<https://leetcode.com/problems/word-break-ii/>

Given a string s and a dictionary of strings wordDict, add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]

Output: ["cats and dog","cat sand dog"]

**Example 2:**

Input: s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]

Output: ["pine apple pen apple","pineapple pen apple","pine applepen apple"]

Explanation: Note that you are allowed to reuse a dictionary word.

**Example 3:**

Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

Output: []

**Constraints:**

* 1 <= s.length <= 20
* 1 <= wordDict.length <= 1000
* 1 <= wordDict[i].length <= 10
* s and wordDict[i] consist of only lowercase English letters.
* All the strings of wordDict are **unique**.

**Attempt 1: 2022-12-04**

**Solution 1:  DFS Memoization (120 min)**

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

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

return helper(s, 0, wordDict, memo);

}

private List<String> helper(String s, int start, List<String> wordDict, Map<Integer, List<String>> memo) {

// Memo quick return

if(memo.containsKey(start)) {

return memo.get(start);

}

// Base case

// e.g for given string as "catsanddog", when start = 10,

// rest.add("") is prepare for concatenation "dog" ahead,

// "dog" + "[empty placeholder]" = "dog[empty placeholder]"

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

if(start == s.length()) {

result.add("");

}

for(int end = start + 1; end <= s.length(); end++) {

if(wordDict.contains(s.substring(start, end))) {

List<String> list = helper(s, end, wordDict, memo);

// The 'list' here will only have one element inside, which is

// the pending construction string (initial as empty string ""),

// so the loop will happen only once, but in case to concatenate

// new term onto existing string, we have to identify if existing

// string is the empty string which we deliberately setup to mark

// you reach the end of string or its not empty string anymore since

// you concatenate term(s) on that pending construction string already

for(String temp : list) {

result.add(s.substring(start, end) + (temp.equals("") ? "" : " ") + temp);

}

}

}

memo.put(start, result);

return result;

}

}

Time Complexity: O(n^2) ~ O(2^n)

In the worst case the runtime of this algorithm is O(2^n).

Consider the input "aaaaaa", with wordDict = ["a", "aa", "aaa", "aaaa", "aaaaa", "aaaaa"]. Every possible partition is a valid sentence, and there are 2^n-1 such partitions. It should be clear that the algorithm cannot do better than this since it generates all valid sentences. The cost of iterating over cached results will be exponential, as every possible partition will be cached, resulting in the same runtime as regular backtracking. Likewise, the space complexity will also be O(2^n) for the same reason - every partition is stored in memory.

Where this algorithm improves on regular backtracking is in a case like this: "aaaaab", with wordDict = ["a", "aa", "aaa", "aaaa", "aaaaa", "aaaaa"], i.e. the worst case scenario for Word Break I, where no partition is valid due to the last letter 'b'. In this case there are no cached results, and the runtime improves from O(2^n) to O(n^2).

**Test case:**

**s = "catsanddog", dict = ["cat", "cats", "and", "sand", "dog"]**

Analysis

c a t s a n d d o g

0 1 2 3 4 5 6 7 8 9

start:0 map:empty

end:3 -> dfs

start:3 map:empty

end:7 -> dfs

start:7 map:empty

end:10 -> dfs

start:10 == s.length()

rest.add("")

map.put(10, "")

return rest

------------------------------------------------

start:7 map:{10=[]} list = dfs(s, wordDict, 10) => ""

end:10 rest.add("dog")

map.put(7,"dog")

return rest

start:3 map:{7=[dog],10=[]} list = dfs(s, wordDict, 7) => "dog"

end:7 rest.add("sand dog")

map.put(3,"sand dog")

return rest

start:0 map:{3=[sand dog],7=[dog],10=[]} list = dfs(s, wordDict, 3) => "sand dog"

end:3 rest.add("cat sand dog") --> 1st solution

map.put(0,"cat sand dog")

return rest

---------------------------------------------

start:0 map:{3=[sand dog],7=[dog],10=[]}

end:4 -> dfs

start:4 map:{3=[sand dog],7=[dog],10=[]}

end:7 -> dfs

start:7

end:10 -> dfs -> return map.get(7) -> [dog] --> DP quick return

---------------------------------------------

start:4 map:{3=[sand dog],7=[dog],10=[]}

end:7 rest.add("and" + " " + "dog")

map.put(4, "and dog")

return rest

start:0 map:{3=[sand dog], 4=[and dog], 7=[dog], 10=[]}

end:4 rest.add("cats" + " " + "and dog") -> 2nd solution

map.put(0, "cats and dog")

return rest

--------------------------------------------

finally map:{0=[cat sand dog, cats and dog], 3=[sand dog], 4=[and dog], 7=[dog], 10=[]}

rest:[cat sand dog, cats and dog]

**Refer to**

<https://leetcode.com/problems/word-break-ii/solutions/44167/my-concise-java-solution-based-on-memorized-dfs/?orderBy=most_votes>

**Explanation**

Using DFS directly will lead to TLE, so I just used HashMap to save the previous results to prune duplicated branches, as the following:

public List<String> wordBreak(String s, Set<String> wordDict) {

return DFS(s, wordDict, new HashMap<String, LinkedList<String>>());

}

// DFS function returns an array including all substrings derived from s.

List<String> DFS(String s, Set<String> wordDict, HashMap<String, LinkedList<String>>map) {

if (map.containsKey(s))

return map.get(s);

LinkedList<String>res = new LinkedList<String>();

if (s.length() == 0) {

res.add("");

return res;

}

for (String word : wordDict) {

if (s.startsWith(word)) {

List<String>sublist = DFS(s.substring(word.length()), wordDict, map);

for (String sub : sublist)

res.add(word + (sub.isEmpty() ? "" : " ") + sub);

}

}

map.put(s, res);

return res;

}

**Solution 2:  More intuitive DFS Memoization (10 min)**

class Solution {

public List<String> wordBreak(String s, List<String> wordDict) {

Map<String, List<String>> memo = new HashMap<String, List<String>>();

return helper(s, wordDict, memo);

}

// dfs returns an array including all substrings derived from s.

private List<String> helper(String s, List<String> wordDict, Map<String, List<String>> memo) {

if(memo.containsKey(s)) {

return memo.get(s);

}

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

for(String word : wordDict) {

if(s.startsWith(word)) {

String remain = s.substring(word.length());

if(remain.length() == 0) {

result.add(word);

} else {

List<String> candidates = helper(remain, wordDict, memo);

for(String candidate : candidates) {

result.add(word + " " + candidate);

}

}

}

}

memo.put(s, result);

return result;

}

}

**Refer to**

<https://leetcode.com/problems/word-break-ii/solutions/44167/my-concise-java-solution-based-on-memorized-dfs/comments/43403>

public List<String> wordBreak(String s, List<String> wordDict) {

return backtrack(s,wordDict,new HashMap<String, List<String>>());

}

// backtrack returns an array including all substrings derived from s.

public List<String> backtrack(String s, List<String> wordDict, Map<String,List<String>> mem){

if(mem.containsKey(s)) return mem.get(s);

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

for(String word: wordDict)

if(s.startsWith(word)) {

String next = s.substring(word.length());

if(next.length()==0) result.add(word);

else for(String sub: backtrack(next, wordDict, mem)) result.add(word+" "+sub);

}

mem.put(s, result);

return result;

}