Michael Gonzalez

CS 2302 Data Structures

Olac Fuentes

02/27/2019

Lab 1 – Sorting Algorithms

For this lab, we have been tasked with implementing several algorithms for finding the median of a list of integers which are Bubble Sort, Merge Sort, Quick Sort, and a modified version of Quicksort. For every method we had to determine its Big-O running time with respect to n, the size of the list, and determine the number of comparisons that each algorithm makes and determine if their analytical running times agree with what we see in class. The biggest challenge that I could see with this assignment is that I would have to figure out how to implement each sorting algorithm in python and being able to then measure the comparisons.

My solution for part 1 was to create a method called “Bubble Sort” that used an “L” a List variable, this list was created outside the method with random numbers and a length of size n. Inside the method, I created a “Counter” variable to count for each comparison done and a “change” variable so that the program can stop when no more changes were needed. Next there are two nested while loops, the first while loop has the condition of “change” meaning only when change has happened we continue this program, inside the loop we assign the head of the current list to a temporary variable “t” this is done to prevent losing the entire list, change the “change” variable to false and add 1 to the counter because a comparison happens on the next while loop. The second while loop has the condition “t.next is not None” this is so we don’t reach out of bonds and we find the end of the list. In this while loop there is an if condition to check whether if “t” is greater than the next term, t.next, this is to change the order of the two nodes so that the smaller one end in the left and updates “change” to True, out of the if statement we update “t” to be “t.next”, the next element in the list, and add 1 to the counter since a comparison was made. The program now loops until there is nothing else to sort and finally prints the number of comparisons done by the program. In comparison with a worst-case scenario Ex. Size = 7 the number of comparisons was 49 which is 7^2 proving that the running time is Big O(n^2).

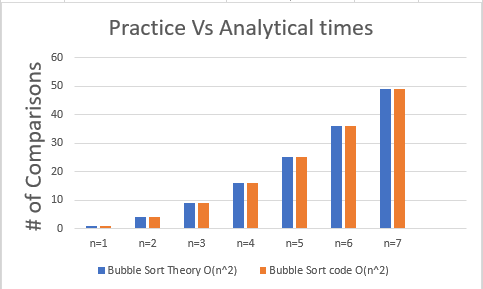
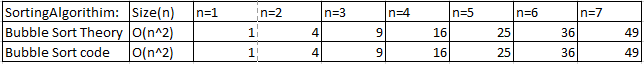
My solution for part 2 was to create a method called MergeSort that used a figure, “h” the head node of the current list. Inside the method it first checks to see ifnthe list is empty or if head.next is node and if it is it returns the head, if not it divides the list into 2 using another method called divideLists using that head node. The divideList method takes a head node from a list and splits it into 2 parts. One before the mid point and one after the midpoint. This was done by setting a fast variable and a slow variable, where the fast traversed the later part of the list and the slow traversed the beginning part of the list, once it travesed it returned the nodes head and mid back into mergesort as L1,L2 respectively. Next it made the recursive call to mergesort again using L1 and once again using L2 to keep dividing the list until it was only one element left. Next it used an updated version of the concate method to join the two lists and once that was all done it returned the sorted head node. The running time for this method, thanks to the master method, is Big O(nlogn). In comparison with a worst-case scenario Ex. Size = 5 the number of comparisons was 12 which is close to 5log2(5) proving that the running time is Big O(nlogn).

My solution for part 3 was to create a method called QuickSort that used a figure, an “L” list variable. Inside the method it will check if the list is empty and if so return just the list. Next it will check if the list is empty and if so it will return , if the list is not empty then the program creates a “piv” variable than holds the Head of the current list and then it will create two new lists called “LeftList” and “RightList” to store the list once it is divided by the pivot and a “temp” variable to hold the next element in the current list. Next, it goes through a while loop with the condition “temp is not None” this ensures us that we don’t go out of bounds and will reach the end of a list. Inside this loop we separate the items into the two list which are either less than or greater than the pivot variable. Outside the for loop we make 2 recursive calls each named a new variable one is called newRight and it calls QuickSort(leftList) and then other is called newLeft QuickSort(RigthList). Once those calls are don we append the piv to the new left and concentrate the new sorted list and return this new list. Based on what we learned in class this method has a Big O(n^2) for worst possible scenario, and that is how I tested it. In comparison to its worst-case scenario and using size = 4 I was able to count 16 comparisons which happens to be 4^2 proving that the running time is Big O(n^2)

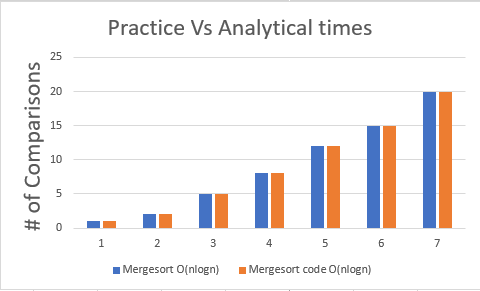
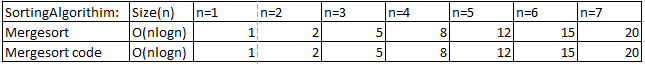
My solution for part 4 was a method called “Rank” that used an “L” a List variable, an “n” to find the element at n position and “Count” to keep track of the number of comparisons since this program is recursive. Inside the method using the “Count” value, it will store that number in a local variable called “count”, next it will check if the list is empty and if so it will return , if the list is not empty then I create “piv” , a pivot, which is the head of the current list and two new lists called “SmallList” and “LargeList” to store the list once it is divided by the pivot and a temp variable to hold the next element in the current list. Next it goes through a while loop with the condition “temp is not None” this ensures us that we don’t go out of bounds and will reach the end of a list. Inside this loop we separate the items into the two list which are either less than or greater than the pivot variable and, in each comparison, we add one to the counter. After the while loop is complete there are 3 if statements that check where the element in rank n is. #1 if n is equal to the length of the SmallList then the element is the pivot of the list and we just add one to counter and return the pivot with the counter. #2 if n is greater than the length of SmallList then the element is in the LargeList then we add one to counter and return the recursive call “rank(LargeList, n, counter). Lastly if n is less than the length of SmallList then the element with rank n is in the SmallList then we add one to counter and make the cursive call rank(SmallList, n, counter). The running time for this method, thanks to the master method, is Big O(nlogn). In comparison with a worst-case scenario Ex. Size = 5 the number of comparisons was 12 which is close to 5log2(5) proving that the running time is Big O(nlogn).

Results and tables

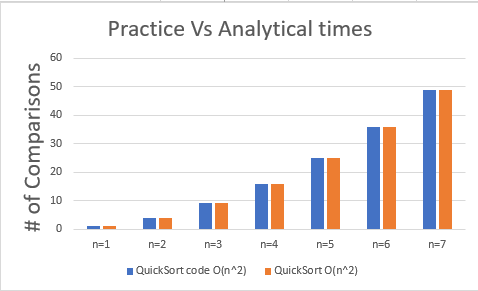
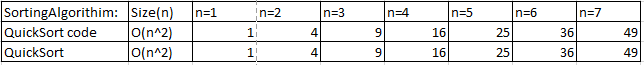
1.

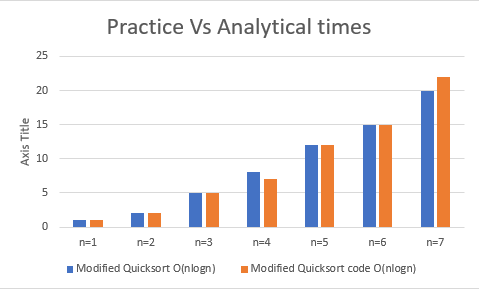
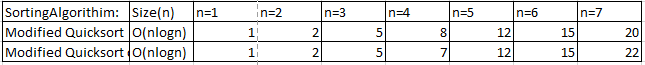
2.

3.

4.

In conclusion, I learned how to program various sorting algorithms and how to measure their respective running times against each other. Aside from the algorithms I have now a somewhat better understanding of Big O and running rime based on comparisons. I have become more comfortable with coding in python than in lab 1and I believe that I will be able to learn more from future labs to come.

I certify that this project is entirely my own work. I wrote, debugged, and tested 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.

– Michael Gonzalez

Appendix – code

1. Bubble Sort

#Author: Michael Gonzalez

#Course: CS 2302 Data Structures

#Lab 2 Part 1

#TA: Anindita Nath

#Purpose:The purpose of this part is to sort a Linked List using the Bubble sort Method

#and return the middle elemnt.

#also we need to measure the number of comparisons Bubble sort makes to compare it to other sorting methods

import random

#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 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):

Xcopy = List()

temp = L.head

while temp is not None:

Append(Xcopy,temp.item)

temp = temp.next

return Xcopy

def ElementAt(L,mid):

if IsEmpty(L):

return None

else:

temp = L.head

while mid is not 0:

temp = temp.next

mid -= 1

return temp.item

def GetLength(L):

if IsEmpty(L):

return 0

else:

count = 0

temp = L.head

while temp is not None:

count += 1

temp = temp.next

return count

def bubbleSort(L):

#Bubble sort

#O(n^2)

#Use a Counter to count how many comparisons happen in bubble sort for an n list

Counter = 0

change = True

while change:

change = True

t = L.head

change = False

Counter += 1

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

Counter += 1

print("The total number of comparisons is: ", Counter)

def Median(L):

C = Copy(L)

bubbleSort(C)

Print(C)

return ElementAt(C,GetLength(C)//2)

def genRanList(n):

L = List()

for i in range(n):

x = random.randrange(0, 101, 1)

Append(L,x)

return L

size = 4

testList = genRanList(size)

#testList = List() #example from Dr. Flores

#Append(testList,30)

#Append(testList,70)

#Append(testList,20)

#Append(testList,10)

#Append(testList,40)

#Append(testList,60)

#Append(testList,50)

#Print(testList)

#testList = List() #worst case Scenario

#Append(testList,70)

#Append(testList,60)

#Append(testList,50)

#Append(testList,40)

#Append(testList,30)

#Append(testList,20)

#Append(testList,10)

#Print(testList)

Y = Copy(testList)

Print(Y)

x = Median(Y)

print(x)

2. Merge Sort

#Author: Michael Gonzalez

#Course: CS 2302 Data Structures

#Lab 2 Part 2

#TA: Anindita Nath

#Purpose:The purpose of this part is to sort a Linked List using the Merge sort Method

#and return the middle elemnt.

#also we need to measure the number of comparisons Merge sort makes to compare it to other sorting methods

import random

#Node Functions

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):

Xcopy = List()

temp = L.head

while temp is not None:

Append(Xcopy,temp.item)

temp = temp.next

return Xcopy

def ElementAt(L,mid):

if IsEmpty(L):

return None

else:

temp = L.head

while mid is not 0:

temp = temp.next

mid -= 1

return temp.item

def GetLength(L):

if IsEmpty(L):

return 0

else:

count = 0

temp = L.head

while temp is not None:

count += 1

temp = temp.next

return count

def mergeLists(l1, l2):

temp = None

if l1 is None:

return l2

if l2 is None:

return l1

if l1.item <= l2.item:

temp = l1

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

else:

temp = l2

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

return temp

# Defining function which will sort the linked list using mergeSort

def mergeSort(head):

if head is None or head.next is None:

return head

l1, l2 = divideLists(head)

l1 = mergeSort(l1)

l2 = mergeSort(l2)

head = mergeLists(l1, l2)

return head

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

def divideLists(head):

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

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

if fast:

fast = fast.next

while fast:

fast = fast.next # fast is incremented twice while slow is incremented once per loop

if fast:

fast = fast.next

slow = slow.next

mid = slow.next

slow.next = None

return head, mid

def Median(L):

C = Copy(L)

C.head= mergeSort(C.head)

Print(C)

return ElementAt(C,GetLength(C)//2)

def genRanList(n):

L = List()

for i in range(n):

x = random.randrange(0, 101, 1)

Append(L,x)

return L

size = 5

testList = genRanList(size)

#testList = List() #example from Dr. Flores

#Append(testList,30)

#Append(testList,70)

#Append(testList,20)

#Append(testList,10)

#Append(testList,40)

#Append(testList,60)

#Append(testList,50)

#Print(testList)

#testList = List() #worst case Scenario

#Append(testList,70)

#Append(testList,60)

#Append(testList,50)

#Append(testList,40)

#Append(testList,30)

#Append(testList,20)

#Append(testList,10)

#Print(testList)

x = Median(testList)

print(x)

3. Quick Sort

#Author: Michael Gonzalez

#Course: CS 2302 Data Structures

#Lab 2 Part 3

#TA: Anindita Nath

#Purpose:The purpose of this part is to sort a Linked List using the Quick sort Method

#and return the middle elemnt.

#also we need to measure the number of comparisons Quick sort makes to compare it to other sorting methods

import random

#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 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):

Xcopy = List()

temp = L.head

while temp is not None:

Append(Xcopy,temp.item)

temp = temp.next

return Xcopy

def ElementAt(L,mid):

if IsEmpty(L):

return None

else:

temp = L.head

while mid is not 0:

temp = temp.next

mid -= 1

return temp.item

def GetLength(L):

if IsEmpty(L):

return 0

else:

count = 0

temp = L.head

while temp is not None:

count += 1

temp = temp.next

return count

def quickSort(L):

if IsEmpty(L):

return L

Print(L)

LeftSide = List()

RigthSide = List()

piv = L.head.item

temp = L.head.next

while temp is not None:

if piv > temp.item:

Append(LeftSide,temp.item)

temp = temp.next

else:

Append(RigthSide,temp.item)

temp = temp.next

newLeft = quickSort(LeftSide)

newRigth = quickSort(RigthSide)

Append(newLeft,piv)

return Concate(newLeft,newRigth)

def Concate(L1,L2):

if IsEmpty(L1):

return L2

elif IsEmpty (L2):

return L1

else:

L1.tail.next = L2.head

L1.tail = L2.tail

return L1

def Median(L):

C = Copy(L)

Y= quickSort(C)

Print(Y)

return ElementAt(Y,GetLength(Y)//2)

def genRanList(n):

L = List()

for i in range(n):

x = random.randrange(0, 101, 1)

Append(L,x)

return L

size = 5

testList = genRanList(size)

#testList = List() # example from Dr. Flores

#Append(testList,30)

#Append(testList,70)

#Append(testList,20)

#Append(testList,10)

#Append(testList,40)

#Append(testList,60)

#Append(testList,50)

#Print(testList)

#testList = List() #worst case Scenario

#Append(testList,70)

#Append(testList,60)

#Append(testList,50)

#Append(testList,40)

#Append(testList,30)

#Append(testList,20)

#Append(testList,10)

#Print(testList)

x = Median(testList)

print(x)

4. Modified Quicksort

#Author: Michael Gonzalez

#Course: CS 2302 Data Structures

#Lab 2 Part 4

#TA: Anindita Nath

#Purpose:The purpose of this part is to sort a Linked List using a Modified versions of quicksort

#where we only make recursive calls to where our middle point might be

#and return the middle elemnt.

#also we need to measure the number of comparisons this Quick sort makes to compare it to other sorting methods

import random

#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 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):

Xcopy = List()

temp = L.head

while temp is not None:

Append(Xcopy,temp.item)

temp = temp.next

return Xcopy

def ElementAt(L,mid):

if IsEmpty(L):

return None

else:

temp = L.head

while mid is not 0:

temp = temp.next

mid -= 1

return temp.item

def GetLength(L):

if IsEmpty(L):

return 0

else:

count = 0

temp = L.head

while temp is not None:

count += 1

temp = temp.next

return count

def rank(L,n,Count):

counter = Count

if IsEmpty(L):

return L

Print(L)

SmallList = List()

LargeList = List()

piv = L.head.item

temp = L.head.next

while temp is not None:

if piv > temp.item:

Append(SmallList,temp.item)

counter += 1

temp = temp.next

else:

Append(LargeList,temp.item)

counter += 1

temp = temp.next

#a) Element with rank n is in the pivot

if n == GetLength(SmallList):

counter += 1

return piv,counter

#b) Element with rank n is in SmallList

elif n > GetLength(SmallList):

counter += 1

n = n - GetLength(SmallList)-1

return rank(LargeList, n, counter)

#c) Element with rank n is in LargeList

elif n < GetLength(SmallList):

counter += 1

return rank(SmallList, n,counter)

def Median(L):

C = Copy(L)

x,Comp = rank(C,GetLength(C)//2,0)

print("the # of Comparisons is: ",Comp)

return x

def genRanList(n):

L = List()

for i in range(n):

x = random.randrange(0, 101, 1)

Append(L,x)

return L

size = 6

testList = genRanList(size)

#testList = List() #example from Dr. Flores

#Append(testList,30)

#Append(testList,70)

#Append(testList,20)

#Append(testList,10)

#Append(testList,40)

#Append(testList,60)

#Append(testList,50)

#Print(testList)

#testList = List() #worst case Scenario

#Append(testList,60)

#Append(testList,50)

#Append(testList,40)

#Append(testList,30)

#Append(testList,20)

#Append(testList,10)

#Print(testList)

mid = Median(testList)

print("the middle is",mid)