1178. Number of Valid Words for Each Puzzle

Hard | Bit Manipulation | Trie | Array | Hash Table | String Leetcode Link

Problem Description

Given a list of words and a list of puzzles, the aim is to figure out how many words from the word list are valid according to a specific pair of conditions for each puzzle. A word is valid with respect to a puzzle if:

1. The word contains the first letter of the puzzle.
2. Every letter of the word is also in the given puzzle.

It's important to note that it is not necessary for all letters of the puzzle to be in the word. For example, for the puzzle "abcdefg", words such as "faced", "cabbage", and "baggage" are valid because they include the first letter 'a' of the puzzle and all other letters in these words are contained within the puzzle. On the other hand, "beefed" and "based" are invalid because "beefed" lacks the letter 'a', and "based" contains the letter 's', which is not present in the puzzle.

The output is an array where each element corresponds to the number of valid words for each puzzle.

Intuition

The intuition behind the solution is based on converting both puzzles and words into binary representations where each bit corresponds to a letter's presence. To tackle this kind of problem, we use bitwise operations to perform checks and counts efficiently.

To represent a set of letters as a binary number, we assign each letter a bit position based on its order in the alphabet. For instance, 'a' corresponds to the least significant bit, and 'z' to the most significant bit of a 26-bit integer, since there are 26 letters in the English alphabet.

We start by creating masks for the words - a mask is a binary number where a bit is set (i.e., 1) if the corresponding letter is in the word. We use a counter to keep track of how many times every possible combination (mask) appears in the list of words. This preprocessing is helpful because we need to compare each word with multiple puzzles, and using masks allows us to do these comparisons quickly using bitwise operations.

For each puzzle, we also create a mask. We then need to count all valid word masks with respect to the puzzle mask. A word mask is valid if it includes the first letter of the puzzle, and all bits set in the word mask are also set in the puzzle mask. To find these valid word masks, we iterate over every submask (a mask with some bits possibly turned off, but still containing the first letter of the puzzle) of the puzzle mask and sum their counts in the counter.

This way, we leverage the precomputed frequencies of each word mask to efficiently calculate the number of valid words for each puzzle.

Solution Approach

The implementation of the solution involves understanding how bitwise operations can effectively represent unique sets of characters and allow for fast subset enumeration. Here's the step-by-step approach, highlighted by key algorithms, data structures, and patterns:

1. **Counter for masks of words**: We create a `Counter` from the `collections` module in Python to keep track of all the unique masks created from the `words` list. To generate a mask, the solution goes through each character in a word and performs an OR `|` operation between the mask (initialized to zero) and a bit shifted by the alphabetical position of the character (`1 << (ord(c) - ord("a"))`). This results in a number where the bits corresponding to letters in the word are set to 1.

2. **Processing the puzzles**: For each puzzle in `puzzles`, we:
   - Compute the puzzle's mask in the same way as the word's mask.
   - Store the first letter's position in `i` because it must be present in each valid word.
   - Initialize `x`, which will accumulate the count of valid words for this puzzle.
3. **Subset enumeration**: Enumerate submasks of the puzzle's mask:
   - The inner while loop does the following: Starting with `j` as the puzzle mask, we repeatedly find submasks by decrementing `j`. The expression `j = (j - 1) & mask` turns off one bit at a time in the submask that is also on in the puzzle's mask.
   - If the submask contains the first letter of the puzzle (`j >> i & 1), we add the count of this mask from our precomputed` Counter`to`x`.
4. **Edge case handling**: For puzzles that don't even have one word matching, the count would simply be added as zero.

5. **Result assembly**: Finally, the counts of valid words (stored in `x`) for each puzzle are appended to the `ans` list, which represents the number of valid words for each puzzle.

Through this application of bitwise operations, subsets enumeration and using a counter to map set bits to frequencies, we achieve an efficient solution to a problem that might seem combinatorial and complex at first glance.

Mathematically, if `|W|` is the length of the word list, and `|P|` is the length of the puzzle list, the algorithm efficiently compresses the O(|W| * |P|) brute-force approach into a faster one by using bit masks and counting common patterns.

Example Walkthrough

Let's use a small example to illustrate the solution approach. Say our word list is `words = ["echo", "hold", "chef"]` and our puzzle list `puzzles = ["chloe", "dolph"]`.

Step 1: Creating masks for words and initializing Counter:

We iterate through each word in `words` to create a binary mask. For instance, for "echo" the binary mask would be:

- e: 1 << (4) ⇒ 00010000
- c: 1 << (2) ⇒ 00000100
- h: 1 << (7) ⇒ 10000000
- o: 1 << (14) ⇒ 0100000000000000

OR all the above: `mask for echo = 11010100`

Perform this process for each word:

- echo: 11010100
- hold: 1000010010010
- chef: 10011000

Our `Counter` would contain the count of the unique masks. From the example `words`, we have unique masks so each would have a count of 1 in the `Counter`.

Step 2: Processing the puzzles:

For "chloe", the binary mask would be computed similarly to `words`. So:

- mask for "chloe" ⇒ 11011101

Step 3 & 4: Enumerating subsets and counting the valid words:

For puzzle "chloe", we find all the subset masks of the puzzle's mask. The first letter 'c' must be included, so the submask must have the bit for 'c' set. Then we enumerate by repeatedly removing one bit until 'c' is the only bit left.

The submasks of "chloe" we consider will be (represented as strings for simplicity): "11011101", "11011100", "11011001", ... , "00000100" (which represents just 'c').

For each submask, we check if our word masks match.

- The mask of "echo" (`11010100`) is a subset of "chloe" mask, it's valid.
- The mask of "hold" (`1000010010010`) is not a subset since 'd' is not in "chloe".
- The mask of "chef" (`10011000`) is also a valid subset since all bits match and the first letter 'c' is present.

So, count for "chloe" is 2.

For "dolph", we would do the same. The word "hold" would be the only match since it's the only word containing 'd' and all its letters are in the puzzle "dolph". So, count for "dolph" is 1.

Step 5: Result assembly:

Finally, we would assemble our results into an array with `[2, 1]`, which indicates the number of valid words for the puzzles "chloe" and "dolph", respectively.

Python Solution

```python
from collections import Counter
from typing import List

class Solution:
    def findNumOfValidWords(self, words: List[str], puzzles: List[str]) -> List[int]:
        # Initialize a counter to keep track of frequency of each unique bitmask
        bitmask_counter = Counter()
        for word in words:
            # Create a bitmask for each word
            bitmask = 0
            for char in word:
                # Set the bit at the position corresponding to the character
                bitmask |= 1 << (ord(char) - ord('a'))
            # Increase the count of this specific bitmask
            bitmask_counter[bitmask] += 1

        # List to store the result for each puzzle
        results = []
        for puzzle in puzzles:
            # Create a bitmask for the puzzle
            puzzle_bitmask = 0
            for char in puzzle:
                # Set the bit for each character in the puzzle
                puzzle_bitmask |= 1 << (ord(char) - ord('a'))

            # Count of valid words for this puzzle
            valid_word_count = 0
            # The first character of the puzzle is essential, store its bit position
            first_char_bit = ord(puzzle[0]) - ord('a')
            # Start with the puzzle's bitmask and generate all submasks
            submask = puzzle_bitmask
            while submask:
                # Check if the submask includes the first character of the puzzle
                if submask >> first_char_bit & 1:
                    # Add count of words that match this submask
                    valid_word_count += bitmask_counter[submask]
                # Generate the next submask by 'turning off' the rightmost 'on' bit
                submask = (submask - 1) & puzzle_bitmask

            # Append the count of valid words for this puzzle to the results list
            results.append(valid_word_count)
        return results

```

Java Solution

```java
class Solution {
    public List<Integer> findNumOfValidWords(String[] words, String[] puzzles) {
        // Create a frequency map to store the number of occurrences of each word's bitmask
        Map<Integer, Integer> frequencyMap = new HashMap<>(words.length);

        // Loop over each word to calculate the bitmask and update the frequency map
        for (String word : words) {
            int bitmask = 0;  // Initialize bitmask for the current word
            for (int i = 0; i < word.length(); ++i) {
                // For each character in the word, update the bitmask
                bitmask |= 1 << (word.charAt(i) - 'a');
            }
            // Merge the current bitmask into the frequency map, summing up the counts
            frequencyMap.merge(bitmask, 1, Integer::sum);
        }

        // Create a list to store the final count of valid words for each puzzle
        List<Integer> result = new ArrayList<>();

        // Loop over each puzzle to calculate count of valid words
        for (String puzzle : puzzles) {
            int bitmask = 0; // Initialize bitmask for the current puzzle
            for (int i = 0; i < puzzle.length(); ++i) {
                // For each character in the puzzle, update the bitmask
                bitmask |= 1 << (puzzle.charAt(i) - 'a');
            }

            // Initialize the valid word count for the current puzzle
            int validWordCount = 0;
            // Calculate the bitmask for the first letter of the puzzle (required in all words)
            int firstLetterBitmask = 1 << (puzzle.charAt(0) - 'a');

            // Iterate over all submasks of the puzzle bitmask to find valid words
            for (int submask = bitmask; submask > 0; submask = (submask - 1) & bitmask) {
                // Check if the submask includes the first letter of the puzzle
                if ((submask & firstLetterBitmask) != 0) {
                    // Add the count of words matching the submask to the total count
                    validWordCount += frequencyMap.getOrDefault(submask, 0);
                }
            }
            // Add the calculated count to the result list
            result.add(validWordCount);
        }
        // Return the list of valid word counts for each puzzle
        return result;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_map>
using namespace std;

class Solution {
public:
    vector<int> findNumOfValidWords(vector<string>& words, vector<string>& puzzles) {
        // Map to store the frequency of each unique character mask for the words
        unordered_map<int, int> frequency;

        // Populate the frequency map with the bitmask representation of words
        for (auto& word : words) {
            int mask = 0; // Initialize bitmask for the current word
            for (char& ch : word) {
                mask |= 1 << (ch - 'a'); // Set the bit corresponding to the character
            }
            frequency[mask]++; // Increase the count for this bitmask representation of the word
        }

        // Vector to store the result, one entry for each puzzle
        vector<int> results;

        // Process each puzzle and find the number of valid words for it
        for (auto& puzzle : puzzles) {
            int mask = 0; // Initialize bitmask for the current puzzle
            for (char& ch : puzzle) {
                mask |= 1 << (ch - 'a'); // Set the bit corresponding to each character
            }

            int count = 0; // Initialize the valid words count for the current puzzle
            int firstCharBit = 1 << (puzzle[0] - 'a'); // Bitmask for the first puzzle character

            // Iterate through all submasks of the puzzle mask
            for (int submask = mask; submask; submask = (submask - 1) & mask) {
                // Check if the submask includes the first character of the puzzle
                if (submask & firstCharBit) {
                    count += frequency[submask]; // If it does, add the word frequency to the count
                }
            }
            // Add the count to the results vector
            results.push_back(count);
        }

        // Return the results vector containing the number of valid words for each puzzle
        return results;
    }
};

```

Typescript Solution

```typescript
function findNumOfValidWords(words: string[], puzzles: string[]): number[] {
    // Create a map to count the frequency of each unique character mask
    const frequencyMap: Map<number, number> = new Map();

    // Generate a bitmask for each word and count their frequency
    words.forEach(word => {
        let bitmask = 0;
        for (const char of word) {
            bitmask |= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0));
        }
        frequencyMap.set(bitmask, (frequencyMap.get(bitmask) || 0) + 1);
    });

    // Initialize an array to store the number of valid words for each puzzle
    const result: number[] = [];

    // Calculate the number of valid words for each puzzle
    puzzles.forEach(puzzle => {
        let puzzleBitmask = 0;
        for (const char of puzzle) {
            puzzleBitmask |= 1 << (char.charCodeAt(0) - 'a'.charCodeAt(0));
        }

        let validWordsCount = 0;
        const firstCharBit = 1 << (puzzle.charCodeAt(0) - 'a'.charCodeAt(0));

        // Use subset generation method to iterate over all subsets of the puzzle bitmask
        for (let subset = puzzleBitmask; subset; subset = (subset - 1) & puzzleBitmask) {
            // Check if the first character of the puzzle is in the current subset
            if (subset & firstCharBit) {
                validWordsCount += frequencyMap.get(subset) || 0;
            }
        }

        // Push the count of valid words for this puzzle into the result array
        result.push(validWordsCount);
    });

    return result;
}

```

Time and Space Complexity

Time Complexity

The given algorithm involves two main parts, constructing a bitmask for each word and puzzle and then solving the puzzles.

1. For each word in the list of words:
   - We iterate through all the characters in the word and create a bitmask representing the unique characters. Assuming the average length of the words is `L`, this operation is `O(L)`.
   - However, we perform this operation for all `N` words. So, the time complexity for this part is `O(N * L)`.
2. For each puzzle in the list of puzzles:
   - Similar to words, we create a bitmask for the puzzle. Assuming the average length of puzzles is `P`, this operation is `O(P)`.
   - The novel part is trying out different combinations of the puzzle's letters represented by the bitmask and checking those combinations in the word counter. In the worst-case scenario, there could be `2^P` combinations if each puzzle has distinct letters.
   - Since we need to check each of these combinations against our counter, and the first letter of the puzzle must be included, we're looking at `O(2^(P-1))` operations for each puzzle. This is because we iterate through all the subsets of the bitmask that includes the first letter.
   - Performing this for all `M` puzzles, the total time complexity for solving puzzles is `O(M * 2^(P-1))`.

Combining the two parts, the total time complexity is `O(N * L + M * 2^(P-1))`. Note that `L` and `P` usually have upper limits (since the question constrains are such that the length of the words and puzzles would not exceed a certain length e.g., 7 for puzzles), so for very large `N` and `M` the dominant term could be either depending on the values of `N`, `M`, `L`, and `P`.

Space Complexity

1. Counter for words (`cnt`): This will store the frequency of each unique bitmask for the words. The space used by the counter will be proportional to the number of unique bitmasks rather than the number of words themselves. Since there are at most `26` characters, and each character could be present or absent, we theoretically can have at most `2^26` entries, though in practice, this will be much less due to the constraints of the problem (e.g., if words are of limited length). Therefore, the space complexity for cnt can be considered `O(26)` assuming the length of the word is limited, but in the unconstrained case, it would be `O(2^26)`.

2. The space complexity for storing the answer (`ans`) is `O(M)`, where `M` is the number of puzzles.

The total space complexity of the algorithm is, therefore, `O(2^26 + M)`, but again, assuming a practical limit to the number of unique bitmasks due to constraints on the word length, the space would be more accurately described as `O(26 + M)`.