<https://leetcode.com/problems/concatenated-words/description/>

Given an array of strings words (**without duplicates**), return *all the* ***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 (not necesssarily distinct) in the given array.

**Example 1:**

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

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".

**Example 2:**

Input: words = ["cat","dog","catdog"]

Output: ["catdog"]

**Constraints:**

1 <= words.length <= 104

1 <= words[i].length <= 30

words[i] consists of only lowercase English letters.

All the strings of words are **unique**.

1 <= sum(words[i].length) <= 105

**Attempt 1: 2023-10-11**

**The L472 just a wrapper for L139, the actual decision logic if a word can be break into two or more other word is exactly same as L139**

**Solution 1: Native DFS (10 min, TLE 42/43)**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

Set<String> set = new HashSet<>(Arrays.asList(words));

for(String word : words) {

// Remove current word from the set because from the

// description, current word must be a concatenation

// of two or more other words in the same given set

set.remove(word);

if(isConcatenatedWord(word, set, 0)) {

result.add(word);

}

// Add back removed current word for next round check

set.add(word);

}

return result;

}

// Exactly same logic as L139 Native DFS

private boolean isConcatenatedWord(String word, Set<String> set, int startIndex) {

if(startIndex == word.length()) {

return true;

}

for(int endIndex = startIndex + 1; endIndex <= word.length(); endIndex++) {

if(set.contains(word.substring(startIndex, endIndex)) && isConcatenatedWord(word, set, endIndex)) {

return true;

}

}

return false;

}

}

Time Complexity: O(N\*2^N)

Space Complexity: O(N\*2^N)

Refer to L139 explain for Time Complexity on Native DFS

**Solution 2: DFS + Memoization (10 min)**

**Style 1: Build static set with default given words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

Set<String> set = new HashSet<>(Arrays.asList(words));

for(String word : words) {

// Remove current word from the set because from the

// description, current word must be a concatenation

// of two or more other words in the same given set

set.remove(word);

Map<Integer, Boolean> memo = new HashMap<>();

if(isConcatenatedWord(word, set, 0, memo)) {

result.add(word);

}

// Add back removed current word for next round check

set.add(word);

}

return result;

}

// Exactly same logic as L139 DFS Memoization

private boolean isConcatenatedWord(String word, Set<String> set, int startIndex, Map<Integer, Boolean> memo) {

if(startIndex == word.length()) {

return true;

}

if(memo.containsKey(startIndex)) {

return memo.get(startIndex);

}

for(int endIndex = startIndex + 1; endIndex <= word.length(); endIndex++) {

if(set.contains(word.substring(startIndex, endIndex)) && isConcatenatedWord(word, set, endIndex, memo)) {

memo.put(startIndex, true);

return true;

}

}

memo.put(startIndex, false);

return false;

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N^4)

In L139 Word Break I, we're making a substring call (which is O(N)) for each possible index (of which there are O(N)), in each DFS call. Since the number of "new"/un-memoized DFS calls is O(N), and each "new"/un-memoized DFS call is doing O(N^2) work, then the runtime of the top down approach is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Style 2: Build dynamic set with sorted given words and only consider all shorter words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

// 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

Arrays.sort(words, (a, b) -> a.length() - b.length());

// The tricky point is we merge 'preWords' build process

// into each round check, that's really a brillant way!

Set<String> preWords = new HashSet<>();

for(String word : words) {

Map<Integer, Boolean> memo = new HashMap<>();

if(isConcatenatedWord(word, preWords, 0, memo)) {

result.add(word);

}

preWords.add(word);

}

return result;

}

// Exactly same logic as L139 DFS Memoization

private boolean isConcatenatedWord(String word, Set<String> set, int startIndex, Map<Integer, Boolean> memo) {

if(startIndex == word.length()) {

return true;

}

if(memo.containsKey(startIndex)) {

return memo.get(startIndex);

}

for(int endIndex = startIndex + 1; endIndex <= word.length(); endIndex++) {

if(set.contains(word.substring(startIndex, endIndex)) && isConcatenatedWord(word, set, endIndex, memo)) {

memo.put(startIndex, true);

return true;

}

}

memo.put(startIndex, false);

return false;

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N^4)

In L139 Word Break I, we're making a substring call (which is O(N)) for each possible index (of which there are O(N)), in each DFS call. Since the number of "new"/un-memoized DFS calls is O(N), and each "new"/un-memoized DFS call is doing O(N^2) work, then the runtime of the top down approach is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Solution 3: DP (10 min)**

**Style 1: Build static set with default given words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

Set<String> set = new HashSet<>(Arrays.asList(words));

for(String word : words) {

// Remove current word from the set because from the

// description, current word must be a concatenation

// of two or more other words in the same given set

set.remove(word);

if(isConcatenatedWord(word, set)) {

result.add(word);

}

// Add back removed current word for next round check

set.add(word);

}

return result;

}

// Exactly same logic as L139 DP

private boolean isConcatenatedWord(String word, Set<String> set) {

int len = word.length();

boolean[] dp = new boolean[len + 1];

dp[len] = true;

for(int i = len - 1; i >= 0; i--) {

for(int j = i + 1; j <= len; j++) {

if(set.contains(word.substring(i, j)) && dp[j]) {

dp[i] = true;

}

}

}

return dp[0];

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N)

In L139 Word Break I, we know DP part is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Style 2: Build dynamic set with sorted given words and only consider all shorter words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

// 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

Arrays.sort(words, (a, b) -> a.length() - b.length());

// The tricky point is we merge 'preWords' build process

// into each round check, that's really a brillant way!

Set<String> preWords = new HashSet<>();

for(String word : words) {

if(isConcatenatedWord(word, preWords)) {

result.add(word);

}

preWords.add(word);

}

return result;

}

// Exactly same logic as L139 DP

private boolean isConcatenatedWord(String word, Set<String> set) {

int len = word.length();

boolean[] dp = new boolean[len + 1];

dp[len] = true;

for(int i = len - 1; i >= 0; i--) {

for(int j = i + 1; j <= len; j++) {

if(set.contains(word.substring(i, j)) && dp[j]) {

dp[i] = true;

}

}

}

return dp[0];

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N)

In L139 Word Break I, we know DP part is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Solution 4: BFS (10 min)**

**Style 1: Build static set with default given words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

Set<String> set = new HashSet<>(Arrays.asList(words));

for(String word : words) {

// Remove current word from the set because from the

// description, current word must be a concatenation

// of two or more other words in the same given set

set.remove(word);

if(isConcatenatedWord(word, set)) {

result.add(word);

}

// Add back removed current word for next round check

set.add(word);

}

return result;

}

// Exactly same logic as L139 BFS

private boolean isConcatenatedWord(String word, Set<String> set) {

Set<Integer> visited = new HashSet<>();

Queue<Integer> q = new LinkedList<>();

q.offer(0);

visited.add(0);

while(!q.isEmpty()) {

int curIdx = q.poll();

for(int i = curIdx + 1; i <= word.length(); i++) {

if(!visited.contains(i) && set.contains(word.substring(curIdx, i))) {

if(i == word.length()) {

return true;

}

q.offer(i);

visited.add(i);

}

}

}

return false;

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N)

In L139 Word Break I, we know BFS part is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Style 2: Build dynamic set with sorted given words and only consider all shorter words**

class Solution {

public List<String> findAllConcatenatedWordsInADict(String[] words) {

List<String> result = new ArrayList<>();

// 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

Arrays.sort(words, (a, b) -> a.length() - b.length());

// The tricky point is we merge 'preWords' build process

// into each round check, that's really a brillant way!

Set<String> preWords = new HashSet<>();

for(String word : words) {

if(isConcatenatedWord(word, preWords)) {

result.add(word);

}

preWords.add(word);

}

return result;

}

// Exactly same logic as L139 BFS

private boolean isConcatenatedWord(String word, Set<String> set) {

Set<Integer> visited = new HashSet<>();

Queue<Integer> q = new LinkedList<>();

q.offer(0);

visited.add(0);

while(!q.isEmpty()) {

int curIdx = q.poll();

for(int i = curIdx + 1; i <= word.length(); i++) {

if(!visited.contains(i) && set.contains(word.substring(curIdx, i))) {

if(i == word.length()) {

return true;

}

q.offer(i);

visited.add(i);

}

}

}

return false;

}

}

Time Complexity: O(N^4) -> additional N for Java substring

Space Complexity: O(N)

In L139 Word Break I, we know BFS part is O(N^3), and the because we have a for loop outside in L472, so total runtime is O(N^4)

**Optimize by sorting ?**

<https://leetcode.com/problems/concatenated-words/solutions/95652/java-dp-solution/>

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()];

}

}

**In L472 DFS better than DP ?**

**Refer to**

<https://leetcode.com/problems/word-break/solutions/43886/evolve-from-brute-force-to-optimal-a-review-of-all-solutions/>

**For dp problems, many times we go into iterative dp directly without even thinking about dfs. This is a great example showing that dfs is better than dp. DFS returns as soon as it finds one way to break the word while dp computes if each substring starting/ending at i is breakable. The test cases of this problem do not show it but it is shown in a similar problem [Concatenated Words](https://discuss.leetcode.com/topic/78569/a-review-of-top-solutions).**

<https://leetcode.com/problems/concatenated-words/solutions/95670/a-review-of-top-solutions/>

**One sub problem is word break. There are three ways to solve word break, [DFS, BFS and DP](https://discuss.leetcode.com/topic/66356/evolve-from-brute-force-to-optimal). DFS is the best way because it terminates as soon as it finds one way to break the word. DP and BFS checks all the substrings thus less efficient. Combined with the sorting idea by [@shawngao](https://discuss.leetcode.com/topic/72113/java-dp-solution) is the best I have.**