

243. Shortest Word Distance

EasyArrayString

Problem Description

In this problem, you are given an array of strings called `wordsDict`, which contains a list of words. You are also given two different strings `word1` and `word2` which you can be certain are present in `wordsDict`. Your task is to find the shortest distance between `word1` and `word2` within `wordsDict`. The distance between two words is the number of words between them in the list (or the absolute difference between their indices in `wordsDict`). You need to consider the case where there may be multiple occurrences of `word1` and/or `word2`, and you should find the minimum distance among all possible pairs of `word1` and `word2`.

Intuition

To find the shortest distance between two words in an array, a straightforward approach is to scan through the array while keeping track of the positions of the two words. Here's the thinking process leading to the solution:

- Initialize two index pointers, `i` and `j`, to `-1`, to represent the most recent positions of `word1` and `word2`, respectively.
- Initialize `ans` (answer) to infinity to keep track of the current minimum distance. `inf` in the code stands for infinity, representing an initially large distance.
- Iterate through `wordsDict` using a loop, keeping track of the index `k` and the current word `w`.
- If `w` matches `word1`, update the position `i` to the current index `k`.
- If `w` matches `word2`, update the position `j` to the current index `k`.
- After each word match, if both `i` and `j` have been updated from their initial value of `-1` (meaning both `word1` and `word2` have been found at least once), calculate the current distance between `word1` and `word2` using `abs(i - j)`.
- Update `ans` with the smallest of its current value and the new distance just calculated.
- After completing the loop, return `ans`.

Solution Approach

The solution uses a one-pass algorithm to find the shortest distance between two words in an array. Here's how it's implemented:

- Initialize indices and answer variable:**
 - Two index variables `i` and `j` are initialized to `-1`, which will keep track of the most recent positions of `word1` and `word2`, respectively.
 - The variable `ans` is initialized to `inf` (infinity), which will be used to keep the smallest distance encountered.
- Iterate over array:** The code uses a loop to iterate through each element of `wordsDict` with enumeration, which provides both index `k` and value `w` for every iteration.
- Find and update positions:** During iteration, if the current word `w` equals `word1`, index `i` is updated to the current index `k`. Similarly, if `w` equals `word2`, index `j` is updated to the current index `k`.
- Calculate distance when both words are found:** After any update to `i` or `j`, the solution checks if both `i` and `j` are not `-1` anymore, indicating that both `word1` and `word2` have been encountered. At this point, it calculates the distance using the absolute difference `abs(i - j)`.
- Update the shortest distance:** The calculated distance is then compared with `ans`. If it is smaller, `ans` is updated with the new distance. This ensures that at the end of the loop, `ans` holds the minimum distance between the two words.
- Return the answer:** After the loop ends, `ans` will contain the shortest distance between `word1` and `word2`, which the function then returns.

This algorithm exhibits a linear time complexity, i.e., $O(n)$, where n is the number of elements in `wordsDict`, as it only requires a single pass through the list. No extra space is used, apart from a few variables for indices and the minimum distance, so the space complexity is $O(1)$. The simplicity and efficiency of this method make it a good choice for this problem.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Imagine our `wordsDict` is `["practice", "makes", "perfect", "coding", "makes"]`, and we are tasked to find the shortest distance between `word1 = "coding"` and `word2 = "practice"`.

- Initialize indices and answer variable:** `i = -1, j = -1, ans = inf`. `i` and `j` will hold the positions of "coding" and "practice" respectively once they are found, and `ans` will keep track of the shortest distance.
- Iterate over array:**
 - At index `k = 0, w = "practice"`. It matches `word2`, so we update `j = 0`.
 - At index `k = 1, w = "makes"`. This doesn't match either `word1` or `word2`.
 - At index `k = 2, w = "perfect"`. This also doesn't match either `word1` or `word2`.
 - At index `k = 3, w = "coding"`. It matches `word1`, so we update `i = 3`.
- Calculate distance when both words are found:**
 - Now we have found both `word1` and `word2` (`i` and `j` are not `-1`). So we compute the distance: `abs(i - j) = abs(3 - 0) = 3`.
- Update the shortest distance:**
 - We compare `3` with `ans` which is `inf`. Since `3` is less, we update `ans = 3`.
- There are no more elements to process**, so the loop ends.
- Return the answer:**
 - At this point `ans = 3`, which is the shortest distance between "coding" and "practice" in the given `wordsDict`.

In conclusion, after walking through the example, the shortest distance between the two words "coding" and "practice" is `3`, as they are three words apart from each other in the list. This demonstrates how the one-pass solution efficiently computes the shortest distance with simple updates to index variables while iterating through the list only once.

Solution Implementation

Python

```
from typing import List

class Solution:
    def shortestDistance(self, wordsDict: List[str], word1: str, word2: str) -> int:
        # Initialize indices for the positions of word1 and word2
        index1 = index2 = -1
        # Initialize the answer as infinite to ensure any actual distance found is smaller
        shortest_distance = float('inf')

        # Loop through the words in the dictionary to find the closest distance
        for index, word in enumerate(wordsDict):
            if word == word1:
                index1 = index # Update the position of word1
            if word == word2:
                index2 = index # Update the position of word2

            # If both words have been found at least once, calculate the distance
            if index1 != -1 and index2 != -1:
                distance = abs(index1 - index2) # Compute absolute difference
                shortest_distance = min(shortest_distance, distance) # Update the shortest distance

        # Return the shortest distance found between the two words
        return shortest_distance
```

Java

```
class Solution {
    // Method to find the shortest distance between two words in a dictionary
    public int shortestDistance(String[] wordsDict, String word1, String word2) {
        // Initialize the minimum distance to a very high value
        int minDistance = Integer.MAX_VALUE;

        // These will hold the last seen positions of word1 and word2
        int lastPosWord1 = -1;
        int lastPosWord2 = -1;

        // Loop through the words dictionary to find the words
        for (int index = 0; index < wordsDict.length; ++index) {
            // If the current word equals word1, update lastPosWord1
            if (wordsDict[index].equals(word1)) {
                lastPosWord1 = index;
            }
            // If the current word equals word2, update lastPosWord2
            if (wordsDict[index].equals(word2)) {
                lastPosWord2 = index;
            }
            // If both last positions are set and not -1, calculate the distance
            if (lastPosWord1 != -1 && lastPosWord2 != -1) {
                // Update the minimum distance if a new minimum is found
                minDistance = Math.min(minDistance, Math.abs(lastPosWord1 - lastPosWord2));
            }
        }
        // Return the minimum distance found
        return minDistance;
    }
}
```

C++

```
#include <vector>
#include <string>
#include <climits> // Include for INT_MAX

class Solution {
public:
    // Function to find the shortest distance between two words in a list
    int shortestDistance(std::vector<std::string>& wordsDict, std::string word1, std::string word2) {
        int shortestDistance = INT_MAX; // Initialize shortest distance with the maximum possible value
        // Use indices word1Index and word2Index to keep track of the most recent positions of word1 and word2
        // Initialize both indices to -1, indicating that these words have not been encountered yet
        int word1Index = -1;
        int word2Index = -1;

        // Loop through all words in the dictionary
        for (int k = 0; k < wordsDict.size(); ++k) {
            if (wordsDict[k] == word1) { // If the current word is word1
                word1Index = k; // Update index of word1
            }
            if (wordsDict[k] == word2) { // If the current word is word2
                word2Index = k; // Update index of word2
            }

            // If both words have been seen at least once
            if (word1Index != -1 && word2Index != -1) {
                // Calculate the distance and update shortestDistance if it's smaller
                shortestDistance = std::min(shortestDistance, std::abs(word1Index - word2Index));
            }
        }

        // Return the shortest distance found
        return shortestDistance;
    }
};
```

TypeScript

```
// Importing necessary functionalities
import { min, abs, MAX_VALUE } from 'math';

// Function to find the shortest distance between two words in a list
function shortestDistance(wordsDict: string[], word1: string, word2: string): number {
    // Initialize shortest distance with the maximum possible value
    let shortestDistance: number = MAX_VALUE;
    // Use indices word1Index and word2Index to keep track of the most recent positions of word1 and word2
    // Initialize both indices to -1, indicating that these words have not been encountered yet
    let word1Index: number = -1;
    let word2Index: number = -1;

    // Loop through all words in the dictionary
    for (let k = 0; k < wordsDict.length; ++k) {
        if (wordsDict[k] === word1) { // If the current word is word1
            word1Index = k; // Update index of word1
        }
        if (wordsDict[k] === word2) { // If the current word is word2
            word2Index = k; // Update index of word2
        }

        // If both words have been seen at least once
        if (word1Index !== -1 && word2Index !== -1) {
            // Calculate the distance and update shortestDistance if it's smaller
            shortestDistance = min(shortestDistance, abs(word1Index - word2Index));
        }
    }

    // Return the shortest distance found
    return shortestDistance;
}
```

```
from typing import List

class Solution:
    def shortestDistance(self, wordsDict: List[str], word1: str, word2: str) -> int:
        # Initialize indices for the positions of word1 and word2
        index1 = index2 = -1
        # Initialize the answer as infinite to ensure any actual distance found is smaller
        shortest_distance = float('inf')

        # Loop through the words in the dictionary to find the closest distance
        for index, word in enumerate(wordsDict):
            if word == word1:
                index1 = index # Update the position of word1
            if word == word2:
                index2 = index # Update the position of word2

            # If both words have been found at least once, calculate the distance
            if index1 != -1 and index2 != -1:
                distance = abs(index1 - index2) # Compute absolute difference
                shortest_distance = min(shortest_distance, distance) # Update the shortest distance

        # Return the shortest distance found between the two words
        return shortest_distance
```

Time and Space Complexity

The time complexity of the code provided is $O(n)$, where n is the length of the `wordsDict` list. This is because the code processes each word in the list exactly once in a single loop.

The space complexity of the code is $O(1)$. It uses a fixed number of variables `i`, `j`, and `ans` that do not depend on the size of the input list. Therefore, the amount of additional memory used does not increase with the size of `wordsDict`.