Previous Next

139. Word Break

Jan. 1, 2017 | 180.5K views

Average Rating: 4.31 (112 votes)

Given a **non-empty** string *s* and a dictionary *wordDict* containing a list of **non-empty** words, determine if *s* can be segmented into a space-separated sequence of one or more dictionary words.

Note:

- The same word in the dictionary may be reused multiple times in the segmentation.
- You may assume the dictionary does not contain duplicate words.

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
```

Solution

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.

Java

```java
public class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        return word_Break(s, new HashSet(wordDict), 0);
    }
    public boolean word_Break(String s, Set<String> wordDict, int start) {
        if (start == s.length()) {
            return true;
        }
        for (int end = start + 1; end <= s.length(); end++) {
            if (wordDict.contains(s.substring(start, end)) && word_Break(s, wordDict, end)) {
                return true;
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^n)$. Consider the worst case where s = "aaaaaaa" and every prefix of s is present in the dictionary of words, then the recursion tree can grow upto n^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.

Java

```java
public class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        return word_Break(s, new HashSet(wordDict), 0, new Boolean[s.length()]);
    }
    public boolean word_Break(String s, Set<String> wordDict, int start, Boolean[] memo) {
        if (start == s.length()) {
            return true;
        }
        if (memo[start] != null) {
            return memo[start];
        }
        for (int end = start + 1; end <= s.length(); end++) {
            if (wordDict.contains(s.substring(start, end)) && word_Break(s, wordDict, end, memo)) {
                return memo[start] = true;
            }
        }
        return memo[start] = false;
    }
}
```

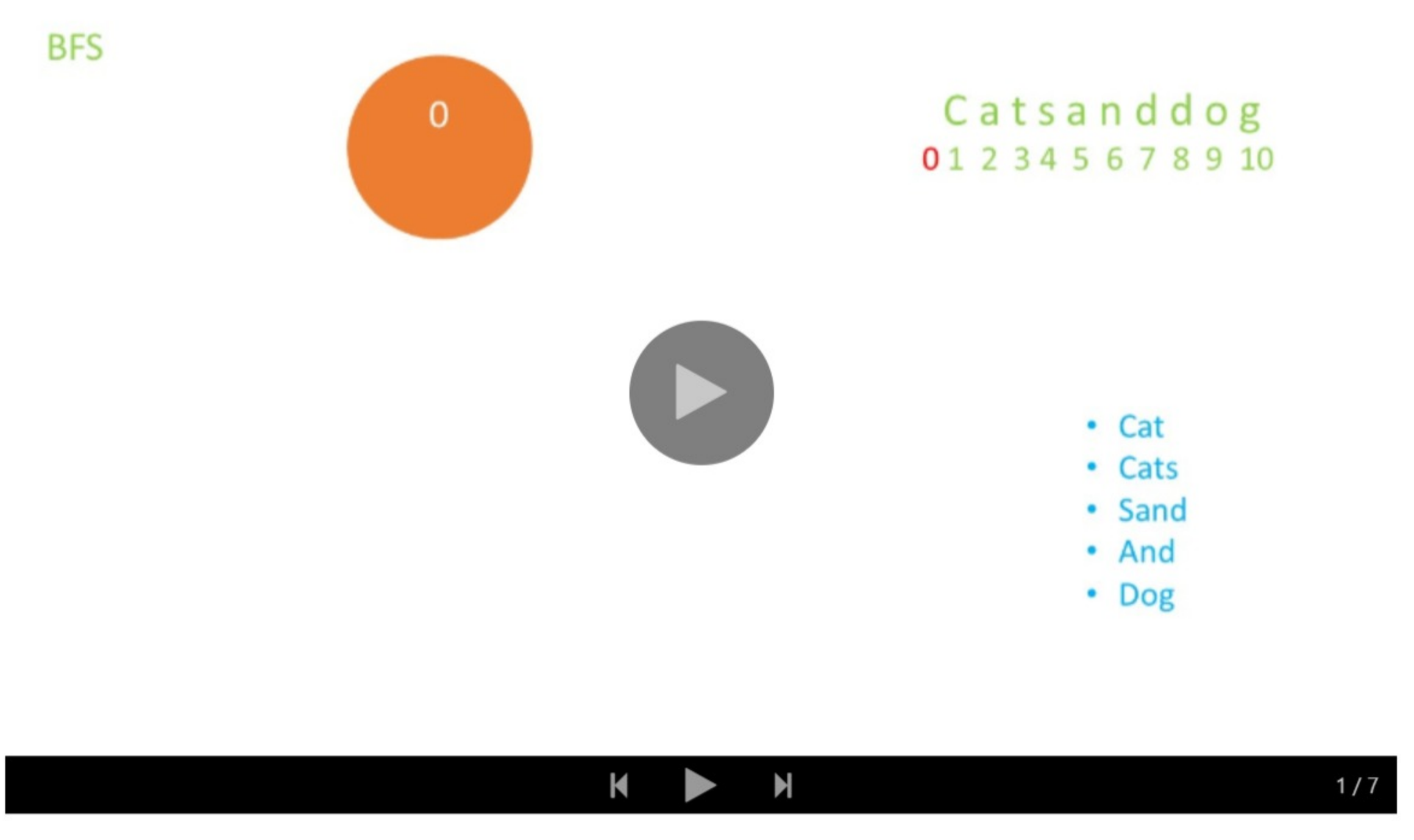
Complexity Analysis

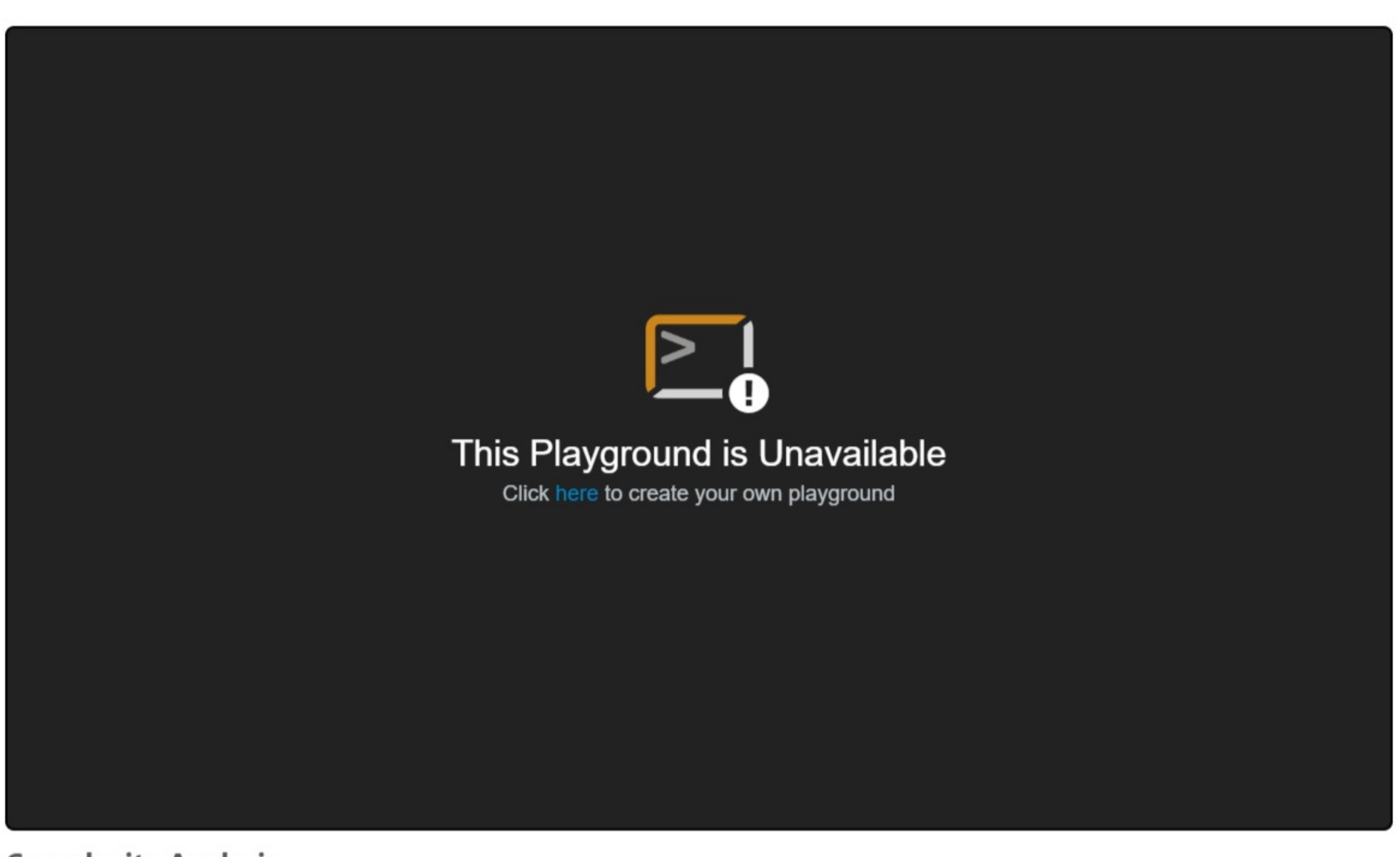
- Time complexity : $O(n^2)$. Size of recursion tree can go up to n^2.
- Space complexity : $O(n)$. The depth of recursion tree can go up to n.

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.





Complexity Analysis

- Time complexity : $O(n^2)$. For every starting index, the search can continue till the end of the given string.
- Space complexity : $O(n)$. Queue of atmost 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. "catsanddog" can be split into two substrings "catsand", "dog". The subproblem "catsand" can be further divided into "cats","and", which individually are a part of the dictionary making "catsand" satisfy the condition. Going further backwards, "catsand", "dog" also satisfy the required criteria individually leading to the complete string "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 $\text{dp}[0]$ 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 $\text{dp}[i]$, we check if the $\text{dp}[j]$ 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 $\text{dp}[i]$ as true, otherwise as false.

Java

```java
public class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        Set<String> 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()];
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two loops are their to fill dp array.
- Space complexity : $O(n)$. Length of p array is $n+1$.

Rate this article:

Previous Next

Comments: 113

Sort By

Type comment here... (Markdown is supported)

Preview Post

pzhang15 510 July 25, 2018 7:51 PM

This is definitely hard level

319 Share Reply

SHOW 10 REPLIES

R0ckyY2 519 September 12, 2018 11:13 AM

The time complexity of the first method (brute force) should be O(2^n), not O(n^n). See my discussion post https://leetcode.com/problems/word-break/discuss/169383/The-Time-Complexity-of-The-Brute-Force-Method-Should-Be-O(2n)-and-Prove-It-Below

88 Share Reply

SHOW 8 REPLIES

katrinivini 59 September 27, 2018 10:04 PM

can someone explain why the dp O(n^2) solutions are not O(n^3)? I believe it's cubed due to the substring method

43 Share Reply

SHOW 12 REPLIES

trungvo 38 March 29, 2018 4:06 AM

Substring's complexity is O(n), so DP solution is O(n^3)

38 Share Reply

SHOW 2 REPLIES

ktslwy 125 June 3, 2018 11:11 AM

I feel like the BFS approach uses the same idea as the recursion+memoization. It's just a different way to present the solution, iterative vs recursive.

27 Share Reply

SHOW 1 REPLY

HugoZh 28 July 31, 2018 7:16 AM

For solution 2, please check whether my thought process for getting time complexity is correct or not.
If we don't use memorization the recurrence relation of time complexity will be :
T(n) = T(n-1) + T(n-2) + T(n-3) + ... + T(1)
Now if we add memorization, by the time we finish doing T(n-1), We already have the memorization result for n - 2, n - 3 ... 1.

Read More

25 Share Reply

SHOW 1 REPLY

redrobin100 23 February 20, 2018 9:11 AM

Solutions from the Discuss section are far better, sorry, these solutions are poorly worded and the analysis is incorrect

23 Share Reply

abhijith97 15 November 2, 2018 4:39 PM

We can use a trie along with the DP to solve the problem in worst case O(N * Max length of word in dictionary). Store all the dictionary words inverted in a trie. At stage i of the DP, start traversing the trie for each letter coming behind i and find out all indices where you get an end of word symbol and call DP on those indices.

15 Share Reply

SHOW 4 REPLIES

StefanPochmann 50491 May 26, 2017 3:45 PM

One more thing: You keep misspelling "memoization" as "memorization".

18 Share Reply

EthanXiaoMa 87 January 25, 2019 8:13 AM

Is there anyone else think bfs approach takes O(n^3) because of substring?

8 Share Reply

SHOW 2 REPLIES

< 1 2 3 4 5 6 ... 11 12 >