# 题目

给你一个字符串 s 和一个字符串列表 wordDict 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 s 则返回 true。

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

提示：

1 <= s.length <= 300

1 <= wordDict.length <= 1000

1 <= wordDict[i].length <= 20

s 和 wordDict[i] 仅由小写英文字母组成

wordDict 中的所有字符串 互不相同

# 分析

## 方法一：动态规划

要解决“判断字符串是否可由字典中单词拼接而成”的问题，核心思路是动态规划，通过记录字符串前i个字符能否被字典中的单词拼接，逐步推导出整个字符串是否可拼接。

解题思路

1、动态规划状态定义：

定义dp[i]表示字符串s的前i个字符（即s[0..i-1]）能否被字典中的单词拼接而成。最终目标是dp[s.length()]。

2、初始化：

dp[0] = true，表示空字符串可以被成功拼接（作为递归的起始条件）。

3、状态转移：

对于每个位置i（从1到s.length()），检查所有可能的起始位置j（从0到i-1）：

- 若dp[j]为true（前j个字符可拼接），且子串s[j..i-1]存在于字典中，则dp[i]为true。

4、优化查询效率：

将字典中的单词存入哈希集合（unordered\_set），使子串是否存在的查询时间复杂度降至O(1)。

代码：

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict) {

// 将字典存入哈希集合，优化查询效率

unordered\_set<string> wordSet(wordDict.begin(), wordDict.end());

int n = s.length();

// dp[i]表示s的前i个字符能否被拼接

vector<bool> dp(n + 1, false);

dp[0] = true; // 空字符串可拼接

for (int i = 1; i <= n; ++i) {

// 检查所有可能的起始位置j

for (int j = 0; j < i; ++j) {

// 若前j个字符可拼接，且s[j..i-1]在字典中，则前i个字符可拼接

if (dp[j] && wordSet.count(s.substr(j, i - j))) {

dp[i] = true;

break; // 找到一个有效拼接即可，无需继续检查

}

}

}

return dp[n];

}

};

代码解释

1、哈希集合存储字典：

将wordDict转换为unordered\_set，使得判断子串是否在字典中的操作从O(m)（m为字典长度）优化为O(1)，提升整体效率。

2、动态规划数组初始化：

dp[0] = true是关键的起始条件，代表空字符串可以被成功拼接，为后续的状态转移提供基础。

3、双重循环实现状态转移：

- 外层循环i遍历字符串的每个位置（从1到n），表示当前要判断的前i个字符。

- 内层循环`j`遍历所有可能的起始位置（从0到i-1），检查前j个字符能否拼接，且j到i之间的子串是否在字典中。

- 一旦找到有效的j，立即将dp[i]设为true并跳出内层循环，减少冗余计算。

4、结果返回：

dp[n]表示整个字符串s能否被字典中的单词拼接，直接返回该值。

复杂度分析

- 时间复杂度：O(n²)，其中n是字符串s的长度。外层循环O(n)，内层循环O(n)，每次子串查询O(k)（k为子串长度，最坏O(n)），整体最坏为O(n³)，但实际中因哈希集合查询和提前break会更优。

- 空间复杂度：O(n + m)，n为dp数组大小，m为字典中单词的总长度（哈希集合存储所需）。

## 方法二：动态规划

思路：

可以使用动态规划的方法来解决。我们可以定义一个布尔类型的数组dp，其中dp[i]表示字符串s的前i个字符是否可以被字典中的单词拼接而成。

首先，我们需要初始化dp数组。显然，空字符串可以被视为由空字符拼接而成，所以dp[0]应该为true。

然后，我们遍历字符串s的每个位置i，从1到s.length()。对于每个位置 i，我们再遍历字典wordDict中的每个单词word，检查是否可以将word与s的前i - word.length()个字符拼接成s的前i个字符。如果可以，那么dp[i]就应该为true。

最后，返回dp[s.length()]，即表示整个字符串 s 是否可以被字典中的单词拼接而成。

代码：

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict) {

int n = s.size();

vector<bool> dp(n + 1, false);

dp[0] = true; // 空字符串可以被视为由空字符拼接而成

// 将字典中的单词存入哈希集合中，方便快速查找

unordered\_set<string> wordSet(wordDict.begin(), wordDict.end());

for (int i = 1; i <= n; i++) {

for (int j = 0; j < i; j++) {

// 检查是否可以将 s[j:i-1] 与 s[0:j-1] 拼接成 s[0:i-1]

string word = s.substr(j, i - j);

if (dp[j] && wordSet.count(word)) {

dp[i] = true;

break;

}

}

}

return dp[n];

}

};

这段代码首先定义了一个布尔类型的数组 dp，并初始化了 dp[0] 为 true。然后，它将字典 wordDict 中的单词存入一个哈希集合 wordSet 中，以便快速查找。接下来，它使用两个嵌套的循环来遍历字符串 s 的每个位置 i 和每个可能的分割点 j。如果 s[j:i-1] 是一个字典中的单词，并且 s[0:j-1] 可以被字典中的单词拼接而成（即 dp[j] 为 true），那么 s[0:i-1] 也可以被字典中的单词拼接而成，因此将 dp[i] 设为 true。最后，返回 dp[n]，即整个字符串 s 是否可以被字典中的单词拼接而成。