<https://leetcode.com/problems/word-break/>

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**.

**Attempt 1: 2023-10-09**

**Solution 1: Native DFS (10 min, TLE 35/46)**

**Style 1: No change on original input String 's' but given startIndex to identify from which position we continue scan original String s**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

// list.contains() cost O(N), better convert 'wordDict' to set

// which support set.contains() in O(1)

return helper(s, new HashSet<String>(wordDict), 0);

}

private boolean helper(String s, Set<String> wordDict, int startIndex) {

if(startIndex == s.length()) {

return true;

}

for(int endIndex = startIndex + 1; endIndex <= s.length(); endIndex++) {

if(wordDict.contains(s.substring(startIndex, endIndex)) && helper(s, wordDict, endIndex)) {

return true;

}

}

return false;

}

}

Time Complexity: O(2^N)

Space Complexity: O(2^N)

**Refer to**

<https://leetcode.com/problems/word-break/solutions/43886/evolve-from-brute-force-to-optimal-a-review-of-all-solutions/>

Recursion, Average O(2^n), java is more expensive due to use of substring

T(n) = T(n-1)+T(n-2)+...+T(1)

=> T(n+1) = T(n)+T(n-1)+T(n-2)+...+T(1)

=>T(n+1) = 2T(n)

public boolean wordBreak(String s, List<String> wordDict) {

return wordBreak(0,s,new HashSet(wordDict));

}

private boolean wordBreak(int p, String s, Set<String> dict){

int n=s.length();

if(p==n) {

return true;

}

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

if(dict.contains(s.substring(p,i)) && wordBreak(i,s,dict)) {

return true;

}

}

return false;

}

**Style 2: Change on original input String 's' as cut off before endIndex and no given startIndex, since 's' changed by directly cut off from ahead, startIndex in recursion always auto as 0**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

return helper(s, new HashSet<String>(wordDict));

}

private boolean helper(String s, Set<String> wordDict) {

if(s.length() == 0) {

return true;

}

for(int endIndex = 1; endIndex <= s.length(); endIndex++) {

if(wordDict.contains(s.substring(0, endIndex)) && helper(s.substring(endIndex), wordDict)) {

return true;

}

}

return false;

}

}

Time Complexity: O(2^N)

Space Complexity: O(2^N)

**Style 3:  Reverse of Style 1 ("顶" 和 "底"条件交换)**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

return helper(s, new HashSet<>(wordDict), s.length());

}

private boolean helper(String s, Set<String> set, int endIndex) {

if(endIndex == 0) {

return true;

}

for(int startIndex = endIndex - 1; startIndex >= 0; startIndex--) {

if(set.contains(s.substring(startIndex, endIndex)) && helper(s, set, startIndex)) {

return true;

}

}

return false;

}

}

Time Complexity: O(2^N)

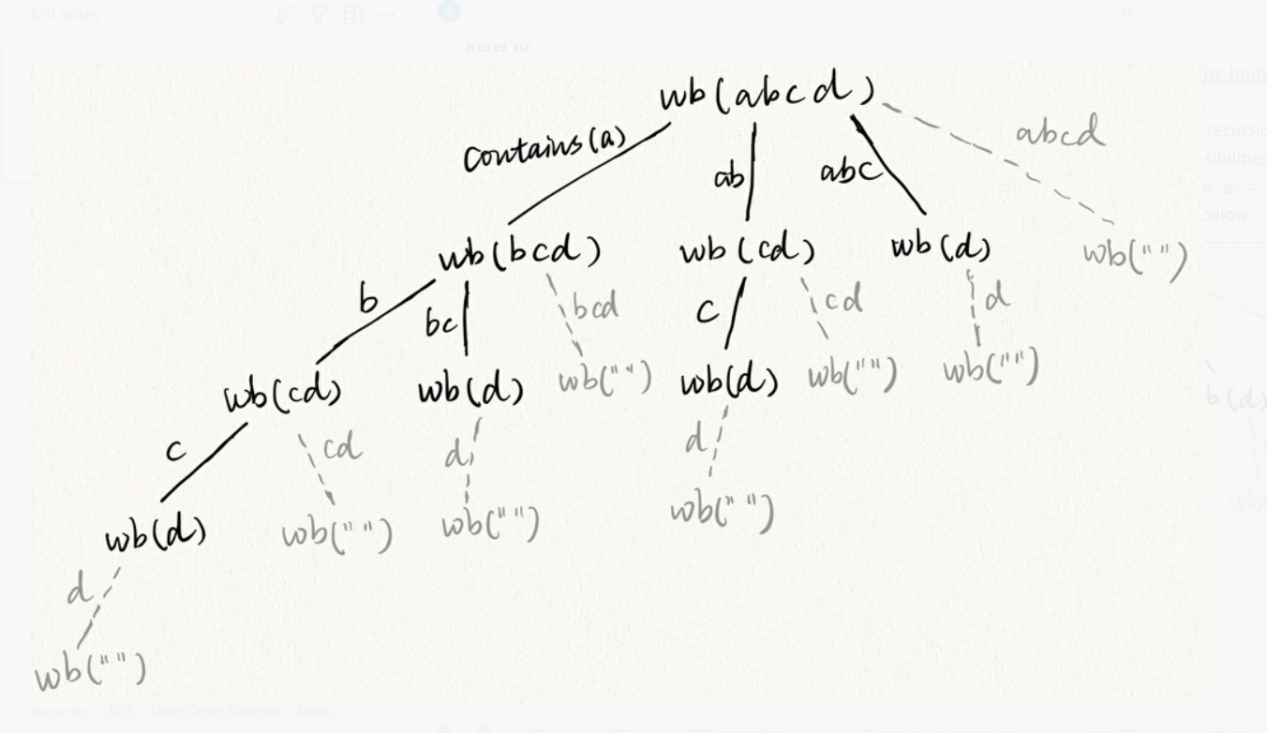
Space Complexity: O(2^N)

**Time Complexity analysis**

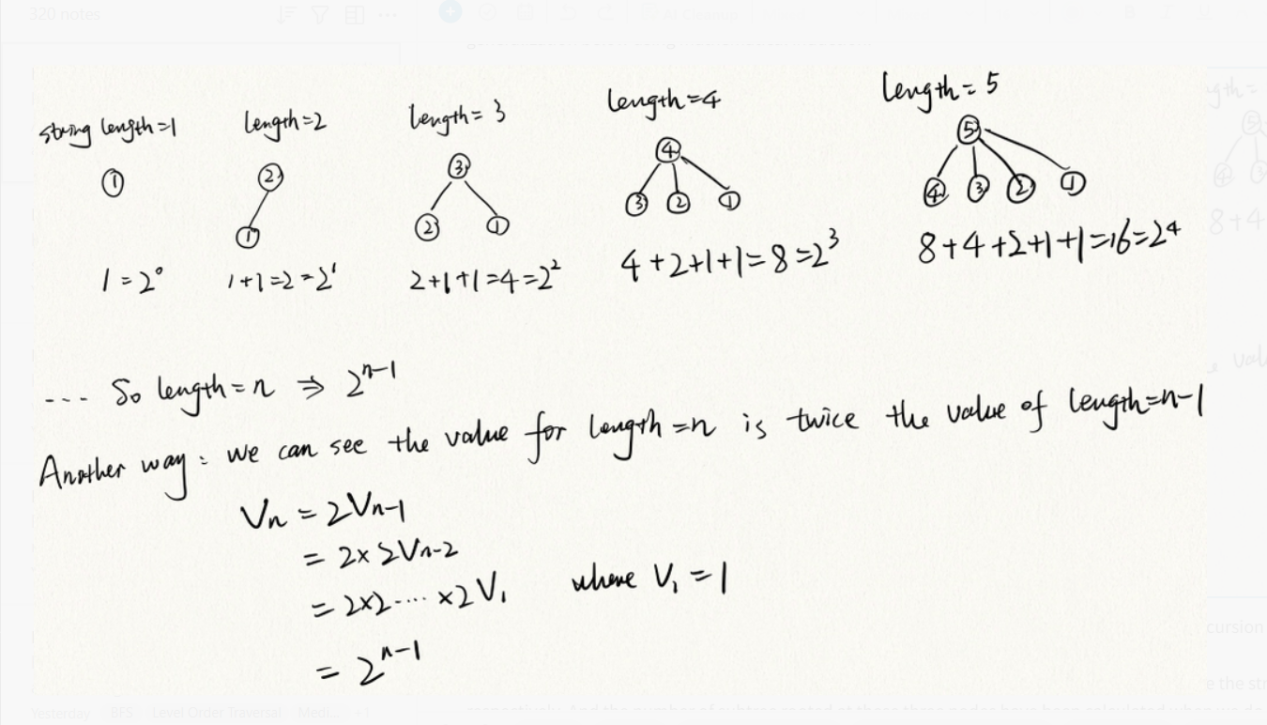
**Refer to**

<https://leetcode.com/problems/word-break/solutions/169383/solved-the-time-complexity-of-the-brute-force-method-should-be-o-2-n-and-prove-it-below/>

The time complexity depends on how many nodes the recursion tree has. In the worst case, the recursion tree has the most nodes, which means the program should not return in the middle and it should try as many possibilities as possible. So the branches and depth of the tree are as many as possible. For the worst case, for example, we take s = "abcd" and wordDict = ["a", "b", "c", "bc", "ab", "abc"], the recursion tree is shown below:



From the code if (set.contains(s.substring(0, i)) && wb(s.substring(i), set)) { }, we can see that only if the wordDict contains the prefix, the recursion function can go down to the next level. So on the figure above, string on the edge means the wordDict contains that string. All the gray node with empty string cannot be reached because if the program reaches one such node, the program will return, which lead to some nodes right to it will not be reached. So the conclusion is for a string with length 4, the recursion tree has 8 nodes (all black nodes), and 8 is 2^(4-1). So to generalize this, for a string with length n, the recursion tree will have 2^(n-1) nodes, i.e., the time complexity is O(2^n). I will prove this generalization below using mathematical induction:



Explanation: the value of a node is the string length. We calculate the number of nodes in the recursion tree for string length=1, 2, ...., n respectively.

For example, when string length=4, the second layer of the recursion tree has three nodes where the string length is 3, 2 and 1 respectively. And the number of subtree rooted at these three nodes have been calculated when we do the mathematical induction.

So time complexity is O(2^n).

**Solution 2: DFS  + Memoization (10 min)**

**Style 1: No change on original input String 's' but given startIndex to identify from which position we continue cut original String s, use input changing index 'i' as memo config**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Map<Integer, Boolean> memo = new HashMap<>();

// list.contains() cost O(N), better convert 'wordDict' to set

// which support set.contains() in O(1)

return helper(s, new HashSet<String>(wordDict), 0, memo);

}

private boolean helper(String s, Set<String> wordDict, int startIndex, Map<Integer, Boolean> memo) {

if(startIndex == s.length()) {

return true;

}

if(memo.containsKey(startIndex)) {

return memo.get(startIndex);

}

for(int endIndex = startIndex + 1; endIndex <= s.length(); endIndex++) {

if(wordDict.contains(s.substring(startIndex, endIndex)) && helper(s, wordDict, endIndex, memo)) {

memo.put(startIndex, true);

return true;

}

}

memo.put(startIndex, false);

return false;

}

}

Time Complexity: O(N^3) -> additional N for Java substring

Space Complexity: O(N^3)

**Style 2: Change on original input String 's' as cut off before endIndex and no given startIndex, since 's' changed by directly cut off from ahead, startIndex in recursion always auto as 0, use input changing 's'  as memo config**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Map<String, Boolean> memo = new HashMap<>();

return helper(s, new HashSet<String>(wordDict), memo);

}

private boolean helper(String s, Set<String> wordDict, Map<String, Boolean> memo) {

if(s.length() == 0) {

return true;

}

if(memo.containsKey(s)) {

return memo.get(s);

}

for(int endIndex = 1; endIndex <= s.length(); endIndex++) {

if(wordDict.contains(s.substring(0, endIndex)) && helper(s.substring(endIndex), wordDict, memo)) {

memo.put(s, true);

return true;

}

}

memo.put(s, false);

return false;

}

}

Time Complexity: O(N^3) -> additional N for Java substring

Space Complexity: O(N^3)

**Style 3:  Reverse of Style 1 ("顶" 和 "底"条件交换)**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Map<Integer, Boolean> memo = new HashMap<>();

return helper(s, new HashSet<>(wordDict), s.length(), memo);

}

private boolean helper(String s, Set<String> set, int endIndex, Map<Integer, Boolean> memo) {

if(endIndex == 0) {

return true;

}

if(memo.containsKey(endIndex)) {

return memo.get(endIndex);

}

for(int startIndex = endIndex - 1; startIndex >= 0; startIndex--) {

if(set.contains(s.substring(startIndex, endIndex)) && helper(s, set, startIndex, memo)) {

memo.put(endIndex, true);

return true;

}

}

memo.put(endIndex, false);

return false;

}

}

**Refer to**

java O(n^3)

<https://leetcode.com/problems/word-break/solutions/43886/evolve-from-brute-force-to-optimal-a-review-of-all-solutions/>

DFS with Memoization. There is redundancy in #1. A substr may be checked multiple times. We can cache the result by memoization. This is the optimal solution.

Boolean[] mem;

public boolean wordBreak(String s, List<String> wordDict) {

mem=new Boolean[s.length()];

return wordBreak(0,s,new HashSet<String>(wordDict));

}

private boolean wordBreak(int p, String s, Set<String> dict){

int n=s.length();

if(p==n) {

return true;

}

if(mem[p]!=null) {

return mem[p];

}

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

if(dict.contains(s.substring(p,i))&&wordBreak(i,s,dict)) {

return mem[p]=true;

}

}

return mem[p]=false;

}

**Why we can use Memoization to reduce the calculation cost ?**

**Refer to**

<https://algorithmstuff.wordpress.com/tag/memoization/>

**Problem description**

Given a String and a dictionary of words, write a program that returns true if the given string can be formed by concatenating one or more of the words in the dictionary, e.g.

Given the string S = “algorithmstuff” and the dictionary dic = {“algorithm”, “stuff”}

Our function must return true since S can be formed by concatenating “algorithm” + “stuff” but given S = “algorithmstuffisgood” and dic = {“algorithms”, “good”, “stuff”} must return false since there’s no way to build S from the strings in the dictionary.

**Solution**

This is a very good problem for a programming interview since it has one “lazy solution” which is very inefficient but it can be converted to a very fast algorithm with just a simple modification, so you could think about the most obvious solution even if it’s slow and once you got a working solution you can think about how to improve it.

This problem is clearly a good candidate for a recursive solution, for each recursive call the idea is to start checking if the first character is in our dictionary, if so, we call the function recursively but now with string S starting with the next character, otherwise we check if the first two characters form a valid word and so on, for instance:

S = “hithere”

dic = { “there”, “hi”}

We start with the string “hithere” at index 0, so we check if the substring “h” is a valid word, since it’s not we continue now with the next character which is “i”, now we check if the substring “hi” is a valid word in the dictionary, it is so we know that “hi” can be formed by concatenating one or more words of the dictionary so we now call the function recursively but starting with the next index which is 2, so now we are gonna have the same function with the substring “there”, we are gonna stop when we reach the end of the string S.

This is basically a DFS algorithm (<http://en.wikipedia.org/wiki/Depth-first_search>) , here the code:

public boolean stringBreak(String s, Set<String> dict) {

return dfs(s, 0, dict);

}

boolean dfs(String s, int i, Set<String> dict)

{

if(dict.contains(s.substring(i)))

return true;

for(int j = i; j < s.length(); ++j)

if(dict.contains(s.substring(i, j + 1)))

if(dfs(s, j + 1, dict, set)) return true;

return false;

}

The solution above works! but it’s too slow since there are many overlapping calls (it does the same thing many times), let me show you with an example:

let’s say we have the string S = “helloworld” and dict = {“he”, “e”, “h”}

**Our recursive function is gonna build a tree like this:**

helloworld

/ \

h-elloworld he-lloworld

|

e-lloworld

**As you can see the substring “lloworld” is gonna be computed two times and the result is gonna be the same, the problem here is that we could end up with too many overlapping calls which would be a waste of time. So, how can we avoid overlapping calls?**

**The solution is to have a record of which substrings we have already computed so that when our function is called again for the same substring we already know the answer, this technique is called MEMOIZATION (<http://en.wikipedia.org/wiki/Memoization>), Here is the code that shows this optimization.**

public boolean stringBreak(String s, Set<String> dict) {

HashSet<Integer> memo = new HashSet<Integer>(); // We just need to keep track of those indexes that we have already computed and the result is false

return dfs(s, 0, dict, memo);

}

boolean dfs(String s, int i, Set<String> dict, HashSet<Integer> memo)

{

if(dict.contains(s.substring(i)))

return true;

if(memo.contains(i)) // if we have already computed the result for this substring we just return the answer

return false;

//otherwise, compute the answer for this substring

for(int j = i; j < s.length(); ++j)

if(dict.contains(s.substring(i, j + 1)))

if(dfs(s, j + 1, dict, set)) return true;

//we just store the results for substrings which result is false

memo.add(i);

return false;

}

**Solution 3: DP (10 min)**

**Style 1: 和Solution 1中的DFS的模式保持一致，"顶" -> 递归起点为输入坐标 = 0，"底"->递归终点坐标到达字符串长度，DP直接从"底"到"顶"，在"底"完成初始化，最终返回到达"顶"的值**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>(wordDict);

int len = s.length();

// dp[i]代表以坐标i为起点的s的子字符串(i = 0代表s本身)

// 是否能由wordDict中的单词构造

boolean[] dp = new boolean[len + 1];

// 从"底"开始，遵循DFS Style 1中的"底"，即坐标为字符串s长度

// 的时候，意味着成功遍历完了整个字符串s返回true

dp[len] = true;

// 如果将这里的i换为startIndex，j换为endIndex，则一下就能

// 发现，完全就是DFS Style 1递归中的递推条件的翻版

for(int i = len - 1; i >= 0; i--) {

for(int j = i + 1; j <= len; j++) {

if(set.contains(s.substring(i, j)) && dp[j]) {

dp[i] = true;

}

}

}

// 返回"顶"，遵循DFS Style 1中的"顶"，DFS递归入口，即坐标为0的时候

return dp[0];

}

}

Time Complexity: O(N^3) -> additional N for Java substring

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/word-break/solutions/43886/evolve-from-brute-force-to-optimal-a-review-of-all-solutions/>

java O(n^3)

For dp problems, many times we go into iterative dp directly without even thinking about dfs. This is a great example showing that dfs is better than dp. DFS returns as soon as it finds one way to break the word while dp computes if each substring starting/ending at i is breakable. The test cases of this problem do not show it but it is shown in a similar problem [Concatenated Words](https://discuss.leetcode.com/topic/78569/a-review-of-top-solutions).

public boolean wordBreak(String s, List<String> wordDict) {

int n=s.length();

boolean[] dp=new boolean[n+1];

dp[n]=true;

Set<String> dict=new HashSet(wordDict);

for(int i=n-1;i>=0;i--) {

for(int j=i+1;!dp[i] && j<=n;j++) {

dp[i] = dp[j] & dict.contains(s.substring(i,j));

}

}

return dp[0];

}

**Style 2: 和Solution 1中的DFS的模式恰好相反，"顶" -> 递归起点为输入坐标 = 字符串长度，"底"->递归终点坐标到达0，DP直接从"底"到"顶"，在"底"完成初始化，最终返回到达"顶"的值**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

Set<String> set = new HashSet<>(wordDict);

int len = s.length();

// dp[i]代表以坐标i为起点的s的子字符串(i = len代表s本身)

// 是否能由wordDict中的单词构造

boolean[] dp = new boolean[len + 1];

// 从"底"开始，遵循DFS Style 3中的"底"，即坐标为0的时候，

// 意味着从右往左遍历完字符串的时候返回true

dp[0] = true;

// 如果将这里的i换为endIndex，j换为startIndex，则一下就能

// 发现，完全就是DFS Style 3递归中的递推条件的翻版

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

for(int j = i - 1; j >= 0; j--) {

if(set.contains(s.substring(j, i)) && dp[j]) {

dp[i] = true;

}

}

}

// 返回"顶"，遵循DFS Style 3中的"顶"，DFS递归入口，即坐标为

// 字符串长度的时候

return dp[len];

}

}

Time Complexity: O(N^3) -> additional N for Java substring

Space Complexity: O(N)

**Solution 4: BFS (30 min)**

class Solution {

public boolean wordBreak(String s, List<String> wordDict) {

// Use a set to record checked index to avoid repeated work.

// This is the key to reduce the running time to O(N^2).

Set<Integer> visited = new HashSet<>();

Set<String> set = new HashSet<>(wordDict);

Queue<Integer> q = new LinkedList<>();

q.offer(0);

visited.add(0);

while(!q.isEmpty()) {

int curIdx = q.poll();

for(int i = curIdx + 1; i <= s.length(); i++) {

// Filter out already visited index

if(!visited.contains(i) && set.contains(s.substring(curIdx, i))) {

if(i == s.length()) {

return true;

}

q.offer(i);

visited.add(i);

}

}

}

return false;

}

}

Time Complexity: O(N^3) -> additional N for Java substring

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/word-break/solutions/43886/evolve-from-brute-force-to-optimal-a-review-of-all-solutions/>

BFS, Average O(n^2) BFS may be better than dp. In dp, for each index i, it checks if the substr starting at i can break. However, if the substr ending before i cannot break, then we do not have to check i. But this is the nature of dp, we visit all the states and derive the next state according to previous states and I don't find a way to improve the dp solution. BFS always starts from a valid index and may visit fewer states. This is similar to [Perfect Squares](https://discuss.leetcode.com/topic/57631/evolve-from-brute-force). But BFS is not guaranteed to be better than dp. If every index is a valid state, then BFS visits the same number of states as dp. The inner for loop of BFS looks more expensive than the inner for loop of dp. So I think which is better is case by case. In general, both BFS and dp visit all the states and are less efficient than dfs.

bool wordBreak(string s, unordered\_set<string>& wordDict) {

queue<int> q({0});

unordered\_set<int> vstd;

int n = s.size();

while(!q.empty()) {

int start = q.front();

q.pop();

if(vstd.count(start)) continue;

vstd.insert(start);

string sub;

for(int i=start;i<n;i++)

if(wordDict.count(sub+=s[i])) {

q.push(i+1);

if(i+1 == n) return 1;

}

}

return 0;

}

**Refer to**

<https://leetcode.com/problems/word-break/solutions/43797/a-solution-using-bfs/>

People have posted elegant solutions using DP. The solution I post below using BFS is no better than those. Just to share some new thoughts.

We can use a graph to represent the possible solutions. The vertices of the graph are simply the positions of the first characters of the words and each edge actually represents a word. For example, the input string is "nightmare", there are two ways to break it, "night mare" and "nightmare". The graph would be

0-->5-->9

|\_\_ \_\_ \_^

The question is simply to check if there is a path from 0 to 9. The most efficient way is traversing the graph using BFS with the help of a queue and a hash set. The hash set is used to keep track of the visited nodes to avoid repeating the same work.

For this problem, the time complexity is O(n^2) and space complexity is O(n), the same with DP. This idea can be used to solve the problem word break II. We can simple construct the graph using BFS, save it into a map and then find all the paths using DFS.

bool wordBreak(string s, unordered\_set<string> &dict) {

// BFS

queue<int> BFS;

unordered\_set<int> visited;

BFS.push(0);

while(BFS.size() > 0)

{

int start = BFS.front();

BFS.pop();

if(visited.find(start) == visited.end())

{

visited.insert(start);

for(int j=start; j<s.size(); j++)

{

string word(s, start, j-start+1);

if(dict.find(word) != dict.end())

{

BFS.push(j+1);

if(j+1 == s.size())

return true;

}

}

}

}

return false;

}

----------------------------------------------------------------

Java version:

public boolean wordBreak(String s, Set<String> dict) {

if (dict.contains(s)) return true;

Queue<Integer> queue = new LinkedList<Integer>();

queue.offer(0);

// use a set to record checked index to avoid repeated work.

// This is the key to reduce the running time to O(N^2).

Set<Integer> visited = new HashSet<Integer>();

visited.add(0);

while (!queue.isEmpty()) {

int curIdx = queue.poll();

for (int i = curIdx+1; i <= s.length(); i++) {

if (visited.contains(i)) continue;

if (dict.contains(s.substring(curIdx, i))) {

if (i == s.length()) return true;

queue.offer(i);

visited.add(i);

}

}

}

return false;

}