Word Break

Solution

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**.

Solution video:
https://youtu.be/wUMFty5TG3A

Solution Article

Approach 1: Brute Force

Algorithm

The naive approach to solve this problem is to use recursion and backtracking. For finding the solution, we check every possible prefix of that string in the dictionary of words, if it is found in the dictionary, then the recursive function is called for the remaining portion of that string. And, if in some function call it is found that the complete string is in dictionary, then it will return true.

> **Caution:** This brute-force approach is included because it is generally a good place to start when trying to develop an efficient solution. However, this approach is not expected to pass all test cases. As such, we will discuss how to improve this solution in the following approaches.

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> word_set(wordDict.begin(), wordDict.end());
        return wordBreakRecur(s, word_set, 0);
    }

    bool wordBreakRecur(string& s, set<string>& word_set, int start) {
        if (start == s.length()) {
            return true;
        }
        for (int end = start + 1; end <= s.length(); end++) {
            if (word_set.find(s.substr(start, end - start)) != word_set.end() and
                wordBreakRecur(s, word_set, end)) {
                return true;
            }
        }
        return false;
    }
};
```

Complexity Analysis

n is the length of the input string.

- Time complexity : $O(2^n)$. Given a string of length n, there are $n + 1$ ways to split it into two parts. At each step, we have a choice: to split or not to split. In the worse case, when all choices are to be checked, that results in $O(2^n)$.
- Space complexity : $O(n)$. The depth of the recursion tree can go upto n.

Approach 2: Recursion with memoization

Algorithm

In the previous approach we can see that many subproblems were redundant, i.e we were calling the recursive function multiple times for a particular string. To avoid this we can use memoization method, where an array $memo$ is used to store the result of the subproblems. Now, when the function is called again for a particular string, value will be fetched and returned using the $memo$ array, if its value has been already evaluated.

With memoization many redundant subproblems are avoided and recursion tree is pruned and thus it reduces the time complexity by a large factor.

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> word_set(wordDict.begin(), wordDict.end());
        // In the memo table, -1 stands for the state not yet reached,
        // 0 for false and 1 for true
        vector<int> memo(s.length(), -1);
        return wordBreakMemo(s, word_set, 0, memo);
    }

    bool wordBreakMemo(string& s, set<string>& word_set, int start, vector<int>& memo)
{
        if (start == s.length()) {
            return true;
        }
        if (memo[start] != -1) {
            return memo[start];
        }
        for (int end = start + 1; end <= s.length(); end++) {
            if (word_set.find(s.substr(start, end - start)) != word_set.end() and
                wordBreakMemo(s, word_set, end, memo)) {
                return memo[start] = true;
            }
        }
        return memo[start] = false;
    }
};
```

Approach 3: Using Breadth-First-Search

Algorithm

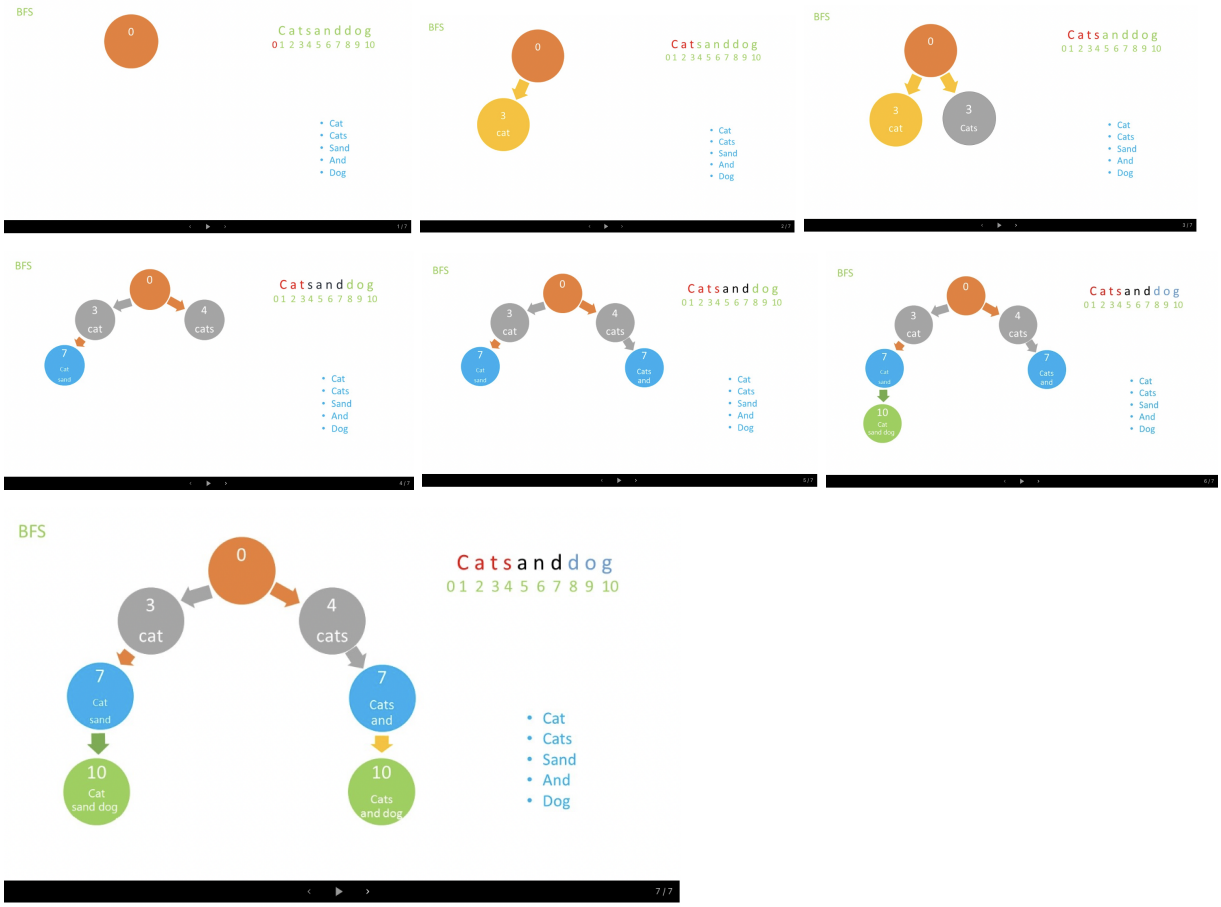
Another approach is to use Breadth-First-Search. Visualize the string as a tree where each node represents the prefix upto index end. Two nodes are connected only if the substring between the indices linked with those nodes is also a valid string which is present in the dictionary. In order to form such a tree, we start with the first character of the given string (say s) which acts as the root of the tree being formed and find every possible substring starting with that character which is a part of the dictionary. Further, the ending index (say i) of every such substring is pushed at the back of a queue which will be used for Breadth First Search. Now, we pop an element out from the front of the queue and perform the same process considering the string $s(i+1, end)$ to be the original string and the popped node as the root of the tree this time. This process is continued, for all the nodes appended in the queue during the course of the process. If we are able to obtain the last element of the given string as a node (leaf) of the tree, this implies that the given string can be partitioned into substrings which are all a part of the given dictionary.

The formation of the tree can be better understood with this example:

Complexity Analysis

n is the length of the input string.

- Time complexity : $O(n^3)$. Size of recursion tree can go up to n^2.
- Space complexity : $O(n)$. The depth of recursion tree can go up to n.



```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> word_set(wordDict.begin(), wordDict.end());
        queue<int> q;
        vector<bool> visited(s.length(), false);
        q.push(0);
        while (!q.empty()) {
            int start = q.front();
            q.pop();
            if (visited[start]) {
                continue;
            }
            for (int end = start + 1; end <= s.length(); end++) {
                if (word_set.find(s.substr(start, end - start)) !=
                    word_set.end()) {
                    q.push(end);
                    if (end == s.length()) {
                        return true;
                    }
                }
            }
            visited[start] = true;
        }
        return false;
    }
};
```

Complexity Analysis

n is the length of the input string.

- Time complexity : $O(n^3)$. For every starting index, the search can continue till the end of the given string.
- Space complexity : $O(n)$. Queue of at most n size is needed.

Approach 4: Using Dynamic Programming

Algorithm

The intuition behind this approach is that the given problem (s) can be divided into subproblems $s1$ and $s2$. If these subproblems individually satisfy the required conditions, the complete problem, s also satisfies the same. e.g. "$\mathrm{catsanddog}$" can be split into two substrings "$\mathrm{catsand}$", "dog". The subproblem "$\mathrm{catsand}$" can be further divided into "cats","and", which individually are a part of the dictionary making "$\mathrm{catsand}$" satisfy the condition. Going further backwards, "$\mathrm{catsand}$", "dog" also satisfy the required criteria individually leading to the complete string "$\mathrm{catsanddog}$" also to satisfy the criteria.

Now, we'll move onto the process of dp array formation. We make use of dp array of size $n+1$, where n is the length of the given string. We also use two index pointers i and j, where i refers to the length of the substring (s') considered currently starting from the beginning, and j refers to the index partitioning the current substring (s') into smaller substrings $s'(0,j)$ and $s'(j+1,i)$. To fill in the dp array, we initialize the element $\mathrm{dp}[0]$ as true, since the null string is always present in the dictionary, and the rest of the elements of dp as false. We consider substrings of all possible lengths starting from the beginning by making use of index i. For every such substring, we partition the string into two further substrings $s1'$ and $s2'$ in all possible ways using the index j (Note that the i now refers to the ending index of $s2'$). Now, to fill in the entry $\mathrm{dp}[i]$, we check if the $\mathrm{dp}[j]$ contains true, i.e. if the substring $s1'$ fulfills the required criteria. If so, we further check if $s2'$ is present in the dictionary. If both the strings fulfill the criteria, we make $\mathrm{dp}[i]$ as true, otherwise as false.

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        set<string> word_set(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.length() + 1);
        dp[0] = true;

        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] and
                    word_set.find(s.substr(j, i - j)) != word_set.end()) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
};
```

Complexity Analysis

n is the length of the input string.

- Time complexity : $O(n^3)$. There are two nested loops, and substring computation at each iteration. Overall that results in $O(n^3)$ time complexity.
- Space complexity : $O(n)$. Length of p array is $n + 1$.