2108. Find First Palindromic String in the Array

Easy Array Two Pointers String Leetcode Link

Problem Description

The problem asks us to find the first string in an array that is palindromic. A palindromic string is a string that reads the same forwards as it does backwards. For example, 'racecar' is a palindrome because reversing it gives the same word. We are given an array of strings and we need to check these strings one by one to see if any of them is a palindrome. The first palindromic string we come across should be returned. If we go through the whole array and find no palindromic strings, we return an empty string, represented by `""`.

Intuition

To solve this problem, the natural approach is to iterate over each string in the array and check whether it is a palindrome. If we find a palindrome, we return it immediately. To check if a string is a palindrome, we need to read it from the front and back and compare each character to see if they match. A convenient way to reverse a string in Python is to use the slicing syntax `string[::-1]`, which creates a new string that is the reverse of the original.

Given that we want to find the *first* palindromic string, we can use Python's `next` function combined with a generator expression to efficiently look for it. The generator expression creates an iterator that will only process each string as needed, which is both memory efficient and potentially time-saving if the palindromic string is found early in the array. The `next` function then tries to get the first element from this generator, and if none is found, it returns a default value, which in this case is the empty string `""`. This succinctly solves the problem without needing extra loops or conditionals to handle the case where no palindromic string is found.

Solution Approach

The solution uses a generator expression that iterates over the list of `words`. This expression takes each word `w` in `words` and checks if it is a palindrome by comparing the word to its reverse, `w[::-1]`. This reversal is done using Python's slice notation, which, when provided with `[::-1]`, starts from the end of the string and steps backwards, effectively reversing it.

Here's a step by step breakdown of the algorithm:

1. The generator expression `(w for w in words if w == w[::-1])` creates an iterator. This iterator goes through each string `w` in the `words` array and yields it if and only if the string is the same as its reversed version. This leverages the fact that a palindrome is by definition a string that does not change when its characters are reversed.

2. Python's `next` function is used to retrieve the next item from the iterator created by the generator expression. In this context, it serves to fetch the first palindrome it encounters while iterating over the generator.

3. The second argument to the `next` function is `""`, which is the default value to return in case the generator does not yield any items, i.e., if there are no palindromic strings in the array.

This solution is elegant and efficient:

- It avoids the need for an explicit loop to iterate over the elements and check them one by one.
- It uses lazy evaluation: the generator expression is only evaluated up to the point where the first palindrome is found, which can save time if a palindrome occurs early in the list.
- There's no need for additional memory to store intermediate results, as the generator doesn't create a temporary list of palindromes, unlike, for example, a list comprehension would.

Overall, the use of `next` and a generator expression allows for an efficient and concise solution to the problem.

Example Walkthrough

Let's go through a small example to illustrate the solution approach step by step. We have the following array of strings:

```python
words = ["hello", "world", "racecar", "level", "sample"]
```

We are looking to find the first string that is palindromic.

1. The first string we check is `"hello"`. We reverse it to see if it matches its original form: "olleh". Since `"hello"` is not equal to `"olleh"`, we continue to the next word.

2. The next string is `"world"`. Reversed, it is `"dlrow"`, which again does not equal `"world"`. So we move on.

3. We come to `"racecar"` and reverse it. The reversed version is `"racecar"` — it matches exactly! Since we found a palindrome, we don't need to check any further; we return `"racecar"`.

To represent this process programmatically:

```python
# use generator expression to iterate over each word and check if it is a palindrome
palindromic_string = next((w for w in words if w == w[::-1]), "")
print(palindromic_string)  # This will output "racecar"
```

The generator expression `(w for w in words if w == w[::-1])` iterates through each word in the `words` array and yields the word if it is a palindrome. The `next` function is then used to get the first item from the iterator. If no palindromic string is found, it will return the default specified by the second argument, which is `""`.

In this example, the `next` function does not need to go through the entire list since it returns as soon as it finds the palindromic string `"racecar"`. Thus, the generator stops iterating after the third element, demonstrating efficient use of resources. If `racecar` had not been a palindrome, the iteration would continue to `"level"`, which is the next palindromic string, or until the list was exhausted.

Python Solution

```python
from typing import List  # Import List from the typing module for type hinting

class Solution:
    def firstPalindrome(self, words: List[str]) -> str:
        """
        Finds the first palindromic word from a list of words.

        Args:
        words (List[str]): A list of strings.

        Returns:
        str: The first palindromic word in the list of words, or an
             empty string if there is no such word.
        """
        # Iterate through each word in the list
        for word in words:
            # Check if the current word is the same as its reversed copy
            if word == word[::-1]:
                return word  # Return the palindromic word

        # Return an empty string if no palindromic word is found
        return ""

# Example usage:
# sol = Solution()
# print(sol.firstPalindrome(["abc", "car", "ada", "racecar", "cool"]))
# This will print "ada" as it is the first palindrome in the list

```

Java Solution

```java
class Solution {
    // This method finds the first palindromic string in an array.
    public String firstPalindrome(String[] words) {
        // Iterate over each word in the array.
        for (String word : words) {
            // Assume initially that each word could be a palindrome.
            boolean isPalindrome = true;
            // Use two pointers to compare characters from the front and the back.
            for (int i = 0, j = word.length() - 1; i < j; ++i, --j) {
                // If corresponding characters do not match, the word is not a palindrome.
                if (word.charAt(i) != word.charAt(j)) {
                    isPalindrome = false;
                    // No need to check remaining characters if a mismatch is found.
                    break;
                }
            }
            // If the word is a palindrome, return it.
            if (isPalindrome) {
                return word;
            }
        }
        // Return an empty string if no palindromic strings are found.
        return "";
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    // Function to find the first palindrome in a list of words.
    string firstPalindrome(vector<string>& words) {
        // Iterate over each word in the vector.
        for (const auto& word : words) {
            // Assume initially that the word is a palindrome.
            bool isPalindrome = true;

            // Check each character from the start and the end of the word moving towards the center.
            for (int start = 0, end = word.size() - 1; start < end; ++start, --end) {
                // If the characters do not match, the word is not a palindrome.
                if (word[start] != word[end]) {
                    isPalindrome = false;
                    break; // Exit the loop early as we already know it's not a palindrome.
                }
            }

            // If the word is a palindrome, return it.
            if (isPalindrome) {
                return word;
            }
        }

        // If no palindrome is found, return an empty string.
        return "";
    }
};

```

Typescript Solution

```typescript
// This function searches for the first palindrome string within an array of words.
// A palindrome is a word that reads the same backward as forward.
// @param words - An array of strings to be checked.
// @returns The first palindrome string if one exists, or an empty string if none is found.
function firstPalindrome(words: string[]): string {
    // Loop through each word in the array to check for palindromes.
    for (const word of words) {
        // Initialize pointers for the start and end of the word.
        let startIndex = 0;
        let endIndex = word.length - 1;

        // Move the pointers towards the center of the word.
        while (startIndex < endIndex) {
            // Check if the characters at the start and end pointers do not match.
            if (word[startIndex] !== word[endIndex]) {
                // If they do not match, break from the loop; the word is not a palindrome.
                break;
            }
            // Move the pointers closer to the center.
            startIndex++;
            endIndex--;
        }

        // If the pointers have met or crossed, the word is a palindrome.
        if (startIndex >= endIndex) {
            // Return the current palindrome word.
            return word;
        }
    }

    // If no palindrome is found in the array, return an empty string.
    return '';
}

```

Time and Space Complexity

Time Complexity

The provided code checks each word in the list to see if it is a palindrome. The time it takes to check if a word is a palindrome depends on the length of the word, as it requires comparing characters from the beginning and the end towards the center.

To determine the time complexity, let `n` be the number of words in the list and `k` be the average length of these words.

Checking if a single word is a palindrome has a complexity of `O(k)` because we need to compare about half of the characters (in the worst case) to their counterparts on the other side of the word.

As we use a generator expression with the `next` function to find the first palindrome, the worst-case scenario happens when the palindrome is the last word in the list or there is no palindrome at all. Therefore, we might need to check every word. In this case, the time complexity would be `O(n * k)`, as we perform an `O(k)` operation `n` times.

Space Complexity

The space complexity of the algorithm is `O(1)`.

This is because it uses a constant amount of additional space regardless of the input size. The generator expression does not create an additional list in memory; it simply iterates over the existing list, and string slicing creates a new string, but since we are only interested in one word at a time, this does not accumulate with the number of words. Thus, the space used does not scale with the number of words `n` or their lengths `k`.