Armando Ortiz

Feb 27, 2019

CS 2302 Data Structures

Lab Report 2

UTEP

Instructor: Olac Fuentes

**Introduction**

The main purpose of this lab was to compare the runtimes of four sorting algorithms: bubble sort, merge sort, quicksort, and a modified quicksort using one recursive call. These algorithms needed to sort copies of a Linked List created with random numbers, and use the function provided by the professor to return the median. Finally, the program needs to count the number of comparisons performed by the different algorithms. This is the method of discovering the runtime of the algorithms, and figuring out which ones sort a list more quickly.

**Proposed Solution Design and Implementation**

The method for solving the problems in this lab was by breaking them up into smaller functions by type of sorting method. The way of finding the median was by using the function that was provided by the professor, which was used with each different sorting function. The real challenge was figuring out how to actually design and implement these sorting algorithms. The modified quicksort function was the one function that was not solved.

The first function that was tackled was bubble sort. This was the one problem that was able to be solved using only loops. The bubble sort was implemented by using two loops, a top one to perform pass throughs of the list, and a second nested loop to perform the comparisons. Inside this second loop is where the items from the nodes would be switched to the ascending order.

Merge Sort was the next sorting algorithm that needed to be solved. This function needed to be solved using two different functions; one for sorting, and the other to actually merge the lists. The sorting function needed to be called recursively. The variables needed were a midpoint integer, found by dividing the length of the list, and two temporary lists. The two lists were for the first half before the midpoint, and for after the midpoint. A loop is then used to append the information from the original list into the two smaller lists. Then, two recursive calls are performed, one for left list, and one for the right list, and these lists are made smaller until their lengths are one.

After, all three lists are called by the merge function to combine them. In this function the lists are placed in a loop used to append their information into the original, now empty, list L. Inside the loop, the two smaller lists are compared, and the item from the node with the smaller number is appended into the original list. This done until one of the nodes is empty. Finally the function checks to see if any of the smaller lists are not empty, and appends this information into the larger list.

Quick Sort was solved after figuring out the solution for merge sort. The sorting function for quick sort is similar to merge sort, except instead of finding a midpoint, the function uses the head node as a pivot. Then, the numbers smaller than the pivot are placed in the left list, and larger numbers are placed in the right list. Then, the left and right lists are placed in two recursive functions until their length is one. Before the lists are combined the pivot is appended to the end of the left list. All three lists are called into the combine function where the items of the two smaller lists are placed in the larger list.

**Experimental Results**

The way of finding the runtime for each of the functions was by making a global variable named comparisons. The way of testing the algorithms was by placing the random list call inside of a loop, with an original list size of 5,and iterated by 5 until the list had a length of 30. The Big-O Notations for the three sorting algorithms are O(N2 ) for bubble sort, O(n log n) for merge sort, and O(n log n) for quicksort. Below is the output of one of the tests.

**Output**

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

The amount of comparisons seems to match the Big O’s of the functions. The comparisons for the bubble sort rises much faster than the other two algorithms. Merge Sort and Quicksort seem to stay around the same amount of comparisons as each other as the lists rise.

**Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Comparisons** | | | |
| **List Size** | **Bubble Sort** | **Merge Sort** | **Quicksort** |
| 5 | 20 | 24 | 19 |
| 10 | 90 | 68 | 53 |
| 15 | 210 | 118 | 88 |
| 20 | 380 | 176 | 167 |
| 25 | 600 | 236 | 223 |
| 30 | 870 | 296 | 276 |

**Graph**



The graph also shows how many more comparisons the bubble sort algorithm makes compared to the other two.

**Conclusion**

This lab showed just how much faster merge sort and quicksort are than bubble sort. The rate at which the number of comparisons for bubble sort will grow as list sizes grow is incredible when compared to the two other sorting algorithms. It also showed that merge sort and bubble sort can be quite similar when it comes to runtime, despite how they actually sort the lists themselves. This lab also showed how useful recursion can be when moving through and changing Linked Lists, and how helpful sorting them can be.

**Appendix**

﻿"""

Author: Armando Ortiz

Course: CS 2302 Data Structures

Assignment: Lab 2

Instructor: Olac Fuentes

T.A.: Anindita Nath, Maliheh Zargaran

Last Modification: Feb 26, 2019

Purpose: The purpose of this program is to compare bubble sort, merge sort,

quicksort, and modified qiucksort to compare their running times.

This is done using linked lists of differing sizes

"""

import numpy as np

comparisons = 0 # Global variable for counting comparisons

#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)

#List Functions

class List(object):

# Constructor

def \_\_init\_\_(self):

self.head = None

self.tail = None

self.length = 0

def IsEmpty(L):

# Checks if List L is empty

return L.head == None

def GetLength(L):

# Finds length of List L

if IsEmpty(L):

return 0

else:

temp = L.head

count = 0

while temp is not None:

temp = temp.next

count +=1

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

L.length += 1

def ElementAt(L,n):

# Finds element at n location for List L

if IsEmpty(L) or (n>GetLength(L)-1):

return None

temp = L.head

for i in range(n):

temp = temp.next

return temp.item

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):

# Makes copy of List L

temp = L.head

copy = List()

while temp is not None:

Append(copy, temp.item)

temp = temp.next

return copy

def randomList(n):

# Creates List of size n using random integers

L = List()

for i in range(n):

Append(L,np.random.randint(1, 101))# Adds random integer to list

return L

# Sorting Functions

# Bubble Sort

def bubbleSort(L):

global comparisons

tempNode = L.head

while tempNode is not None:# Iterate through each item on list until done

t = L.head

while t.next is not None:# Iterates pass of list

if t.item > t.next.item:# Compares item to next item in list

temp = t.next.item

t.next.item = t.item

t.item= temp

t = t.next

comparisons = comparisons + 1

tempNode = tempNode.next

# Merge Sort Functions

# Function for merging the lists

def merge(L,L1,L2):

global comparisons

while (L1.head is not None) and (L2.head is not None):# Compare

if L1.head.item <= L2.head.item:# Adds L1 item if L1 is smaller

Append(L,L1.head.item)

L1.head = L1.head.next

elif L2.head.item < L1.head.item:# Adds L2 item if L2 is smaller

Append(L,L2.head.item)

L2.head = L2.head.next

comparisons = comparisons +1

if L1.head is None:# Adds rest of L1 if L2 is empty

while L2.head is not None:

Append(L,L2.head.item)

L2.head = L2.head.next

comparisons = comparisons + 1

elif L2.head is None:# Adds rest of L2 if L1 is empty

while L1.head is not None:

Append(L,L1.head.item)

L1.head = L1.head.next

comparisons = comparisons + 1

def mergeSort(L):

global comparisons

if GetLength(L) > 1:

mid = (GetLength(L))//2 # Finds midpoint

L1 = List()# Left List

L2 = List()# Right List

i = 0;

while L.head is not None:

if i < mid:# Adds items on list before midpoint

Append(L1,L.head.item)

else:# Adds items on list after midpoint

Append(L2,L.head.item)

L.head = L.head.next

i = i + 1

comparisons = comparisons +1

mergeSort(L1)# Recurrence for left side of list

mergeSort(L2)# Recurrence for right side of list

merge(L,L1,L2)# Call to merge lists

# Quicksort Functions

# Function to combine lists

def combine(L,L1,L2):

global comparisons

while L1.head is not None:# Adds left half of list

Append(L,L1.head.item)

L1.head = L1.head.next

comparisons = comparisons + 1

while L2.head is not None:# Adds right side to list

Append(L,L2.head.item)

L2.head = L2.head.next

comparisons = comparisons + 1

# Quicksort Function

def quickSort(L):

global comparisons

if GetLength(L) > 1:

pivot = L.head.item# Item to compare to rest of list

L1=List()# Left List

L2=List()# Right List

L.head = L.head.next# List item after Pivot

while L.head is not None:# Move through List

if L.head.item < pivot:# Items lower than pivot

Append(L1,L.head.item)

else:# Rest of the items

Append(L2,L.head.item)

comparisons = comparisons + 1

L.head = L.head.next

quickSort(L1)# Recurrence for left side

quickSort(L2)# Recurrence for right side

Append(L1,pivot)# Append pivot to end of Left List

combine(L,L1,L2)# Call to combine the lists

# Find Median

# Median of Bubble Sorted List

def bubbleMedian(L):

global comparisons

C = Copy(L)

bubbleSort(C)

print("Bubble Sort",end=' ')

Print(C)

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

# Median of Merge Sorted List

def mergeMedian(L):

C = Copy(L)

mergeSort(C)

print("Merge Sort",end=' ')

Print(C)

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

# Median of Quick Sorted List

def quickMedian(L):

C = Copy(L)

quickSort(C)

print("Quick Sort:",end=' ')

Print(C)

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

# Main

i = 5

while i <= 30:# Loop to test comparisons

print("Test Size: ",i)

L = randomList(i)

print("Original L")

Print(L)

comparisons = 0

print("Median: ", bubbleMedian(L))

print("Comparisons",comparisons)

print("")

comparisons = 0

print("Median: ", mergeMedian(L))

print("Comparisons: ",comparisons)

print("")

comparisons = 0

print("Median: ", quickMedian(L))

print("Comparisons: ",comparisons)

print("")

print("")

i = i + 5

A screenshot of a cell phone

Description automatically generated