Lab 2 had us sort a randomly generated list, sort it and then return the median value of the list, the middle value. We had to use 3 different algorithms to sort the list, and a 4th algorithm that strictly finds the median value of the list without sorting the list.

The first algorithm was bubble sort, the way I approached this solution was to compare each value to the next one in the list and swapping them if the left one is greater than the next in the list, thus sorting the list. I would be repeating this process until the list is gone through entirely without ever doing a swap which I controlled with a Boolean variable called done, to signify when the sorting was done. This process was fairly straight forward and easy to code, I used two loops, one while loop ran until done was not false and the second traversed the list and compared values.

The next algorithm I attempted was quick sort, this was quite a challenge but in the end was much easier in hindsight. To begin with my code, I created a pivot which would be the head of the list. This is done under the assumption it is an unsorted list where the majority of values within it are not in order. The next part would be the create two empty lists, L1 and L2 which would hold items less than and greater than the pivot respectively. I began to go throughout the list appending the items into the appropriate list, L1 or L2, and would recursively call the function to sort the lists L1 and L2 as well. Trusting in the call to sort the lists I would have to re connect my lists together which I would do by appending the pivot to the end of L1. This gave me trouble in testing, but I will go in further detail later during my experiments. After connecting the pivot, I would make the tail of L1.tail.next point the L2.head. From there the originally List’s head would be L1.head and the tail would be L2’s tail as they two lists are now connected.

The final sorting algorithm is Merge sort. Breaking down the problem it was similar to quick sort where I saved one half of the list in a list called L1 and the other half in another called L2. However, the similarities end there, once I finish the splitting of the lists, I called the main function again to sort both L1 and L2. From there I empty out the main list L with a function to empty the list, then call a function to merge L1 and L2. The merge function compares the items in L1 and L2 and appending the smaller item back into the original list until one list has been completely appended, once that is done it appends the rest of the remaining list since nothing else could be compared.

The algorithm to find the median value without sorting takes ideas from the quick sort algorithm. I did also add an argument called medpos which contains where the median position would be in the list if the list was sorted, the length of the list divided by 2. I started the function by using a pivot and getting all values less than and greater than the pivot and putting them into a list. The next part of this function sees if the length of L1 is larger than the median position, which would mean it’s in the left half of the list, if so we would call the function using L1 as the list and keeping medpos the same. Otherwise we check if the length of L1 is the same as medpos, if this is true then the median value would be the pivot which has the same number of items before as median position, lastly if neither prior case is true we would search the right half of the list by calling the function with L2 and subtracting the length of L1 and an extra 1, to account for the pivot. Taking L1.length +1 away from median position is to adjust where the median position would be in the list L2.

Whilst experimenting for the correct outputs for bubble sort I would generate a randomly sized array of size 1 to 50 with random numbers from 0 to 100 to ensure I would not generate a case specific solution for the algorithms and ensure they would be robust in outputs consistently sorting them.

Bubble sort gave me little issues while testing, which makes sense at it’s the simplest of the algorithms and is extremely straight forward. Bubble sort has a big O run time(High end case of runtimes) of n^2 where n is the number of items in the list. While seeing if this is true in practice, I ran each algorithm ten times recording number of comparisons made to see if this is consistent and is true. Bubble sort was quite often near n^2 in the number of algorithms made and was by far the slowest at sorting the lists of all algorithms.

With few cases having less comparisons made than others like 31 and 33 being similar to 29, this is due to the lists already having sorted items within the list, unlike case 8 which made 42 comparisons which is near 8^2(64).

Quick sort has a range of big o runtimes, in the average case it should run at n logn , but in the worst case scenario it has a big o of n^2, and this inconsistency it can sort much slower than merge sort.

This chart reflects that well as there are times like the case with 35 items running much faster than the case containing 33 items as 33 must have had a lot more sorted items already within it than 35 did which would create this case. After crunching the number of comparisons made and the length of the lists I found that quick sort was also fairly close to n log n times as well, except the few random cases like 33 and 43 which had more than normal comparisons. While testing my code I did run into the issue of a none pointer exception which I solved by creating cases for if L1 was none to just prepend the pivot to L2 and make the list L’s head and tail that of L2’s head and tail which would solve my issue of getting the None pointer exception.

Merge sort had the most consistent number of comparsions throught each run on average doing near n logn work each time reflecting the on paper run time of O(n). This gave my graph a nice natural slope except for case 33 which was a odd case for all non bbuble sort methods, likely due to it being naturally sorted already making the program do unnesecary work. While testing the code for merge sort I found that in order to properly have it work I did have to empty the list L otherwise I ran into odd issues, hence the creation of the MakeNewL function which just resets all values as if the list was just made. From there it was failry easy to impliment the merging process of the code which was basic comparisons then adding the rest of the list after the comparisons were done.

The final algorithm was tricky at first and really only gave me issues on the cases of what to return, the pivot, the search of L1, or the search in L2. On paper it should have a runtime of N going one by one through the list, but on average it ran anywhere from N to log n runtime in my cases of running the program.

I had one case of size 6, which doesn’t show on my graphs as the numbers made are small in comparison to the rest where the number of comparisons made here was 5 which is just shy of n runtime, yet I had other runtimes like 46 having 86 comparisons made, just a bit shy of 2N comparisons which still maintains the N runtime, and having other cases like 33 and 43 which made nearing n log n comparisons. However, on average the modified quicksort algorithm that finds only the median makes significantly less comparisons than the other algorithms by quite the large margin.

This lab tought me how there are so many ways to do one things, like sorting an array and finding the medain value, and modifying known code to fit the desire, like modifying quick sort to find the median value in roughly O(n) time. It also brought to light how awful bubble sorting is compared to other much more efficient algorithms like merge sort and why we should try to make our programs efficient as well to redue the amount of unnecessary actions.

Appendix ---

# -\*- coding: utf-8 -\*-

"""

Course 2302(Data Structures)

Instructor: Olac Fuentes

Teaching Assistant: Anithdita Nath

Lab 2

Last Edited on 2.22.2019

@author: Seth

"""

#Node Functions

import random

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

self.Len = 0

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

L.Len +=1

else:

L.tail.next = Node(x)

L.tail = L.tail.next

L.Len +=1

def Prepend(L,x):

##inserts x at begingin of list L

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

L.Len+=1

else:

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

L.Len +=1

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 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

L.Len -=1

else:

L.head = L.head.next

L.Len\_=1

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

L.Len-=1

else:

temp.next = temp.next.next

L.Len-=1

def BubbleS(L):

global numComp

if IsEmpty(L):

return None

else:

t = L.head

done = False

while done != True:#runs loop until boolean variable done stays true through every comparison

done = True

t = L.head

while t.next is not None:

if t.item > t.next.item:#if current item is less than next then swap

nextItem = t.next.item

t.next.item = t.item

t.item = nextItem

done = False#if swap is made then the list is not sorted yet

numComp +=1

t = t.next

def QuickS(L):

global numComp

if L.Len > 1:

pivot = L.head.item

L1 = List()

L2 = List()

t = L.head.next#removes pivot from comparisons

while t != None:

if t.item < pivot:#sorting of items into lists less than or greater than pivot

Append(L1, t.item)

else:

Append(L2, t.item)

numComp+=1

t = t.next

QuickS(L1)#sorts list less than pivot

QuickS(L2)#sorts list greater than pivot

if IsEmpty(L1):

Append(L1, pivot)

else:

Prepend(L2, pivot)

if IsEmpty(L1):

L.head = L2.head

L.tail = L2.tail

else:

L1.tail.next = L2.head#connects the left and right list

L.head = L1.head

L.tail = L2.tail#This line and previous connect the sorted halves to the original list

def MergeS(L):

if L.Len > 1:

L1 = List()

L2 = List()

t = L.head

for i in range(L.Len //2):

Append(L1, t.item)

t=t.next

while t != None:

Append(L2, t.item)

t = t.next

#previous 2 loops separates L1 and L2 into halves of L

MergeS(L1)

MergeS(L2)

#^ sorts split lists

makeNewL(L)#empties L to have L1 and L2 be appending into L

merge(L, L1, L2)

def merge(L,L1, L2):

global numComp

#Appends sorted Lists into L

t1 = L1.head

t2 = L2.head

while t1 != None and t2 != None:

if t1.item < t2.item:#appends T1 items

Append(L, t1.item)

t1 = t1.next

else:

Append(L, t2.item)#appends T2 items

t2 = t2.next

numComp+=1

if t2 is None:

while t1 != None:

Append(L, t1.item)

t1 = t1.next

if t1 is None:

while t2 != None:

Append(L, t2.item)

t2 = t2.next

# ^ ^ for when one list t finishes and the other has remaining elements

def makeNewL(L):

#empties all values in L so that when you merge you reuse the same list

L.head = None

L.tail = None

L.Len = 0

def ModdedQuickS(L, MedPos):

global numComp

if L.Len <= 1:

return L.head.item

pivot = L.head.item

L1 = List()

L2 = List()

t = L.head.next#removes pivot from comparisons

while t != None:

if t.item < pivot:#sorting of items into lists less than or greater than pivot

Append(L1, t.item)

else:

Append(L2, t.item)

numComp +=1

t = t.next

if L1.Len > MedPos :

#if we know the median number is in the left list

return ModdedQuickS(L1, MedPos)

elif(L1.Len == 0 and MedPos == 0):#if the pivot is in the median position(In cases of if the Med position is 0 and L1 has nothing in it)

return pivot

elif(L1.Len == MedPos):#if the median position so happens to be the pivot

return pivot

else:

return ModdedQuickS(L2, MedPos - L1.Len - 1)

def Copy(L):

#copies list L into a new list with the same values

copy = List()

t = L.head

while t != None:

Append(copy,t.item)

t = t.next

return copy

def Median(L):

C = Copy(L)

BubbleS(C)

t = C.head

for i in range(C.Len//2):

t = t.next

return t.item

def Median2(L):

C = Copy(L)

MergeS(C)

t = C.head

for i in range(C.Len//2):

t = t.next

return t.item

def Median3(L):

C = Copy(L)

QuickS(C)

t = C.head

for i in range(C.Len//2):

t = t.next

return t.item

def Median4(L):

C = Copy(L)

print(ModdedQuickS(C, C.Len//2))

myL = List()#creates random List of size 0 -50 with integers from 0-100

for i in range(random.randrange(50)):

Prepend(myL, random.randrange(100))

print('Original list: ', end = '')

Print(myL)#shows original list, unsorted

numComp = 0

print('\n',myL.Len)

print('Sorted by bubble sort, median is: ', end = ' ')

print(Median(myL), ' number comparisons: ', numComp)

numComp = 0

print('Sorted by merge sort, median is: ', end = ' ')

print(Median2(myL), ' number comparisons: ', numComp)

numComp = 0

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

print(Median3(myL), ' number comparisons: ', numComp)

numComp = 0

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

Median4(myL)

print('number comparisons: ', numComp)

“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 provided inappropriate assistance to any student in the class.”

- Seth Abel Flores