```java
public class HashTagTokenizer {

    public static void main(String[] args) {

        String hashTag = args[0];
        String[] dictionary = readDictionary("dictionary.txt");
        breakHashTag(hashTag, dictionary);
    }

    public static String[] readDictionary(String fileName) {
        String[] dictionary = new String[3000];

        In in = new In(fileName);

        int index = 0;
        while (!in.isEmpty() && index < dictionary.length) {
            String line = in.readLine(); // reads a lkine from the file
            if (line != null && !line.isEmpty()) {
                dictionary[index] = line; // store line in array
                index++;
            }

        }

        return dictionary;
    }

    public static boolean existInDictionary(String word, String[] dictionary) {
        int length = dictionary.length;
        for (int i = 0; i < length; i++) {
            if (word.equals(dictionary[i])){
                return true;
            }
        }
        return false;
    }
```

```java
    public static void breakHashTag(String hashtag, String[] dictionary) {

        hashtag = hashtag.toLowerCase(); // pre-processing. turns hashtag String into lower case
        // Base case: do nothing (return) if hashtag is an empty string.
        if (hashtag.isEmpty()) {
            return;
        }

        boolean wordFound = false;
        int N = hashtag.length();

        for (int i = 1; i <= N; i++) {
            String innerWord = hashtag.substring(0, i);

            if (existInDictionary(innerWord, dictionary)) { // IF inner word is valid word in dictionary it prints it

                System.out.println(innerWord);

                // initiates the recursive case for the rest of the hashtag, starting at the end
                // of the current "inner word"
                // checks if the current index i (which also represents the length of the
                // current innerWord that has been found to match a word in the dictionary) is less than the total length N
                if (i < N) {
                    breakHashTag(hashtag.substring(i), dictionary);
                }
                wordFound = true;
                break;
            }

        }
    }

}
```

```java
public class SpellChecker {

    public static void main(String[] args) {
        String word = args[0];
        int threshold = Integer.parseInt(args[1]);
        String[] dictionary = readDictionary("dictionary.txt");
        String correction = spellChecker(word, threshold, dictionary);
        System.out.println(correction);
    }

    public static String tail(String str) {
        if (str == null || str.length() <= 1) {
            return "";
            // there is no tail so we return the empty String
        } else {
            // String has more than one character, we return substring that excludes the
            // first letter
            return str.substring(1);
        }
    }
```

```java
public static int levenshtein(String word1, String word2) {
    word1 = word1.toLowerCase();
    word2 = word2.toLowerCase();

    // Base Case
    if (word1.isEmpty()) {
        return word2.length();
    }
    if (word2.isEmpty()) {
        return word1.length();
    }

    // initializing "cost" fo9r each levenshtein operation
    int cost;
    if (word1.charAt(0) == word2.charAt(0)) {
        cost = 0; // If the first characters of both words are the same, cost is 0
    } else {
        cost = 1; // If the first characters are different, cost is 1
    }

    // Recursive case
    // calculate distances for deletion, insertion, and substitution
    int deletionCost = levenshtein(tail(word1), word2) + 1;
    int insertionCost = levenshtein(word1, tail(word2)) + 1;
    int substitutionCost = levenshtein(tail(word1), tail(word2)) + cost;

    // since we are looki9ng for the minimum amount of operations we need to return
    // a min
    return Math.min(Math.min(deletionCost, insertionCost), substitutionCost);

}

public static String[] readDictionary(String fileName) {
    String[] dictionary = new String[3000];

    In in = new In(fileName);
```

```
        int index = 0;
        while (!in.isEmpty() && index < dictionary.length) {
            String line = in.readLine(); // reads a line from the file
            if (line != null && !line.isEmpty()) {
                dictionary[index] = line; // store line in array
                index++;
            }

        }

        return dictionary;
    }

    public static String spellChecker(String word, int threshold, String[] dictionary) {
        String closestMatch = word;
        int minDistance = threshold + 1; // Initialize with a practical limit based on the threshold, we are ooking for min distance within the threshold

        for (String dictWord : dictionary) {
            int distance = levenshtein(word.toLowerCase(), dictWord.toLowerCase());
            if (distance <= threshold && distance < minDistance) {
                minDistance = distance;
                closestMatch = dictWord;
                if (distance == 0) {
                    break; // Break early if an exact match is found
                }
            }
        }

        // Check if the minimum distance found is within the threshold
        if (minDistance <= threshold) {
            return closestMatch; // A sufficiently similar word was found within the threshold
        } else {
            return word; // No similar word within the threshold was found, return the original word
        }
    }
```

```
}
```