For lab 2 of the CS 2302 Data Structures class we have to create a linked list and sort it using four different algorithms. For the algorithms we have to use bubble sort, merge sort, quicksort, and a modified version of quicksort. After sorting the linked list we have to create a function that will return the element in the middle of the sorted linked list.We also had to count how many times we compare elements.

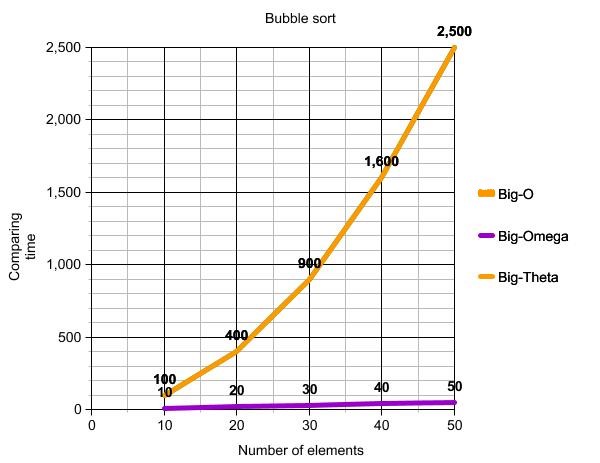
So what is a linked list? A linked list is a linear data structure where each element is a separate object. Unlike an array a linked list is not stored at contiguous locations, insted the elements are linked using pointers. We called this objects Nodes and each node has a pointer that stores the element and a pointer that stores the nexts Node’s location. Using this information we are able to create a sorting algorithm that uses the logic of a linked list with the logic of bubble sort, merge sort, and quicksort.

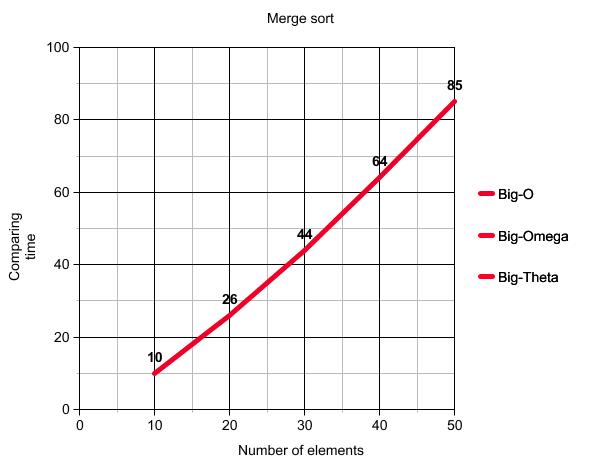
The first algorithm that I implemented was bubble sort. Bubble sort is a simple sorting algorithm that repeatedly steps through the list and compares adjacent pairs and swaps them if they are in the wrong order. I created a function named Buuble\_sort() that would take a linked list, the length of the list and a counter for the number of comparisons. First we start of by checking if our list is empty or the list contains only one element. If this is the case then the list is already sorted and we do not need to do anything. Next we create a for loop that would traverse through the whole linked list using the length of our list. Inside the for loop we create a temporary pointer to the first Node in our list and a temporary pointer to the second Node in our list. After that we traverse the linked list until we get to the last Node. If in the way to the last Node and element is out of place we swap them, else we just select the next two Nodes to compare. This is done until we have gone through the whole linked list n number of times, where n is the number of Nodes in the list. After the list has been sorted I print the list out and return the number of comparisons.

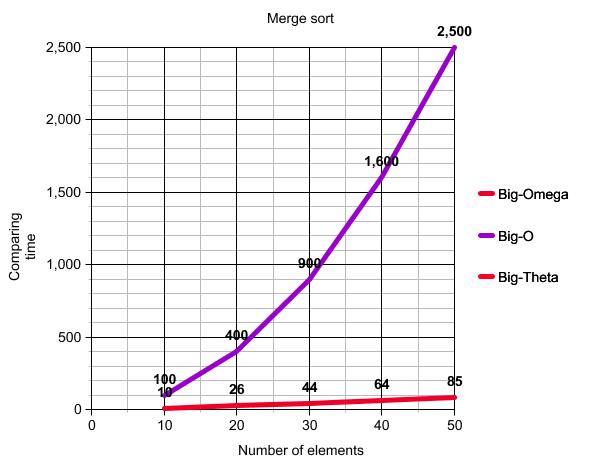
The second algorithm I implement is Merge Sort. Merge sort is a divide and conquer algorithm. It divides the imputed linked list in two halves by using the middle element as the point of division. To get the middle Node I created a function that takes the first Node of a linked list. After that i created to pointers one called slow that points to the first Node and one called fast that points to the second Node. Then i created a while loop that checks if the pointer fast is not None meaning that it still points to a Node. If so we make the pointer equal to the next node of fast by using fast.next and we go to an if statement. In the if statement we check again if the fast pointer is still pointing to a Node. If so we make the pointer slow equal to the next Node in line and the pointer fast to the next Node in line. Once the pointer faste is equal to None we brake off the while loop and return the Node that the pointer slow is pointing to that is the middle node of the linked list. After that we do the same by calling merge sort again on the left side of the linked list and the right side of the linked list untill or first Node is None or the second Node is None. After that we called the function Sorted\_Merge() that takes the left and right side of the divide linked list. Here we start to merge the two halves from steps 2 and 3 from Merge\_Sort(). In Sorted\_Merge() we start to connect the nodes in the right order for that we have some base cases. If the left side of the linked list is empty then we return the right side. If the right side of the linked list is empty then we return the left side. If neither of this base cases is true then we compare the left first element and the right first element depending if the left element is smaller or bigger we we add left or the right Node and linked the next Node by calling Sorted\_Merge() with left.next and right or left and right.next. In the end we return the first element after been sort back to Merge\_Sort() and there we return the sorted list.

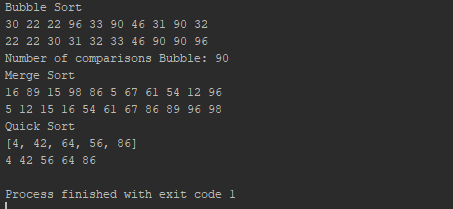
The last algorithm I had to do was Quicksort. Like Merge sort Quicksort is also a divide and conquer algorithm. It picks an element as a pivot. This element can be the first, last, midian, or random element in the linked list. In my algorithm I picked the last element in my list. The key part of Quicksort is the function named Partition(). This function takes the last element as the pivot and places the pivot at its correct spot in the linked list. IT also places all the smaller elements from the linked list to the left of the pivot and all the larger elements in the list to the right of the pivot. For me it was hard to use Quicksort to sort a linked list. So what I did was to convert my linked list to a regular python list and then using Quicksort to sort the array. I first start by getting the first pivot and setting it in the correct spot. I then do the same to the left side of the pivot and the right side of the pivot. After that is done and they are all sorted I recreated the Linked List and returned it.

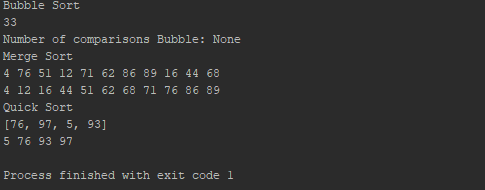
In conclusion I found Bubble sort to be the easiest sorting algorithm to work with. I was able to apply the linked list format much easier than Merge sort and Quicksort. Second would have to be Merge sort as all you are doing is creating smaller instances of the linked list and then connecting them in the right order. Lastly Quicksort was the hardest as I did not know how to apply it to the Linked List format.











import random

# 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 Bubble\_Sort(L,l,c):

if IsEmpty(L) or L.head.next == None:

return

for x in range(l):

current = L.head

c\_next = L.head.next

while c\_next != None:

c+=1

if current.item > c\_next.item:

temp = current.item

current.item = c\_next.item

c\_next.item = temp

current = c\_next

c\_next = c\_next.next

Print(L)

return c

def Merge\_Sort(L):

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

return L

middle = Get\_Middle(L)

next\_middle = middle.next

middle.next = None

left = Merge\_Sort(L)

right = Merge\_Sort(next\_middle)

sorted\_list = Sorted\_Merge(left,right)

return sorted\_list

def Sorted\_Merge(left,right):

result = None

if left == None:

return right

if right == None:

return left

if left.item <= right.item:

result = left

result.next = Sorted\_Merge(left.next,right)

else:

result = right

result.next = Sorted\_Merge(left,right.next)

return result

def Get\_Middle(L):

fast = L.next

slow = L

while fast != None:

fast = fast.next

if fast != None:

slow = slow.next

fast = fast.next

return slow

def Quick\_Sort(A,low,high):

if low < high:

pivot = Partition(A,low,high)

Quick\_Sort(A,low,pivot-1)

Quick\_Sort(A,pivot+1,high)

L = List()

for x in A:

Append(L,x)

Print(L)

return Get\_Middle(L).item

def Partition(A,low,high):

i = low-1

pivot = A[high]

for j in range(low,high):

if A[j] <= pivot:

i += 1

A[i], A[j] = A[j],A[i]

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

return i+1

n=12

print('Bubble Sort')

counter = 0

L = List()

lenght = random.randint(0,n)

for x in range(lenght):

Append(L, random.randint(0,100))

Print(L)

counter = Bubble\_Sort(L,lenght,counter)

print('Number of comparisons Bubble:',counter)

print('Merge Sort')

counter = 0

L = List()

lenght = random.randint(0,n)

for x in range(lenght):

Append(L, random.randint(0,100))

Print(L)

L.head = Merge\_Sort(L.head)

Print(L)

print('Quick Sort')

L = List()

l= []

lenght = random.randint(0,n)

for x in range(lenght):

l.append(random.randint(0,100))

print(l)

L = Quick\_Sort(l,0,len(l)-1)

Print(L)

Academic dishonesty includes but is not limited to cheating, plagiarism and collusion. Cheating may involve copying from or providing information to another student, possessing unauthorized materials during a test, or falsifying data (for example program outputs) in laboratory reports. Plagiarism occurs when someone represents the work or ideas of another person as his/her own. Collusion involves collaborating with another person to commit an academically dishonest act. Professors are required to - and will - report academic dishonesty and any other violation of the Standards of Conduct to the Dean of Students.

* Anthony Herrera