Tyler Salas

1:30-2:50 Class

Dr. Fuentes

80596221

Lab 2 Report

Lab two, assigned this week, aimed to implement four different types of sorting algorithms into Python. The methods were performed on lists that were randomly generated, (the non-native lists to Python but instead the lists we went over in class (Linked Lists)) from there the median of the sorted list was to be found and returned with a Median function specified in the assignment paper online. All the while the sorting method is to contain a counter variable that is to count the number of comparisons made to reach the sorted list.

The way I planned to solve this is to simply implement the sorting algorithms to the randomly generated list, from there, inside of the different methods for the algorithms, I would place a counter variable in the appropriate place to add the correct number of comparisons made by the algorithm. For each algorithm’s output, I print the original list, the sorted list, the median, and the number of comparisons made by said algorithm as in the example below.

Bubble Sort:

Original List:

74 51 0 98 100 27 74 14 58 61 29 35 6 25 17

Sorted List:

0 6 14 17 25 27 29 35 51 58 61 74 74 98 100

Median: 35

Number of Comparisons: 69

*The comparisons in the tables below are made with the same lists respectively (10 items, test 1 comparison in bubble sort is the same list as merge sort 10 items test 1 comparisons)*

**Bubble Sort**

For this sorting algorithm the basic idea was to travel from left to right, comparing each element of the list with the element to the right of said element in the list, switching the two if they were not in the correct order. (Original element was greater than the next element) There lied a Boolean variable in the list (change) that determined if the list changed from the previous round of comparisons made, if so, the method’s while loop ran again until the variable equaled false by the end, meaning there was no change in the list. (All the elements were in order)

**Comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** |
| **15 Items** | 45 | 52 | 41 | 55 | 56 |
| **25 Items** | 160 | 166 | 124 | 119 | 133 |
| **50 Items** | 376 | 589 | 632 | 650 | 647 |
| **100 Items** | 2621 | 2498 | 2800 | 2859 | 2318 |

The run time for bubble sort is O(n^2) where n is the number of items being sorted. It is the worst run time algorithm between numbers one, two and three by quite a significant margin. We can see from the results of the number of comparisons that bubble sort is terribly inefficient and that as the number of the items in the list were increased, the number of comparisons grew significantly.

**Merge Sort**

The merge sort algorithm was a recursive algorithm that continuously cuts the size of the original list by two, creating a left and a right list for each recursive call made by the function. This process is repeated until the length of the list is less than two, from there the function reaches its base case and begins working on the actual sorting process of the list. The function then begins to compare the first element in line for the left and right-side lists. If the left side item is less than the right-side item, then that item is put first into a temp list that holds the reference to the original list and vice-versa. Once either the left or right-side list is equal to none, a new while loop is called that puts the remainder of either list into the remaining slots of the temp list.

**Comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** |
| **15 Items** | 41 | 42 | 38 | 43 | 43 |
| **25 Items** | 86 | 86 | 88 | 86 | 85 |
| **50 Items** | 221 | 226 | 221 | 224 | 220 |
| **100 Items** | 538 | 530 | 538 | 536 | 552 |

The run time for merge sort is O(nlogn) where n is the number of items being sorted. It is supposed to be the best sorting algorithm out of the original three proposed in the lab. From the results of the comparisons made, we can tell that merge sort is a very efficient algorithm to use definitely the best or the three full sorting algorithms. The comparisons did not increase nearly as much as bubble-sorts did and through all the tests of the same length lists, the number of comparisons stayed nearly the same.

**Quick Sort**

I divided the quick sort algorithm into three different functions. The function quickSort which is the original call, whose only function is to call another function quickSorter which if the first element is less than the last, (base case) terminates, else calls a function called partition that holds the bulk of importance of the algorithm. The partition function sets the first item in the list called to a variable called pivot, another variable leftmark is set to the second element of the list called, and another variable rightmark is set to the last element of the list called. The variable leftmark then moves from left to right if the item of leftmark is less than the item of pivot. The variable rightmark does the same except it moves from right to left if the item is greater than the item of pivot. If the right mark is less than the left, the while loop terminates, else the leftmark and rightmark items are exchanged, putting those two items in order and creating a split point, the pivot item is exchanged for the split point and the partition function terminates. The recursice calls return to quickSorter and the method quickSorter is called with parameters that pertain to the splitpoint, splitting the list in two

**Comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** |
| **15 Items** | 39 | 37 | 45 | 50 | 38 |
| **25 Items** | 85 | 95 | 102 | 91 | 106 |
| **50 Items** | 313 | 233 | 246 | 216 | 250 |
| **100 Items** | 652 | 605 | 590 | 661 | 598 |

.

The run time for merge sort is O(nlogn) where n is the number of items being sorted. It is supposed to be the second-best sorting algorithm of numbers one, two and three being only slightly behind merge sort. From the number of comparisons made we can infer that quick sort is an efficient algorithm to use. The number of comparisons had somewhat of a wide range in same length lists, making it slightly unpredictable, but the number managed to stay relatively low.

**Quick Sort Modified**

This method was made to be a modified version of the original quick sort. The idea I used to make the method was to get the median of the original list created then compare all the items in the list. If the majority of the items were greater, I would call the recursive quickSorter2b method to the respective side of the splitpoint (Vice versa if the majority of items were less) which would simply process the quicksort method on only the side with greatest number of items in the list. I will be completely honest and state that the method did not work for every instance of randomly generated list though it seemed to work a solid four out of five times though I’m not sure what the error is due too.

The instance of the method below seemed to work fine, returning the correct median with a large margin of comparisons spared.

Quick Sort Modified

Original List:

51 47 14 68 70 14 40 28 80 17 81 33 78 75 93

Sorted List:

14 14 17 28 33 40 47 51 80 70 81 68 78 75 93

Median: 51

Number of Comparisons: 26

Though the instance below this did not work. I observed that when the lists had a duplicate number inside the list, the program seemed to get confused

Quick Sort Modified

Original List:

37 88 27 9 76 7 98 40 89 89 67 29 97 54 34

Sorted List:

7 9 27 29 34 37 98 40 89 89 67 76 97 54 88

Median: 40

Number of Comparisons: 22

0.0

**Comparisons**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test 1** | **Test 2** | **Test 3** | **Test 4** | **Test 5** |
| **15 Items** | 28 | 28 | 45 | incorrect | 30 |
| **25 Items** | 64 | 55 | 80 | incorrect | 106 |
| **50 Items** | 301 | 208 | 176 | 185 | 242 |
| **100 Items** | 430 | 506 | incorrect | 409 | 405 |

I’d imagine the time complexity worst case would be the same as the original quick sort, but with the potential to be significantly less than quick sort. I think the benefits to quick sort shine when you implement this modification to the algorithm. The number of comparisons made by this algorithm are significantly lower than even that of merge sort. The only issue being it only worked an average of four out of five times, due to some bug in the code. Though I’d imagine if done correctly, the advantages this algorithm brings to finding a median are very well.

If anybody wanted to replicate tests in my code, to change the number of items in the list, all you’d have to do is find the first line of where the main method starts and change the variable number in create list.

When we look at the three different algorithms implemented in the lab, we can conclude that obviously the algorithms have an order from best to worst. The modified quick sort algorithm worked the fastest in finding what was needed, though it did not work sometimes. Merge sort would be the second best in finding the median though it does have the advantage if it is ever needed to sort the entire list. Quick sort did a fine job in finding the median though I’d only think you’d use this sorting method over merge sort for the modified version. And lastly bubble sort seemed to be the simplest yet worst sorting algorithm there could be. Overall this lab taught me the efficiency and inefficiency of different sorting algorithms and how to implement them.

#CS2302

#Tyler Salas

#Lab2

#Dr.Fuentes

#Anindita Nath

#Sort lists with different methods and count numbers of comparison

import random

import time

class Node(object):

# Constructor

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

self.item = item

self.next = next

#List Functions

class List(object):

# Constructor

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

self.head = None

self.tail = None

def IsEmpty(L):

return L.head == None

def Prepend(L,x):

if IsEmpty(L):

L.head = Node(x)

L.tail = L.head

else:

L.head = L.head.next

L.head = Node(x)

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 getLength(L):

count = 0

if IsEmpty(L):

return 0

else:

temp = L.head

while temp is not None:

count += 1

temp = temp.next

return count

def copy(L):

t = List()

temp = L.head

while temp is not None:

Append(t, temp.item)

temp = temp.next

return t

def createList(n):

L = List()

for i in range(n):

Append(L,random.randint(0,101))

return L

def copyLeft(L, mid):

t = List()

temp = L.head

for i in range(mid):

Append(t, temp.item)

temp = temp.next

return t

def copyRight(L, mid):

t = List()

temp = L.head

for i in range(mid):

temp = temp.next

while temp is not None:

Append(t, temp.item)

temp = temp.next

return t

def bubbleSort(L):

global count

change = True

while change:

t = L.head

change = False

while t.next is not None:

if t.item > t.next.item:

count += 1

temp = t.item

t.item = t.next.item

t.next.item = temp

change = True

t = t.next

def getMedian(L):

t = L.head

mid = getLength(L)//2

for i in range(mid):

t = t.next

return t.item

def concatenate(L1, L2):

if IsEmpty(L1):

if IsEmpty(L2):

return

else:

L1.head = L2.head

L1.tail = L2.tail

return

if IsEmpty(L2):

return

else:

L1.tail.next = L2.head

L1.tail = L2.tail

return

def mergeSort(L):

global count

temp = L.head

if getLength(L) >= 2:

mid = getLength(L)//2

lefthalf = copyLeft(L,mid)

righthalf = copyRight(L,mid)

mergeSort(lefthalf)

mergeSort(righthalf)

ltemp = lefthalf.head

rtemp = righthalf.head

while ltemp is not None and rtemp is not None:

count += 1

if ltemp.item < rtemp.item:

temp.item = ltemp.item

ltemp = ltemp.next

else:

temp.item = rtemp.item

rtemp = rtemp.next

temp = temp.next

while ltemp is not None:

temp.item = ltemp.item

ltemp = ltemp.next

temp = temp.next

while rtemp is not None:

temp.item = rtemp.item

rtemp = rtemp.next

temp = temp.next

def getItem(L,x):

temp = L.head

i = 0

while i < x and temp.next is not None:

temp = temp.next

i += 1

return temp

def quickSort(L):

quickSorter(L,0,getLength(L)-1)

def quickSorter(L,first,last):

if first<last:

splitpoint = partition(L,first,last)

quickSorter(L,first,splitpoint-1)

quickSorter(L,splitpoint+1,last)

def partition(L,first,last):

global count

temp = getItem(L,first)

pivot = temp

leftmark = first+1

rightmark = last

fst = getItem(L,leftmark)

right = getItem(L,last)

done = False

while not done:

if right.item < fst.item:

t = fst.item

fst.item = right.item

right.item = t

while leftmark <= rightmark and fst is not None and fst.item <= pivot.item:

count += 1

leftmark = leftmark + 1

fst = fst.next

while right.item > pivot.item and rightmark >= leftmark:

count += 1

rightmark = rightmark -1

right = getItem(L,rightmark)

if rightmark <= leftmark:

done = True

else:

t = fst.item

fst.item = right.item

right.item = t

t = pivot.item

pivot.item = right.item

right.item = t

return rightmark

def quickSort2(L):

quickSorter2(L,0,getLength(L)-1)

def quickSorter2(L,first,last):

if first>=last:

return

else:

splitpoint = partition(L,first,last)

det = detMedian(L)

if det:

quickSorter2b(L,first,splitpoint-1)

else:

quickSorter2b(L,splitpoint+1,last)

def quickSorter2b(L,first,last):

if first >= last:

return

else:

splitpoint = partition(L,first,last)

quickSorter(L,first,splitpoint-1)

quickSorter(L,splitpoint+1,last)

def detMedian(L):

median = getItem(L,getLength(L)//2)

temp = L.head

tall = 0

short = 0

while temp is not None:

if median.item > temp.item:

short += 1

else:

tall += 1

temp = temp.next

if short <= tall:

return True

else:

return False

def bMedian(L):

C = copy(L)

bubbleSort(C)

return getItem(C,getLength(C)//2)

def mMedian(L):

C = copy(L)

mergeSort(C)

return getItem(C,getLength(C)//2)

def qMedian(L):

C = copy(L)

quickSort(C)

return getItem(C,getLength(C)//2)

def q2Median(L):

C = copy(L)

quickSort2(C)

return getItem(C,getLength(C)//2)

#Initializing lists to be used all lists are made the same for accurate results

L1 = createList(100)

L2 = copy(L1)

L3 = copy(L1)

L4 = copy(L1)

count = 0

b = bMedian(L1)

count = 0

m = mMedian(L2)

count = 0

q = qMedian(L3)

count = 0

q2 = q2Median(L4)

#Bubble Sorting and print original/new/comparisons of List

start = time.time()

print("Bubble Sort:")

count = 0

print("Original List: ")

Print(L1)

bubbleSort(L1)

print("Sorted List:")

Print(L1)

print("Median:",b.item)

print("Number of Comparisons: ",count)

end = time.time()

print(end - start)

print()

#Merge Sorting and print original/new/comparisons of List

start = time.time()

print("Merge Sort")

print("Original List: ")

count = 0

Print(L2)

mergeSort(L2)

print("Sorted List:")

Print(L2)

print("Median:",m.item)

print("Number of Comparisons: ",count)

end = time.time()

print(end - start)

print()

#Quick Sorting and print original/new/comparisons of List

start = time.time()

print("Quick Sort")

print("Original List: ")

count = 0

Print(L3)

quickSort(L3)

print("Sorted List:")

Print(L3)

print("Median:",q.item)

print("Number of Comparisons: ",count)

end = time.time()

print(end - start)

print()

#Quick Sorting and print original/new/comparisons of List

start = time.time()

print("Quick Sort Modified")

print("Original List: ")

count = 0

Print(L4)

quickSort2(L4)

print("Sorted List:")

Print(L4)

print("Median:",q2.item)

print("Number of Comparisons: ",count)

end = time.time()

print(end - start)

print()

I, Tyler Salas, did not receive any inappropriate help with this lab and completed it by myself.