In this lab I am supposed to program three sorting methods which are bubble sort, merge sort, and quick sort along with a modified recursive quicksort. With these sorting methods I’m supposed to sort a linked list from the order of least to greatest. Also, we have to return the middle value of each sorted listed.

**Bubble sort**

In order to be able to create a bubble sorting program that can sort a linked list in python; the first thing I had to do was to understand how bubble sort works.

***How bubble sort works***

Bubble sort is a sorting method that compares the values of a list that are next to each other; for instance, if my list had the values in a list as (2,1,5,4) then the Bubble sort algorithm would compare the first value to the one next to it until it reaches the end of the list and of the list is still not sorted then the algorithm will traverse through the list and see which values need to be changed in position. and in this example we can see 2 > 1 which would make the algorithm to switch the position of those two values making the list appear as (1,2,5,4). Then the algorithm keeps comparing so it compares 2 and 5 which is 2 < 5 which is correct since the list is in order from least to greatest still and no change is needed. The algorithm keeps going by comparing the value 5 and 4 which when compared is 5 > 4 which is incorrect so now the algorithm would switch the positions of 5 and 4 in order to make the list to change to (1,2,4,5) which the list is sorted. However, if the list was still not sorted then the algorithm would have to keep sorting the list until its completely sorted from least to greatest.

***Proposed Solution***

For the possible solutions to make this algorithm possible was to be able to find a way to compare two values and make the program to compare two values and decide if they need to be switched if one of the values that comes before them in order is greater than the value next to it. However, this program needs to use linked lists instead of a python native list due to accomplish the requirement of sorting a linked list with bubble sort.

***Implementation***

When creating the bubble sorting algorithm that will sort a linked list, I first needed to apply the linked list algorithms that were given in class in order to be able to use linked list and its functions. The first thing that I had to do was to be able to keep traversing through the linked list until every value in the list is sorted. In order to do that I needed to make a while loop that contains a value named “change” that is declared true; the value “change” would traverse the list until the value becomes false which means that no more changes in the sorting algorithm is possible. The next part of the program would be to follow a basic rule which is to never lose the head of the linked list, which is done by storing my head in a temporary variable which keeps my original list from being changed. After finding a way to not lose my head, I would then need to declare my “change” value as false which comes into play later on in the code. After that I need to traverse my linked list until the next value in my temporary list is None. After finding a way to traverse my temporary list, I had to make my algorithm to compare my current item in the list to the one next to it and if the current item was larger then I have to switch my values position in the list.

When finding out that if my current item is larger than the one next to it then I had to create a code that switches the position only between those two items. The way that the items where switched was to store my current item into another temporary variable to store the value for the meantime. Next I needed to affect my temporary linked list by declaring the current item to the one that was being compared to it in order to make the changed item into my current item. Now we need to use the current item that was stored in my temporary variable and assign it to the position that is next to out new current item which will now change the temporary list by switching both items. After switching then we declare the value “change” as true to show that there was a switch done and next we need to traverse to the next item on the list where we continue from the current item we had.

***Experimental results***

For this program I will be running a randomly generated linked list that will be used in order to test the bubble sort method. This random generated linked list will be 3,5,2,3,4 and will be sorted by the bubble sort algorithm. When tracing the method with the current linked list created we first declare the variable “change” as true which is used in the while loop to keep sorting until it becomes false. The we have to store the linked list in a temporary list in order to not lose our head or affect the original linked list. After making a temporary list we then need to traverse through the list by going into the while loop and keep going until the next value of the temporary list is None. Inside the while loop we compare the first value and the second value of the list which is 3 and 5, since these two values are from greatest to least we don’t change anything and the value “change” stays as false; then we have to move on to the new current item, which is compared to the one net to it where be begin to compare 5 and 2 which is going to get switched since 5 > 2 and change becomes true. Now after switching those values the list now looks like 3,2,5,3,4, next the program continues to compare the next element which is 5 and the one next to it which is 3 ;since 5 > 3 those values get switched making the list become 3,2,3,5,4 and change stays as true. Now we compare 5 and 4, but since 5 > 4 then we switch those values making the list as 3,2,3,4,5 and change stays as true; since we got to the end of the list we still have to keep traversing until there are no changes in our sort.

The second traversal compares 3 and 2, since 3 > 2 the code switches the positions of the items making the list 2,3,3,4,5 and change stays as true. As we can see right now the sorting is complete, but the algorithm still has to compare the values since we haven’t exited either loop until we reach the end of the temporary list and our “change” becomes false after no more changes are needed. After traversing we reach the end of our program exiting the while loop and making the change false since our linked list is sorted; our output of the sorted list is 2,3,3,4,5 and when we get the middle position by the method median it tells us that the middle element is 3 which is correct. Also, if we were to try this with an empty list, the bubble sort will still return an empty list and won’t sort anything. If we only had one value in the linked list then the program would only return that value since it can’t compare it to any other value.



Big O:

O(n^2)

**Merge Sort**

For this part of my program I have to creating an algorithm that can use merge sort to merge a list, but in order to know how the algorithm should be created I first need to understand how merge sort works.

***How Merge Sort works***

Merge sort is a sorting method that sorts a list by first dividing it by half until you have two split lists, but with the halves that we obtained still have to divide them by half or until all the values of that half are separated from each other. After all the values are split from each other, then the sorting algorithm starts to compare the values separated and sorts them from least to greatest. Whichever values were compared then the program must merge those two values if they switched positions or not. When they merge the make a new list and then the values of that list compare to the values of another list and they become sorted if they are not in order, which then they merge again. The lists must merge until there’s only one list left however in that final list the two opposite lists must compare each value in their own list to the other one in order to make a sorted list before they merge into one list at the end.

***Proposed Solution***

For this program, I knew that I first needed to split my list by half which can be considered as the left and right list, I have to do this in order to follow the rules of merge sort. However, after splitting my list, I would need to keep splitting my list until it reaches the limit of being unable to split my list anymore. After the entire list becomes split, then I would have to compare my two values that are adjacent to each other and if they are not in order then I would switch the positions they are in. Even if they switch or not I would have to merge the two values into a list, when we return that list then I’m going to have another list that either went through the same process or not, but I would have to compare the values of my two merged lists and put them in order and then merge it into another sorted list. The comparisons and merging would only stop after the entire list is sorted and when we only have one merged sorted list remain.

***Implementation***

In order to create my merge sort algorithm then I would need to have 3 method, one to pass the list and call the functions to split and merge the sorted list. The second method will be to divide the list by half until no more splits are possible. The final method is to compare the values in the list and then merge them in to a new sorted list.

To begin with the first method, I had to make a base case to make sure that if the list is empty or if it only has 1 variable then the method would return the list we have since there is no need to sort it. If that base condition doesn’t apply to the list then we continue to create a variable called middle which would call the method split and splits the list. Since the middle variable is going into the method split, I would then have to create the split method before anything else in the merge sort algorithm. In my split method I created two variables one called fast which is storing the next node of the original list and the other variable is called slow which holds the list. The purpose of these two variables is that they will actually be our pointers that will get the middle position of the lists. Once the two variables have been declared I would need the pointers to go through a while loop which will allow the fast pointer to move two positions ahead and move the slow pointer by one. In the while loop I had to make the fast pointer iterate through the list until it reaches the end of the linked list. Inside the while loop the fast pointer will point at the next node in its list and then there will be an if statement that if fast is not None then the next following actions will be taken. In the if statement, slow is now pointing at the next node in its own linked list and fast points at the next node in the next node again. Until our fast variable reaches the end of the linked list we will be able to exit the list and return the slow variable from where it last pointed.

Back to the first method, after declaring the variable middle to the list we returned from the split method I had to make another variable called nextMiddle which will point at the value after the middle which will be used to get the right side of the list. After declaring the previous variable, I had to assign the node next to my middle value as None, so it gets rid of its remaining list. Now I created two variables that are called left and right which the purpose of them is to get the two split lists and pass them through the method that we’re in recursively. Since the lists that we have are being passed recursively then it would go to through the same process that I have explained and will continue to split those lists until its not possible anymore. However, I declared a value that is called sortedList which calls the method where it compares my values, sorts them in order and merges them by inserting them in a brand-new sorted list.

In my sorting method I have to create an empty variable called Result which its purpose is to store the new sorted linked list and return it at the end. After that, I have to make base cases which will check if my left list is empty then it returns the right side of the list only and vice versa for when the right side is empty. After checking that then I had to compare my left sorted linked list with the right sorted linked list and it was compared as if the value of the left side is less than or equal to the value at the right linked list the result will hold the item of the left list. Then the result value needs to store its next node and in order to do that it has to call the sorting method recursively and pass the next node of the left side and pass the right side of the list, but without making any changes to it. However, if the left link list item is greater than the value in the right link list then the Result will store the right link list item in its current Node and then call the sorting method recursively and pass an unaffected left link list and passes the next position of the right link list. This process will be passed until no more merging is possible and will return a sorted merge sort linked list .

***Experimental results***

To trace this code, we will use a random generated linked list that is not in order and the values for this list is 3,2,4,4,1. When I begin to use this linked list and pass it to the method called merge\_Sort the method will first check if my list is empty or if it only has one item in it and if this applies to my list then it would return the list and the code end right there, but in this case my list doesn’t meet that base case. After that there is the variable called middle which calls the split method and passes my list to it. In the split method the variable fast is pointing to my second value of the list which is 2 and the variable slow is pointing at the head of my list which is 3. Then my list has to go through a while loop where if fast is not None then it would iterate through the list till it becomes None. In the while loop my fast variable now moves forward another node which is 4. After the, there is an if statement and if my fast value is not none then my slow value would point at the next position which is 2 and my fast value moves to next position again which is 4. However, since fast is still not None, the while loop conditions are still not broken meaning that another increase in to fast pointer would happen meaning my fast is now 1. However, since the fast pointer is now at my last node then it would go through one final if statement and increase the slow pointer to point at the node that hold 4 and the fast pointer now points at None. Since fast is now equal to None then we are able to break from the while loop and return the slow pointer to our merge\_Sort method.

Back to our merge\_Sort method, middle will now hold the position where the slow pointer left off; After that, the value nextMiddle is declared to the next position after the middle and the position after our middle value is now declared as None as a result splitting the list into 3,2,4 and 4,1. Now we have the variables left and right which will pass the two list recursively in to the merge\_Sort method in order to split all the variables again; also left holds 3,2,4 and right holds 4,1. When passing the lists recursively, I will be left with the left side splitting the entire left side as (3), (2), (4), and the right side splits it into (4),(1) after the entire recursion takes place. After the splitting of the entire list then the value sorted List will call the method sorted\_Merge and pass my left and right list. In the sorted\_Merge method the variable Result is None and after that there are base cases that if the left Side is None then we return the right list and the second base case if is if the right list is empty then we return the left side of the list. After those base cases we compare the item of the left side and the right side and since the left side is now 2,3,4 and the right is 1,4 then we compare the first values. So, 2 > 1 which in this case we have to make the 1 be placed first into the list called result and then pass the left side with no positions change and with the right side now pointing at 4 which is passed recursively to the sorting method. So now we compare 2 and 4 and since 2 < 4 we place 2 in the next node of the result list making the list now as 1,2; now we have to move to the next position of the left list and keep the right same. Then we compare 3 and 4 and since 3<4 we insert 3 into the new list making it 1,2,3. Now we compare the final values which is 4 and 4 and since their both equal then they would be both inserted into the list however, at different recursions which make 1,2,3,4,4. Since the list is now sorted, this list is returned to the merge\_Sort method and it outputs the new list.



BigO:

O(n log(n))

**Quick Sort**

For this part of my program I have to creating an algorithm that can use quick sort to sort a linked list, but in order to know how the algorithm should be created I first need to understand how quick sort works.

***How Quick Sort works***

The way quick sort works is by using divide and conquer but also this sorting algorithm requires recursion, but its still possible to recreate with for or while loops however it would be really difficult to create. In quick sort the divide part is decided by the used where it wants to split the linked list however there is an element called the pivot which indicates the position that a linked list should be split. The pivot can be declared in any point of the linked list which it can be in the head of the list, at the tail of the list, or randomly in the list. After knowing where to divide the conquer part of the quick sort comes to play which is when all the elements to the left of the pivot must be less than the pivot and the ones of the right of the pivot must be greater than the pivot. If those conditions don’t meet then the values that are in the incorrect position will be moved either left or right of the pivot. After its been moved then the quick sort algorithm will sort the values of the left of the pivot and the right of the pivot.

***Proposed Solution***

Since I now know how quick sort works I can tell that quick sort fits linked lists better than any other sorting method. I know that I will need a pivot that will be dividing the linked list at a certain position, but however I would prefer to use the pivot as the head of my linked list. Also, I know that I will need to find a way to store the values greater than my pivot on the right side of the list and anything less than the pivot on the left side of the list. After comparing my pivot with the values of my list, then I will need to sort the values in both the left and right side from my linked list and sort them from least to greatest. After sorting once I would need to call my method recursively to keep sorting my list. After sorting I would need to combine my new lists together.

***Implementation***

To begin making my program I first made a base case that checks if my list is empty then I return the list. After that I made two variables to store my left and ride link list values of the pivot which will be used later in my code. Once that was done I had to make sure my pivot was pointing at the head of my linked list which this depends on preference. After declaring the position of my pivot, I created a temp variable that would store the next value after the head. Now since everything is declared, I now had to traverse my linked list until my temp became None but the way I traversed was to use a while loop. Inside my while loop I begin to compare my pivot to the values of my temporary list and if the pivot is greater than my temporary item, then that item would use the append method from the linked code provided in class; What the append method does is that it adds values at the end of a linked list. However, when using the append method I have to declare which side I want to append my values at, for instance if my pivot is greater than my item then I append the item into the left linked list and if the pivot is less than the item I would append my item in the right side of the list. After appending I would need to keep traversing through my linked list by moving on to the next node until it reaches None. Once my list reaches the end I would declare two variables called newLeft and newRight which will call my method recursively and pass the left and right linked lists. When the method is called my left side and my right side will both be sorted until it reaches the end of both lists and have been sorted. Now when both the lists are sorted I need to append my left to my pivot to make sure that they are both connected and finally I use the Concate method to combine my sorted left and right lists.

***Experimental results***

To test this method, I will use the linked list with the values 5,2,1,4,4 to trace my quick sort method. To begin tracing the algorithm first has to check if my list is empty due to the base case that my method has and since its not empty this doesn’t apply to the list. Continuing through the code I see that there are my 2 empty left and right linked lists to store my values later on and I see that the pivot is declared to the list of my head which means that my pivot is the value 5; after learning where my pivot is held I see that a temporary linked list is created to hold everything after my head in the linked list.

Now in the while loop the code will keep traversing through my temporary list until it becomes None. In the loop the program compares my pivot to my item in my temporary list, so since 5 is greater than 2, the value 2 will then be stored in the left side of the list and then the linked list keeps traversing. The second traversal compares 5 and 1 and since 5 is greater than one, then my value 1 gets appended to the left side of the list making the left side of the list 2,1. Next we compare 5 and 4 and since 5 is greater than 4 then it gets appended on the left side of the list making it 2,1,4; and the same thing occurs with the final four of my list and creates a finished linked list of 2,1,4,4. Since we reach the end of the linked list we break out from the while loop.

Now with my left side of the list created I need to use the newLeft list to call the method with recursion while passing my left linked list. Now since the method is called the same process occurs where the pivot is our head which is 2 and compares the rest of the list. The comparison 2 > 1, makes 1 be appended to the left side of the list since its less. Next we compare 2>4 and since 4 is greater we append it to the right side of the list which gets repeated for the next 4. Now we have the left side 1 and right side 4,4. Since these can’t be sorted anymore we have to connect the left side to the pivot by using append so now we have 1,2. To end this part we have to connect both left and right sides making 1,2,4,4. But now we need to connect this left list to the pivot that was originally used before which is 5. We use append to connect the left side of the list to the pivot making the sorted list as 1,2,4,4,5.

A screenshot of a cell phone

Description automatically generated

Big O: O(n^2)

**Modified Quick Sort**

For this program I have to implement a modified version of quicksort that makes a single recursive call instead of the two made by normal quicksort, processing only the sublist where the median is known to reside.

***Proposed Solution***

For this code I know that I could rely on the sorting method that was previously used in my previous quick sort and since I need to find the median of my sorted list I would need to use the median method to be able to obtain the element of the middle. Additionally, I am only able to recursively call the method once instead of using two before. So since I know I don’t have to return a sorted list and my goal is to sort the list and obtain the middle value.

***Implementation***

To implement my idea into the code I first need to make a second median method that would this problem. In my second median method I first made a copy of my linked list so I don’t affect it or lose it. After making my copy there is 1 value being declared to my modified quick sort method and the value is element returns the median value.

In my modified quick sort, I used the first half of my previous quick sort method to sort the linked list. However, the difference here is that I use if statements after, which if my n is equal to the size of the left linked list then I return the pivot. Then I have an else if statement that if my n is greater than the length of the list of my left side them I subtract the value n from the length of the left side and subtract it again by 1 to reduce the side of my list but then I return the method and recursively call it by passing the right side linked list and my n, which would find the big O notation of my method by using recurrence. Then my final else if is if my n is less than the length of the left list then it calls the method recursively to sort the method till my n is the size of the length in order to return the pivot.

***Experimental results***

Here I will be tracing the values 5,4,5,3,4 and this time the method will find the sorted median of my list. When going through the method the pivot will be the head and my temp will hold the rest of my list. In the method we compare 5 with 4 and since 5 is greater we have to place 4 on the left side of the list. Then we compare 5 and 5 and since their equal a 5 moves to the right side. Then 5 >3 which moves 3 to the left linked list and then it compares 5>4 and here we can see 4 gets moved to the left linked list; Now we have the left storing 4,3,4 and the right storing 5. Then since we reach the end of the method we need to keep sorting the lists till they are fully sorted but this time we need to keep doing it till n = the length of the left side in order to return the pivot which is our middle point in quick sort after being sorted. Since our length is not equal to n, but were less than n we return the method by passing the left side and begin to sort it . Since our values are 4,3,4 we know our pivot is 4 and temp is the rest of the list. When comparing we find that 4 is greater than 3 and moves it to the left side of the list and then when 4 is equals to 4 that value is moved to the right. Now that we split the 3 values we know that n is equal to our pivot since it’s the middle of our linked list in this occasion and since n is equal to the length of the left side then we return the pivot which is 4.

.A screenshot of a cell phone

Description automatically generated

Appendix

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

"""

Created on Thu Feb 21 12:26:02 2019

@author: Alexis

"""

import random

class Node(object):

# Constructor

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

self.item = item

self.next = next

#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 copy(L):

temp = L.head

list2 = Node (0)

list2.head = temp

while temp is not None:

temp = temp.next

list2.next = temp

return list2

def IsSorted(L):

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

print('sorted')

return True

temp = L.head

while temp is not None:

if temp.item > temp.next.item:

print('not sorted')

return False

return True

def Concate(L1,L2):

if IsEmpty(L1):

return L2

elif IsEmpty(L2):

return L1

L1.tail.next = L2.head

L1.tail = L2.tail

return L1

#------------------------------------------------------------------------------

#BUBBLE SORT ALGORITHM

def bubble\_Sort(L):

change = True

while change:

t = L.head

change = False

while t.next is not None:

if t.item > t.next.item:

temp=t.item

t.item = t.next.item

t.next.item=temp

change = True

t=t.next

#------------------------------------------------------------------------------

#MERGE SORT ALGORITHM

def merge\_Sort(L):

#n log n runtime

if L == None or L.next == None:

return L

middle = split(L)

nextMiddle = middle.next

middle.next = None

left = merge\_Sort(L)

right = merge\_Sort(nextMiddle)

sortedList= sorted\_Merge(left, right)

return sortedList

def split(L):

fast = L.next

slow = L

while fast != None:

fast = fast.next

if fast != None:

slow = slow.next

fast = fast.next

return slow

def sorted\_Merge(leftSide, rightSide):

Result = None

if leftSide == None:

return rightSide

if rightSide == None:

return leftSide

if leftSide.item <= rightSide.item:

Result = leftSide

Result.next = sorted\_Merge(leftSide.next, rightSide)

else:

Result = rightSide

Result.next = sorted\_Merge(leftSide, rightSide.next)

return Result

#------------------------------------------------------------------------------

#QUICK SORT ALGORITM

def quick\_Sort(L):

if IsEmpty(L) or L.next == None:

return L

leftSide = List()

rightSide = List()

piv = L.head.item

temp = L.head.next

while temp!=None:

if piv>temp.item:

Append(leftSide,temp.item)

temp = temp.next

else:

Append(rightSide,temp.item)

temp = temp.next

newLeft = quick\_Sort(leftSide)

newRight = quick\_Sort(rightSide)

Append(newLeft,piv)

return Concate(newLeft,newRight)

#------------------------------------------------------------------------------

#Modified Quick sort

def modified\_quickSort(L, n):

if IsEmpty(L):

return L

Print(L)

left\_Side = List()

right\_Side= List()

piv = L.head.item

temp = L.head.next

while temp is not None:

if piv>temp.item:

Append(left\_Side,temp.item)

temp = temp.next

else:

Append(right\_Side,temp.item)

temp = temp.next

#to get big O

#if my n is the pivot

if n== GetLength(left\_Side):

return piv

#if my n is in the left side of my list

elif n >= GetLength(left\_Side):

#n = n - GetLength(left\_Side)-1

return modified\_quickSort(right\_Side, n)

#if my n is in the right side of my list

elif n <= GetLength(left\_Side):

return modified\_quickSort(left\_Side,n)

#------------------------------------------------------------------------------

def GetLength(L):

temp = L.head

count = 0

while temp is not None:

count+=1

temp = temp.next

return count

def ElementAt(L, mid):

current = L.head

count = 0

while (current):

if (count == mid):

return current.item

count += 1

current = current.next

return 0;

def Median(L):

C = copy(L)

element= ElementAt(C,GetLength(C)//2)

print('Sorted Value of middle element is %d' %element)

def Median2(L):

C = copy(L)

element= modified\_quickSort(C,GetLength(C)//2)

print('Sorted Value of middle element is ', element)

return C

#------------------------------------------------------------------------------

# MAIN

L = List()

#print(IsEmpty(L))

for i in range(5):

t = random.randint(1,5)

Append(L,t)

#leftSide= List() tried to make them global variables for merge and quick sort

#rightSide= List()

#Print(L)

#UNCOMMENT TO TEST THE SORTINGS

#bubble\_Sort(L)

#L.head=merge\_Sort(L.head)

#L=quick\_Sort(L)

#Median(L)

#Print(L)

Median2(L)