给你一个字符串 `s` 和一个字符串列表 `wordDict` 作为字典。请你判断是否可以利用字典中出现的单词拼接出 `s` 。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

示例 1：

```
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以由 "leet" 和 "code" 拼接成。
```

示例 2：

```
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以由 "apple" "pen" "apple" 拼接成。
     注意，你可以重复使用字典中的单词。
```

示例 3：

```
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

我们定义 dp[i]\textit{dp}[i]dp[i] 表示字符串 sss 前 iii 个字符组成的字符串 s[0..i−1]s[0..i-1]s[0..i−1] 是否能被空格拆分成若干个字典中出现的单词。从前往后计算考虑转移方程，每次转移的时候我们需要枚举包含位置 i−1i-1i−1 的最后一个单词，看它是否出现在字典中以及除去这部分的字符串是否合法即可。公式化来说，我们需要枚举 s[0..i−1]s[0..i-1]s[0..i−1] 中的分割点 jjj ，看 s[0..j−1]s[0..j-1]s[0..j−1] 组成的字符串 s1s_1s
1

（默认 j=0j = 0j=0 时 s1s_1s
1

为空串）和 s[j..i−1]s[j..i-1]s[j..i−1] 组成的字符串 s2s_2s
2

是否都合法，如果两个字符串均合法，那么按照定义 s1s_1s
1

和 s2s_2s
2

拼接成的字符串也同样合法。由于计算到 dp[i]\textit{dp}[i]dp[i] 时我们已经计算出了 dp[0..i−1]\textit{dp}[0..i-1]dp[0..i−1] 的值，因此字符串 s1s_1s
1

是否合法可以直接由 dp[j]dp[j]dp[j] 得知，剩下的我们只需要看 s2s_2s
2

是否合法即可，因此我们可以得出如下转移方程：

dp[i]=dp[j] && check(s[j..i−1]) \textit{dp}[i]=\textit{dp}[j]\ &&\ \textit{check}(s[j..i-1])
dp[i]=dp[j] && check(s[j..i−1])
其中 check(s[j..i−1])\textit{check}(s[j..i-1])check(s[j..i−1]) 表示子串 s[j..i−1]s[j..i-1]s[j..i−1] 是否出现在字典中。

对于检查一个字符串是否出现在给定的字符串列表里一般可以考虑哈希表来快速判断，同时也可以做一些简单的剪枝，枚举分割点的时候倒着枚举，如果分割点 jjj 到 iii 的长度已经大于字典列表里最长的单词的长度，那么就结束枚举，但是需要注意的是下面的代码给出的是不带剪枝的写法。

对于边界条件，我们定义 dp[0]=true\textit{dp}[0]=truedp[0]=true 表示空串且合法。

有能力的读者也可以考虑怎么结合字典树 Trie\text{Trie}Trie 来实现，这里不再展开。

Java
TypeScript
Golang
C
C#
C++
```
public class Solution {
    public boolean wordBreak(String s, List wordDict) {
        Set wordDictSet = new HashSet(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordDictSet.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
}
```
复杂度分析

时间复杂度：O(n2)O(n^2)O(n
2
)，其中 nnn 为字符串 sss 的长度。我们一共有 O(n)O(n)O(n) 个状态需要计算，每次计算需要枚举 O(n)O(n)O(n) 个分割点，哈希表判断一个字符串是否出现在给定的字符串列表需要 O(1)O(1)O(1) 的时间，因此总时间复杂度为 O(n2)O(n^2)O(n
2
)。

空间复杂度：O(n)O(n)O(n) ，其中 nnn 为字符串 sss 的长度。我们需要 O(n)O(n)O(n) 的空间存放 dp\textit{dp}dp 值以及哈希表亦需要 O(n)O(n)O(n) 的空间复杂度，因此总空间复杂度为 O(n)O(n)O(n)。