**CS 2302 - Data Structures**

**Fall 2019**

**Project 2 - Option A**

**Overview**

In December 2007, Activision and Vivendi Games (owner of Blizzard Entertainment and Sierra Entertainment) decided to merge into a single company: Activision Blizzard. Due to this event, a lot of system infrastructure had to be merged; including employee databases. Let’s assume each company had its own employee database, where each employee was given a unique ID.

Let’s imagine that Activision Blizzard made an interesting decision when merging the two employee databases. Instead of creating a new ID for every Activision Blizzard employee, they decided to let employees keep their old IDs. They knew this could cause problems if two employees coming from different companies shared the same ID, so they decided to create a system to detect these collisions.

Your job is to tackle this problem using your linked-list skills! You are given two 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.

After inspecting the files for a minute, you make the following observations:

* There are 4k records in *activision.txt*
* There are 2k records in *vivendi.txt*
* Activision’s IDs start at 0 and end at 6000 (inclusive)
* Vivendi’s IDs start at 0 and end at 5000 (inclusive)

Your job is to write a Python 3 program that:

* Reads the employee IDs from both files and creates a single (just one) liked list that stores all of the IDs.
* Implements the following solutions to find employee ID duplicates:
  + 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.

Determine the big-O running time of each of the previous solutions (1 to 4). Illustrate your results by means of plots and/or tables. Create your own *activision.txt* and *vivendi.txt* files to perform this analysis.

Use the following Node class to construct your linked list:

**class** Node(object):

item = -1

next = **None**

**def** \_\_init\_\_(self, item, next):

self.item = item

self.next = next

**What you need to do**

**Part 1 - Due Thursday, September 19, 2019**

Upload the progress you have made. Have at least two of the above-mentioned solutions already implemented.

**Part 2 - Due Tuesday, September 24, 2019**

Final due date (everything finished).

**Extra Credit**

Modify your implementation of merge sort by replacing recursion with a stack. You do not have to implement your own Stack data structure, you can uses Python’s.

**Rubric**

|  |  |  |  |
| --- | --- | --- | --- |
| **Criteria** | **Proficient** | **Neutral** | **Unsatisfactory** |
| **Correctness** | The code compiles, runs, and solves the problem. | The code compiles, runs, but does not solve the problem (partial implementation). | The code does not compile/run, or little progress was made. |
| **Space and Time complexity** | Appropriate for the problem. | Can be greatly improved. | Space and time complexity not analyzed |
| **Problem Decomposition** | Operations are broken down into loosely coupled, highly cohesive methods | Operations are broken down into methods, but they are not loosely coupled/highly cohesive | Most of the logic is inside a couple of big methods |
| **Style** | Variables and methods have meaningful/appropriate names | Only a subset of the variables and methods have meaningful/appropriate names | Few or none of the variables and methods have meaningful/appropriate names |
| **Robustness** | Program handles erroneous or unexpected input gracefully | Program handles some erroneous or unexpected input gracefully | Program does not handle erroneous or unexpected input gracefully |
| **Documentation** | Non-obvious code segments are well documented | Some non-obvious code segments are documented | Few or none non-obvious segments are documented |
| **Report** | Covers all required material in a concise and clear way with proper grammar and spelling. | Covers a subset of the required material in a concise and clear way with proper grammar and spelling. | Does not cover enough material and/or the material is not presented in a concise and clear way with proper grammar and spelling. |