244. Shortest Word Distance II

Medium Design Array Hash Table Two Pointers String

Leetcode Link

Problem Description

The problem provides us with the task of designing a data structure that must be able to compute the shortest distance between any two distinct strings within an array of strings. This implies that we first need to initialize the data structure with an array of strings, which we'll refer to as `wordsDict`. Once initialized, we should be able to query this data structure with two different strings, and it should return the smallest possible index difference between these two strings within the `wordsDict` array.

This could be visualized as an array where we want to find the minimum distance between two elements, where the elements are the positions of the words we're interested in. For example, if `wordsDict` is `["practice", "makes", "perfect", "coding", "makes"]`, and we query for the distance between `"coding"` and `"practice"`, the returned value should be `3`, as the closest "practice" to "coding" is three indices away.

Intuition

To efficiently find the shortest distance between two words in the dictionary, a preprocessing step is required during initialization. We traverse the `wordsDict` array once and create a hash map where keys are the distinct words from the array, and values are lists of indices where each key word is located in the original array. This preprocessing step allows for a quick lookup of the positions of any word, facilitating the computation of the distances between any two words.

Once the positions are mapped, to find the shortest distance between `word1` and `word2`, we get their list of indices from our hash map. We need to find the minimum difference between any two indices from these lists. The lists are already sorted because the indices were appended in the order they were encountered during initialization.

A two-pointer approach efficiently solves the problem of finding the minimum difference. Start with the first index of each list, and at each step, move the pointer that points to the smaller index to the next index in its list. The intuition behind this is that we try to close the gap between `word1` and `word2` by moving forward in the list where the current index is smaller. This approach will traverse the two lists simultaneously and will always give the smallest possible difference in indices (thus the shortest distance) between `word1` and `word2`. The process repeats until we have fully traversed one of the lists, ensuring that no potential shorter distance is missed.

The overall time complexity for the `shortest` operation, after the initial preprocessing, is O(N + M), where N and M are the number of occurrences of `word1` and `word2` in the dictionary, respectively. The preprocessing itself is O(K), where K is the total number of words in the dictionary, with the space complexity also being O(K) for storing the hash map.

Solution Approach

In the reference solution, the `WordDistance` class is defined, which implements the required functionality. The class uses a hash map (dictionary in Python) to store the indices of each word as they appear in `wordsDict`. Python's `defaultdict` from the `collections` module is used to handle the automatic creation of entries for new words, with each entry being a list that gets appended with the current index (`i`) every time the word (`w`) is encountered in the array.

Here is a breakdown of the implementation:

1. **Initialization (`__init__`)**: This method takes a list of strings `wordsDict` as input. It iterates over `wordsDict`, enumerating each word with its index. For every word-index pair, it appends the index to the list that corresponds to the word in the hash map `self.d`. By the end of this process, we have a dictionary where each word is associated with a list of indices indicating its positions in the `wordsDict` array.

```
self.d = defaultdict(list)
for i, w in enumerate(wordsDict):
    self.d[w].append(i)
```

2. **Finding Shortest Distance (`shortest`)**: This method calculates the shortest distance between two words `word1` and `word2`. By accessing `self.d[word1]` and `self.d[word2]`, it retrieves the two lists of indices for the given words. The algorithm then initializes two pointers, `i` and `j`, starting at `0`. These pointers will traverse the index lists.

   The variable `ans` is initialized to infinity (`inf`). It will be updated with the minimum difference between indices found so far as we iterate through the lists of indices.

   Following a two-pointer approach, the algorithm compares the indices at the current pointers and updates `ans` with the smaller of the current difference and the previously stored `ans`. Based on which index is smaller, the corresponding pointer (`i` or `j`) is incremented to move to the next index in the respective list.

   The loop continues until one list is fully traversed, ensuring that the minimum distance has been found:

```
a, b = self.d[word1], self.d[word2]
ans = inf
i = j = 0
while i < len(a) and j < len(b):
    ans = min(ans, abs(a[i] - b[j]))
    if a[i] <= b[j]:
        i += 1
    else:
        j += 1
return ans
```

In the above loop, at each step, because the two lists are sorted by the nature of their construction, we can guarantee that moving the pointer past the smaller index will give us a pair of indices that is potentially closer than the previous one. This algorithm effectively finds the minimum distance between the two words, which is the absolute difference between their closest indices.

The `WordDistance` class can then be instantiated with a string array `wordsDict`, and the `shortest` method can be called to find the shortest distance between any two words.

Example Walkthrough

Let's demonstrate how the `WordDistance` class and its methods work using a simple example.

Suppose our `wordsDict` is `["the", "quick", "brown", "fox", "quick"]`. We initialize the `WordDistance` class with this list of strings. During the initialization process, the class builds the hash map `self.d` which will contain:

- `"the"` → `[0]`
- `"quick"` → `[1, 4]`
- `"brown"` → `[2]`
- `"fox"` → `[3]`

Now, let's say we want to find the shortest distance between the words `"brown"` and `"quick"`. We call the `shortest` method with these two words. Here's what happens inside this method:

1. We access the two lists of indices for the words `"brown"` and `"quick"` from our hash map, which gives us `a = [2]` and `b = [1, 4]`.
2. We initialize two pointers `i` and `j` and a variable `ans` which will keep track of the minimum distance. Initially, `i = 0, j = 0, ans = inf`.
3. We enter a while-loop, which will continue until we have fully traversed one of the lists.
4. Inside the loop, we compare `a[i]` with `b[j]`. At the start, `a[i] = 2` and `b[j] = 1`. The difference `|2 - 1| = 1` is less than `ans`, so we update `ans = 1`.
5. Since the current value at `a[i]` is greater than `b[j]`, we increment `j` to move to the next index in list `b`.
6. Now, `i = 0` and `j = 1`, which means `a[i] = 2` and `b[j] = 4`. The difference `|2 - 4| = 2` is not less than the current `ans`, so `ans` remains `1`.
7. We've reached the end of list `a`, so the loop terminates.

Since we have examined all possible pairs of indices for "brown" and "quick", we have found the shortest distance to be `1`, which is the minimum index difference between these words in the `wordsDict`.

Finally, we return `ans` which is `1`, concluding that the shortest distance between `"brown"` and `"quick"` in the `wordsDict` array is `1`.

Python Solution

```
from collections import defaultdict
from math import inf

class WordDistance:
    def __init__(self, words_dict: List[str]):
        # Initializing a default dictionary to store the indices of each word
        self.indices_map = defaultdict(list)

        # Loop through the list of words and populate the indices map
        for index, word in enumerate(words_dict):
            self.indices_map[word].append(index)

    def shortest(self, word1: str, word2: str) -> int:
        # Retrieve the list of indices for the two words
        indices_word1, indices_word2 = self.indices_map[word1], self.indices_map[word2]
        # Initialize the minimum distance to infinity
        min_distance = inf
        # Initialize two pointers for the lists of indices
        i, j = 0, 0

        # Iterate until we reach the end of one of the lists of indices
        while i < len(indices_word1) and j < len(indices_word2):
            # Update the minimum distance as the smaller between the previous
            # minimum distance and the current distance between word1 and word2
            min_distance = min(min_distance, abs(indices_word1[i] - indices_word2[j]))
            # Move the pointer of the list which currently has a smaller index forward
            if indices_word1[i] <= indices_word2[j]:
                i += 1
            else:
                j += 1

        # Return the minimum distance found
        return min_distance

# Example of usage:
# word_distance = WordDistance(["practice", "makes", "perfect", "coding", "makes"])
# result = word_distance.shortest("coding", "practice")
# result would have the shortest distance between "coding" and "practice" in the list

```

Java Solution

```
// Class to calculate the shortest distance between two words in a dictionary.
class WordDistance {
    // Create a map to hold lists of indices for each word
    private Map<String, List<Integer>> wordIndicesMap = new HashMap<>();

    // Constructor takes an array of words and populates the map with the words and their indices.
    public WordDistance(String[] wordsDict) {
        for (int i = 0; i < wordsDict.length; ++i) {
            // For each word, add the current index to its list in the map.
            // If the word is not already in the map, create a new list for it.
            wordIndicesMap.computeIfAbsent(wordsDict[i], k -> new ArrayList<>()).add(i);
        }
    }

    // Method to find the shortest distance between two words.
    public int shortest(String word1, String word2) {
        // Retrieve the lists of indices for the two words.
        List<Integer> indicesWord1 = wordIndicesMap.get(word1);
        List<Integer> indicesWord2 = wordIndicesMap.get(word2);

        // Initialize the answer with a high value to ensure it gets replaced.
        int shortestDistance = Integer.MAX_VALUE;

        // Pointers to iterate through the lists for both words.
        int i = 0;
        int j = 0;

        // Iterate through both lists to find the smallest distance.
        while (i < indicesWord1.size() && j < indicesWord2.size()) {
            int indexWord1 = indicesWord1.get(i);
            int indexWord2 = indicesWord2.get(j);

            // Update the shortest distance with the minimum distance found so far.
            shortestDistance = Math.min(shortestDistance, Math.abs(indexWord1 - indexWord2));

            // Move the pointer that points to the smaller index forward.
            if (indexWord1 <= indexWord2) {
                ++i;
            } else {
                ++j;
            }
        }
        // Return the shortest distance found.
        return shortestDistance;
    }
}

/**
 * The following comments are provided as an example of how you might use the WordDistance class.
 * WordDistance object can be created by passing in a dictionary of words.
 * You can then call the 'shortest' method to find the shortest distance between any two words.
 */

// Example of using the WordDistance class.
/*
WordDistance wordDistance = new WordDistance(wordsDict);
int result = wordDistance.shortest("word1", "word2");
*/

```

C++ Solution

```
#include <vector>
#include <string>
#include <unordered_map>
#include <climits>
using namespace std;

class WordDistance {
public:
    // Constructor takes a reference to a vector of strings as the dictionary
    WordDistance(vector<string>& wordsDict) {
        // Store the indices of each word's occurrences in the dictionary
        for (int i = 0; i < wordsDict.size(); ++i) {
            wordIndices[wordsDict[i]].push_back(i);
        }
    }

    // Function to find the shortest distance between two words
    int shortest(string word1, string word2) {
        // Retrieve the list of indices for each word
        auto indicesWord1 = wordIndices[word1];
        auto indicesWord2 = wordIndices[word2];

        int indexWord1 = 0; // Current index in indicesWord1
        int indexWord2 = 0; // Current index in indicesWord2
        int minimumDistance = INT_MAX; // Initialize minimum distance as maximum possible int value

        // Loop until we reach the end of one of the index lists
        while (indexWord1 < indicesWord1.size() && indexWord2 < indicesWord2.size()) {
            // Update minimumDistance with the smallest difference between indices
            minimumDistance = min(minimumDistance, abs(indicesWord1[indexWord1] - indicesWord2[indexWord2]));

            // Move to the next index in the list, which has the smaller current index value
            if (indicesWord1[indexWord1] < indicesWord2[indexWord2]) {
                indexWord1++;
            } else {
                indexWord2++;
            }
        }
        return minimumDistance; // Return the smallest distance found
    }

private:
    // Dictionary to store words and their occurrences' indices
    unordered_map<string, vector<int>> wordIndices;
};

// The class can be used as follows:
// WordDistance* obj = new WordDistance(wordsDict);
// int shortestDistance = obj->shortest(word1, word2);

```

Typescript Solution

```
// Dictionary to store words and their occurrences' indices
const wordIndices: { [word: string]: number[] } = {};

// Function to initialize the word indices dictionary with a list of words
function initializeWordDistance(wordsDict: string[]): void {
    for (let i = 0; i < wordsDict.length; ++i) {
        const word = wordsDict[i];
        if (wordIndices[word] === undefined) {
            wordIndices[word] = [];
        }
        wordIndices[word].push_back(i);
    }
}

// Function to find the shortest distance between two words
function shortest(word1: string, word2: string): number {
    // Retrieve the list of indices for each word
    const indicesWord1 = wordIndices[word1];
    const indicesWord2 = wordIndices[word2];

    let indexWord1 = 0; // Current index in indicesWord1
    let indexWord2 = 0; // Current index in indicesWord2
    let minimumDistance = Number.MAX_SAFE_INTEGER; // Initialize minimum distance as largest possible safe integer value

    // Loop until we reach the end of one of the index lists
    while (indexWord1 < indicesWord1.length && indexWord2 < indicesWord2.length) {
        // Update minimumDistance with the smallest difference between indices
        minimumDistance = Math.min(minimumDistance, Math.abs(indicesWord1[indexWord1] - indicesWord2[indexWord2]));

        // Move to the next index in the list, which has the smaller current index value
        if (indicesWord1[indexWord1] < indicesWord2[indexWord2]) {
            indexWord1++;
        } else {
            indexWord2++;
        }
    }

    return minimumDistance; // Return the smallest distance found
}

// Usage example:
// initializeWordDistance(wordsDict);
// let shortestDistance = shortest(word1, word2);

```

Time and Space Complexity

Initialization (`__init__`)

- Time Complexity: O(N), where `N` is the length of `wordsDict`. This is because we're iterating through each word in the list and inserting the index into the dictionary.
- Space Complexity: O(N), as in the worst-case scenario, we're storing the index of each word within the dictionary, which would require space proportional to the number of words.

Shortest Method (`shortest`)

- Time Complexity: O(L + M), where `L` is the length of the list associated with `word1` and `M` is the length of the list associated with `word2`. In the worst case, we traverse each list once in a single run of the while loop.
- Space Complexity: O(1), because we're only using a few variables (`i`, `j`, `ans`, `a`, `b`) for calculations and not utilizing any additional space that is dependent on the size of the input.