CS 120 (Spring 22): Introduction to Computer Programming II

Long Project #10 - Bob

`https://www.youtube.com/watch?v=JUQDzj6R3p4`

due at 5pm, Tue 5 Apr 2022

REMEMBER: The `itertools` and `copy` libraries in Python are **banned.**

1 Background: Combinatorics

One of the most expensive things computers ever do, in terms of CPU time, are problems of the form "try all possible combinations of X." These sorts of problems take **exponential time** - which we often write $O(2^n)$. You thought $O(n^2)$ was bad? This is far, far worse - with only $n = 20$, we're already up at a cost of 1 million - and by $n = 30$, it's a billion!

In this project, you will be taking a dictionary of words (just a hundred or two) and combining them in many different ways. This is going to be exponentially slow - which means that we'll only be able to do a **very small number of rounds** of our algorithm.

Could we do better? Could we avoid the exponential explosion? Maybe - but that would make the program a lot more complex. **This is worth doing!** But it's not something we can do in a Short Problem for 120.

2 Overview

Name your file `bob.py` .

In this project, you will read words from a text file. There's no format to the file, other than that the words are separated by whitespace. You will do a little bit of "cleaning up" of the words (getting rid of punctuation, etc.) and then you will combine strings of the words together, looking for palindromic phrases:
`https://en.wikipedia.org/wiki/Palindrome#Sentences_and_phrases`
You will then combine these words, in various ways, looking for palindromes.

3 Background: Sets in Python

A reminder, if you haven't used them before: a `set` in Python is a data structure which only keeps a single copy of each element. Unlike an array, it doesn't keep them in any particular order, but it is possible to write a `for` loop over them - you just don't know what order they will be in. Likewise, it's possible to sort the elements of a set; just pass the set to the `sorted()` function, which will return an array, containing all of the values in the set, in order.

For this project, you will only need a few additional features of a `set`. First, you will need to create an empty one:

```
x = set()
```

and second, you will need to be able to add a new value to the set:

```
x.add("abc123")
```

(Note that `add()` is kind of like `append()` on an array, except that order doesn't matter, and adding a duplicate is simply a NOP.)

Finally, note that the values placed in a set have the same requirement as keys used in a dictionary: they must be **immutable.** So you can place integers, strings, and tuples in a set - but not arrays, dictionaries, or other sets. (Python has an immutable version of set, named `frozenset`, but you won't need it for this project.)

4 Input

You will read two lines of input from the keyboard. The first will be the name of a file (if it doesn't exist, report an error and terminate the program; see the testcases for the correct error message). You will read the file, and get all of the words from it.

The second line of input will either be the word "dump" (with no spaces or other text), or it will be an integer. If it is "dump", then simply print out all of the words that you've read from the file, sorted. (See the testcases for the proper format.)

If the second line of input is not "dump", then convert it to an integer (you may assume that the input will convert OK, you don't need to do error checking). You will then search for palindromes in three ways: three simple loops that you will write yourself, and then you will run the algorithm I've provided, up to size n, printing out whatever palindromes you find.

4.1 File Format and Word Filtering

The file that you read has no particular format; it's just text. The words will be separated by whitespace.

Every word that you read from the file must be filtered, to remove spurious characters. First, remove all characters other than upper and lowercase letters; second, convert it to lowercase.

Finally, check the length: any word that is shorter than 2 characters, after filtering, should be **ignored;** don't store it into your list of words.

(spec continues on the next page)

5 Required Output, Part 1; Simple Palindromes

(Skip this if the user types "dump".)

Scan the list of words that you've read from the file, and print out all palindromes in the list. These are **single words** which are palindromes themselves, like "racecar" or "wow."

Print the words in alphabetical order. See the testcases to see exactly the format.

6 Required Output, Part 2: 2- and 3-Word Palindromes

(Skip this if the user types "dump".)

Using nested `for` loops, find all 2-word and 3-word palindromes that can be created from your input words, such as "warsaw was raw" or "stressed desserts." Mix the 2- and 3-word palindromes into a single group, and print them out (with no spaces between words), one per line. Print them in alphabetical order, and if there are any duplicates, only print one copy of each. (See the testcases for the format.)

7 Required Output, Part 3: Many Palindromes, By Length

(Skip this if the user types "dump".)

Finally, you will generate many palindromes by combining words together. The algorithm below will find single-word, 2-word, and 3-word palindromes (if they are short), but in theory could also find longer ones. However, we'll cut this algorithm off after a few rounds, because the number of words it generates (and thus the total CPU time) gets **huge.** Thus, in practice, many of the simple palindromes you found, earlier in the program, won't get listed here - because they have too many characters.

The basic idea of this algorithm is to build up a list of all possible phrases that can be made by combining the words together in various sequences. We will scan through these words, starting with the phrases with the fewest number of characters; when we find palindromes, we will print them out, and when we find things that are *NOT* palindromes, we will create new, longer phrases by adding new words to the end of the phrase.

7.1 Data Structure

Our data structure will be an array (or, if you prefer, a dictionary) of sets. Each element in the array (or, if you prefer, each key in the dictionary) represents a different length - so all of the length 3 phrases will be together in a set, all of the length 4 phrases will be together in a different set, and so on.

Each set will contain strings. Each string will be a phrase, made by combining one or more words together, without spaces between them.

Note that the strings in the sets are **not** necessarily palindromes; instead, they are phrases which we hope to **turn into** palindromes eventually. From time to time, we will find a palindrome, and put it in the set; we'll print it out later. However, **most** of the strings in any given set are **not** going to be palindromes.

7.2 Algorithm

The algorithm works as follows:

```
put all of the words from the file into the data structure
    put each word in its proper set, based on its length
    include palindromes in this initial setup

for all lengths, from 1 (inclusive) to the max length n (inclusive):
    print out a header line, showing how many words are in the set (see testcases)
    scan the set for length n:
        for each palindrome:
            print it out (print the palindromes in alphabetical order)
        for each non-palindrome:
            extend it, by adding all possible words to it (create many new phrases)
            store each new phrase in the data structure, in its proper set
            never extend palindromes
            also never *ADD* a palindrome word to an existing phrase
```

8 Turning in Your Solution

You must turn in your code using GradeScope. Name your file `bob.py` .