Sergio Gramer – 88521512
Lab #2: (Sorting Methods) Report

In Lab 2 for CS2302 we are creating methods to sort lists using three different sorting methods: Bubble Sort, Merge Sort and Quick Sort. We are creating non-native python lists (linked lists) as our input data structure. This data structure has been included by the Professor and is a singly-linked list or does not have the ability to see previous nodes. By sorting a random a list we are tasked to find the "median" value, meaning that after we sort the list; we should be able to find the middle value of the integers by finding the middle element of the list and returning it. Instead of finding the Big-O running time we were tasked to find the number of comparisons each sorting method would do.

My initial proposed solution design and implementation for Bubble Sort was to traverse a list and use a Boolean value "swap" to state whether a swap in the list had been made. If so, it would re-iterate through the list once again. If the algorithm found that by comparing the first value vs. the next if any one of the two were higher a swap would be made and the next value would become the first and vice versa. After this, the first value would become the next and so forth.

NOTE: Comments have been removed from code for readability

The following is a piece of my Bubble Sort Method:

```
def BubbleSort(L) :
global bubblecount
swap = True
ptr = L.head

while swap  == True :
ptr = L.head
swap = False

while ptr.next is not None :
bubblecount += 1

if ptr.item > ptr.next.item :
temp = ptr.next.item
ptr.next.item = ptr.item
ptr.item = temp
ptr = ptr.next
swap = True

else :
ptr = ptr.next

return L
```

For Merge Sort my implementation was going to be a bit different. I would still have to iterate through the list but this time I would have to create two lists in order to shorten the problem. These two lists were then sent back to the method recursively which would shorten the problem up until we had only one item left and our base case would return that value, thus jumping out of the recursion. After this I would have to begin comparing which of the two values were bigger and by using the Append method included in our program, append it to a list. My first problem was that I was only recursively calling the same method which would split the lists. After this, in that same method I would try to begin merging all of the lists together. This specific implementation was failing and had to be redone. My final implementation was to create two separate methods, one to split the lists and one to merge them back together in order from least to greatest.

The following is a piece of my Merge Sort Method:

```
def MergeSort(L) :
   if IsEmpty(L):
      return L

   if SizeList(L) == 1 :
      return L

   ptr = L.head
   L1 = List()
   L2 = List()
   mid = SizeList(L) // 2

   for i in range(mid) :
      Append(L1, ptr.item)
      ptr = ptr.next

   while ptr is not None :
      Append(L2, ptr.item)
      ptr = ptr.next

   L1 = MergeSort(L1)
   L2 = MergeSort(L2)
   L = Merge(L1, L2)

   return L
```

Quick Sort also, gave me a hard time, for the initial design and implementation. Again, I was attempting at recursively calling the method to break the list up into two separate lists after finding a pivot point. Still trying to piece them back together in the same method I was getting negative results. I had to create a separate method to merge the two lists back together. Almost like in Merge Sort except this time my first list would be inserted into a third list first and then append the pivot point which was stored as a separate variable to compare to the rest of the list during the split phase.

The following is a piece of my Quick Sort Method:

```
def QuickSort(L) :
   global qscount
   L1 = List()
   L2 = List()
   if IsEmpty(L) :
      return L

   if SizeList(L) == 1 :
      return L

   else:
      piv = L.head
      ptr = L.head.next

      while ptr != None :

         if ptr.item < piv.item :
            Append(L1, ptr.item)
            qscount += 1

         else :
            Append(L2, ptr.item)
            qscount += 1

         ptr = ptr.next

      L1 = QuickSort(L1)
      L2 = QuickSort(L2)
      L = QuickSortMerge(L1, L2, piv)
   return L
```

Some of the test cases that I did raised some concerns for me. In class, I was taught that all of the methods were different in running times with Quick Sort being the fastest out of all of them. It was stated that Quick Sort is not much faster than Merge Sort and therefore we can conclude that they are almost the same. My results concluded that; sometimes Quick Sort is faster than Merge Sort but as my number increased Merge Sort had less comparisons made in the program. Attached are some pictures of my results.

```
In [2]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab2.1.py', wdir='/Users/
SergioGramer/Desktop/Spring2019/CS2302')
Size of lists:  3
BubbleSort begin.
39 93 72
39 72 93
Bubble Sort Median:  39
Bubble sort compared:  4  times.

MergeSort begin.
39 93 72
39 72 93
Merge Sort Median:  39
Merge compared:  2  times.

QuickSort begin:
39 93 72
39 72 93
Quick Sort Median:  39
Quick Sort compared:  3  times.
```

As we can see from the above picture. At size of list: 3 quick sort compared more times than merge sort.

```
In [3]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab2.1.py', wdir='/Users/
SergioGramer/Desktop/Spring2019/CS2302')
Size of lists:  5
BubbleSort begin.
32 10 26 87 81
10 26 32 81 87
Bubble Sort Median:  26
Bubble sort compared:  8  times.

MergeSort begin.
32 10 26 87 81
10 26 32 81 87
Merge Sort Median:  26
Merge compared:  6  times.

QuickSort begin:
32 10 26 87 81
10 26 32 81 87
Quick Sort Median:  26
Quick Sort compared:  6  times.
```

Equal comparisons.

```
In [4]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab2.1.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302')
Size of lists:  7
BubbleSort begin.
87 57 94 81 22 91 78
22 57 78 81 87 91 94
Bubble Sort Median:  78
Bubble sort compared:  30  times.

MergeSort begin.
87 57 94 81 22 91 78
22 57 78 81 87 91 94
Merge Sort Median:  78
Merge compared:  14  times.

QuickSort begin:
87 57 94 81 22 91 78
22 57 78 81 87 91 94
Quick Sort Median:  78
Quick Sort compared:  11  times.
```

```
In [5]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab2.1.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302')
Size of lists:  11
BubbleSort begin.
84 80 65 96 70 56 97 40 12 36 62
12 36 40 56 62 65 70 80 84 96 97
Bubble Sort Median:  62
Bubble sort compared:  90  times.

MergeSort begin.
84 80 65 96 70 56 97 40 12 36 62
12 36 40 56 62 65 70 80 84 96 97
Merge Sort Median:  62
Merge compared:  27  times.

QuickSort begin:
84 80 65 96 70 56 97 40 12 36 62
12 36 40 56 62 65 70 80 84 96 97
Quick Sort Median:  62
Quick Sort compared:  31  times.
```

```
In [6]: runfile('/Users/SergioGramer/Desktop/Spring2019/CS2302/Lab2.1.py', wdir='/Users/SergioGramer/Desktop/Spring2019/CS2302')
Size of lists:  21
BubbleSort begin.
101 49 21 69 2 43 93 98 97 68 87 15 32 21 38 73 87 96 28 14 21
2 14 15 21 21 21 28 32 38 43 49 68 69 73 87 87 93 96 97 98 101
Bubble Sort Median:  43
Bubble sort compared:  380  times.

MergeSort begin.
101 49 21 69 2 43 93 98 97 68 87 15 32 21 38 73 87 96 28 14 21
2 14 15 21 21 21 28 32 38 43 49 68 69 73 87 87 93 96 97 98 101
Merge Sort Median:  43
Merge compared:  67  times.

QuickSort begin:
101 49 21 69 2 43 93 98 97 68 87 15 32 21 38 73 87 96 28 14 21
2 14 15 21 21 21 28 32 38 43 49 68 69 73 87 87 93 96 97 98 101
Quick Sort Median:  43
Quick Sort compared:  82  times.
```

In the last picture when the size of the list is 21; we can see that quick sort does a considerable higher amount of comparisons.

In conclusion, I learned that lists are a lot different to navigate in Python than in Java. I was having a hard time adjusting to how to traverse the lists and the nodes. Also, though (in my opinion) the merge sort was a lot harder to implement than the quick sort it works better at comparing these integer linked lists than quick sort. I had believed that quick sort was going to be exponentially faster than merge sort but I stand corrected. Even though bubble sort was easier to implement and looks like less code to read, the comparisons that it makes to adjust a list is incredibly higher than both quick sort and merge sort. Thus, making it the sorting method I would least use to compare and sort a list like this. Even though the applications at a smaller scale are almost identical with merge sort being the quickest by one next to quick sort being quicker than bubble sort by one comparison as well.

APPENDIX

lab2.1.py (90% of the linked list implementation was done by Dr. Fuentes, with the exception of SizeList() and ElementAt() methods) Sorting methods were done by me (Sergio Gramer)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 18 17:24:11 2019

@author: SergioGramer
"""
import copy
import random

def Copy(L):
    ptr = L.head
    L1 = List()

    while ptr is not None:
        Append(L1,ptr.item)
        ptr = ptr.next

    return L1

def Median(L):
    C = copy.copy(L)
    return ElementAt(C, SizeList(C)//2)

def ElementAt(L, n) :
    ptr = L.head
    count = 1

    if IsEmpty(L) :
        return

    else :

        while count != n :
            ptr = ptr.next
            count += 1

        if count == n :
            return ptr.item

        else :
            return None

#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 SizeList(L) :
    counter = 1
    co = L.head

    while co.next is not None :
        counter += 1
        co = co.next

    return counter

def BubbleSort(L) :
    global bubblecount
    swap = True #variable to make sure we swap again and check list when at least 1 was swapped
```

```
    ptr = L.head

    while swap  == True :
        ptr = L.head
        swap = False

        while ptr.next is not None :
            bubblecount += 1

            if ptr.item > ptr.next.item :
                temp = ptr.next.item #swapping variables from 1 node to another
                ptr.next.item = ptr.item
                ptr.item = temp
                ptr = ptr.next
                swap = True

            else :
                ptr = ptr.next #will ensure traversal of list in none applies

    return L

def MergeSort(L) :
    if IsEmpty(L): #ensures we don't send empty lists recursively
        return L   #need to check these 2 first before assigning ptr to L.head

    if SizeList(L) == 1 : #we don't have anything to split if list is of size 1
        return L

    ptr = L.head
    L1 = List() #Will be the 2 lists we use to split the original list into smaller
    L2 = List() #problems for recursion
    mid = SizeList(L) // 2 #will be used to know when to stop the first list

    for i in range(mid) :
        Append(L1, ptr.item) #begins creating the first list
        ptr = ptr.next

    while ptr is not None :
        Append(L2, ptr.item) #begins creating the second list
        ptr = ptr.next

    L1 = MergeSort(L1) #recursively makes the first list smaller until it cant
    L2 = MergeSort(L2) #recursively makes the second list smaller until it cant
    L = Merge(L1, L2) #my method to merge the 2 lists

    return L

def Merge(L1, L2) : #Will Merge the 2 lists for merge sort
    global mergecount
    L3 = List()
    temp1 = L1.head #will be used to traverse the lists and comapre
    temp2 = L2.head

    while temp1 is not None and temp2 is not None : #ensuring we are not comparing
        if(temp1.item > temp2.item) : #if the first item is greater add the second to the final list
            mergecount += 1 #global counter to keep track of compares
            Append(L3, temp2.item)
            temp2 = temp2.next

        else : #meaning second item is greater so add the other
            mergecount += 1
            Append(L3, temp1.item)
            temp1 = temp1.next

    if temp1 == None : # we ran out of temp 1 variables to compare
        while temp2 is not None : #adding temp2 to list meanwhile not none
            Append(L3, temp2.item)
            temp2 = temp2.next

    if temp2 == None :
```

```
        while temp1 is not None : #we ran out of temp 2 variables to compare
            Append(L3, temp1.item) #adding left over temp1 variables to list meanwhile not none
            temp1 = temp1.next

    return L3

def QuickSort(L) :
    global qscount
    L1 = List() #will be the 2 lists used to split the problem into smaller pieces
    L2 = List()

    if IsEmpty(L) : #checks if list is empty before assigning pivot to anything
        return L     #return if empty

    if SizeList(L) == 1 : #if list is 1 we don't need to split it into smaller list
        return L

    else:
        piv = L.head #pivot will be used as the point where we start comparing
        ptr = L.head.next #our pointer will start after pivot because its the 2 item in list

        while ptr != None : #while pointer is not none begin assignation of varibles

            if ptr.item < piv.item : #all items smaller than pivot go into one list
                Append(L1, ptr.item)
                qscount += 1

            else :
                Append(L2, ptr.item) #all other items are greater so go into another list
                qscount += 1

            ptr = ptr.next #doesn't let iteration stop

        L1 = QuickSort(L1) #recursively make problem smaller
        L2 = QuickSort(L2) #recursivelt make second lsit smaller
        L = QuickSortMerge(L1, L2, piv) #this method will merge both lists sends pivot as well to add to list
    return L

def QuickSortMerge(L1, L2, piv) : #quick sort merging method
    L3 = List() #will be our final list
    temp = L1.head #will allow us to iterate the first list
    temp2 = L2.head #will allow us to iterate the second list

    while temp is not None : # going through smaller list to add before pivot
         Append(L3, temp.item)
         temp = temp.next

    Append(L3, piv.item) #adds the pivot before adding the bigger elements

    while temp2 is not None : #going through bigger list to add after pivots
        Append(L3, temp2.item)
        temp2 = temp2.next

    return L3

bubblecount = 0 #comparisons counters
mergecount = 0
qscount = 0

n = 20
L = List() #Creates a random list of size 6 from random ints of 0 to 100
for i in range(n+1) :
    Append(L,random.randint(0,101))

#copying list to all the other lists so we can use the same list for all sorting
LBS = Copy(L) #this is the BubbleSort list
LMS = Copy(L) #this is the MergeSort list
LQS = Copy(L) #this is the QuickSort list

print('Size of lists: ', n+1)
```

```
print('BubbleSort begin.') #LBSF = List Bubble Sort Final
Print(LBS)
LBSF = BubbleSort(LBS)
Print(LBSF)
print('Bubble Sort Median: ',Median(LBSF))
print('Bubble sort compared: ', bubblecount, ' times.', "\r\n")

print('MergeSort begin.') # LMSF = List Merge Sort Final
Print(LMS)
LMSF = MergeSort(LMS)
Print(LMSF)
print('Merge Sort Median: ',Median(LMSF))
print('Merge compared: ', mergecount, ' times.', "\r\n")

print('QuickSort begin: ')
Print(LQS)
LQSF = QuickSort(LQS)
Print(LQSF)
print('Quick Sort Median: ', Median(LQSF))
print('Quick Sort compared: ', qscount, ' times.', "\r\n")
```

ACADEMIC CERTIFICATION

I “Sergio Gramer” 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.

Signed: 02/26/2018

Sergio Gramer

____