422. Valid Word Square

Easy Array Matrix

Problem Description

A *word square* is a sequence of strings arranged in a square grid where the string at each row is the same as the string at each column if both are read from left to right or top to bottom. The `kth` row and column are considered the same if their characters match for all the indexes from 0 to `k`. The task is to check whether the given array of strings, `words`, can form such a valid word square.

Intuition

The intuitive approach to solving this problem is to simulate the process of reading the strings along the rows and columns and ensuring they match. To accomplish this, iterate through each word and character in the array of strings, simultaneously checking the corresponding character in the column. If the index we are checking exceeds either the number of words (for column validation) or the length of the corresponding word (for row length), or if the characters do not match, the array does not form a valid word square, and we return `false`. If no mismatches are found throughout the iteration, we can conclude that the words form a valid word square, and therefore return `true`.

Solution Approach

The solution applies a straightforward algorithm, iterating through each word and its characters in the given `words` list using a nested loop. The outer loop keeps track of rows and the inner loop keeps track of columns in the would-be word square grid.

For each character `c` at position `(i, j)` in the row `i` (where `i` is the index of the word in the list and `j` is the character index in the word), we want to confirm if there is a corresponding character at the position `(j, i)` of the column `j` that matches `c`.

Several key checks are performed for early detection of an invalid word square:

- **Index Check for Columns:** `if j >= m`, ensures that there isn't an attempt to access a column outside of the words list bounds. For a valid word square, the number of rows and columns should be the same (`m`).
- **Index Check for Row Lengths:** `if i >= len(words[j])`, ensures the character at `(i, j)` does not exceed the length of the word that is supposed to appear in column `j`, which would render the square invalid.
- **Character Matching Check:** `if c != words[j][i]`, verifies that the character at `(i, j)` matches the corresponding character at `(j, i)`, following the rule for word square validation.

If any of the above checks fail, the function immediately returns `False`, indicating that the `words` do not form a valid word square.

Otherwise, if all characters match properly based on their corresponding rows and columns, and no index out-of-range conditions are encountered, the whole nested loop will complete successfully. After the loops conclude without encountering an inconsistency, the method returns `True`, indicating that the `words` represent a valid word square.

The code implementation is efficient, as it only requires a simple check of each character's position in the grid relative to the other words, and thus it operates in `O(n^2)` time complexity on average, where `n` is the length of the longest word. This is because each character in each word is being compared once. The space complexity is `O(1)` since no additional data structures are used beyond the input list itself.

Example Walkthrough

Let's understand the solution approach using a small example. Consider the array of strings, `words = ["AREA","BALL","DEAR","LADY"]`. We want to find out if these words can form a valid word square. Below is a step-by-step walk-through of the solution algorithm:

1. Start with the first word "AREA". Initialize `i = 0` for the first row.
   - Examine "AREA" for each character `j` from 0 to 3 (since "AREA" has 4 letters).
   - For `j = 0`, check if `words[0][0]` ('A') equals `words[0][0]` ('A'). They match. Continue.
   - For `j = 1`, check if `words[0][1]` ('R') equals `words[1][0]` ('B'). They do not match. This means it cannot form a valid word square. Return `False`.

Using the above example, the algorithm will find a mismatch at the second character of the first word. The character 'R' in row 0 doesn't match the character 'B' in column 0 of the word "BALL". Hence, as per the character matching check, we conclude that the given `words` cannot form a valid word square, and the algorithm will return `False`.

If instead, the array were `words = ["BALL","AREA","LEAD","LADY"]`, the checking would proceed as follows:

1. Start with the first word "BALL". Proceed with character checks:
   - `words[0][0]` ('B') equals `words[0][0]` ('B').
   - `words[0][1]` ('A') equals `words[1][0]` ('A').
   - `words[0][2]` ('L') equals `words[2][0]` ('L').
   - `words[0][3]` ('L') equals `words[3][0]` ('L').
2. Move to the second word "AREA" (`i = 1`).
   - `words[1][0]` ('A') equals `words[0][1]` ('A').
   - `words[1][1]` ('R') equals `words[1][1]` ('R').
   - `words[1][2]` ('E') equals `words[2][1]` ('E').
   - `words[1][3]` ('A') equals `words[3][1]` ('A').
3. Continue with "LEAD" (`i = 2`).
   - `words[2][0]` ('L') equals `words[0][2]` ('L').
   - `words[2][1]` ('E') equals `words[1][2]` ('E').
   - `words[2][2]` ('A') equals `words[2][2]` ('A').
   - `words[2][3]` ('D') equals `words[3][2]` ('D').
4. Finally, check "LADY" (`i = 3`).
   - `words[3][0]` ('L') equals `words[0][3]` ('L').
   - `words[3][1]` ('A') equals `words[1][3]` ('A').
   - `words[3][2]` ('D') equals `words[2][3]` ('D').
   - `words[3][3]` ('Y') equals `words[3][3]` ('Y').

No inconsistencies are found, and all characters match both horizontally and vertically. Hence, this set of words can form a valid word square, and the algorithm would return `True`.

Solution Implementation

Python

```python
class Solution:
    def validWordSquare(self, words: List[str]) -> bool:
        # Get the number of words, which corresponds to the square's dimension
        num_words = len(words)

        # Loop through each word in the list
        for row, word in enumerate(words):
            # Loop through each character in the current word
            for col, char in enumerate(word):
                # Check if the current column is >= the number of words
                # Or if the current row is >= the length of the word at the current column index
                # Or if the character doesn't match the transposed character (symmetry check)
                if col >= num_words or row >= len(words[col]) or char != words[col][row]:
                    return False
        # If no mismatch was found, return True indicating the words form a valid word square
        return True

# Example of how to use it:
# sol = Solution()
# result = sol.validWordSquare(["abcd", "bnrt", "crm", "dt"])
# print(result) # Should output True if it's a valid word square or False otherwise
```

Java

```java
class Solution {
    public boolean validWordSquare(List<String> words) {
        // Get the number of words in the list
        int numRows = words.size();

        // Iterate over each word in the list
        for (int rowIndex = 0; rowIndex < numRows; ++rowIndex) {
            int numCharsInRow = words.get(rowIndex).length();

            // Iterate over each character in the current word
            for (int colIndex = 0; colIndex < numCharsInRow; ++colIndex) {
                // Check if the current column index is outside the number of rows
                // or if the row index is outside the length of the word at the current column index
                if (colIndex >= numRows || rowIndex >= words.get(colIndex).length()) {
                    return false; // The word square is invalid
                }

                // Check if the characters at the mirrored positions are not equal
                if (words.get(rowIndex).charAt(colIndex) != words.get(colIndex).charAt(rowIndex)) {
                    return false; // The word square is invalid
                }
            }
        }
        // If all checks pass, the word square is valid
        return true;
    }
}
```

C++

```cpp
class Solution {
public:
    // Function to check if the given vector of strings forms a valid word square
    bool validWordSquare(vector<string>& words) {
        // Get the number of words in the vector
        int numberOfRows = words.size();

        // Iterate through each word
        for (int row = 0; row < numberOfRows; ++row) {
            // Get the length of the current word
            int wordLength = words[row].size();

            // Iterate through each character in the current word
            for (int col = 0; col < wordLength; ++col) {
                // Check for three conditions:
                // 1. The column should not be beyond the number of rows
                // 2. The row should not be beyond the length of the word in the current column
                // 3. The characters at position [row][col] and [col][row] should match
                if (col >= numberOfRows ||                                   // Condition 1
                    row >= words[col].size() ||                              // Condition 2
                    words[row][col] != words[col][row]) {                    // Condition 3
                    // If any of the above conditions fail, return false
                    return false;
                }
            }
        }

        // If all conditions are satisfied, return true
        return true;
    }
};
```

TypeScript

```typescript
// This function checks if the array of words forms a valid word square
function validWordSquare(words: string[]): boolean {
    const squareSize = words.length; // squareSize represents the number of words

    // Iterate over each word in the array
    for (let rowIndex = 0; rowIndex < squareSize; ++rowIndex) {
        const wordLength = words[rowIndex].length; // Length of the current word

        // Iterate over each character in the word
        for (let columnIndex = 0; columnIndex < wordLength; ++columnIndex) {
            // Check if the columnIndex exceeds the number of words (squareSize),
            // or rowIndex exceeds the length of the word at columnIndex,
            // or if the character doesn't match the transposed character
            if (columnIndex >= squareSize ||
                rowIndex >= words[columnIndex].length ||
                words[rowIndex][columnIndex] !== words[columnIndex][rowIndex]) {
                return false; // The word square is not valid
            }
        }
    }

    return true; // All checks passed, the word square is valid
}
```

```python
class Solution:
    def validWordSquare(self, words: List[str]) -> bool:
        # Get the number of words, which corresponds to the square's dimension
        num_words = len(words)

        # Loop through each word in the list
        for row, word in enumerate(words):
            # Loop through each character in the current word
            for col, char in enumerate(word):
                # Check if the current column is >= the number of words
                # Or if the current row is >= the length of the word at the current column index
                # Or if the character doesn't match the transposed character (symmetry check)
                if col >= num_words or row >= len(words[col]) or char != words[col][row]:
                    return False
        # If no mismatch was found, return True indicating the words form a valid word square
        return True

# Example of how to use it:
# sol = Solution()
# result = sol.validWordSquare(["abcd", "bnrt", "crm", "dt"])
# print(result) # Should output True if it's a valid word square or False otherwise
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n * m^2)`, where `n` is the number of words in the input list and `m` is the maximum length of a word in the list. This is because the code uses a nested for-loop that iterates over every character in every word (`n * m`), and for each character, it potentially checks whether a character exists in the corresponding position in another word (up to `m` comparisons). However, in reality, the number of operations reduces as the loop checks for out-of-boundary conditions that may break the loop sooner; but in the worst-case scenario, we consider the maximum possible operations.

Space Complexity

The space complexity of the code is `O(1)`. This is because there are no data structures that grow proportionally with the input size. The algorithm only uses a fixed amount of space for variables in the loops regardless of the input size.