2131. Longest Palindrome by Concatenating Two Letter Words

Medium Greedy Array Hash Table String Counting [Leetcode Link]

Problem Description

In this problem, we have an array of strings called `words`. Each string is composed of exactly two lowercase English letters. Our objective is to construct the longest possible palindrome by selecting and concatenating some elements from the given `words` array. We are allowed to use each element only once.

A **palindrome** is defined as a sequence that reads the same both forwards and backwards. For example, "radar" and "level" are palindromes.

We are required to return the length of the longest palindrome that can be created. If no palindromes can be formed using the elements of `words`, the function should return 0.

Intuition

To solve this problem, we need to find pairs of words that can contribute to the palindrome. There are two distinct scenarios to consider:

1. **Matching pairs**: A pair of words where one is the reverse of the other (e.g., "ab" and "ba"). These can be placed symmetrically on either side of the palindrome.
2. **Palindrome pairs**: Words that are palindromes in themselves (e.g., "aa", "bb"). They can be placed in the middle of the palindrome or paired with themselves to be placed symmetrically.

Our strategy is to count the frequency of each word and then iterate over our count dictionary to calculate the contribution of each word to the longest palindrome. For each word, we check:

- If it's a palindrome pair (the word is the same when reversed), we can use it in pairs in the palindrome (e.g., "cc"..."cc" in the middle). The count of these words can only contribute in even numbers for both halves of the palindrome. If there's an odd count, one instance can be placed in the middle.
- If it's a matching pair (word and the reverse are different), we can use as many of these pairs as the minimum count of the word and its reversed pair.

An additional consideration is that we can only place at most one palindrome word in the very center of the palindrome.

The primary steps of the solution include:

- Counting each word's occurrence using a `Counter` structure for efficient lookup.
- Iterating over the unique words and calculating their potential contribution to the palindrome length.
- If possible, adding an extra pair of letters if we have leftover palindrome pairs to maximize the palindrome length.

Based on this logic, the provided solution code computes the longest possible palindrome efficiently.

Solution Approach

The solution employs a `Counter` from Python's collections module to efficiently count the occurrences of each word in the `words` array. A `Counter` is essentially a dictionary where each key is an element from the input array and its corresponding value is the number of times that element appears.

We then iterate through the items in the `Counter` to determine how each word can contribute to the length of the palindrome. The code consists of a few key steps:

- Initialize two accumulators: `ans` represents the length of the potential palindrome, and `x` represents the count of center elements that appear an odd number of times.
- For every unique word `k` and its count `v` in the counter:
  - If the word is a palindrome itself (the same forwards and backwards, i.e., `k[0] == k[1]`):
    - The half-count of such words `v // 2` contributes to both halves of the palindrome. So the contribution would be double the half-count times two `v // 2 * 2 * 2`.
    - If there's an odd count of a palindrome word (`v & 1`), we increment `x`, because it means we have an instance of this word that can potentially be placed in the center of the palindrome.
  - If the word is not a palindrome and has a complementary word that is its reverse (`k[::-1]`) in the array:
    - The contribution of such pairs is limited by the lesser count of the pair, hence we use `min(v, cnt[k[::-1]]) * 2`.
- After iterating through all words, we check if `x` is greater than 0, which would mean that we have at least one word that can be placed at the center. If it's possible, add 2 to `ans` to account for the center element.
- Return `ans` as the length of the longest palindrome.

In summary, by using a counter to track word frequencies and understanding the two types of pairs (matching and palindrome pairs), we can efficiently calculate the maximum possible palindrome length. The solution is quite elegant as it minimizes the amount of work necessary by directly computing the contribution each word can make to the final palindrome structure without having to explicitly build the palindrome.

Example Walkthrough

Let's consider a small example to illustrate the solution approach using an array of strings `words = ["ab", "ba", "aa", "cc", "cc", "aa"]`.

Following the solution approach, we first use a Counter to count occurrences:

- `ab`: 1
- `ba`: 1
- `aa`: 2
- `cc`: 2

Next, we iterate over these counts and calculate how they contribute to the potential palindrome:

- The word "ab" has a count of 1. By checking its reverse, we find "ba" also has a count of 1. They can be used as a pair to contribute 2 to the palindrome length (`"ab" + "ba"` or `"ba" + "ab"` on either side).
- The word "aa" is a palindrome itself and has a count of 2. This means it can contribute twice its half-count (which is `1`) to both halves of the palindrome, adding 4 to the length (`"aa" + "aa"` on either side).
- The word "cc" is also a palindrome and has a count of 2. It contributes another 4 to the length like "aa" (`"cc" + "cc"` on either side).

Since both "aa" and "cc" have even counts, there's no additional contribution to the center, so we have `x = 0`.

Adding together the contributions we get `2 (from "ab" and "ba") + 4 (from "aa") + 4 (from "cc")` equals `10`, which is the length of the longest palindrome that can be created.

Lastly, since there is no extra odd-count palindrome word to place in the center (`x` is 0), we don't add anything more to `ans`.

Therefore, the length of the longest palindrome that can be created from the given `words` array is `10`.

Python Solution

```python
from collections import Counter

class Solution:
    def longestPalindrome(self, words: List[str]) -> int:
        # Create a count of all words
        word_count = Counter(words)
        length_of_palindrome = central_letter_pair_count = 0

        # Counter.items() contains pairs of (word, count)
        for word, count in word_count.items():
            # Check if the word is a pair of identical letters, like "aa"
            if word[0] == word[1]:
                # Count of such word pairs that can be used as the central part of the palindrome
                central_letter_pair_count += count % 2
                # Add the count (rounded down to the nearest even number) times 2 to the length
                length_of_palindrome += (count // 2) * 4
            else:
                # For other pairs, add the minimum count of the pair and its reverse pair to the length
                length_of_palindrome += min(count, word_count[word[::-1]]) * 2

        # If there are pairs of identical letters, we could place exactly one in the center of the palindrome
        length_of_palindrome += 2 if central_letter_pair_count else 0

        return length_of_palindrome

```

Java Solution

```java
class Solution {
    public int longestPalindrome(String[] words) {
        // Create a map to store the frequency of each word
        Map<String, Integer> wordCount = new HashMap<>();
        // Loop through the array of words to populate the map with word counts
        for (String word : words) {
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
        }

        // Initialize variables to store the length of palindrome formed and a flag for center character allowance
        int palindromeLength = 0, singleCenter = 0;

        // Iterate over the map entries to calculate the maximum possible palindrome length
        for (Map.Entry<String, Integer> entry : wordCount.entrySet()) {
            String word = entry.getKey();
            String reversedWord = new StringBuilder(word).reverse().toString();
            int count = entry.getValue();

            // If the word is the same as its reversed version and has two identical characters,
            // it can potentially be a center of the palindrome
            if (word.charAt(0) == word.charAt(1)) {
                // Check if there's an odd count, if so one instance can be used as a center
                singleCenter += count % 2;
                // Add the even part into the palindrome length (doubled, as palindromes are symmetric)
                palindromeLength += (count / 2) * 2 * 2;
            } else {
                // If the word is not its own reversed, pair it with its reverse occurrence count
                palindromeLength += Math.min(count, wordCount.getOrDefault(reversedWord, 0)) * 2;
            }
        }

        // If there is at least one single center word, add 2 to the length for the center character of palindrome
        palindromeLength += singleCenter > 0 ? 2 : 0;

        // Return the total calculated palindrome length
        return palindromeLength;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int longestPalindrome(vector<string>& words) {
        // The 'wordCount' map will store each unique word and its frequency.
        unordered_map<string, int> wordCount;

        // Count the occurrence of each word in the input vector.
        for (auto& word : words) {
            wordCount[word]++;
        }

        // Initialize the length of the longest palindrome to 0 and a variable 'middleInserted' to check for the middle character in
        int longestLength = 0, middleInserted = 0;

        // Traverse each entry in the map to calculate the max length of palindrome.
        for (auto& [key, value] : wordCount) {
            // Create the reverse of the current word.
            string reversedKey = key;
            reverse(reversedKey.begin(), reversedKey.end());

            // Check if the word consists of identical letters (e.g., "gg", "aa").
            if (key[0] == key[1]) {
                // If it's a palindrome by itself, increment 'middleInserted' if an odd count.
                middleInserted += value & 1;

                // Add to the 'longestLength' an even number of this word (pairs).
                longestLength += (value / 2) * 2 * 2; // multiply by 2 - two letters per word, another multiply by 2 - forming pairs.
            } else if (wordCount.count(reversedKey)) {  // If the reversed word is also in the map.
                // Use only the minimum count from the word and its reversed counterpart for a palindrome.
                longestLength += min(value, wordCount[reversedKey]) * 2; // multiply by 2 because each word forms a pair with its rev
            }
        }

        // Adjust the 'longestLength' with 2 extra letters if the middle character can be inserted to make the palindrome longer.
        if (middleInserted > 0) {
            longestLength += 2;
        }

        return longestLength; // Return the longest palindrome length calculated.
    }
};

```

Typescript Solution

```typescript
let wordCount: Record<string, number> = {};

function longestPalindrome(words: string[]): number {
    // Clear the word count for each invocation
    wordCount = {};

    // Count the occurrence of each word in the input array.
    words.forEach(word => {
        wordCount[word] = (wordCount[word] || 0) + 1;
    });

    // Initialize the length of the longest palindrome to 0
    // and a variable to check for the middle character in a palindrome.
    let longestLength: number = 0;
    let middleInserted: number = 0;

    // Traverse each entry in the map to calculate the max length of palindrome.
    Object.entries(wordCount).forEach(([key, value]) => {
        // Create the reverse of the current word.
        const reversedKey: string = key.split('').reverse().join('');

        // Check if the word consists of identical letters (e.g., "gg", "aa").
        if (key[0] === key[1]) {
            // If it's a palindrome by itself, increment 'middleInserted' if an odd count.
            middleInserted += value & 1;

            // Add to the 'longestLength' an even number of this word (pairs).
            longestLength += (value >> 1) * 4; // bitwise shift used for integer division by 2 and multiply by 4 for pairs
        } else if (wordCount[reversedKey] !== undefined) { // If the reversed word also exists.
            // Use only the minimum count from the word and its reversed counterpart for a palindrome.
            longestLength += Math.min(value, wordCount[reversedKey]) * 2;

            // After pairing, remove the entries so they are not re-counted
            wordCount[reversedKey] = 0;
            wordCount[key] = 0;
        }
    });

    // Adjust the 'longestLength' with 2 extra letters if the middle character can be inserted to make the palindrome longer.
    if (middleInserted > 0) {
        longestLength += 2;
    }

    return longestLength; // Return the longest palindrome length calculated.
}

```

Time and Space Complexity

Time Complexity

The time complexity of the code is determined by a few factors:

1. Constructing the `Counter` object from the `words` list. This operation has a time complexity of `O(n)`, where `n` is the number of words in the input list, since it has to count the occurrences of each unique word.
2. Iterating over the items in the counter. In the worst case, each word is unique, so this operation could iterate up to `n` times.
3. For each word, we may check for its reverse in the counter. This is a constant time operation `O(1)` given the nature of hashmaps (dict in Python).

Therefore, the time complexity of the loop is also `O(n)`.

Adding the contributions from these operations, the overall time complexity is `O(n)`. There is no nested iteration or operations that would increase the order of complexity beyond linear.

Space Complexity

The space complexity of the code is primarily affected by the storage requirements of the Counter object:

1. The `Counter` object stores elements as keys and their counts as values. In the worst case, every word in `words` is unique, which means the Counter will store `n` keys and values. Hence, space complexity for the Counter is `O(n)`.
2. The integer variables `ans` and `x` use a constant amount of space, `O(1)`.

So, the overall space complexity of the function is `O(n)` due to the space taken by the Counter object. All other variables only contribute a constant space overhead which does not depend on the input size.

Therefore, the final space complexity is `O(n)`.