Toshiki Kamio

CS 2302 Lab#2

Spring 2019

1. Introduction

This report is for Spring 2019 CS2302 Lab2. The task is to implement four algorithms for finding the median of a list of integers, using objects of the List class described in class, and compare running time for various list lengths. To generate data to test methods, a method that receives an integer n and builds and returns a list of random integers of length n is also required.

The algorithms to compare are the following:

1. Sort list using bubble sort, then return the element in the middle.
2. Sort list using merge sort, then return the element in the middle.
3. Sort list using quicksort, then return the element in the middle.
4. Implement a modified version of quicksort that makes a single recursion call instead of the two made by normal quicksort, processing only the sublist where the median is known to reside.

1. Proposed solution design and implementation

The solution to find the median is to sort the list and return the element in the middle. Pseudo code can be,

Def Median(L):

C = Copy(L)

Sort(C)

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

1. Experimental results

-Experiment environment is as below.

Computer: ASUS Vivobook 15

Processor: AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx 2.00GHz

Memory: 8.00 GB (6.95 GB usable)

OS: Windows 10 Home

Python version: Python 3.7

-The experiment is to apply algorithms in introduction to list with various number of elements. This report describes the cases of n = 10, 100, 1000, 2000, 3000.

-The big-O running time with respect to n for every method are as following.

def PrintNodes()

def PrintNodesReverse()

def IsEmpty

def Append()

def Print()

def PrintRec()

def Remove()

def PrintReverse()

def GetLength()

def GenerateList()

def Copy()

def ElementAt()

def BubbleSort()

def MergeSort()

def Merge()

def QuickSort()

def Partition()

def ModifiedQuickSort()

-The experiment results are illustrated in graph and summarized in table as following.

\* Bubble sort is applied secondary axis because it’s number of comparisons is enormous compared with other algorithms.

-Number of Comparisons

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of Element | 10 | 100 | 1,000 | 2,000 | 3,000 |
| Bubble Sort | 69 | 7,703 | 808,059 | 3,281,706 | 7,414,866 |
| Merge Sort | 22 | 542 | 8,716 | 19,415 | 30,913 |
| Quicksort | 21 | 640 | 11,473 | 24,118 | 37,954 |
| Modified Quicksort | 18 | 444 | 10,321 | 18,604 | 30,725 |

-Elapsed Time (sec)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number of Element | 10 | 100 | 1,000 | 2,000 | 3,000 |
| BubbleSort | - | 0.0156 | 0.6161 | 1.9437 | 4.3691 |
| MergeSort | - | 0.0065 | 0.0308 | 0.1159 | 0.1667 |
| QuickSort | - | - | 0.0102 | 0.0387 | 0.1342 |
| ModifiedQuickSort | - | - | 0.0104 | 0.0323 | 0.0391 |

\* ”-” means too small to appear

1. Conclusion

What I learned from this project was how to implement each sorting to linked list data structure. Also, I confirmed that each algorithm generally follows as theory. For example, bubble sort increased the number of comparisons as O(n^2), which is showed in graph. Quicksort had the greatest number of comparisons among remaining algorithms. While merge sort had fewer number of comparisons at n = 1,000 than modified quicksort, modified quicksort had the fewest number of comparisons in other cases.

1. Appendix

Source codes are following.

'''

CS2302 Spring 2019

Author: Toshiki Kamio

Assignment: Lab#2

Instructor: Olac Fuentes

T.A.:

Anindita Nath

Maliheh Zargaran

Date of last modified: 3/4/2019

Purpose of Program:

This program writes several algorithms for finding the median of a list

of integers, using objects of List class described in class, and compare

their running time for various list length. In addition, a method that

receives an integer n and builds and return a list of random integers of

length n is written to generate data to test algorithms.

'''

import random

import time

##########################################################

#Code given in class

#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 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()

#Code given in class end

########################################################

# Get the length of list

def GetLength(L):

if L.head == None:

return 0

length = 1;

temp = L.head

while temp.next != None:

temp = temp.next

length = length + 1

return length

# Generate a list of length n

def GenerateList(L, n):

for i in range(n):

Append(L, random.randint(0, 100000))

# Copy a list

def Copy(L1):

L2 = List()

temp = L1.head

while temp is not None:

Append(L2, temp.item)

temp = temp.next

return L2

# Return the item at n of list.

# n starts from 0, 1, 2...

def ElementAt(L, n):

if L.head is None:

return

temp = L.head

for i in range(n):

temp = temp.next

return temp.item

# Implement bubble sort

# Return the number of comparison

def BubbleSort(L):

if L.head == None or L.head.next == None:

return

comparison = 0

swapped = True

prev = L.head

current = L.head.next

while swapped == True:

# List may be sorted and next pass not needed

swapped = False

prev = L.head

current = L.head.next

while current.next is not None:

if prev == L.head:

comparison += 1

if L.head.item > current.item:

prev.next = current.next

current.next = prev

L.head = current

swapped = True # Next pass still needed

current = current.next.next

if current.next == L.tail:

comparison += 1

if current.item > current.next.item:

prev.next = current.next

current.next = None

prev.next.next = current

L.tail = current

swapped = True # Next pass still needed

prev = prev.next

break

comparison += 1

if current.item > current.next.item:

prev.next = current.next

current.next = current.next.next

prev.next.next = current

swapped = True # Next pass still needed

prev = prev.next

prev = prev.next

current = current.next

return comparison

# Implement merge sort

# Return the number of comparison

def MergeSort(L):

if L.head is None:

return 0

comparison = 0

length = GetLength(L)

if length > 1:

# Merge sort the first half

firstHalf = List()

temp = L.head

count = 0

while count < length // 2:

Append(firstHalf, temp.item)

temp = temp.next

count += 1

comparison += MergeSort(firstHalf)

# Merge sort the second half

secondHalf = List()

while temp is not None:

Append(secondHalf, temp.item)

temp = temp.next

comparison += MergeSort(secondHalf)

# Merge firstHalf with secondHalf into list

comparison += Merge(firstHalf, secondHalf, L)

return comparison

# Method to merge two sorted list

def Merge(L1, L2, Ltemp):

Ltemp.head = None

Ltemp.tail = None

temp1 = L1.head

temp2 = L2.head

comparison = 0

while temp1 is not None and temp2 is not None:

comparison += 1

if temp1.item < temp2.item:

Append(Ltemp, temp1.item)

temp1 = temp1.next

else:

Append(Ltemp, temp2.item)

temp2 = temp2.next

while temp1 is not None:

Append(Ltemp, temp1.item)

temp1 = temp1.next

while temp2 is not None:

Append(Ltemp, temp2.item)

temp2 = temp2.next

return comparison

# Implement quicksort

# Return the number of comparison

def QuickSort(L):

if L.head is None:

return 0

comparison = 0

Low = List()

High = List()

pivot = L.head

L.head = L.head.next

pivot.next = None

comparison += Partition(pivot, L, Low, High)

comparison += QuickSort(Low)

comparison += QuickSort(High)

L.head = pivot

L.tail = pivot

if Low.tail is not None:

Low.tail.next = pivot

L.head = Low.head

if High.head is not None:

pivot.next = High.head

L.tail = High.tail

return comparison

# Method to implement partition in quicksort

def Partition(pivot, L, Low, High):

if L.head is None:

return 0

temp = L.head

comparison = 0

while temp is not None:

comparison += 1

if temp.item <= pivot.item:

Append(Low, temp.item)

else:

Append(High, temp.item)

temp = temp.next

return comparison

# Implement modified quicksort that makes single recursive call

# instead of the two made by the normal quicksort, processing only

# the sublist where the median is known to reside.

def ModifiedQuickSort(L):

if L.head is None:

return 0

comparison = 0

Low = List()

High = List()

pivot = L.head

L.head = L.head.next

pivot.next = None

comparison += Partition(pivot, L, Low, High)

L.head = pivot

L.tail = pivot

if GetLength(Low) == GetLength(High):

pass

elif GetLength(Low) > GetLength(High):

comparison += QuickSort(Low)

else:

comparison += QuickSort(High)

if Low.tail is not None:

Low.tail.next = pivot

L.head = Low.head

if High.head is not None:

pivot.next = High.head

L.tail = High.tail

return comparison

# Find the median by bubble sort

def Median1(L):

C = Copy(L)

BubbleSort(C)

size = GetLength(C)

if size % 2 == 1:

return ElementAt(C, size//2)

else:

return (ElementAt(C, size//2) + ElementAt(C, size//2 - 1)) // 2

# Find the median by merge sort

def Median2(L):

C = Copy(L)

MergeSort(C)

size = GetLength(C)

if size % 2 == 1:

return ElementAt(C, size//2)

else:

return (ElementAt(C, size//2) + ElementAt(C, size//2 - 1)) // 2

# Find the median by quicksort

def Median3(L):

C = Copy(L)

QuickSort(C)

size = GetLength(C)

if size % 2 == 1:

return ElementAt(C, size//2)

else:

return (ElementAt(C, size//2) + ElementAt(C, size//2 - 1)) // 2

# Find the median by modified quicksort

def Median4(L):

C = Copy(L)

ModifiedQuickSort(C)

size = GetLength(C)

if size % 2 == 1:

return ElementAt(C, size//2)

else:

return (ElementAt(C, size//2) + ElementAt(C, size//2 - 1)) // 2

test\_Length = 1000

test = List()

GenerateList(test, test\_Length)

# Measure time elapsed

start1 = time.time()

Median1(test)

elapsed\_time1 = time.time() - start1

start2 = time.time()

Median2(test)

elapsed\_time2 = time.time() - start2

start3 = time.time()

Median3(test)

elapsed\_time3 = time.time() - start3

start4 = time.time()

Median4(test)

elapsed\_time4 = time.time() - start4

# Report time elapsed

print("elapsed\_time using bubble sort:{:.6g}".format(elapsed\_time1) + "[sec]")

print("elapsed\_time using merge sort:{:.6g}".format(elapsed\_time2) + "[sec]")

print("elapsed\_time using quicksort:{:.6g}".format(elapsed\_time3) + "[sec]")

print("elapsed\_time using modified quicksort:{:.6g}".format(elapsed\_time4) + "[sec]")

C1 = Copy(test)

C2 = Copy(test)

C3 = Copy(test)

C4 = Copy(test)

# Report the number of comparisons

print("The number of comparison using bubble sort is: ", BubbleSort(C1))

print("The number of comparison using merge sort is: ", MergeSort(C2))

print("The number of comparison using quicksort is: ", QuickSort(C3))

print("The number of comparison using modified quicksort is: ", ModifiedQuickSort(C4))

# Report median by each sorting

print("The median using bubble sort is: ", Median1(test))

print("The median using merge sort is: ", Median2(test))

print("The median using quicksort is: ", Median3(test))

print("The median using modified quicksort is: ", Median4(test))

1. Academic Honesty Certification

“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”

Toshiki Kamio