Rigobeto Quiroz

2/26/19

Lab2 Report

CS2302 1:30PM – 2:50 PM

Description:

For this lab I had to create three different types of sorting and a variation of a sorting algorithm. Bubble, Merge, Quick, and 1 recursion call of Quick, giving a Linked list of unknown number of elements they must sort the list in ascending order, and record the number of comparisons they are making and return the midpoint of the list of n elements.

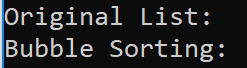
The way I was able to complete this lab was applying different techniques for sorting a singly List. For a sorting method I simply changed the information inside the nodes instead of changing the nodes itself, meanwhile in other methods I had to create new linked list that could store information as I resized the list itself. Then re-creating a new sorted linked list with the pieces of the smaller lists. As that was happening, I sorted the nodes when reconstructing the list or with a pivot point, and then I would merge all the lists together to create the sorted list.

**Bubble Sorting:**

For bubble sorting(O(n^2)) I gave it a singly linked list. As it received it, it would create a pointer to head that would be used to go through the list without losing our main head. As it went through the list using nested while loops, it’s going to check the current node and the next node, making sure they are in the correct order, if they are not then we would copy the information inside one of the nodes and swap the insides of the node and then move to the next node and repeat the same process until the list does have any more nodes to check.

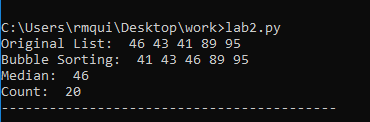
**Input: Empty List**

**Output:**



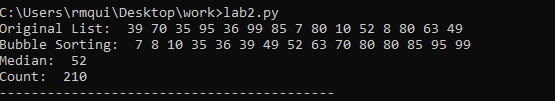
**Input: 5 elements**

**Output:**



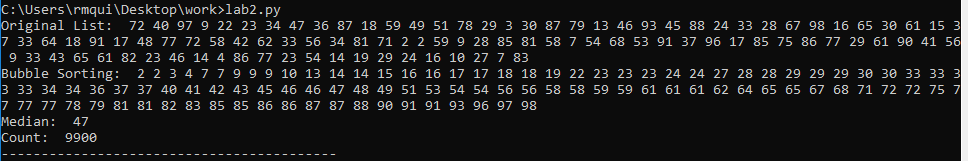
**Input: 15 elements**

**Output:**



**Input: 100 elements**

**Output:**



**Merge Sort:**

For merge sorting(nlogn) I gave it a singly linked list. As it received it, it would create a pointer to head that would be used to help me in creating another linked list. The way I did that was when reaching the midpoint of the list I created L1 list, then with the other half would create L2, another list. Then I would do a recursion call until we only had one element in each linked list. Then I called the merge method which would compare each node and create a sorted list. Any element that was not added would later be added with while loops.

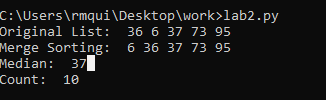
**Input: Empty List**

**Output:**



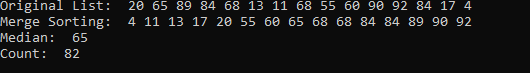
**Input: 5 elements**

**Output:**



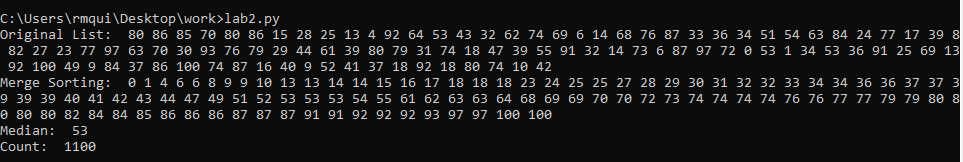
**Input: 15 elements**

**Output:**



**Input: 100 elements**

**Output:**

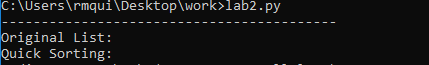


**Quick Sort:**

For Quick sort (Nlogn) I gave it a singly linked list. This sorting method will require a pivot point to spread the list into two. One that is less than the pivot point and another that is greater than the pivot point, and we will add our pivot point to the end of our first list so that we do not lose it. Then two recursion calls will be made with the split lists method creating even shorter list. Once each list has one element we will go back to each recursion call and sort each element with its pivot point until we reach our original list .

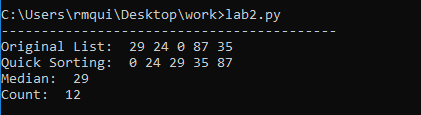
**Input: Empty List**

**Output:**



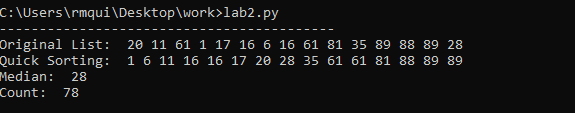
**Input: 5 elements**

**Output:**



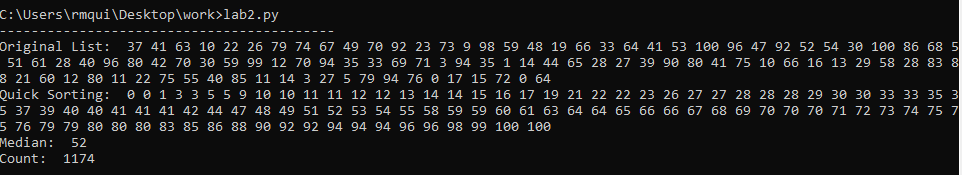
**Input: 15 elements**

**Output:**



**Input: 100 elements**

**Output:**

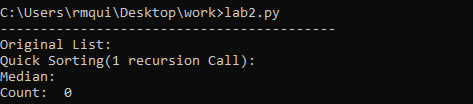


**Mod QuickSort(1 recursion call):**

For Mod QuickSort(1 recursion Call) I gave it a singly linked list. This method will work the same as quick sort, it will create two linked list, one list will store elements that are less than the pivot point and the other list will store elements that are bigger than our pivot point. Then we will check 3 conditions. If the size of our first list is greater than the length of the original list divided by two will know that the midpoint should be located in L1, otherwise we can assume that the midpoint is located in the other list. If in the special case we notice that the length of the shorten list and the length of the original list is the same, then we know that out pivot point is out midpoint.

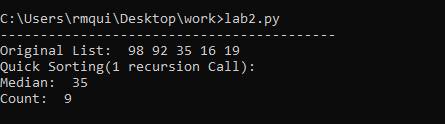
**Input: Empty List**

**Output:**



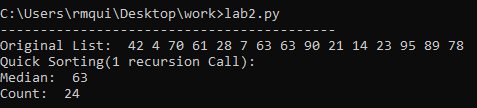
**Input: 5 elements**

**Output:**



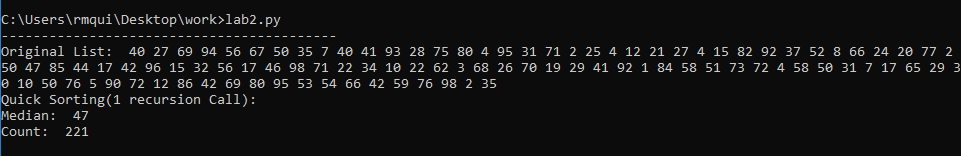
**Input: 15 elements**

**Output:**



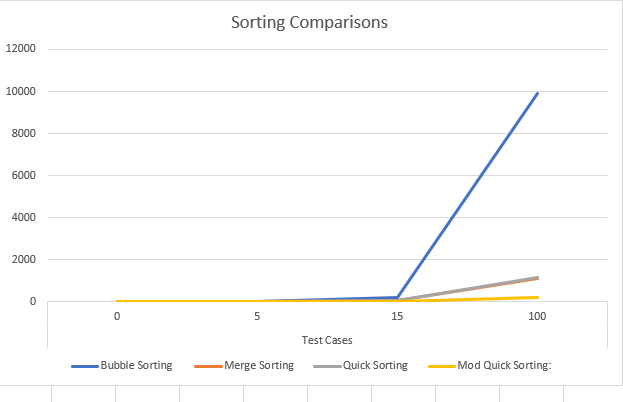
**Input: 100 elements**

**Output:**



What I learned from this lab was how to apply bubble, merge and quick sort to a singly linked list. I was also able to determine which sorting algorithm is better than the other based on the number of comparisons each algorithm made. I was also able to learn the pseudo code that is behind each of the sorting algorithms such as using a pivot point or just comparing each element until you are at the end of the list.

**Chart (Sorting Comparisons):**



**Appendix:**

**# Author: Rigoberto Quiroz**

**# Section: 1:30 PM - 2:50 PM**

**# This Program will generate a linked list of random int elements then we will**

**# sort the list with various methods(Bubble sort, merge sort, quickSort, and**

**# a single recursion call quickSort). As we are sorting the linked list we will**

**# set a counter that will count how many comparisons that sorting will have.**

**import random**

**#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()**

**def bubbleSort(L):**

**global count**

**temp1 = L.head**

**count = 0**

**#Nested Loop, sorts each element one by one**

**while temp1 is not None:**

**temp = L.head**

**#Comparison**

**while temp.next is not None:**

**count = count + 1**

**if temp.item > temp.next.item:**

**a = temp.item**

**temp.item = temp.next.item**

**temp.next.item = a**

**temp = temp.next**

**temp1 = temp1.next**

**#returns sorted list**

**return L**

**def quickSort(L):**

**global count**

**if Size(L) > 1:**

**# Selects a pivot point**

**pivot = L.head.item**

**a = L.head.next**

**L1 = List()**

**L2 = List()**

**while a is not None:**

**count = count + 1**

**# Sorts elements lower than pivot to left list and the rest to the right list**

**if a.item <= pivot:**

**Append(L1,a.item)**

**else:**

**Append(L2,a.item)**

**a = a.next**

**# Edits the list**

**L1 = quickSort(L1)**

**L2 = quickSort(L2)**

**# Since we lose our pivot point, we have to re-add it**

**Append(L1, pivot)**

**# merge the to list**

**return merge(L1,L2)**

**else:**

**return L**

**def merge(L1,L2):**

**if IsEmpty(L1):**

**return L2**

**if IsEmpty(L2):**

**return L1**

**L1.tail.next = L2.head**

**L1.tail = L2.tail**

**return L1**

**def modQuickSort1(L):**

**if Size(L)//2 == 1:**

**return modQuickSort(L,Size(L))**

**else: # even length**

**return (modQuickSort(L, (Size(L) - 1) // 2))**

**def modQuickSort(L, K):**

**global count**

**if Size(L) > 1:**

**# Selects a pivot point**

**pivot = L.head.item**

**a = L.head.next**

**L1 = List()**

**L2 = List()**

**while a is not None:**

**count = count + 1**

**# Sorts elements lower than pivot to left list and the rest to the right list**

**if a.item <= pivot:**

**Append(L1,a.item)**

**else:**

**Append(L2,a.item)**

**a = a.next**

**if Size(L1) > K:**

**return modQuickSort(L1,K)**

**if Size(L1) == K:**

**return pivot**

**if Size(L1) < K:**

**return modQuickSort(L2, K - Size(L1))**

**else:**

**return L**

**# Merge Sort**

**def mergeSort(L):**

**if not IsEmpty(L) and L.head.next is not None:**

**temp = L.head**

**#Create two list, and split the original List in two halfs**

**L1, L2 = splitList(L)**

**L1 = mergeSort(L1)**

**L2 = mergeSort(L2)**

**sortedList = merge1(L1,L2)**

**return sortedList**

**else:**

**return L**

**def merge1(L1, L2):**

**global count**

**sL = List()**

**currentNode1 = L1.head**

**currentNode2 = L2.head**

**#Compares the two list items and depending on which item we add we will advance**

**# to the next element of that list.**

**while currentNode1 is not None and currentNode2 is not None:**

**count = count + 1**

**if currentNode1.item < currentNode2.item:**

**Append(sL,currentNode1.item)**

**currentNode1 = currentNode1.next**

**else:**

**Append(sL,currentNode2.item)**

**currentNode2 = currentNode2.next**

**#Get any elements that were left over.**

**while currentNode1 is not None:**

**Append(sL, currentNode1.item)**

**currentNode1 = currentNode1.next**

**while currentNode2 is not None:**

**Append(sL, currentNode2.item)**

**currentNode2 = currentNode2.next**

**return sL**

**def splitList(L):**

**temp = L.head**

**L1 = List()**

**L2 = List()**

**n = 0**

**# splits list into two halfs**

**while n < Size(L)//2:**

**Append(L1,temp.item)**

**n = n + 1**

**temp = temp.next**

**while n < Size(L):**

**Append(L2,temp.item)**

**n = n + 1**

**temp = temp.next**

**return L1, L2**

**def Size(L):**

**if L is None:**

**return 0**

**#Gets length of list**

**temp = L.head**

**count = 0**

**while temp is not None:**

**count = count + 1**

**temp = temp.next**

**return count**

**def Median(L):**

**# gets the midpoint of list**

**if Size(L) % 2 != 0:**

**temp = L.head**

**n = 0**

**while n < Size(L) // 2:**

**temp = temp.next**

**n = n + 1**

**return temp.item**

**temp = L.head**

**n = 0**

**while n < Size(L)//2:**

**temp = temp.next**

**n = n + 1**

**return temp.item**

**L = List()**

**for i in range(100):**

**Append(L,random.randint(0,100))**

**count = 0**

**'''**

**print('Original List: ',end =' ')**

**Print(L)**

**print('Bubble Sorting: ', end = ' ')**

**Print(bubbleSort(L))**

**print('Median: ',end = ' ')**

**print(Median(bubbleSort(L)))**

**print('Count: ', count)**

**print('------------------------------------------')**

**'''**

**'''**

**count = 0**

**print('Original List: ',end =' ')**

**Print(L)**

**print('Merge Sorting: ', end = ' ')**

**Print(mergeSort(L))**

**print('Median: ',end = ' ')**

**print(Median(mergeSort(L)))**

**print('Count: ', count)**

**count = 0**

**print('------------------------------------------')**

**print('Original List: ',end =' ')**

**Print(L)**

**print('Quick Sorting: ', end = ' ')**

**Print(quickSort(L))**

**print('Median: ',end = ' ')**

**print(Median(quickSort(L)))**

**print('Count: ', count)**

**count = 0**

**print('------------------------------------------')**

**print('Original List: ',end =' ')**

**Print(L)**

**print('Quick Sorting(1 recursion Call): ', end = ' ')**

**Print(modQuickSort(L))**

**print('Median: ',end = ' ')**

**print(Median(modQuickSort(L)))**

**print('Count: ', count)**

**'''**

**print('------------------------------------------')**

**print('Original List: ',end =' ')**

**Print(L)**

**print('Quick Sorting(1 recursion Call): ', end = ' ')**

**print('\nMedian: ',end = ' ')**

**print(modQuickSort(L,Size(L)//2))**

**print('Count: ', count)**