CS 2302 Data Structures

Spring 2019

MW 10:30-11:50 in CCSB 1.0202

LAB # 2

Claudio Angel Garcia

Instructor: Olac Fuentes

TA: Anindita Nath, Maliheh Zargaran

IA: Eduardo Lara

Peer leader: Erick

Introduction

For this lab we are trying to sort a given non-native Python list that has random numbers in it, and then after sorting want to return the median of this list. We want to achieve this by trying different methods of sorting, bubble sorting, merge sorting, and quick sorting, this to compare the number of comparison made by each method and the running times to contrast which one is better and faster.

Solution

To be able to solve these problems I had to think out of the box and out of what we normally know (Array lists, linked lists) because we had to use a list made of nodes, and that was basically the where the difficulty of this lab was because the behavior was a little different from a normal linked list.

For the bubble sort I had to go with the standard way to solve it, by comparing the first node’s item with the second node’s item, and if the first node is larger than the next one then swap them just how bubble sorting suppose to be. I did this with two while loops, one with a that ends when the list is already sorted and another one that basically goes trough the entire list swapping elements. After that the list is sorted and I use the Median method to call it and return the median element of the already sorted list.

For the merge sort I had to think how the divide and conquer algorithm could be implemented for this type of list. So first, I have to split the list in two different sublists, and continue to do this up until every element is alone, then this is when the sorting conquer and sorting starts. The program starts to compare each sublist from the top down up until we got the final sorted sorted list. I used a method that helped me to split the list and have one half of the list and use that in a recursive call that keeps doing the sorting in that half and so on. Another method I utilize was one that only did comparisons and returns the variables ‘a’ or ‘b’ depending on their numerical values, this by comparing them, this merges the lists. Finally, on the main merge method I just do recursive calls applying the sorting for the left side and right side of the list and finally returning a sorted list. After that the list is sorted and I use the Median method to call it and return the median element of the already sorted list.

For quick sort it uses the same idea of the divide and conquer algorithm, except this one picks an element to be a pivot and it does partitions depending on the pivot. There’re a few ways to pick a pivot, for this case a picked the first element of my list (L.head) to be the pivot.

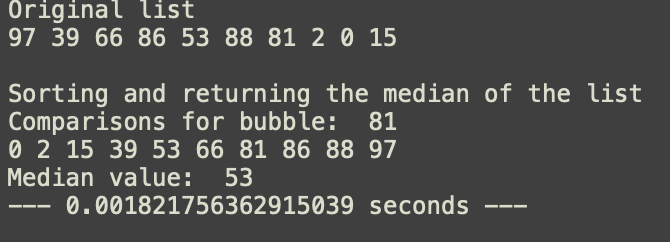
Then I resituate the other elements of the list on each side of the pivot, so that on one side are all minors that the pivot, and the other side have the larger than the pivot. The elements equal to the pivot can be placed both to their right and to their left, depending on the desired implementation. At this moment, the pivot occupies exactly the place that will correspond to him in the ordered list. Repeat recursively for each sublist as long as they contain more than one element. Once this process is finished all the elements will be sorted.

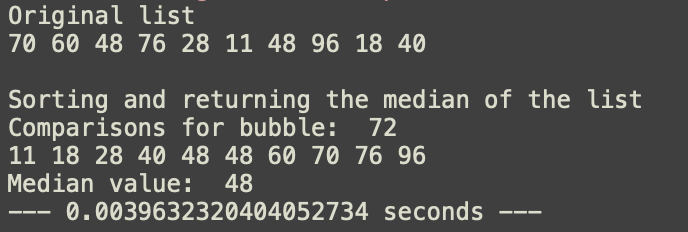
Experimentation

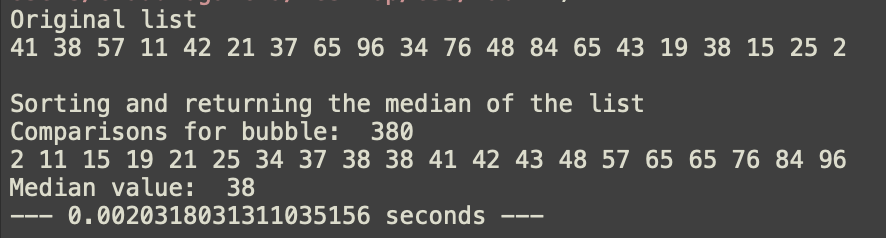
* Bubble sort running time = О(n2)

First two images n = 10

Third image n = 20

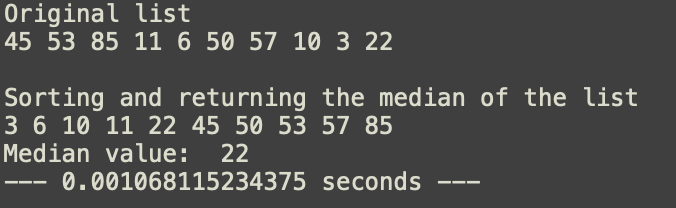
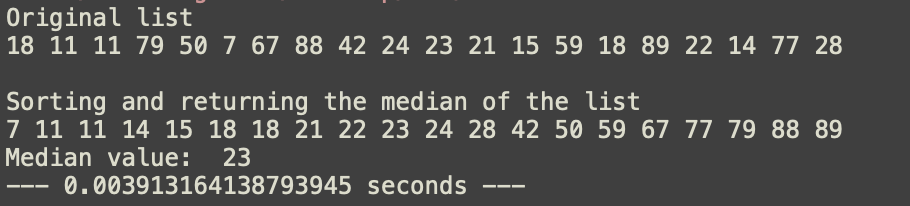
As we can see the number of comparisons matches with the running time.





* Merge sort running time = O(n log n)

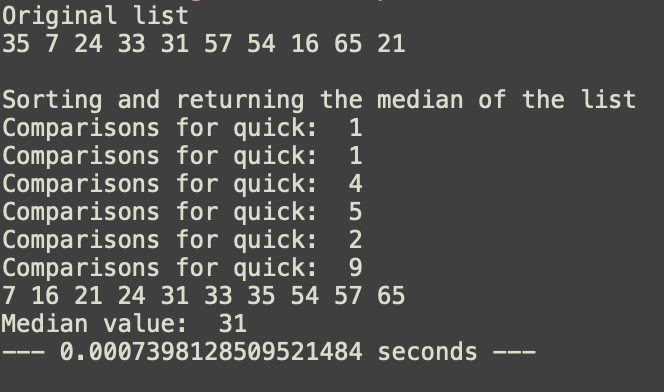
First image n = 10

Second image n = 20

* Quick sort running time = O(log n)

First image n = 10

Second image n = 20

Last number is the total number of comparisons

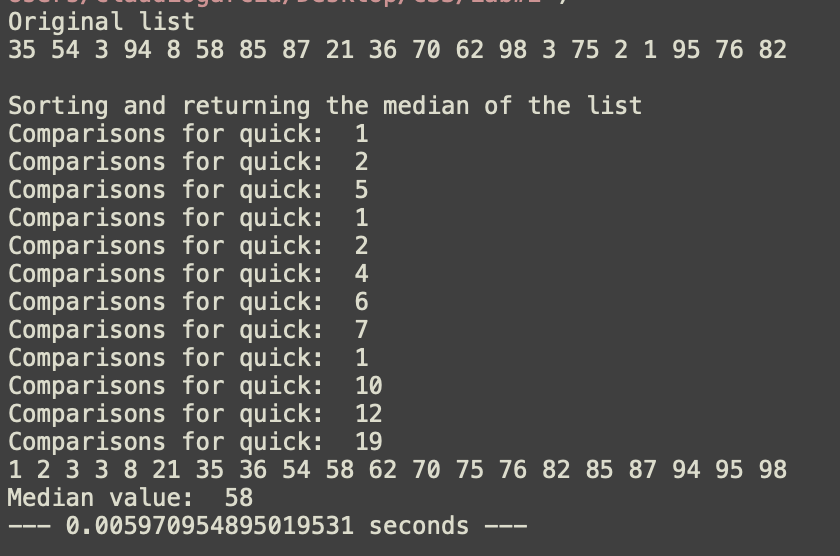


Chart that represents which running time is better

https://www.quora.com/Which-is-faster-quick-sort-or-bubble-sort-and-why

Conclusion

With this lab I learned how to implement different types of sorting using linked lists in Python. I also learned more about the running times and how they actually are shown in a program.

Appendix

#!/usr/bin/env python3

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

"""

Created on Fri Feb 22 18:58:58 2019

@author: claudiogarcia

CS 2302 Data Structures Spring 2019

MW 10:30-11:50 in CCSB 1.0202

Instructor: Olac Fuentes

TAs: Anindita Nath, Maliheh Zargaran. IA: Eduardo Lara

Peer Leader: Erick Macik

Lab 2

Claudio Garcia 80628314

The purpose of this program is to sort a list with random values and

then sort them using 3 different methods for sorting, bubble sort, merge sort,

and quick sort.

"""

## UNCOMMENT OR COMMENT METHODS TO TRY DIFFERENT SORTS IN MEDIAN METHOD

import random

import copy

import time

#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 getTail(L):

cur = L

while (cur != None and cur.next != None):

cur = cur.next

return cur

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 Prepend(L, x):

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

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

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()

############################### Lab 2 #################################

def GetLength(L):

temp = L.head

counter = 0

while temp is not None:

counter +=1

temp = temp.next

return counter

def getMiddle(h):

##Base case

if h == None:

return h

fast = h.next

slow = h

## Move fast by two and slow by one

## Finally slow will point to middle node

while fast is not None:

fast = fast.next

if(fast!=None):

slow = slow.next

fast=fast.next

return slow

def sortedM(a, b, c):

result = None;

if a == None:

return b

if b == None:

return a

if a.item <= b.item:

result = a

result.next = sortedM(a.next, b, c)

else:

result = b

result.next = sortedM(a, b.next, c)

return result

##################### MEDIAN METHOD ##################

def Median(L):

C = copy.deepcopy(L)

#C.head = mergeSort(C.head) ##Sorting

#bubble(C)

#quickSort(C)

Print(C)

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

##################### MEDIAN METHOD ##################

def ElementAt(L, n):

if GetLength(L)%2 == 1:

for i in range(n):

L.head = L.head.next

return L.head.item

else:

for i in range(n-1):

L.head = L.head.next

mid = L.head.item

L.head = L.head.next

return mid

############################### Lab 2 #################################

def printMiddle(h):

slow\_ptr = h

fast\_ptr = h

if h!= None:

while fast\_ptr != None and fast\_ptr.next != None:

fast\_ptr = fast\_ptr.next.next

slow\_ptr = slow\_ptr.next

return slow\_ptr

def bubble(L):

c = 0

unsorted = True

while unsorted:

temp = L.head

unsorted = False

while temp.next is not None:

if temp.item > temp.next.item:

t = temp.item

temp.item = temp.next.item

temp.next.item = t

unsorted = True

temp = temp.next

c +=1

print('Comparisons for bubble: ', c)

def mergeSort(h):

c = 0

## Base case : if head is null

if h == None or h.next == None:

return h

middle = getMiddle(h)

nextofmiddle = middle.next

middle.next = None

#Apply mergeSort on left list

left = mergeSort(h)

#Apply mergeSort on right list

right = mergeSort(nextofmiddle)

#Merge the left and right list

sortedlist = sortedM(left, right, c)

return sortedlist

def quickSort(L):

counter = 0

if GetLength(L) >1:

pivot=L.head.item

L1 = List()

L2 = List()

t = L.head.next

#populates two different lists depending on .item value

while t != None:

if t.item<pivot:

Append(L1, t.item)

else:

Append(L2, t.item)

t = t.next

counter +=1

quickSort(L1)

quickSort(L2)

#appending if empty to sort

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('Comparisons for quick: ', counter)

def createList(L, num):

for i in range(num):

Append(L,random.randrange(100))

L = List()

createList(L, 20)

print('Original list')

Print(L)

print(' ')

start\_time = time.time()

print('Sorting and returning the median of the list')

print('Median value: ', Median(L))

print("--- %s seconds ---" % (time.time() - start\_time))

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.

- Claudio Garcia