Yahir Rivas

Instructor: Dr. Olac Fuentes

CS 2302

**Lab 2 Report**

For Lab 2 I was asked to submit the implementation of 4 different sorting methods. The methods we had to implement were bubble sort, mergesort, quicksort, and another version of quicksort with only one recursive call. The goal of this lab was identifying the running times and differences between different sorting methods.

**Bubble Sort**

I started with the implementation of bubble sort which consists of comparing every element of the list with the next one swapping their positions until all of them are in the correct order. The way I implemented this was making a while loop that checks if the element in the current index is more than the element in the next position then swapping them. The Big O function of this method is O(n2).

**List Length: 10**

32 42 99 20 49 96 39 6 84 84

6 20 32 39 42 49 84 84 96 99

The median element is: 49

29 comparisons

**List Length: 25**

7 43 95 43 89 53 94 64 52 65 35 91 81 52 41 20 4 58 62 46 13 83 6 71 89

4 6 7 13 20 35 41 43 43 46 52 52 53 58 62 64 65 71 81 83 89 89 91 94 95

The median element is: 53

182 comparisons

**List Length: 50**

31 68 7 67 76 50 38 95 97 63 39 41 98 88 19 65 95 75 70 59 13 37 93 68 8 6 57 71 88 16 6 66 7 25 35 39 36 56 95 65 32 5 94 7 39 81 85 31 74 89

5 6 6 7 7 7 8 13 16 19 25 31 31 32 35 36 37 38 39 39 39 41 50 56 57 59 63 65 65 66 67 68 68 70 71 74 75 76 81 85 88 88 89 93 94 95 95 95 97 98

The median element is: 59

688 comparisons

**Quick Sort**

For this method I used recursion, first chose a pivot and created two lists to separate the elements in a list. All elements smaller than pivot are stored in one list while larger items than the pivot are stored on another list. Then make a list merging all the elements, for this the pivot can either be larger than any other number, smaller than any other number or the most likely outcome, be in between the list. So we make 3 cases for this, if it is smaller than any other number, just add the pivot to the left list, if it is larger add the pivot as the first element of the right list, or if it is in-between the list then add the pivot to left list and connect it to the right list. The Big O function of this method is O(nlogn).

**List Length: 10**

14 54 87 24 65 1 2 56 13 22

14 54 87 24 65 1 2 56 13 22

The median element is: 24

42 comparisons

**List Length: 25**

84 4 13 88 62 78 36 52 4 53 0 21 2 16 92 14 47 62 35 29 15 94 54 3 54

84 4 13 88 62 78 36 52 4 53 0 21 2 16 92 14 47 62 35 29 15 94 54 3 54

The median element is: 36

216 comparisons

**List Length: 50**

13 71 31 30 10 30 91 11 86 84 24 65 90 50 47 40 24 80 97 77 97 76 76 85 40 100 1 52 79 48 7 51 8 16 21 86 63 56 32 45 67 41 55 31 54 68 0 5 89 16

13 71 31 30 10 30 91 11 86 84 24 65 90 50 47 40 24 80 97 77 97 76 76 85 40 100 1 52 79 48 7 51 8 16 21 86 63 56 32 45 67 41 55 31 54 68 0 5 89 16

The median element is: 51

486 comparisons

**Modified Quick Sort**

This method is like quick sort, we still divide the list into smaller or larger numbers than the pivot. The difference is that the method now contains another parameter that is used to get the length of the list. Now we have 2 cases, if the length of the smaller list is greater than the length, then we call the method with the smaller list and the length. And the second case is when the length of the list is the same it returns the pivot which will now be the median number.

**List Length: 10**

83 11 31 100 22 58 74 100 69 28

83 11 31 100 22 58 74 100 69 28

The median element is: 69

53 comparisons

**List Length: 25**

63 34 81 96 13 70 45 97 24 98 9 49 58 24 99 5 66 74 72 59 3 17 97 22 2

63 34 81 96 13 70 45 97 24 98 9 49 58 24 99 5 66 74 72 59 3 17 97 22 2

The median element is: 58

130 comparisons

**List Length: 50**

89 47 52 38 73 24 34 9 34 10 74 72 81 44 75 98 64 69 77 14 11 46 85 31 0 14 47 27 13 94 14 62 58 68 38 92 87 69 15 25 36 4 75 55 26 98 96 67 47 25

89 47 52 38 73 24 34 9 34 10 74 72 81 44 75 98 64 69 77 14 11 46 85 31 0 14 47 27 13 94 14 62 58 68 38 92 87 69 15 25 36 4 75 55 26 98 96 67 47 25

The median element is: 47

372 comparisons

from random import random

# List 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=' ')

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()

def ElementAt(L, n):

if GetLength(L)<n:

return None

else:

temp = L.head

for i in range(n):

temp = temp.next

return temp.item

def GetLength(L):

temp = L.head

counter = 0 #counter

while temp is not None:

counter = counter + 1

temp = temp.next

return counter #return the length

def Preppend(L, n):

if IsEmpty(L):

L.head = Node(n)

def Copy(L):

C = List()

temp = L.head

while temp is not None:

Append(C, temp.item) # add each item to list C

temp = temp.next

return C

def bubbleSort(L):

global count

change = True

while change: # checks for swap in previous iteration

t = L.head

change = False

count = count + 1

while t.next is not None:

if t.item > t.next.item: # if current element is more than next, then swap

temp = t.item

t.item = t.next.item

t.next.item = temp

change = True

count = count + 1

t = t.next

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

def QuickSort(L):

global count

RightList = List() # larger numbers list

LeftList = List() # Smaller numbers List

if GetLength(L) <= 1: # if length is less or equal to 1 change nothing

return L

p = L.head.item # pivot

Left = List() # create a list for left side

Right = List() # create a list for right side

temp = L.head.next

while temp is not None:

if temp.item < p: # if current element is smaller than pivot

count = count + 1

Append(Left, temp.item) # add element to left side

else: # if current element is equal or larger than pivot

count = count + 1

Append(Right, temp.item) # add element to right side

temp = temp.next

LeftList = QuickSort(Left)

RightList = QuickSort(Right)

SortedList = List()

if IsEmpty(LeftList): # if pivot is the smallest number

Append(SortedList, p) # add pivot to the sorted list

SortedList.head.next = RightList.head

SortedList.tail = RightList.tail

return SortedList

elif IsEmpty(RightList): # if pivot is the largest number

Append(LeftList, p)

return LeftList

else: # pivot in middle

Append(LeftList, p)

LeftList.tail.next = RightList.head

LeftList.tail = RightList.tail

return LeftList

def QuickSortM(L, n):

global count

p = L.head.item

Left = List()

Right = List()

temp = L.head.next

while temp is not None:

if temp.item < p: # if smaller than pivot

count = count + 1

Append(Left, temp.item) # add element on left side

else: # else equal or greater than pivot

count = count + 1

Append(Right, temp.item) # add element on right side

temp = temp.next

if GetLength(Left) < n: # median not left of pivot

count = count + 1

return QuickSortM(Right, n - GetLength(Left) - 1) # Search right side for median

elif GetLength(Left) > n: # median not right of pivot

count = count + 1

return QuickSortM(Left, n) # search left for median

else:

return p

def createList(L, n):

for i in range(n):

Append(L, int(random()\*100))

def Median(L):

C = Copy(L)

#print("The median element is: ", bubbleSort(C))

print("The median element is: ", ElementAt(QuickSort(C), GetLength(C)//2))

#print("The median element is: ", QuickSortM(C, GetLength(C)//2))

return

**Conclusion**

There are different ways to implement sorting methods and the way you approach the problem affects how efficient everything in your program runs. Using linked lists to implement these methods also made it more challenging but it made me know how to use it better and implement methods for these linked lists. This helped out to recognize how the sorting methods affect your program and how fast and efficient it ran.