Concatenated Words

Given a list of words (**without duplicates**), please write a program that returns all concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example:

```
Input: ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratca
tdogcat"]

Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]

Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";
 "dogcatsdog" can be concatenated by "dog", "cats" and "dog";
"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
```

Note:

1. The number of elements of the given array will not exceed `10,000`
2. The length sum of elements in the given array will not exceed `600,000`.
3. All the input string will only include lower case letters.
4. The returned elements order does not matter.

Solution 1

Do you still remember how did you solve this problem?
https://leetcode.com/problems/word-break/

If you do know one optimized solution for above question is using `DP`, this problem is just one more step further. We iterate through each `word` and see if it can be formed by using other `words`.

Of course it is also obvious that a `word` can only be formed by `words` shorter than it. So we can first sort the input by length of each `word`, and only try to form one `word` by using `words` in front of it.

```
public class Solution {
    public static List<String> findAllConcatenatedWordsInADict(String[] words) {
        List<String> result = new ArrayList<>();
        Set<String> preWords = new HashSet<>();
        Arrays.sort(words, new Comparator<String>() {
            public int compare (String s1, String s2) {
                return s1.length() - s2.length();
            }
        });

        for (int i = 0; i < words.length; i++) {
            if (canForm(words[i], preWords)) {
                result.add(words[i]);
            }
            preWords.add(words[i]);
        }

        return result;
    }

    private static boolean canForm(String word, Set<String> dict) {
        if (dict.isEmpty()) return false;
 boolean[] dp = new boolean[word.length() + 1];
 dp[0] = true;
 for (int i = 1; i <= word.length(); i++) {
     for (int j = 0; j < i; j++) {
  if (!dp[j]) continue;
  if (dict.contains(word.substring(j, i))) {
      dp[i] = true;
      break;
  }
     }
 }
 return dp[word.length()];
    }
}
```

written by shawngao original link here

Solution 2

Most of lines are adding words into Trie Tree
This solution is like putting two pointers to search through the tree. When find a word, put the other pointer back on root then continue searching.
But I'm not sure about the time complexity of my solution. Suppose word length is len and there are n words. Is the time complexity O(len * n ^ 2)?

```
public class Solution {
    class TrieNode {
        TrieNode[] children;
        String word;
        boolean isEnd;
        boolean combo; //if this word is a combination of simple words
        boolean added; //if this word is already added in result
        public TrieNode() {
            this.children = new TrieNode[26];
            this.word = new String();
            this.isEnd = false;
            this.combo = false;
            this.added = false;
        }
    }
    private void addWord(String str) {
        TrieNode node = root;
        for (char ch : str.toCharArray()) {
            if (node.children[ch - 'a'] == null) {
                node.children[ch - 'a'] = new TrieNode();
            }
            node = node.children[ch - 'a'];
        }
        node.isEnd = true;
        node.word = str;
    }
    private TrieNode root;
    private List<String> result;
    public List<String> findAllConcatenatedWordsInADict(String[] words) {
        root = new TrieNode();
        for (String str : words) {
            if (str.length() == 0) {
                continue;
            }
            addWord(str);
        }
        result = new ArrayList<>();
        dfs(root, 0);
        return result;
    }
    private void dfs(TrieNode node, int multi) {
     //multi counts how many single words combined in this word
        if(node.isEnd && !node.added && multi > 1) {
            node.combo = true;
            node.added = true;
            result.add(node.word);
        }
        searchWord(node, root, multi);
```

```
        searchWord(node, root, multi);
    }
    private void searchWord(TrieNode node1, TrieNode node2, int multi) {
        if (node2.combo) {
            return;
        }
        if (node2.isEnd) {
            //take the pointer of node2 back to root
            dfs(node1, multi + 1);
        }
        for (int  i = 0; i < 26; i++) {
            if (node1.children[i] != null && node2.children[i] != null) {
                searchWord(node1.children[i], node2.children[i], multi);
            }
        }
    }
}
```

written by huiyuyang original link here

Solution 3

Let's discuss whether a word should be included in our answer.
Consider the word as a topologically sorted directed graph, where each node is a letter, and an edge exists from i to j if word[i:j] is in our wordlist, [and there is no edge from i=0 to j=len(word)-1]. We want to know if there is a path from beginning to end. If there is, then the word can be broken into concatenated parts from our wordlist. To answer this question, we DFS over this graph.

Code:

```
S = set(A)
ans = []
for word in A:
  if not word: continue
  stack = [0]
  seen = {0}
  M = len(word)
  while stack:
    node = stack.pop()
    if node == M:
      ans.append(word)
      break
    for j in xrange(M - node + 1):
      if (word[node:node+j] in S and
          node + j not in seen and
          (node > 0 or node + j != M)):
        stack.append(node + j)
        seen.add(node + j)

return ans
```

written by awice original link here