**Introduction**

The problem that was given is to find the median of a list of random numbers. Easiest way to get the median is to sort the list and find the element at length/2. On this lab assignment, we will sort in three different ways, bubble sort, merge sort, and quick sort. Another way to find the median is by modifying the quicksort. Therefore, there are four ways to find the median and at the end, compare the numbers of total comparison for each algorithm.

At first, the program will let the user to input an integer which will be the length of the list. By calling “createList” method, a list will be created with random numbers. “Median” method will receive a list and it will call a sorting method to find the median. It will display the original list and the median on the screen at the end. In addition, I added to keep track of number of comparison as well as the time that takes to find the median for each algorithm.

**Bubble Sort**

The algorithm of bubble sort is to swap whenever there is a pair that is not in ascending order. Therefore, I will need two while loops that will check if the list is in order and to iterate through the list. Inside of the second while loop, it requires to have an if-statement to check if the pair is not in right order. After having a sorted list, call a method ElementAt to get the item at median.

Enter the length of the list.5

List: 100 29 45 81 49

Using bubble Sort.

The median is: 49

12 comparison

0 seconds.

Enter the length of the list.10

List: 74 15 90 17 26 66 35 32 82 39

Using bubble Sort.

The median is: 39

45 comparison

0 seconds.

**Merge Sort**

This sorting algorithm is recursively, thus, I will need a base case which is if the length is less than 2 return L. Next step is to separate a list into two different lists, with these two new lists do the recursive call until base case is executed. After returning the list, have a while loop which will run as long as the length of the original list is same as the new list which will contain both left and right list. Possible conditions which can appear is: 1. Right list is empty 🡪 append item of left list, 2. Left list is empty 🡪 append item of right list, 3. Item of right list is smaller than item left list 🡪 append item of right list, and 4. Else 🡪 append item of left list. After while loop, a list which contains all the items from left and right will be returned.

Enter the length of the list.5

List: 36 69 34 22 2

Using Merge Sort.

The median is: 34

7 comparison

0 seconds.

Enter the length of the list.10

List: 91 57 28 59 35 43 48 82 69 83

Using Merge Sort.

The median is: 59

22 comparison

0 seconds.

**Quick Sort**

This algorithm is a little bit similar to merge sort since the original list is separated into two list. In this case, partition of a list is determined by whether items are greater than pivot. After having two list, do recursive calls until the base case is executed, and lists are returned. As fact, we know that anything inside of a smaller list is less than pivot and larger list is greater than pivot. Now, all that has to be done is to connect each part of lists and return it. There are three possibilities to do this. The first case is if smaller list is empty, larger list will be linked after the pivot. Second, if larger list is empty, smaller list will be linked before pivot. For the last case, pivot will be added to smaller list and linked to larger list since pivot.

Enter the length of the list.5

List: 6 3 75 52 47

Using Quick Sort.

The median is: 47

7 comparison

0 seconds.

Enter the length of the list.10

List: 36 48 42 72 22 83 56 55 32 56

Using Quick Sort.

The median is: 55

22 comparison

0 seconds.

**Modified Quick Sort**

This algorithm is similar to quick sort since it divides the original list into two lists according to pivot is less or greater than items. As parameter, L (list) and n (length of L divided two) is passed in method, so any list that is smaller than n is impossible to have the median so it will do the recursive with larger list and n will be subtracted the length of the smaller list -1. If length of smaller list is greater than n, parameter for the recursive call is smaller list and n. whenever the length of list is same as n, it will return p which is the median.

Enter the length of the list.5

List: 14 20 87 56 30

Using Quick Sort.

The median is: 30

10 comparison

0 seconds.

Enter the length of the list.10

List: 87 4 34 26 61 97 19 11 66 66

Using Quick Sort.

The median is: 61

28 comparison

0 seconds.

**Runtime comparison**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **BS** | **MS** | **QS** | **MQS** |
| 5 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |
| 50 | 0 | 0 | 0 | 0 |
| 100 | 20 | 15 | 14 | 0 |
| 1000 | 801 | 523 | 88 | 31 |
| 5000 | 23408 | 12546 | 965 | 16 |
| 10000 | 98780121 | 55717 | 3171 | 266 |

**Comparing the number of comparison**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **n** | **BS** | **MS** | **QS** | **MQS** |
| 5 | 12 | 7 | 7 | 10 |
| 10 | 45 | 22 | 22 | 28 |
| 50 | 2009 | 215 | 200 | 121 |
| 100 | 9900 | 542 | 600 | 208 |
| 5000 | 24075154 | 55211 | 159651 | 11782 |
| 10000 | 96350364 | 120284 | 585777 | 43580 |

"""""""""""""""""""""

Lab2 - find median

2/22/2019 - Ken M. Amamori

CS2302 MW 10:30 - 11:50

"""""""""""""""""""""

from random import random

import copy

import time

""""""""""""""""""""""""""""""""""""

#List Functions

class List(object):

# Constructor

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

self.head = None

self.tail = None

class Node(object):

# Constructor

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

self.item = item

self.next = next

""""""""""""""""""""""""""""""""""""

""""""""""""""""""""""""""""""""""""

#return a copy of L

def Copy(L):

C = List()

temp = L.head

while temp is not None: #iterate every node

Append(C, temp.item) #append each item into C

temp = temp.next

return C

#return the item at n of L

def ElementAt(L, n):

if getLength(L)<n: #base case

return null

else:

temp = L.head

for i in range(n): #iterate for n times

temp = temp.next

return temp.item #return the item at n

#dicide which algorithm to use

def Median(L):

C = Copy(L) #get the copy of L

#print("Using bubble Sort.\n\tThe median is: ", BubbleSort(C)) #get the median using Bubble Sort

#print("Using Merge Sort.\n\tThe median is: ", ElementAt(MergeSort(C), getLength(C)//2)) #get the median using Merge Sort`

#print("Using Quick Sort.\n\tThe median is: ", ElementAt(QuickSort(C), getLength(C)//2)) #get the median using quick sort

print("Using Modified Quick Sort.\n\tThe median is: ", MQS(C, getLength(C)//2)) # get median using modified QS

return

#return the median after sorting the list in ascending order using modified quick sort

def MQS(L, n):

global count

p = L.head.item #pivot

SmallL = List() #list for smaller numbers

LargeL = List() #list for larger numbers

temp = L.head.next

while temp is not None: #go through each node

if temp.item < p: #smaller than pivot

count+=1

Append(SmallL, temp.item)

else: #equal or greater than pivot

count+=1

Append(LargeL, temp.item)

temp = temp.next

if getLength(SmallL)<n: #median not in smallL

count+=1

return MQS(LargeL, n-getLength(SmallL)-1) #search on LargeL and get right the index for median

elif getLength(SmallL)>n: #median not in LargeL

count+=1

return MQS(SmallL, n) #search on SmallL and get the right index for median

else:

return p #median

#return the sorted list

def QuickSort(L):

global count

if getLength(L)<=1: #base case

return L

p = L.head.item #pivot

SmallL = List() #list for smaller numbers

LargeL = List() #list for larger numbers

temp = L.head.next #get the next node

while temp is not None: #iterate every node

if temp.item < p: #smaller than pivot

count+=1

Append(SmallL, temp.item)

else: #greater or equal to pivot

count+=1

Append(LargeL, temp.item)

temp = temp.next

SL = List() #Smaller numbers List

SL = QuickSort(SmallL) #recursive call

LL = List() #larger numbers list

LL = QuickSort(LargeL) #recursive call

newL = List() #new list which will be combined

if IsEmpty(SL): #if pivot is the smallest number

Append(newL, p)

newL.head.next = LL.head

newL.tail = LL.tail

return newL

elif IsEmpty(LL): #if pivot is the largest number

Append(SL, p)

return SL

else: #pivot in middle

Append(SL, p)

SL.tail.next = LL.head

SL.tail = LL.tail

return SL

#return the length of L

def getLength(L):

temp = L.head

c = 0 #counter

while temp is not None: #iterate every node

c+=1

temp = temp.next

return c #return the length

#merge sort: sort the list in ascending order and return the list

def MergeSort(L):

global count

len = getLength(L) #get the length of L

if len <= 1: #base case

return L

leftL = List() #create a list for left side

rightL = List() #create a list for right side

temp = Copy(L) #get the copy of L

c=0

#separete the list in two

for i in range(len//2): #itetrate the list up to len/2

Append(leftL, temp.head.item) #append current item into leftL

temp.head = temp.head.next

c+=1

while c < len: #iterate the rest of the list

Append(rightL, temp.head.item) #append current item into rightL

temp.head = temp.head.next

c+=1

LL = List() #LeftList

LL = MergeSort(leftL) #recursive call for the left side of the list

RL = List() #RightList

RL = MergeSort(rightL) #recursive call for right side

TL = List() #Total List

while getLength(TL)!=len: #keep iterating until TL has all items

if IsEmpty(RL): #Right list empty

Append(TL, LL.head.item)

LL.head = LL.head.next

elif IsEmpty(LL): #Right list empty

Append(TL, RL.head.item)

RL.head = RL.head.next

elif RL.head.item < LL.head.item: #when RL item is smaller than LL item

count+=1

Append(TL, RL.head.item)

RL.head = RL.head.next

else: #when LL item is smaller than RL item

count+=1

Append(TL, LL.head.item)

LL.head = LL.head.next

return TL #return the merged list

#sort the list in ascending order using bubble sort and return the median

def BubbleSort(L):

c=True

global count

while c: #checks if there was any swap in previus iteration

temp = L.head

while temp.next is not None: #traverse every nodes except the last one

count = count+1

if temp.item > temp.next.item: #true then swap

#count = count+1

tnum = temp.item

temp.item = temp.next.item

temp.next.item = tnum

c=False

temp = temp.next #get the next node

c = not c #get the inverse to see if there is any need to continue checking the list

return ElementAt(L, getLength(L)//2) #return the median

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 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

# generate a list of length n with random numbers

def createList(L, n):

for i in range(n):

Append(L, int(random()\*101)) #append a number between 0 and 100 into L

count = 0 #global variable in order to keep track of comparison

n = int(input("Enter the length of the list.")) #input from the user

L = List() #creating a list

createList(L, n) #create a list of length n

print("List: \t", end='')

Print(L) #print the original list

start = int(time.time()\*1000) #starting time

Median(L) #get the median of the list (L)

end = int(time.time()\*1000) #ending time

print(count, "comparison") #num of total comparison

print(end-start , "seconds.") #resutl time

**Conclusion**

As conclusion, I found that there are many ways to find the median by sorting a list and bubble sort takes long time and compares more than any algorithm. At the same time, it was an effective assignment since I learned and get to practice the basic way to use linked list in python since I have never used before. Also, I understand the algorithm of how to implement merge, quick and bubble sort.

I certify that this project is entirely my own work. I wrote, debugged, and tested the code begin 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.