Ericka Najera

CS2302

25 February 2019

Lab 2

Lab 2 was about making sorting modules for a singly linked list class with no native python list. The modules were bubble sort, merge sort, quicksort, an alternate quicksort with only one recursive call. To generate a equal case for each module we have to use a random list of numbers.

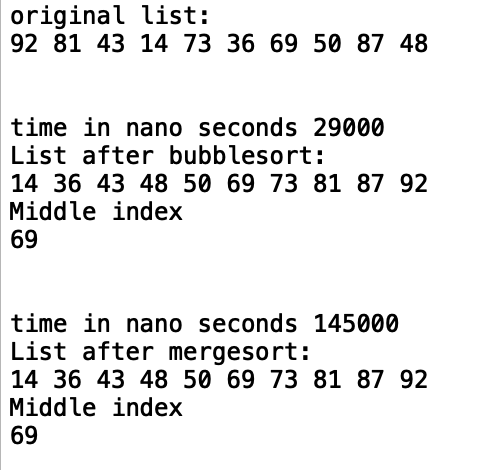
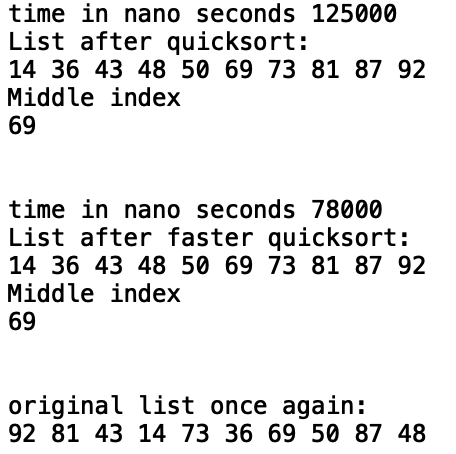
To start of my program, I had to build the list of random numbers. I took a for loop to run for n times which was the number input to decide how long the list would be. I used the format of the append method to create the list. I used the math random to make sure the list was completely random from numbers 1-100. If the list would be empty, then I would create the head. If it wasn’t empty, I would add a node to the tail of the list. I implemented the append, prepend, print, getmid, isempty, getlength modules to help my program work more efficiently.

As for the module for bubble sort I knew it was a linear module so it would be a for loop or while loop. Since I was working with singly linked lists and recursion my base case was to check if the list was empty. I knew I need to go through the list until no items where switched. So I would have to create another while loop and a Boolean variable. After the list got sorted, I had to return the item in the middle using the getmid module.

Quicksort was a two-call recursion so I would have to create two lists and append to either list depending if they were bigger or smaller than the pivot. In my case to prevent any confusion I made the pivot the head of the list every time. Before each new call I would append the pivot the left list to keep the lists sorted. After all the lists were sorted separately, I would have to join them together to the initial list. I took the properties of the list class and used the head and tail. The head of the left list became the head and the tail of that list was saved to not lose the ending of the left list. Then the next node would be where the right list would be attached then the tail of the right list would become the official tail.

As for the merge sort I thought it would be the same as quicksort but it keeps either not changing the list at all of it would give me an error. Initially I did it the same way for quick sort but having a module to merge all the sorted lists, but it didn’t work out in the way I set up the recursion module. The way to approach this module is to divide the list first into sub lists until you can’t divide anymore. After that you merge each smallest list while putting them in order, until you get the last lists put together. I approached it setting three different modules one to separate the list into two lists every time by setting a for loop with the middle index. All items to the left of the middle would go on one list and the remaining would go the right one; returning two different lists at the end. Then one to merge the sorted lists which in this case would sort from the lists of length 1 up to the bigger list. I did a while loop to run while both left and right lists had at least one item. Then I did another two while loops to catch any remaining item in either of them. Considering there would be only one item remaining in either case I just sent the remaining to the list. Then the module would return a new list. The final module was the actual merge sort recursion module. I create two lists from sending the list to the separating module. From there a called the merge sort method for each list so it will recursively separate until there is only one in each list. After that we can recursively return the merged list that will eventually return the final sorted list. The only thing I couldn’t get to work was returning the middle number of the list so I just returned the list and called the getmid module manually.

For the quick sort module that need to have only one recursive call, I had the initial idea of creating only one list and prepend items smaller than the pivot and appending items bigger than the pivot. After attempting that method, it didn’t work because the list would eventually return unsorted or endless recursion. Then I did the same as the quicksort module I had done and since the instructions said to forget the middle list then I would only call one side at a time. I decided to do the if statement to decide which list was bigger and eventually call recursion on only one list at a time. At first, I did the assembly of the list in each if statement and it didn’t work but then I moved it to after the if statements and it magically worked.

 To run my modules, I created a list but since if I created a list, I would generate a random list, so I decided to make a copy method to make a copy of the list and not lose the original list.

I also implemented the time.time\_ns() before and after the calling the methods to get the time for the running times to plot my graph. The running the times of my programs based on my code is:

Bubblesort: T(n) = T(n-1)+n = O(n^2) , Quick Sort : T(n) = 2T(n/2)+n= O(nlogn) , Merge Sort: T(n)= 2T(n/2) +2n = O(nlogn) Faster Quick Sort: T(n) = T(n/2)+n= O(logn). According to the real running times calculated, my programs are correct in their running times.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 10 | 100 | 500 | 1000 | 1500 |
| Bubblesort | 25000 | 2815000 | 77037000 | 316913000 | 718110000 |
| Mergesort | 125000 | 2628000 | 45361000 | 161955000 | 375938000 |
| quicksort | 80000 | 2245000 | 31432000 | 188916000 | 274303000 |
| Fast Quicksort | 64000 | 2158000 | 22452000 | 180787000 | 264274000 |

Appendix:

﻿# Ericka Najera Lab 2 MW 10:30-11:50

import random

import time;

class Node(object):

# Constructor

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

self.item = item

self.next = next

class List(object):

# Constructor

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

self.head = None

self.tail = None

def IsEmpty(L):

#checks if the list is empty and returns a boolean

if L.head == None:

return True

return False

def buildlist(L,n):

#builds a list base on n (number of nodes)

for i in range(n):

#each item will be a randome number

num = random.randint(1,101)

#if the list is empty it becomes the head

if IsEmpty(L):

L.head = Node(num)

L.tail = L.head

#else the next item after the last item become a new node

else:

L.tail.next = Node(num)

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 getlength(L):

#returns length of a list

#saves to now loose the head

temp = L.head

counter =0

#each loop increments counter is a node is not null

while temp is not None:

counter +=1

temp = temp.next

return counter

def getmid(L):

#returns the middle of the list

temp = L.head

counter = 0

#the counter will go till the middle node and return it

while counter!= (getlength(L)//2):

temp = temp.next

counter+=1

return temp.item

def copylist(L,newlist):

#makes a new copy of the list

temp = L.head

newlist.head = Node(temp.item)

newlist.tail = newlist.head

temp=temp.next

for i in range(getlength(L)-1):

newlist.tail.next = Node(temp.item)

newlist.tail = newlist.tail.next

temp=temp.next

def swapnum(prev,curr):

#swaps node with the next one

temp=prev.item

prev.item = curr.item

curr.item = temp

def compare(L):

#compares a node to the next one

if L.item > L.next.item:

return True

else:

return False

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 bubblesort(L):

#if the list is less than 1 it will return the head

if getlength(L)<1:

return L.head.item

else:

sorted = False

temp = L.head

#in each swap sorted will become false and until the list is sorted

while sorted is False:

sorted = True

current=temp

#the loop goes through the list and it will check and swap if necessary

while current is not None and current.next is not None:

if compare(current):

swapnum(current,current.next)

sorted = False

current = current.next

#returns the middle node of the sorted list

midnum = getmid(L)

return midnum

def quicksort(L):

#base case

if getlength(L)>1:

#the pivot will always be the first node

pivot = L.head.item

L1 = List()

L2 = List()

#the temp is created to not loose the head of L

temp = L.head.next

while temp is not None:

#if the item is smaller than the pivot it will append to left or bigger to the right

if temp.item<pivot:

Append(L1,temp.item)

else:

Append(L2,temp.item)

temp=temp.next

#append the pivot the left side

Append(L1,pivot)

#call recursion for each side

quicksort(L1)

quicksort(L2)

#the head will be the head of the left and the tail the last item of the left

L.head = L1.head

L.tail = L1.tail

#attach the head of right to the last node of left

L.tail.next =L2.head

#the tail of the right will become the new tail

L.tail = L2.tail

#get the middle of the new list and return

midnum = getmid(L)

return midnum

def prepend(L,x):

#inserts node at begginning of list

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

new\_node = Node(x)

new\_node.next = L.head

L.head = new\_node

def fastquicksort(L):

#quicksort with only one recursion happening at a time

#same first steps as normal quicksort

if getlength(L)>1:

pivot = L.head.item

L1 = List()

L2 = List()

temp = L.head.next

while temp is not None:

if temp.item<pivot:

Append(L1,temp.item)

else:

Append(L2,temp.item)

temp=temp.next

Append(L1,pivot)

#if the left side is bigger than the left it will forget the other list

if getlength(L1)>getlength(L2):

quicksort(L1)

#if the right is bigger send that list instead

else:

quicksort(L2)

#this part is the same as the quick sort attachment of the lists

L.head = L1.head

L.tail = L1.tail

L.tail.next =L2.head

L.tail = L2.tail

midnum = getmid(L)

return midnum

def seperate(L):

#splits the list into two and returns two new lists

if L.head is None or L.head.next is None:

left = L

right = None

else:

left = List()

right= List()

temp = L.head

#go through the list using the length

for i in range(getlength(L)):

#if the index is less than the middle all nodes will be on the left

if i < (getlength(L)//2):

Append(left,temp.item)

#after middle all nodes go the right

else:

Append(right,temp.item)

temp = temp.next

return left,right

def merge(a,b):

result = List()

a= a.head

b=b.head

#go through both lists

while a is not None or b is not None:

#if the one is none then append the remaining of the other list

if a is None:

Append(result,b.item)

b=b.next

elif b is None:

Append(result,a.item)

a=a.next

else:

#if both exist check the item

#append the smaller node first

if a.item < b.item:

Append(result,a.item)

a = a.next

else:

Append(result,b.item)

b = b.next

return result

def mergesort(L):

#base case

if L is None or L.head.next is None:

return L

#split the list into two

left, right = seperate(L)

#save the recursion for each side

left = mergesort(left)

right = mergesort(right)

#returns the list after they get merged together

return merge(left, right)

print("\n")

L = List()

buildlist(L,10)

print('original list:')

Print(L)

print("\n")

BS=List()

copylist(L,BS)

start = time.time\_ns()

bubblesort(BS)

end = time.time\_ns()

print('time in nano seconds',end-start)

print('List after bubblesort:')

Print(BS)

print('Middle index')

print(bubblesort(BS))

print("\n")

MS=List()

copylist(L,MS)

start = time.time\_ns()

MS=mergesort(MS)

end = time.time\_ns()

print('time in nano seconds',end-start)

print('List after mergesort:')

Print(MS)

print('Middle index')

print(getmid(MS))

print("\n")

QS = List()

copylist(L,QS)

start = time.time\_ns()

quicksort(QS)

end = time.time\_ns()

print('time in nano seconds',end-start)

print('List after quicksort:')

Print(QS)

print('Middle index')

print(quicksort(QS))

print("\n")

FS=List()

copylist(L,FS)

start = time.time\_ns()

fastquicksort(FS)

end = time.time\_ns()

print('time in nano seconds',end-start)

print('List after faster quicksort:')

Print(FS)

print('Middle index')

print(fastquicksort(FS))

print("\n")

print('original list once again:')

Print(L)

I Ericka Najera certify that this project is entirely my own work. I wrote, debugged, and teste 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.

