Saturday, March 29, 2025 3:08 AM

Word Break :-

Word Break

Difficulty: **Medium** Accuracy: **40.86%** Submissions: **152K+** Points: **4**

You are given a string **s** and a list **dictionary[]** of words. Your task is to determine whether the string **s** can be formed by concatenating one or more words from the **dictionary[]**.

Note: From **dictionary[]**, any word can be taken any number of times and in any order.

Examples :

Input: s = "ilike", dictionary[] = ["i", "like", "gfg"]
Output: true
Explanation: s can be breakdown as "i like".

Input: s = "ilikegfg", dictionary[] = ["i", "like", "man", "india", "gfg"]
Output: true
Explanation: s can be breakdown as "i like gfg".

Input: s = "ilikemangoes", dictionary[] = ["i", "like", "man", "india", "gfg"]
Output: false
Explanation: s cannot be formed using dictionary[] words.

Constraints:
1 ≤ s.size() ≤ 3000
1 ≤ dictionary.size() ≤ 1000
1 ≤ dictionary[i].size() ≤ 100

Try more examples

```
bool solve( int i, unordered_set<int>s){
    if(i==n) return true;
    if(dp[i]!=-1) return dp[i];
    bool ans= false;   string s;
    for (int j= i; j<n; j++){
        s= s+st
        ans= ans | (s.count (st.substr(i,j+1
```

[Naive Approach] Using Recursion – $O(2^n)$ Time & $O(n)$ Space :-

The idea is to consider each prefix & search for it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, otherwise we try next prefix. If we have tried all prefixes, & none of them resulted in a soln, we return false.

```
bool wordBreakRec (int i, string& s, vector<string> dic){
    if(i==s.length()) return true;
    int n= s.length();
    string prefix= "";
    for(int j=i; j<n; j++){
        prefix+= s[j];
        if(find(dic.begin(), dic.end(), prefix) != dic.end()&&
            wordBreakRec(j+1, s, dic)) return true;
        }
        return false;
}

bool wordBreak (string & s, vector<string> dic ){
        return wordBreakRecur( 0, s, dic);
    }
```

Using Top-Down DP- $O(n^2)$ Time & $O(n+m)$ Space :-

1. Optimal Substructure
2. Overlapping Subproblems

```
vector<string> & dic, vector<int>&dp)
```

1. Optimal Substructure
2. Overlapping Subproblems

```
bool wordBreakRec(int ind, string &s, vector<string> &dic, vector<int>&dp)
{
    if(ind >= s.size()) return true;
    if(dp[ind] != -1) return dp[ind];
    bool possible = false;
    for(int i=0; i<dic.size(); i++){
        string temp = dic[i];
        if(temp.size() > s.size()-ind) continue;
        bool ok = true;
        for(int j=0; j<temp.size(); j++){
            if(temp[j] != s[k]){
                ok = false;
                break;
            }
            else
                k++;
        }
        if(ok){
            possible |= wordBreak(ind+temp.size(), s, dic, dp);
        }
    }
    return dp[ind] = possible; }

bool wordBreak(string s, vector<string> &dic){
    int n = s.size();
    vector<int> dp(n+1, -1);
    return wordBreakRec(0, s, dic, dp);
}
```

manual string matching

[Expected Approach-2] Using Bottom Up DP - O(n×m×k) time & O(n) space :-

The idea is to use bottom-up dynamic programming to determine if a string can be segmented into dictionary words. Create a boolean array dp[] where each position dp[i] represents whether the substring from 0 to that position can be broken into dictionary words.

Step by step approach :-

1. Start from the beginning of the string & mark it as valid (base case)
   i.e. dp[0] = true;
2. For each position, check if any dictionary word ends at that position & leads to

i.e. dp[0] = true;

2. For each position, check if any dictionary word ends at that position & leads to an already valid position.
3. If such a word exists, mark the current position as valid, i.e. dp[i]=true.
4. At the end return the last entry of dp[].

```
bool wordBreak(string &s, vector<string> &dictionary){
    int n= s.size();
    vector<bool> dp(n+1,0);
    dp[0] = 1;
  // Traverse through the given string
    for(int i=1; i<=n ;i++){
     //Traverse through the dictionary words
     for(string &w: dictionary){
     // check if current word is present
     // the prefix before the word is also
     // breakable
      int start = i-w.size();
      if( start >=0 && dp[start] && s.substr(start, w.size())==w){
              dp[i] = 1;
              break;
     }}}
     return dp[n];
    }
```

Important :-

```
class Solution {
  public:
    bool wordBreak(string &s, vector<string> &dictionary) {
        // code here
        unordered_set<string> st;
        for(auto &d:dictionary)st.insert(d);
        int n=s.size();
        vector<int> dp(n, -1);
        return solve(s, 0, st, dp);

    }
    bool solve(string &s, int i, unordered_set<string> st, vector<int> dp){
        int n=s.size();
        if(i==n)return 1;
        if(dp[i]!=-1)return dp[i];
        bool ans=0;
        string ss;
        for(int j=i;j<n;j++){
            ss=ss+s[j];
            ans=ans||(st.count(ss)>0)&&(solve(s, j+1, st, dp));
        }

        return dp[i]=ans;
    }
};
```

The Time Complexity of this approach is O(n^3) due to this part (each time new string is created). That's why we use manual string matching.