Complete Documentation: Palindrome Pairs Problem

Intuition

The problem involves identifying pairs of indices (i, j) such that the concatenation of words[i] and words[j] forms a palindrome. A palindrome reads the same forward and backward. To solve this, we leverage that a palindrome can be constructed by combining words and their reversed forms in specific ways. For instance, the reverse of a word might match the beginning or end of another word, or the word itself could have palindromic prefixes or suffixes.

Key Observations

A few fundamental observations guide the solution:

1. A word and its exact reverse will always form a palindrome (e.g., "abcd" and "dcba").
2. Splitting a word into prefixes and suffixes helps identify where palindromic conditions apply. If one part (prefix or suffix) is a palindrome, the reverse of the other part, if present in the input, forms a valid pair.
3. An empty string is a special case since it can pair with any palindromic word in the list.

Approach

To solve the problem efficiently, we first create a reverse map of the input words, where each reversed word is stored as a key along with its index. This allows quick lookups to check whether the reverse of a substring exists in the input. For every word in the list, we iterate through all possible splits into a prefix and suffix.

We then check if:

1. The prefix is a palindrome and the reverse of the suffix exists in the reverse map.
2. The suffix is a palindrome and the reverse of the prefix exists in the reverse map. These checks ensure that we capture all valid palindrome pairs.

Special Cases

Special cases include handling empty strings and self-palindromic words. When the input contains an empty string, it must be paired with all words that are palindromes themselves, as concatenating these will result in valid palindromes. Additionally, self-palindromic words, like "aba", can pair with an empty string and still form valid results. These edge cases require dedicated handling during iteration.

Efficiency

The algorithm achieves efficiency by using a hash map to store the reversed words, enabling O(1) lookup for reverse matches. Iterating through each word and processing its splits introduces a complexity of $O(n \times L^2)$, where n is the number of words, and LL is the average length of words. This complexity ensures scalability for large inputs with up to 5000 words, each with lengths up to 300, as the constraints require.

Challenges

One challenge is avoiding duplicate pairs. Since palindromes are symmetric, the pair (i, j) might appear multiple times in different forms during the checks. Using a set to store results ensures uniqueness without additional overhead. Another challenge is correctly identifying and handling edge cases like empty strings and overlaps between prefixes and suffixes.

Conclusion

The solution combines efficient data structures (hash maps) with logical checks on palindromic conditions to identify all valid pairs. By focusing on substring splits and reverse lookups, the approach ensures correctness and optimal performance. The algorithm balances clarity and efficiency, making it a robust solution for the given problem.