📓 Concatenated Words - Documentation

1. Problem Statement

Given an array of unique strings words, return all the concatenated words in the list.

A concatenated word is defined as a string that is entirely formed by at least two shorter words from the same list.

Constraints:

- $1 <= words.length <= 10^4$
- $1 <= words[i].length <= 30$
- words[i] consists only of lowercase English letters.
- All strings in words are unique.
- $1 <= sum(words[i].length) <= 10^5$

2. Intuition

We want to find which words in the list can be formed by concatenating at least two other words from the same list.
Since the same word cannot be used in its own construction, we temporarily remove it while checking.
To avoid redundant recursive checks, we use memoization.

3. Key Observations

- Each word should be formed by combining at least two other words from the list.
- We can use recursion to split the word at every possible point and check if both halves are valid.
- Using a set for word lookup makes searching efficient (O(1)).
- Memoization helps avoid recomputing results for the same subwords.

4. **Approach**

- Convert the list of words into a set for constant-time lookups.
- For each word in the list:
    - Temporarily remove the word from the set.
    - Use a recursive helper function canForm(word):
        - Try splitting the word at every index.
        - If the prefix exists in the set and the suffix is in the set or can be formed recursively, mark it as valid.
        - Memoize the result.
    - If valid, add the word to the result.
    - Add the word back to the set.

5. **Edge Cases**

- Words like "a", "b" (length 1) should be skipped — they can't be formed by 2 or more words.
- An empty list should return an empty result.
- Words already in the dictionary should not use themselves to be formed.

6. **Complexity Analysis**

🕒 Time Complexity

- Let N = number of words, L = max length of a word
- Each word may take up to $O(L^2)$ for splitting and recursive checking.
- Total time: $O(N * L^2)$ in the worst case.
- Memoization ensures we don't repeat work for the same subword.

💾 Space Complexity

- O(N + L):
    - O(N) for storing the set.
    - O(L) for the recursion stack and memoization per word.

7. Alternative Approaches

a) Trie-Based Solution

Build a trie of words and recursively check prefixes — slightly more efficient for large datasets.

b) Dynamic Programming

Use DP to check if a word can be segmented like in the "Word Break" problem. Similar logic with a bottom-up approach.

8. Test Cases

✅ Example 1:

Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]

✅ Example 2:

Input: ["cat","dog","catdog"]
Output: ["catdog"]

✅ Edge Case:

Input: ["a","b","ab"]
Output: ["ab"]

9. Final Thoughts

- This problem is a variation of the classic "Word Break" problem with an extra constraint (at least two words).
- Using recursion + memoization is clean and efficient for this size of input.
- Consider Trie or DP if working with longer words or more constraints.
- The approach is scalable and works well due to the optimization through memoization.