Word Break Problem

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. See following examples for more details.
This is a famous Google interview question, also being asked by many other companies now a days.

```
Consider the following dictionary
{ i, like, sam, sung, samsung, mobile, ice,
  cream, icecream, man, go, mango}
Input:  ilike
Output: Yes
The string can be segmented as "i like".
Input:  ilikesamsung
Output: Yes
The string can be segmented as "i like samsung"
or "i like sam sung".
```

Recursive implementation:

The idea is simple, we consider each prefix and search for it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix).

```
#include <algorithm>

#include <iostream>

#include <string>

#include <vector>


using namespace std;


// Function to check if the given word can be broken

// down into words from the wordList

bool WordBreak(const vector<string>& wordList,

                              const string& word)

{

        // If the word is empty, it can be broken down into

        // an empty list of words
```

```
	if (word.empty())
		return true;

	int wordLen = word.length();

	// Check if the word can be broken down into
	// words from the wordList
	for (int i = 1; i <= wordLen; ++i) {
		string prefix = word.substr(0, i);

		if (find(wordList.begin(), wordList.end(), prefix)
				!= wordList.end()
			&& WordBreak(wordList, word.substr(i))) {
			return true;
		}
	}

	return false;
}

int main()
{
	vector<string> wordList
		= { "mobile", "samsung", "sam", "sung", "man",
			"mango", "icecream", "and", "go", "i",
			"like", "ice",	"cream" };

	bool result = WordBreak(wordList, "ilikesamsung");
```

```
    cout << boolalpha << result << endl; // Output: true

    return 0;
}
```

If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.
We strongly recommend to see **substr** function which is used extensively in following implementations.

```
// A recursive program to test whether a given
// string can be segmented into space separated
// words in dictionary
#include <iostream>
using namespace std;

/* A utility function to check whether a word is
present in dictionary or not. An array of strings
is used for dictionary. Using array of strings for
dictionary is definitely not a good idea. We have
used for simplicity of the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile","samsung","sam","sung",
                           "man","mango","icecream","and",
                           "go","i","like","ice","cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
```

```
            return true;
        return false;
}

// returns true if string can be segmented into space
// separated words, otherwise returns false
bool wordBreak(string str)
{
        int size = str.size();

        // Base case
        if (size == 0) return true;

        // Try all prefixes of lengths from 1 to size
        for (int i=1; i<=size; i++)
        {
            // The parameter for dictionaryContains is
            // str.substr(0, i) which is prefix (of input
            // string) of length 'i'. We first check whether
            // current prefix is in dictionary. Then we
            // recursively check for remaining string
            // str.substr(i, size-i) which is suffix of
            // length size-i
            if (dictionaryContains( str.substr(0, i) ) &&
                    wordBreak( str.substr(i, size-i) ))
                    return true;
        }

        // If we have tried all prefixes and
```

```
        // none of them worked
        return false;
}

// Driver program to test above functions
int main()
{
        wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
        wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
        wordBreak("")? cout <<"Yes\n": cout << "No\n";
        wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
        return 0;
}
```

Output

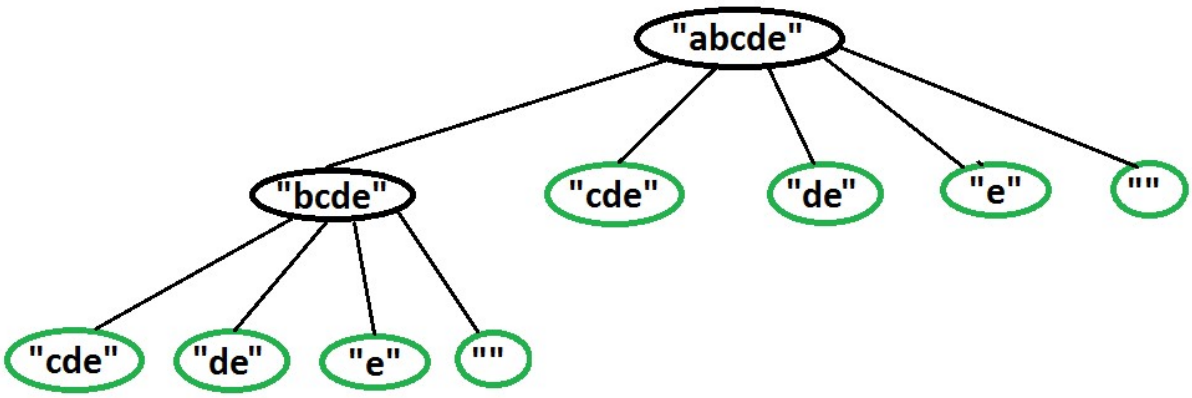
```
Yes
Yes
Yes
Yes
Yes
No
```

Time Complexity: The time complexity of the above code will be O(2^n).
Auxiliary Space: The space complexity will be O(n) as we are using recursion and the recursive call stack will take O(n) space.

Dynamic Programming

Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string

“abcde” in the worst case.



Partial recursion tree for input string "abcde". The subproblems encircled with green color are overlapping subproblems

```
// A Dynamic Programming based program to test whether a given string can
// be segmented into space separated words in dictionary
#include <iostream>
#include <string.h>
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
An array of strings is used for dictionary. Using array of strings for
dictionary is definitely not a good idea. We have used for simplicity of
the program*/
int dictionaryContains(string word)
{
        string dictionary[] = {"mobile","samsung","sam","sung","man","mango",
                                                "icecream","and","go","i","like","ice","cream"};
        int size = sizeof(dictionary)/sizeof(dictionary[0]);
        for (int i = 0; i < size; i++)
                if (dictionary[i].compare(word) == 0)
```

```
                return true;
        return false;
}

// Returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
        int size = str.size();
        if (size == 0) return true;

        // Create the DP table to store results of subproblems. The value wb[i]
        // will be true if str[0..i-1] can be segmented into dictionary words,
        // otherwise false.
        bool wb[size+1];
        memset(wb, 0, sizeof(wb)); // Initialize all values as false.

        for (int i=1; i<=size; i++)
        {
                // if wb[i] is false, then check if current prefix can make it true.
                // Current prefix is "str.substr(0, i)"
                if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
                        wb[i] = true;

                // wb[i] is true, then check for all substrings starting from
                // (i+1)th character and store their results.
                if (wb[i] == true)
                {
                        // If we reached the last prefix
```

```
                if (i == size)
                        return true;

                for (int j = i+1; j <= size; j++)
                {
                        // Update wb[j] if it is false and can be updated
                        // Note the parameter passed to dictionaryContains() is
                        // substring starting from index 'i' and length 'j-i'
                        if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
                                wb[j] = true;

                        // If we reached the last character
                        if (j == size && wb[j] == true)
                                return true;
                }
        }
    }

    /* Uncomment these lines to print DP table "wb[]"
    for (int i = 1; i <= size; i++)
        cout << " " << wb[i]; */

    // If we have tried all prefixes and none of them worked
    return false;
}

// Driver program to test above functions
int main()
{
```

```
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}
```

The time complexity of the given implementation of the wordBreak function is O(n^3), where n is the length of the input string.
The space complexity of the implementation is O(n), as an extra boolean array of size n+1 is created. Additionally, the dictionary array occupies a space of O(kL), where k is the number of words in the dictionary and L is the maximum length of a word in the dictionary. However, since the dictionary is a constant and small-sized array, its space complexity can be considered negligible. Therefore, the overall space complexity of the implementation is O(n).

Optimized Dynamic Programming:
In this approach, apart from the dp table, we also maintain all the indexes which have matched earlier. Then we will check the substrings from those indexes to the current index. If anyone of that matches then we can divide the string up to that index.
In this program, we are using some extra space. However, its time complexity is O(n*n) if n>s or O(n*s) if s>n where s is the length of the largest string in the dictionary and n is the length of the given string.

```
// A Dynamic Programming based program to test
// whether a given string can be segmented into
// space separated words in dictionary
#include <bits/stdc++.h>
using namespace std;
```

```
/* A utility function to check whether a word
is present in dictionary or not. An array of
strings is used for dictionary. Using array
of strings for dictionary is definitely not
a good idea. We have used for simplicity of
the program*/
int dictionaryContains(string word)
{
	string dictionary[]
		= { "mobile", "samsung", "sam", "sung", "man",
			"mango", "icecream", "and", "go", "i",
			"like", "ice",	"cream" };
	int size = sizeof(dictionary) / sizeof(dictionary[0]);
	for (int i = 0; i < size; i++)
		if (dictionary[i].compare(word) == 0)
			return true;
	return false;
}

// Returns true if string can be segmented into space
// separated words, otherwise returns false
bool wordBreak(string s)
{
	int n = s.size();
	if (n == 0)
		return true;

	// Create the DP table to store results of subproblems.
	// The value dp[i] will be true if str[0..i] can be
```

```
// segmented into dictionary words, otherwise false.
vector<bool> dp(n + 1, 0); // Initialize all values
// as false.

// matched_index array represents the indexes for which
// dp[i] is true. Initially only -1 element is present
// in this array.
vector<int> matched_index;
matched_index.push_back(-1);

for (int i = 0; i < n; i++) {
        int msize = matched_index.size();

        // Flag value which tells that a substring matches
        // with given words or not.
        int f = 0;

        // Check all the substring from the indexes matched
        // earlier. If any of that substring matches than
        // make flag value = 1;
        for (int j = msize - 1; j >= 0; j--) {

                // sb is substring starting from
                // matched_index[j]
                // + 1 and of length i - matched_index[j]
                string sb = s.substr(matched_index[j] + 1,
                                                        i - matched_index[j]);

                if (dictionaryContains(sb)) {
```

```
                        f = 1;
                        break;
                    }
                }

                // If substring matches than do dp[i] = 1 and
                // push that index in matched_index array.
                if (f == 1) {
                    dp[i] = 1;
                    matched_index.push_back(i);
                }
            }
            return dp[n - 1];
}

// Driver code
int main()
{
            wordBreak("ilikesamsung") ? cout << "Yes\n"
                                                    : cout << "No\n";
            wordBreak("iiiiiiii") ? cout << "Yes\n"
                                                : cout << "No\n";
            wordBreak("") ? cout << "Yes\n" : cout << "No\n";
            wordBreak("ilikelikeimangoiii") ? cout << "Yes\n"
                                                                : cout << "No\n";
            wordBreak("samsungandmango") ? cout << "Yes\n"
                                                            : cout << "No\n";
            wordBreak("samsungandmangok") ? cout << "Yes\n"
                                                            : cout << "No\n";
```

```
    return 0;
}
```

Word Break Problem | (Trie solution)

Exercise:

The above solutions only find out whether a given string can be segmented or not. Extend the above Dynamic Programming solution to print all possible partitions of input string.

Examples:

```
Input: ilikeicecreamandmango
Output:
i like ice cream and man go
i like ice cream and mango
i like icecream and man go
i like icecream and mango
Input: ilikesamsungmobile
Output:
i like sam sung mobile
i like samsung mobile
```

Word Break Problem | (Hashmap solution):

In this approach first, we are storing all the words in a Hashmap. after that, we traverse the input string and check if there is a match or not.

- C++
- Java
- Python3
- C#
- Javascript

```cpp
#include<bits/stdc++.h>
using namespace std;
bool CanParseUtil(unordered_map<string, bool>mp, string word)
{
    // if the size id zero that means we completed the word. so we can return true
    int size = word.size();
    if(size == 0)
    {
        return true;
```

```
    }
    string temp = "";
    for(int i = 0; i < word.length(); i++)
    {
        temp += word[i];
        // if the temp exist in hashmap and the parsing operation of the remaining word
true, we can return true.
        if(mp.find(temp) != mp.end() && CanParseUtil(mp, word.substr(i+1)))
        {
            return true;
        }
    }
    // if there is a mismatch in the dictionary, we can return false.
    return false;
}
string CanParse(vector<string>words, string word)
{
    int start = 0;
    // store the words in the hashmap
    unordered_map<string, bool>mp;
    for(auto it : words)
    {
        mp[it] = true;
    }
    return CanParseUtil(mp,word ) == true ? "YES" : "NO";
}


int main() {
    vector<string>words{"mobile","samsung","sam","sung",
                            "man","mango","icecream","and",
                             "go","i","like","ice","cream"};
    string word = "samsungandmangok";
    cout << CanParse(words, word) << endl;
}
```


Output

NO