Lab 2 - Data Structures Report Brian Ricardo Perez

Lab 2 is a culmination of all we have learned thus far, the task at hand is to receive two files of data full of IDs and put both set of IDs into one linked list. After doing so, implement four of the solution scenarios given by the professor, giving and plotting the Big O of each different solution, each being a different way to determine duplicates (and I removed them, but it was never explicitly asked for them to be deleted)

For each of the four solutions even though they were different methods of solving a problem, they solve the same problem nonetheless. At different memory usages, time complexities, and other variables can make one method more efficient than the other. As for how I solved each problem, it was not as much a choice as usual, for I had four solutions to implement:

○ Solution 1: Compare every element in the list with every other element in the list using nested loops

○ Solution 2: Sort the list using bubble sort, then determine if there are duplicates by comparing each item with the item that follows it in the list (if there are duplicates in the original list, they must be neighbors in the sorted list).

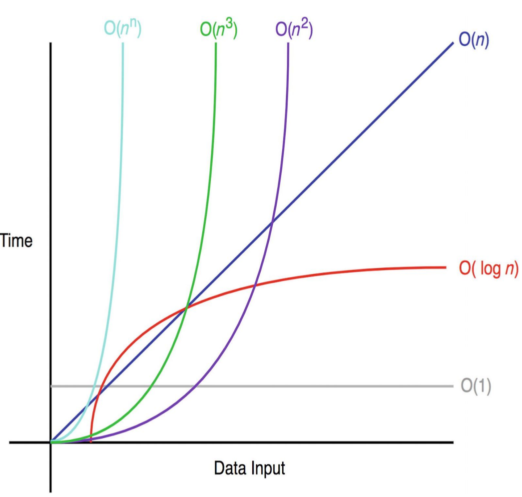
○ Solution 3: Sort the list using merge sort (recursive), then determine if there are duplicates by comparing each item with the item that follows it in the list.

○ Solution 4: Take advantage of the fact that the range of the integers in the list is fixed (0 to m, where m is the largest ID you can find in the linked list). Use a boolean array seen of length m+1 to indicate if elements in the array have been seen before. Then determine if there are duplicates by performing a single pass through the unsorted list. Hint: while traversing the list, seen[item] = True if integer item has been seen before in the search.

When it came to each implementation, I just wanted to make it as simple as possible, even though on some I can tell maybe I could have done it better. Either way, just take the guides for the implementation and do them accordingly. I put each solution into its own method on the linked list method, and within its own method would call others to help whether it be size(), biggest() or more specific, merg_sort(). Besides the basics and the solutions I did not add anything crazy to the Linked List method

For solution 1 the Big O time is n^2, it goes through each list, the list length of times, checking each element of linked list, with itself using 2 while loops. Very simple not much to it, from my testing it was about the third fastest method (with 2 ties for first elsewhere). For the second, it just iterates through itself back and forth checking each element with itself. When it came to the third solution with recursion, it was nice, using merge sort to dive and conquer the sorting and deleting. Being able to cut in half over and over again with recursion was nice. For the fourth, I'll be honest had there not been instructions for this solution it would not have crossed my mind to make an array of the biggest element size and then marking true if its been seen was to me amazing. And if it was my guess, I would say that that method would be the fastest and looks to be an O(1) constant time complexity.

BIG O's of S's



S1: O(n^2)

S2: O(n^n)

S3: O(1)

S4 : O (1)

n	S1	S2	S3	S4
3000	≈2.05 Seconds	+10m	1.15 Seconds	1.47
6000	5.49	X	Max me.	3.2
-> 1000	0.46	1:05 min	0.39	0.45 Seconds

When it came to testing I found out quite a few things. The order of efficiency came to be as such: Solution 4 and 3 were tied in time complexity but S3 seemed to have some kind of cap at 3000 elements in the list, and solution 1 was shortly behind where as solution 2 came to be very slow and time consuming I thought S4's time complexity to be n^n. Like I stated before, I do not know or could not figure why the merge sort capped at about 3000 elements in the recursion method, besides that it was stated that it hit the max depth of recursion, maybe I just wrote the method wrong making it not work for higher limits, I will research that more later. Here's the error @ around 3000 elements or more

```
  File "/Users/brianperez/** Data Structures /Lab 2/
Lab2Start.py", line 200, in merge_lists
    if int(list1.item) <= int(list2.item):

RecursionError: maximum recursion depth exceeded while
calling a Python object
```

All in all it was a great learning experience especially since before doing this I had never thought of how much a difference each method could change efficiency, but that probably comes with the fact that the problems I was handling were too simple otherwise to even be an issue. I look forward to the degree of difficulty ahead of me. After seeing the time complexities I did, from instant to forever, working in efficiency will be something I will have to take into consideration from now on.

Appendix:

```
"""
Created on Mon Sep  20 12:40:04 2019
Course: CS 2302 - Data Structures
Author: Brian Perez
Assignment: Lab 2
Instructor: Diego Aguirre
D.O.L.M.: 10/7/19

"""
def create_list(LL):
    file = open("vivendi.txt", 'r')
    file2 = open("activision.txt", 'r')

    for x in file.read().split('\n'):
        LL.add_last(x)
    for x in file2.read().split ('\n'):
        LL.add_last(x)


#    for elements_b in file2.read().split('\n'):
#        print (elements_b)
class Node:
    item = -1
    next = None

    def __init__(self, item, next):
       self.item = item
       self.next = next

class Linked_list:

    def __init__(self, head=None):
        self.head = head

    def add_last(self, item):

        if self.head is None:  # If the list is empty, add a new head
            self.head = Node(item, self.head)
            return

        current = self.head
        while current.next is not None:  # Looking for the second to last node
            current = current.next

        current.next = Node(item, None )

    def add(self, index, item):
        if index == 0:
            self.head = Node(item, self.head)
            return

        if index < 0:  # Don't do anything if index is invalid
            return

        current = self.head
        for i in range(index - 1):  # Looking for the node at position index - 1
            if current is None:
```

```
            return

        current = current.next

    if current is not None:
        current.next = Node(item, current.next)

def index_of(self, item):
    current = self.head
    i = 0
    while current:
        if int(current.item) == int(item):
            return i
        i += 1
        current = current.next
    return -1

def print_list(self):
    current = self.head

    while current is not None:
        print(current.item)
        current = current.next

def size(self):
    current = self.head

    length = 0
    while current:
        length += 1
        current = current.next

    return length

def biggest(self):
    current = self.head
    biggest = 0
    while current:
        if biggest < int(current.item):
            biggest = int(current.item)
        current = current.next
    return biggest

def remove(self, index):
    if index < 0:  # Don't do anything if index is invalid
        return

    if index == 0:  # Handling special case - when the item to remove is the head
        self.remove_first()
        return

    current = self.head

    for i in range(index - 1):  # Looking for the second to last node
        if current is None:
            return

        current = current.next

    if current is not None and current.next is not None:
        current.next = current.next.next
```

```
def bubble_sort (self):
    swap_counts  = 0
    start = self.head
    p = self.head
    q = p.next
    while p.next :
        if q is None :
            if swap_counts == 0 :
                print("stopped")
                return
        swap_counts = 0
        if int(p.item) > int(q.item) :
            temp = q.item
            q.item = p.item
            p.item = temp
            swap_counts += 1
        if swap_counts != 0:
            p = start
            q = p.next
        else :
            p = q
            q = p.next

def next_same_delete(self):
    current = self.head
    if current.next is None :
        return
    while current.next:
        if current.item == current.next.item:
            current.next = current.next.next
        current = current.next

def solution1(self):
    current = self.head
    second = current
    while current:
        while second.next:   # check second.next here rather than second
            if second.next.item == current.item:   # check second.next.data, not second.data
                second.next = second.next.next   # cut second.next out of the list
            else:
                second = second.next   # put this line in an else, to avoid skipping items
        current = current.next
        second = current

def solution2 (self):
    self.bubble_sort()
    self.next_same_delete()

def solution3(self):
    self.head = merge_sort(self.head)
    #self.print_list()
    #print("\n")
    self.next_same_delete()

def solution4(self):
    seen = [False] * (self.biggest()+1)
    current = self.head
    while current:
        if seen[int(current.item)] == True:
            self.remove(self.index_of(int(current.item)))
```

```
            else:
                seen[int(current.item)] = True
            current = current.next
        return

def merge_sort(head):
    if head is None or head.next is None:
        return head
    list1, list2 = divide_lists(head)
    list1 = merge_sort(list1)
    list2 = merge_sort(list2)
    head = merge_lists(list1, list2)
    return head

def merge_lists(list1, list2):
    temp = None
    if list1 is None:
        return list2
    if list2 is None:
        return list1
    if int(list1.item) <= int(list2.item):
        temp = list1
        temp.next = merge_lists(list1.next, list2)
    else:
        temp = list2
        temp.next = merge_lists(list1, list2.next)
    return temp

def divide_lists(head):
    slow = head                    # slow is a pointer to reach the mid of linked list
    fast = head                   # fast is a pointer to reach the end of the linked list
    if fast:
        fast = fast.next
    while fast:
        fast = fast.next           # fast is incremented twice while slow is incremented once per loop
        if fast:
            fast = fast.next
            slow = slow.next
    mid = slow.next
    slow.next = None
    return head, mid



def main () :
    LL = Linked_list()
    create_list(LL)
    #LL.print_list()
    print("\n")

    #LL.solution1() #second fastest
    #LL.solution2() #slowest
    LL.solution3() #fastest when fit
    #LL.solution4() #fastest

    LL.print_list()

main ()
```