**CS 2302 - Data Structures**

**Fall 2019**

**Project 2 - Option B**

**Overview**

In 2015, Mark Burnett released 10 million passwords people use to access online accounts. Your job is to download the passwords file from [this link](https://xato.net/today-i-am-releasing-ten-million-passwords-b6278bbe7495), and write a Python 3 program that finds the 20 most used password.

To accomplish this, your program must create a linked list to store all of the passwords contained in the file. Your linked list must not contain any duplicates, so each node must store the following values:

* + A unique password extracted from the file (string)
  + The number of times the password appears in the file (integer)
  + A reference called ‘next’ that points to the next node in the linked list

You are required to code two different solutions to create this list:

* + Solution A: Every time you read a password from the file, check (using a loop) if that password has already been added to the linked list. That is, you need to traverse the linked list to see if that password has been added already. If the password is already in your linked list, update the number of times the password has been seen in the file. Otherwise, add a the password to the linked list.

* + Solution B: This is a variation of Solution A. Instead of traversing the linked list to check if a password has been seen before, we will be using what is called [a dictionary](https://docs.python.org/3/tutorial/datastructures.html#dictionaries). Read the following code snippet to understand how to use a dictionary in a similar context:

list\_with\_duplicates = [**"utep"**, **"go"**, **"utep"**, **"utep"**, **"miners"**, **"go"**, **"miners"**]

dict = {}

**for** item **in** list\_with\_duplicates:

**if** item **in** dict: *# You can assume this operation takes O(1)*

dict[item] = dict[item] + 1

**else**:

dict[item] = 1

print(dict[**"go"**]) *# 2*

print(dict[**"utep"**]) *# 3*

print(dict[**"miners"**]) *# 2*

Once the linked list has been created, implement the following solutions to find the 20 passwords that appear the most in the file:

* + Solution A: Sort the list (in descending order) using bubble sort, and print the 20 most used passwords along with the number of times they appear in the file.
  + Solution B: Sort the list (in descending order) using merge sort, and print the 20 most used passwords along with the number of times they appear in the file.

Determine the big-O running time of each of the previous solutions (2 + 2). Illustrate your results by means of plots and/or tables. Create your own password files to perform this analysis.

Use the following Node class to construct your linked list:

**class** Node(object):

password = **""**

count = -1

next = **None**

**def** \_\_init\_\_(self, password, count, next):

self.password = password

self.count = count

self.next = next

**What you need to do**

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

Upload the progress you have made. Have at least one 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 |
| **Code Review** | Useful feedback was provided to team members.  Feedback received from team members was used to improve the code. | Feedback was provided to team members, but it was not very useful.  Feedback received from team mates was partially used to improve the code | Little to no feedback was provided to team mates.  Received feedback was not used to improve the code. |
| **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. |