Wesly Lim

Spiketrap.io Coding Challenge

Algorithm and Procedure:

The inputs are given as a list of strings and a dictionary mapping *GameID* to *GameAlias:*

str_list = ["string string string", "string string string", ...]
dict = {key1:[val, val, val...], key2:[val, val, val...], …]

- Iterate through every element in *str_list* and separate all words and punctuation marks into a list, separated by a space.
- Iterate through every word in the list to see if it is a possible candidate for an alias. An alias list of possible candidates is built.
  - If the word is a possible candidate, call the helper function *_detect_game(list, words, index)* to help check if the next words are in the alias list. Returns the alias and end index if it is an alias and None otherwise.
    - Note that punctuation ([.,!?;]) is ignored when attached to words using the re library (Python regular expressions)
- When alias is found, replace it with the string *"TAG{GameID, GameAlias}".*
- Program returns the modified *str_list,* prints it, and then ends.

Time Complexity and Proof:

The following is the proof and time complexity for each step:

1) As the baseline, every word in the string list must be accessed; if the *str_list* has *n* elements, the program will visit every single word. This takes *O(n).*

2) In the dictionary *dict,* there are *m* different *GameAlias*es. This algorithm checks if each word in *str_list* is a potential candidate *GameAlias.* This means for each word, the algorithm also visits every value in *dict* of size *m*. This takes *O(m).*

3) Each *GameAlias* may contain more than one word, so the program must look ahead the next few words. However, the code is optimized so that it does not look through the whole *dict.* Instead, another list of potential candidates for *GameAlias* is created.
   - The size of the optimized list $j \geq m$
   - Note that the whole string in *str_list* is not checked; it is cut off at the maximum *GameAlias* length:
     - $n \geq max(len(GameAlias), len(string[start:]) = k$
   - This takes $O(n * m) \geq O(j * k)$.

The brute force method checks every single word in the string in *str_list* and checks to see if each word or phrase (collection of words) is an alias. This takes $O(n^2 * m^2)$. The optimized code takes $O(n * m * j * k) \geq O(n^2 * m^2)$. This might not seem like a huge improvement, but $O(n * m * j * k) = O(n^2 * m^2)$ is only true in the instances where *j* is the size of *dict* $(j = m)$ and if *GameAlias* is as long as the whole string in *str_list* $(k = n)$. On average, the optimized code is significantly faster.

Look at the first string in the provided example string list given: "I liked <u>the last Destiny game</u>, now I play <u>Fortnite</u>". The constructed optimized list is of size $j = 2$ (because "the" appears in two *GameAliases)* while $m = 13$. In this example, $k = 4$ (because <u>the last Destiny game</u> has 4 words) while $n = 10$.

In conclusion, the optimized time complexity for running the Game Detector is $O(n * m * j * k)$, which is a lot better than the brute force method on average.

Technologies/How to Run Code:

Python 2 was used to code this project, and Sublime Text was used as the text editor. The code is ran on console with the following:

```
python Game_Detector.py
```

All files must also be within the same directory if all unit tests want to be used as well. The assertions at the bottom tests the basic and edge cases of the code.

Files submitting:

```
    Game_Detector.py
    test_cases.py
    Documentation.pdf
```