Solution: Word Break II

Let's solve the Word Break II problem using the Dynamic Programming pattern.

We'll cover the following

Statement

You are given a string, `s`, and an array of strings, `wordDict`, representing a dictionary. Your task is to add spaces to `s` to break it up into a sequence of valid words from `wordDict`. We are required to return an array of all possible sequences of words (sentences). The order in which the sentences are listed is not significant.

> **Note:** The same dictionary word may be reused multiple times in the segmentation.

Constraints:

- $1 \leq$ `s.length` ≤ 20
- $1 \leq$ `wordDict.length` ≤ 1000
- $1 \leq$ `wordDict[i].length` ≤ 10
- `s` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are unique.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach to solve this problem is to use a traditional recursive strategy in which we take each prefix of the input string, `s`, and compare it to each word in the dictionary. If it matches, we take the string's suffix and repeat the process.

Here is how the algorithm works:

- **Base case:** If the string is empty, there are no characters in the string that are left to process, so there'll be no sentences that can be formed. Hence, we return an empty array.

- Otherwise, the string will not be empty, so we'll iterate every word of the dictionary and check whether or not the string starts with the current dictionary word. This ensures that only valid word combinations are considered:
    - If it doesn't start with the current dictionary word, no valid combinations can be formed from this word, so we move on to the next dictionary word.
    - If it does start with the current dictionary word, we have two options:
        - If the length of the current dictionary word is equal to the length of the string, it means the entire string can be formed from the current dictionary word. In this case, the string `s` is directly added to the result without any further processing.
        - **Recursive case:** Otherwise, the length of the current dictionary word will be less than the length of the string. This means that the string can be broken down further. Therefore, we make a recursive call to evaluate the remaining portion (suffix) of the string.
    - We'll then concatenate the prefix and the result of the suffix computed by the recursive call above and store it in the result.
- After all possible combinations have been explored, we return the result.

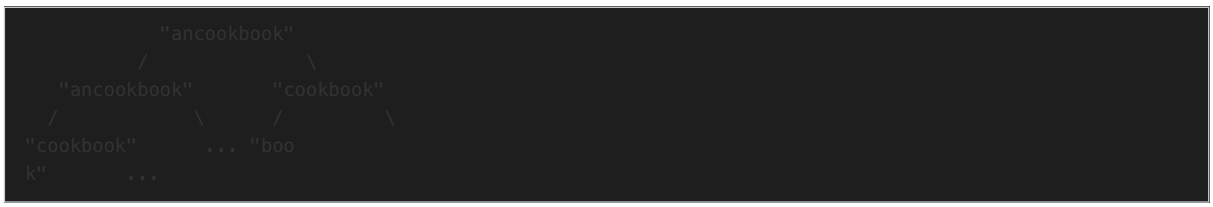
The worst-case time complexity of this solution is when all the substrings that can be extracted from the input string are present in the dictionary. The time complexity in this case is $O((w + n) \times 2^n)$.

The space complexity of this solution is $O(n + w)$, since the solution uses the stack to store the data for each recursive call, and there may be, at most, $n + w$ calls in any given branch of the recursive call tree.

Optimized approach using dynamic programming

Since the recursive solution to this problem is very costly, let's see if we can reduce this cost in any way. Dynamic programming helps us avoid recomputing the same subproblems. Therefore, let's analyze our recursive solution to see if it has the properties needed for conversion to dynamic programming.

- **Optimal substructure:** Given an input string ,`s`, that we want to break up into dictionary words, we find the first word that matches a word from the dictionary, and then repeat the process for the remaining, shorter input string. This means that, to solve the problem for input q, we need to solve the same problem for p, where p is at least one character shorter than q. Therefore, this problem obeys the optimal substructure property.
- **Overlapping subproblems:** The algorithm solves the same subproblems repeatedly. Consider input string "ancookbook" and the dictionary ["an", "book", "cook", "cookbook"]. The following is the partial call tree for the naive recursive solution:



From the tree above, it can be seen that the subproblem "cookbook" is evaluated twice.

To take advantage of these opportunities for optimization, we will use bottom-up dynamic programming, also known as the tabulation approach. This is an iterative method of solving dynamic programming problems. The idea is that if a prefix of the input string matches any word w in the dictionary, we can split the string into two parts: the matching word and the suffix of the input string. We start from an empty prefix which is the base case. The prefix would eventually develop into the complete input string.

Here's how the algorithm works:

- We initialize an empty lookup table, `dp`, of length, $n + 1$, where `dp[i]` will correspond to the prefix of length `i`. This table will be used to store the solutions to previously solved subproblems. It will have the following properties:
    - The first entry of the table will represent a prefix of length 0, i.e., an empty string "".
    - The rest of the entries will represent the other prefixes of the string `s`. For example, the input string "vegan" will have the prefixes "v", "ve", "veg", "vega", and "vegan".
    - Each entry of the table will contain an array containing the sentences that can be formed from the respective prefix. At this point, all the arrays are empty.
- For the base case, we add an empty string to the array corresponding to the first entry of the `dp` table. This is because the only sentence that can be formed from an empty string is an empty string itself.
- Next, we traverse the input string by breaking it into its prefixes by including a single character, one at a time, in each iteration.
    - For the current prefix, we initialize an array, `temp`, that will store the valid sentences formed from that prefix. Let's suppose that the input string is "vegan", and that the current prefix is "vega".
    - For all possible suffixes of the current prefix, we check if the suffix exists in the given dictionary. In our example, this would mean checking the dictionary for the suffixes "vega", "ega", "ga", and "a". For each suffix, it will either match a dictionary word, or not:
        - If it does, we know that the suffix is a valid word from the dictionary and can be used as part of the solution. Therefore, in the `dp` table, we retrieve all the possible sentences for the prefix to the left of this suffix. Supposing that the current suffix of "vega" is "a", and that "a" is present in the dictionary, we would retrieve all the sentences already found for "veg". This means that we reuse the solutions of the subproblem smaller than the current subproblem. Now, we form new sentences for the current prefix by appending a space character and the current suffix (which is a valid dictionary word) to each of the retrieved sentences. Supposing that the valid sentences for the subproblem "veg" are "v eg", and "ve g", we will add these new sentences for the current subproblem, "vega": "veg a", "v eg a", and "ve g a". We add the new sentences to the `temp` array of this prefix.
        - If the suffix is not present in the dictionary, no sentences can be made from the current prefix, so the `temp` array of that prefix remains empty.
    - We repeat the above steps for all suffixes of the current prefix.
    - We set the entry corresponding to the current prefix in the `dp` table equal to the `temp` array.
- We repeat the steps above for all prefixes of the input string.
- After all the prefixes have been evaluated, the last entry of the `dp` table will be an array containing all the sentences formed from the largest prefix, i.e., the complete string. Therefore, we return this array.

The slides below illustrate how the algorithm runs:

s	v	e	g	a	n

wordDict	"veg"	"vegan"	"an"

index	prefix	dp
0	""	[ ]
1	"v"	[ ]
2	"ve"	[ ]
3	"veg"	[ ]
4	"vega"	[ ]
5	"vegan"	[ ]

We will be finding the solutions for every prefix one by one.

Let's look at the code for this solution below:

Java

```java
class WordBreak {
    public static List<String> wordBreak(String s, List<String> wordDict) {
        // Initializing the dp table of size s.length + 1
        List<List<String>> dp = new ArrayList<>(s.length() + 1);
        for (int i = 0; i <= s.length(); i++) {
            dp.add(new ArrayList<>());
        }

        // Setting base case
        dp.get(0).add("");

        // For each substring in the input string, repeat the process.
        for (int i = 1; i <= s.length(); i++) {
            String prefix = s.substring(0, i);
            List<String> temp = new ArrayList<>();

            // Iterate over the current prefix and break it down into all possible suffixes.
            for (int j = 0; j < i; j++) {
                String suffix = prefix.substring(j);

                // Check if the current suffix exists in wordDict. If it does, we know that it is a valid
                // and can be used as part of the solution.
                if (wordDict.contains(suffix)) {

                    // Retrieve the valid sentences from the previously computed subproblem
                    for (String substring : dp.get(j)) {
                        // Merge the suffix with the already calculated results, excluding the leading sp
                        temp.add(substring + (substring.isEmpty() ? "" : " ") + suffix);
```

Solution summary

To recap, the solution to this problem can be divided into the following six main steps:

1. We create a 2D table where each entry corresponds to a prefix of the input string. At this point, each entry contains an empty array.
2. We iterate over all prefixes of the input string. For each prefix, we iterate over all of its suffixes.
3. For each suffix, we check whether it's a valid word, i.e., whether it's present in the provided dictionary.
4. If the suffix is a valid word, we combine it with all valid sentences from the corresponding entry (in the table) of the prefix to the left of it.
5. We store the array of all possible sentences that can be formed using the current prefix in the corresponding entry of the table.
6. After processing all prefixes of the input string, we return the array in the last entry of our table.

Time complexity

The time complexity of this solution is $O(n^2(w + 2^n))$, where n is the length of the input string, and w is the number of words in the dictionary.

Explanation:

To understand the worst case time complexity of this solution, consider the input string "abcd" and the dictionary ["a", "b", "c", "d", "ab", "bc", "cd", "abc", "bcd", "abcd"]. To ensure that the maximum possible number of iterations are performed at every step of our solution, we have included all the possible substrings of length 1, length 2, length 3, and length 4.

- There are 4 prefixes in the string, so the outer loop runs 4 times. In terms of the length of the input string, n, the outer loop takes $O(n)$ time.
- Since we iteratively consider prefixes of increasing size, the number of suffixes increases as well. In our example, there is 1 suffix in the prefix "a", 2 suffixes in the prefix "ab", 3 in "abc", and 4 in "abcd". Therefore, the inner loop runs $(1 + 2 + 3 + 4) = 10$. In terms of the length of the input string, n, the inner loop takes $O(\frac{n(n+1)}{2})$ time, that is, $O(n^2)$.
- Since the dictionary contains all possible suffixes of the input string, the current suffix in the iteration of the inner loop will always be present in the dictionary. Therefore, it will take $O(w)$ time to search for each suffix in the dictionary. The cost up to this point is $O(n^2 \times w)$.
- The following valid sentences can be created for each prefix, based on the provided dictionary:

Prefix	Valid sentences
"abc"	["abc", "ab c", "a bc", "a b c"]
"abcd"	["abcd", "abc d", "ab cd", "ab c d", "a bcd", "a bc d", "a b cd", "a b c d"]

As we can see, the number of valid sentences for each prefix doubles every time a character is added to the prefix:

$$1 + 2 + 4 + 8 = 15$$

$$2^0 + 2^1 + 2^2 + 2^3 = 2^4 - 1$$

Therefore, the general formula for the total valid sentences for a string of length n becomes:

$$2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{n-1} = 2^n - 1$$

In the worst-case scenario above, we ensure that the solution of the next smaller subproblem has the maximum possible number of sentences. So, given that each suffix exists in the dictionary, it will take $O(n^2 \times 2^n)$ to generate these sentences.

Therefore, the overall time complexity becomes $O(n^2(w + 2^n))$.

Space complexity

The space complexity of this solution is $O(2^n)$.

Explanation:

As explained above, for an input string of length n, the total number of valid sentences in the `dp` table are:

$$2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{n-1} = 2^n - 1$$

Therefore, the total space occupied by the `dp` table will be $O(1 + 2^n - 1)$, which is simplified to $O(2^n)$.