1255. Maximum Score Words Formed by Letters

Problem Description

The given problem is an optimization challenge where we have to find the maximum score achievable by forming valid sets of words from a given list. Each word has an associated score based on the characters it contains and a score table provided for the alphabet. The words come from a list called `words`, and we are given a list of available single `letters` which may contain duplicates. Each letter in `letters` can be used only once. The score for each letter is given by an array `score`, where `score[0]` corresponds to the score of 'a', `score[1]` to the score of 'b', and so on until `score[25]`, which corresponds to the score of 'z'.

The goal is to select words from the `words` list in such a way that we can form them from the letters in the `letters` list without reusing any single letter more than its available count. We cannot use any word more than once. We should return the highest possible score that can be obtained from any combination of the words formed.

Intuition

To solve the problem, we think of all possible subsets of the provided `words` array. For each subset, we calculate the score of its constituent words if it's possible to create the subset with the given `letters`. We only consider a subset valid if each letter required to form this subset of words is available the required number of times in the `letters` list. We use bitwise operations to generate possible subsets.

Generating subsets: We use a binary representation of numbers to iterate over all possible subsets. For a list of n words, there are 2^n possible subsets. We can represent the subsets using a bitmask of length n, where the j-th bit indicates whether or not to include the j-th word in the subset.

Counting letters in words: We use a data structure, `Counter`, to count the occurrences of each letter in both the currently considered subset of words and the available letters.

Checking validity and score calculation: For a chosen subset, we check if it can be created with the given letters by ensuring the count of each letter in the subset does not exceed the count of available letters. If valid, we calculate the subset's score by summing the individual character scores. This is done by referring to the supplied `score` array using the character's ASCII value adjusted with the ASCII value of 'a' to correctly index into the score list.

Maximizing the score: We compare each valid subset's score to the current known maximum score and update the maximum score if a higher one is found.

Optimization: Although the proposed solution methodically generates all potential subsets and assesses each one, it is not efficient for large lists of words due to its exponential time complexity. However, this brute-force approach provides a direct and complete search of the solution space, which is acceptable for smaller inputs.

Solution Approach

The implementation of the solution can be explained step by step as follows:

1. **Initialization**: A `Counter` object named `cnt` is created to store the count of each available letter in the `letters` list. Next, the length of the `words` list is stored in variable `n`, and a variable `ans` is initialized to zero to keep track of the maximum score.
2. **Iterating over subsets**: A `for` loop is used to iterate through numbers i to $2^n - 1$ representing all possible subsets where n is the number of words. Inside this loop, the binary representation of i represents a subset where each bit in the representation of the number indicates the presence (1) or absence (0) of a word at that index in the subset.
3. **Building a subset**: For each number i in the loop, we use a list comprehension to collect words corresponding to set bits in the binary representation of the number. This is done with `[words[j] for j in range(n) if i >> j & 1]`. Then, a `Counter` object named `cur` is created to store the count of each letter in this subset of words.
4. **Checking if the subset is valid**: The `all` function is used to verify that for all characters `c` and their count `v` in the subset `cur`, there is a sufficient number of letters in `cnt` (`v <= cnt[c]`). This ensures that the current subset can be formed from the available letters without reusing any letter more than available.
5. **Calculating the score of the subset**: If the subset is valid, the score is computed using the `sum` function. For each character `c` and its count `v` in `cur`, the expression `v * score[ord(c) - ord('a')]` calculates the total score of the character in the subset by multiplying its count with its respective score in `score`. The `ord` function gets the ASCII value of the character, and subtracting `ord('a')` gives the index of the score for that character in the score array.
6. **Updating the maximum score**: The score of the current valid subset `s` is compared with the current `ans`, and if `s` is greater, `ans` is updated to `s`.
7. **Returning the result**: After the loop finishes executing, the final value of `ans` is returned, which contains the maximum score obtained from any valid subset of words.

The solution uses a **bitmasking technique** to generate all possible subsets of words and a **Counter data structure** to efficiently count the occurrences of letters for available letters and each subset of words. The **brute-force approach** checks each subset for validity and calculates its score independently. Despite its simplicity, the time complexity of this solution is $O(2^n * (n + L))$, where n is the number of words, L is the length of the longest word, and L is the number of unique characters in `letters`, which may not be performant for large datasets.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach:

Assume we have the following inputs:

- `words = ["dog", "cat", "dad", "good"]`
- `letters = ["a", "a", "c", "d", "d", "g", "o", "o"]`
- `score = [1, 0, ..., 2, ..., 0, ..., 1, ..., 0, 3]` // score for "a" is 1, "c" is 2, "d" is 1, "g" is 2, "o" is 1

The score array only shows the scores for the relevant characters for brevity.

1. **Initialization**: We calculate the count of each letter in the `letters` list. The counter would look like `cnt = {'a': 2, 'c': 1, 'd': 2, 'g': 1, 'o': 2}`. The number of words `n = 4`, and the starting maximum score `ans = 0`.
2. **Iterating over subsets**: We generate numbers from i to $2^n - 1$ which is from 0 to 15 for $n = 4$. Each representation will be from `0000` to `1111` in binary, corresponding to taking none or all the words, respectively.
3. **Building a subset**: When the loop is at $i = 3$ (binary `0011`), it includes including `words[0]` and `words[1]` which are "dog" and "cat". We determine the count of letters in the subset, `cur = Counter("dogcat") = {'d': 1, 'o': 1, 'g': 1, 'c': 1, 'a': 1}`.
4. **Checking if the subset is valid**: We compare `cur` against `cnt` and find that all letters in `cur` are less than or equal to those in `cnt` (`{'d': 1, 'o': 1, 'g': 1, 'c': 1, 'a': 1} <= {'a': 2, 'c': 1, 'd': 2, 'g': 1, 'o': 2}`). Therefore, the subset is valid.
5. **Calculating the score of the subset**: The score of the subset "dogcat" is calculated as follows:
   - "dog" contributes `1*score['d'] + 1*score['o'] + 1*score['g']` which is `1*1 + 1*1 + 1*2 = 4` because `score[3]` for 'd', `score[14]` for 'o', and `score[6]` for 'g'.
   - "cat" contributes `1*score['c'] + 1*score['a'] + 1*score['t']` which is `1*2 + 1*1 + 1*0 = 3` because `score[2]` for 'c', `score[0]` for 'a', and `score[19]` for 't'.
   - Total subset score is `4 + 3 = 7`.
6. **Updating the maximum score**: Since `7` is greater than `0`, we update `ans` to `7`.
7. **Continuing**: The loop continues, testing the validity of other subsets and updating `ans` as necessary. After testing all possible subsets, the highest score found is returned. For instance, if the algorithm later encounters the subset "good" with a higher score than `7`, `ans` will be updated accordingly.
8. **Returning the result**: After completion, suppose the highest score was achieved by the subset "good" and it was `8`. Then, `ans` will be `8`, which is the result returned by the function.

This walkthrough exemplifies the steps and considerations of the solution approach on a smaller scale of inputs, demonstrating how the brute-force method selects and sums up word scores to find the maximum possible score given the constraints.

Python Solution

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

class Solution:
    def max_score_words(self, words: List[str], letters: List[str], score: List[int]) -> int:
        # Count the frequency of each letter available
        cnt = Counter(letters)
        # Determine the total number of words
        number_of_words = len(words)
        # Initialize the maximum score
        maximum_score = 0
        # Iterate over all combinations of words using binary representation
        # Each bit in `i` represents whether to include a word at that position or not
        for i in range(1 << number_of_words):
            # Use a Counter to track the frequency of letters in the chosen combination of words
            current_word_counter = Counter(''.join(words[j] for j in range(number_of_words) if (i >> j) & 1))
            # Check if the combination of words can be built with the given letters
            if all(current_word_counter[letter] <= cnt[letter] for letter in current_word_counter):
                # Calculate the score for this combination of words
                current_score = sum(count * score[ord(letter) - ord('a')] for letter, count in current_word_counter.items())
                # Update the maximum score if the current score is higher
                maximum_score = max(maximum_score, current_score)

        # Return the maximum score found
        return maximum_score
```

Java Solution

```java
class Solution {
    /**
     * Calculates the maximum score achievable by forming words from the given letters.
     *
     * @param words  An array of words to be considered for scoring.
     * @param letters An array of letters available to form words.
     * @param score  An array where the score for each alphabet letter.
     * @return The maximum score achievable.
     */
    public int maxScoreWords(String[] words, char[] letters, int[] score) {
        // Count the occurrences of each letter in the given letters array.
        int[] letterCounts = new int[26];
        for (char letter : letters) {
            letterCounts[letter - 'a']++;
        }

        int wordCount = words.length;
        int maxScore = 0; // Keep track of the maximum score.

        // Loop through each combination of words possible.
        for (int mask = 0; mask < (1 << wordCount); ++mask) {
            int[] currentCounts = letterCounts.clone(); // To hold the count for current combination.
            // Check each word in the current combination.
            for (int i = 0; i < wordCount; ++i) {
                if ((mask >> i & 1) == 1) { // If the i-th word is included.
                    for (char c : words[i].toCharArray()) {
                        currentCounts[c - 'a']--; // Count the characters for the word.
                    }
                }
            }

            // Verify if current combination can be made from available letters.
            boolean isCombinationValid = true;
            int currentScore = 0;
            for (int i = 0; i < 26; ++i) {
                if (currentCounts[i] < letterCounts[i]) { // More letters required than available.
                    isCombinationValid = false;
                    break;
                }
                currentScore += currentCounts[i] * score[i]; // Accumulate score for the current combination.
            }

            // Update maxScore if this combination is valid and its score is higher.
            if (isCombinationValid && maxScore < currentScore) {
                maxScore = currentScore;
            }
        }

        // Return the maximum score after evaluating all combinations.
        return maxScore;
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    // Function to calculate the maximum score of a set of words given a set of letters and their corresponding scores.
    int maxScoreWords(vector<string>& words, vector<char>& letters, vector<int>& score) {
        int letterCounts[26] = {0}; // Array to store the count of each letter in 'letters'
        // Count the occurrence of each letter in 'letters'
        for (char letter : letters) {
            letterCounts[letter - 'a']++; // Increment the count for the corresponding letter
        }

        int numWords = words.size(); // Total number of words
        int maxScore = 0; // Variable to store the maximum score

        // Loop through all combinations of words
        for (int wordCombination = 0; wordCombination < (1 << numWords); ++wordCombination) {
            int wordCounts[26] = {0}; // Current letter counts for the current combination of words
            // Build the count for the current combination
            for (int wordIndex = 0; wordIndex < numWords; ++wordIndex) {
                // Check if the current word is included in the current combination
                if ((wordCombination >> wordIndex) & 1) {
                    // If so, count each letter in the word
                    for (char& letter : words[wordIndex]) {
                        wordCounts[letter - 'a']++; // Increment the count for the corresponding letter
                    }
                }
            }

            bool isValidCombination = true; // Flag to check if the combination is valid
            int currentScore = 0; // Score for the current combination
            // Check if the current combination is valid and calculate its score
            for (int i = 0; i < 26; ++i) {
                // If any letter count exceeds what is available, the combination is invalid
                if (wordCounts[i] > letterCounts[i]) {
                    isValidCombination = false;
                    break; // No need to continue if the combination is invalid
                }
                // Calculate score for the current combination
                currentScore += wordCounts[i] * score[i];
            }

            // If the combination is valid and the score is higher than the maximum found so far, update maxScore
            if (isValidCombination && maxScore < currentScore) {
                maxScore = currentScore;
            }
        }

        // Return the maximum score possible with the given words and letters
        return maxScore;
    }
};
```

Typescript Solution

```typescript
// Initializes the letter count array to track the occurrences of each letter.
const letterCounts: number[] = Array(26).fill(0);
// Variables to hold the words, letters, and score for global access.
let words: string[], letters: string[], score: number[];
// Number of words to be considered.
let maxScore: number = 0;

// Preprocesses the 'letters' array to populate the 'letterCounts'.
function preprocessLetters() {
    for (const letter of letters) {
        letterCounts[letter.charCodeAt(0) - 'a'.charCodeAt(0)]++;
    }
}

// Calculates the maximum score for a combination of words.
function calculateMaxScore() {
    const numWords = words.length;
    // Iterates through all possible combinations of words using bitwise operations.
    for (let combination = 0; combination < (1 << numWords); combination++) {
        const wordCounts = Array(26).fill(0); // Counts for the current combination.
        // Builds the count for the current combination by iterating through each word.
        for (let wordIndex = 0; wordIndex < numWords; wordIndex++) {
            // Checks if the word at 'wordIndex' is included in the current combination.
            if ((combination >> wordIndex) & 1) {
                // Increments the count for each letter in the word.
                for (const letter of words[wordIndex]) {
                    wordCounts[letter.charCodeAt(0) - 'a'.charCodeAt(0)]++;
                }
            }
        }

        let isValidCombination: boolean = true; // Flag to check the combination's validity.
        let currentScore: number = 0; // Score accumulator for the current combination.
        // Calculating score and checking the validity of the combination.
        for (let i = 0; i < 26; i++) { // Iterates over the alphabet.
            // If the letter count exceeds available, mark the combination as invalid.
            if (wordCounts[i] > letterCounts[i]) {
                isValidCombination = false;
                break;
            }
            // Summing up the score for the current combination.
            currentScore += wordCounts[i] * score[i];
        }

        // Update maxScore if the current score is greater and the combination is valid.
        if (isValidCombination && maxScore < currentScore) {
            maxScore = currentScore;
        }
    }
}

// Function to initiate the calculation of max score by processing the input data.
function maxScoreWords(inputWords: string[], lettersInput: string[], scoreInput: number[]): number {
    words = wordsInput;
    letters = lettersInput;
    score = scoreInput;
    preprocessLetters(); // Fills the letterCounts based on the letters available
    calculateMaxScore(); // Calculates the score for each possible combination
    return maxScore; // Returning the highest score found
}
```

Time and Space Complexity

The given code snippet is a solution to the problem of finding the maximum score from forming words using given letters with each letter having a score.

Time Complexity:

To analyze the time complexity, let's consider n to be the number of words and l to be the total number of letters across all words.

1. The code uses a bit mask to generate all possible combinations of words, which results in 2^n combinations as it iterates over a range of size 2^n.
2. For each combination, it attempts to create a Counter object for the selected words. This operation is linear with respect to the length of the concatenated string of chosen words. In the worst case, this could be $O(l)$ when all words are chosen.
3. The `all` function inside the loop checks if all letters in the current combination are less than or equal to the count of available letters. This runs in $O(26)$ since there are 26 lowercase English letters, in the worst case where every letter is used.
4. The sum function, which calculates the total score of current combination of words, will iterate over each character in the concatenated string. This is $O(l)$ in the worst case for the same reason as step 2.
5. Updating the maximum answer is $O(1)$ for each of the 2^n combinations.

Combining these steps, the worst-case time complexity is $O(2^n * (l + 26 + 1 + 1))$, which simplifies to $O(2^n * l)$ as l dominates the constant 28 and 1.

Space Complexity:

Now, let's look at the space complexity:

1. The counter `cnt` stores the frequency of each letter available. This is at most $O(26)$, which is a constant $O(1)$ because we only have 26 letters.
2. The `cur` counter inside the loop could store at most $O(26)$ entries due to the lowercase letters, so this is also $O(1)$.
3. The list `[words[j] for j in range(n) if i >> j & 1]` is ephemeral and its space complexity in terms of the number of characters it can hold is up to $O(l)$, when all words are selected.
4. The combination bitmask takes up to $O(n)$ space to store the indices of words.

Considering all the above, the space complexity of the code is dominated by the space needed for the ephemeral list of selected words and the bitmask, and hence it is $O(l + n)$.

To summarize, the time complexity is $O(2^n * l)$, and the space complexity is $O(l + n)$.