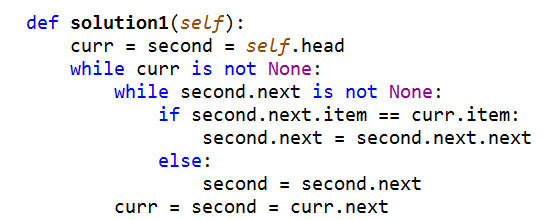
**Introduction:**

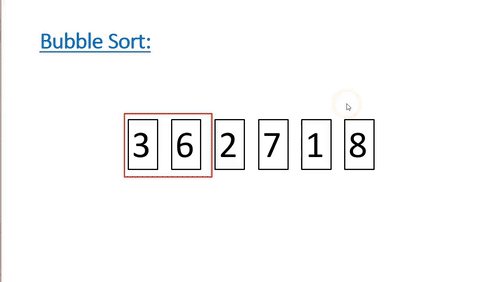
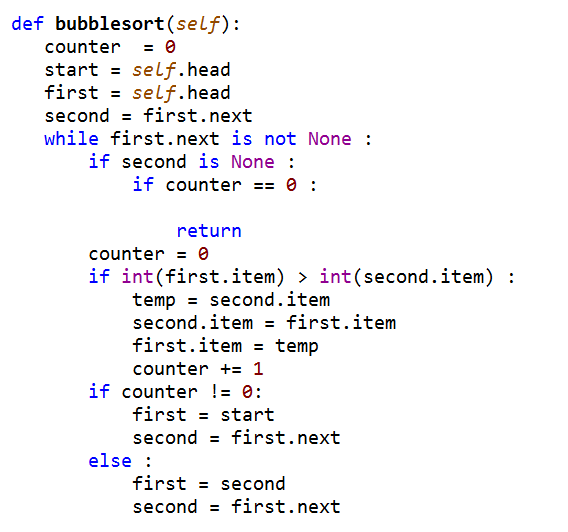
My job was to read to files [*activision.txt*](https://www.dropbox.com/s/r0822hnsdznnh31/activision.txt?dl=0) and [*vivendi.txt*.](https://www.dropbox.com/s/q09ilkvyfu8uege/vivendi.txt?dl=0) Each of these files contains the IDs of all of the company’s active employees creating only one single linked list with this 4 solutions.

* + 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.

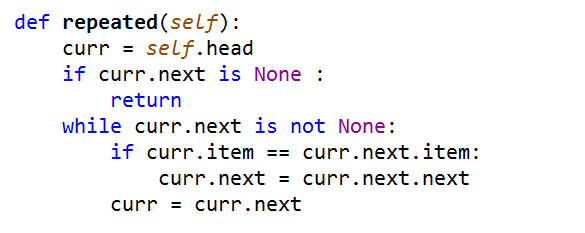
**Proposed solution design and implementation:**

In order of finishing this lab first I had to read the two files using nested which was easy, then I had to compare to each one and if we have any duplicate, we erase them because we know that are in the same LinkedList.



The second problem includes bubble sort, we need to compare each one by swapping if the first is bigger than the second one, I will attach an illustration which help us in order to know how bubble sort works better

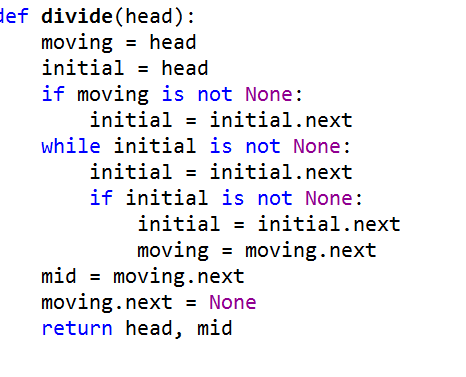
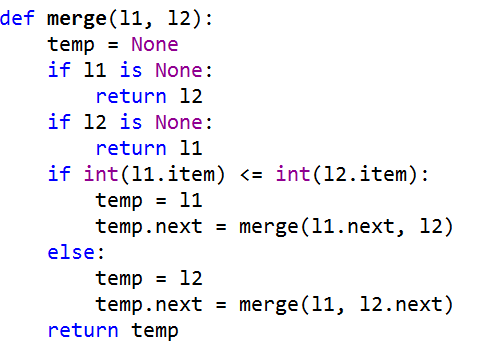
If we know that we have duplicates we need to erase them from the linkedlist, basically what I did was creating another method called repeated that unpoint the number from the linked list



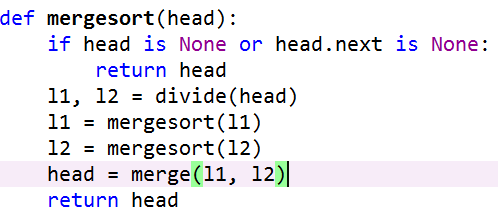
For problem 3, I needed to sort the list using merge sort in a recursive way, then I had to determine if there are duplicates by comparing each item with the item that follows it in the list.

In my merge sort solution, I had to use three methods in order to be cleaner,

The first and more important method is the split, because you need to split the input to be faster I started with two initial heads checking id the next node is not none, if is not our transversal head will move to the next one ( this if statement works as a base case because we need to do this recursively), then we need to move the initial and the moving head to the next node once we did this we are ready to split using the last node unpointing it and returning its head and its middle.

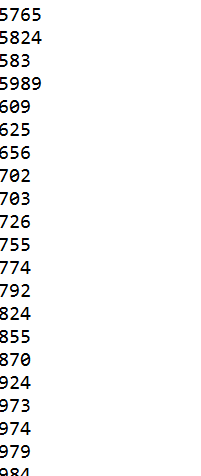
 Then knowing that we have two lists we have to check if any of them are empty, then compare the two lists with its item, if the first item is less or equal than its second wee need to do a recursive call with the next node and the same second one, if the case is greater or equal we finally have to do another recursive call with the first node and the second next one.

Finally the last step we have to do is to send our linked list and use them into a main merge sort method which receives it and split the linked list in two, finally after we split it we call the same method which will check in the step 2 if one of them is breaking the case where L2 > than L1



**Experimental results:**

I received a problem at first the merge sort method was not working as it suppose to work, than a T.A. told me that I have to compare them using int() because the computer was not checking correctly what it was checking was only the directions of the l1.item and l2.item



After this we are ready to compare bubble sort and merge sort, so I added a time counter which will help us to know which one is better, as my computer is not very powerful, what I did is to create two shorter files and run them and comparing its time.

|  |  |
| --- | --- |
| Bubble Sort | Merge Sort |
|  |  |
| Big O(n2) | Big O(nlogn) |
| Even if in a short text file, they are pretty much the same speed it is slower when we have a bigger file | Using merge sort, it is faster we use divide and conquer. |
| We can see a huge difference in space complexity which bubbles ort takes less O(1) | In space complexity merge sort takes O(n) which is worst than bubble sort |

**Conclusion:**

This lab was challenging, after finishing my lab I could know the difference between merge and quick sort, it is really important to know what your problem is in order to know which one you need to use

**Source Code:**

import time

class Node:

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

self.item = item

self.next = next

class LinkedList:

def \_\_init\_\_(self, head=None):

self.head = head

def add\_last(self, item):

if self.head is None:

self.head = Node(item, self.head)

return

curr = self.head

while curr.next is not None:

curr = curr.next

curr.next = Node(item)

def add\_first(self, item):

self.head = Node(item, self.head)

def add(self, index, item):

if index == 0:

self.head = Node(item, self.head)

return

if index < 0:

return

curr = self.head

for i in range(index - 1):

if curr is None:

return

curr = curr.next

if curr is not None:

curr.next = Node(item, curr.next)

def print\_list(self):

current = self.head

while current is not None:

print(current.item)

current = current.next

def contains(self, item):

return self.index\_of(item) != -1

def index\_of(self, item):

curr = self.head

i = 0

while curr is not None:

if curr.item == item:

return i

i += 1

curr = curr.next

return -1

def get(self, index):

if index < 0:

return None

curr = self.head

for i in range(index):

if curr is None:

return None

curr = curr.next

return None if curr is None else curr.item

def get\_first(self):

if self.head is None:

return

else:

return self.head.item

def get\_last(self):

if self.head is None:

return None

curr = self.head

while curr.next is not None:

curr = curr.next

return curr.item

def remove\_same(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

curr = self.head

for i in range(index - 1):

if curr is None:

return

curr = curr.next

if curr is not None and curr.next is not None:

curr.next = curr.next.next

def remove\_first(self):

if self.head is not None:

self.head = self.head.next

def remove\_last(self):

if self.head is None or self.head.next is None:

self.head = None

return

curr = self.head

while curr.next.next is not None:

curr = curr.next

curr.next = None

def remove(self, index):

if index < 0:

return

if index == 0:

self.remove\_first()

return

curr = self.head

for i in range(index - 1):

if curr is None:

return

curr = curr.next

if curr is not None and curr.next is not None:

curr.next = curr.next.next

def size(self):

curr = self.head

length = 0

while curr is not None:

length += 1

curr = curr.next

return length

def is\_empty(self):

return self.head is None

def biggest(self):

curr = self.head

biggest = 0

while curr is not None:

if biggest < int(curr.item):

biggest = int(curr.item)

curr = curr.next

return biggest

def repeated(self):

curr = self.head

if curr.next is None :

return

while curr.next is not None:

if curr.item == curr.next.item:

curr.next = curr.next.next

curr = curr.next

def bubblesort(self):

counter = 0

start = self.head

first = self.head

second = first.next

while first.next is not None :

if second is None :

if counter == 0 :

return

counter = 0

if int(first.item) > int(second.item) :

temp = second.item

second.item = first.item

first.item = temp

counter += 1

if counter != 0:

first = start

second = first.next

else :

first = second

second = first.next

def solution1(self):

curr = second = self.head

while curr is not None:

while second.next is not None:

if second.next.item == curr.item:

second.next = second.next.next

else:

second = second.next

curr = second = curr.next

def solution2 (self):

self.bubblesort()

self.repeated()

def solution3(self):

self.head = mergesort(self.head)

self.repeated()

def solution4(self):

bool\_array = [False] \* (self.biggest()+1)

curr = self.head

while curr is not None:

if bool\_array[int(curr.item)] == True:

self.remove(self.index\_of(int(curr.item)))

else:

bool\_array[int(curr.item)] = True

curr = curr.next

return

def mergesort(head):

if head is None or head.next is None:

return head

l1, l2 = divide(head)

l1 = mergesort(l1)

l2 = mergesort(l2)

head = merge(l1, l2)

return head

def merge(l1, l2):

temp = None

if l1 is None:

return l2

if l2 is None:

return l1

if int(l1.item) <= int(l2.item):

temp = l1

temp.next = merge(l1.next, l2)

else:

temp = l2

temp.next = merge(l1, l2.next)

return temp

def divide(head):

moving = head

initial = head

if moving is not None:

initial = initial.next

while initial is not None:

initial = initial.next

if initial is not None:

initial = initial.next

moving = moving.next

mid = moving.next

moving.next = None

return head, mid

def read(LL):

file = open("vivendi.txt", 'r')

file2 = open("activision.txt", 'r')

for i in file.read().split('\n'):

LL.add\_last(i)

print(i)

for i in file2.read().split ('\n'):

LL.add\_last(i)

print(i)

def main():

LL = LinkedList()

read(LL)

start\_time = time.time()

LL.solution1()

#LL.solution2()

#LL.print\_list()

#print("--- %s seconds ---" % (time.time() - start\_time))

LL.solution3()

LL.print\_list()

print("--- %s seconds ---" % (time.time() - start\_time))

#LL.solution4()

Main

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Academic Honesty**

“I certify that this project is entirely my own work. I wrote, debugged, and tested the code being presented, performed the experiments, and wrote the report. I also certify that I did not share my code or report or provided inappropriate assistance to any student in the class.”