953. Verifying an Alien Dictionary

Easy | Array | Hash Table | String | Leetcode Link

Problem Description

In the given problem, we're dealing with a hypothetical alien language that uses the same lowercase letters as English but has a different ordering of these letters. This order is represented by the string `order`, where each character's position indicates its rank in the alien alphabetical order.

We're given a list of words called `words`, which are supposedly written in this alien language. Our task is to determine if these words are sorted according to the alien lexicographical order. To put it simply, we need to confirm if each word would appear before the following word in a dictionary organized by the provided alien alphabet.

The lexicographical order is similar to what we're familiar with in an English dictionary—'apple' comes before 'banana' because 'a' is before 'b' in the English alphabetical order. However, for this problem, we need to ignore the English alphabetical order and instead use the provided `order` string to determine the sorting sequence.

To solve this, we have to read the words list using the alien alphabet and confirm that each subsequent word is either the same up to a point and then greater according to the alien alphabet, or entirely greater if compared at the same position in each word.

In essence, we need to write a function that checks this sorting without reordering the list—a simple 'yes' or 'no' to whether the list is already sorted as per the alien dictionary.

Intuition

The intuition behind the provided solution approach is to use a dictionary named `m` to represent the mapping of each character in the alien language to its corresponding index according to `order`. Once we have this mapping, it is much easier to compare the words.

During the comparison, we want to ensure that for each position `i` up to the length of the longest word (in this case, arbitrarily chosen as 20, assuming no word is longer than 20 characters), the following holds true:

- If the character at position `i` in the previous word has a higher index in our alien dictionary than the character at position `i` in the current word, we should immediately return `False`, as this indicates that the list is not sorted.
- If characters at position `i` are the same, we continue our checks without concluding anything just yet because the subsequent characters might determine the order.
- If at this position `i`, we do not encounter any characters that are out of order, and all comparisons were either equal or in sorted order, we can return `True` as this means the list is sorted correctly up to this character.
- If after checking all positions `i`, we haven't returned `False`, it is safe to assume that the list is sorted according to the alien language, and we return `True`.

The code smartly exits early when it finds evidence the words are not in order, and avoids unnecessary comparisons for characters beyond the length of the shortest word, due to the handling of current and previous character index (`curr` and `prev`) with default values ensuring proper comparison even if one of the words is shorter.

Solution Approach

The solution provided makes use of a hash table (or dictionary in Python) and single-pass comparisons to implement the algorithm.

Here's a step-by-step breakdown of the implementation:

1. **Mapping the Order**: We start by creating a dictionary `m` where the key is each character of the alien alphabet and the value is its position in the `order` string. This mapping allows us to efficiently translate each character in the alien words into its corresponding rank in the alien language.

   ```python
   m = {c: i for i, c in enumerate(order)}
   ```

   Using `enumerate`, each character `c` is paired with its index `i`, effectively creating a rank ordering for the alien alphabet.

2. **Comparing Characters in Words**: The code uses a `for` loop to iterate over the positions of the characters in the words, up to a predefined limit of 20 (assuming that none of the words will be longer than that). For each position `i`, it compares characters from different words at the same index.

   ```python
   for i in range(20):
       # ...
   ```

3. **Tracking and Comparing Ranks of Characters**: Inside the loop, two variables `prev` and `curr` are used to track the rank of the current and previous characters, respectively. Initially, `prev` is set to -1 to ensure that the first comparison is valid.

   ```python
   prev = -1
   ```

4. **Checking Sorted Order**: As we iterate over each word `x` in `words`, we check if the current character's rank is less than the previous one. Since we are performing lexicographical comparison, any current character (`curr`) with a lower rank than previous (`prev`) would mean the list is not sorted.

   ```python
   curr = -1 if i >= len(x) else m[x[i]]
   if prev > curr:
       return False
   ```

5. **Validating Sequential Characters**: If the previous character is the same as the current one, `valid` is set to `False`. `valid` is used to track whether all characters compared up to that point have been the same, in which case we need to check further.

   ```python
   if prev == curr:
       valid = False
   ```

6. **Early Termination**: If at any point, we successfully pass through all the words and their characters at a certain index without returning `False`, and `valid` remains `True`, it means we found a lexicographical order and can return `True` immediately. Otherwise, we continue.

   ```python
   if valid:
       return True
   ```

7. **Final Verdict**: If we reach the end of our positional checks (the `for` loop) without finding any characters out of order, we conclude that the words list is sorted according to the alien dictionary.

   ```python
   return True
   ```

Key algorithms and data structures used in this implementation include the hash table for constant-time look-up, and a comparison framework that examines each word character-by-character using its index. The process of early termination after successful sequential checks leverages a common pattern in sorting algorithms to optimize performance by avoiding unnecessary work.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Imagine we are given the following alien order and a list of words:

```python
order = "hlabcdefgijkmnopqrstuvwxyz"
words = ["hello", "leetcode"]
```

We will walk through checking if the words are sorted according to this alien language.

1. **Mapping the Order**:
   - Using the provided `order`, we create a mapping `m` that will let us compare the characters based on their alien language order.
   - The mapping would result in: `{'h': 0, 'l': 1, 'a': 2, ... 'z': 25}`.
2. **Comparing Characters in Words**:
   - We set up a loop that will go up to 20, which is more than enough for our example.
3. **Tracking and Comparing Ranks of Characters**:
   - We initialize `prev = -1` to indicate the start of comparisons.
   - We will compare the ranks of characters at each position within the words.
4. **Checking Sorted Order**:
   - Starting with "hello", we get the rank of 'h' as 0 for position 0. There's no previous word, so we move on.
   - Next is "leetcode":
     - We compare 'l' (at position 0) of "leetcode" with 'h' of "hello".
     - In the mapping `m`, 'l' has a rank of 1 and 'h' has a rank of 0. Since 1 is greater than 0, the order is maintained.
   - Continue the loop until either the words are found to be in disorder or all characters are successfully compared.
5. **Validating Sequential Characters**:
   - When comparing characters at the same position, if they are equal, we proceed with further positions as we need more characters to determine the order.
   - If all characters are equal up to the minimum length of the two words, we also proceed to the next word in the list.
6. **Early Termination**:
   - In our example, if we found 'g' in "hello" greater than 'e' in "leetcode" at any given position, we would terminate and return `False`.
   - However, since 'h' is less than 'l' at the first position and there are no other characters in disorder, we do not terminate early and continue.
7. **Final Verdict**:
   - After checking all positions and finding no disorder, we return `True`, meaning the list is sorted according to the alien dictionary.

In practice, the comparison process stops at the first instance where `prev > curr`. As "hello" and "leetcode" are in correct order in this alien language (since 'h' < 'l'), the result of our checks would be `True`. The words are indeed sorted according to the alien alphabet given.

Python Solution

```python
class Solution:
    def isAlienSorted(self, words: List[str], order: str) -> bool:
        # Create a mapping from each alien character to its order index for easy comparison.
        order_index = {char: index for index, char in enumerate(order)}

        # Iterate up to 20 characters, assuming no word is longer than 20 characters.
        for i in range(20):
            previous_index = -1  # Start with a comparison index of -1 for the first character.
            is_sorted_until_now = True  # Flag to keep track of whether the order is sorted up to the current character index.

            # Go through all the words to compare the characters at the current index i.
            for word in words:
                # If the current index is greater than the word length, use a default index of -1.
                # Else, use the mapped order index of the current character.
                current_index = -1 if i >= len(word) else order_index[word[i]]

                # If the previous character's index is greater than the current character's index, the list is not sorted.
                if previous_index > current_index:
                    return False

                # If the previous character's index is the same as the current, we can't be sure if it is sorted and need to check th
                if previous_index == current_index:
                    is_sorted_until_now = False

                # Update the previous_index to the current character's order index for the next iteration.
                previous_index = current_index

            # If all words are sorted until now, no need to check further characters.
            if is_sorted_until_now:
                return True

        # If no problems were found in the above loop, the words list is sorted according to the alien language.
        return True

```

Java Solution

```java
class Solution {

    // Method to check if the words are sorted as per the given alien dictionary order
    public boolean isAlienSorted(String[] words, String order) {
        // Mapping from character to its position in the alien language
        int[] charIndexMapping = new int[26];
        for (int i = 0; i < order.length(); ++i) {
            // Subtract 'a' to convert char into an index (0-25), then assign its order
            charIndexMapping[order.charAt(i) - 'a'] = i;
        }

        searchLoop:
        for (int i = 0; i < 20; ++i) { // Iterate at most up to 20 times, the maximum word length
            int previousIndex = -1; // Initialize previous character index as -1
            boolean isDistinct = true; // Flag to check if all chars at position i were unique so far

            // Iterate over words to compare characters at index i
            for (String word : words) {
                // If i is greater than word length, currIndex is -1; else, get char's alien order index
                int currentIndex = i >= word.length() ? -1 : charIndexMapping[word.charAt(i) - 'a'];

                // If previous index is greater than the current index, words are not sorted
                if (previousIndex > currentIndex) {
                    return false;
                }

                // If previous index is equal to the current index, we can't confirm order; isDistinct is false
                if (previousIndex == currentIndex) {
                    isDistinct = false;
                }

                previousIndex = currentIndex; // Update previousIndex for next iteration
            }

            // If all words had distinct characters or reached the end, exit from the loop
            if (isDistinct) {
                break searchLoop;
            }
        }

        return true; // Return true if words are sorted as per alien language
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if the words vector is sorted according to the provided alien dictionary order.
    bool isAlienSorted(vector<string>& words, string order) {
        // Create a mapping from character to its index in the alien language
        vector<int> charIndexMap(26);
        for (int i = 0; i < 26; ++i) {
            // order[i] - 'a' calculates the offset index from 'a' for the character in the alien language
            charIndexMap[order[i] - 'a'] = i;
        }

        // Compare each word with the next one to see if they are in the correct order
        for (auto it = words.begin(); it != prev(words.end()); ++it) {
            // If the current word is greater than the next word, return false
            if (!isCorrectOrder(*it, *next(it), charIndexMap)) {
                return false;
            }
        }

        // If all the words are in the correct order, return true
        return true;
    }

private:
    // Helper function to determine if two words are in the correct order according to the alien dictionary.
    bool isCorrectOrder(const string &word1, const string &word2, const vector<int>& charIndexMap) {
        int len1 = word1.size(), len2 = word2.size();
        for (int i = 0; i < min(len1, len2); ++i) {
            // Compare character by character according to the alien dictionary order
            int char1Index = charIndexMap[word1[i] - 'a'];
            int char2Index = charIndexMap[word2[i] - 'a'];

            // If characters are different and correctly ordered, word1 is less than word2, stop comparing
            if (char1Index < char2Index) return true;

            // If characters are not in the correct order, return false
            if (char1Index > char2Index) return false;
        }

        // If words are equal for min(len1, len2), the shorter word should come first
        return len1 <= len2;
    }
};

```

Typescript Solution

```typescript
function isAlienSorted(words: string[], order: string): boolean {
    // Create a mapping of each character to its position in the new language order
    const charPositionMap = new Map<string, number>();
    for (const char of order) {
        charPositionMap.set(char, charPositionMap.size);
    }

    // Iterate through the pairs of adjacent words
    for (let i = 1; i < words.length; i++) {
        const firstWord = words[i - 1];
        const secondWord = words[i];

        // Find the minimum length to compare both words
        const minLength = Math.min(firstWord.length, secondWord.length);
        let areWordsEqual = false;

        // Compare characters of both words
        for (let j = 0; j < minLength; j++) {
            if (charPositionMap.get(firstWord[j]) > charPositionMap.get(secondWord[j])) {
                // The first word is greater than the second, so the list is not sorted
                return false;
            }
            if (charPositionMap.get(firstWord[j]) < charPositionMap.get(secondWord[j])) {
                // The first word comes before the second, so we can break the comparison
                areWordsEqual = true;
                break;
            }
        }

        // If all characters are equal, but the length of the first word is greater, the list is not sorted
        if (!areWordsEqual && firstWord.length > secondWord.length) {
            return false;
        }
    }

    // If all words are properly sorted or equal, return true
    return true;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code consists of a few different components:

- Creating a mapping (dictionary) `m` from characters to their alien order indices has a complexity of `O(1)` because the number of characters in the `order` string is constant (assuming a fixed alphabet size, e.g., 26 for the English alphabet).
- The outer loop runs a fixed 20 times which is a constant factor and thus does not impact the scalability of the algorithm. The value 20 probably assumes that no word will be longer than 20 characters, which seems to be an arbitrary limit. However, this loop should ideally be in terms of the length of the longest word in `words`. It would be more accurate to consider it as `O(L)` where `L` is the length of the longest word in `words`.
- Inside the outer loop, there is an inner loop that iterates over each word in the `words` list. If the number of words is `N`, this gives us a complexity of `O(N)` for the inner loop.
- Within the inner loop, the operations are constant time, involving index lookup and comparison.

Putting it all together, even with the arbitrary limit of 20 iterations, the actual scalable factor is the length of the words and the number of words. So the time complexity is `O(L * N)`.

Space Complexity

- The space complexity for creating the `m` dictionary is `O(1)`, again because the alphabet size is constant.
- The code does not use any additional significant memory that scales with the input size, as all other variables are of fixed size regardless of the input.

Thus, the overall space complexity of the code is `O(1)`.