Lab 2: Search Algorithms

CS 2303 Data Structures

Laura Berrout

ID# 80607326

February 22, 2019

# Introduction

The median of a list L is the element a such that half of the elements in L are smaller than a and half of them are larger than a. For example, the median of list L = f20; 10; 45; 1; 12g is 12, since 12 is greater than 1 and 10 and smaller than 20 and 45. An easy way to and the median is to sort the list and return the element in the middle:

def Median(L):

C = Copy(L)

Sort(C)

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

Your task for this lab is to:

* Implement several algorithms for finding the median of a list of integers, using objects of the List class described in class.
* Compare their running times (measured as the number of comparisons each algorithm makes) for various list lengths.
* To generate data to test your methods, write a method that receives an integer n and builds and returns a list of random integers of length n.
* The algorithms to compare are the following:

1. Bubble Sort
2. Merge Sort
3. Quick Sort
4. Recursive Quick Sort

# Methodology

To start, the methodology was setup as follows:

1. Create a list class
2. Create a list of objects
3. Generate a list of random numbers
4. Generate a list with different lengths
5. To find the mean, first to sort the list and then find the middle element of the list. To sort, we’ll use the next algorithms:
   1. Bubble sort
   2. Merge sort
   3. Quick sort
   4. Recursive Quick sort.
6. Find the median by getting the middle element by n/2.
7. Find the running time of each algorithms.

1) Bubble Sort

## Proposed solution design and implementation

The algorithm followed was the next one:

1. First pass:
   1. Compare the first two elements
   2. Swap if the first is greater than the second
   3. Check the next two (second and third element)
   4. Return to step a until reach at the end.
2. Second pass:
   1. Compare the first two elements until reaches the end
   2. If it didn’t do any swap, the list is sorted. If it did, does another pass.

For the function was created a Boolean variable to stop the next while loop. Inside the while loop a counter is started and also a temporally variable that is equals to the head of the list. Then while the *tmp.next* is not None, and the item in the tmp variable is less than the next item (tmp.next.item), another temporarily variable (save) is going to store the current item, the current item is equal to the next and the next is equal to the stored variable, and count + 1; if not then we only go to the next item. Once the while loop is finished, that is, tmp.next is None, check if the counter is 0, if true then sorted is also true and the while loop is terminated.

## Experimental results

1. First tested to get to compare the first and second element and swap them if the first is greater than the second.
2. Adding a while loop, changes the entire list by swapping the first with the second, the second with the third, the third with the forth, and so on.
3. Adding a count variable to count how many swaps do in the loop with help us determine when the list is fully sorted. When the count is equal to 0, we can say the list is sorted.

Add another while loop to pass as many times as necessary though the list until is sorted and changing *tmp = self.head* to be call inside the second while loop. That way we start again at the beginning of the list.

The result is shown on the next figure:

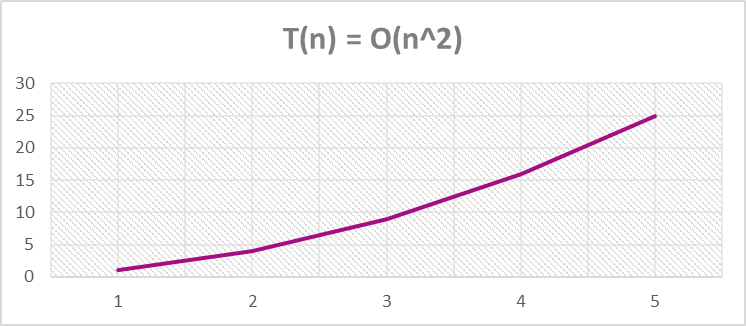


## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 0 recall, so a = 0 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +n^2 (2 while loop) | T(n) = aT(f(n)) + g(n)  T(n) = 0\*(T(n-1)) + n^2  Then:  **T(n) = O(n^2)** |

Running time: 1554503577.865304 ns



2) Merge Sort

## Proposed solution design and implementation

The algorithm of Merge Sort is the next:

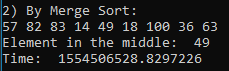
1. Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).
2. Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

For Merge sort, was created several functions to performed different methods:

1. Length. Returns the length of the list, with a counter takes
2. Split. Split the list by 2, this helps to divide the list to smaller list, by the middle so can be sorted later.
3. Sort. Takes two lists and returns the Sorted list. Takes the smaller list and sort them and merge them into one list and returns this list.
4. MergeSort. Call the functions above and returns the sorted list.

## Experimental results

Unfortunately, was missing a last sort, since merges the lists, doesn’t not sorts later the sorted list, so a last MergeSort had to be call to the “Sorted list” and gives the final sorted list. The results are shown in the next figure:

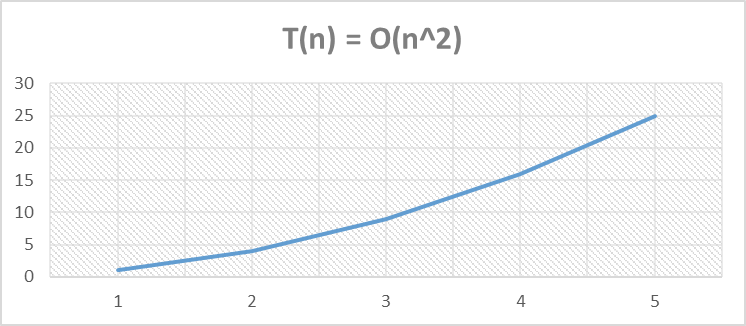


## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 0 recall, so a = 0 2. Size of the list: n, so T(n-1) 3. Recursive at the end, +n^2 (2 while loop) | T(n) = aT(f(n)) + g(n)  T(n) = 0\*(T(n-1)) + n^2  Then:  **T(n) = O(n^2)** |

Running time: 1554506540.0098693 ns



3) QuickSort

## Proposed solution design and implementation

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

To create the function, first was tested with an array (because of low understanding to create a quicksort algorithm, and then try to implement to the list).

First was created a partition function, do pick the last and first and compared to the pivot.

## Experimental results

Was tested with the same list to get the same result as the other algorithm, and the result are as follows:

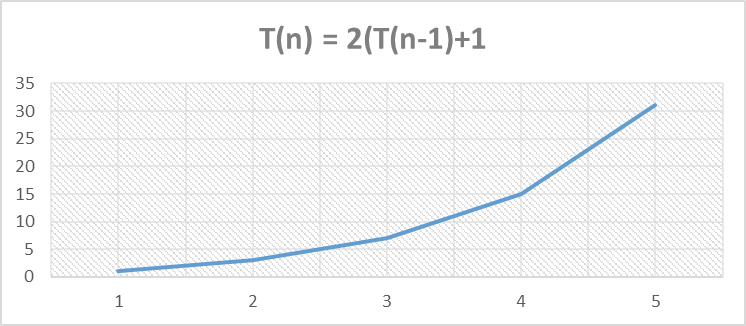


## Time of execution

To get the running complexity and the Big-O notation, the steps were as follows:

|  |  |
| --- | --- |
| 1. 2 recall, so a = 2. 2. Size of the list: n, so T(n/2) 3. Recursive at the end, +1 | T(n) = aT(f(n)) + g(n)  T(n) = 2(T(n/2)) + 1 |
|  | Then: T(n+1)  **T(n) = O(log(n))** |

Running time: 0.015623807907104492 ns



# Appendix

## Source code:

"""

Course: Data Structures CS2302

Author: Laura Berrout

Assignment: Lab #2

Instructor: Dr. Olac Fuentes

T.A.: Anindita Nath

Date of last modification: 02/22/2019

Purpose: implement several algorithms for

nding the median of a list of

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

their running times (measured as the number of comparisons each algorithm

makes) for various list lengths. To generate data to test your methods,

write a method that receives an integer n and builds and returns a list of

random integers of length n.

"""

import random

import time

#List of class objects

class Node(object):

def \_\_init\_\_(self,item,next=None):

self.item = item

self.next = next

class List(object):

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

self.head = None

self.tail = None

def IsEmpty(L):

return L.head == None

def Append(L,x):

#Inserts at the end

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

L.tail.next = Node(x)

L.tail = L.tail.next

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 Push(self,new\_data):

#Inserts at the beginning

new\_node = Node(new\_data)

new\_node.next = self.head

self.head = new\_node

def Insert(self,new\_data,location): #inserts an element in a specific location

position = 2

tmp = self.head

new\_node = Node(new\_data)

while position < location:

position +=1

tmp = tmp.next

tmp2 = tmp.next

tmp.next = new\_node

new\_node.next = tmp2

def Print(L):

temp = L.head

while temp is not None:

print(temp.item, end=' ')

temp = temp.next

print()

def Copy(L):

temp = L.head

new\_list = List()

while temp.next is not None:

Append(new\_list,temp.item)

temp = temp.next

Append(new\_list,temp.item)

return new\_list

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Sort Algorithms \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#Create a list of size n with random numbers

A = List()

n = 9

for i in range(n):

rand = random.randrange(101)

Append(A,rand)

print("Random List:")

Print(A)

print()

AL = []

def ListArray(AL,L):

tmp = L.head

i=0

while tmp is not None:

AL.append(tmp.item)

tmp = tmp.next

def Getmiddle(L,n):

tmp = L.head

mid = n//2

i=0

while tmp.next is not None:

if i==mid:

print(tmp.item)

return

else:

i+=1

tmp = tmp.next

#Getmiddle(A,n)

#1) Bubble Sort

def BubbleSort(self):

sorted = False

while sorted != True:

count = 0

tmp = self.head

while tmp.next is not None:

if (tmp.item > tmp.next.item):

save = tmp.item

tmp.item = tmp.next.item

tmp.next.item = save

count +=1

tmp = tmp.next

if count == 0:

sorted = True

B = Copy(A)

print("1) By Bubble Sort:")

BubbleSort(B)

Print(B)

print("Element in the middle: ", end=' ')

Getmiddle(B,n)

print()

#2) Merge Sort

def Length(L):

tmp = L.head

count = 0

while tmp.next is not None:

count +=1

tmp = tmp.next

return count

def Split(self): #divides a list by half and returns two lists

tmp = self.head

n = Length(self) #n is the number of elements in the list

if n==1:

return self

else:

mid = n//2 +1 #midpoint of the list

L1 = List()

L2 = List()

count = 0

while count < mid:

Append(L1,tmp.item)

tmp = tmp.next

count +=1

tmp = self.head

for i in range(mid+1):

L2.head = tmp

tmp = tmp.next

return L1,L2

def MergeSort(self):

n = Length(self)

left=List()

right=List()

Sorted = List()

tmp = self.head

if n<1:

return self

elif n==1:

Append(left,tmp.item)

tmp = tmp.next

Append(right,tmp.item)

else:

left, right = Split(self)

MergeSort(left)

MergeSort(right)

Sorted = Sort(left,right)

return Sorted

def Sort(L1,L2): #sorted the two lists and returns one list

tmp1 = L1.head

tmp2 = L2.head

Sorted = List()

while tmp1 is not None and tmp2 is not None:

if tmp1.item <= tmp2.item:

Append(Sorted,tmp1.item)

tmp1 = tmp1.next

else:

Append(Sorted,tmp2.item)

tmp2 = tmp2.next

while tmp1 is not None:

if tmp1 is not None:

Append(Sorted,tmp1.item)

tmp1 = tmp1.next

while tmp2 is not None:

if tmp2 is not None:

Append(Sorted,tmp2.item)

tmp2 = tmp2.next

return Sorted

def Sorttt(L):

tmp = L.head

while tmp.next is not None:

if tmp.item>tmp.next.item:

MergeSort(L)

tmp = tmp.next

return L

B = Copy(A)

Sorted = List()

start = time.time()

Sorted = MergeSort(B)

Sorted = MergeSort(Sorted)

Sorted = MergeSort(Sorted)

Sorted = MergeSort(Sorted)

#Sorted = Sorttt(Sorted)

Sorted = MergeSort(Sorted)

print("2) By Merge Sort:")

Print(Sorted)

print("Element in the middle: ", end=' ')

Getmiddle(Sorted,n)

end = time.time()

print("Time: ", end=' ')

print(end)

print()

#3) QuickSort

def QuickSort(L,n):

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

return L

tmp = L

pivot = L.head.item

left = 0 #left pointer

right = n #right pointer

lelement = tmp.head.item #first element

relement = tmp.tail.item #last element

if (lelement > pivot):

swapping = lelement

lelement = relement

relement = swapping

if (relement < pivot):

swapping = relement

relement = lelement

lelement = swapping

return tmp

def partition(L,low,high):

i = ( low-1) # index of smaller element

pivot = L[high] # pivot

for j in range(low , high):

if L[j] <= pivot:

i = i+1

L[i],L[j] = L[j],L[i]

L[i+1],L[high] = L[high],L[i+1]

return ( i+1 )

def Sort(L,low,high):

if low < high:

mid = partition(L,low,high)

Sort(L, low, mid-1)

Sort(L, mid+1, high)

D = Copy(A)

print("3) By QuickSort:")

#QuickSort(D,n)

#Print(D)

DL =[]

ListArray(DL,D)

Sort(DL,0,len(DL)-1)

Dsort = List()

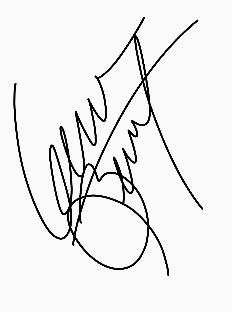
for i in range(n):

Append(Dsort,DL[i])

Print(Dsort)

print("Element in the middle: ", end=' ')

Getmiddle(Dsort,n)



Laura Berrout, February 22, 2019

“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.”