Lab 2 Report

Nicole Favela

CS 2302

February 27, 2019

# Intro

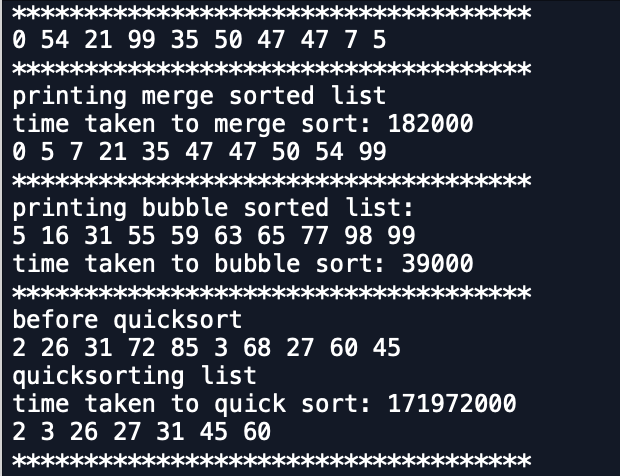
The task I was faced with for this lab was to implement bubble sort, merge, sort and quicksort on a list of integers. Using the linked list basic functionalities provided in class, I implemented the sorting algorithms and returned the median number of the lists.

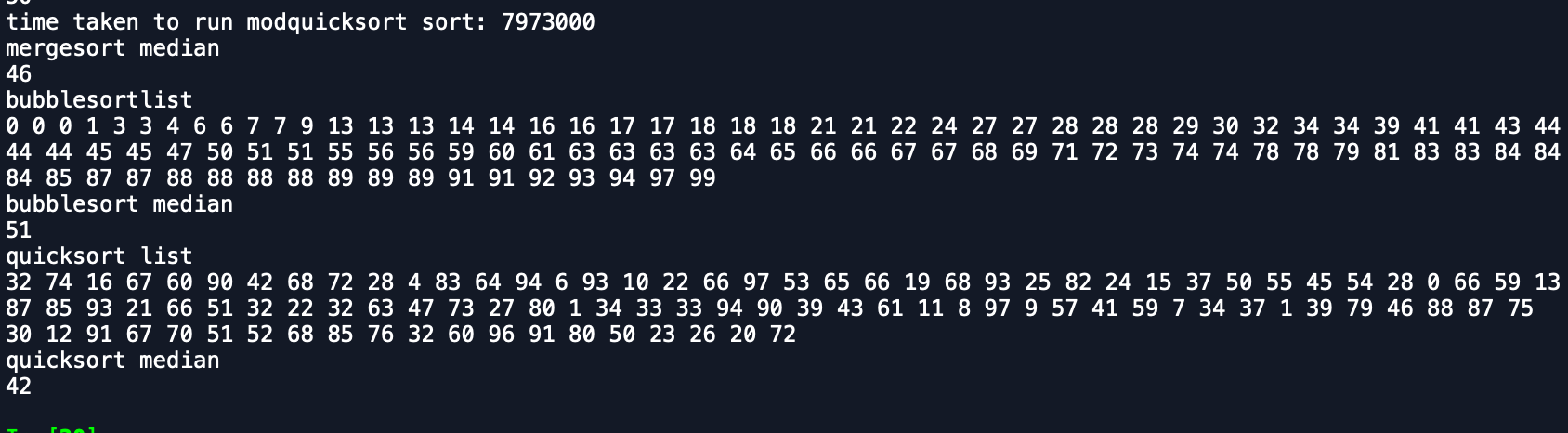
**Approach**

My approach to bubble sort was to model it after bubble sort in arrays as a reference. Since bubble sort is a fairly simple sorting algorithm that only looks at adjacent elements and swaps their positions if they are out of order I knew I needed to keep track of the exchanges. Since it is using linked lists instead of arrays, I needed to store the data in a temporary variable in order to change the data within neighboring nodes. After there are no more exchanges to be made, it is clear that the list is sorted, and the list is returned in ascending order. My approach to merge sort was to recursively split the list into smaller and smaller elements until they are of size 1, then unite them in their correct order. This involved using a couple different methods to aid in the process. First, divideLists, returns the head and middle of two lists created. This makes the recursive size smaller every call. MergeLists is used to unite the lists at the end based on the value of item. Recursive calls are made on the rest of l1 and l2. Finally, mergeSort returns the head of the sorted list. For quicksort, I knew it would be challenging to implement it in linked lists due to the lack of indices. First, I tried to make a kind of artificial index for the linked lists, but I ended up just using a loop and making comparisons similar to bubble sort but with 2 empty lists. I used to empty lists to collect the data that is smaller than the pivot (L1) and larger than the pivot (L3). I then made 2 recursive calls with both lists and prepending or appending the pivot depending on its value. After I had to lists completed, I just connected then at the end. For that method, I also created prepend which adds and element to the beginning of the list. ModQuickSort works similarly to quicksort except it takes the larger sub list and recursively makes it smaller until the median element of that list is returned or is equal to the pivot. I also created the function getElementAt which was necessary to find the median of the list. This function simply used a counter I called index to match is to the number of iterations in the loop which would return the element at the specified position much like the index of an array.

**Sample runs and output**

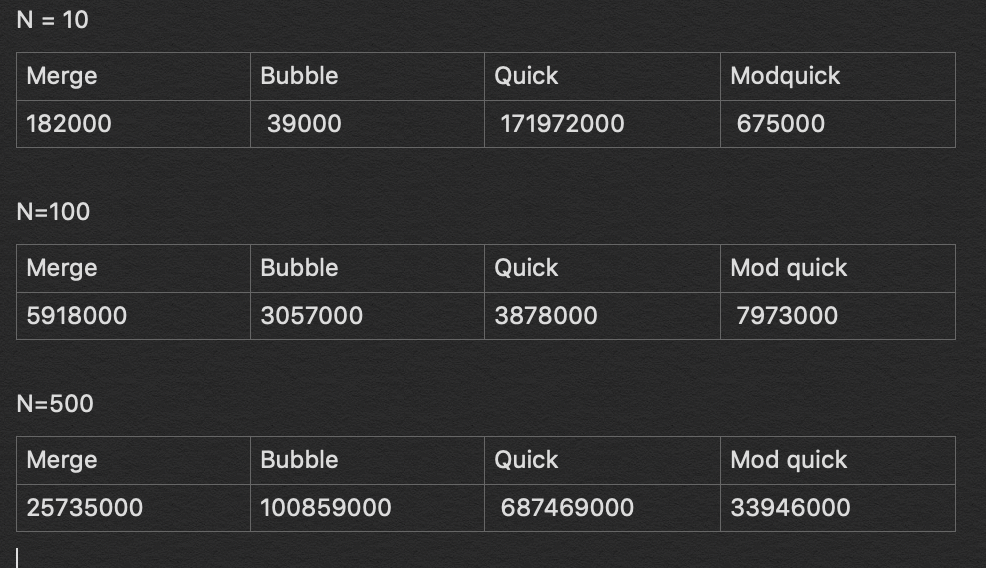
For testing my code, I used several different sized linked lists. I wanted to make sure my code worked properly for different input sizes. Quicksort was the most challenging for me. I used quicksort of arrays of a reference and knew it would be difficult to implement in linked lists because the algorithm works by choosing a pivot and partitions smaller elements to the left of the pivot and larger ones to the right. I tried to mimic this by creating L2 and L3 which are 2 empty lists and simply appending to them depending on their value in relation to the pivot which I made the head of the original Linked list, L. I kept adding items to the lists using a while loop and made recursive calls to sub lists and continued as long as the lists are greater than length 1. I found that when doing several runs of this method, I encountered some instances where the function would not completely sort the list or leave the pivot out. I presume the reason had to do with one of the lists sometimes being completely empty. The functions seems to work in the majority of cases, however, and if given more time, it would be a simple fix. I started with 5 items in my linked list and then 10 and gradually increase with the highest N=500. For reference my machine has a 2.3 GHz intel core i5 processor and is running macOS Mojave. Runtimes are also affected by the number of things running on a machine and the amount of RAM available as well.



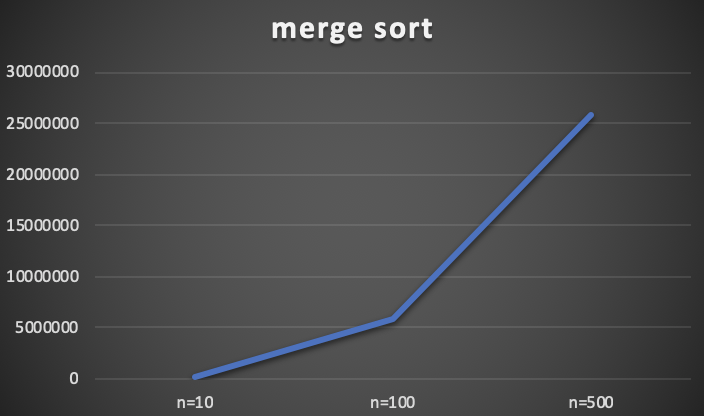


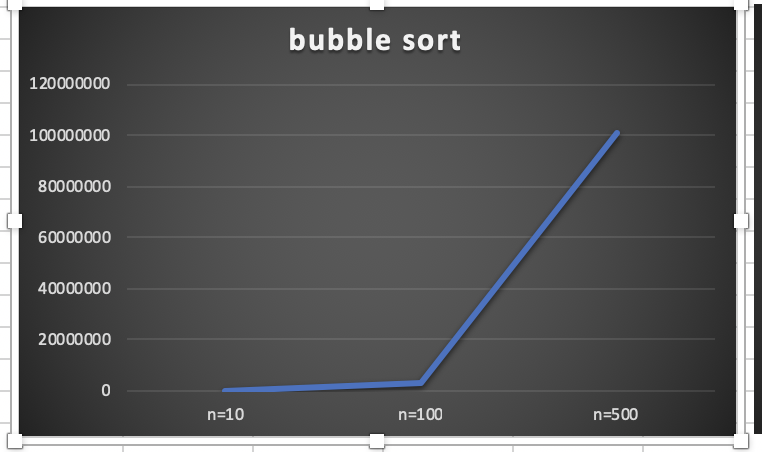
**Experimental results**

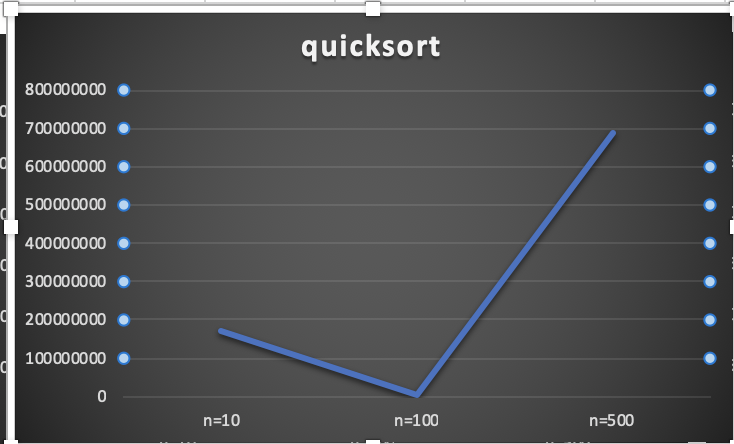
For the runtime analysis of my code, I used a for loop to execute each of my functions 10 times. I tested my methods with linked lists of size 10, 100, and 500. Due to the processing limitations of my machine I was unable to reach n=1000. Using another for loop and my append function I first populated my linked lists of L1, L2, L3 and L4 with random integers from 0 to 100. When I tested this the first time, I was using time.time() to record the start and end times and subtracted them to obtain my results. I was getting numbers in 1/e^-05, however, and chose to switch to time.time\_ns() to calculate runtimes in nanoseconds for the purposes of graphing and comparisons. The average case runtime of bubble sort is O(n^2). It is evident from the graph that as n gets larger, the time taken for bubble sort to completely sort the linked list increases in quadratic time. Of course, more n values between ranges would have provided a clearer picture of this. The runtime of merge sort is O(nlogn) in all cases. This is due to the fact that merge sort is continually dividing the list by 2. Once the linked lists have merged back together in their correct place, the list is sorted accounting for the n extra work. Similarly, quicksort’s average case is O(nlogn) for the same reason of dividing the work of sorting a list of size n. However, if the list is already sorted, the runtime degrades to O(n^2). This is avoided mostly, however, by generating random elements. The runtime of modified quicksort is similar but processes only one part of the original list. The table of results is as follows.

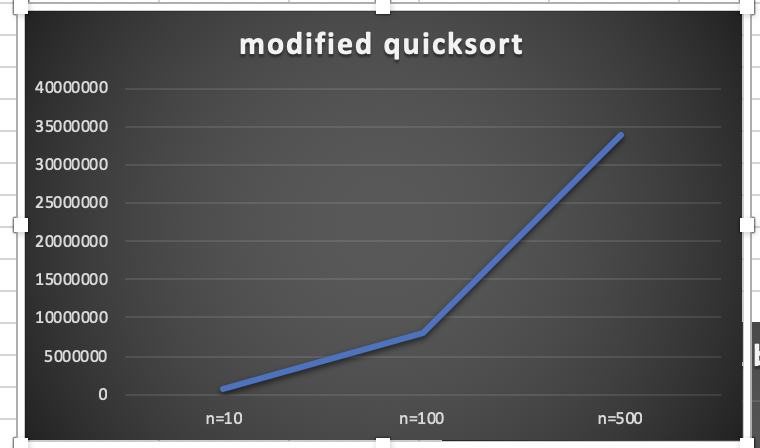


I also generated the following graphs in excel in order to show the algorithms behavior when input sizes increase. The x axis shows the input sizes or linked list of size n. The y axis represents the time taken in nanoseconds for the function to execute and sort the lists.









**Conclusions**

From this lab, I learned to implement sorting algorithms in linked lists. Previously, I had only dealt with these algorithms in arrays or tracing on paper. It was challenging to implement the same things in linked lists due to the lack of indices. This lab helped me to practice linked lists and helped me to learn more ways of ordering them using recursion.

**Appendix (source code)**

﻿#CS2302

#Nicole Favela

#Lab2

#instructor: Olac Fuentes

#TAs: Anindita Nath and Maliheh Zargaran

import random

size = 0

import copy

#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):

global size

# Inserts x at end of list L

if IsEmpty(L):

size+=1

L.head = Node(x)

L.tail = L.head

else:

size+=1

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 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()

#gets length of list

def getLength(L):

temp = L.head

count = 0

while temp is not None:

count = count+1

temp = temp.next

return count

#gets element at specific index

def getElementAt(L,i):

index = 0

temp = L.head

while temp != None:

if index == i:

return temp.item

index = index+1

temp = temp.next

#get median element of list

def Median(L):

C = copy.copy(L)

temp = (getLength(C))//2

return temp

#get median element of list for bubblesort

def Median1(L):

C = copy.copy(L)

bubblesort(C)

temp = (getLength(C))//2

return temp

#get median element of list for merge

def Median2(L):

C = copy.copy(L)

mergeSort(C.head)

temp = (getLength(C))//2

return temp

#get median element of list for quick

def Median3(L):

C = copy.copy(L)

quicksort(C)

temp = (getLength(C))//2

return temp

#combines lists

def mergeLists(L1, L2):

temp = None

if L1 is None: #if one list in None returns other

return L2

if L2 is None:

return L1

if L1.item <= L2.item: #compares items

temp = L1 #stores temp and calls mergeList on rest of list

temp.next = mergeLists(L1.next, L2)

else:

temp = L2

temp.next = mergeLists(L1, L2.next) #stores temp and moves to l2

return temp

#merge sorts list

def mergeSort(head):

if head is None or head.next is None:

return head

L1, L2 = divideLists(head) #calls divideList

L1 = mergeSort(L1) #recursive call to l1

L2 = mergeSort(L2) #recursive call to l2

head = mergeLists(L1, L2) #returns head of merged list

return head

#divides list for mergesort

def divideLists(head):

curr = head

quick = head

if quick:

quick = quick.next #starts at next element

while quick:

quick = quick.next #moves to next next element

if quick: #if fast is not None

quick = quick.next #move to next

curr = curr.next

mid = curr.next

curr.next = None

return head, mid

#quicksorts L recursively

def quicksort(L):

if getLength(L) > 1: #List of length 1 is already sorted

L2 = List() #less than pivot

L3 = List() #greater than pivot

currpivot = L.head.item #pivot value

temp = L.head.next #used to iterate

while temp is not None:

if currpivot > temp.item: #appends smaller items to L2

x = temp.item

Append(L2,x)

else:

x = temp.item #creates list of items greater

Append(L3,x)

temp = temp.next

quicksort(L2) #recursively gets rest of lists

quicksort(L3)

#puts pivot in correct location

if L2.head != None:

prepend(L3,currpivot)

else:

Append(L2,currpivot)

if L2.head != None: #connects lists

L.head = L2.head

L.tail = L3.tail

L2.tail.next = L3.head

else:

L.head = L3.head

L.tail = L3.tail

#modied quicksort

def ModQuickSort(L,mid):

if IsEmpty(L):

return

else:

lo = List() #strores items less than pivot

maximum = List() #stores items greater than pivot

pivot=L.head.item #pivot is first item

temp=L.head.next

while temp!=None:

if pivot<temp.item: # if item is bigger tha pivot

Append(lo,temp.item) #stores larger item in greater list

else:

Append(maximum, temp.item)

temp=temp.next #moves to next

if getLength(lo)>mid: #if length of least is greater than index of mid

return ModQuickSort(lo,mid) # get middle of smaller list

elif (getLength(lo))==mid: #if they are equal

return pivot #return pivot value

else:

return ModQuickSort(maximum,mid-getLength(lo)-1)

#adds item to beginning of list

def prepend(L,x):

if L.head is None:

L.head = Node(x)

L.tail = Node(x)

else:

newNode = Node(x)

newNode.next = L.head

L.head = newNode

#bubble sorts list

def bubblesort(L):

if IsEmpty(L): #if list is empty return

return

exchanged = False #checks for exchanges

temp=L.head

while temp.next is not None:

if temp.item>temp.next.item: #compares adjacent elements

temp2=temp.item #stores that element in temp

temp.item=temp.next.item #swaps

temp.next.item=temp2

exchanged= True

temp=temp.next #noves to next

if exchanged == True:

bubblesort(L) #recursively goes though list

#creates lists

L1 = List()

for i in range(5): #populates list

t = random.randrange(100)

Append(L1,t)

L2 = List()

for i in range(5):

t = random.randrange(100)

Append(L2,t)

L3 = List()

for i in range(5):

t = random.randrange(100)

Append(L3,t)

L4= List()

for i in range(5):

t = random.randrange(100)

Append(L4,t)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

Print(L1)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print('printing merge sorted list')

L1.head = mergeSort(L1.head)

Print(L1)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print('printing bubble sorted list:')

bubblesort(L2)

Print(L2)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print('before quicksort')

Print(L3)

print('quicksorting list')

C = copy.copy(L3)

quicksort(C)

Print(C)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

mid = getLength(L4)//2

print('list before modquicksort')

Print(L4)

print('mod quicksorting list')

print(ModQuickSort(L4,mid))

print('mergesort median')

print(getElementAt(L1,Median1(L1)))

print('bubblesortlist')

Print(L2)

print('bubblesort median')

print(getElementAt(L2,Median2(L2)))

print('quicksort list')

Print(L3)

print('quicksort median')

print(getElementAt(L3,Median3(L3)))