tests are conducted for unsorted, nearly sorted, and sorted lists. This allows the strengths and weaknesses of the algorithms to be evaluated. Each sub-test was further divided into three more sub-tests with k equal to 0, the middle index, and the last index. I thought that the k value was extremely important since for some methods, like the modified QuickSort, the program can terminate immediately if the k value equals the index returned from the partition function. The tests and their results are described in further detail in the Experimental Results section.

**Calculations of Comparisons**

In order to correctly calculate the number of comparisons made by each algorithm on a list, L, three tests were performed on each of the algorithms. Test Case 1 had L= [1], Test Case 2 had L= [1,2], and Test Case 3 had L = [1,2,3]. The number of comparisons were calculated by hand and also reported through the sum of the global counter. This counter was used in each algorithm to count the number of comparisons. Through these test cases, I will be able to ensure that the global counter was calculating the correct number of comparisons. Once all tests were completed and the global counter’s result matched the number of comparisons I calculated, I was able to rely on the counter for larger list lengths. With these tests and calculations, I was able to identify if the number of comparisons matched the analytical runtime complexities seen in practice.

**Bubble Sort**

Case 1:

L = [1]

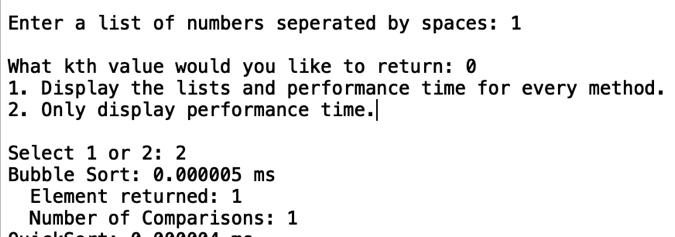
k = 0

Calculation:

Bubble: L = [1], k = 0, endIndex = 0

Base case met – 1 comparison

Total: 1 comparison



Case 2:

L = [2,1]

k = 0

Calculation:

Bubble: L = [2, 1], k = 0, endIndex = 1

Base case – 1 comparison

For loop initial – 1 comparison

Left, right comparison – 1 comparison

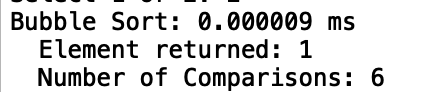
For loop – 1 comparison

If swapped – 1 comparison

Bubble: L = [1,2], k = 0, endIndex = 0

Base case – 1 comparison

Total: 6 comparisons



Case 3:

L = [3, 2, 1]

k = 0

Calculation:

bubble: L = [3, 2, 1], k = 0, endIndex = 2

Base case – 1 comparison

For loop initial – 1 comparison

Left, right comparison – 1 comparison

For loop – 1 comparison

Left, right comparison – 1 comparison

For loop – 1 comparison

If swapped – 1 comparison

bubble: L = [2,1,3], k = 0, endIndex = 1

Base case – 1 comparison

For loop initial – 1 comparison

Left, right comparison – 1 comparison

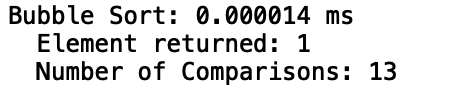
For loop – 1 comparison

If swapped – 1 comparison

bubble: L = [3,2,1], k = 0, endIndex = 0

Base case – 1 comparison

Total: 13 comparisons



**QuickSort**

Case 1:

L = [1]

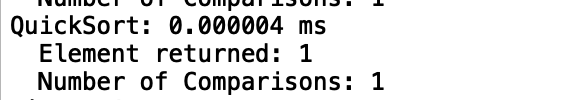
k = 0

Calculation:

quick\_sort: L = [1], k = 0

Base case met – 1 comparison

Total: 1 comparison



Case 2:

L = [2,1]

k = 0

Calculation:

quick\_sort: L = [2,1], k = 0, low = 0, high = 1

Base case – 1 comparison

Partition List = [1,2], low = 0, right = 0

Partition While loop initial – 1 comparison

First Inner While loop – 2 comparisons

Second Inner While loop – 1 comparison

Swap initial comparison – 1 comparison

End of while loop – 1 comparison

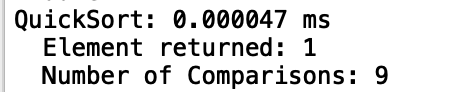
quick\_sort: L = [1,2], k = 0, low = 0, high = 0

Base case – 1 comparison

quick\_sort: L = [1,2], k = 0, low = 2, high = 1

Base case – 1 comparison

Total: 9 comparisons



Case 3:

L = [3, 2, 1]

k = 0

Calculation:

quick\_sort: L = [3,2,1], k = 0, low = 0, high = 2

Base case – 1 comparison

Partition list = [3,1,2], low = 0, right = 1

Parition While loop initial – 1 comparison

First Inner while loop – 1 comparison

Second Inner while loop – 1 comparison

Swap condition= 1 comparison

L= [1,3,2], low = 1, right = 0

Partition While loop – 1 comparison

Replacing Pivot = [1,2,3]

quick\_sort: L = [1,2,3], low = 0, high = 0

Base case – 1 comparison

quick\_sort: L = [1,2,3], low = 2, high = 2

Base case – 1 comparison

Total – 8 comparisons



**Modified QuickSort**

Case 1:

L = [1]

k = 0

Calculation:

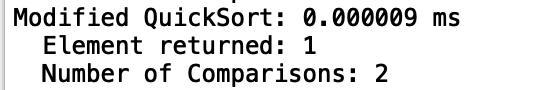
modified\_quick: L = [1], k = 0, low = 0, high = 0

Partition: L = [1], low = 0, right = -1

Partition initial while loop – 1 comparison

Base case – 1 comparison

Total: 2 comparisons



Case 2:

L = [2, 1]

k = 0

Calculation:

modified\_quick: L = [2,1], k = 0, low = 0, high = 1

Partition: L= [1,2], low = 0, right = 0

Partition while loop initial – 1 comparison

First while loop – 2 comparisons

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2], middle = 1

k equal to middle – 1 comparison

k less than middle – 1 comparison

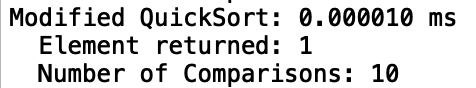
modified\_quick: L = [1,2], k = 0, low = 0, high = 0

Partition: L = [1,2], low = 0, right = -1

Partition while loop initial – 1 comparison

k equal to middle – 1 comparison

Total: 10 comparisons



Case 3:

L = [3, 2, 1]

k = 0

Calculation:

modified\_quick: L = [3, 2, 1], k = 0, low = 0, high = 2

Partition: L= [3, 1, 2], low = 0, right = 1

Partition while loop initial- 1 comparison

First while loop – 1 comparison

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2, 3], middle = 1

k equal to middle – 1 comparison

k less than middle – 1 comparison

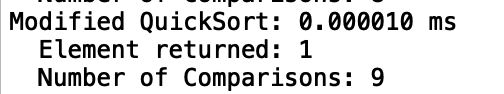
modified\_quick: L = [1, 2, 3], k = 0, low = 0, high = 0

Partition: L = [1,2], low = 0, right = -1

Partition while loop initial – 1 comparison

k equal to middle – 1 comparison

Total: 9 comparisons



**QuickSort with a Stack**

Case 1:

L = [1]

k = 0

Calculation:

quick\_sort\_with\_stack: L = [1], k = 0

Stack: Record: low = 0, high = 0

Initial while loop – 1 comparison

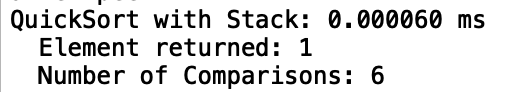
Stack peek – 1 comparison

Low, high - 1 comparison

Stack pop – 2 comparisons

While loop – 1 comparison

Total: 6 comparisons



Case 2:

L = [2, 1]

k = 0

Calculation:

quick\_sort\_with\_stack: L = [2, 1], k = 0

Stack: Record 1: low = 0, high = 1

Initial while loop – 1 comparison

Stack peek – 1 comparison

Low, high comparison – 1 comparison

Partition: L = [2,1], low = 0, right = 0

Partition while loop initial – comparison

First while loop – 2 comparisons

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2], middle = 1

Stack pop – 2 comparisons

Record 1B: low = 2, high = 1

Stack push – 1 comparison

Record 1A: low = 0, high = 0

Stack push – 1 comparison

Stack now has: Record 1A, Record 1B

While loop – 1 comparison

Stack peek – 1 comparison

Low, high comparison – 1 comparison

Stack pop – 2 comparisons

Stack now has: Record 1B

While loop – 1 comparison

Stack peek – 1 comparison

Low, high comparison – 1 comparison

Stack pop – 2 comparisons

Stack now has: None

While loop – 1 comparison

Total: 24 comparisons



Case 3:

L = [3, 2, 1]

k = 0

Calculation:

quick\_sort\_with\_stack: L = [3, 2, 1], k = 0

Stack: Record 1: low = 0, high = 2

Initial while loop – 1 comparison

Stack peek – 1 comparison

Low, high comparison – 1 comparison

Partition: L = [3, 1, 2], low = 0, right = 1

Partition while loop initial- 1 comparison

First while loop – 1 comparison

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2, 3], middle = 1

Stack pop – 2 comparisons

Record 1B: low = 2, high = 2

Stack push – 1 comparison

Record 1A: low = 0, high = 0

Stack push – 1 comparison

Stack now has: Record 1A, Record 1B

While loop – 1 comparison

Stack peek – 1 comparison

Low, high comparison – 1 comparison

Stack pop – 2 comparisons

Stack now has: Record 1B

While loop – 1 comparison

Stack peek – 1 comparison

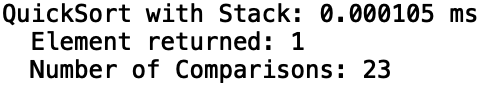
Low, high comparison – 1 comparison

Stack pop – 2 comparisons

Stack now has: None

While loop – 1 comparison

Total: 23 comparisons



**Iterative Modified QuickSort**

Case 1:

L = [1]

k = 0

Calculation:

iterative\_select\_modified\_quick: L = [1], k = 0

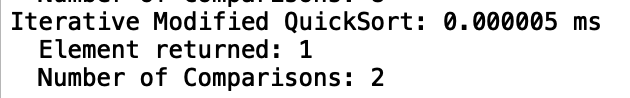
Partition: L = [1], low = 0, right = -1

Partition while loop initial - 1 comparison

L = [1], middle = 0

While loop check – 1 comparison

Total: 2 comparisons



Case 2:

L = [2,1]

k = 0

Calculation:

iterative\_select\_modified\_quick: L = [2,1], k = 0, low = 0, high = 1

Partition: L = [1,2], low = 0, right = 0

Partition while loop initial- 1 comparison

First while loop – 2 comparisons

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2], middle = 1

While loop – 1 comparison

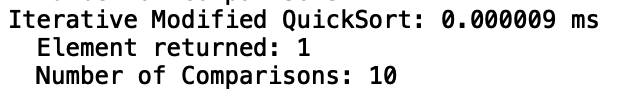
k less than middle – 1 comparison

Partition: L = [1,2], low = 0, right = -1

Partition while loop initial – 1 comparison

While loop – 1 comparison

Total: 10 comparisons



Case 3:

L = [3, 2, 1]

k = 0

Calculation:

iterative\_select\_modified\_quick: L = [3, 2, 1], k = 0, low = 0, high = 2

Partition: L = [3, 1, 2], low = 0, right = 1

Partition while loop initial- 1 comparison

First while loop – 1 comparison

Second while loop – 1 comparison

Swap condition – 1 comparison

Partition while loop – 1 comparison

L = [1, 2, 3], middle = 1

While loop – 1 comparison

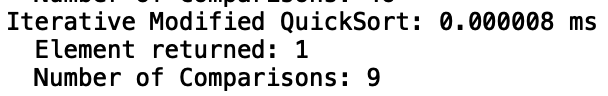
k less than middle – 1 comparison

Partition: L = [1,2], low = 0, right = -1

Partition while loop initial – 1 comparison

While loop – 1 comparison

Total: 9 comparisons



Based on the test cases above, it is evident that the programed counter is correctly counting the number of comparisons. In all instances, the hand calculated comparisons matched the counters. Due to this, the experimental results display the comparisons and performance time for the algorithms. Analysis of the physical implementation versus the theoretical runtime complexities will be demonstrated below.

**Experimental Results**

It is important to note that for each trial, the sorting methods received the same copies of the list. For Part 1 and Part 2, each sorting algorithm has four test cases. Test Case 1 compares the running time for unsorted lists, Test Case 2 compares the running time for nearly sorted lists, Test Case 3 compares the running time for sorted lists, and Test Case 4 computes the number of comparisons for each algorithm. Each test case sorts lists of size 100, 500, 1000, 2000, and 3000. Additionally, for all tests, the first trial had k equal to 0, the second trial had k equal to the middle index of the array and the third trial had k equal to the last element. I thought that these test cases were crucial to later compare the running times between all the methods. I was also able to see how the data matched the theoretical runtime complexity functions. As for Test Case 4, the number of comparisons were calculated by the global. For Test Case 4, k was set to the last element in the list. The number of comparisons were plotted for sorted, unsorted, and nearly sorted elements. In order to see the number of comparisons versus the theoretical number of comparisons, the bounding Big O notation functions are graphed as well.

NOTE: The tests below summarize results obtained from the automated tests. View Appendix B to see examples of these tests and customized tests.

**Part 1:**

**Bubble Sort**

Case 1 (“Unsorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 2 (“Nearly Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 3 (“Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000





Test Case 4 (“Comparison”):

k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000





As the cases demonstrate, Bubble Sort’s efficiency can range dramatically. For 3000 elements, Bubble Sort can have a runtime complexity as high as 0.94 ms or as low as 0.000412 ms. The graph summarizing the performance times from Test Case 1 to Test Case 3 also demonstrates the best and worst cases for Bubble Sort. When the list is sorted, the method only needs to go through the entire list once before ending the algorithm. This is why the line representing the Bubble Sort’s performance on the sorted list is low when compared to the other list types and linear. Additionally, Test Case 4 fully depicts how the algorithm performed when compared to the Big-O notation functions. As I had hypothesized, the best case scenario of Bubble sort is O(n) if the list is completely sorted. Furthermore, both the unsorted and nearly sorted functions are exactly on O(n^2). In fact, Bubble Sort was the only algorithm whose runtime complexities almost exactly matched the theoretical complexities.

**QuickSort**

Case 1 (“Unsorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 2 (“Nearly Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 3 (“Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000





Test Case 4 (“comparison”):

k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



From the graph summarizing Test Case 1 to Test Case 3’s performance, it is evident that the data is closely related. This was expected since it is very rare for a pivot to continuously be the smallest or largest value. However, it is important to note that the QuickSort performed on the sorted list had a faster performance time then the other lists. This is due to the pivot’s selection. In the program, I have the pivot continuously picked as the middle element. Since the list is sorted, the list is being perfectly halved for each recursive call, unlike the unsorted list. Thus, the best case runtime complexity of O(nlogn) is proven through the number of comparisons and performance time. Additionally, graph from Test Case 4 clearly shows that QuickSort, on average, has a runtime complexity of O(nlogn). Although the nearly sorted and unsorted number of comparisons grow at a faster rate of O(nlogn), they are nowhere near O(n^2). For these tests, the worst case runtime complexity is not evident due to the pivot selection. Thus, it safe to conclude that the actual number of comparisons matches the theoretical number of comparisons.

**Modified Quick Sort**

Case 1 (“Unsorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 2 (“Nearly Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 3 (“Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000





Test Case 4 (“Comparison”):

k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



The graph summarizing the results from Test Case 1 to Test Case 3 demonstrates the performance time for the modified QuickSort with unsorted, nearly sorted, and sorted lists. Just like QuickSort, the performance time for the sorted list is faster than the other lists. This proves the efficiency of selecting a pivot from the center. If I had decided to pick a pivot at the end, then the performance time for the sorted lists would have been monumentally longer. From the graph in Test Case 4, it is evident that the runtime complexity is closer to O(n) then the runtime complexity data collected from QuickSort. This further supports the notion that the average and best case runtime complexity for the modified QuickSort is O(n). Although the number of comparisons for all the lists was higher than O(n), it is important to note the meaning of Big-O notation. According to Big-O notation, there exists a constant that would make the function greater than T(n) for all n values greater than or equal to one. Based on the values gathered in the table, the constant is approximately two. This yet again aligns with the proposed equation of T(n) = 2n – 1. Additionally due to the pivot selection, the worst case runtime complexity was not evident.

**Part 2:**

**Quick Sort with a Stack**

Case 1 (“Unsorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 2 (“Nearly Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 3 (“Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000





Test Case 4 (“comparison”):

k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



The graph above summarizing the performance times from Test Case 1 to Test Case 3 demonstrates the close relationship between the sorted, unsorted, and nearly sorted lists. However, it is important to note that the nearly and unsorted lists took the longest period of time to sort. This is because the stack has to push all the objects representing the recursive calls and later pop them. QuickSort with a stack also has the lowest performance times for the sorted lists due to the pivot. By selecting the central pivot, the partition method evenly divided the list segment each time. The graph from Test Case 4 also clearly demonstrates that the runtime complexity for QuickSort with a stack is O(nlogn). If a line representing O(n^2) was graphed, the QuickSort with stack plots would be unnoticeable. It is also important to note that QuickSort with a stack has the highest runtime complexity, when compared with the other QuickSorts. This can be attributed to the stack’s implementation. Since recursion can do multiple tasks simultaneously, the stack is performing the tasks one by one. Thus, the number of comparisons is greater than O(nlogn), however the difference is not starkly noticeable. The QuickSort with stack may have been closer to O(n^2) for the sorted lists if the pivot had been selected at either end rather than the central element. However, the algorithm was created to prevent this scenario.

**Iterative, Modified Quick Sort**

Case 1 (“Unsorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 2 (“Nearly Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



Case 3 (“Sorted”):

Trial 1: k = 0

Trial 2: k = 49, 224, 499, 999, 1499

Trial 3: k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000

****



Test Case 4 (“Comparison”):

k = 99, 499, 999, 1999, 2999

List lengths = 100, 500, 1000, 2000, 3000



The graph above summarizing Test Case 1 to Test Case 3, demonstrates the average runtime complexity for iterative modified QuickSort with unsorted, nearly sorted, and sorted lists. Yet again, the iterative modified QuickSort performance times for the sorted lists were the fastest due to the pivot selection. Test Case 4’s graph demonstrates that the iterative modified QuickSort is very close to O(n). This supports the calculated theoretical runtime complexity of O(n). In fact, the plot for the sorted lists in Test Case 4’s graph is approximately two times the number of elements. This further supports the notion that T(n) = n + (n/2) + (n/4) + …. Since the sorted list is partitioned in half each time, it is essentially performing the number of comparisons described in the equation. As can be seen in this test along with the other QuickSort tests, the possibility of repeatedly having the pivot as the lowest or highest element is slim. However, it is still crucial to note the worst case in the event that it does occur.

**Worst Case Performance Time Comparison:**

The graphs below demonstrate the worst case performance times for all five of the algorithms. The data depicted on this graph are from Test Cases 1 to 3 for each algorithm. By graphing all the functions together, it is easier to view their relation to one another and their efficiency. The second graph still includes the same data from the first graph; however, it omits the performance time for Bubble Sort. In doing so, the relationship between the graphs is easier to identify.





By viewing the graphs above it is very clear that Bubble Sort had the worst runtime complexity. In theory, Bubble Sort has a worst case time complexity of O(n^2), the performance time graph clearly shows this trend. Furthermore, QuickSort and the QuickSort with a stack had an increasing growth rate. Thus, it is evident, that both algorithms had a runtime complexity of O(nlogn). It is important to note, however, that the QuickSort with a stack had the second longest performance time. This is because it has to perform each task one at a time. The graph also demonstrates that the iterative modified QuickSort and the modified QuickSort had essentially the same performance. This was expected since both algorithms use the same logic. Additionally, both the modified QuickSort and the iterative modified QuickSort grow linearly. This supports their believed theoretical runtime complexity of O(n).

**Worst Case Number of Comparisons:**

The table and graphs below depict the number of comparisons made by all five functions when sorting lists of length 100, 500,1000, 2000, and 3000. For each list, k was assigned to the last element of the list. Each test for the method was performed using the automated tests created in the program. The lists were all unsorted to identify how each function would respond. The second graph contains the same data from the first graph, but it has Bubble Sort removed in order to see the relationship between the other functions. Lastly, the number of comparisons was calculated by using a global counter. This global counter would increment based on the number of comparisons.





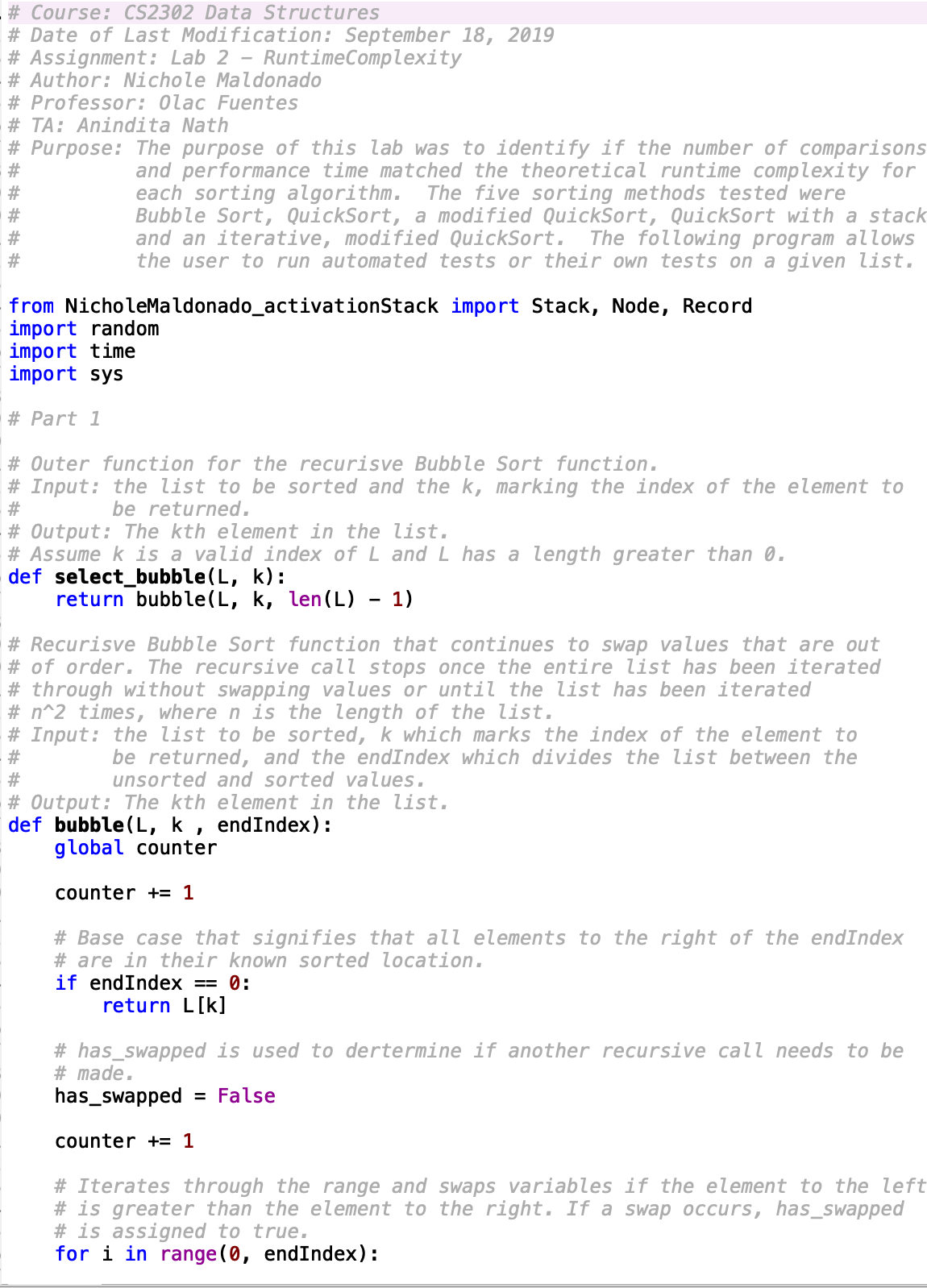
****

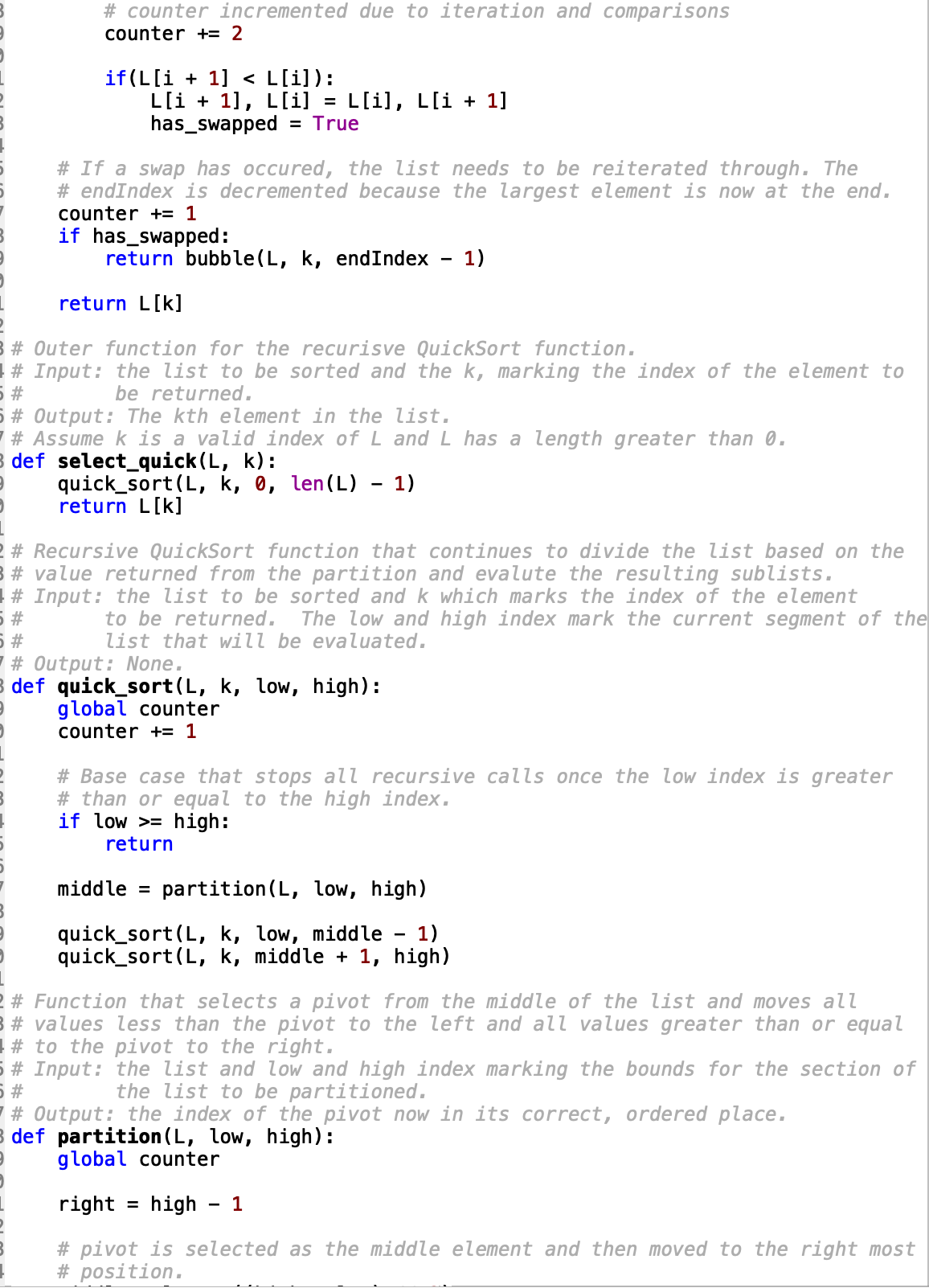
From the graphs it can be concluded that Bubble Sort had the largest number of comparisons. It was also the only function whose values closely corresponded to the theoretical runtime complexity. The second graph clearly shows that QuickSort and QuickSort with a stack had values nearest to O(nlogn). The iterative modified QuickSort and the modified QuickSort had comparison values closets to O(n). Although these algorithms did not exactly correspond to their theoretical runtime complexity, this demonstrates an important attribute of asymptotic notation. Asymptotic notation focuses on the growth rate of a bounding function. Thus, for Big-O notation, there exists some constant c that would make all the comparisons of a given function greater than or equal to the worst-case T(n). This constant is evident in the graphs from earlier tests cases.

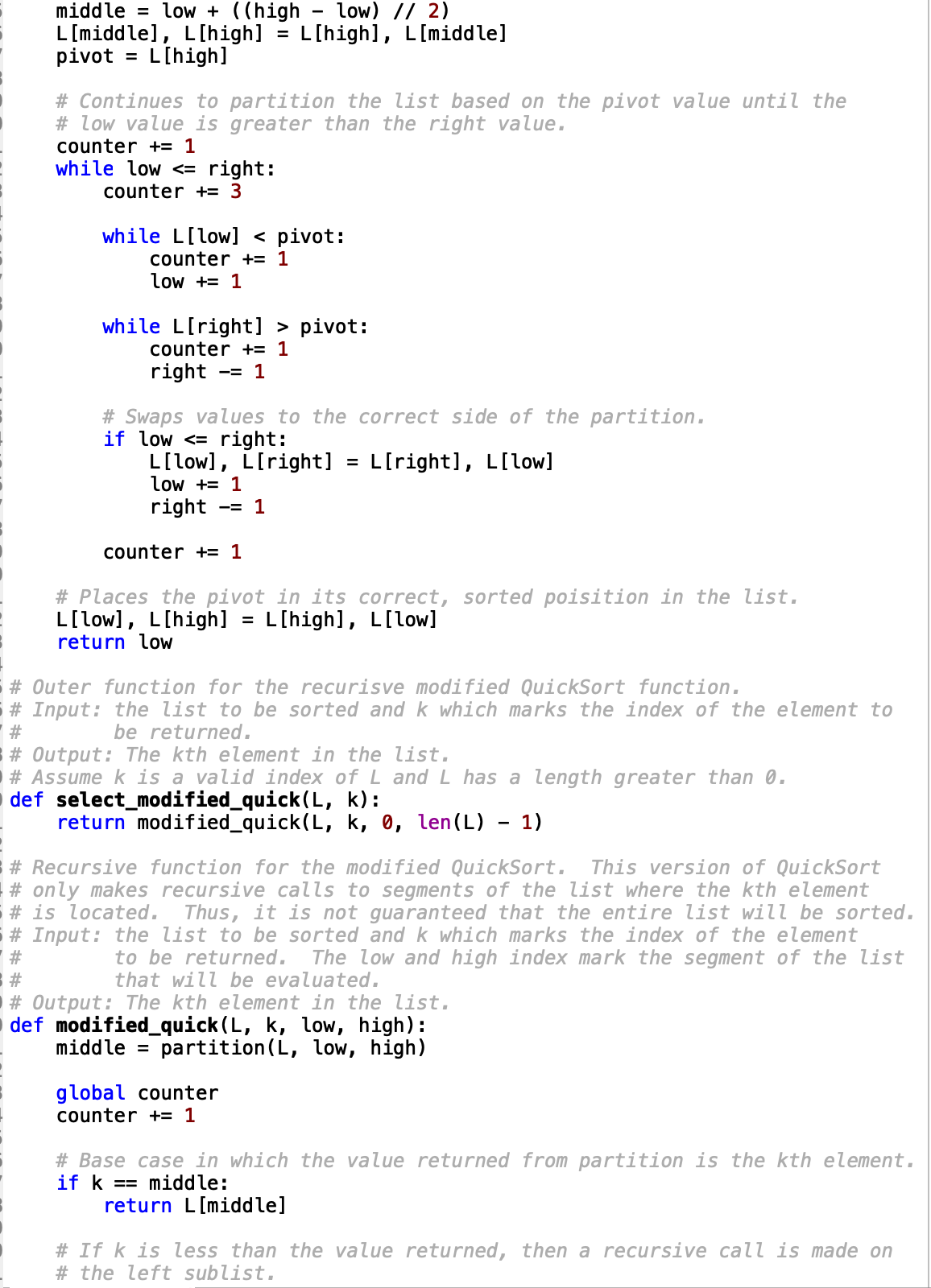
**Conclusion**

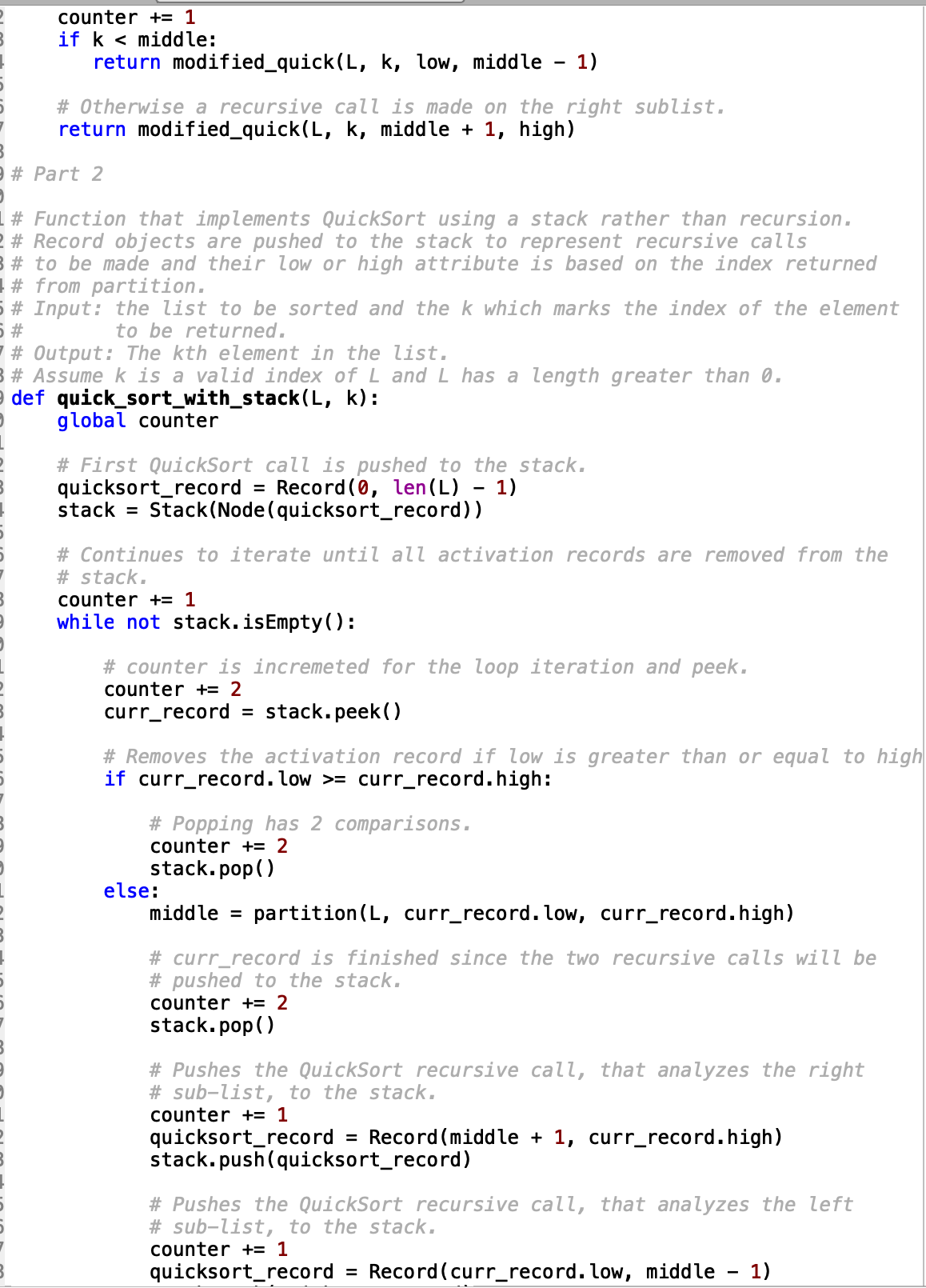
In all, I learned from the lab that different sorting algorithms have their own benefits and drawbacks. For example, Bubble Sort had the lowest performance times for the sorted lists. However, in all other instances, Bubble Sort had the highest performance times. In fact, Bubble Sort was the only sorting algorithm whose performance time reached 0.1 ms. Additionally, this lab gave me a better insight as to how QuickSort is implemented using activation records. Before completing this lab, I just viewed QuickSort as the function described by the code. However, after making my own stack implementation, I have a greater respect for how the code is internally implemented. At first, I added an instruction pointer attribute to the Record class, but later realized that it was unneeded. This lab also showed me a new way to retrieve the kth element from an unsorted list. Before I would have thought to sort the entire list and then retrieve the element. However, after implementing the modified QuickSort, I now know that it is a more efficient algorithm, especially if you do not need the list sorted in the end. One interesting notion that I found in the lab was how the number of comparisons can actually differ from the theoretical Big O-notation. Although the iterative modified QuickSort, in theory, had an average case of O(n), the values attained were higher. Lastly, I learned how to compare a program against the Big O-notation. When I was calculating the performance times, I realized that I needed to directly compare the results with the Big O-notation functions. Thus, I implemented a counter and was able to evaluate the number of comparisons for larger lists. In all, this lab was very helpful in allowing me to identify the runtime complexity for my own code and determine if my calculations were correct.

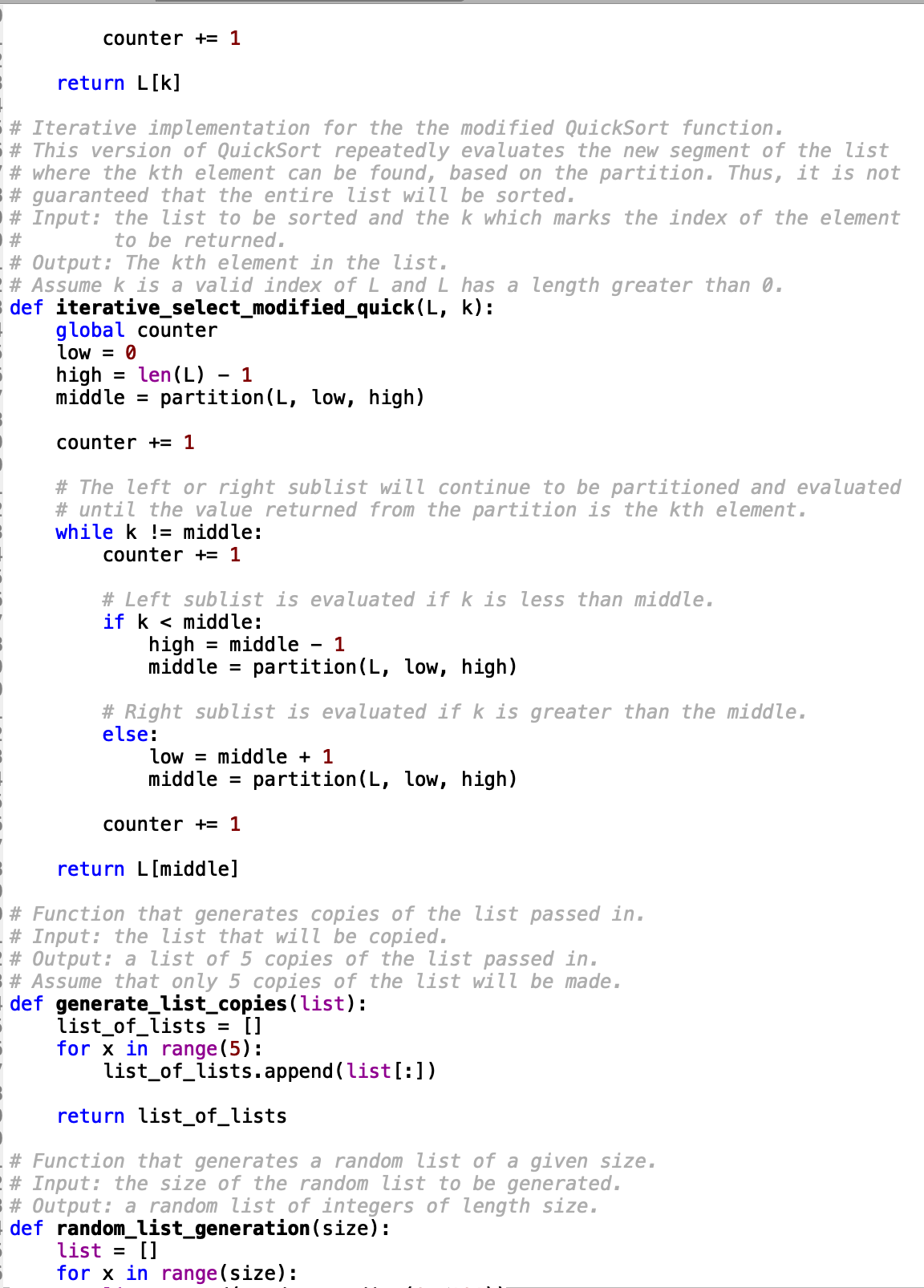
**Appendix A**

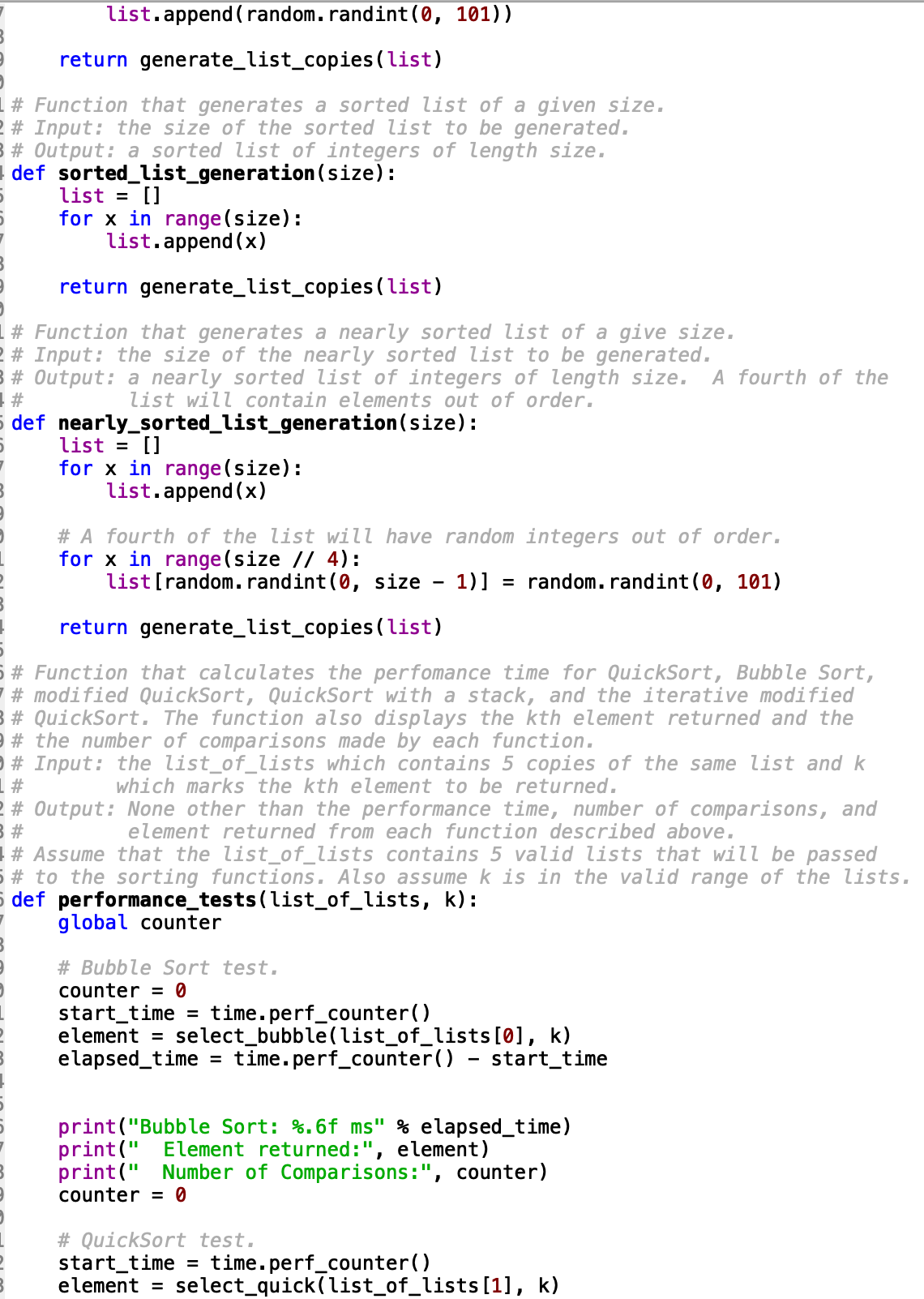
****

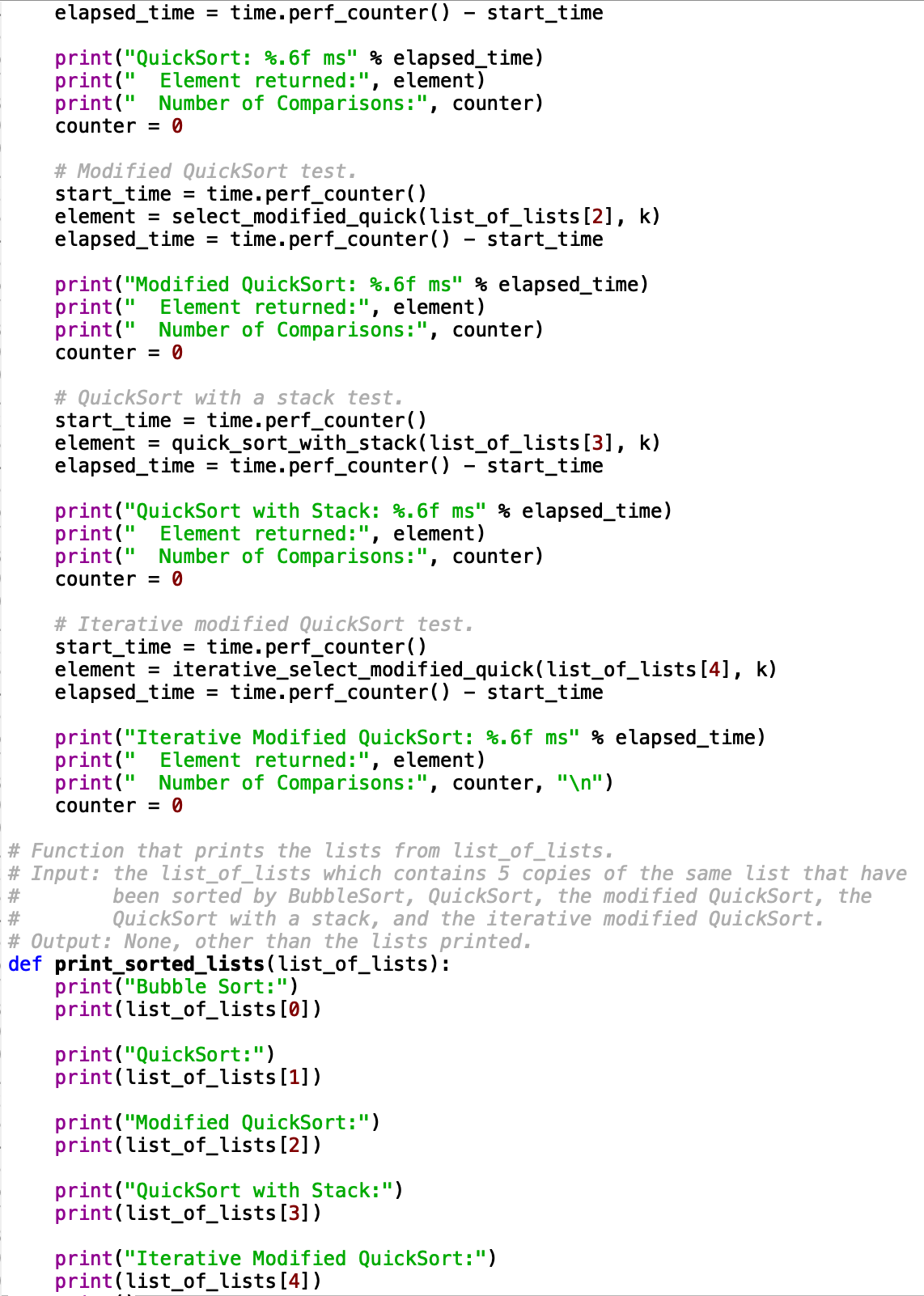
****

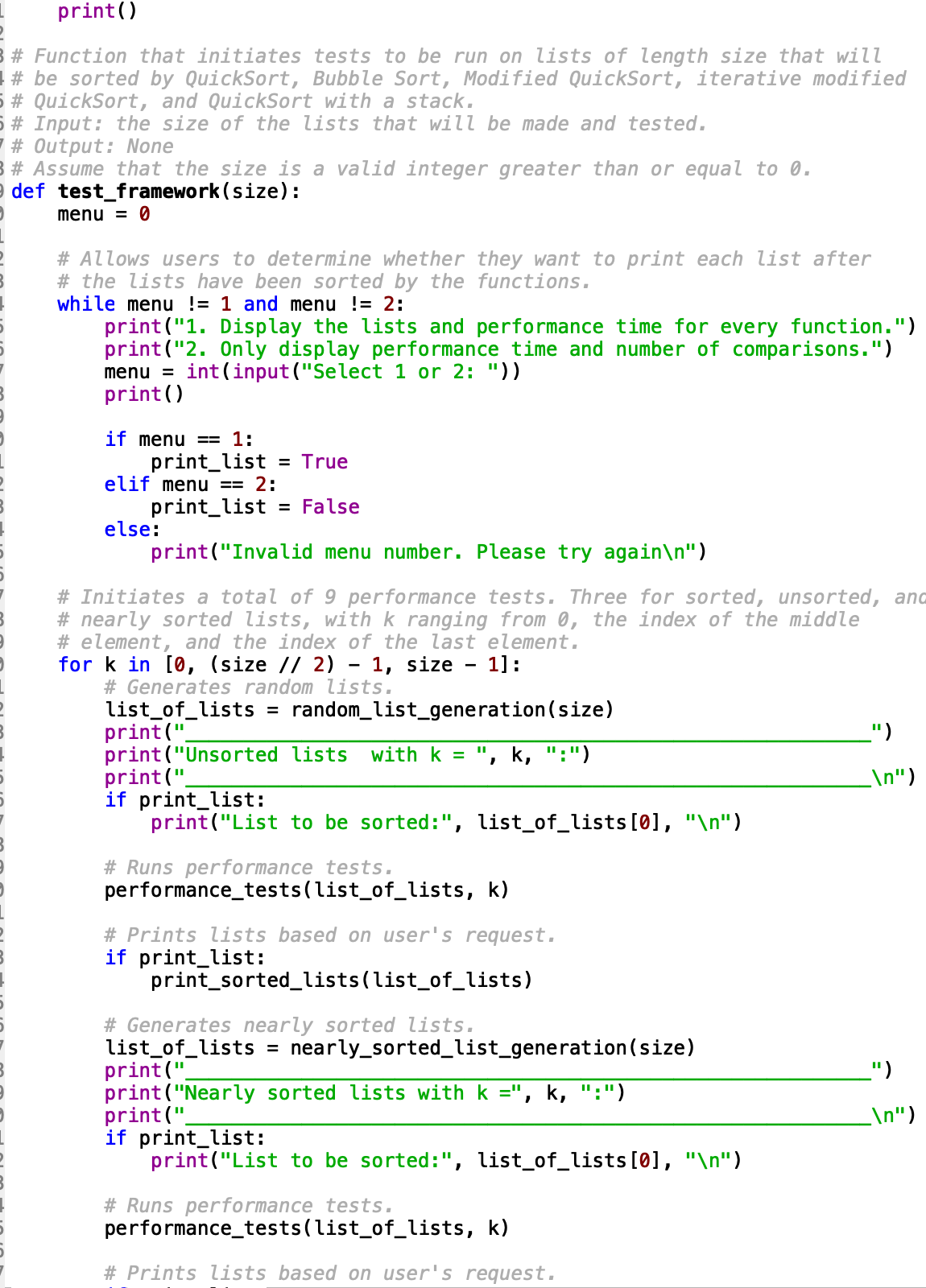
****

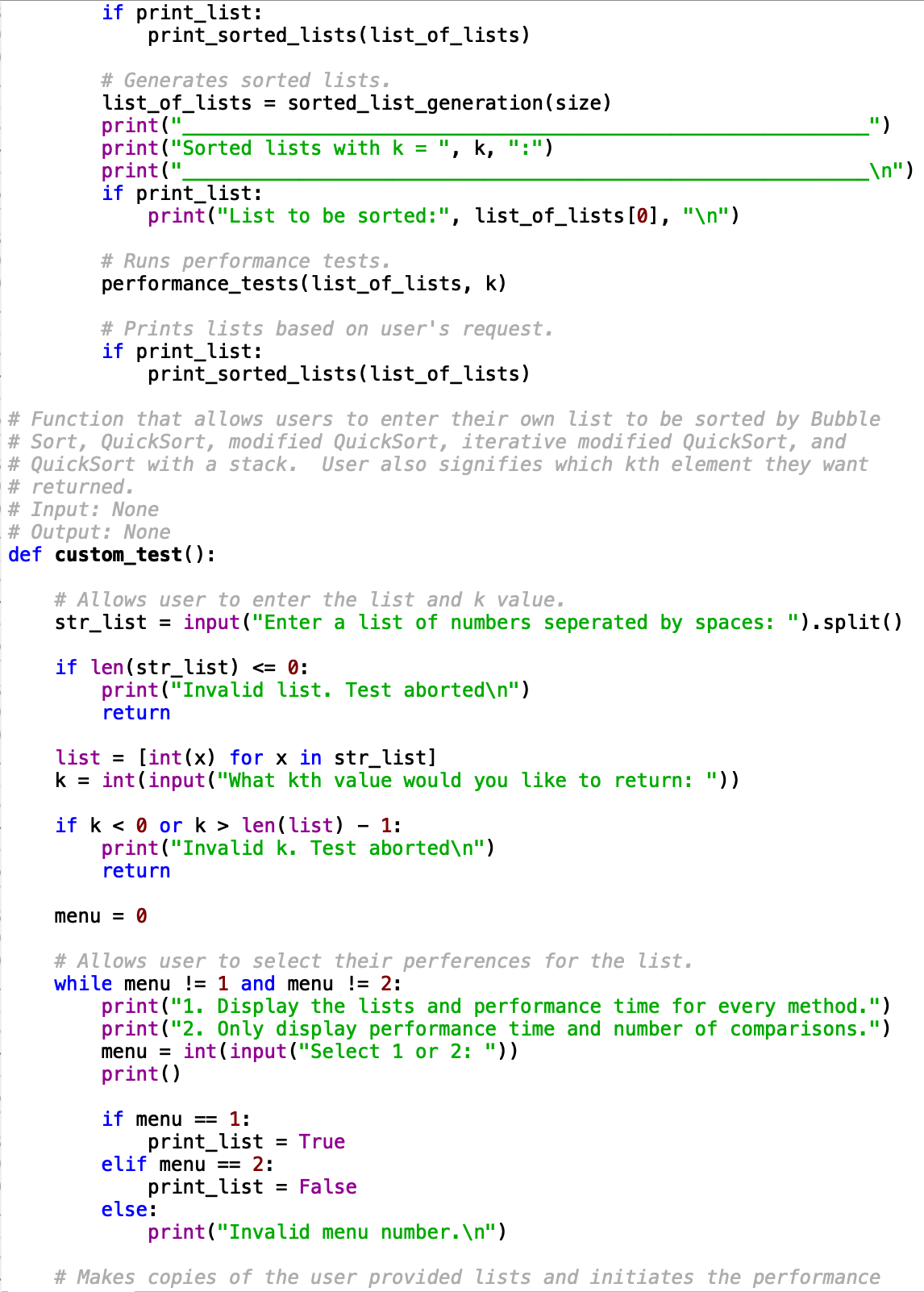
****

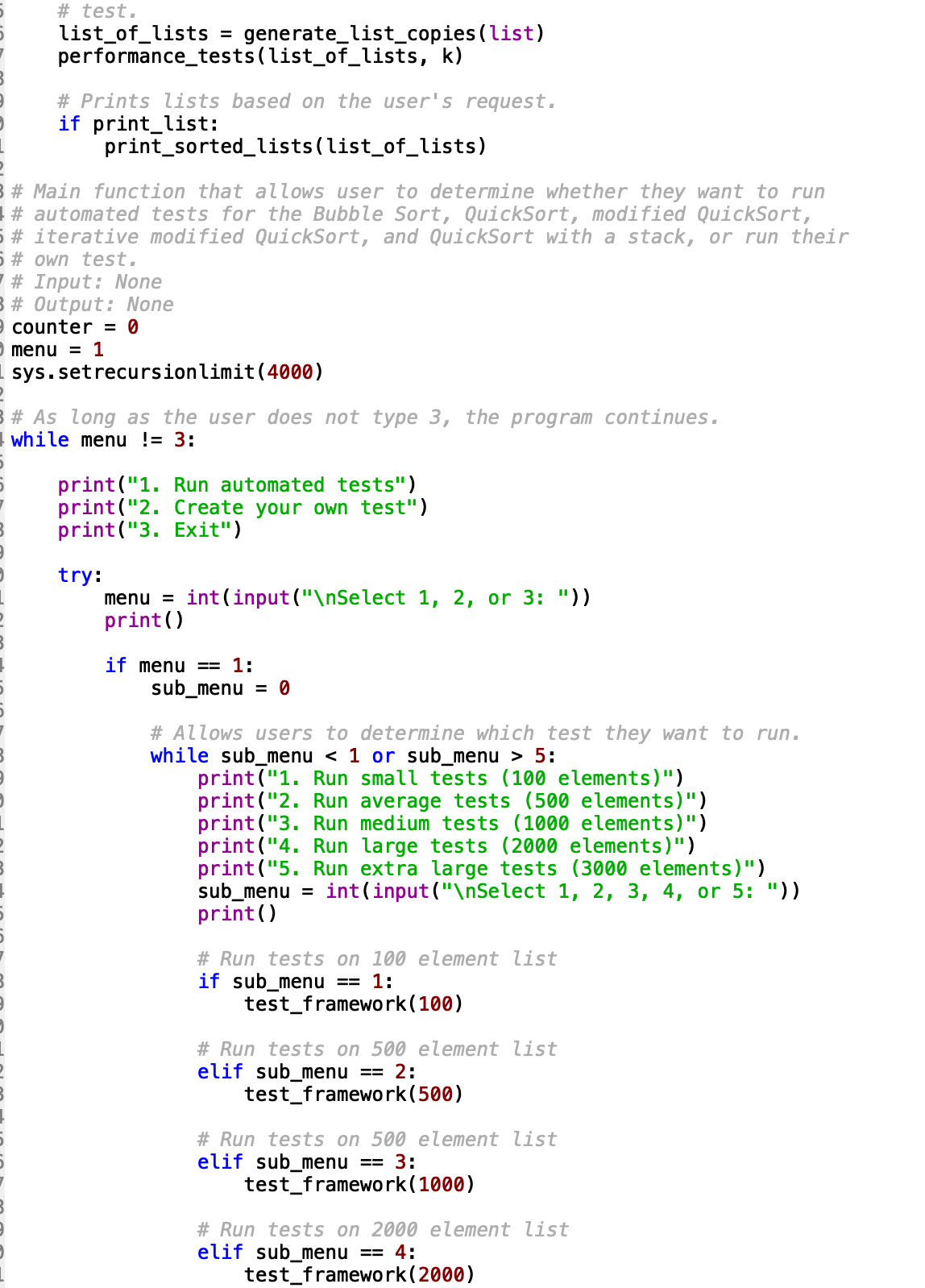
****

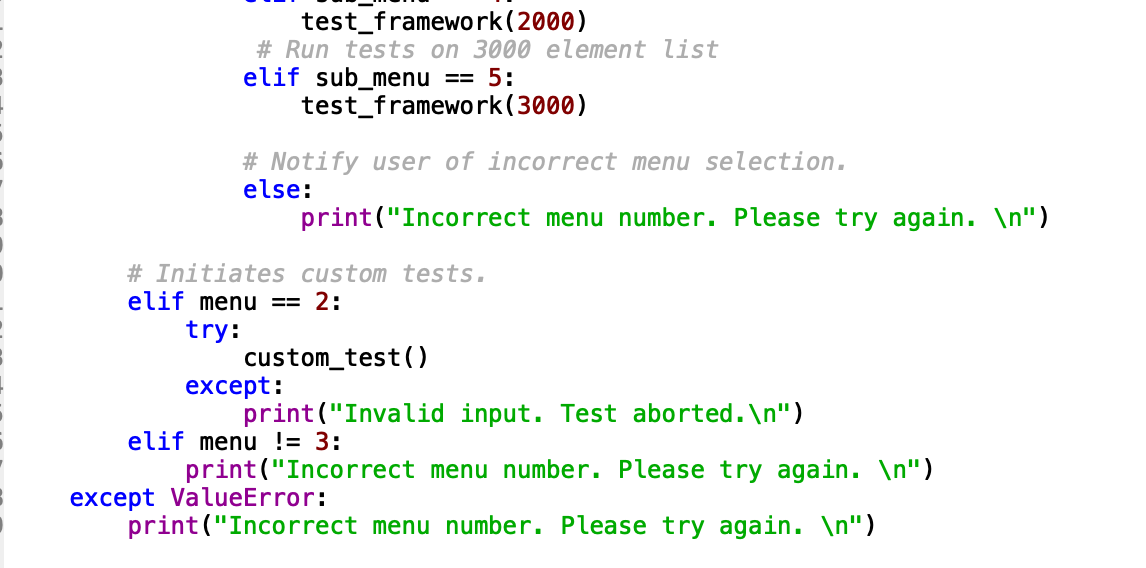
****

****

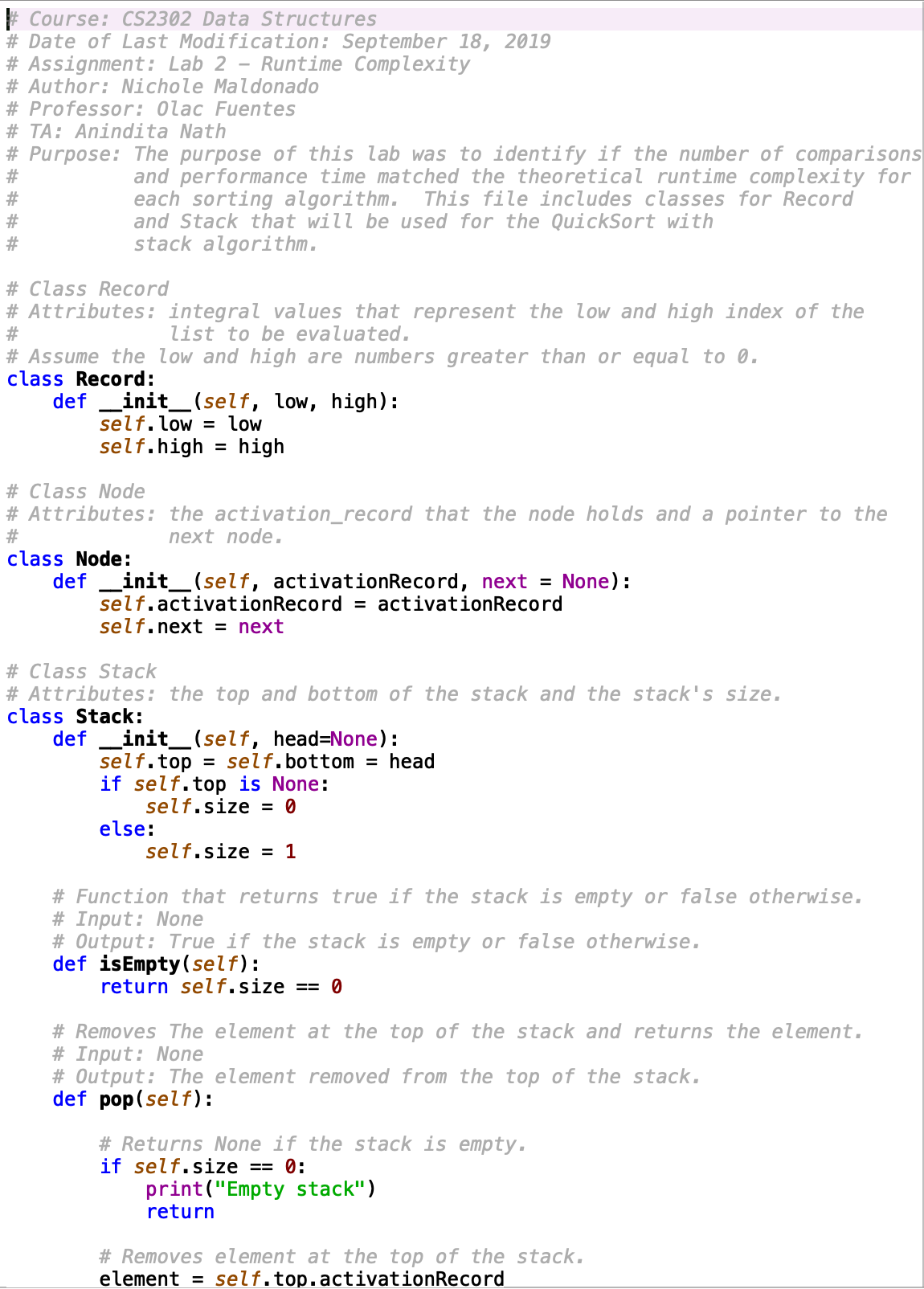
****

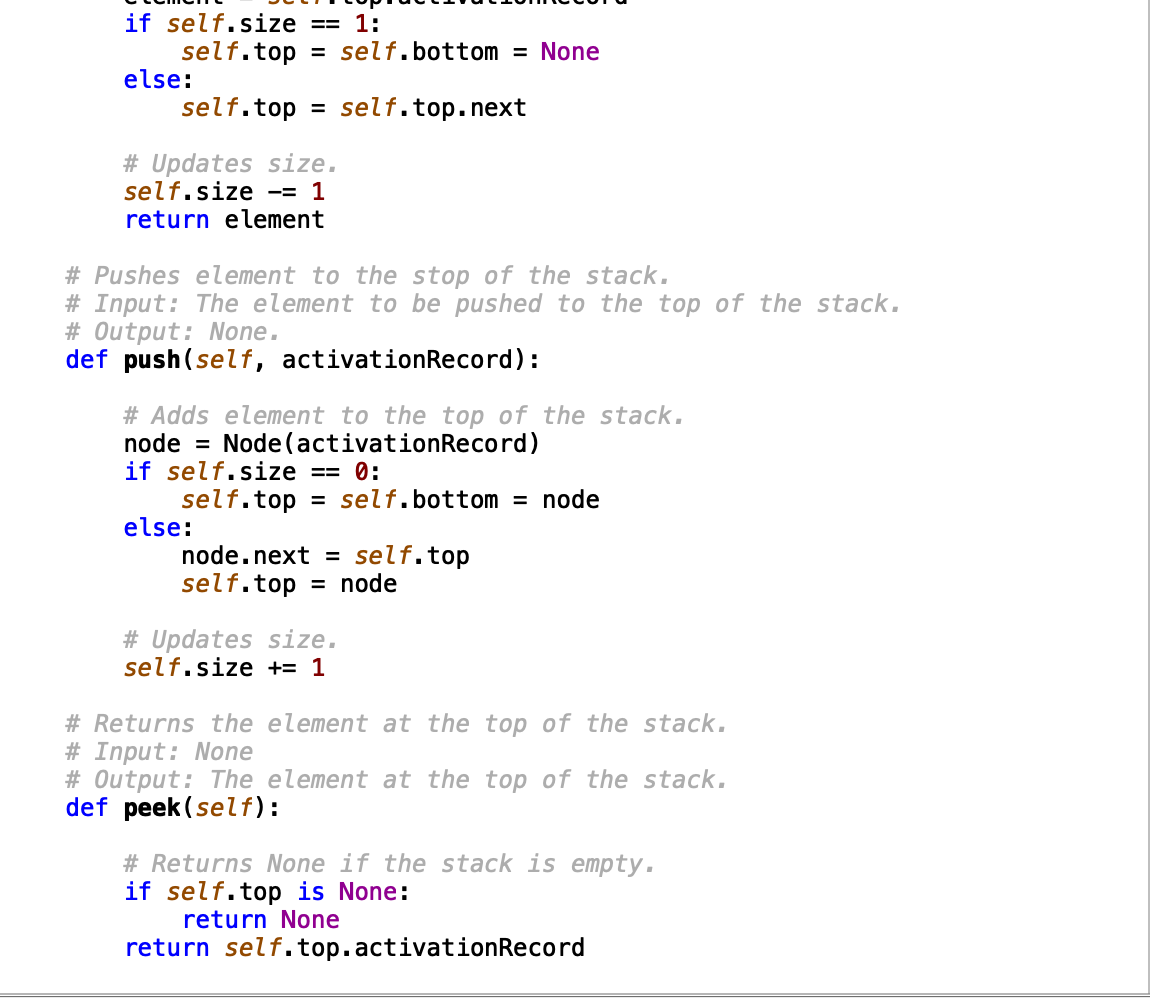
****

****

****

**Record Class and Stack Class:**

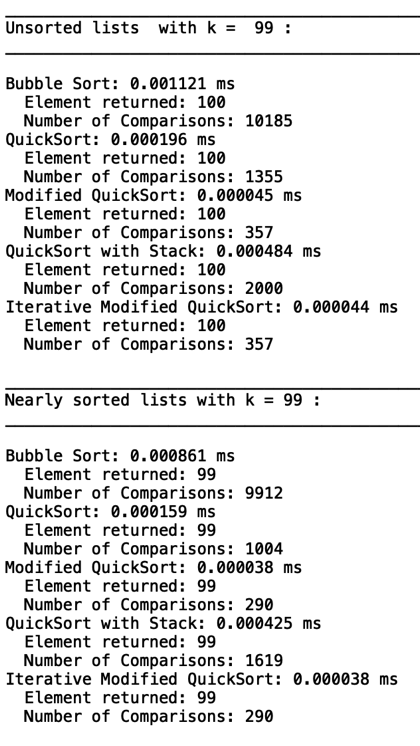
****

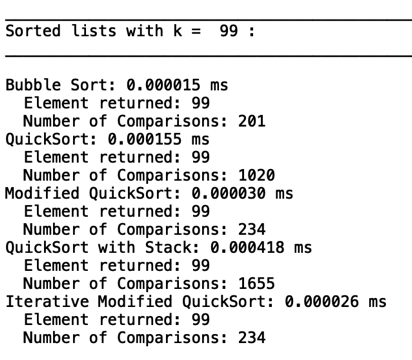
****

**Appendix B**

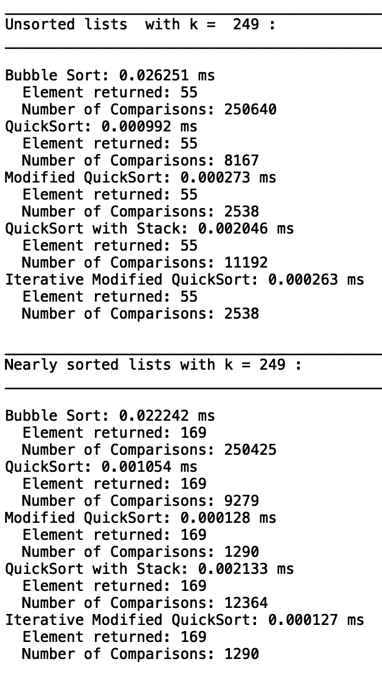
**Automated Test Run:**

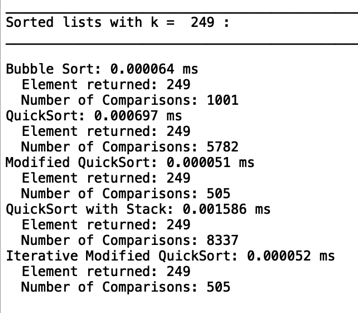
**100 elements:**

****

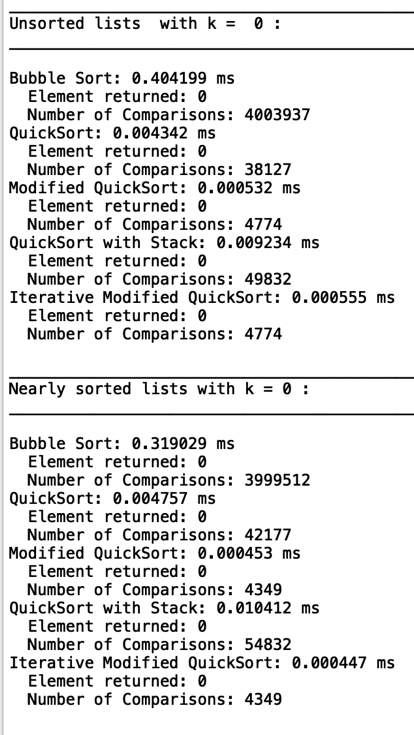
****

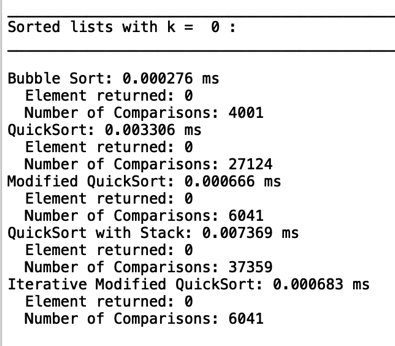
**500 elements:**

****

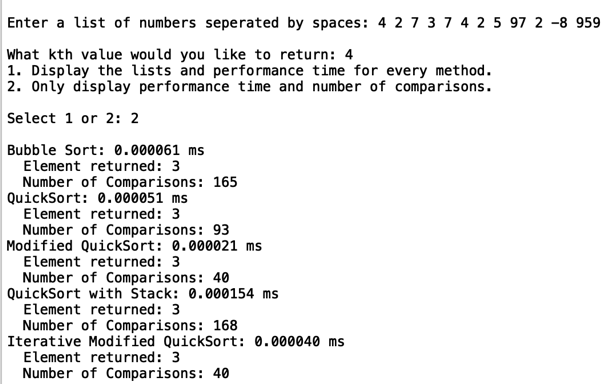
****

**2000 elements:**

****

****

**Custom Test:**

****

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.

X



-----------------------------------