2515. Shortest Distance to Target String in a Circular Array

Easy Array String

Problem Description

In this problem, we are given a list of strings called `words` that is arranged in a circular fashion. What this means is that if you were to look at the array of words, the word that comes after the last one is the first word of the list, creating a loop-like structure.

We are also given a `target` string that we need to find in the `words` list, starting the search from a given `startIndex`. The goal is to figure out the shortest distance from this starting index to the index where the target string is located.

The search can proceed in either direction - moving to the next word in the list or to the previous one. When searching, each step taken counts as `1`, regardless of the direction.

To summarize:

- If the target string is present, we need to find and return the minimum number of steps required to reach it from the startIndex.
- If the target string is not in the list, we should return `-1`.

Keep in mind that the next element of `words[i]` is `words[(i + 1) % n]` and the previous element is `words[(i - 1 + n) % n]`, where `n` is the total number of elements in `words`, effectively making it a circular array.

Intuition

To solve this problem, the intuition is to iterate through the entire circular list to locate the target string. As we encounter the given target, we calculate how far it is from our `startIndex`.

We consider two scenarios to find the shortest path:

1. Moving clockwise (forward) to reach the target.
2. Moving counter-clockwise (backward) to reach the target.

Let's consider an example where we have a list of 6 words and our `startIndex` is at position 1. If our target is at position 4, we can reach it in 3 steps by moving forward or in 3 steps by moving backward (since the array is circular). We always want the shortest number of steps.

The python code defines a function called `closetTarget` that calculates the minimum number of steps required to reach the target word from the `startIndex`. The distance is calculated in both directions for each occurrence of the target and the minimum is stored. After checking all the words, the function returns the smallest distance found. If the target word is absent, the function will return `-1`.

Solution Approach

The implementation leverages a simple linear search along with modular arithmetic to handle the circular nature of the array.

The algorithm proceeds as follows:

1. We initialize a variable `ans` to `n`, which is the length of the `words` list. This initial value acts as a placeholder for the scenario where the target is not found.
2. We then loop over the `words` list using `enumerate` to get both the index `i` and the word `w` at each iteration.
3. For every word `w` that matches the `target`, we calculate the distance `t` from `startIndex` in two ways:
   - Direct distance: `t = abs(i - startIndex)`
   - Circular distance: `n - t`, which represents the distance if we were to go around the array in the opposite direction.
4. We update `ans` with the minimum of its current value, the direct distance `t`, and the circular distance `n - t`. This ensures that after scanning all words, `ans` holds the shortest distance required to reach `target` from `startIndex`.
5. If we finish the loop and `ans` remains equal to `n`, it means that the target word was not found in the `words` list. In this case, we return `-1`.
6. Otherwise, we return the value stored in `ans`, which is the shortest distance found.

An important detail to note is the use of modular arithmetic to deal with the circular indexing, but since we are using Python's `abs` function to calculate the direct distance, we do not explicitly use the modulus operator `%` in our distance calculations. Instead, the modulus operator would be necessary if we were manually wrapping around the indices.

This algorithm will work efficiently as it only requires a single pass over the list, giving it a time complexity of O(n), where n is the number of words in the list. We don't use any additional data structures, giving us a space complexity of O(1).

Example Walkthrough

Consider the following example to illustrate the solution approach:

- Let's assume `words = ["apple", "banana", "cherry", "date", "elderberry", "fig"]` is our list of words arranged in a circular fashion.
- The `target` string that we are looking for is `"date"`.
- Our `startIndex` is `2`, which means we start our search from the word `"cherry"`.

Following the steps outlined in the solution approach:

1. We set `ans = 6` as there are 6 words in our list. This is our initial best-case scenario value, which we will update if we find the word.
2. We enumerate through our list to get each word and its index:
   - Index `0`, Word `"apple"`: Not our target.
   - Index `1`, Word `"banana"`: Not our target.
   - Index `2`, Word `"cherry"`: This is our starting index.
   - Index `3`, Word `"date"`: This is our target!
3. Since we have found our target at index `3`, we calculate the direct distance from our `startIndex` (`2`) to our target index (`3`):
   - Direct distance: `t = abs(3 - 2) = 1`
   We also calculate the circular distance by subtracting this direct distance from the total number of words (`n`):
   - Circular distance: `n - t = 6 - 1 = 5`
4. We update `ans` with the minimum of its current value (`6`), the direct distance (`1`), and the circular distance (`5`). This update gives us `ans = min(6, 1, 5) = 1`.
5. Since we successfully found the word, we don't need to return `-1`, and we can skip to step 6.
6. We return the value stored in `ans`, which is `1`, indicating that the target word `"date"` is `1` step away from the starting index `2` (the word `"cherry"`) in the list.

Thus, the function `closetTarget` will return `1` for this example, because the shortest path to the target `"date"` from `startIndex 2` is by moving forward one step in the circular array.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def closestTarget(self, words: List[str], target: str, start_index: int) -> int:
        # The length of the list of words.
        num_words = len(words)
        # Initialize the minimum distance with the number of words,
        # which is an upper bound on the distance we can find.
        min_distance = num_words

        # Iterate through each word in the list along with their index.
        for index, word in enumerate(words):
            # Check if the current word is the target we're searching for.
            if word == target:
                # Calculate the absolute distance from the current word to the start index.
                distance = abs(index - start_index)
                # The minimum distance is updated with the smallest of the current minimum,
                # the new distance, or the circular distance (num_words - distance).
                min_distance = min(min_distance, distance)

        # If min distance is still num_words, it means the target was not found.
        # Hence, return -1. Otherwise, return the found minimum distance.
        return -1 if min_distance == num_words else min_distance
```

Java

```java
class Solution {
    public int closestTarget(String[] words, String target, int startIndex) {
        // Get the length of the words array.
        int n = words.length;
        // Initialize the answer with the maximum possible distance which is n.
        int closestDistance = n;

        // Iterate through the words array to find the closest target.
        for (int i = 0; i < n; ++i) {
            // Current word at index i.
            String currentWord = words[i];
            // Check if the current word matches the target word.
            if (currentWord.equals(target)) {
                // Calculate the direct distance from the start index to the current index.
                int directDistance = Math.abs(i - startIndex);
                // Calculate the distance assuming we can wrap around the array.
                int wrappedDistance = n - directDistance;
                // Choose the smaller of the two distances to find the closest position.
                closestDistance = Math.min(closestDistance, Math.min(directDistance, wrappedDistance));
            }
        }
        // If the closestDistance is still n, that means the taroet was not found.
        // In that case, return -1. Otherwise, return the closest distance found.
        return closestDistance == n ? -1 : closestDistance;
    }
}
```

C++

```cpp
#include <vector>
#include <string>
#include <cmath> // For abs() function
#include <algorithm> // For min() function

class Solution {
public:
    // Function to find the closest index of a target string from the given startIndex
    // within an array of words.
    // Parameters:
    // - words: A vector of strings representing the array of words.
    // - target: The string target we are trying to find the closest index to.
    // - startIndex: The index from which we start searching for the closest occurrence.
    // Returns: The minimum distance to the closest occurrence of the target word. If not found, returns -1.
    int closestTarget(vector<string>& words, string target, int startIndex) {
        int n = words.size(); // The number of words in the vector
        int minDistance = n; // Initialize the minimum distance with the maximum possible value, the size of the words vector

        // Iterate over the vector to find the occurrences of the target string
        for (int i = 0; i < n; ++i) {
            // If the current word matches the target word
            if (words[i] == target) {
                // Calculate the distance from the startIndex to the current index
                int currentDistance = abs(i - startIndex);
                // Find the minimum distance considering the circular array (wrapping around)
                minDistance = min(minDistance, min(currentDistance, n - currentDistance));
            }
        }

        // If minDistance is unchanged, the target was not found; return -1.
        // Otherwise, return the minimum distance to the nearest target word.
        return minDistance == n ? -1 : minDistance;
    }
};
```

TypeScript

```typescript
// Finds index of the closet 'taroet' string in 'words' array from a soecified 'startIndex'.
function closetTaroet(words: string[], taroet: string, startIndex: number): number {
    const wordsCount = words.length; // Total number of words in the array

    // Loop through the words array until the mid-point
    for (let offset = 0; offset <= wordsCount >> 1; offset++) {
        // Calculate the index to the left of the start index, wrapping if necessary
        const leftIndex = (startIndex - offset + wordsCount) % wordsCount;
        if (words[leftIndex] === taroet) {
            return offset; // Return the offset if target is found
        }

        // Calculate the index to the right of the start index, wrapping if necessary
        const rightIndex = (startIndex + offset) % wordsCount;
        if (words[rightIndex] === taroet) {
            return offset; // Return the offset if target is found
        }
    }
    // Return -1 if the target is not found
    return -1;
}
```

```python
from typing import List

class Solution:
    def closestTaroet(self, words: List[str], target: str, start_index: int) -> int:
        # The length of the list of words.
        num_words = len(words)
        # Initialize the minimum distance with the number of words,
        # which is an upper bound on the distance we can find.
        min_distance = num_words

        # Iterate through each word in the list along with their index.
        for index, word in enumerate(words):
            # Check if the current word is the target we're searching for.
            if word == taroet:
                # Calculate the absolute distance from the current word to the start index.
                distance = abs(index - start_index)
                # The minimum distance is updated with the smallest of the current minimum,
                # the new distance, or the circular distance (num_words - distance).
                min_distance = min(min_distance, distance)

        # If min distance is still num words, it means the taroet was not found.
        # Hence, return -1. Otherwise, return the found minimum distance.
        return -1 if min_distance == num_words else min_distance
```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is `O(n)`, where `n` is the number of words in the input list `words`. This is because there is a single loop that iterates over each word in the list exactly once to check if it matches the `target`.

Space Complexity

The space complexity of the provided code is `O(1)` regardless of the size of the input. This is because the code only uses a few extra variables (`n`, `ans`, `i`, `w`, `t`) that do not depend on the size of the input list `words`.