**Course Name:** 2302 **Author:** Olugbenga Iyiola **ID:** 80638542 **Instructor:** Olac Fuentes **TA:** Nath Anindita **LAB #2 Report**

**Introduction**

The purpose of this lab is to implement several algorithms for ﬁnding the median of a list of integers, using objects of a Linkedlist class, and comparing their running times (measured as the number of comparisons each algorithm makes) for various list lengths. The algorithms sorts the elements of a list in ascending order and returns the median and the various sorting algorithms implemented in this experiment include bubble sort, merge sort, quick sort and optimized quick sort.

**Proposed Solution Design and Implementation**

Four different sorting algorithms are used in this lab and we determine the big-O running time with respect to various input sizes, n and we determine the number of comparisons that each algorithm makes and to know if their analytical running times agree with theory.

**Bubble Sort**

The first sorting algorithm is bubble sort which is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order and this algorithm can be easily optimized by observing that the *n*-th pass finds the *n*-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last *n* − 1 items when running for the *n*-th time. The pseudocode is as follows;

*For I = 0 to N - 2*

*For J = 0 to N - 2*

*If (A(J) > A(J + 1)*

*Temp = A(J)*

*A(J) = A(J + 1)*

*A(J + 1) = Temp*

*End-If*

*End-For*

*End-For*

Average and worst case complexity of bubble sort is O(n2). Also, it makes O(n2) swaps in the worst case. Bubble sort is adaptive. It means that for almost sorted array it gives O(n) estimation.

**Merge Sort**

Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list. The pseudocode is as follows;

**Recursively sort the first half of the input array.  
Recursively sort the second half of the input array.  
Merge two sorted sub-lists into one list.**

C = output [length = n]  
A = 1st sorted array [n/2]  
B = 2nd sorted array [n/2]  
i = 0 or 1 (depending on the programming language)  
j = 0 or 1 (depending on the programming language)

for k = 1 to n

if A(i) < B(j)  
C(k) = A(i)  
i = i + 1

else if A(i) > B(j)  
C(k) = B(j)  
j = j + 1

The list of size N is divided into a max of logN parts, and the merging of all sublists into a single list takes O(N)time, the worst case run time of this algorithm is O(NLogN)

**Quick Sort**

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. The pseudocode is as follows;

*Quicksort(A,p,r) {*

*if (p < r) {*

*q <- Partition(A,p,r)*

*Quicksort(A,p,q)*

*Quicksort(A,q+1,r)*

*}*

*}*

*Partition(A,p,r)*

*x <- A[p]*

*i <- p-1*

*j <- r+1*

*while (True) {*

*repeat*

*j <- j-1*

*until (A[j] <= x)*

*repeat*

*i <- i+1*

*until (A[i] >= x)*

*if (i<-=""> A[j]*

*else*

*return(j)*

*}*

*}*

[Mathematical analysis](https://en.wikipedia.org/wiki/Analysis_of_algorithms) of quicksort shows that, [on average](https://en.wikipedia.org/wiki/Best,_worst_and_average_case), the algorithm takes [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*) comparisons to sort *n* items. In the [worst case](https://en.wikipedia.org/wiki/Best,_worst_and_average_case), it makes O(*n*2) comparisons

**Modified Quick Sort**

This quick sort method is optimized by making only a single recursive call instead of the two made by the normal quicksort. This is achieved by using a conditional statement in the algorithm to ensure that only the sublist where the median is known to reside is processed.

**Experimental Result**

System Specification: HP Windows 10, 1.60GHZ Intel® Celeron® , 4.GB RAM, 64-bit operating system

The results of the various test cases for each of the sorting algorithms are shown below:

**Bubble Sort**

|  |  |  |
| --- | --- | --- |
| **N(Input)** | **Number of Comparisons** | **Runtime in nanaseconds** |
| **11** | **25** | **1\*10-9** |
| **101** | **2560** | **1.5\*10-9** |
| **500** | **62218** | **1.5\*10-9** |
| **1001** | **241448** | **1\*10-10** |

Recurrence Equation: T(n)=T(n−1)+(n−1) which, using the master theorem, gives us O(n2).

**Merge Sort**

|  |  |  |
| --- | --- | --- |
| **N(Input)** | **Number of Comparisons** | **Runtime in nanaseconds** |
| **11** | **7** | **1\*10-9** |
| **101** | **51** | **1.5\*10-9** |
| **501** | **126** | **1.5\*10-9** |
| **1001** | **501** | **1.5\*10-9** |

Recurrence Equation: T(n) = 2T(n/2) + O(n) which, using the master theorem, gives us O(n log(n)).

**Quick Sort**

|  |  |  |
| --- | --- | --- |
| **N(Input)** | **Number of Comparisons** | **Runtime in nanaseconds** |
| **11** | **10** | **1\*10-9** |
| **101** | **44** | **1.5\*10-9** |
| **501** | **185** | **1.5\*10-9** |
| **1001** | **821** | **1\*10-10** |

Recurrence Equation: T(n) = n + T(n-1) which, using the master theorem, gives us O(n2).

**Modified Quick Sort**

|  |  |  |
| --- | --- | --- |
| **N(Input)** | **Number of Comparisons** | **Runtime in nanaseconds** |
| **11** | **4** | **1\*10-9** |
| **101** | **44** | **1.5\*10-9** |
| **500** | **343** | **1.5\*10-9** |
| **1001** | **773** | **1\*10-10** |

Recurrence Equation: T(n)=T(i)+T(n-i)+c⋅n which, using the master theorem, gives us O(logn).

**CONCLUSION**

The fastest of all the algorithms tested was the modified quicksort while the slowest was bubble sort. It is observed that some of the factors that affect complexity and running time include algorithmic complexity, startup costs, additional space requirements, use of recursion , worst-case behavior, assumptions about input data, caching, and behavior on already-sorted or nearly-sorted data

**Appendix**

***Programmed by Olac Fuentes***

#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()

L = List()

print(IsEmpty(L))

for i in range(5):

Append(L,i)

Print(L)

PrintRec(L)

PrintReverse(L)

Remove(L,2)

Print(L)

Remove(L,20)

Print(L)

Remove(L,0)

Print(L)

Remove(L,4)

Print(L)

print(L.head.item)

print(L.tail.item)

**Wikipedia**

[**https://en.wikipedia.org/wiki/Sorting\_algorithm#Comparison\_of\_algorithms**](https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms)

**Academic Dishonesty**

This work was done by me without any act or practice of academic dishonesty