Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: February25, 2019

Instructor: Olac Fuentes

Assignment: Lab 2 Median

TA: Anindita Nath & Maliheh Zaragan

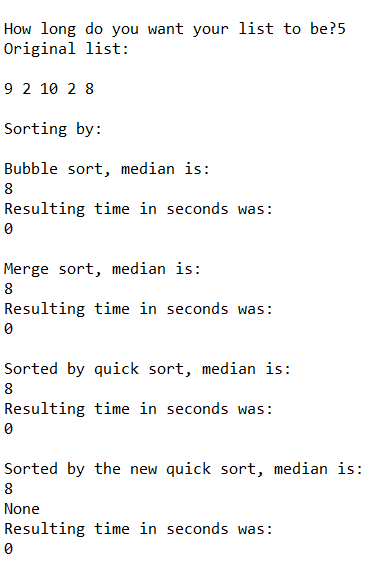
**Introduction:**

In python, you can use Linked Lists to do many functions, one of these functions is to find the median of a certain list. But there is one issue, for you to get the best result, you must sort the List by ascending order to find your value. We had the task to implement three different sorting algorithms to find our Median; the three were Bubble Sort, Merge Sort, and Quicksort. But to make sure which was the best one, we were tasked on comparing these algorithms to find the fastest one.

**Proposed solution design and implementation:**

1. Bubble Sort: There is not a lot of freedom when trying to design the implementation of bubble sort, the idea is to pass higher numbers to the end of the list over and over until we reach the actual sorted list. Perhaps the biggest challenge was to implement this algorithm using linked lists, as using arrays would have been a little easier to do. Either way, I would have to find a way to alter the list with each term moving to the end if necessary, as well as making sure that if the list is correct, that the method would check its status once and end.
2. Merge Sort: There is not a lot of freedom when trying to design the implementation of merge sort, the idea is to split the values of the list by halves and sort each term to their respective place over and over until we reach the actual sorted list. Perhaps the biggest challenge was to implement this algorithm using linked lists, as using arrays would have been a little easier to do. Either way, I would have to find a way to split the list into two, and use the merge() method to create the newly formed list. In order words, I would then need to use two methods for the one function.
3. Quick Sort: There is not a lot of freedom when trying to design the implementation of quick sort, the idea is to select a number from the list and create a breaking point for the list to break the list into pieces like merge sort until given the specific sorted list. Perhaps the biggest challenge was to implement this algorithm using linked lists, as using arrays would have been a little easier to do. Either way, I would have to find a way to alter the list in the context of the list, as well as grab a good candidate to split the list up.
4. Quick Sort 2: There is not a lot of freedom when trying to design the second implementation of quick sort, the idea is to select a number from the list and create a breaking point for the list to break the list into pieces like merge sort until given the specific sorted list. Perhaps the biggest challenge is to find a way to call the list over again using only one recursive call in the method, unlike the first quicksort instead of the two

**Experimental results**:

The example on the left represents a normal output of 5 terms in the given linked list. After the result of the new quick sort, there was a term that outputted none. I tried to find ways to remove it, but it remained. Other than that, the four methods display the same result, which is a big factor of this lab. I should also mention that although I don’t mention it in the report, the code has a median method for each individual sorting algorithm.

1. Bubble Sort(0(: For this method, I had to create a variable called Completed that would tell me whenever the list was completed in its sorting process. While Completed wasn’t true, the list would sort itself until the desired results were made, and finally returning the middle element in that list.
2. Merge Sort(0(:: For this method, I had to create three lists, one for the left terms, one for the right terms, and an empty one to become a vessel for the results of the other two lists. My process was pretty much straight forward from the implementation process, but I had to add a Merger method to combine my two new lists into one by indicating whenever the terms of one list is empty, to fill in the void for the other one. At the end, the new list will get its median, and the process was over.
3. Quick Sort(0(:: For the first quick sort, I made two empty lists that if the current term was less than the middle, it would be added to the first list, else on the second. The process would continue until gathered the desired list and would later return the middle element.
4. Quick Sort 2(0(:: For the second quick sort, I made two empty lists that if the current term was less than the middle, it would be added to the first list, else on the second. The big change from this method from the first Quick Sort is that this method calls itself once for both lists, whilst the first Quick sort calls itself twice for each list and dilemma. The process would continue until gathered the desired list and would later return the middle element.

Determining the big O notations for my methods wasn’t as difficult, although getting the seconds was. The table below was the result of the four methods and the time it took for them to complete each assignment in seconds.

|  |  |  |  |
| --- | --- | --- | --- |
| Method | 100 terms | 1,000 terms | 2,000 terms |
| Bubble Sort | **3** | **373** | **808** |
| Merge Sort | **3** | **40** | **80** |
| Quick Sort 1 | **0** | **33** | **40** |
| Quick Sort 2 | **0** | **8** | **6** |

As shown above, the fastest method was the second Quick Sort, followed by the first Quick Sort, then Merge Sort, and finally Bubble Sort.

**Conclusions**:

With this lab, I was able to learn to code better using the Python language, including using many sorting algorithms to find a specific integer. I was able to learn to solve different problems by using Linked Lists throughout my lab.

**Appendix :**

**Median.py**

"""

Author: Carlos Fernando Castaneda

Class : CS 2302

Date Modified: February2, 2019

Instructor: Olac Fuentes

Assingment: Lab 2 Median

TA: Anindita Nath & Maliheh Zaragan

Purpose: to implement several algorithms for finding the median of a list of integers, using

objects of the List class described in class, and compare their running times (measured as the number of

comparisons each algorithm makes) for various list lengths.

"""

#Used to make the random list

import random

#Used to calculate the time for each method

import time

#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)

#List Functions

class List(object):

# Constructor

def \_\_init\_\_(self):

self.head = None

self.tail = None

self.Len = 0

#Makes a new empty list

def NewList(L):

L.head = None

L.tail = None

L.Len = 0

#Makes a new List with a size of n terms, as listed in the instructions

def RandomList(n):

L = List()

i = n

while i>0:

Append(L,random.randint(0,n\*2))

i = i - 1

return L

#Sorting Algorithm Functions

def BubbleSort(L):

if IsEmpty(L):

return None

else:

Current = L.head

Completed = False

while Completed != True:

Completed = True

Current = L.head

while Current.next is not None:

if Current.item > Current.next.item:

nextItem = Current.next.item

Current.next.item = Current.item

Current.item = nextItem

Completed = False

Current = Current.next

def QuickSort(L):

if L.Len > 1:

middle = L.head.item

List1 = List()

List2 = List()

Current = L.head.next

while Current != None:

if Current.item < middle:

Append(List1, Current.item)

else:

Append(List2, Current.item)

Current = Current.next

QuickSort(List1)

QuickSort(List2)

if IsEmpty(List1):

Append(List1, middle)

else:

Prepend(List2, middle)

if IsEmpty(List1):

L.head = List2.head

L.tail = List2.tail

else:

List1.tail.next = List2.head

L.head = List1.head

L.tail = List2.tail

def MergeSort(L):

if L.Len > 1:

L1 = List()

L2 = List()

NewLength = L.Len//2

Current = L.head

for i in range(NewLength):

Append(L1, Current.item)

Current= Current.next

while Current != None:

Append(L2, Current.item)

Current = Current.next

MergeSort(L1)

MergeSort(L2)

NewList(L)

MergeList(L, L1, L2)

def QuickSort2(L, Median):

List1 = List()

List2 = List()

if L.Len <= 1:

return L.head.item

middle = L.head.item

Current = L.head.next

while Current != None:

if Current.item < middle:

Append(List1, Current.item)

else:

Append(List2,Current.item)

Current = Current.next

if List1.Len > Median :

return QuickSort2(List1, Median)

elif(List1.Len == 0 and Median == 0):

return middle

elif(List1.Len == Median):

return middle

else:

return QuickSort2(List2, Median - List1.Len - 1)

#Median Algorithms

def MedianBubble(L):

C = Copy(L)

BubbleSort(C)

NewLength = C.Len//2

Current = C.head

for i in range(NewLength):

Current = Current.next

return Current.item

def MedianMerge(L):

C = Copy(L)

MergeSort(C)

NewLength = C.Len//2

Current = C.head

for i in range(NewLength):

Current = Current.next

return Current.item

def MedianQuick(L):

C = Copy(L)

QuickSort(C)

NewLength = C.Len//2

Current = C.head

for i in range(NewLength):

Current = Current.next

return Current.item

def MedianQuick2(L):

C = Copy(L)

LengthCopy = C.Len//2

print(QuickSort2(C, LengthCopy))

#States if the current List is empty or not

def IsEmpty(L):

return L.head == None

# Inserts x at end of list L

def Append(L,x):

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

L.Len = L.Len + 1

else:

L.tail.next = Node(x)

L.tail = L.tail.next

L.Len = L.Len + 1

# Inserts x at beginging of list L

def Prepend(L,x):

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

L.Len = L.Len + 1

else:

L.head=Node(x,L.head)

L.Len = L.Len + 1

# Prints list L's items in order using a loop

def Print(L):

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print() # New line

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

# Appends sorted Lists into L

def MergeList(L,L1,L2):

#Grabs the two head of each respective list. Called the variables current as it is the current item the algorithm is analyzing

Current1 = L1.head

Current2 = L2.head

while Current1 != None and Current2 != None:

#Adds the lowest term first of either list

if Current1.item < Current2.item:

Append(L, Current1.item)

Current1 = Current1.next

else:

Append(L, Current2.item)

Current2 = Current2.next

#Clarifies that if either list contains any elements, if so, they will add any remaining items to the new list

if Current1 is None:

while Current2 != None:

Append(L, Current2.item)

Current2 = Current2.next

if Current2 is None:

while Current1 != None:

Append(L, Current1.item)

Current1 = Current1.next

#Copies the content of a list and returns that copy

def Copy(L):

copy = List()

t = L.head

while t != None:

Append(copy,t.item)

t = t.next

return copy

#Makes a random list

ListSize = int(input('How long do you want your list to be?'))

L = RandomList(ListSize)

print('Original list: ')

print('')

Print(L)

print('')

print('Sorting by:')

print('')

print('Bubble sort, median is: ')

start1 = int(time.time()\*1000)

print(MedianBubble(L))

end1 = int(time.time()\*1000)

print('Resulting time in seconds was: ')

print(end1-start1)

print('')

print('Merge sort, median is: ')

start2 = int(time.time()\*1000)

print(MedianMerge(L))

end2 = int(time.time()\*1000)

print('Resulting time in seconds was: ')

print(end2-start2)

print('')

print('Sorted by quick sort, median is: ')

start3 = int(time.time()\*1000)

print(MedianQuick(L))

end3 = int(time.time()\*1000)

print('Resulting time in seconds was: ')

print(end3-start3)

print('')

print('Sorted by the new quick sort, median is: ')

start4 = int(time.time()\*1000)

print(MedianQuick2(L))

end4 = int(time.time()\*1000)

print('Resulting time in seconds was: ')

print(end4-start4)

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 provide inappropriate assistance to any student in the class.

