PAOLA TERRAZAS

CS2302

SORTING METHODS TO FIND MEDIAN

DR. FUENTES

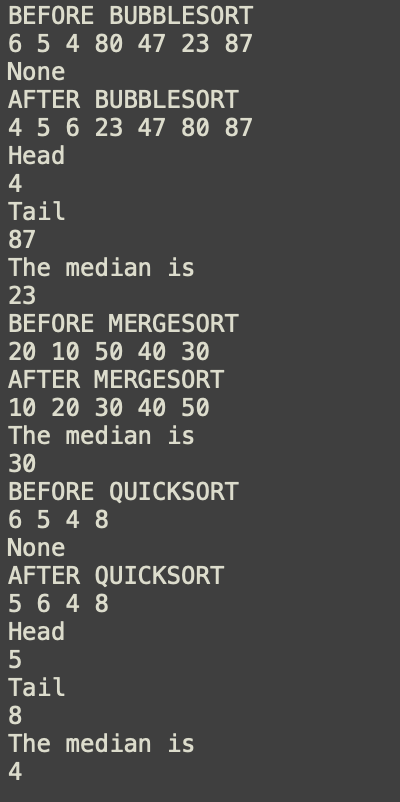
TA - ANINDITA NATH

2.25.2019

REPORT

For this lab, we were asked to sort a list using different sorting functions known as bubble sort, merge sort and quick sort to find the median of a list. First, I worked with bubble sort as it seemed the easiest. I broke it down into parts, bubble sort goes through every element in the list one by one comparing it to its previous element. If the element to the right is smaller than the element to the left, it will switch them. Therefore, bubble sort was simple. Then, I looked into quick sort. Quick sort’s algorithm was more difficult than bubble sort’s. Quick sort begins by dividing the elements into parts repeatedly until it is not possible to divide further, that is the partition method in my code. The key element for partitioning the elements is known as the pivot. Left partition contains all elements that are smaller than the pivot and the right partition contains the elements that are greater than the pivot. Then, we have merge sort. Merge sort is known as an “external algorithm based on divide and conquer strategy.” It begins by splitting the elements into two parts again and again until only one is left. It uses three storage ‘arrays’ where two are used for storing each half and the third one is used to store the final sorted list recursively. To find the median of the list, I created a method known as median that gets you the position of the median simply by diving the list by 2. Then, when I call that method, I traverse through the list ‘median’ times to get the median that is already in order because the sorting was done before. We were also asked to implement a modified version of quicksort that makes a single recursive call instead of two and that was quite challenging for me. As far as determining the big-O running time for every method, my bubble sort method is O(n^2). My merge sort and quick sort methods are O(nlogn) with the loops and recursive calls, they take O(nlogn) comparisons to sort n items.

Shown below is the output of my program, I labeled before sorting and after sorting so the difference and sorting is more obvious as well as the head and tail of the list.



I learned from this lab the efficiency of these three methods, learning that merge sort is the best one. I created a graph to show the comparisons the way I gathered in my code, shown below.

8000

Comparisons

0

Length of List

bubble sort

merge sort

quicksort

**Source Code**

# PAOLA TERRAZAS

# CS2302

# FINDING THE MEDIAN OF A LIST USING SORTING METHODS

# DR. FUENTES

# TA - ANINDITA NATH

# 2.20.2019

#Node Functions

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

def IsEmpty(L):

return L.head == None

def GetLength(L):

# Returns the number of items in list

temp = L.head # the temp is what will go through the list BEGINNING at the head

count = 0 # your iterator BEGINNING at 0 need to initialize

while temp is not None:

count += 1

temp = temp.next

return count

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 InsertAfter(L,x,item):

s = Search(L,x)

if s in None:

Append(L,item)

else:

s.next = Node(item,s.next)

def 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

def Search(L,x):

temp = L.head

while temp is not None:

if temp.item == x:

return temp

temp = temp.next

return None

def Prepend(L,x):

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

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

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 PrintRec(L):

# Prints list L's items in order using recursion

PrintNodes(L.head)

print()

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

else:

L.head = L.head.next

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

else:

temp.next = temp.next.next

def PrintReverse(L):

# Prints list L's items in reverse order

PrintNodesReverse(L.head)

print()

def bubbleSort(myList):

current = myList.head

while current is not None:

prev = current.next

while prev is not None:

if prev.item < current.item:

temp = current.item

current.item = prev.item

prev.item = temp

prev = prev.next

current = current.next

def median(L):

mid = GetLength(L) //2

return mid

def ElementAt(L,x):

temp = L.head

count = 0

while count < x:

temp = temp.next

count += 1

return temp

def quick\_sort(L,head,tail):

if head < tail:

split = partition(L,head,tail)

quick\_sort(L,head,split-1)

quick\_sort(L,split+1,tail)

def partition(L,head,tail):

count = 0

temp = ElementAt(L,head)

pivot = temp

leftside = head + 1

rightside = tail

piv = ElementAt(L,tail)

right = ElementAt(L,tail)

sorted = False

while not sorted:

if right.item < piv.item:

p = piv.item

piv.item = right.item

right.item = p

while leftside <= rightside and piv is not None and piv.item <= pivot.item:

count = count + 1

leftside = leftside + 1

piv = piv.next

while right.item > pivot.item and rightside >= leftside:

count = count + 1

rightside = rightside -1

right = ElementAt(L,rightside)

if rightside < leftside:

sorted = True

p = pivot.item

pivot.item = right.item

right.item = p

return rightside

def mergeLists(list\_one, list\_two):

temp = None

if list\_one is None:

return list\_two

if list\_two is None:

return list\_one

if list\_one.item <= list\_two.item:

temp = list\_one

temp.next = mergeLists(list\_one.next, list\_two)

else:

temp = list\_two

temp.next = mergeLists(list\_one, list\_two.next)

return temp

def mergeSort(head):

if head is None or head.next is None:

return head

list\_one, list\_two = divideLists(head)

list\_one = mergeSort(list\_one)

list\_two = mergeSort(list\_two)

head = mergeLists(list\_one, list\_two)

return head

# Defining function which will divide a linked list into two equal linked lists

def divideLists(head):

first = head #first is a pointer to reach the mid of linked list

last = head #last is a pointer to reach the end of the linked list

if last:

last = last.next

while last:

last = last.next #last is incremented twice while first is incremented once per loop

if last:

last = last.next

first = first.next

mid = first.next

first.next = None

return head, mid

L = List()

Append(L,6)

Append(L,5)

Append(L,4)

Append(L,80)

Append(L,47)

Append(L,23)

Append(L,87)

L2 = List()

Append(L2,6)

Append(L2,5)

Append(L2,4)

Append(L2,8)

L3 = List()

Append(L3,20)

Append(L3,10)

Append(L3,50)

Append(L3,40)

Append(L3,30)

print("BEFORE BUBBLESORT")

Print(L)

print(bubbleSort(L))

print("AFTER BUBBLESORT")

Print(L)

print("Head")

print(L.head.item)

print("Tail")

print(L.tail.item)

print("The median is")

middle = median(L)

temp = L.head

for i in range(middle):

temp = temp.next

#print(temp.item)

print(temp.item)

print ("BEFORE MERGESORT")

Print(L3)

L3.head = mergeSort(L3.head)

print ("AFTER MERGESORT")

Print(L3)

print("The median is")

middle = median(L3)

temp = L3.head

for i in range(middle):

temp = temp.next

#print(temp.item)

print(temp.item)

print("BEFORE QUICKSORT")

Print(L2)

print(quick\_sort(L2,0,GetLength(L2)-1))

print("AFTER QUICKSORT")

Print(L2)

print("Head")

print(L2.head.item)

print("Tail")

print(L2.tail.item)

print("The median is")

middle = median(L2)

temp = L2.head

for i in range(middle):

temp = temp.next

#print(temp.item)

print(temp.item)