804. Unique Morse Code Words

Easy Array Hash Table String

Leetcode Link

Problem Description

International Morse Code defines a standard encoding where each letter from 'a' to 'z' corresponds to a unique sequence of dots (represented by '.') and dashes (represented by '-'). The task is to determine the number of unique Morse code representations of a given list of words.

Each word in the array `words` is a sequence of English lowercase letters, and the goal is to transform each word into its Morse code equivalent and then count the number of unique Morse code representations in the array. For example, the word "cab" would be transformed into "-.-..--..." by concatenating the Morse code for 'c' ("-.-."), 'a' (".-"), and 'b' ("-...").

To provide a solution, we should follow these steps:

- Convert each letter in a word to its corresponding Morse code by referencing the provided list.
- Concatenate all the Morse code representations for the letters in the word to form the Morse code representation of the word.
- Add the Morse code for each word to a set, which automatically filters out duplicates.
- Finally, return the count of unique Morse code representations in the set.

Intuition

The intuition behind the solution is to use the uniqueness property of sets in Python. Sets in Python can only contain unique elements, so by using a set, we can automatically ensure that only unique Morse code transformations of words are counted.

Another important observation is that the Morse code for each letter can be directly accessed using the ASCII value of the letter. This is done by subtracting the ASCII value of 'a' from the ASCII value of the current letter, resulting in the index of the Morse code for that letter. For instance, `'c' - 'a'` gives us the index of the Morse representation for the letter 'c'.

The steps in the solution code include:

1. Initialize an array with the Morse code representations for each of the 26 English lowercase letters.
2. Transform words into their Morse code equivalents using list comprehensions and string join operations.
3. Use a set to collect unique Morse code representations. Adding the transformations to the set ensures that duplicates are not counted.
4. Return the length of the set, which represents the number of unique Morse code representations among all words provided.

The simplicity of the solution results in an efficient approach to solving the problem with minimal additional storage space and only a single pass through the array of words.

Solution Approach

The solution to this problem involves a simple yet effective algorithm that mainly leverages Python's set data structure and array indexing.

Step-by-Step Implementation:

1. **Array of Morse Code Representations**: An array `codes` is initialized to store the Morse code for each letter. This array follows the sequence of the English alphabet from 'a' to 'z'.

2. **Conversion to Morse Code**: For each word in the `words` array, we convert it to its Morse code representation. This is achieved by iterating over each character in the word, finding its index in the alphabet (by subtracting the ASCII value of 'a' from the ASCII value of the letter), and then looking up the corresponding Morse code in the `codes` array.

3. **String Concatenation**: Python's join operation is used to concatenate the individual Morse codes into a single string representing the entire word.

4. **Set for Uniqueness**: A set `s` is employed to store the unique Morse code transformations of the words. As each Morse code string is created from a word, it is added to the set using a set comprehension. If the string is already in the set, it won't be added again, thus maintaining only unique entries.

5. **Count Unique Representations**: Finally, the length of the set `s` is returned. Since sets do not contain duplicates, the length of the set directly corresponds to the number of unique Morse code transformations.

Data Structures Used:

- **Array**: The Morse code representations are stored in an array where each index corresponds to a letter in the English alphabet.

- **Set**: A set is used to automatically handle the uniqueness of the Morse code transformations. Its properties ensure that it only contains unique Morse code strings.

Algorithmic Patterns Used:

- **Lookup**: This approach uses a simple lookup pattern where Morse codes are accessed via indices based on character ASCII values.

- **Set Comprehension**: The solution takes advantage of set comprehensions to build the set of unique Morse code transformations in a concise and readable way.

In summary, the algorithm converts each word to its Morse code representation, collects these into a set to filter out duplicates, and counts the number of elements in the set to determine the number of unique Morse code transformations.

Example Walkthrough

Let's take a set of words `["gin", "zen", "gig", "msg"]` and walk through the solution approach to determine the unique Morse code representations:

1. **Array of Morse Code Representations**: First, we prepare the `codes` array with the Morse code for each letter:

```
codes = [".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-", ".-..","--","-.","---",".--.","--.-",".-.","...","-
```

2. **Conversion to Morse Code**: Next, for each word in `["gin", "zen", "gig", "msg"]`, we convert it into Morse code. For example:

   - Taking "gin", we'll find the index for 'g', 'i', 'n' which are 6, 8, 13 respectively (0-indexed). Their Morse codes are "--.", "..", "-.".
   - Concatenate them together to get the Morse representation for "gin": `--...-.`.

3. **String Concatenation**: Repeat the process for the other words:

   - "zen" becomes `--...-.`
   - "gig" becomes `--...--.`
   - "msg" becomes `--...--.`.

4. **Set for Uniqueness**: Now, let's add each Morse code representation of the words to a set `s`:

```
s = {"--...-.", "--...-.", "--...--.", "--...--."}
```

   Note that the Morse codes for "gin" and "zen" are identical, as are those for "gig" and "msg", which the set will automatically consolidate.

5. **Count Unique Representations**: Lastly, we determine the unique Morse code representations by the count of the set `s`. In this case, the set reduces to `{"--...-.", "--...--."}`, which has a length of 2.

The output for this set of words is `2`, meaning there are two unique Morse code representations among the provided words.

Python Solution

```python
class Solution:
    def uniqueMorseRepresentations(self, words: List[str]) -> int:
        # Define the Morse code representations for each lowercase alphabet letter
        morse_codes = [".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....",
                       "..", ".---", "-.-", ".-..", "--", "-.", "---", ".--.",
                       "--.-", ".-.", "...", "-", "..-", "...-", ".--", "-..-",
                       "-.--", "--.."]

        # Use a set to store unique Morse code transformations of the words
        unique_transformations = set()

        # Loop through each word in the provided list of words
        for word in words:
            # Transform the word into a Morse code representation
            morse_word = ''.join([morse_codes[ord(char) - ord('a')] for char in word])
            # Add the transformed Morse code word to the set of unique transformations
            unique_transformations.add(morse_word)

        # Return the count of unique Morse code transformations
        return len(unique_transformations)

```

Java Solution

```java
// Solution class to find the number of unique Morse code representations from a list of words.
class Solution {
    public int uniqueMorseRepresentations(String[] words) {
        // Array of Morse code representations for each letter from a to z.
        String[] morseCodes = new String[] {
            ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....",
            "..", ".---", "-.-", ".-..", "--", "-.", "---", ".--.",
            "--.-", ".-.", "...", "-", "..-", "...-", ".--", "-..-",
            "-.--", "--.."
        };

        // Set to store unique Morse code transformations of the words.
        Set<String> uniqueTransformations = new HashSet<>();

        // Iterate through each word in the input list.
        for (String word : words) {
            // StringBuilder to accumulate Morse code for the current word.
            StringBuilder morseWord = new StringBuilder();

            // Convert each character in the word to its corresponding Morse code.
            for (char ch : word.toCharArray()) {
                morseWord.append(morseCodes[ch - 'a']); // Subtract 'a' to get the index.
            }

            // Add the Morse code transformation to the set.
            uniqueTransformations.add(morseWord.toString());
        }

        // Return the size of the set, which is the number of unique Morse representations.
        return uniqueTransformations.size();
    }
}

```

C++ Solution

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

class Solution {
public:
    // Function to count the unique Morse code representations for a list of words.
    int uniqueMorseRepresentations(vector<string>& words) {
        // Array of Morse code representations for each alphabet character.
        vector<string> morseCodes = {
            ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....", "..", ".---",
            "-.-", ".-..", "--", "-.", "---", ".--.", "--.-", ".-.", "...", "-",
            "..-", "...-", ".--", "-..-", "-.--", "--.."
        };

        // Using a set to store unique Morse code transformations of the words.
        // Sets in C++ are generally ordered; unordered_set is typically more efficient.
        unordered_set<string> uniqueTransforms;

        // Loop over each word in the list of words.
        for (const auto& word : words) {
            string transformedWord;
            // Loop over each character in the word and convert to Morse code.
            for (const char& letter : word) {
                // Append the corresponding Morse code for the character to the transformed word string.
                // 'a' has an ASCII value of 97, so 'a' - 'a' will be 0, 'b' - 'a' will be 1, and so on,
                // thus mapping characters to correct Morse code strings.
                transformedWord += morseCodes[letter - 'a'];
            }
            // Insert the transformed word into the set.
            uniqueTransforms.insert(transformedWord);
        }

        // The size of the set represents the number of unique Morse code transformations.
        // Sets do not allow duplicate elements, so the count will only be of unique items.
        return uniqueTransforms.size();
    }
};

```

Typescript Solution

```typescript
// Morse code representations for the 26 letters of the English alphabet.
const morseCodes = [
    '.-',   // a
    '-...', // b
    '-.-.', // c
    '-..',  // d
    '.',    // e
    '..-.', // f
    '--.',  // g
    '....', // h
    '..',   // i
    '.---', // j
    '-.-',  // k
    '.-..', // l
    '--',   // m
    '-.',   // n
    '---',  // o
    '.--.', // p
    '--.-', // q
    '.-.',  // r
    '...',  // s
    '-',    // t
    '..-',  // u
    '...-', // v
    '.--',  // w
    '-..-', // x
    '-.--', // y
    '--..', // z
];

/**
 * Converts an array of English words to their unique Morse code representations
 * and returns the count of unique Morse code strings.
 * @param {string[]} words - The array of words to be converted into Morse code.
 * @return {number} - The count of unique Morse code representations.
 */
function uniqueMorseRepresentations(words: string[]): number {
    // Transform each word into its Morse code representation
    // and store the unique Morse code strings in a Set.
    const uniqueMorseTransformations = new Set(
        words.map(word => {
            return word
                // Split each word into characters.
                .split('')
                // Convert each character to its corresponding Morse code.
                .map(character => morseCodes[character.charCodeAt(0) - 'a'.charCodeAt(0)])
                // Join the Morse code sequence to get the word's Morse representation.
                .join('');
        })
    );

    // Return the size of the set, which represents the count of unique Morse code strings.
    return uniqueMorseTransformations.size;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code can be analyzed as follows:

- There is one list, `codes`, that maps each letter of the English lowercase alphabet to its corresponding Morse code representation. This list is initialized only once, and this operation is `O(1)` because the size of the Morse code alphabet (and hence the list) is constant and does not scale with the size of the input.
- The main operation is the set comprehension `{... for word in words}`, which iterates over each word in `words`.

For each word:

- Iterating over each character in the word takes `O(n)` time, where `n` is the length of the word.
- Accessing the Morse code for each character is an `O(1)` operation.
- Joining the Morse codes to form a single string has a time complexity of `O(m)`, where `m` is the total length of the Morse code for the word, which is linear in the length of the word.

Assuming `w` is the number of words and the average length of a word is represented as `avg_len`, then the time complexity becomes `O(w * avg_len)`. For `w` iterations, and `avg_len` being the average time per iteration.

Space Complexity

The space complexity can be analyzed as follows:

- The list `codes` has a fixed size (constant space) of 26 elements, which corresponds to the number of letters in the English alphabet. So, it is `O(1)`.
- The set `s` will contain at most `w` unique Morse representations if all words have unique Morse code translations. Since each word translates to a different length string based on its characters, let's denote `max_morse_len` as the maximum length of these Morse code strings for any word. Hence, the space complexity is `O(w * max_morse_len)`.

Therefore, the overall space complexity of the function would be `O(w * max_morse_len)` reflecting the space needed to store the unique transformations.