139. Word Break

Medium

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

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

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]

Output: true

Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]

Output: true

Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".

Note that you are allowed to reuse a dictionary word.
```

Example 3:

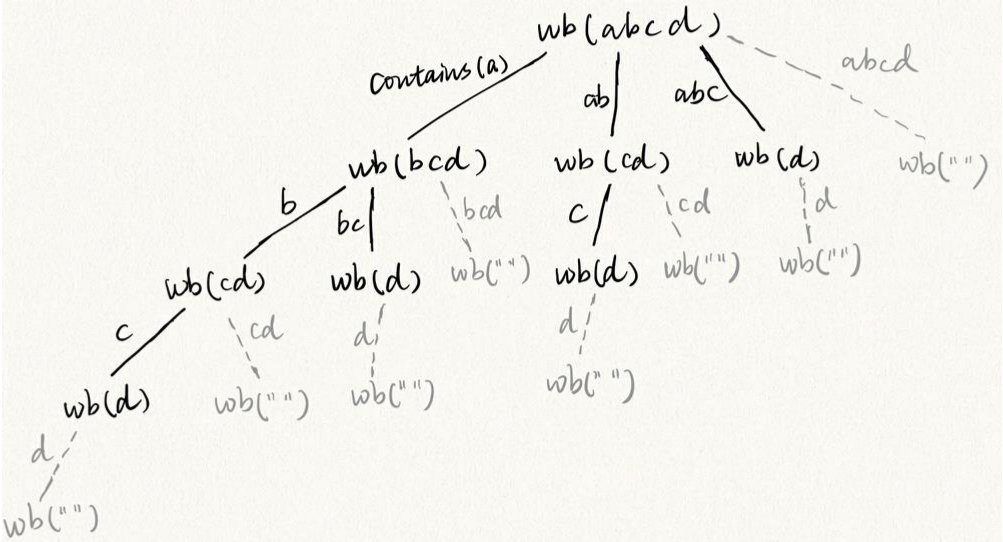
```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]

Output: false
```

Constraints:

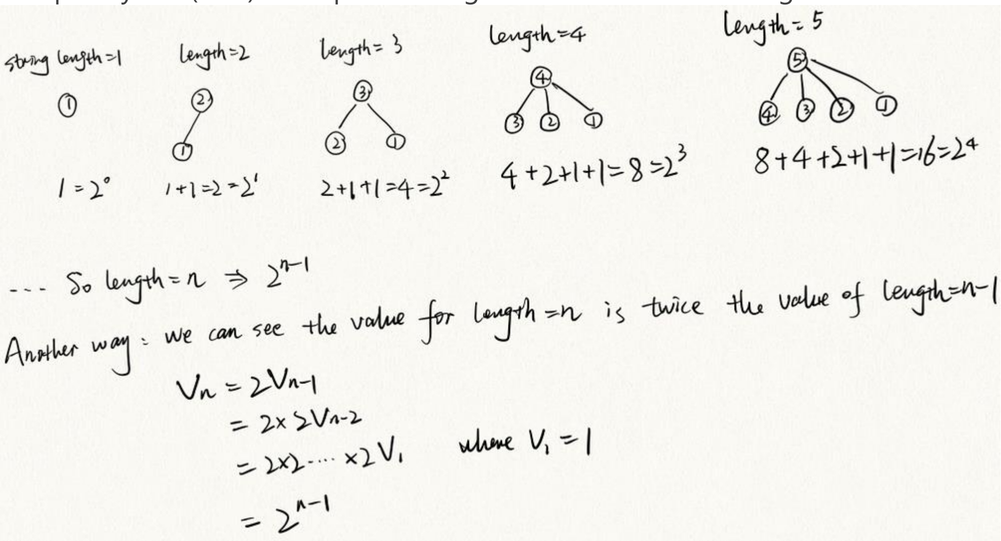
- `1 <= s.length <= 300`
- `1 <= wordDict.length <= 1000`
- `1 <= wordDict[i].length <= 20`
- `s` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are **unique**.

Approach 1: recursion

The time complexity depends on how many nodes the recursion tree has. In the worst case, the recursion tree has the most nodes, which means the program should not return in the middle and it should try as many possibilities as possible. So the branches and depth of the tree are as many as possible. For the worst case, for example, we take `s = "abcd"` and `wordDict = ["a", "b", "c", "bc", "ab", "abc"]`, the recursion tree is shown below:



From the code `if (set.contains(s.substring(0, i)) && wb(s.substring(i), set)) { }`, we can see that only if the wordDict contains the prefix, the recursion function can go down to the next level. So on the figure above, string on the edge means the wordDict contains that string. All the gray node with empty string cannot be reached because if the program reaches one such node, the program will return, which lead to some nodes right to it will not be reached. So the conclusion is for a string with length 4, the recursion tree has 8 nodes (all black nodes), and 8 is 2^(4-1). So to generalize this, for a string with length n, the recursion tree wil have 2^(n-1) nodes, i.e., the time complexity is O(2^n). I will prove this generalization below using mathmatical induction:



... So length=n ⇒ 2^{n-1}

Another way: we can see the value for length=n is twice the value of length=n-1

$$V_n = 2V_{n-1}$$
$$= 2\times 2V_{n-2}$$
$$= 2\times 2 \cdots \times 2V_1 \quad \text{where } V_1 = 1$$
$$= 2^{n-1}$$

Explanation: the value of a node is the string length. We calculate the number of nodes in the recursion tree for string length=1, 2, ..., n respectively.

For example, when string length=4, the second layer of the recursion tree has three nodes where the string length is 3, 2 and 1 respectively. And the number of subtree rooted at these three nodes have been calculated when we do the mathmatical induction.

So time complexity is O(2^n)

```
class Solution {

    public boolean wordBreak(String s, List<String> wordDict) {
       return helper(0,s,new HashSet<String>(wordDict));
    }
    public boolean helper(int ind,String s,HashSet<String> wordDict)
    {
         if(s.length()==0)
            return true;
        for(int i=ind;i<s.length();i++)

        {
            if(wordDict.contains(s.substring(ind,i+1)))
            {
                if(helper(i+1,s,wordDict))
                {
                    return true;
                }
            }
        }
        return false;
    }
}
```

Approach 2:

Dynamic programming

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
```

```
    Boolean[] map=new Boolean[s.length()+1];
   return helper(0,s,new HashSet<String>(wordDict),map);


}
public boolean helper(int ind,String s,HashSet<String> wordDict,Boolean[] map)
{
     if(s.length()==ind)
        return true;
    if(map[ind]!=null)
        return map[ind];
    String temp="";
    for(int i=ind;i<s.length();i++)

    {
        temp+=s.charAt(i);
        if(wordDict.contains(temp))
        {
            if(helper(i+1,s,wordDict,map))
            {
                map[ind]=true;
                return true;
            }
        }
    }
```

```
        map[ind]=false;

        return false;

    }

}
```