Laila Romero

CS 2302 Data Structures

February 27, 2019

Lab 2

* Introduction

For this lab we were trying to implement several algorithms for finding the median of a list of integers, using objects of the List class, and compare their running times for various list lengths.

• Proposed Solution Design and Implementation

For bubble sort, I decided to break the sorting into two recursive methods of bubbleSort and bubbleSortPass. The first one, bubbleSort, sorted the head of the list and with each new pass and called upon bubbleSortPass to sort the remaining nodes after every pass. So bubbleSortPass would compare one element to the next and swap them if the first was larger than the second. It would continue this until it reached the end of the list and then start at the head again. Once the list was sorted it would stop.

For the mergeSort, I created 3 methods called divideLists, mergeSort and mergeLists. I first divided the lists into 2 parts using divideList. The lists then keep dividing until there are two elements. Then I sorted both lists separately with mergeSort with traversed through the divided lists and brought the elements back together after comparing them and swapping them into the correct order. Lastly, I called mergeLists to bring together the two sorted lists and do one more comparison of the elements in both lists and swap out the elements that were not in order. After this all the elements were sorted properly.

For quick sort, you need to choose a pivot element and then partition the other elements from there. I believe it is best to choose the end as the pivot and work from there. After choosing a pivot you traverse through the list and place all the smaller items on the left and the larger items on the right. You then choose the last element in the new lists to be your new pivots and repeat this process until you are left with one element. After this you regroup the newly sorted lists and you are done.

To determine the median of the list I took the length of the list and divided it by 2. I then return the value in the middle.

For randomizing the lists, I tried to set the List to a range using a for loop, then I set the for loop to print random numbers using random.randint.

• Experimental results

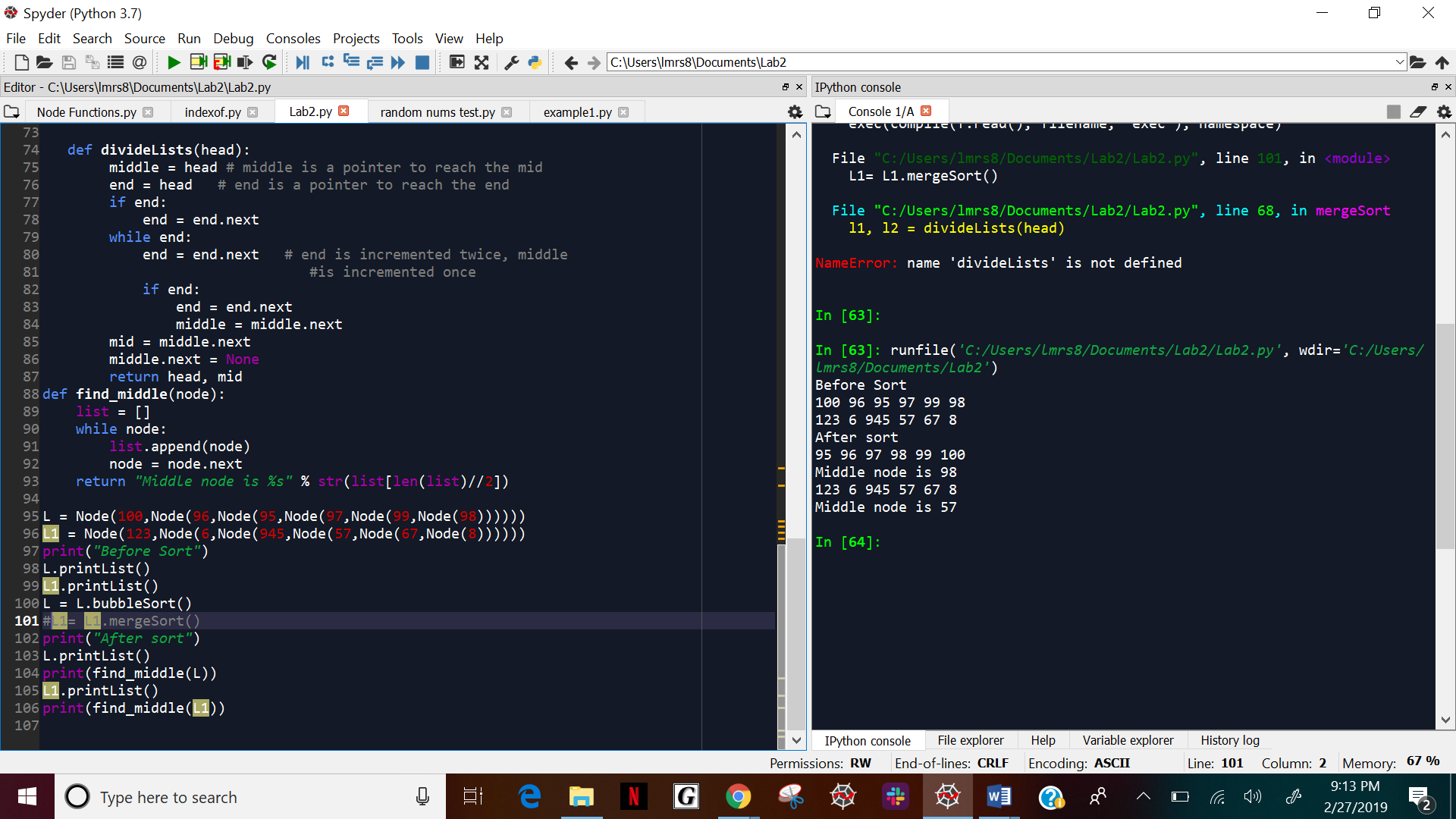
For this project I tried to break down every search method using algorithms and recursion. For the bubbleSort the running time was O(n^2). I tested the sorting using lists of 4 elements, 8 elements and 16 elements. For 4 elements, the number of comparisons can vary from 4 to 6. For 6 elements it would vary from 5 comparisons to 15 comparisons. For 8 elements it would be from 7 to 22 comparisons.

For merge sort the running time is O(nlogn). For 4 elements it would be 3 comparisons. For 6 elements it would be 12 comparisons. And for the 8 elements it was 16 comparisons.

For quick sort the running time is O(n^2). Although I was unable to get the quicksort methods functioning, I was able to calculate the number of comparisons that would have been made if it were to work. I was also unsuccessful in storing random integers in the linked list and getting them to wok with the sorting methods so I had to manually make random lists to test them with the sorting methods.

The results given in the table below are based off random lists of different levels of unorder.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting | Length 4 | Length 6 | Length 8 | Running Time |
| BubbleSort | 9 comparisons | 25 comparisons | 23 comparisons | O(n^2) |
| MergeSort | 3 comparisons | 12 comparisons | 14 comparisons | O(nlogn) |
| QuickSort | 3 comparisons | 6 comparisons | 12 comparisons | O(n^2) |



• Conclusion

In conclusions, I learned how to apply the different sorting algorithms or merge, bubble, and quick sort to linked list. I also learned how to analyze every sorting algorithm individually and see how each worked. From this I was able to derive the running times of the algorithms and see which one worked the most efficient. I was also able to learn how to split a linked list and find the median of the list. Although I was unsuccessful with some of the sorting algorithms and randomization of lists, I understood the logic behind all of them.

• Appendix – Source codes

class Node:

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

self.data = data

self.next = next

def \_\_str\_\_(self):

return str(self.data)

def printList(self):

print(self.data, end=' ')

if self.next:

self.next.printList()

else:

print()

def bubbleSort(self):

start = self

result = start.bubbleSortPass() # start sorting first node

start = result[0]

while result[1]:

result = start.bubbleSortPass()

start = result[0] # change the head

return start

def bubbleSortPass(self): # simulates one pass of bubble sort

if(self.next):

nextNode = self.next

# check if node need to be swapped

if(self.data > self.next.data):

self.next = nextNode.next # swap the nodes

nextNode.next = self

result = self.bubbleSortPass() # use for further sort

nextNode.next = result[0] # check if head changed

return (nextNode, True)

else:

result = self.next.bubbleSortPass()

self.next = result[0]

return (self, result[1])

else :

return (self, False)

def mergeLists(l1, l2):

temp = None

if l1 is None:

return l2

if l2 is None:

return l1

if l1.data <= l2.data:

temp = l1

temp.next = mergeLists(l1.next, l2)

else:

temp = l2

temp.next = mergeLists(l1, l2.next)

return temp

def mergeSort(head):

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

return head

l1, l2 = divideLists(head)

l1 = mergeSort(l1)

l2 = mergeSort(l2)

head = mergeLists(l1, l2)

return head

def divideLists(head):

middle = head # middle is a pointer to reach the mid

end = head # end is a pointer to reach the end

if end:

end = end.next

while end:

end = end.next # end is incremented twice, middle

#is incremented once

if end:

end = end.next

middle = middle.next

mid = middle.next

middle.next = None

return head, mid

def find\_middle(node):

list = []

while node:

list.append(node)

node = node.next

return "Middle node is %s" % str(list[len(list)//2])

L = Node(100,Node(96,Node(95,Node(97,Node(99,Node(98))))))

L1 = Node(123,Node(6,Node(945,Node(57,Node(67,Node(8))))))

print("Before Sort")

L.printList()

L1.printList()

L = L.bubbleSort()

L1= L1.mergeSort()

print("After sort")

L.printList()

print(find\_middle(L))

L1.printList()

print(find\_middle(L1))