HW 7: Recursion

In this homework assignment you will solve a series of problems using recursion.

1. HashTag Tokenizer (#iLoveComputerScience)

A hashtag is a word or phrase that is prefixed with a hash sign (#) and is widely used on social media platforms to categorize posts on specific topics. Typically, a hashtag is formed by concatenating several words together and eliminating all white spaces. For instance, #ilovethecampus is an example of a hashtag.

In this exercise, our objective is to develop a program that processes a hashtag (excluding the hash sign) and outputs the individual words that comprise the hashtag. To determine whether a given string is a valid word, we will utilize a dictionary containing the 3,000 most commonly used words in English. The exercise will be completed in three stages: First, we will write a function to read and load the dictionary from a file. Next, we will write another function that checks whether a given word is present in the dictionary. Finally, we will write a recursive function that accepts a hashtag (without the hash sign) as input and recursively decomposes it into its constituent words, printing them out in the process.

For example, for the hashtag "ilovetheuniversity" the program will print the words as follows:

```
% java HashTagTokenizer ilovetheuniversity
i
love
the
university
```

We now proceed to describe the three functions that, taken together, will perform this operation.

Read Dictionary

This function, `public static String[] readDictionary(String fileName)`, is designed to read a local file of words, and store them in an array. You have been provided with a file named "`dictionary.txt`", which contains the 3,000 most frequently used words in English. The structure of the file is straightforward, with each line representing a single word. The function starts by building an array of 3,000 strings. It then reads each line from the file, and stores it in the array (one after the other, in the order in which they are read). To implement the reading, use the `In` class, as we did in previous homeworks and lectures. The given `HashTagTokenizer.java` file contains a partial implementation of this function. Please complete the implementation.

Exist in Dictionary

The function `public static boolean existInDictionary(String word, String[] dictionary)` is designed to take a string as an input and determine its presence in the dictionary. It returns `true` if the word is found within the dictionary array, and `false` if it is not.

Break Hashtag

This function receives two inputs: a hashtag (as a `String`) and a dictionary (an array of `String`). Its purpose is to print each word embedded within the hashtag on a separate line. The primary approach taken here is to incrementally analyze prefixes of the hashtag, starting with the first character (which can be obtained using `hashtag.substring(0,1)`) and gradually extending the length of the prefix. Initially, the function checks if the single-character prefix is a valid word in the dictionary, using the `existInDictionary` method. If this prefix is indeed a word, it is printed, and the function recursively calls itself, this time with the hashtag minus its first character. If the prefix is not a valid word, the prefix length is increased by one character (now analyzing `hashtag.substring(0, 2)`, and the process repeats.

This recursive approach continues until either a valid prefix-word is found (and subsequently printed), or the end of the hashtag is reached. At this point, the function should end without printing, as it indicates either the entire hashtag has been parsed into valid words, or a non-dictionary word has been encountered. For example, with the hashtag "iloverecursion", the function will print the words "i" and "love" as recognized in the dictionary, stopping when it reaches "recursion" since "recursion" is not a recognized word in this dictionary.

```
% java HashTagTokenizer iloverecursion

i
love
```

Note that all the words in the provided dictionary are in lowercase. To allow the user to include uppercase letters in the hashtag, it is necessary to convert the entire hashtag to lowercase before processing it. By the way, preparing the input before executing the main logic on it is typically referred to as "pre-processing". To convert the hashtag to lowercase you can use the `String.toLowerCase()` method. This conversion ensures that the hashtag is consistently formatted for comparison against the dictionary entries.

Complete the recursive implementation of the `breakHashTag` function by following the guidelines provided in the body of the function.

Once implemented, you can run your program on different hashtags and check the results.

2. Levinstein Distance and Spell Checking

The Levenshtein distance, also known as the *edit distance*, is a string metric for measuring the difference between two strings. It is named after the Soviet mathematician Vladimir Levenshtein, who studied this distance in 1965. The edit distance between two words is the minimum number of single-character edits (insertions, deletions or substitutions) needed to transform one word into the other word. Here are some examples:

```
levenshtein(“hello”, “hell”);  // 1 (one deletion)
levenshtein(“hello”, “hell0”); // 1 (one substitution)
levenshtein(“love”, “i”);      // 4 (one substitution, three deletions)
levenshtein(“concensus”, “consensus”);         // 1
```

One of the main applications of the Levinstein distance is spell checking. A straightforward method for spell checking involves comparing the similarity between a misspelled word and words from an English dictionary. The similarity can be calculated based on the edit distance between the words. In this exercise we will implement a recursive function that calculates the edit distance between two words. We will then use it in another function for implementing a basic spell checking application.

Levenshtein

In this exercise, we will develop a recursive implementation of the Levenshtein distance function. Your task is to implement the following function: `public static int levenshtein(String word1, String word2)`. The function accepts two strings as input and returns the edit distance between these two words, as an integer. Your implementation can be based directly on the following formal definition of the Levenshtein distance:

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) & \text{if } \operatorname{head}(a) = \operatorname{head}(b), \\ 1 + \min \begin{cases} \operatorname{lev}\big(\operatorname{tail}(a), b\big) \\ \operatorname{lev}\big(a, \operatorname{tail}(b)\big) \\ \operatorname{lev}\big(\operatorname{tail}(a), \operatorname{tail}(b)\big) \end{cases} & \text{otherwise} \end{cases}$$

(reference: https://en.wikipedia.org/wiki/Levenshtein_distance)

- `a, b` are the two words (word1 and word2 in our implementation). $|a|$ is the length of a (number of characters).
- `lev(a,b)` should be implemented by our `levenshtein(String word1, String word2)` function.
- `head(a)`, is the first character of the string `a`.
- `tail(a)`, is the word `a`, excluding the first character. If the length of `a` is 1, then `tail(a)` is an empty string.

Our goal is to incorporate the Levenshtein distance in the spell checker. Therefore, it is essential that your implementation will be case-insensitive. For example, the edit distance between 'SPELL' and 'spell' should be calculated as 0, indicating no difference despite the variation in case. This approach ensures that the spell checker accurately assesses the similarity of words regardless of their case.

Read Dictionary

Same implementation as in `HashTagTokenizer.java`. We will use the same `dictionary.txt` file.

Spell Checker

The function `public static String spellChecker(String word, int threshold, String[] dictionary)` receives a word, a threshold value for distance, and a dictionary as inputs. It returns the word from the dictionary that most closely resembles the given word. This similarity is determined using the Levenshtein function: The word in the dictionary with the smallest Levenshtein distance to the given word is considered the most similar. If the smallest distance found is greater than the threshold value, this indicates that no word in the dictionary is sufficiently similar to the given word. In this case, the function returns the original word. Here are some examples:

```
spellChecker("hell0", 1, dictionary);  // "hello" since levenshtein("hell0", "hello") == 1

spellChecker("hello", 1, dictionary);  // "hello" since levenshtein("hello", "hello") == 0

spellChecker("coooool", 2, dictionary);  // "coooool" since the most similar word from the
                                            dictionary is at a distance greater than 2

spellChecker("coooool", 3, dictionary);  // "control" since levenshtein("coooool",
                                           "control") == 3)

spellChecker("concensus", 1, dictionary); // "consensus"
```

Your task is to develop an iterative version of the `spellChecker` function. Write a loop that calculates the edit distance between the given word and each word in the dictionary. Identify the word with the minimum distance, check whether this distance exceeds the given threshold, and proceed with the appropriate action based on this evaluation.

The supplied `SpellChecker.java` file includes a `main` function that accepts two command-line arguments: the word to be checked and the threshold value. For proper grading of your work, make sure that the `main` implementation remains unchanged upon submission.

Here is an example of running the spell checker:

```
% java SpellChecker lisense 2

license
```

You are encouraged to test this implementation on frequent spelling errors to observe the outcomes. While the typical threshold value used is 2, feel free to experiment with other values (for more liberal or more strict error checking).

Efficiency Issues

The recursive implementation of the Levenshtein algorithm is simple, elegant, and inefficient. This may result in slow execution. Think about how this algorithm can be made more efficient. There is no need to write or submit these ideas, but do think about them.

Another source of inefficiency is the array structure of the dictionary. More suitable data structures can be used, as you will learn later in this and other CS courses.

Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines. Submit the following filed only:

- `HashTagTokenizer.java`
- `SpellChecker.java`

Submission deadline: March 15th, 2024, 23:55

Problem Set (self-study, not to be submitted)

1. Write a recursive function that checks if a given string is a palindrome.

2. Write a recursive function that computes the sum of digits of a given integer.

3. Write a recursive function that counts the number of zeros in a given integer.

4. Write a recursive function that prints the binary representation of a given integer.

5. Write a recursive function that returns the greatest common divisor (GCD) of two given integers.