|  |
| --- |
| CS 2302 Data Structures |
| Lab Report #2 |
| Spring 2019 |

|  |
| --- |
| Sebastian Gomez Hurtado  2-27-2019 |

**Lab Report**

Introduction:

The task was to create a list made up of random numbers of random length, sort them in 4 possible ways and find the middle element to prove that the 4 sorting methods output the same middle element for the same list. The sorting methods were bubble sort, in which you compare every element with each other and swap them if necessary, merge sort, which is a recursive method that split the list into two smaller lists that will merge into one sorted list, quicksort, which is also recursive and that involves also splitting your list into two but having as reference a pivot, which is the head of the list, and a better version of quicksort in order to improve its time complexity.

Proposed Solution and Implementation:

I divided the assignment into 2, which was doing each sorting algorithm and creating the missing methods for the given code for finding the median. I started with bubble sort, which has a running time of O(n^2). Using the pseudocode given in class, I based my code on that and made added some code in order to work. I used a Boolean variable to be able to swap the number of times necessary so the list could be sorted and added a counter to keep track of the comparisons made.

For the merge sort algorithm, which has a running time of O(nlogn), I split the received list into 2 equal-sized list, L1 and L2. I called the method recursively to also do that for L1 and L2 and the set the original list to the merged version of L1 and L2. In order for these to happen, I created another method called Merge, which sorted from least to greatest the list1 and list2 and put on another list. When this happened, the original list got assigned the values of the new merged list and the list got sorted. The problem that I got is that the program compiled but it did not sort the list.

Finally, for the quick sort, which its running time is O(n^2), I assigned the first element of the received list to a variable called pivot that worked as a point of division to divide the list into two. If the item on the list was less than the pivot, the element was put on a list called L1 and if it was greater than the pivot, it was assigned to a list named L2. After doing that, I used recursion to do the same for list L1 and for list L2. When the recursion calls were finished, I appended the pivot to L1 if it was not empty, and if yes, prepend it to L2. Finally, I checked the if the first list was empty, if yes, then there was no list and I set the original list to L2, meaning the head of the main list equal to the head of the second list and the same with the tail, and if it was not empty, before involving the original list, I combined L1 and L2 by setting the next of the tail of the first list to be the head of the second list and then assign to the main list the head of the first list and the tail of the second list. Now L, the main list, was sorted.

For the improved or optimized quick I thought of it as the same but instead of doing two recursive call, to improve its time, just do one because there would only be one list. In other words, is the same logic, you have a pivot which is the head of the main list you combine or switch the elements if they are less than or more than the pivot, but in the optimized version, I thought of it by only having one list, and if the item was less than the pivot, append it to the new list, which was L1, and if greater than, prepend it to L1. Then by doing one recursive call, because only one was needed because it was only one list, it kept sorting the list. Then I append the pivot and set the main list to the sorted list. The problem was that it did sort the list but not completely but the partial solution was that even though it did not sort it completely, it sorted it in a way that the median was the same as for the other methods.

The last part to see if every method gave the same median for the same random list, I created 4 different methods called median, median1, median2, median3, and median 4. Every method was made up of the same methods, being Copy, Print, and ElementAt, the only thing that was different was the sorting method. I imported to the program the random utility in order to do a list of random length with items of random value. After that I did the Copy method, which copied every element of the main list to another list to not loose the values of the main list, and I did the ElementAt method that received the list and the length of half of it to return the middle element of the sorted list. Also, to compare each sorting algorithm, I implemented a counter in each of them to keep track of the comparisons made.

To show the differences, the table below will contrast the different amount of comparisons depending of the length of the list.

|  |  |  |  |
| --- | --- | --- | --- |
| Length of the List | Median  # of Comparisons | Median3  # of Comparisons | Median4  # of Comparisons |
| Range from 0 to 20 | 27 | 36 | 34 |
| Range from 20 to 50 | 486 | 253 | 284 |
| Range from 50 to 100 | 1665 | 343 | 599 |

This are only approximations and it varies depending on the length.

Conclusions:

Sorting algorithms has various methods and within each method, its own way to do the required task. I learned the fundamentals of using a list of nodes and its different uses. I tried my best to code all of the methods and learned a lot from the process of thinking and doing them. I hope to keep improving my programming skills.

**I , Sebastian Gomez, certify that this project is entirely my own work, I wrote, debugged, and tested the code being presented, performed 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.**

Appendix:

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

"""

Created on Wed Feb 20 19:26:29 2019

@author: Sebastian Gomez

Course: Data Structure 2302

Assignment: Lab 2

Instructor: Olac Fuentes

T.A: Anindita Nath and Maliheh Zargaran

Purpose: Sort a list and return the middle element of a sorted list

"""

import random

#Node Functions

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):

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 Prepend(L,x):

#Inserts x at the beggining of the list

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

temp = L.head

L.head = Node(x)

L.head.next = temp

def getLength(L):

temp = L.head

count = 0

while temp is not None:

count = count + 1

temp = temp.next

return count

def BubbleSort(L):

done = True

counter = 0

while done:

done = False

temp = L.head

n = 0

while temp.next is not None and n < getLength(L):

if temp.item > temp.next.item:

counter = counter + 1

value = temp.item

value2 = temp.next.item

temp.item = value2

temp.next.item = value

done = True

temp = temp.next

n = n + 1

print('Number of comparisons: ',counter)

def QuickSort(L):

if getLength(L) > 1:

counter = 0

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)

counter = counter + 1

temp = temp.next

QuickSort(L1)

QuickSort(L2)

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

L.head = L1.head

L.tail = L2.tail

print('Number of comparisons: ',counter)

def QuickSortOptimized(L):

if getLength(L) > 1:

counter = 0

pivot = L.head.item

L1 = List()

temp = L.head.next

while temp is not None:

if temp.item < pivot:

Append(L1,temp.item)

else:

Prepend(L1,temp.item)

counter = counter + 1

temp = temp.next

QuickSort(L1)

Append(L1,pivot)

L.head = L1.head

L.tail = L1.tail

def Merge(L1,L2):

merged = List()

temp = L1.head

temp2 = L2.head

if temp is None:

return temp2

if temp2 is None:

return temp

if temp.item <= temp.item:

Append(merged,temp.item)

Merge(temp.next,temp2)

else:

Append(merged,temp2.item)

Merge(temp,temp2.next)

return merged

def MergeSort(L):

if getLength(L) > 1:

L1 = List()

L2 = List()

temp = L.head

i = 0

count = 0

if i < getLength(L)//2:

Append(L1,temp.item)

elif i > getLength(L)//2 and i < getLength(L):

Append(L2,temp.item)

temp = temp.next

i = i + 1

count = count + 1

print(count)

MergeSort(L1)

MergeSort(L2)

L = Merge(L1,L2)

def Copy(L):

copy = List()

temp = L.head

while temp is not None:

Append(copy,temp.item)

temp = temp.next

return copy

def ElementAt(C,median):

if IsEmpty(C):

return None

else:

temp = C.head

i = 0

while temp is not None and i < median:

temp = temp.next

i = i + 1

return temp.item

def Median(L):

C = Copy(L)

BubbleSort(C)

Print(C)

med = ElementAt(C,getLength(C)//2)

return med

print()

def Median2(L):

C = Copy(L)

MergeSort(C)

Print(C)

med = ElementAt(C,getLength(C)//2)

return med

print()

def Median3(L):

C = Copy(L)

QuickSort(C)

Print(C)

med = ElementAt(C,getLength(C)//2)

return med

print()

def Median4(L):

C = Copy(L)

QuickSortOptimized(C)

Print(C)

med = ElementAt(C,getLength(C)//2)

return med

print()

L = List()

n = random.randint(0,50)

for i in range(n):

rand = random.randint(0,n\*2)

Append(L,rand)

Print(L)

print('The middle element is',Median(L))

print('--------------------------------------------------------------')

print('The middle element is',Median2(L))

print('--------------------------------------------------------------')

print('The middle element is',Median3(L))

print('--------------------------------------------------------------')

print('The middle element is',Median4(L))