**CS 2302 – Data Structures**

**Spring 2019**

**Lab 2**

The task for this lab was to find the median in a list of integers using the List class. To sort the lists, different sorting algorithms had to be implemented. These algorithms were bubble sort, merge sort, and two different quicksort algorithms. We also had to find the running time for each algorithm using various list lengths.

To find the median in a list of integers, several different methods had to be designed for them to be solved properly. For the List class, methods called “ElementAt” and “GetLength” had to be created to find an element at a certain index and to get the length of the list.

A method called “RandomList” also had to be made. The purpose behind it was to create a random list of integers with a range of 1 – 100. There was also a method called “Copy” which creates a copy of the original linked list to be used for each different sorting algorithm. Each sorting algorithm had a method when called, would create a copy of the linked list provided, sort the copy using a sorting algorithm called using a specifically designed method, and then return the median integer in the sorted list. A global counter is used to find the running time of each sorting algorithm.

For the bubble sort algorithm, a new method had to be created that took in a list input. The first lines of code are used to check if the linked list is empty. If not, a variable, SortDone, would be created storing a value of “False”. A while loop would then run while the value in SortDone is still “False”. During the while loop, SortDone would be set to “True” and a temporary variable would be created using the head of the linked list. Another while loop would run while the value in the temporary variable is not empty. An if statement then checks if the “next” value is greater than the temporary value. If true, a new variable, swap, is created that that stores the value in the temporary variable. Then the value in the temporary variable is replaced with the next value. The value stored in swap is then put in the spot where the value originally was. Finally, SortDone is then set to “False”. After the if statement is checked, the temporary variable pointer is moved to the next value.

For the merge sort algorithm, a new method had to be created that took in a list input. The first lines of code are used to check if the linked list is empty. If not, a temporary variable is created using the head of the list. After checking if the length of the list is greater than 1, two new empty lists are created. Two for loops are then used to split the list in half, storing the first half of the original list in the first list and the second half in the second list. Then the merge sort method is recursively called twice, the first call using the first list and the second call using the second list. After the recursive calls are done, the two lists are merged used a new method, “ListMerge”, and stored in the head of the original list. The “ListMerge” method is a method used that merges both lists provided to the method into a single list in ascending order.

For the first quicksort algorithm, a new method had to be created that took in a list input. The first lines of code are used to check if the linked list is empty. If not and the list has a length greater than 1, two new empty lists are created along with a new variable, pivot, that stores the value in the head of the list. The pointer of the list is then moved to the next value. A while loop then runs while the head of the list is not empty. An if statement checks if the value stored in the head of the list is less than the value in pivot. If so, the value is added to the first list. Another statement checks if the value in the head of the list is greater than or equal to the value in pivot. If so, the value is added to the second list. After both statements, the pointer is moved to the next value. After the while looped is complete, the value in pivot is added to the end of the first list. The method is then recursively called twice, the first time using the first list and the second time using the second list. After the calls are complete, a method called “Concate” is used to combine the linked lists. The “Concate” method uses both linked lists as an input. A temporary variable is created using the head of the first list and a while loop is used to reach the end of the list. Once the end is reached, the second list is added to the end of the first list. The first list is then returned resulting in a single linked list.

The alternate quicksort algorithm, unlike the original quicksort, only sorts the parts of the list where the median is found. It uses similar code to the first quicksort algorithm. The only difference is the recursive call. An if and else statement are added to determine which list is sorted. If the length of the first list is length than or equal to the length of the second list, a recursive call is used that sorts the first list. If not, then a recursive call sorts the second list.

Each sorting algorithm was tested using various lists of different lengths. While each list that is created has random integers, the lengths of the list that were used were 0, 1, 5, 10, 100, and 1000. Several example outputs are provided using lists with different length.

**Length 0 Trial:**

Initial count: 0

Bubble Sort Median: None

Final counter: 0

Initial count: 0

Merge Sort Median: None

Final counter: 0

Initial count: 0

Quicksort 1 Median: None

Final counter: 0

Initial count: 0

Quicksort 2 Median: None

Final counter: 0

**Length 1 Trial:**

Initial count: 0

Bubble Sort Median: 99

Final counter: 0

Initial count: 0

Merge Sort Median: 99

Final counter: 0

Initial count: 0

Quicksort 1 Median: 99

Final counter: 0

Initial count: 0

Quicksort 2 Median: 99

Final counter: 0

**Length 5 Trial:**

Initial count: 0

Bubble Sort Median: 68

Final counter: 12

Initial count: 0

Merge Sort Median: 68

Final counter: 7

Initial count: 0

Quicksort 1 Median: 68

Final counter: 17

Initial count: 0

Quicksort 2 Median: 68

Final counter: 15

**Length 10 Trial:**

Initial count: 0

Bubble Sort Median: 46

Final counter: 54

Initial count: 0

Merge Sort Median: 46

Final counter: 21

Initial count: 0

Quicksort 1 Median: 46

Final counter: 45

Initial count: 0

Quicksort 2 Median: 46

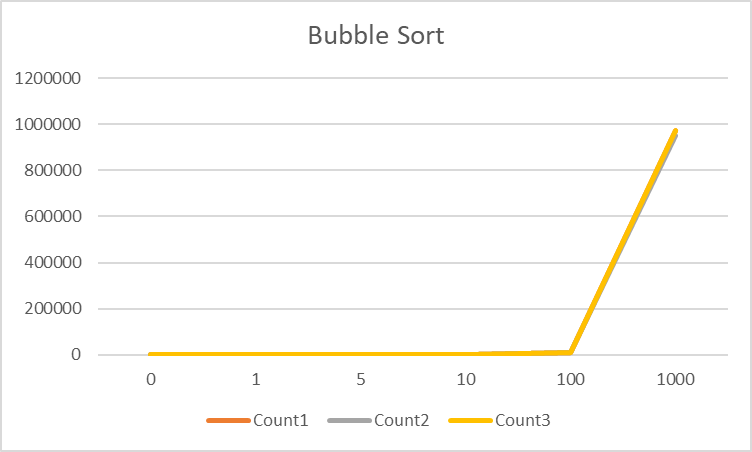
Final counter: 41

Bubble Sort Median Running Time: O()

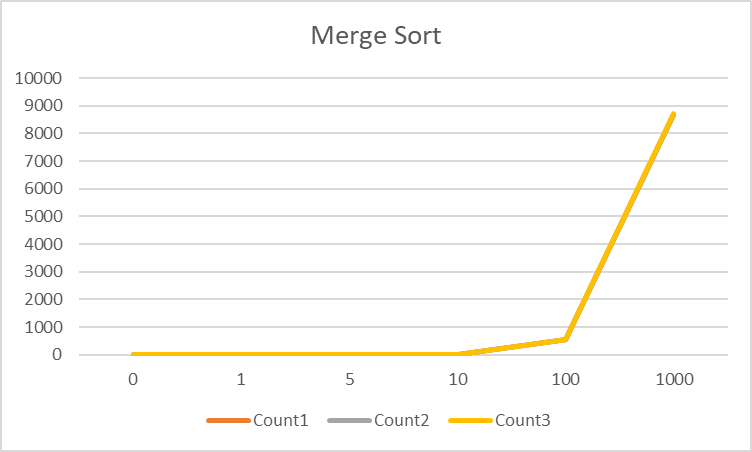
Merge Sort Median Running Time: O(n log n)

Quicksort Median Running Time: O(n log n)

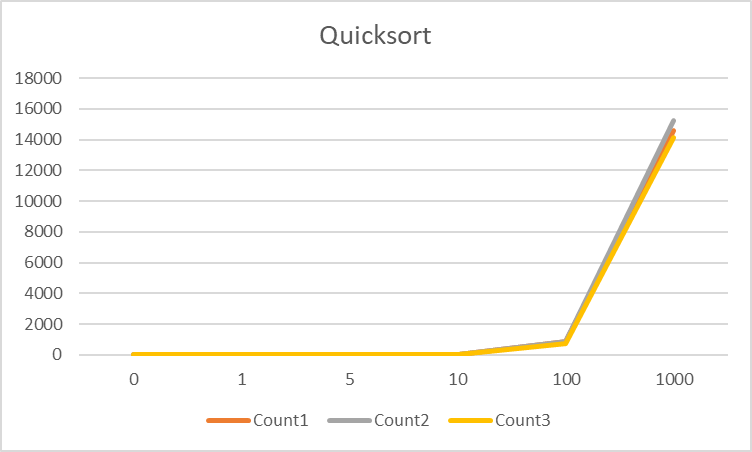
Alternate Quicksort Median Running Time: O(n log n)



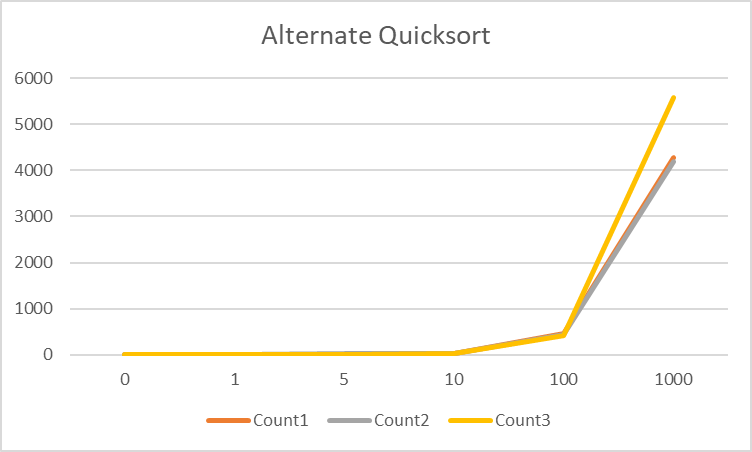
Bubble Sort Graph



Merge Sort Graph



Quicksort Graph



Alternate Quicksort Graph

During this lab, I learned how to implement different sorting algorithms to sort a list of integers in ascending order. I also learned how to implement a global counter variable to track running time for each method. I also learned different List class methods such as getting the element at a certain index, merging lists together, and adding one list to the end of another.

**Appendix**

import random

#Method to increment global counter

#Node Functions

class Node(object):

# Constructor

def \_\_init\_\_(self, item, next=None):

self.item = item

self.next = next

#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 ElementAt(L,i):

#Finds the element stored at index i in list L

if IsEmpty(L):

return None

else:

temp = L.head

count = 0

while count < i:

temp = temp.next

count+= 1

return temp.item

def GetLength(L):

#Returns the length of list L

temp = L.head

count = 0

while temp is not None:

count+= 1

temp = temp.next

return count

def RandomList(i):

#Creates a list of random int values from 1-100 with length i

L = List()

for n in range(i):

Append(L,random.randint(1,101))

return L

def Copy(L):

#Creates a copy of list L

C = List()

temp = L.head

while temp is not None:

Append(C, temp.item)

temp = temp.next

return C

def ListMerge(L1, L2):

#Merges lists L1 and L2 into a single linked list in ascending order

if L1 is None:

return L2

if L2 is None:

return L1

if(L1.item<L2.item):

L1.next = ListMerge(L1.next,L2)

increment()

return L1

else:

L2.next = ListMerge(L1,L2.next)

increment()

return L2

def Concate(L1,L2):

#Attaches the head of list L2 to the end of list L1

if L1.head is None:

return L2

if L2.head is None:

return L1

temp = L1.head

while temp.next is not None:

temp = temp.next

temp.next = L2.head

return L1

def BubbleSortMedian(L):

#Method that returns the median value of a list sorted using bubble sort

C = Copy(L)

BubbleSort(C)

return ElementAt(C,GetLength(C)//2)

def BubbleSort(L):

#Sorts linked list L using bubble sort algorithm

if IsEmpty(L):

return

sortDone = False

while sortDone is False:

sortDone = True

temp = L.head

while temp.next is not None:

if temp.next.item<temp.item:

#Swaps values in a list if one value is greater than the value

#in front of it

swap = temp.item

temp.item = temp.next.item

temp.next.item = swap

sortDone = False

temp = temp.next

increment()

#Method that returns the median value of a list sorted using merge sort

def MergeSortMedian(L):

C = Copy(L)

MergeSort(C)

return ElementAt(C,GetLength(C)//2)

def MergeSort(L):

#Sorts linked list L using merge sort algorithm

if IsEmpty(L):

return

temp = L.head

if GetLength(L)>1:

L1 = List()

L2 = List()

for i in range(GetLength(L)//2):

#Creates a new list with first half of list L

Append(L1,temp.item)

temp = temp.next

for i in range(GetLength(L)-GetLength(L)//2):

#Creates a new list with second half of list L

Append(L2,temp.item)

temp = temp.next

#Recursive calls of method "MergeSort" using newly created lists

MergeSort(L1)

MergeSort(L2)

#Stores new list created from "ListMerge" method

L.head = ListMerge(L1.head,L2.head)

#Method that returns the median value of a list sorted using quicksort

def QuicksortMedian1(L):

C = Copy(L)

Quicksort1(C)

return ElementAt(C,GetLength(C)//2)

#Sorts linked list L using quicksort algorithm

def Quicksort1(L):

if IsEmpty(L):

return

if GetLength(L)>1:

L1 = List()

L2 = List()

pivot = L.head.item

L.head = L.head.next

while L.head is not None:

#Creates new lists by comparing values in L.head to value in pivot

if L.head.item<pivot:

Append(L1,L.head.item)

increment()

elif L.head.item>=pivot:

Append(L2,L.head.item)

increment()

L.head = L.head.next

Append(L1,pivot)

increment()

#Recursive calls of method "Quicksort1" using newly created lists

Quicksort1(L1)

Quicksort1(L2)

#Combines lists L1 and L2 using method "Concate"

Concate(L1,L2)

L.head = L1.head

#Method that returns the median value of a list sorted using alternate

#quicksort

def QuicksortMedian2(L):

C = Copy(L)

Quicksort2(C)

return ElementAt(C,GetLength(C)//2)

#Sorts list L using alternate quicksort

def Quicksort2(L):

if IsEmpty(L):

return

if GetLength(L)>1:

L1 = List()

L2 = List()

pivot = L.head.item

L.head = L.head.next

while L.head is not None:

#Creates new lists by comparing values in L.head to value in pivot

if L.head.item<pivot:

increment()

Append(L1,L.head.item)

elif L.head.item>=pivot:

increment()

Append(L2,L.head.item)

L.head = L.head.next

Append(L1,pivot)

increment()

#Recursive calls for "Quicksort2" based on if else conditions

if GetLength(L1)>=GetLength(L2):

Quicksort2(L1)

else:

Quicksort2(L2)

#Combines lists L1 and L2 using method "Concate"

Concate(L1,L2)

L.head = L1.head

#Global counter

count = 0

#Increments global counter by 1

def increment():

global count

count = count+1

#Resets global counter

def reset():

global count

count = 0

#Main Method

L = List()

L = RandomList(10)

print('Random List:')

Print(L)

print()

print('Initial count: ',count)

print('Bubble Sort Median: ',BubbleSortMedian(L))

print('Final counter: ',count)

reset()

print()

print('Initial count: ',count)

print('Merge Sort Median: ',MergeSortMedian(L))

print('Final counter: ',count)

reset()

print()

print('Initial count: ',count)

print('Quicksort 1 Median: ',QuicksortMedian1(L))

print('Final counter: ',count)

reset()

print()

print('Initial count: ',count)

print('Quicksort 2 Median: ',QuicksortMedian2(L))

print('Final counter: ',count)

reset()

print()