**CS 2302 Data Structures**

**Fall 2019**

**Lab Report #2**

Due: September 24th, 2019

Professor: Olac Fuentes

TA: Anindita Nath

**Introduction**

For this lab, we were asked to find the kth smallest element in a list of integers using five sorting algorithm implementations. For this lab, it is key to remember the properties of lists, iteration and recursion, stacks, and commonly known sorting algorithms such as bubble sort and quicksort since they are required for all exercises. The main objective of this lab is to write a program that implements five sorting algorithms to sort a list of integers in such a way that the kth smallest element can be found for each operation.

**Proposed Solution Design and Implementation**

**Operation #1 (Bubble sort):**

For this operation, I implemented an iterative bubble sort algorithm that accepts a list of integers and sorts it in ascending order by traversing through the list and swapping adjacent elements if they are in the wrong order. This operation continues to traverse through the list until there is a whole pass without any swaps. At that point, the list is considered sorted. To do this, I implemented a Boolean flag that indicates whether the list is sorted or unsorted.

The big-O running time of this operation is **O(n2)** because it traverses through the list more than once to be able to sort it.

**Operation #2 (Quicksort):**

For this operation, I implemented a recursive quicksort algorithm that accepts a list of integers and the indices of the first and last elements of the list and sorts it in ascending order by partitioning the list into two sublists: the smaller sublist containing all elements that are less than the pivot and the larger sublist containing all elements that are greater than the pivot. After partitioning the list, this operation makes two recursive calls in order to sort the smaller and larger sublists. To do this, I implemented an additional function that takes care of partitioning the list and returning the pivot. This additional function, known as the partition function, sets the last element of the list as the pivot and places the pivot at its correct position in the sorted list by traversing through the list and swapping elements if they are in the wrong sublist.

Given that this operation is recursive, it has a recurrence relation of *T(n) = 2T(n/2) + n* because the original call makes two recursive calls, and the list is partitioned into two sublists, which means that the size of the recursive problem is one-half of the original list. By applying the Master Method, the big-O running time of this operation is **O(n log n)**.

**Operation #3 (Modified quicksort):**

For this operation, I implemented a modified version of the recursive quicksort algorithm that makes only one recursive call. Like quicksort, this one uses the partition function to partition the list into two sublists by a pivot. Unlike quicksort, this one does not make the two recursive calls to sort the two sublists. Instead, it checks if the element in position k is the pivot, an element in the smaller sublist, or an element in the larger sublist. If it is the first option, then the kth element has been found. If it is either the second or third options, then the operation will make a single corresponding recursive call to sort the sublist that contains the kth element. However, this operation will sort the corresponding sublist until the kth element matches the pivot of the corresponding sublist.

Given that this operation is also recursive, it has a recurrence relation of *T(n) = T(n/2) + n* because the original call only makes one recursive call, and the list is partitioned into two sublists, which means that the size of the recursive problem is one-half of the original list. By applying the Master Method, the big-O running time of this operation is **O(n)**.

**Operation #4 (Quicksort stack implementation):**

For this operation, I implemented a quicksort sorting algorithm as a stack. To do this, I created a stack that stores the list of integers and the indices of the first and last elements of the list. While the stack is not empty, this operation pops the top of the stack and stores it in a variable. That way, this operation can access the necessary information to use the partition function to partition the list into two sublists by a pivot. Then, the operation appends twice. The first time is to store the information to sort the smaller sublist, and the second time is to store the information to sort the larger sublist. This process is repeated until the stack becomes empty. At that point, the list has been sorted.

The big-O running time of this operation is **O(nlog n)** because it traverses through the list, or sublist, once to sort the pivot, and the list is continuously partitioned into two sublists until it has been completely sorted.

**Operation #5 (Modified quicksort iterative implementation):**

For this operation, I implemented the modified version of the quicksort algorithm using a while loop. To do this, I wrote inside the while loop the call to the partition function. This allows the operation to sort the pivot in each iteration. Then, the operation checks if the element in position k matches the pivot. If they match, then the kth element has been found. Otherwise, the operation checks if the kth element is in the smaller sublist or in the larger sublist. If it is the former, then the index of the last element of the list is updated so that the next iteration sorts the pivot in the smaller sublist. If it is the latter, then the index of the first element of the list is updated so that the next iteration sorts the pivot in the larger sublist. This operation will continuously sort the pivot in each subsequent sublist until the kth element matches one of these pivots.

The big-O running time of this operation is **O(n)** because it traverses through a single list, or sublist, once to sort the pivot.

For each operation, I implemented a function that accepts a list of integers and an integer value of k, which represents the position of the kth smallest element in the list, sorts the list accordingly, and returns the element in position k. Additionally, I included the total running time and the number of comparisons made over the course of one function call to each operation.

**Experimental Results**

**Operation #1:**

For this operation, I will test randomly generated lists of integers of sizes 10, 50, 100, 500, and 1000. Additionally, I will test that the operation returns the first element (k == 0) for the lists of sizes 10 and 50, the middle element (k == 49) for the list of size 100, and the last element (k == 499 and k == 999) for the lists of sizes 500 and 1000, respectively.

Case 1:

Size of list = 10

Position k of element to be found = 0



Case 2:

Size of list = 50

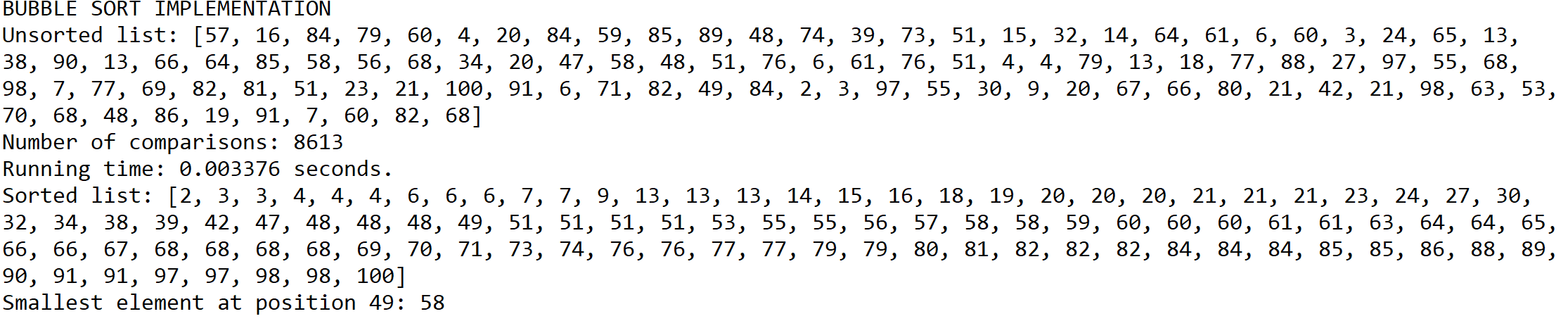
Position k of element to be found = 0



Case 3:

Size of list = 100

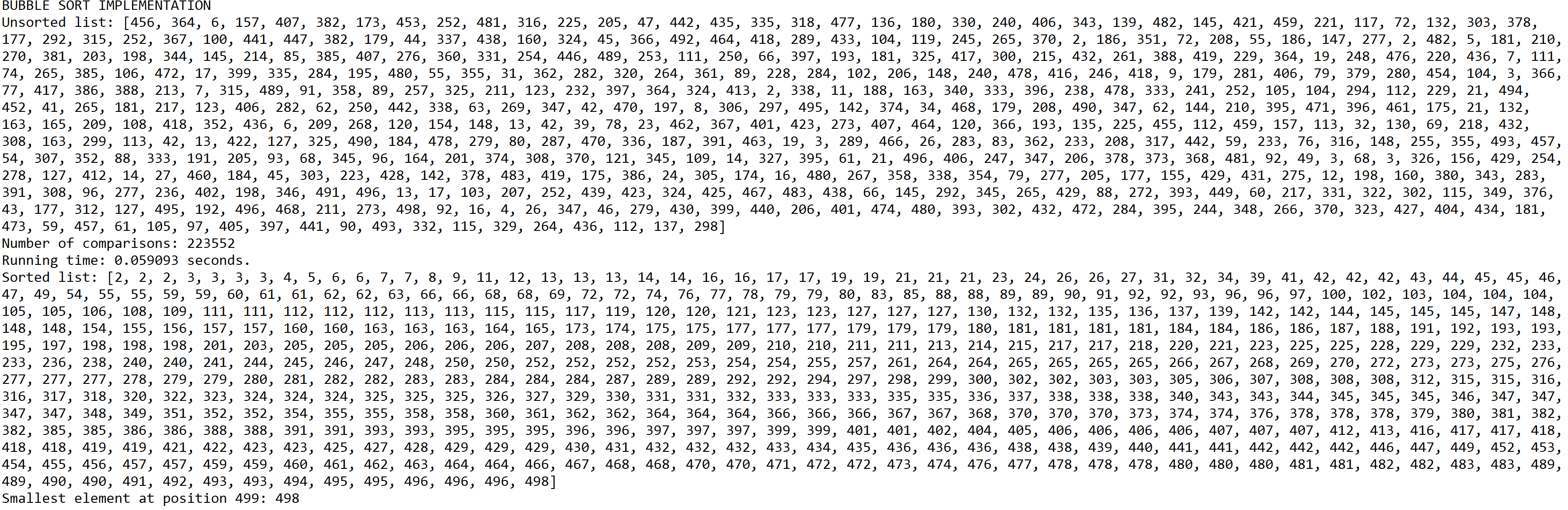
Position k of element to be found = 49



Case 4:

Size of list = 500

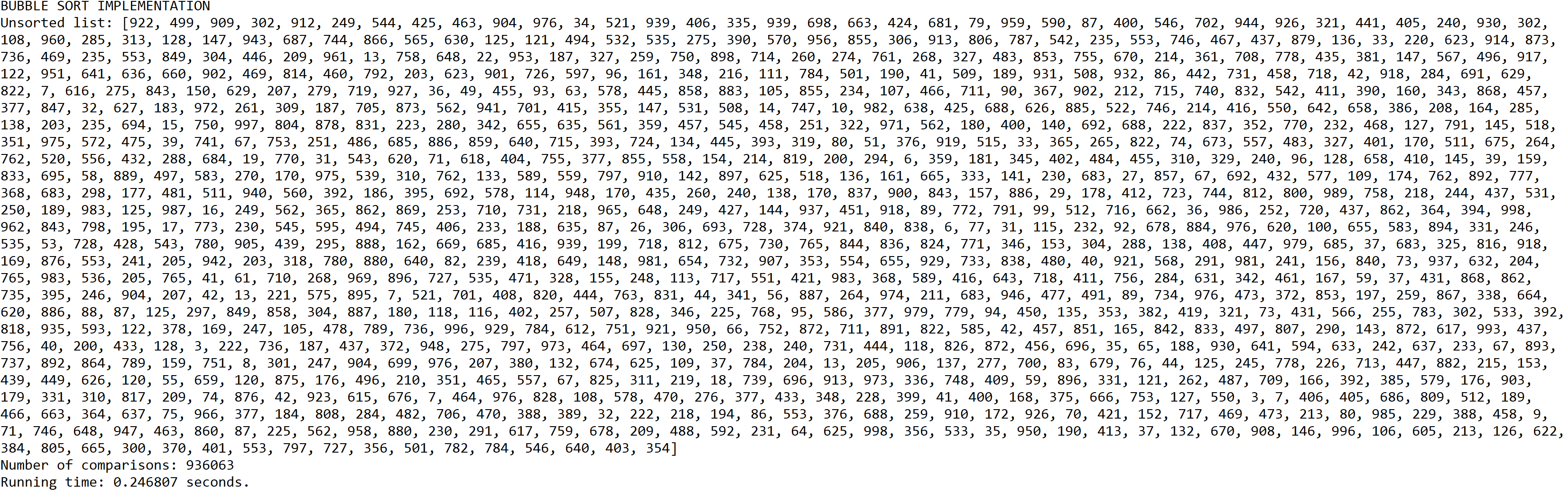
Position k of element to be found = 499

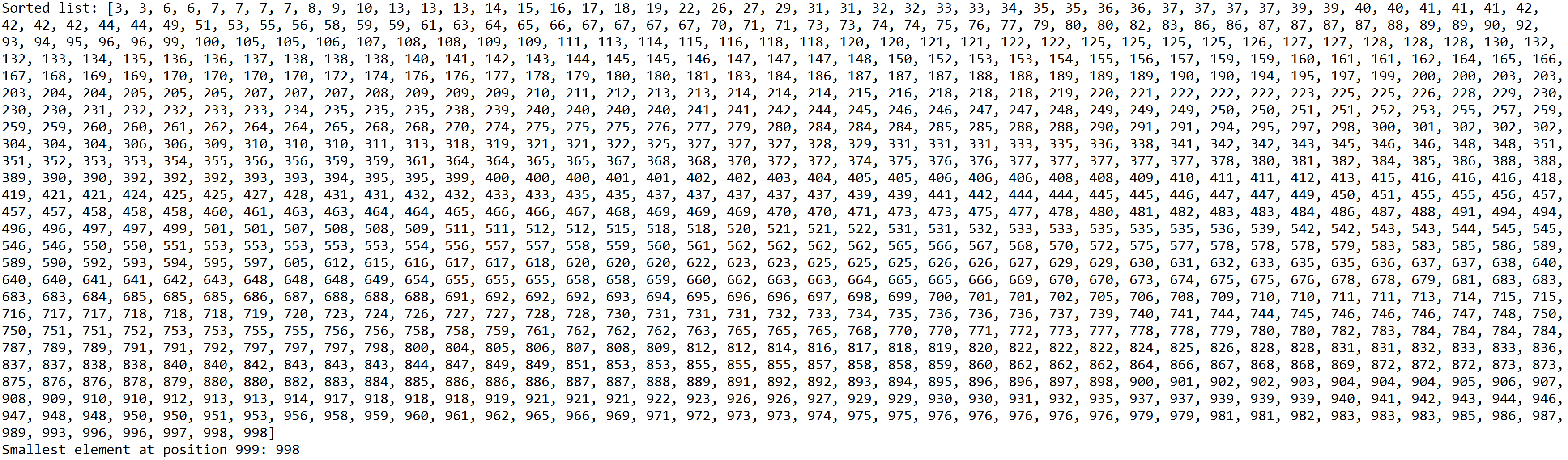


Case 5:

Size of list = 1000

Position k of element to be found = 999





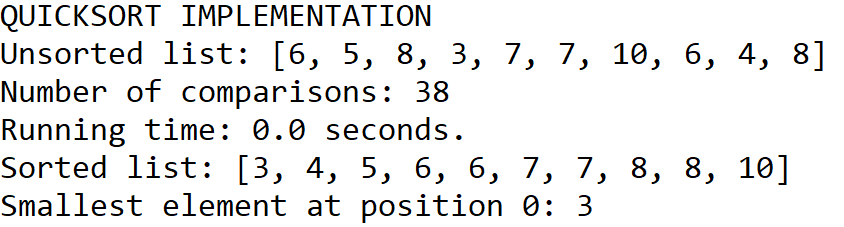
**Operation #2:**

For this operation, I will test randomly generated lists of integers of sizes 10, 50, 100, 500, and 1000. Additionally, I will test that the operation returns the first element (k == 0) for the lists of sizes 10 and 50, the middle element (k == 49) for the list of size 100, and the last element (k == 499 and k == 999) for the lists of sizes 500 and 1000, respectively.

Case 1:

Size of list = 10

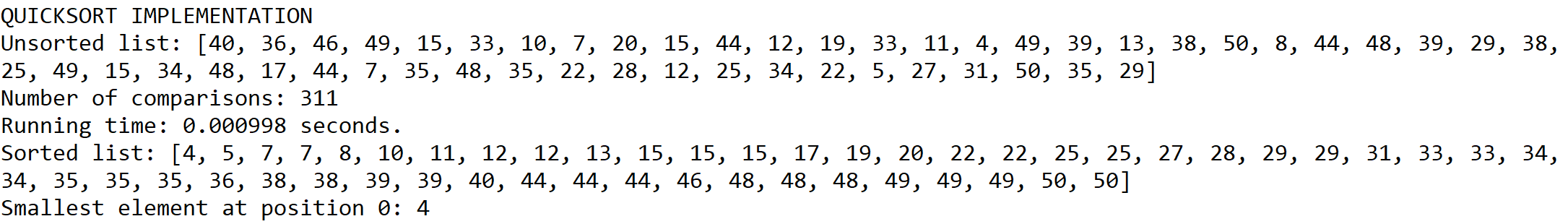
Position k of element to be found = 0



Case 2:

Size of list = 50

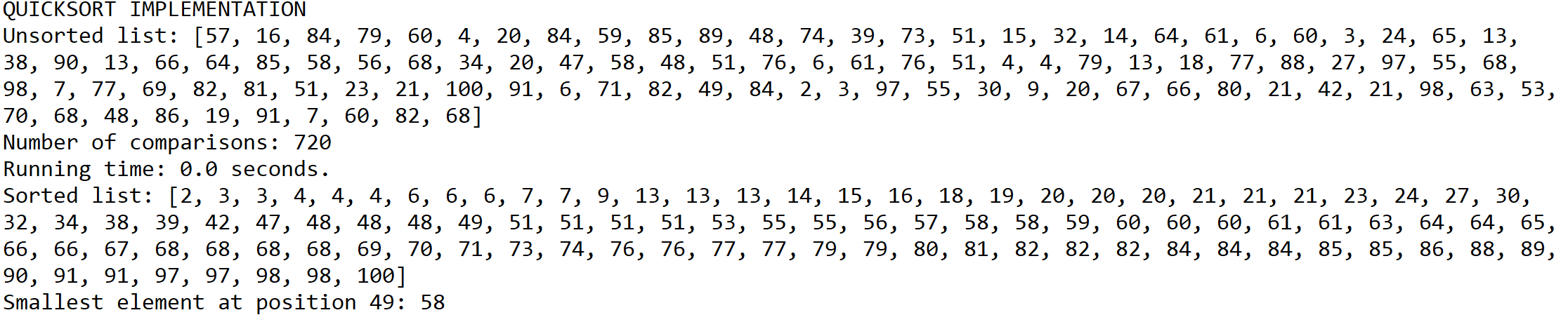
Position k of element to be found = 0



Case 3:

Size of list = 100

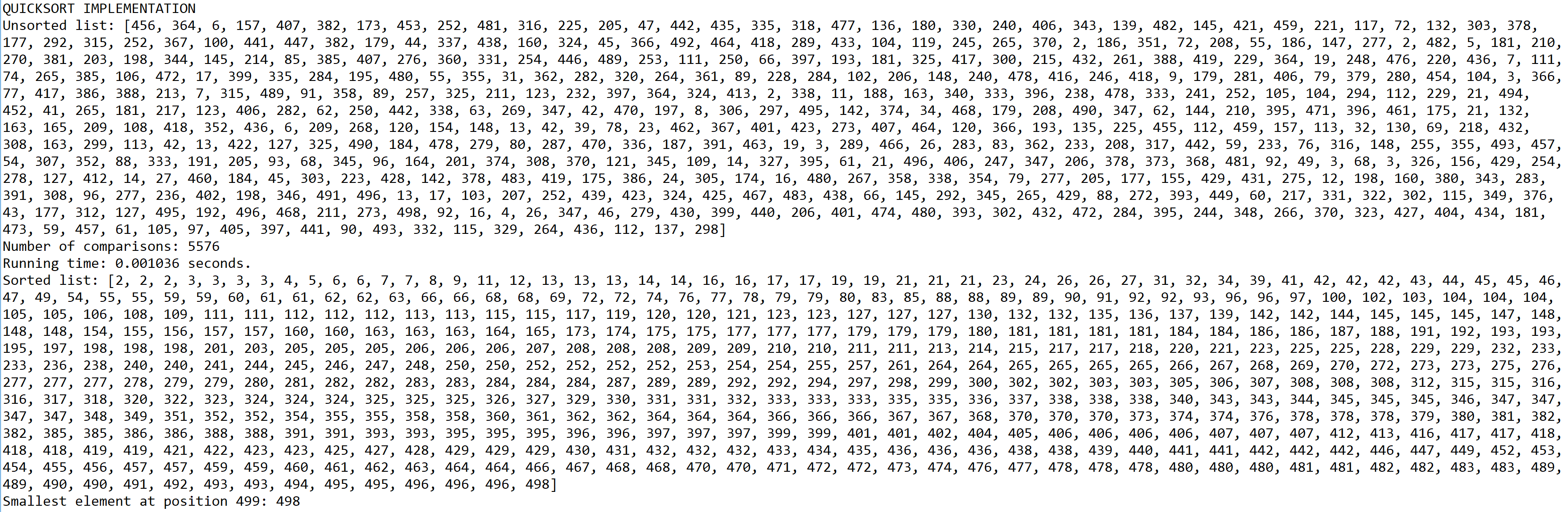
Position k of element to be found = 49



Case 4:

Size of list = 500

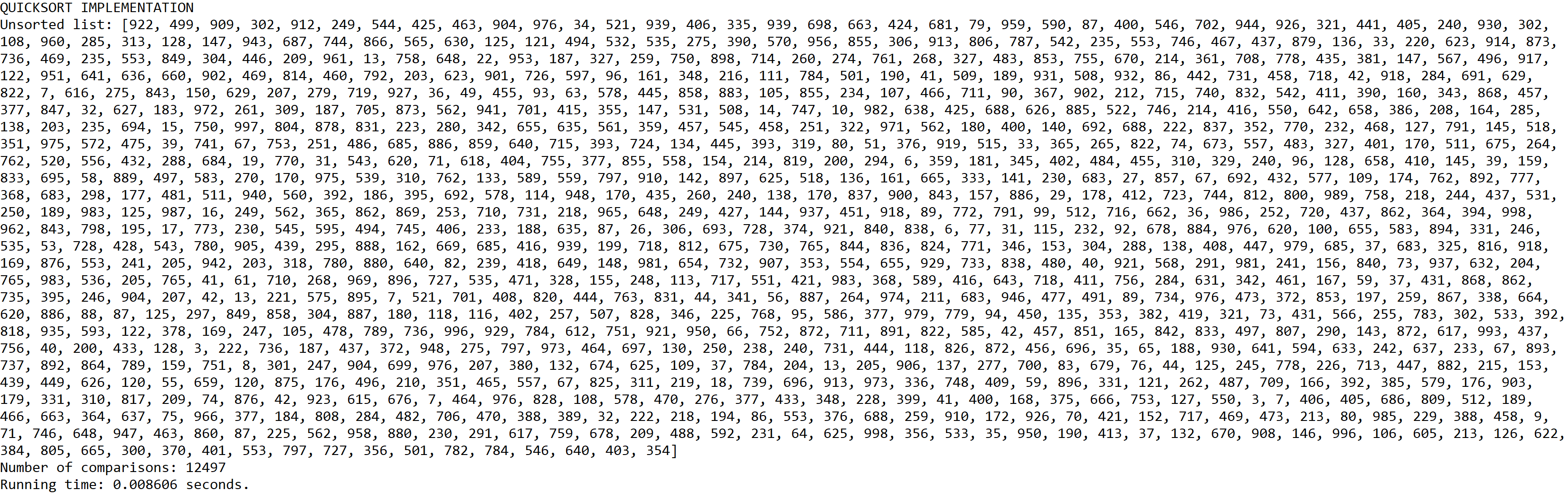
Position k of element to be found = 499

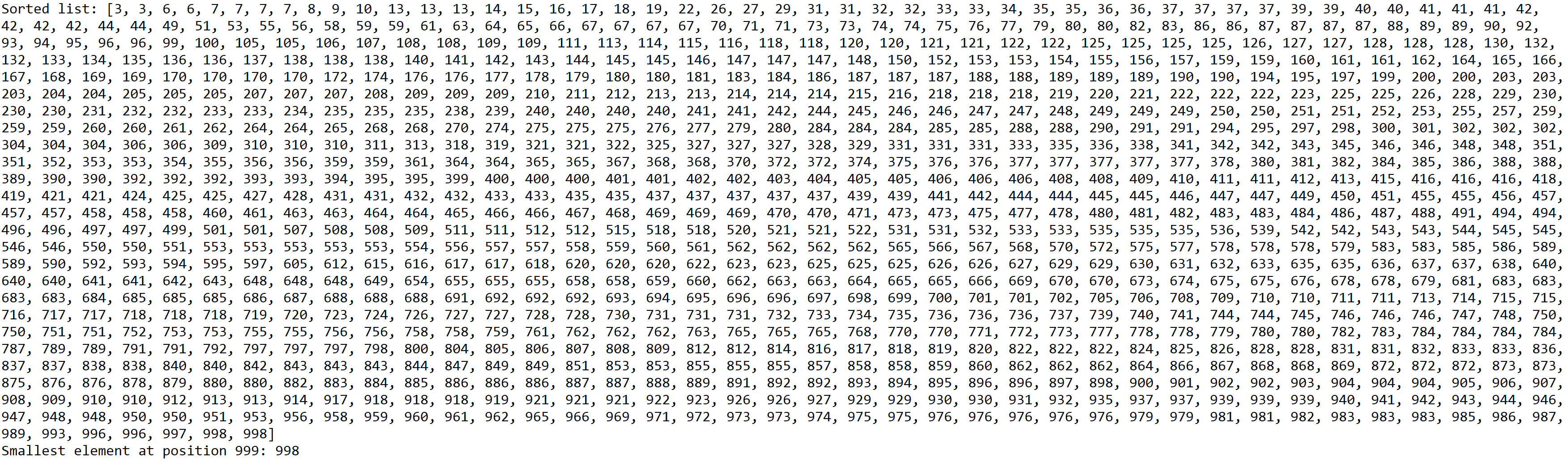


Case 5:

Size of list = 1000

Position k of element to be found = 999





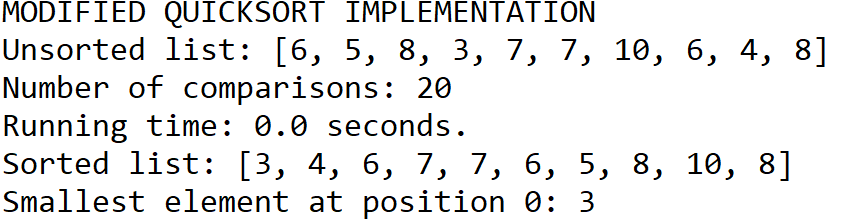
**Operation #3:**

For this operation, I will test randomly generated lists of integers of sizes 10, 50, 100, 500, and 1000. Additionally, I will test that the operation returns the first element (k == 0) for the lists of sizes 10 and 50, the middle element (k == 49) for the list of size 100, and the last element (k == 499 and k == 999) for the lists of sizes 500 and 1000, respectively.

Case 1:

Size of list = 10

Position k of element to be found = 0



Case 2:

Size of list = 50

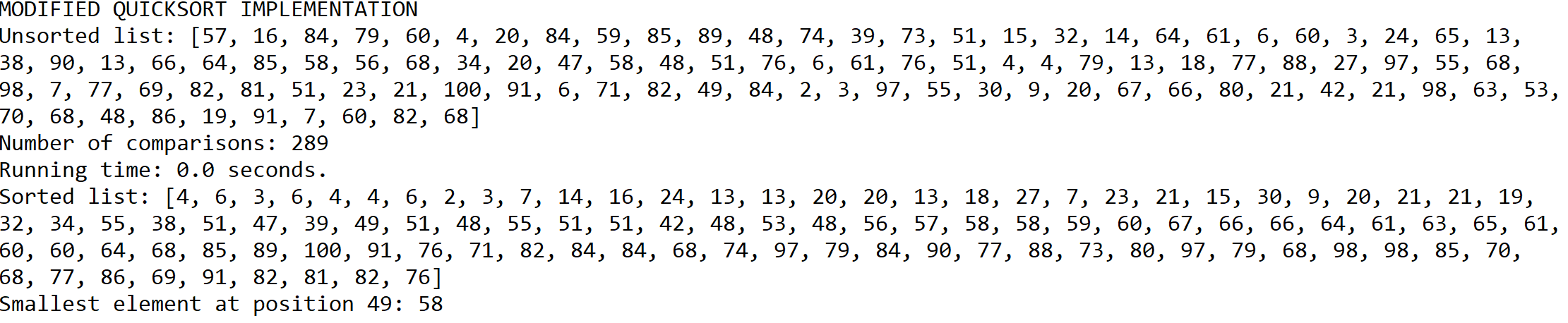
Position k of element to be found = 0



Case 3:

Size of list = 100

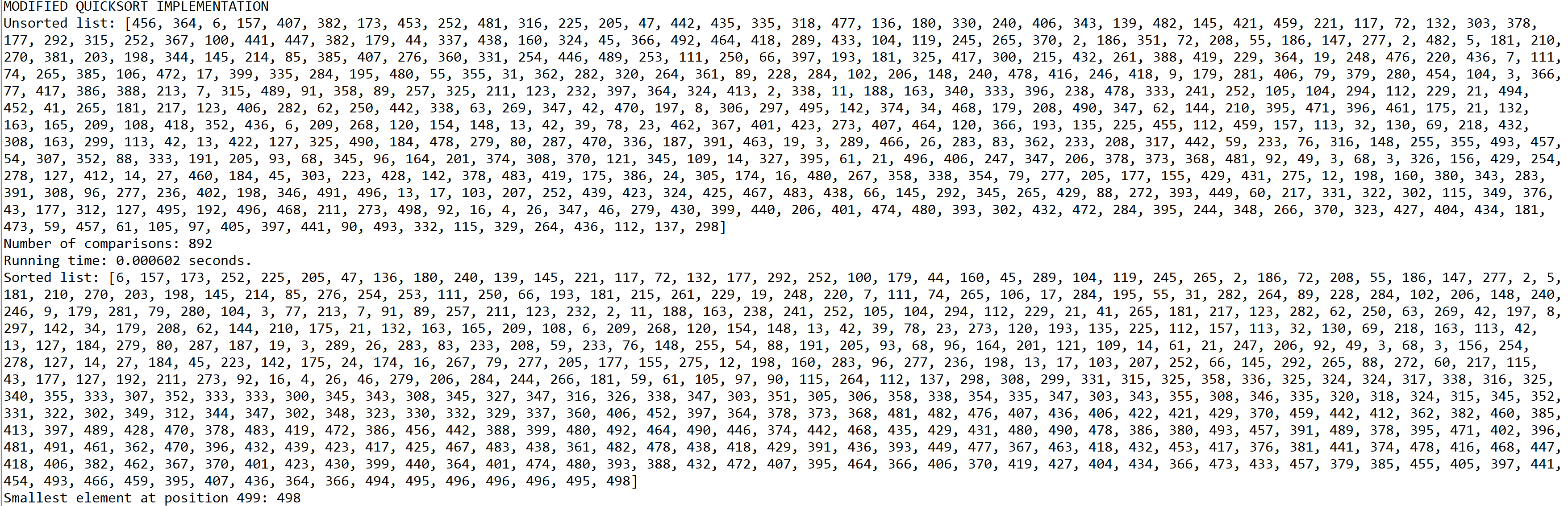
Position k of element to be found = 49



Case 4:

Size of list = 500

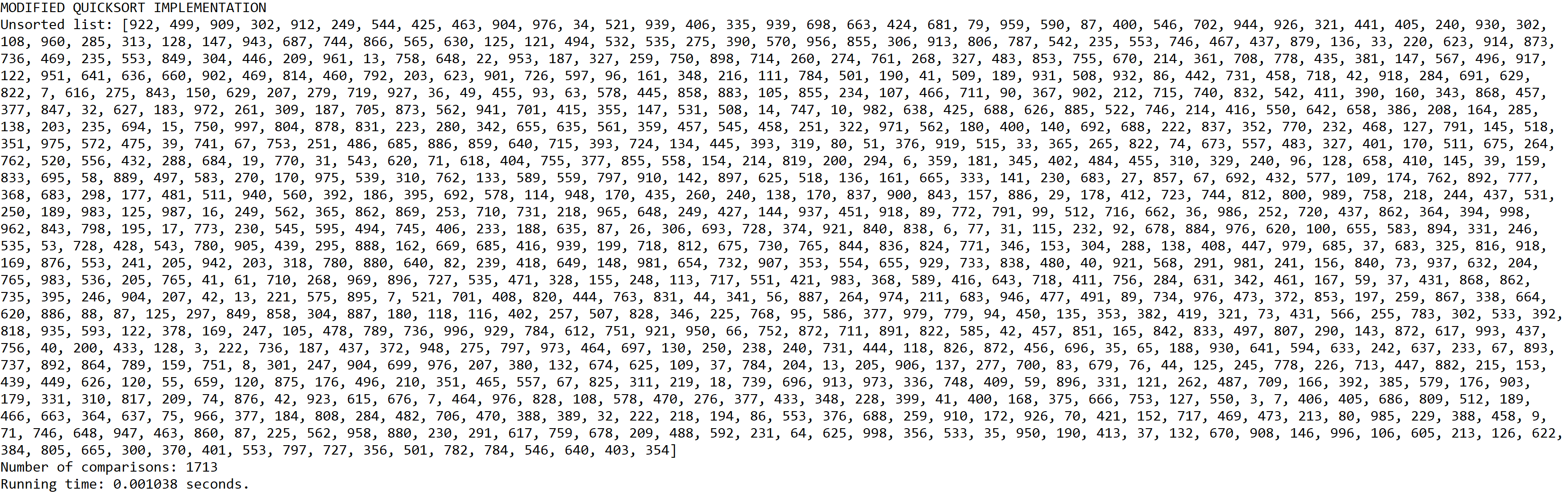
Position k of element to be found = 499

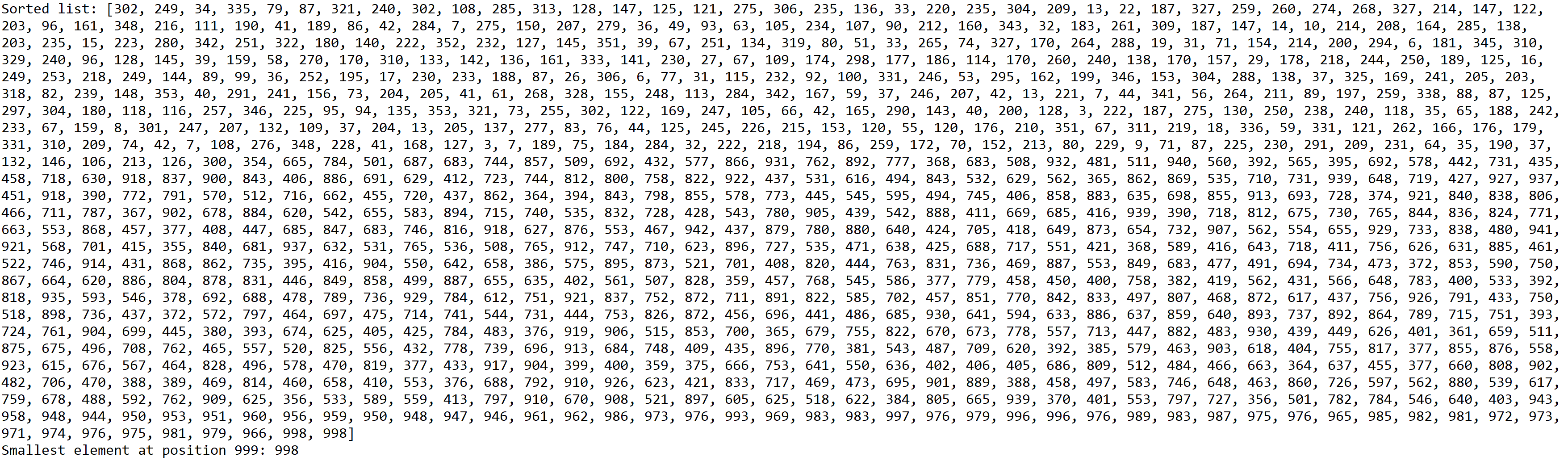


Case 5:

Size of list = 1000

Position k of element to be found = 999





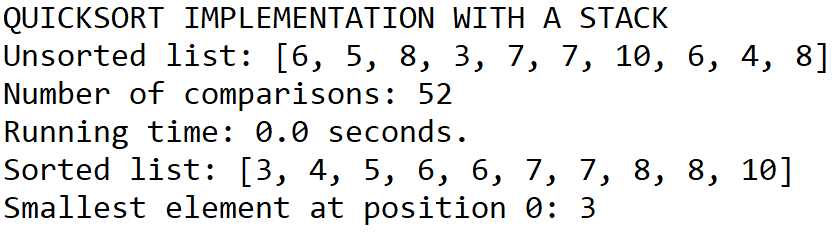
**Operation #4:**

For this operation, I will test randomly generated lists of integers of sizes 10, 50, 100, 500, and 1000. Additionally, I will test that the operation returns the first element (k == 0) for the lists of sizes 10 and 50, the middle element (k == 49) for the list of size 100, and the last element (k == 499 and k == 999) for the lists of sizes 500 and 1000, respectively.

Case 1:

Size of list = 10

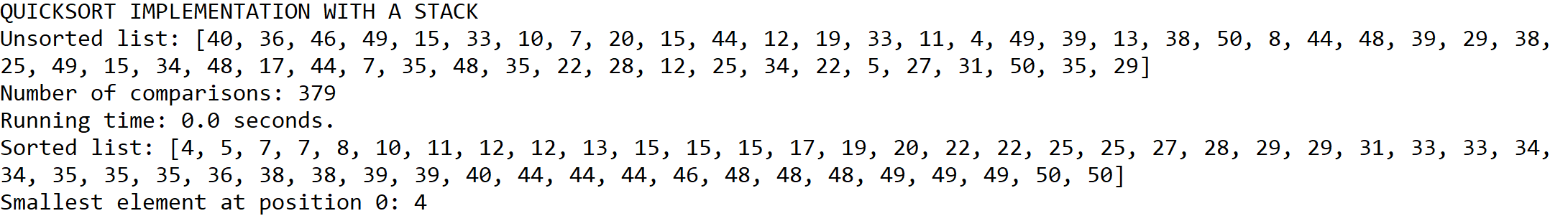
Position k of element to be found = 0



Case 2:

Size of list = 50

Position k of element to be found = 0



Case 3:

Size of list = 100

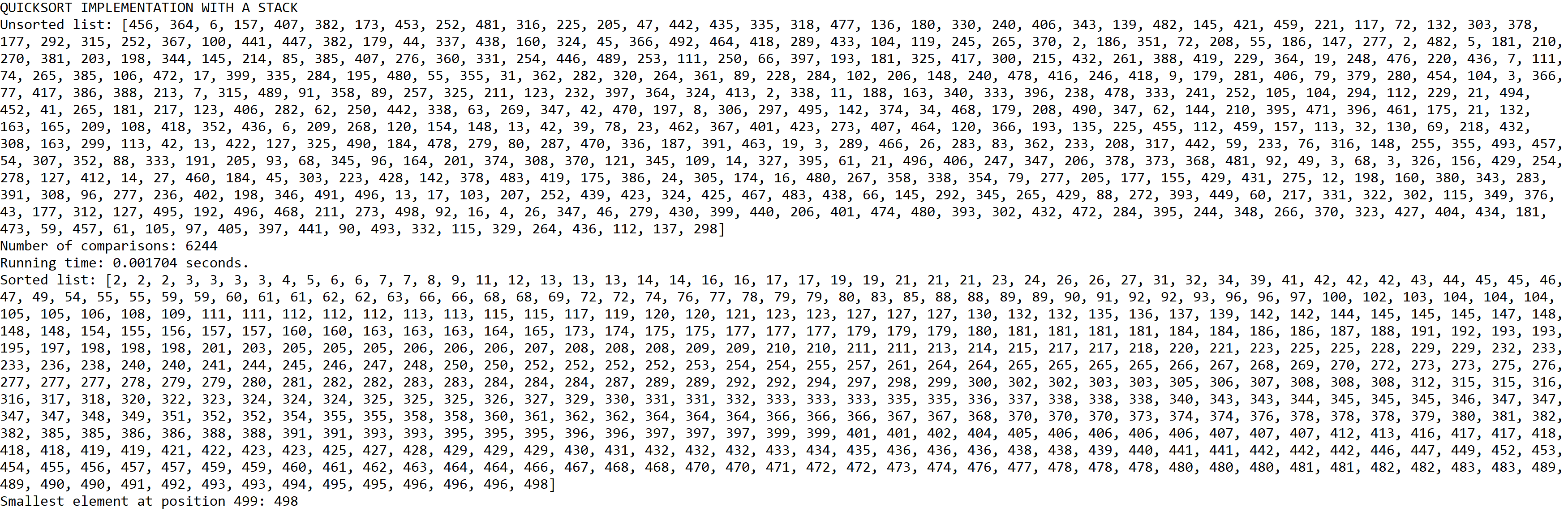
Position k of element to be found = 49



Case 4:

Size of list = 500

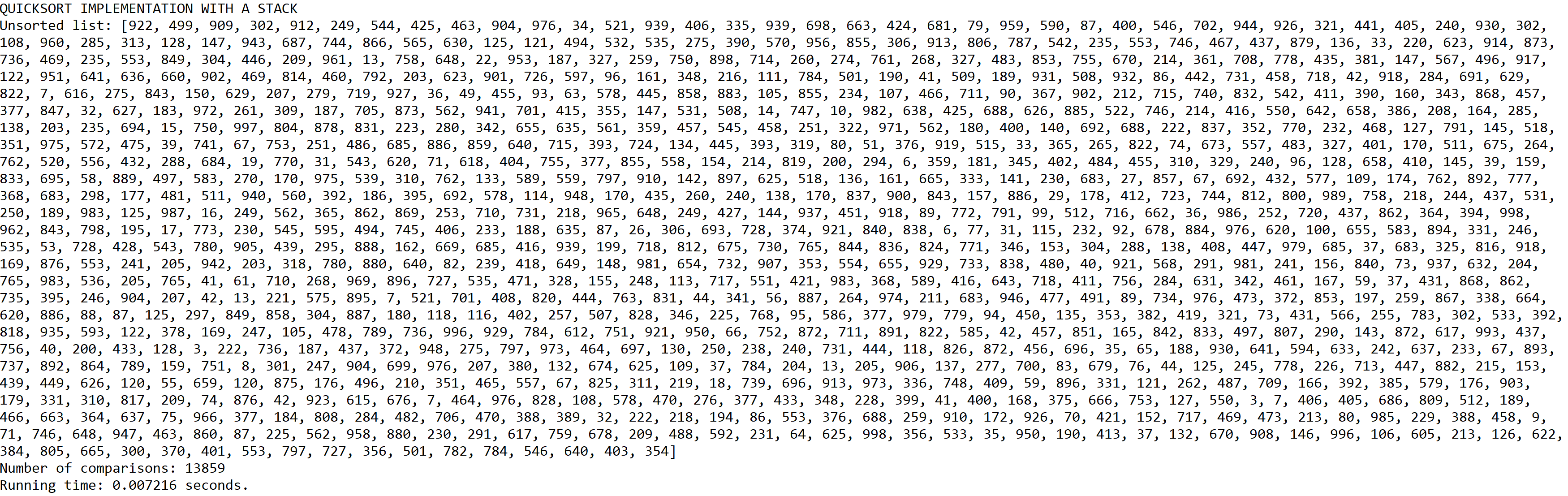
Position k of element to be found = 499

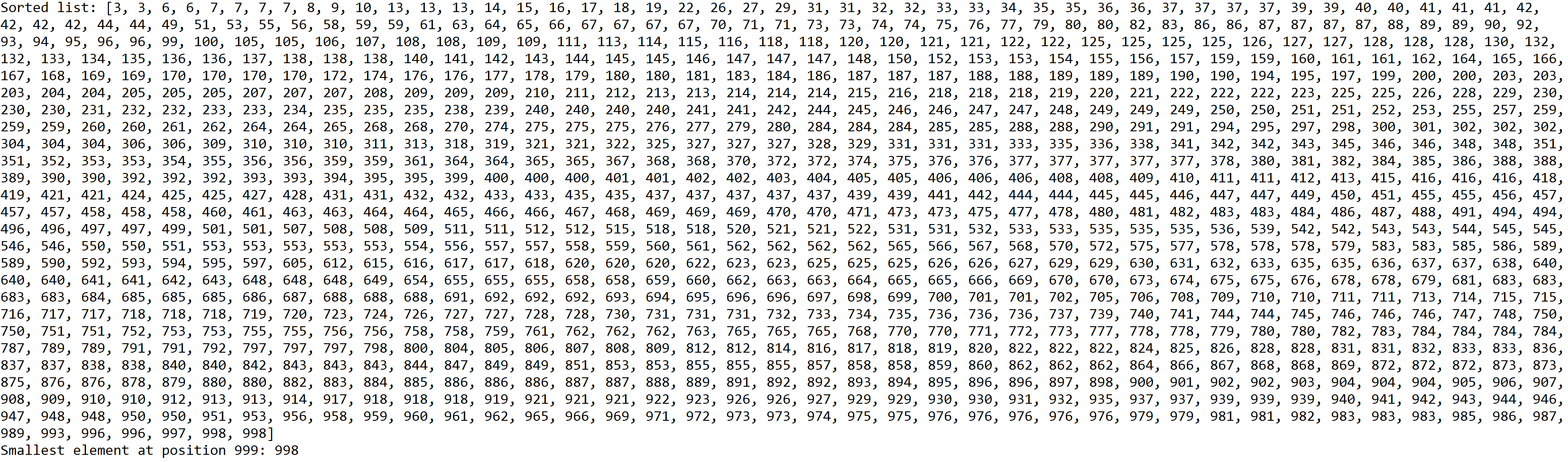


Case 5:

Size of list = 1000

Position k of element to be found = 999





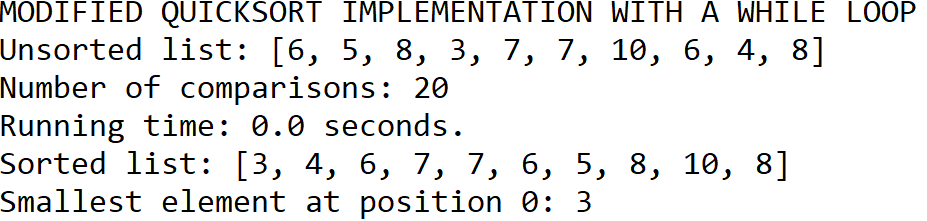
**Operation #5:**

For this operation, I will test randomly generated lists of integers of sizes 10, 50, 100, 500, and 1000. Additionally, I will test that the operation returns the first element (k == 0) for the lists of sizes 10 and 50, the middle element (k == 49) for the list of size 100, and the last element (k == 499 and k == 999) for the lists of sizes 500 and 1000, respectively.

Case 1:

Size of list = 10

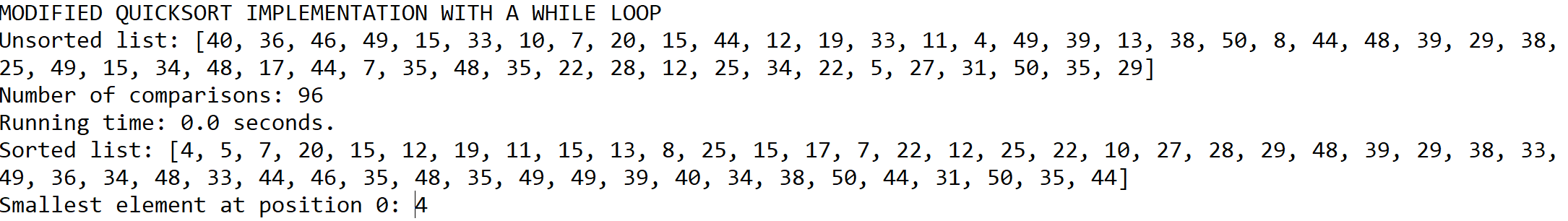
Position k of element to be found = 0



Case 2:

Size of list = 50

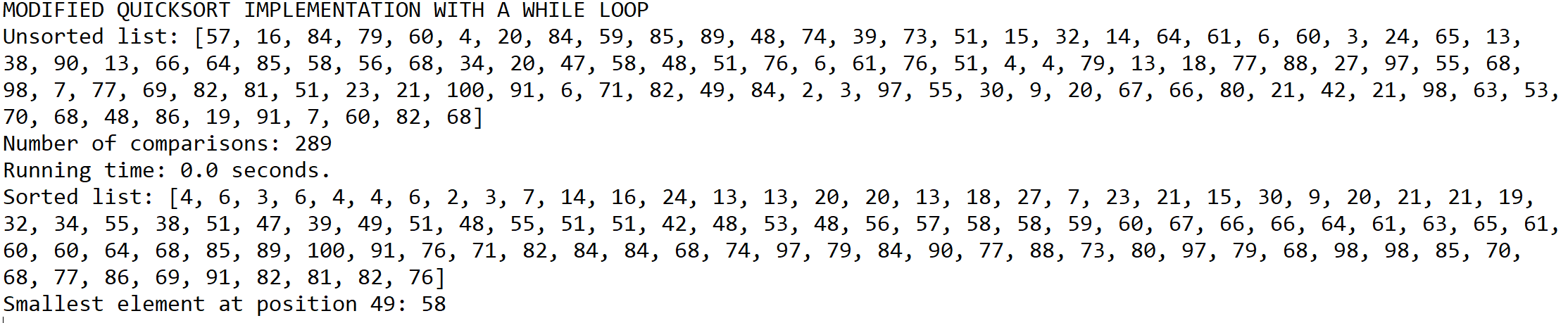
Position k of element to be found = 0



Case 3:

Size of list = 100

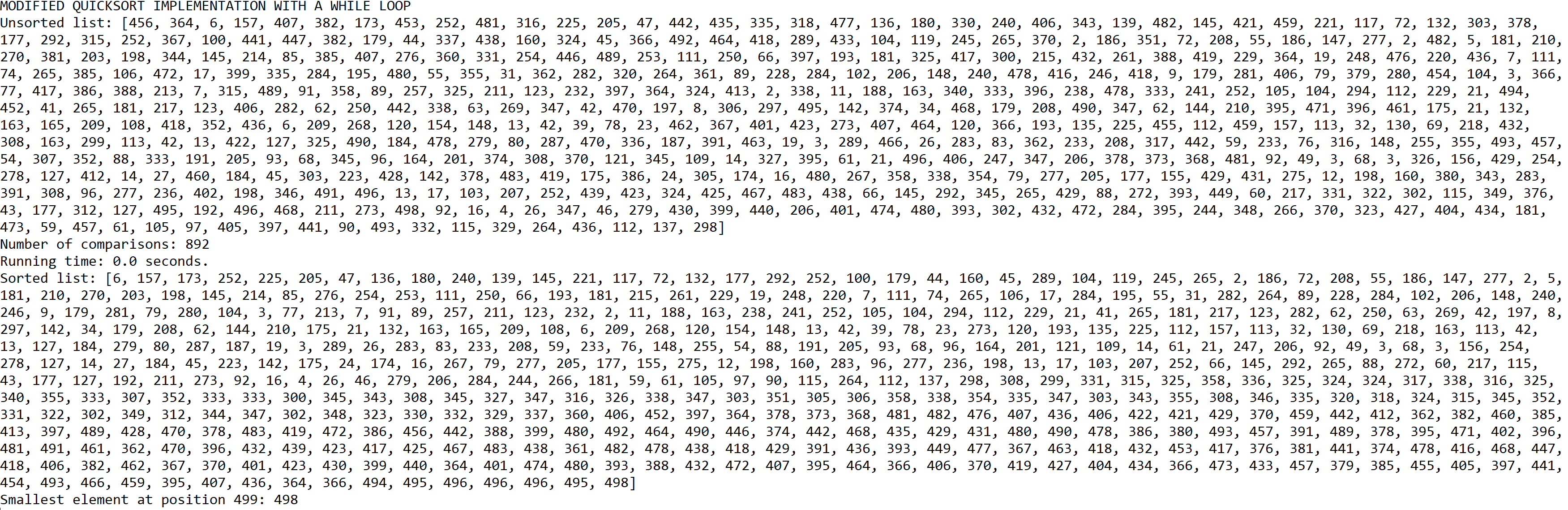
Position k of element to be found = 49



Case 4:

Size of list = 500

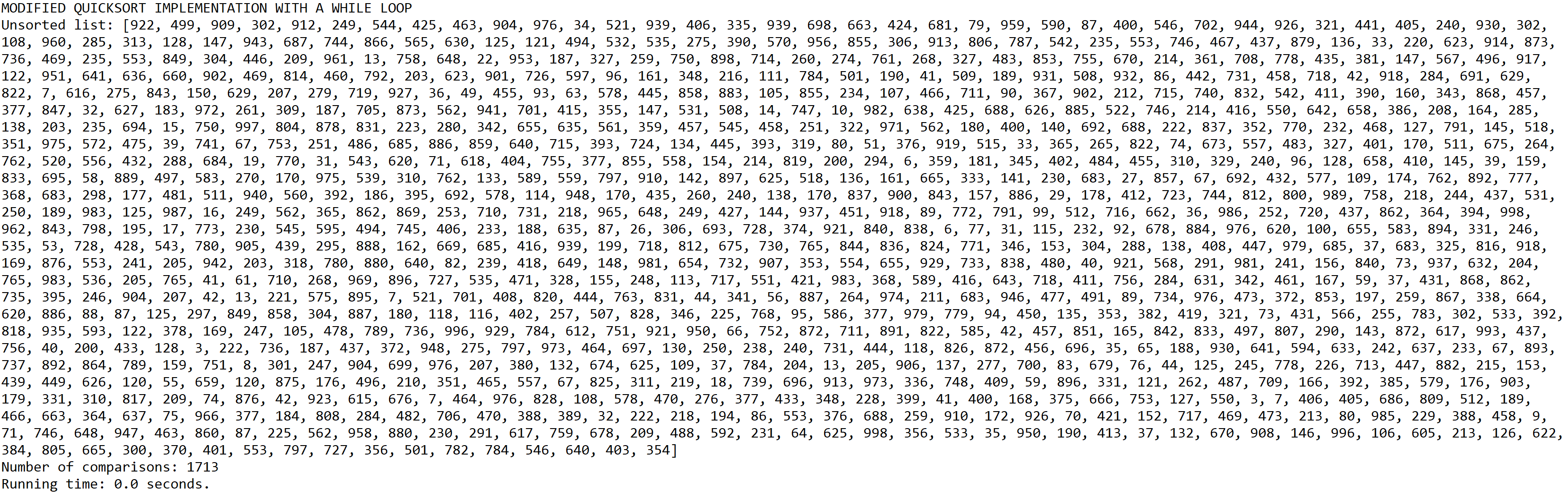
Position k of element to be found = 499

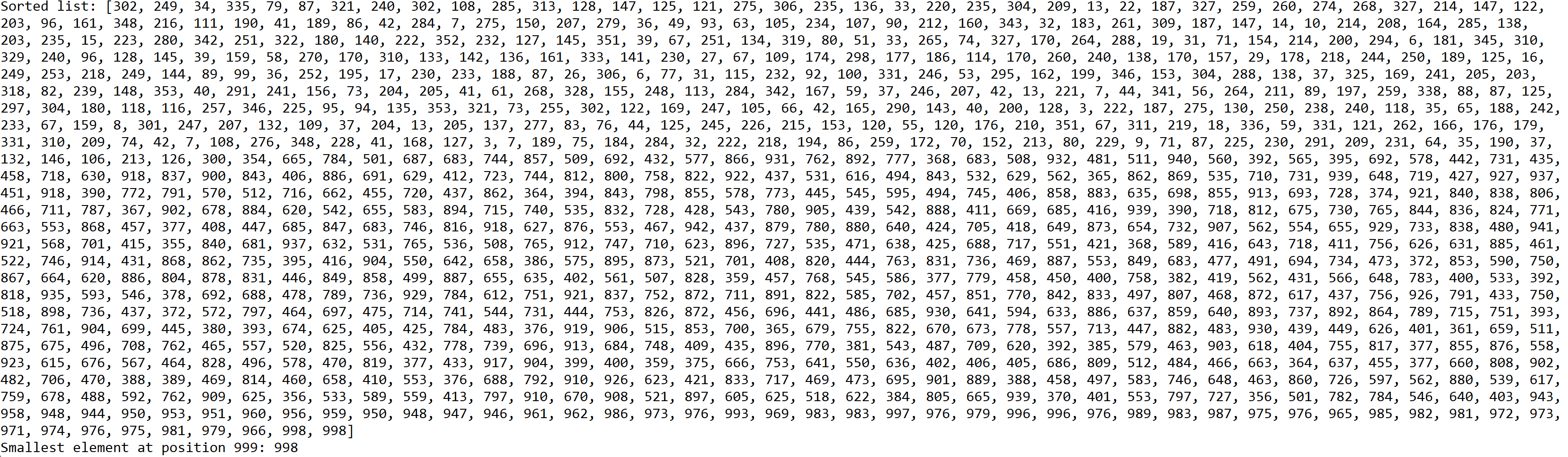


Case 5:

Size of list = 1000

Position k of element to be found = 999





**Graphs of number of comparisons and running times for each operation**

N = Size of the list(10, 50, 100, 500, 1000)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 |
| Operation 1 | 72 | 2156 | 8613 | 223552 | 936063 |
| Operation 2 | 38 | 311 | 720 | 5576 | 12497 |
| Operation 3 | 20 | 96 | 289 | 892 | 1713 |
| Operation 4 | 52 | 379 | 860 | 6244 | 13859 |
| Operation 5 | 20 | 96 | 289 | 892 | 1713 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10 | 50 | 100 | 500 | 1000 |
| Operation 1 | 0.000000 | 0.001311 | 0.003376 | 0.059093 | 0.246807 |
| Operation 2 | 0.000000 | 0.000998 | 0.000000 | 0.001036 | 0.008606 |
| Operation 3 | 0.000000 | 0.000000 | 0.000000 | 0.000602 | 0.001038 |
| Operation 4 | 0.000000 | 0.000000 | 0.001000 | 0.001704 | 0.007216 |
| Operation 5 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |

**Tables of best, worst, and average running times for each operation**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Analysis of running times in seconds for Operation #1 | | | | | |
|  | 10 | 50 | 100 | 500 | 1000 |
| Trial 1 | 0.000000 | 0.001036 | 0.003958 | 0.055897 | 0.228287 |
| Trial 2 | 0.000000 | 0.000997 | 0.000997 | 0.070808 | 0.284941 |
| Trial 3 | 0.000000 | 0.000995 | 0.001954 | 0.048913 | 0.202202 |
| Trial 4 | 0.000000 | 0.001027 | 0.003989 | 0.056399 | 0.237240 |
| Trial 5 | 0.000000 | 0.002069 | 0.002990 | 0.043844 | 0.211852 |
| Trial 6 | 0.000000 | 0.000997 | 0.002992 | 0.089759 | 0.209519 |
| Trial 7 | 0.000000 | 0.000000 | 0.003989 | 0.059981 | 0.212639 |
| Trial 8 | 0.000000 | 0.000000 | 0.000000 | 0.086731 | 0.214608 |
| Trial 9 | 0.000000 | 0.001996 | 0.002991 | 0.079784 | 0.209241 |
| Trial 10 | 0.000000 | 0.000000 | 0.003011 | 0.054898 | 0.213881 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Best case | 0.000000 | 0.000000 | 0.000000 | 0.043844 | 0.202202 |
| Worst case | 0.000000 | 0.002069 | 0.003989 | 0.089759 | 0.284941 |
| Average case | 0.000000 | 0.000912 | 0.002687 | 0.064701 | 0.222441 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Analysis of running times in seconds for Operation #2 | | | | | |
|  | 10 | 50 | 100 | 500 | 1000 |
| Trial 1 | 0.000000 | 0.000000 | 0.000000 | 0.000998 | 0.003079 |
| Trial 2 | 0.000000 | 0.000000 | 0.000000 | 0.002992 | 0.001040 |
| Trial 3 | 0.000000 | 0.000118 | 0.000000 | 0.000999 | 0.001754 |
| Trial 4 | 0.000000 | 0.000000 | 0.000000 | 0.000997 | 0.001033 |
| Trial 5 | 0.000000 | 0.000000 | 0.000998 | 0.000998 | 0.001040 |
| Trial 6 | 0.000000 | 0.000850 | 0.000000 | 0.001995 | 0.004379 |
| Trial 7 | 0.000000 | 0.000000 | 0.000998 | 0.000997 | 0.003600 |
| Trial 8 | 0.000000 | 0.000997 | 0.000000 | 0.000997 | 0.002044 |
| Trial 9 | 0.000890 | 0.000000 | 0.000998 | 0.000998 | 0.000997 |
| Trial 10 | 0.000000 | 0.000000 | 0.000994 | 0.001040 | 0.002037 |
|  |  |  |  |  |  |
| Best case | 0.000000 | 0.000000 | 0.000000 | 0.000997 | 0.000997 |
| Worst case | 0.000890 | 0.000997 | 0.000998 | 0.002992 | 0.004379 |
| Average case | 0.000089 | 0.000197 | 0.000399 | 0.001301 | 0.002100 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Analysis of running times in seconds for Operation #3 | | | | | |
|  | 10 | 50 | 100 | 500 | 1000 |
| Trial 1 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 2 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 3 | 0.000000 | 0.000000 | 0.000000 | 0.000951 | 0.000000 |
| Trial 4 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000626 |
| Trial 5 | 0.000000 | 0.000000 | 0.000000 | 0.000997 | 0.000000 |
| Trial 6 | 0.000000 | 0.000000 | 0.000998 | 0.000000 | 0.000997 |
| Trial 7 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000999 |
| Trial 8 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 9 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 10 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000623 |
|  |  |  |  |  |  |
| Best case | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Worst case | 0.000000 | 0.000000 | 0.000998 | 0.000997 | 0.000999 |
| Average case | 0.000000 | 0.000000 | 0.000100 | 0.000195 | 0.000325 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Analysis of running times in seconds for Operation #4 | | | | | |
|  | 10 | 50 | 100 | 500 | 1000 |
| Trial 1 | 0.000000 | 0.000000 | 0.000000 | 0.001000 | 0.000997 |
| Trial 2 | 0.000000 | 0.000000 | 0.000000 | 0.002992 | 0.003430 |
| Trial 3 | 0.000000 | 0.000000 | 0.000997 | 0.002002 | 0.000998 |
| Trial 4 | 0.000000 | 0.000000 | 0.000996 | 0.002041 | 0.003367 |
| Trial 5 | 0.000000 | 0.000000 | 0.000997 | 0.000997 | 0.001040 |
| Trial 6 | 0.000000 | 0.000000 | 0.000000 | 0.000997 | 0.002380 |
| Trial 7 | 0.000000 | 0.000000 | 0.000000 | 0.000979 | 0.006128 |
| Trial 8 | 0.000000 | 0.000998 | 0.000000 | 0.001995 | 0.003810 |
| Trial 9 | 0.000000 | 0.000000 | 0.000998 | 0.001995 | 0.002383 |
| Trial 10 | 0.000000 | 0.000000 | 0.000997 | 0.001999 | 0.000000 |
|  |  |  |  |  |  |
| Best case | 0.000000 | 0.000000 | 0.000000 | 0.000979 | 0.000000 |
| Worst case | 0.000000 | 0.000998 | 0.000998 | 0.002992 | 0.006128 |
| Average case | 0.000000 | 0.000100 | 0.000499 | 0.001700 | 0.002453 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Analysis of running times in seconds for Operation #5 | | | | | |
|  | 10 | 50 | 100 | 500 | 1000 |
| Trial 1 | 0.000000 | 0.000000 | 0.000000 | 0.000972 | 0.000000 |
| Trial 2 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.001185 |
| Trial 3 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.001035 |
| Trial 4 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 5 | 0.000000 | 0.000000 | 0.000000 | 0.000998 | 0.000000 |
| Trial 6 | 0.000000 | 0.000000 | 0.000000 | 0.000999 | 0.000000 |
| Trial 7 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.001039 |
| Trial 8 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.001995 |
| Trial 9 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Trial 10 | 0.000997 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
|  |  |  |  |  |  |
| Best case | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| Worst case | 0.000997 | 0.000000 | 0.000000 | 0.000999 | 0.001995 |
| Average case | 0.000100 | 0.000000 | 0.000000 | 0.000297 | 0.000525 |

As the results show, the analytical running times agree with the number of comparisons each algorithm makes. This is evident by the fact that the two graphs, which plot the number of comparisons and the running times for each operation, are similar. In both graphs, Operation #1 (bubble sort implementation) increases at such a significant rate that the rest of the operations appear to be close to one another. Regardless, this observation indicates that the bubble sort implementation is the worst implementation for sorting a list of integers and returning the element at position k because it must traverse through the entire list multiple times until it can be completely sorted.

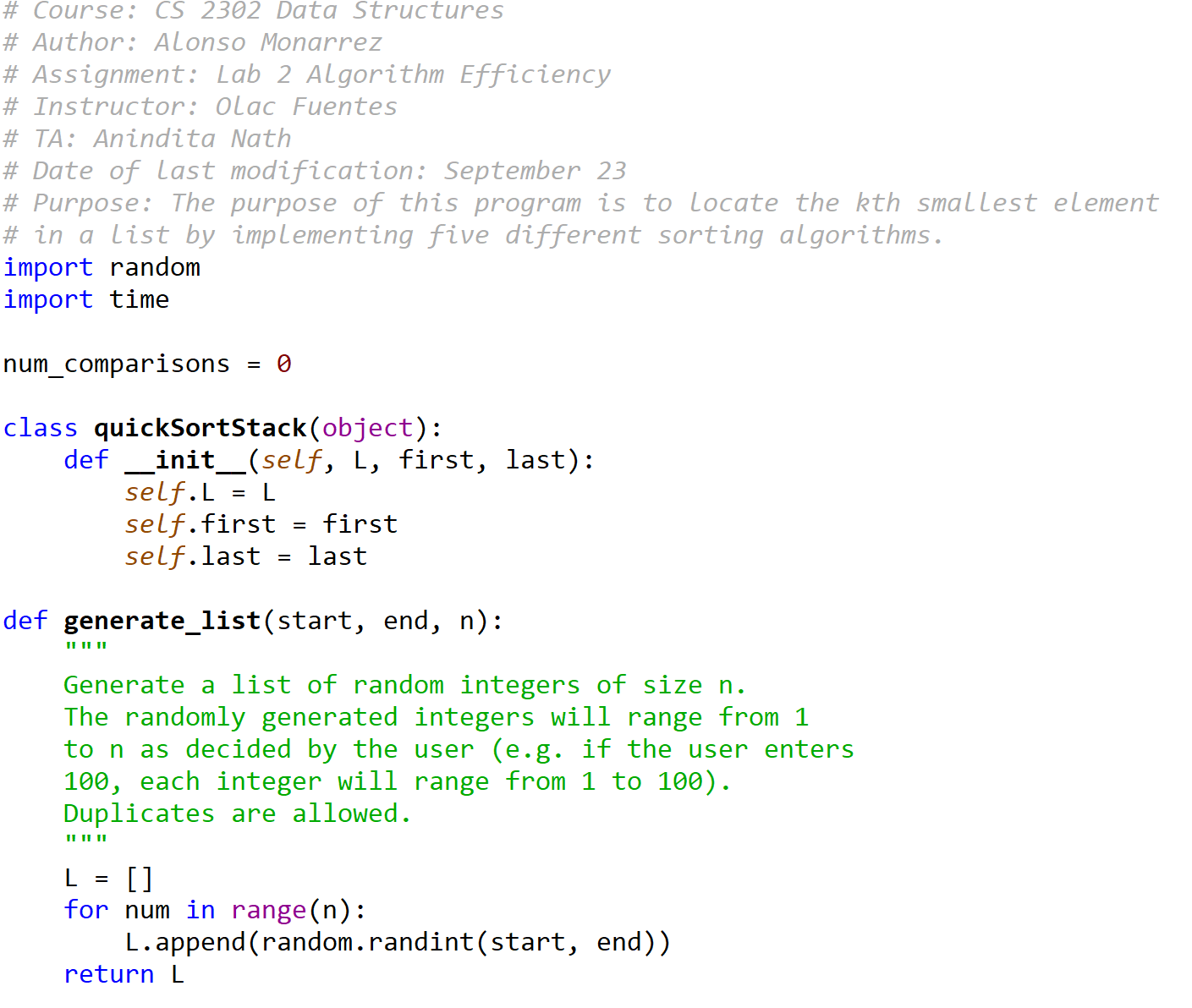
By analyzing the best, worst, and average cases of running times per operation, it can be determined that Operations #3 (modified quicksort implementation) and #5 (modified quicksort implementation with a while loop) are the best implementations for sorting a list of integers and returning the element at position k since their running times across all three cases are the smallest compared to the rest of the operations. Additionally, both operations only need to traverse through approximately one-half of the original list in order to sort the pivot and determine whether it matches the value of k.

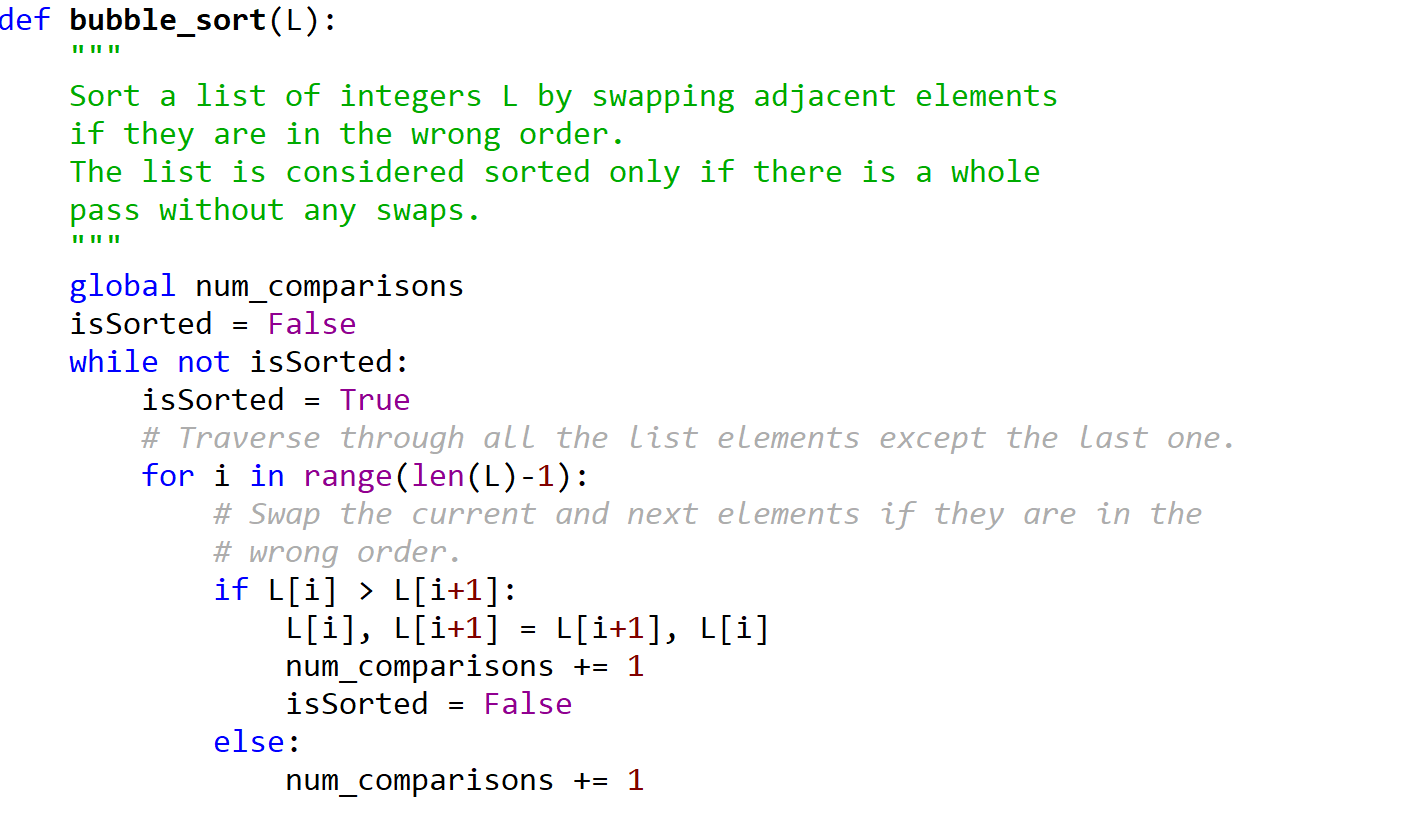
**Conclusion**

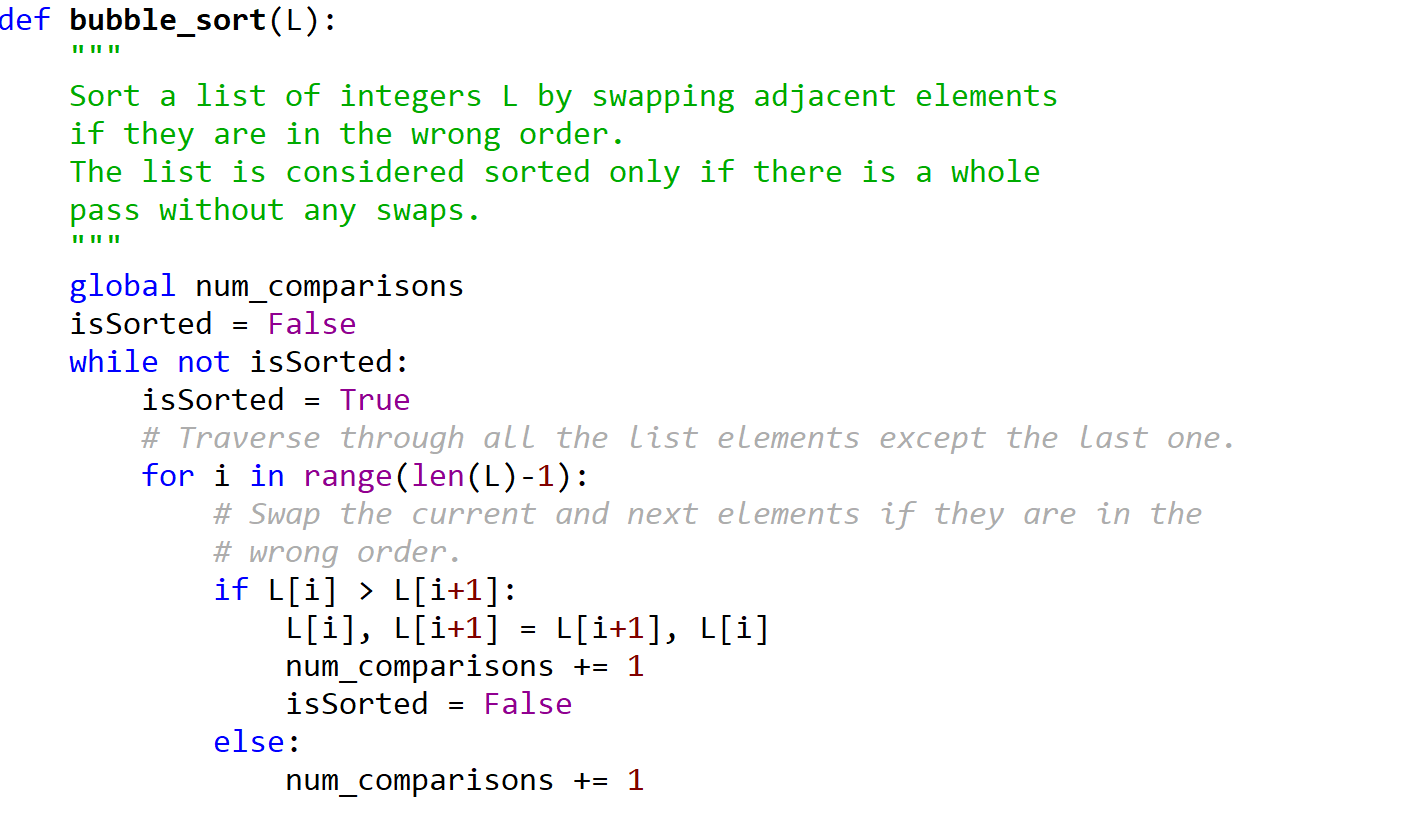
This lab helped me understand sorting algorithms better by allowing me to use activation records as a guide to trace the steps quicksort takes in order to sort an entire list. Once I understood how quicksort works, I was able to modify this sorting algorithm in different ways that would make it more efficient to solve the problem. For example, I understood that quicksort partitions the original list into two sublists by a pivot that is placed in its correct position in the sorted list. Because of this fact, I was able to implement a recursive and an iterative function that only need to sort a few elements as pivots by traversing only one of the two sublists. That way, it is possible to implement a solution to this problem that does not need to traverse through the entire list in order to sort it and find the smallest element at position k.

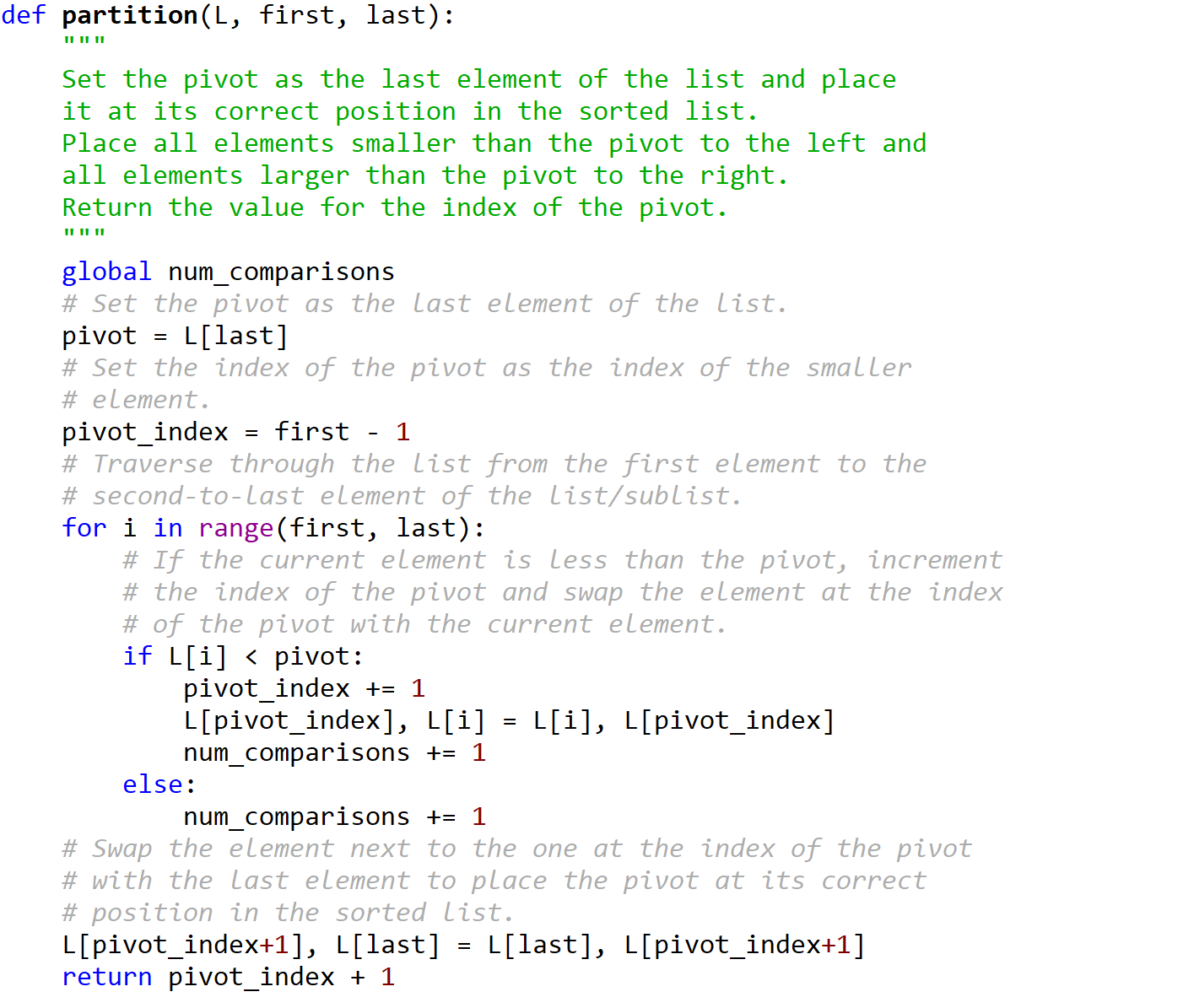
Additionally, this lab helped me understand big-O running time analysis better by allowing me to analyze each operation and determine the steps each one takes to sort the original list. That way, I was able to figure out the big-O running time of sorting algorithms that were derived from other commonly known algorithms such as bubble sort and quicksort.

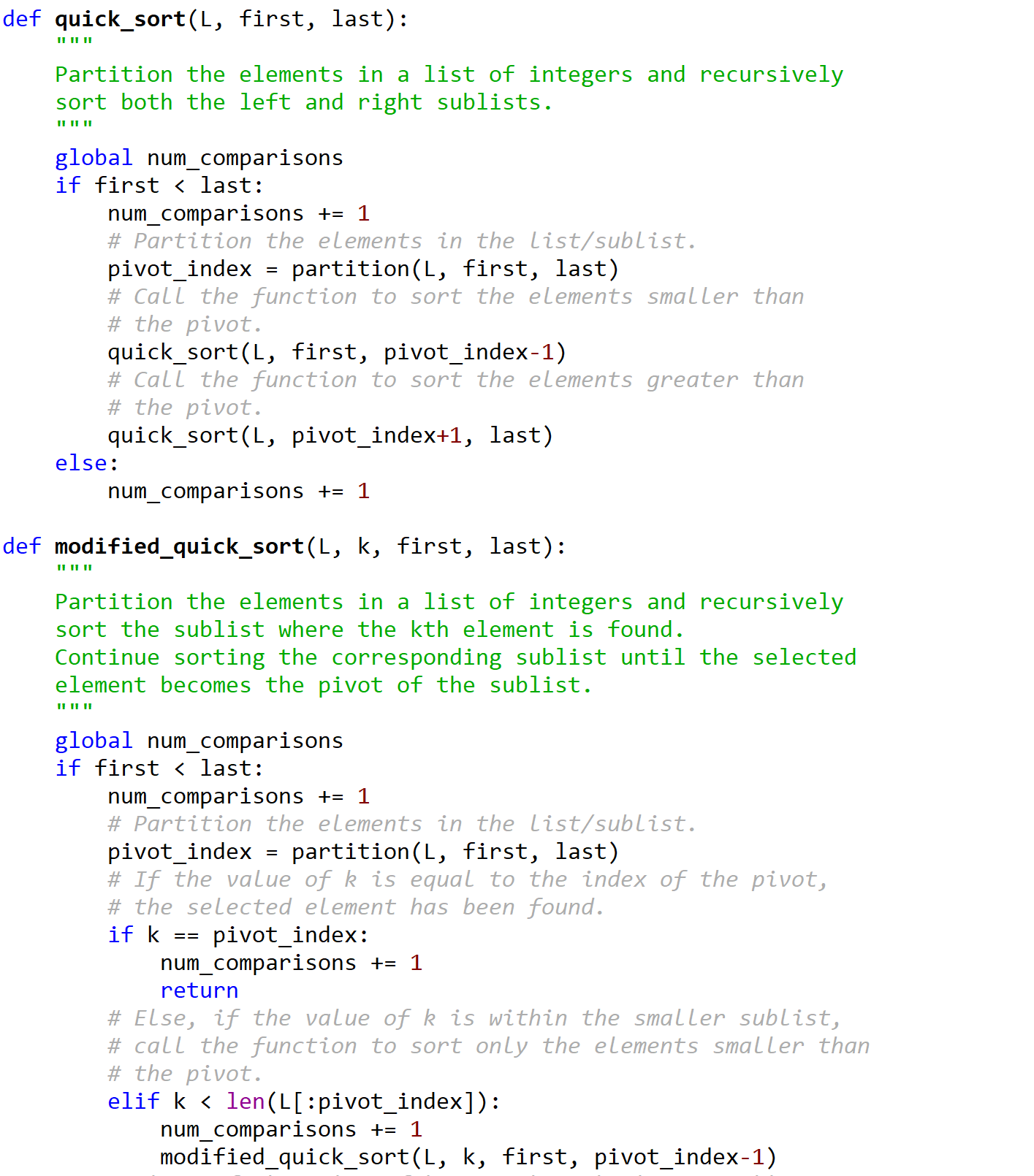
**Appendix**

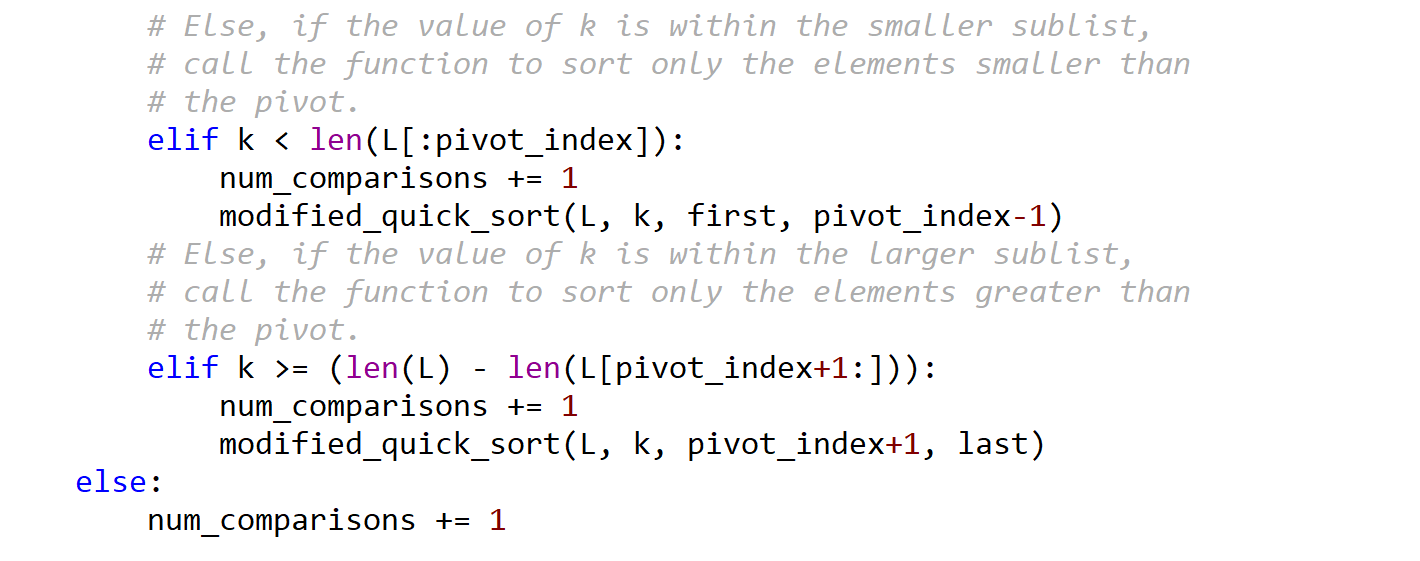


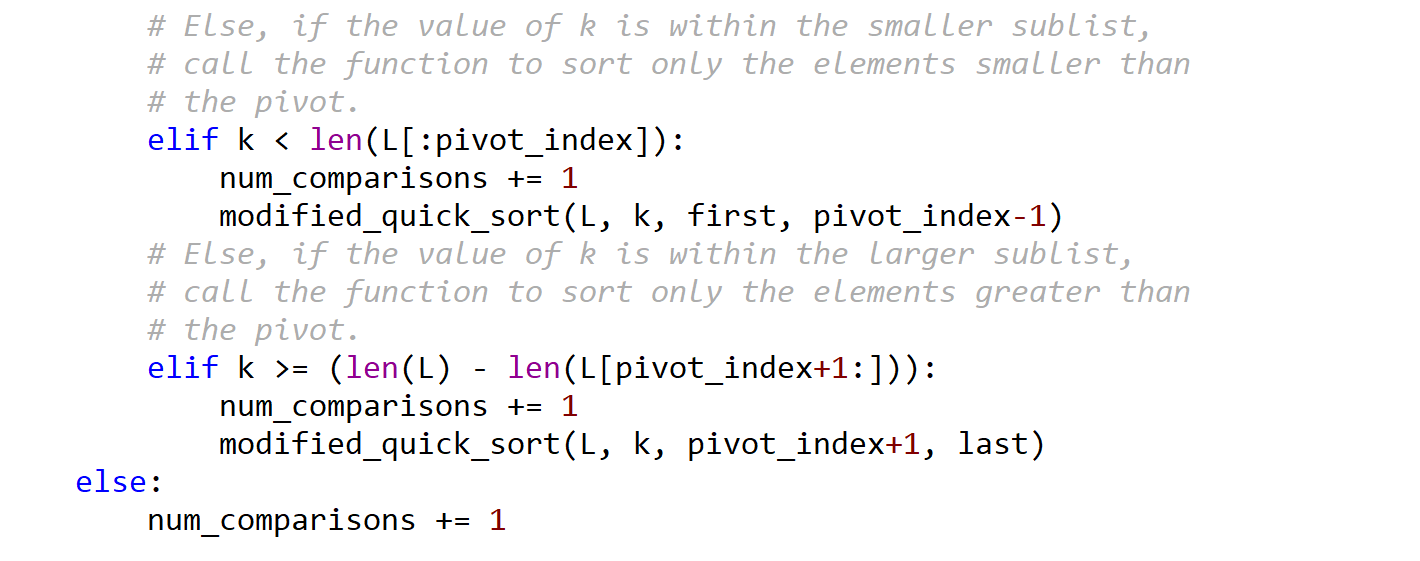


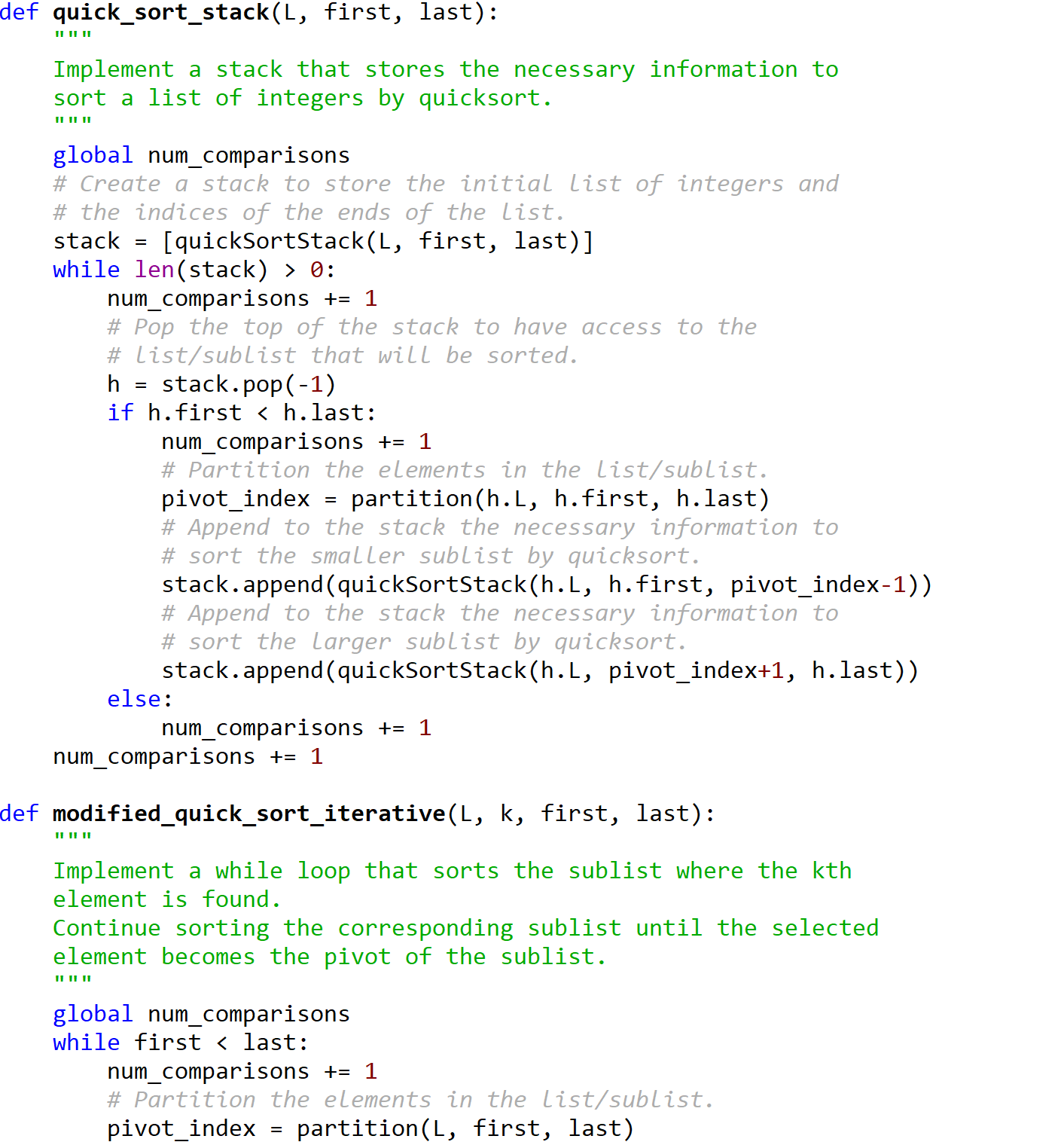


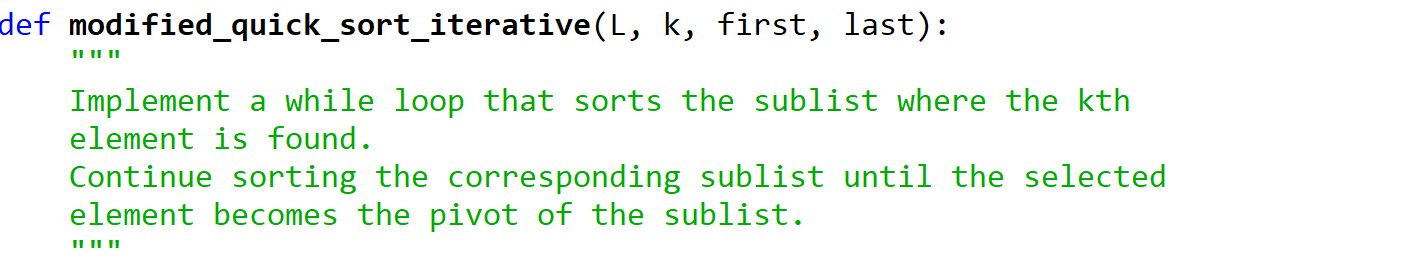




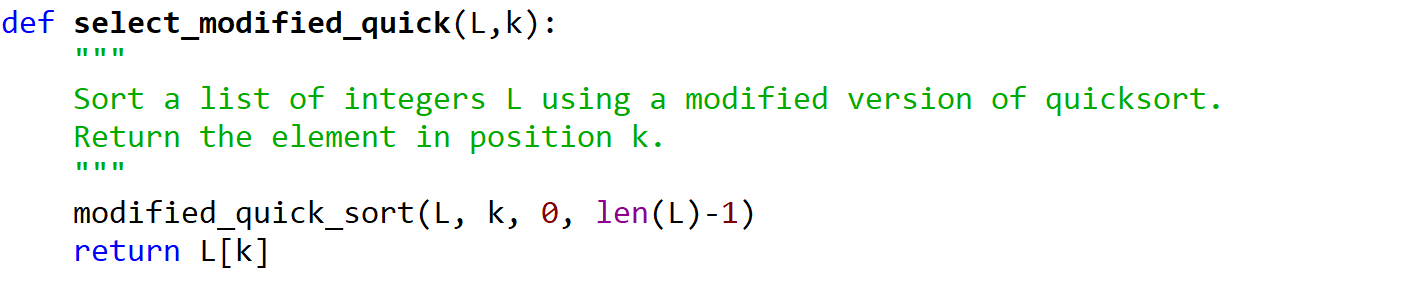


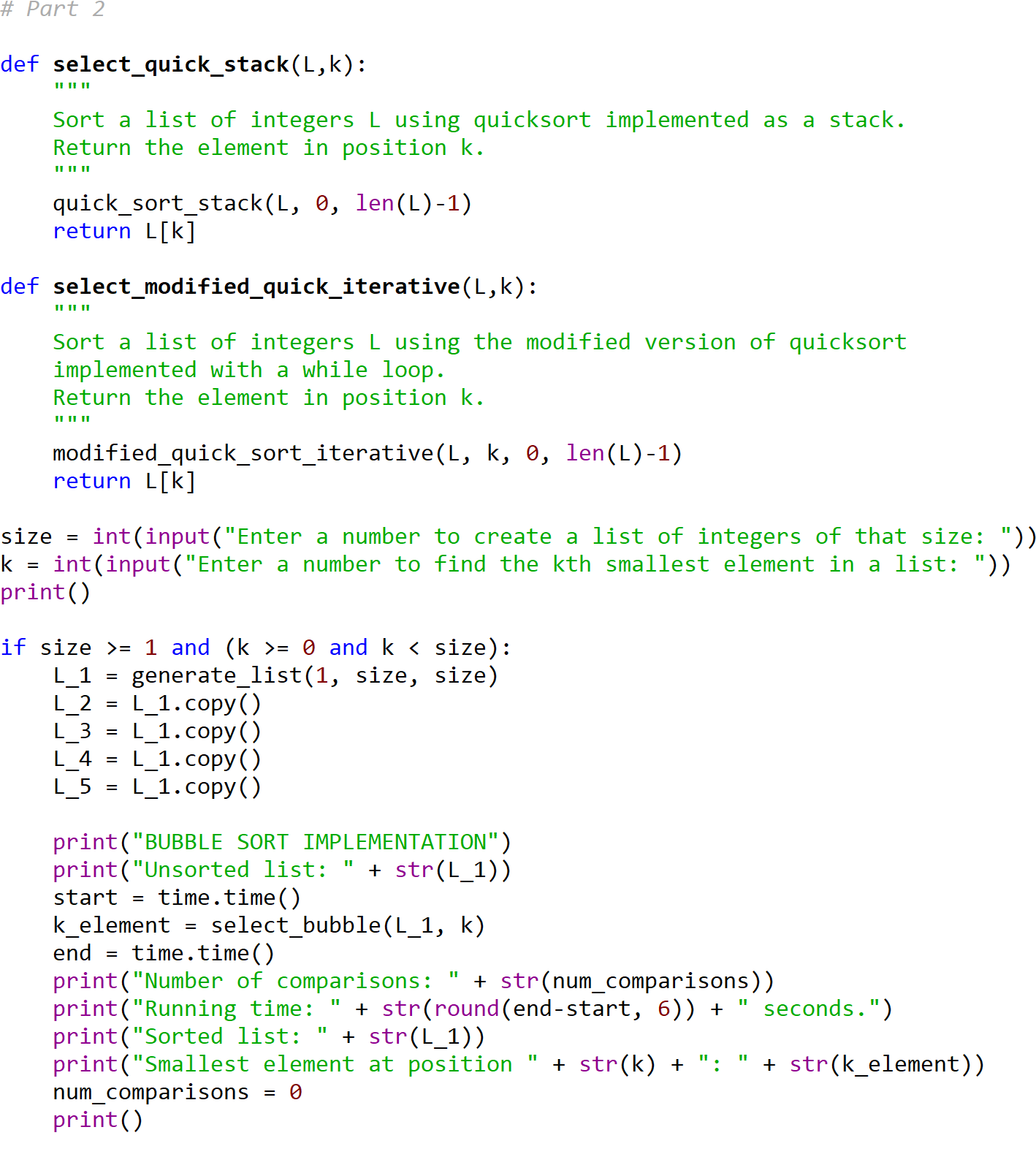


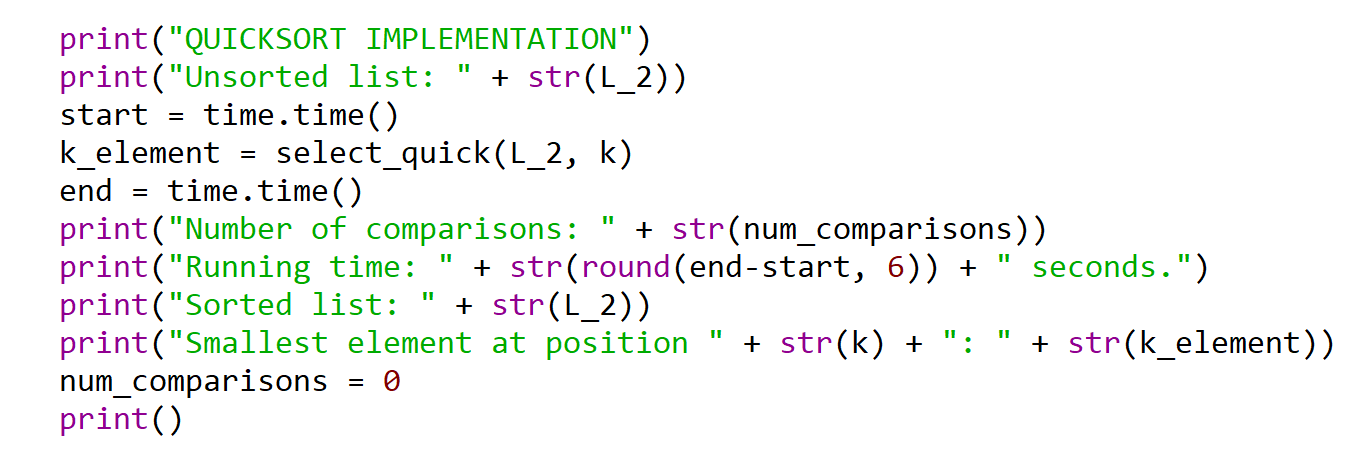


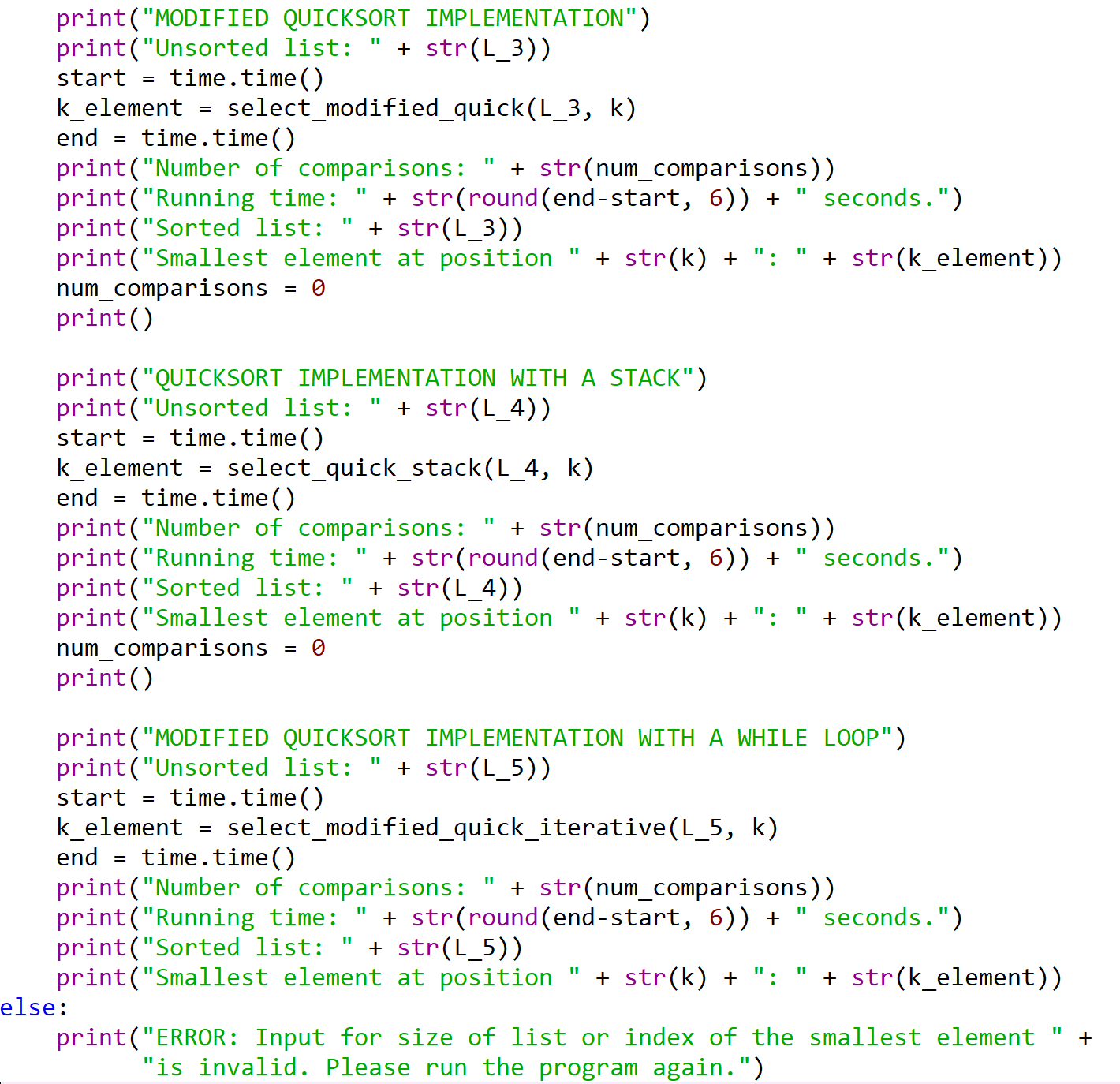












I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.