Solution: Verifying an Alien Dictionary

Let's solve the Verifying an Alien Dictionary problem using the Topological Sort pattern.

We'll cover the following

Statement

You're given a list of words with lowercase English letters in a different order, written in an alien language. The order of the alphabet is some permutation of lowercase letters of the English language.

We have to return TRUE if the given list of words is sorted lexicographically in this alien language.

Constraints:

- $1 \leq$ `words.length` ≤ 100
- $1 \leq$ `words[i].length` ≤ 20
- `order.length` $== 26$
- All the characters in `words[i]` and `order` are lowercase English letters.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which to follow based on considerations such as time complexity and implementation constraints.

Naive approach

The naive approach for this problem is to iterate over the order list and words simultaneously. Start from the first word, then compare the order with all other words present in the list.

The time complexity for the naive approach is $O(n^3)$, since we're iterating the order list and the complete list two times in nested loops. The space complexity of this naive approach is constant because we didn't use any extra memory.

Optimized approach using topological sort

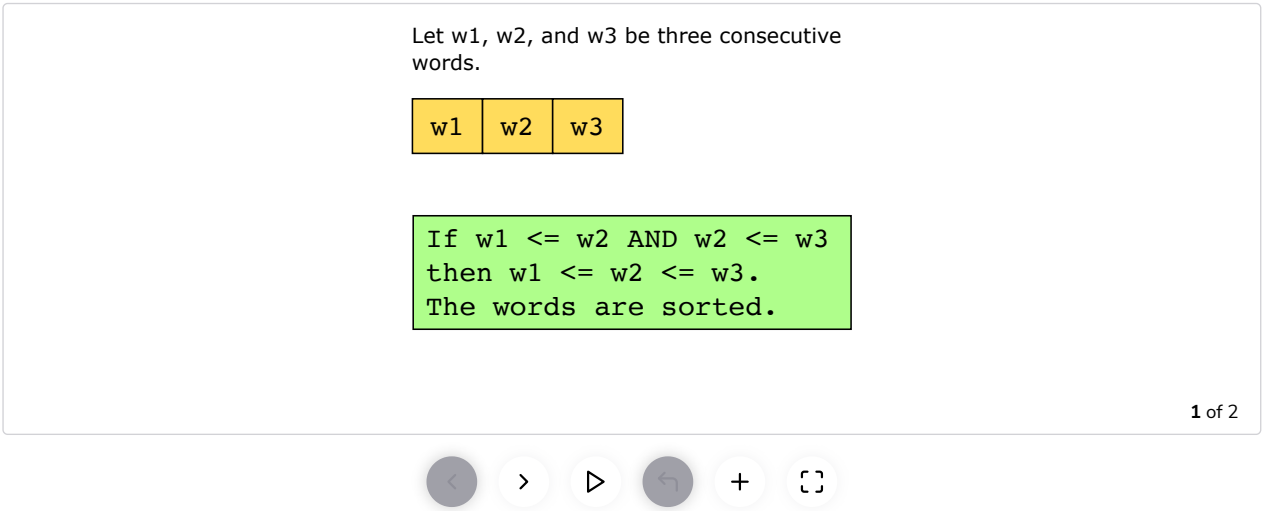
We can solve this problem using the topological sort pattern. Topological sort is used to find a linear ordering of elements that depend on or prioritize each other. For example, if A depends on B or if B has priority over A, B is listed before A in topological order.

For this problem, we're given the list of words and the order of the alphabet. Using the order of the alphabet, we must check if the list of words is sorted lexicographically.

We can check adjacent words to see if they are in the correct order. For each word, the word on its right should be lexicographically larger, and the one on its left should be lexicographically smaller.

One thing to notice here is that we don't need to compare all the words. We can just compare each pair of adjacent words instead. If all the pairs of adjacent words are in order, we can safely assume that the integrity is intact. Conversely, if any pair of adjacent words isn't in order, we can assume that our order is not correct.

Let's review the illustration to better understand the intuition described above with the help of a sample list of three words, $[w1, w2, w3]$:



We need an efficient way of storing the order of each letter and its ranking provided in the order list. Once we do that, we move on to the comparison part. To compare two adjacent words, `words[i]` and `words[i + 1]`, we iterate over the letters one by one and find the first index where the letter in both the words is different. If `words[i]` has a smaller letter than the corresponding one in `words[i + 1]`, we break this iteration because we know these two words are in the right order.

If `words[i]` has a lexicographically larger letter, we immediately return FALSE because we've found a pair of consecutive words that aren't in the correct order.

We also need to consider the boundaries. When we iterate over a word, we need to ensure that the other word hasn't ended, that is, all the letters in the two words match up to the point where one of the strings ends. For example, in "educated" and "educate," we can't iterate over all the letters of "educated" because the word "educate" is shorter. In this case, we must examine the length of each word. If the words are the same length or the former word is shorter, then the words list is sorted. However, if the latter word is shorter, then the words list is not sorted.

Now that we have an overview of the solution, we can move on to the actual implementation of it.

Let's discuss the algorithm for the above approach:

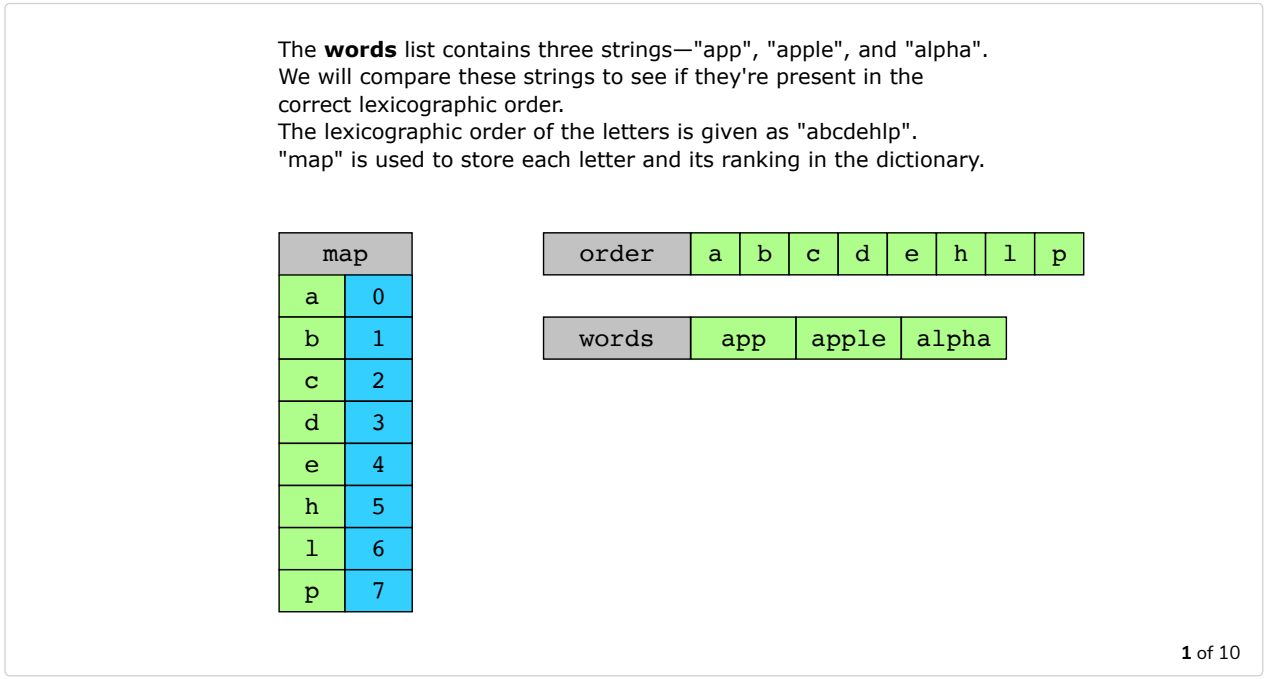
1. We initialize a hash map to record the relations between each letter and its ranking in the order list.
2. We iterate over the words and compare each pair of adjacent words.
    - We find the first index in two consecutive words (`words[i]` and `words[i + 1]`) where the letter in the two words is different.
        - If `words[i + 1]` ends before `words[i]` and no different letters are found, then we need to return FALSE because `words[i + 1]` should come before `words[i]`.
        - If we find the first different letter and the two letters are in the correct order, then we exit from the current iteration and proceed to the next pair of words.

- If we find the first different letter and the two letters are in the wrong order, then we safely return FALSE.

3. By the time we reach the end of the outer loop, we have examined all the pairs of adjacent words and ensured that they are all sorted. Therefore, we return TRUE.

Let's visualize the algorithm with a simplified example with just three alien words and an order that consists of only eight characters:

```
words = [app, apple, alpha]
order = "abcdehlp"
```



Let's look at the code for this solution below:

Java

```java
class VerifyDictionary {

  public static boolean verifyAlienDictionary(String[] words, String order) {
    // If there is only one word to check, this is a trivial case with
    // not enough input (minimum two words) to run the algorithm.
    // So we return True
    if(words.length == 1)
      return true;
    // Declare a hash map to store the characters of the words
    Map<Character, Integer> orderMap = new HashMap<>();
    // Traverse order and store the rank of each letter in orderMap
    for(int i = 0; i < order.length(); i++) {
      orderMap.put(order.charAt(i), i);
    }

    for(int i = 0; i < words.length - 1; i++) {
      // Traverse each character in a word
      for(int j = 0; j < words[i].length(); j++){
        // If all the letters have matched so far, but the current word
        // is longer than the next one, the two are not in order and
        // we return False
        if(j >= words[i + 1].length())
          return false;
        // Check if the letters in the same position in the two words
        // are different
        if(words[i].charAt(j) != words[i + 1].charAt(j)){
          // Check if the rank of the letter in the current word is
```

```
27            // Check if the rank of the letter in the current word is
28            // greater than the rank in the same position in the next word
```

Solution summary

To recap, the solution to this problem can be divided into the following parts:

1. Fill the hash map with each character of the order string and its adjacent index.
2. Iterate a word list.
3. Iterate `words[i]` string and compare each character with the character of an adjacent word from the list, `words[i + 1]`.
    - If there is no different character in the selected words, and the `words[i + 1]` end before words[i], return FALSE.

- If there is a different character and the two words are not in the correct order, we return FALSE.
- If the loop ends after iterating all of the words, we return TRUE.

Time complexity

Storing the letter-order relation of each letter takes $O(n)$ time, where n is the length of `order`.

Next, we examine each pair of words in the outer loop. In the inner loop, we check each letter in the current word. Effectively, we are simply iterating over all of the letters in the list of words. So, the time complexity of this step is $O(m)$, where m is the total number of letters in the list of words.

Hence, the total time complexity is $O(m + n)$. However, the length of `order`, n, is fixed as 26. Therefore, the time complexity is $O(m + 26)$, that is, $O(m)$.

Space complexity

We have used a hash map to store the letter-order relations for each word in `order`. Due to the fixed length of letters in `order`, this approach takes $O(1)$ space in memory.

Back
Verifying an Alien Dict...

Next
Course Schedule II

Mark as Completed