**Author:**  *Nguyen, Abram*

**Assignment:** *Lab 2 Report*

**Course:** *CS 2302 - Data Structures*

**Instructor:**  *Fuentes, Olac*

**T.A.:**  *Nath, Anindita*

Introduction:

I’m using user-defined objects, nodes and lists, to implement different sorting algorithms. I’ve implemented bubble sort, merge sort, quick sort, and a modified version of quick sort. I’ll compare the run times of these algorithms, the run time being the number of comparisons a method or algorithm executes with respect to ‘n’, the size of a linked list.

Proposed solution design and implementation:

I used each of the following algorithms to sort a list of random numbers and find the median of that list, which would be the middle of the list after sorting. My program would output the same 4 values, but each value was found using different methods.

**Bubble Sort**: Iterate through a list n times, n being the length of the list. Every iteration will contain 1 less element than the iteration before it. For every one of these iterations, compare every adjacent element. If an element's value is greater than the element directly after it, swap these two elements. Larger elements will "float" to the end of the list. Continue until the list is exhausted.

O(n2)

**Merge Sort**: Separate a list, L, into 2 equal(as possible) lists. Repeat this process recursively until the lists can no longer be divided. Merge the lists together again. If an element 'A' value is less than another element 'B' value, append 'B' onto 'A' (A -> B). I used two methods to implement this algorithm. The first method was the method that divided up the list, the second was the method that arranged nodes of the list into order.

M(n) = 2\*M(n / 2) + n a = 2, b = 2, k = 1

O(n log n) by master method

**Quick Sort**: Select a pivot. In this case, the first element in the list. Iterate through the rest of the list, separate all values that are less than and more than the pivot into 2 lists. Continue this process recursively, reform the list by reconnecting the two separate lists with pivot between them. The elements smaller than the pivot would end up being placed to the left of the pivot, and the larger elements would be placed to the right of the pivot.

Q(n) = 2 \* Q(n / 2) + n a = 2, b = 2, k = 1

O(n log n) by master method

**Modified Quick Sort**: A modified version of quicksort that makes a single recursive call instead of the two made by normal quicksort, processing only the sublist where the median is known to reside. Before making my recursive call, I had to decide between 3 choices. The first was if there was only one element to compare, it would be the pivot, which would be the middle value. The other choices were each a recursive call. The recursive call that I would take depended on whether the value ‘mid’ was less than or greater than the length of the list ‘lessL’, where the values smaller than the pivot are. This method will compare only a portion of the numbers in a list instead of the entire list.

R(n) = 1 \* R(n / 2) + n a = 1, b = 2, k = 1

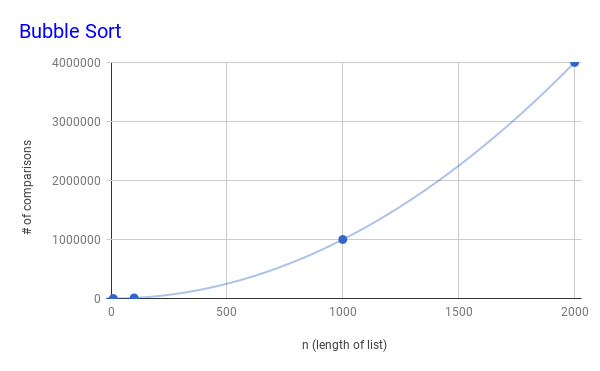
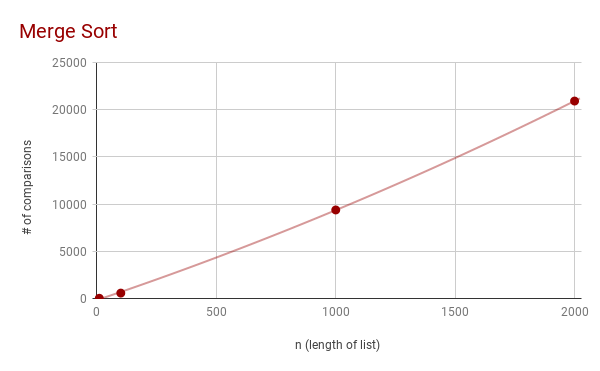
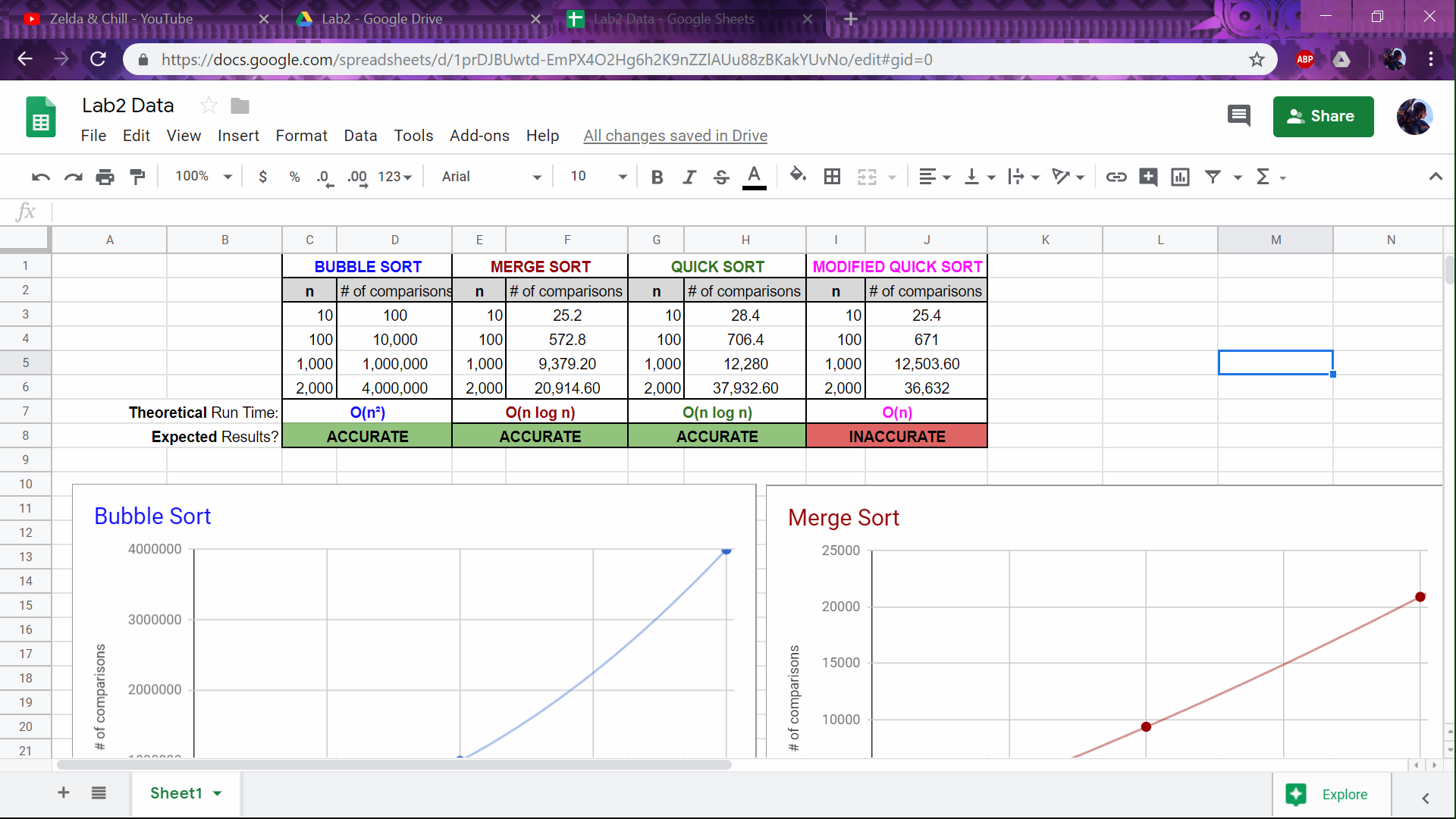
O(n) by master method

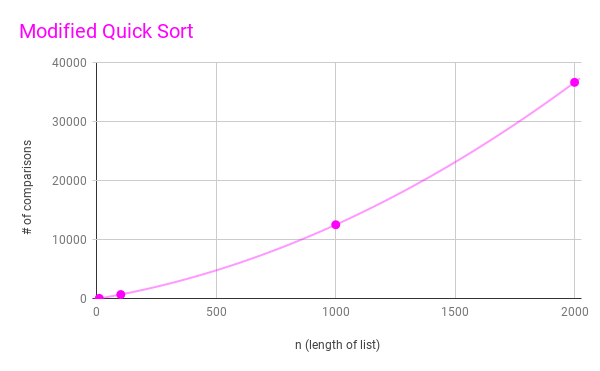
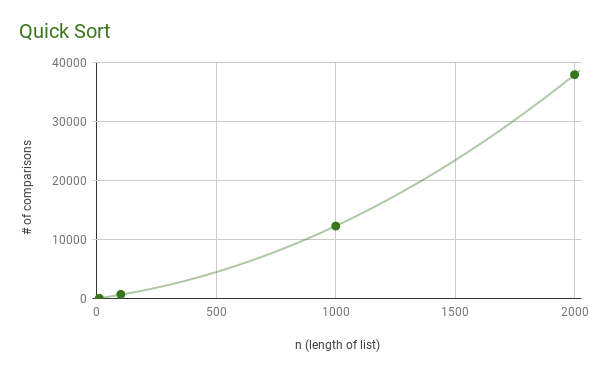
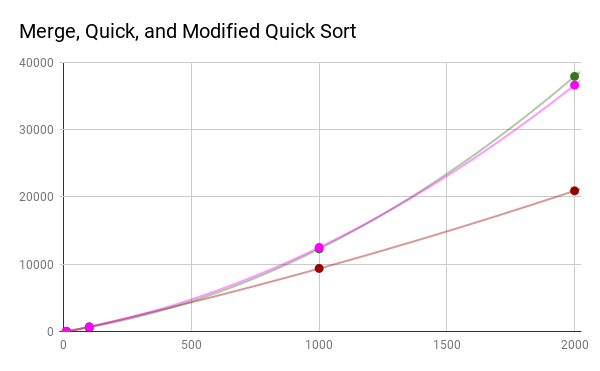
Experimental results:

For my experiments, I did my best to keep track of the number of comparisons each sorting algorithm made. I used different arrays varying in size. The sizes of the arrays I used were 10, 100, 1000, and 2000. I came up with a number of comparisons 5 times for each run and calculated the average.

Because of the fact that merge sort, quick sort, and modified quick sort used recursion in their algorithms, it was difficult to correctly pin down how many comparisons were made in each run. I added a print statement that would print the letter ‘c’ every time a comparison was made, then pasted that block of c’s into a text editor to count how many times ‘c’ was printed. I think my method of printing ‘c’ was inaccurate, however.

My experiment with bubble sort, merge sort and quick sort seemed to be accurate for the most part, while modified quick sort seemed to be mostly inaccurate. Quick sort was slower than merge sort even though it has the same theoretical run time, as expected.





I hadn’t included bubble sort in this last graph because of how quickly the number of comparisons graph grew compared to the other three. Merge, quick and modified quick sort would’ve turned out to be unreadable.

Conclusions:

I learned about the different factors that affect an algorithm’s run time. The number of comparisons executed isn’t always the same when running an algorithm with different values. There are best-case and worst-case scenarios to consider for algorithms. For example, whether or not quick sort is efficient relies on the initial order of a list of values. This lab also gave me more insight and understanding on what big-O notation really means, it’s not always exact.

Appendix:

|  |  |
| --- | --- |
| """ |  |
|  | Author: Nguyen, Abram |
|  | Assignment: Lab 2 - Experiment |
|  | Course: CS 2302 - Data Structures |
|  | Instructor: Fuentes, Olac |
|  | T.A.: Nath, Anindita |
|  | Last modified: 2/22/19 |
|  |  |
|  | Purpose of program: The purpose of this program is to demonstrate and |
|  | recognize the efficiencies of different sorting |
|  | algorithms as well as their advantages and |
|  | disadvantages. |
|  | """ |
|  | from random import randint |
|  |  |
|  | # NODE Functions ############################################################## |
|  | class Node(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self, item, next=None): |
|  | self.item = item |
|  | self.next = next |
|  |  |
|  | def PrintNodes(N): |
|  | if N != None: |
|  | print(N.item, end=' ') |
|  | PrintNodes(N.next) |
|  |  |
|  | def PrintNodesReverse(N): |
|  | if N != None: |
|  | PrintNodesReverse(N.next) |
|  | print(N.item, end=' ') |
|  |  |
|  |  |
|  | # LIST Functions ############################################################## |
|  | class List(object): |
|  | # Constructor |
|  | def \_\_init\_\_(self): |
|  | self.head = None |
|  | self.tail = None |
|  |  |
|  | def IsEmpty(L): |
|  | return L.head == None |
|  |  |
|  | def Append(L,x): |
|  | # Inserts x at end of list L |
|  | if IsEmpty(L): |
|  | L.head = Node(x) |
|  | L.tail = L.head |
|  | else: |
|  | L.tail.next = Node(x) |
|  | L.tail = L.tail.next |
|  |  |
|  | def Print(L): |
|  | # Prints list L's items in order using a loop |
|  | temp = L.head |
|  | while temp is not None: |
|  | print(temp.item, end=' ') |
|  | temp = temp.next |
|  | print() # New line |
|  |  |
|  | def PrintRec(L): |
|  | # Prints list L's items in order using recursion |
|  | PrintNodes(L.head) |
|  | print() |
|  |  |
|  | def Remove(L,x): |
|  | # Removes x from list L |
|  | # It does nothing if x is not in L |
|  | if L.head==None: |
|  | return |
|  | if L.head.item == x: |
|  | if L.head == L.tail: # x is the only element in list |
|  | L.head = None |
|  | L.tail = None |
|  | else: |
|  | L.head = L.head.next |
|  | else: |
|  | # Find x |
|  | temp = L.head |
|  | while temp.next != None and temp.next.item !=x: |
|  | temp = temp.next |
|  | if temp.next != None: # x was found |
|  | if temp.next == L.tail: # x is the last node |
|  | L.tail = temp |
|  | L.tail.next = None |
|  | else: |
|  | temp.next = temp.next.next |
|  |  |
|  | def PrintReverse(L): |
|  | # Prints list L's items in reverse order |
|  | PrintNodesReverse(L.head) |
|  | print() |
|  |  |
|  | ############################################################################### |
|  | # NEWLY IMPLEMENTED ########################################################### |
|  | ############################################################################### |
|  |  |
|  | # Receives integer n, builds, then returns a list of -------------------------- |
|  | # random integers of length n within range of 0 and k |
|  | def randList (n, k): |
|  | L = List() |
|  | for i in range(n): |
|  | Append(L, randint(0,k)) |
|  | return L |
|  |  |
|  | # Returns a copy of a given list |
|  | def copyList(L): |
|  | C = List() |
|  | temp = L.head |
|  | while temp != None: |
|  | Append(C, temp.item) |
|  | temp = temp.next |
|  | return C |
|  |  |
|  | # Returns an element at a given position of a list ---------------------------- |
|  | def elementAt(L, pos): |
|  | if L.head == None: |
|  | return None |
|  | temp = L.head |
|  | i = 0 |
|  | while i < pos: |
|  | temp = temp.next |
|  | i += 1 |
|  | return temp |
|  |  |
|  | # Returns the number of items in a list --------------------------------------- |
|  | def getLength(L): |
|  | count = 0 |
|  | temp = L.head |
|  | while temp != None: |
|  | count += 1 |
|  | temp = temp.next |
|  | return count |
|  |  |
|  | # Inserts a new node at the beginning of a list ------------------------------- |
|  | def prepend(L, data): |
|  | if IsEmpty(L): |
|  | L.head = Node(data) |
|  | L.tail = L.head #now list of length 1 |
|  | else: |
|  | L.head = Node(data, L.head) |
|  |  |
|  | ############################################################################### |
|  | # SORTING METHODS ############################################################# |
|  | ############################################################################### |
|  |  |
|  | # MERGE SORT: Separate a list, L, into 2 equal(as possible) lists. ------------ |
|  | # Repeat this process recursively until the lists can no longer be |
|  | # divided. Merge the lists together again. If an element 'A' value |
|  | # is less than another element 'B' value, append 'B' onto 'A' (A -> B) |
|  | # M(n) = 2\*M(n / 2) + n |
|  | # O(n log n) by master method |
|  | def mergeSort(L): |
|  | if getLength(L) > 1: |
|  | L1 = List() |
|  | L2 = List() |
|  | temp = L.head |
|  | i = 0 |
|  | #first half of list L |
|  | while i < getLength(L)//2: |
|  | #move items from L to L1 |
|  | Append(L1, temp.item) |
|  | Remove(L, temp.item) |
|  | temp = temp.next |
|  | i+=1 |
|  | #second half of list L |
|  | while temp != None: |
|  | #move items from L to L2 |
|  | Append(L2, temp.item) |
|  | Remove(L, temp.item) |
|  | temp = temp.next |
|  |  |
|  | mergeSort(L1) #recursively sort first half |
|  | mergeSort(L2) #recursively sort second half |
|  | mergeLists(L, L1, L2) |
|  |  |
|  | # merge two lists into ascending order |
|  | def mergeLists(L, L1, L2): |
|  | t = L1.head |
|  | u = L2.head |
|  | #comparisons of t and u |
|  | while t != None and u != None: |
|  | print("c", end="") |
|  | if t.item < u.item: |
|  | Append(L, t.item) |
|  | t = t.next |
|  | else: |
|  | Append(L, u.item) |
|  | u = u.next |
|  | #append items of 't' and 'u' to 'L' |
|  | while t != None: |
|  | Append(L, t.item) |
|  | t = t.next |
|  | while u != None: |
|  | Append(L, u.item) |
|  | u = u.next |
|  |  |
|  |  |
|  | # BUBBLE SORT: Iterate through list n times, n being the length of ------------ |
|  | # the list. Every iteration will contain 1 less element than |
|  | # the iteration before it. For every one of these iterations, |
|  | # compare every adjacent element. If an element's value is |
|  | # greater than the element directly after it, swap these two |
|  | # elements. Larger elements will "float" to the end of the list. |
|  | # Continue until the list is exhausted. |
|  | # O(n^2) |
|  | def bubbleSort(L,count): |
|  | t = L.head |
|  | u = L.head |
|  | while t != None: #number of passes = length of list |
|  | while u.next != None: #compare every adjacent element |
|  | #if out of order, switch the values of the nodes being compared |
|  | count = count + 1 |
|  | if u.item > u.next.item: |
|  | temp = u.item |
|  | u.item = u.next.item |
|  | u.next.item = temp |
|  | u = u.next #iterate thru list |
|  | count = count + 1 |
|  | t = t.next |
|  | u = L.head #start again at the beginning for the next pass |
|  | return count |
|  |  |
|  | # QUICK SORT: Select a pivot. In this case, the first element in the list. ---- |
|  | # Iterate through the rest of the list, separate all values that are |
|  | # less than and more than the pivot into 2 lists. Continue this process |
|  | # recursively, reform the list by reconnecting the two separate lists |
|  | # with pivot between them. |
|  | # Q(n) = 2 \* Q(n / 2) + n |
|  | # O(n log n) by master method |
|  | def quickSort(L): |
|  | if getLength(L) > 1: |
|  | lessL = List() |
|  | moreL = List() |
|  | pivot = L.head.item |
|  | temp = L.head.next |
|  | #sort out lists |
|  | while temp != None: |
|  | #'lessL' |
|  | print("c", end="") |
|  | if pivot > temp.item: |
|  | Append(lessL, temp.item) |
|  | #'moreL' |
|  | else: |
|  | Append(moreL, temp.item) |
|  | temp = temp.next |
|  | #2 recursive calls |
|  | quickSort(lessL) #list < pivot |
|  | quickSort(moreL) #list >= pivot |
|  | #append/prepend pivot, pivot is not forgotten |
|  | if lessL.head != None: |
|  | prepend(moreL, pivot) |
|  | else: |
|  | Append(lessL, pivot) |
|  | #attach 'lessL' and 'moreL' as list 'L' |
|  | if lessL.head != None: |
|  | L.head = lessL.head |
|  | L.tail = moreL.tail |
|  | lessL.tail.next = moreL.head |
|  | else: |
|  | #just attach 'moreL' if 'lessL' is empty |
|  | L.head = moreL.head |
|  | L.tail = moreL.tail |
|  |  |
|  | # A modified version of quicksort that makes a single recursive call ---------- |
|  | # instead of the two made by normal quicksort, processing only the |
|  | # sublist where the median is known to reside. |
|  | # rank(L, x) |
|  | # MOD(n) = 1 \* MOD(n / 2) + n |
|  | # O(n) by master method |
|  | def quickSortMOD(L, mid): |
|  | #from original 'quickSort(L)' method |
|  | if getLength(L) > 0: |
|  | pivot = L.head.item |
|  | lessL = List() |
|  | moreL = List() |
|  | temp = L.head.next |
|  | #sort out lists |
|  | while temp != None: |
|  | #'lessL' |
|  | print("c",end="") |
|  | if temp.item < pivot: |
|  | Append(lessL, temp.item) |
|  | #'moreL' |
|  | else: |
|  | Append(moreL, temp.item) |
|  | temp = temp.next |
|  |  |
|  | #implement quick sort with only one recursive call... |
|  | #base case, if the pivot is the median: |
|  | if mid==getLength(lessL) or (mid==0 and mid==getLength(lessL)): |
|  | return pivot |
|  | #decide which sublist to process: |
|  | #if median is in 'moreL' |
|  | if mid > getLength(lessL): |
|  | #1 recursive call |
|  | return quickSortMOD(moreL, mid-getLength(lessL)-1) |
|  | #if median is in 'lessL' |
|  | if mid <= getLength(lessL): |
|  | #1 recursive call |
|  | return quickSortMOD(lessL, mid) |
|  | ############################################################################### |
|  | # METHOD CALLS ################################################################ |
|  | ############################################################################### |
|  |  |
|  | def bubbleCount(L): |
|  | C = copyList(L) |
|  | count = 0 |
|  | print(getLength(C), "→", end=" ") |
|  | print(bubbleSort(C,count)) |
|  |  |
|  | def mergeCount(L): |
|  | C = copyList(L) |
|  | print(getLength(C), "→", end=" ") |
|  | mergeSort(C) |
|  | print() |
|  |  |
|  | def quickCount(L): |
|  | C = copyList(L) |
|  | print(getLength(C), "→", end=" ") |
|  | quickSort(C) |
|  | print() |
|  |  |
|  | def quickModCount(L): |
|  | C = copyList(L) |
|  | print(getLength(C), "→", end=" ") |
|  | quickSort(C) |
|  | print() |
|  |  |
|  | #different sized lists |
|  | A = randList(10, 100) |
|  | B = randList(100, 100) |
|  | C = randList(1000, 100) |
|  | D = randList(2000, 100) |
|  |  |
|  | #count how many comparisons BUBBLE SORT makes for different sized lists |
|  | print("BUBBLE SORT:", end=" ") |
|  | print("") |
|  | bubbleCount(A) |
|  | bubbleCount(B) |
|  | bubbleCount(C) |
|  | bubbleCount(D) |
|  |  |
|  | #count how many comparisons MERGE SORT makes for different sized lists |
|  | print("MERGE SORT:", end=" ") |
|  | print("") |
|  | mergeCount(A) |
|  | #mergeCount(B) |
|  | #mergeCount(C) |
|  | #mergeCount(D) |
|  |  |
|  | #count how many comparisons QUICK SORT makes for different sized lists |
|  | print("QUICK SORT:", end=" ") |
|  | print("") |
|  | quickCount(A) |
|  | #quickCount(B) |
|  | #quickCount(C) |
|  | #quickCount(D) |
|  |  |
|  | #count how many comparisons MODDED QUICK SORT makes for different sized lists |
|  | print("MODDED QUICK SORT:", end=" ") |
|  | print("") |
|  | quickModCount(A) |
|  | #quickModCount(B) |
|  | #quickModCount(C) |
|  | #quickModCount(D) |
|  |  |
|  | print() |
|  | print("Some tests are initially commented out.") |
|  | ############################################################################### |
|  |  |
|  | #End of program |

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.

* Abram Nguyen