

# 139. Word Break

Medium Trie Memorization Array Hash Table String Dynamic Programming

Leetcode Link

## Problem Description

The LeetCode problem asks us to determine if a given string `s` can be split into a sequence of one or more words that exist in a given dictionary `wordDict`. To put it simply, the problem is asking whether we can break the string `s` into chunks where each chunk matches a word in the provided list of words (which we call a dictionary). A word from the dictionary can be used multiple times if needed.

For example, if `s` is "leetcode" and `wordDict` contains "leet" and "code", then the answer is `true` since "leetcode" can be segmented into "leet code".

## Intuition

The intuition behind the solution is to use Dynamic Programming (DP), which is a method for solving complex problems by breaking them down into simpler subproblems. The idea here is that if we can break the string `s` up to a given point, then we can independently check the remainder of the string for other words from `wordDict`. We can cache results to avoid redundant computations for the same substrings.

One way to visualize this solution is by thinking of a chain. If breaking the chain at a given link (character of string `s`) gives us two parts, where the left part has been seen before and is confirmed to be composed of words in `wordDict`, and the right part is a known dictionary word, then the entire chain up to that point can be composed of dictionary words.

In the solution provided, the list `f` is used to store the results of the subproblems. Here's what it represents:

- `f[0]` is `True` because a string with no characters can trivially be segmented (it's an empty sequence, which is considered a valid segmentation).
- `f[i]` for `i>0` is `True` if and only if the string up to the `i`-th character can be segmented into one or more of the dictionary words. So `f[i]` is set to `True` if, for any `j` where `0 <= j < i`, `f[j]` is `True` and the substring `s[j:i]` is in `wordDict`.

The algorithm loops over the length of the string, checking at each character if there is a valid word ending at this character and, simultaneously, if the string before this word can be split into valid words. If both these conditions are ever fulfilled at some index `i`, we set `f[i]` to `True`.

By the time we reach the end of the string if `f[n]` (where `n` is the length of the string) is `True`, we know the entire string can be split up according to the `wordDict`. Otherwise, `f[n]` will be `False`, indicating that the string `s` cannot be segmented into a sequence of one or more dictionary words.

## Solution Approach

The solution utilizes dynamic programming, which involves solving smaller subproblems and using their solutions to effectively solve larger problems. In this context, the subproblems are answering the question: "Can the string up to the `i`-th character be segmented into a sequence of dictionary words?"

Let's delve into the algorithm and the data structures used:

- We start by initializing a set `words` from `wordDict` for fast lookups of words in the dictionary.
- An array `f` is created with a size of `n+1` where `n` is the length of the string `s`. The array `f` is initialized to all `False` except for `f[0]` which is `True`. This represents that it's always possible to segment an empty string.
- We then iterate over the string `s` from the first character to the last, checking at each point whether the string up to that point can be segmented. This is achieved by using a nested loop, where the outer loop iterates from 1 to `n` and represents the end of the current substring being checked (`i`), and the inner loop (`j`) iterates from 0 to `i-1` and represents different start points of the substring.
- At each position `i`, we check all possible substrings `s[j:i]` where `j` ranges from 0 up to `i-1`. For a substring `s[j:i]` to be valid, two conditions must be met:
  - The substring `s[j:i]` must be in `words`.
  - The `f[j]` entry must be `True`, signifying that the string can be segmented up to the `j`-th character.
- If both conditions are satisfied for any `j`, we set `f[i]` to `True` because we've found that the string can be segmented up to the `i`-th character.
- Finally, `f[n]` indicates whether the entire string can be segmented. If it's `True`, then the string can be split into a valid sequence of dictionary words, and we return `True`. If it's `False`, then there is no way to split the entire string into valid dictionary words, and we return `False`.

In summary, this problem is efficiently solved by breaking it down into smaller parts, solving each part individually, and then combining those solutions to find the answer to the original problem. The `any()` function in Python provides a succinct way to check whether any substring `s[j:i]` fulfills our two conditions, making the code more concise and readable.

## Example Walkthrough

Let's assume `s = "applepenapple"` and `wordDict = ["apple", "pen"]`. We want to check if `s` can be split into words contained in `wordDict`.

- Initialize the set `words` from `wordDict` for quick lookups:

```
1 words = {"apple", "pen"}
```
- Initialize the array `f` with `False` and set `f[0] = True` (an empty string can always be segmented):

```
1 f = [False] * (len(s) + 1)
2 f[0] = True
```
- Then we iterate over the length of the string `s` from 1 through `n`, which is 15 in this case for "applepenapple".
- For each `i` from 1 to 15, we will check all substrings `s[j:i]` for `j` from 0 up to `i-1`.
- Consider the case when `i = 5`, we are looking at the substring "apple":
  - Is "apple" (`s[0:5]`) in `words`? Yes, it is.
  - Is `f[0]` True? Yes, because we initialized it as such, indicating that up to the previous characters (none in this case), the string can be segmented.
  - Therefore, `f[5]` becomes `True`.
- Move to `i = 8`, the substring is "pen" (`s[5:8]`):
  - Is "pen" in `words`? Yes, it is.
  - Is `f[5]` True? Yes, because we know "apple" can be segmented till `i = 5`.
  - We set `f[8]` to `True`.
- At `i = 15`, our substring is "apple" again (`s[8:15]`):
  - Is "apple" in `words`? Yes.
  - Is `f[8]` True? Yes, due to previous segments.
  - `f[15]` is set to `True`.
- Finally, we check `f[n]`, which is `f[15]` in this case. It is `True`, which means "applepenapple" can be segmented as "apple pen apple", all words from `wordDict`.

By following these steps, we have broken down the problem to show how `s` can be segmented using `wordDict`, illustrating the dynamic programming approach described in the solution.

## Python Solution

```
1 class Solution:
2     def wordBreak(self, s: str, word_set: List[str]) -> bool:
3         # Initialize a set with dictionary words for quicker look-up
4         words = set(word_set)
5
6         # Get the length of the string to process
7         length_of_s = len(s)
8
9         # Initialize a list where f[i] represents whether s[:i] can be segmented into words in the dictionary
10        can_break = [True] + [False] * length_of_s
11
12        # Iterate over all the positions of the string
13        for i in range(1, length_of_s + 1):
14            # Determine if there is a j where s[:j] can be segmented and s[j:i] is in the dictionary
15            can_break[i] = any(can_break[j] and s[j:i] in words for j in range(i))
16
17        # Return the value for the whole string, whether it can be segmented or not
18        return can_break[length_of_s]
19
```

## Java Solution

```
1 class Solution {
2     public boolean wordBreak(String s, List<String> wordDict) {
3         // Convert the list of words into a hash set for efficient look-up
4         Set<String> wordSet = new HashSet<>(wordDict);
5
6         // Get the length of the string 's'
7         int strLength = s.length();
8
9         // Initialize a boolean array to keep track of possible word breaks
10        // f[i] is true if first i characters of the string can be segmented into dictionary words
11        boolean[] wordBreakTracker = new boolean[strLength + 1];
12
13        // Empty string is a valid decomposition
14        wordBreakTracker[0] = true;
15
16        // Check each substring starting from length 1 to strLength
17        for (int i = 1; i <= strLength; ++i) {
18
19            // Try different break points
20            for (int j = 0; j < i; ++j) {
21
22                // If the string up to j can be broken into valid words, and the substring from j to i is in the dictionary
23                // Then, mark the position i as true in wordBreakTracker
24                if (wordBreakTracker[j] && wordSet.contains(s.substring(j, i))) {
25                    wordBreakTracker[i] = true;
26
27                    // Break out of the loop since we have found a valid break point
28                    break;
29                }
30            }
31
32            // The last entry in wordBreakTracker tells if the entire string can be segmented or not
33            return wordBreakTracker[strLength];
34        }
35    }
36 }
37
```

## C++ Solution

```
1 #include <string>
2 #include <vector>
3 #include <unordered_set>
4 #include <string>
5
6 class Solution {
7 public:
8     // Function to determine if the string s can be segmented into space-separated
9     // sequence of one or more dictionary words.
10    bool wordBreak(string s, vector<string>& wordDict) {
11        // Create a set containing all the words in the dictionary for constant time look-up.
12        unordered_set<string> word_set(wordDict.begin(), wordDict.end());
13
14        // Get the length of the string.
15        int strLength = s.size();
16
17        // Array to hold the status of substrings, f[i] is true if s[0..i-1] can be segmented.
18        bool canSegment[strLength + 1];
19
20        // Initialize the canSegment array with false values.
21        memset(canSegment, false, sizeof(canSegment));
22
23        // Empty string is always a valid segmentation.
24        canSegment[0] = true;
25
26        // Start checking for all substrings starting from the first character.
27        for (int i = 1; i <= strLength; ++i) {
28            // Check all possible sub-strings ending at position i.
29            for (int j = 0; j < i; ++j) {
30                // If s[0..j-1] can be segmented and s[j..i-1] is in the word set,
31                // then set canSegment[i] to true.
32                if (canSegment[j] && word_set.count(s.substr(j, i - j))) {
33                    canSegment[i] = true;
34                    // Break because we found a valid segmentation until i.
35                    break;
36                }
37            }
38
39            // Return true if the whole string can be segmented, otherwise false.
40            return canSegment[strLength];
41        }
42    }
43 };
44
```

## Typescript Solution

```
1 function wordBreak(s: string, wordDict: string[]): boolean {
2     const wordSet = new Set(wordDict); // Create a Set from the wordDict for efficient look-up.
3     const stringLength = s.length; // Store the length of the input string.
4     // Initialize an array 'canBreak' to keep track of whether the string can be segmented up to each index.
5     const canBreak: boolean[] = new Array(stringLength + 1).fill(false);
6     canBreak[0] = true; // The string can always be segmented at index 0 (empty string).
7
8     // Iterate over the string.
9     for (let endIndex = 1; endIndex <= stringLength; ++endIndex) {
10        // Check for every possible partition position 'startIndex'.
11        for (let startIndex = 0; startIndex < endIndex; ++startIndex) {
12            // If the string can be segmented up to 'startIndex' and the substring from 'startIndex' to 'endIndex'
13            // is in the word dictionary, update 'canBreak' for 'endIndex' and break out of the loop.
14            if (canBreak[startIndex] && wordSet.has(s.substring(startIndex, endIndex))) {
15                canBreak[endIndex] = true;
16                break;
17            }
18        }
19    }
20    // Return whether the string can be segmented up to its entire length.
21    return canBreak[stringLength];
22 }
23
```

## Time and Space Complexity

The given code defines a method `wordBreak` that checks whether you can segment a string `s` into a space-separated sequence of one or more dictionary words from `wordDict`.

### Time Complexity

To calculate the time complexity, we need to consider the operations performed within the loop:

- We iterate over the length of the string `s` which gives us an  $O(n)$  where `n` is the length of the input string.
- Inside each iteration, the `any()` function is called which in worst-case will iterate over `i` elements.
- For each of these elements, we check if `f[j]` is `True` (constant time) and `s[j:i]` in `words` which is  $O(i-j)$  in the worst case since looking up in a set is  $O(1)$  but slicing the string is  $O(i-j)$ .

The worst-case time complexity is when `s[j:i]` is a word for all `j` and `i`, so we have to check every possible substring. Thus, the time complexity becomes  $O(n^2 * k)$  where `k` is the maximum length of the word because for each `i`, we perform `i` checks and each check could take up to `k` time due to string slicing.

### Space Complexity

The space complexity can be analyzed by looking at the storage used:

- The set `words` is  $O(m)$  where `m` is the total number of characters across all words in `wordDict`.
- The list `f` is  $O(n+1)$  which simplifies to  $O(n)$  where `n` is the length of the input string `s`.

Therefore, the overall space complexity is  $O(n + m)$ .

Note: Due to Python's internal implementation of string slicing, it can be argued that string slicing is done in  $O(1)$  under normal circumstances due to string interning and slicing just pointing to a subpart of the existing string without actually creating a new one. However, to be safe, especially considering the worst-case scenarios where slicing may not benefit from such optimizations, we generally consider the string slicing time complexity as  $O(k)$ .