<https://www.lintcode.com/problem/829/>

**Description**

Given a pattern and a string str, find if str follows the same pattern.

Here **follow** means a full match, such that there is a [bijection](https://baike.baidu.com/item/%E5%8F%8C%E5%B0%84/942799?fr=aladdin) between a letter in pattern and a **non-empty** substring in str.(i.e if a corresponds to s, then b cannot correspond to s. For example, given pattern = "ab", str = "ss", return false.)

You may assume both pattern and str contains only lowercase letters.

**Example**

**Example 1**

Input:

pattern = "abab"

str = "redblueredblue"

Output: true

Explanation: "a"->"red","b"->"blue"

**Example 2**

Input:

pattern = "aaaa"

str = "asdasdasdasd"

Output: true

Explanation: "a"->"asd"

**Example 3**

Input:

pattern = "aabb"

str = "xyzabcxzyabc"

Output: false

**Attempt 1: 2022-12-24**

**Solution 1: Backtracking with additional Set to guarantee {char, string} mapping is unique (60 min)**

**Style 1: With return type**

public class Solution {

    /\*\*

    \* @param pattern: a string,denote pattern string

    \* @param str: a string, denote matching string

    \* @return: a boolean

    \*/

    public boolean wordPatternMatch(String pattern, String str) {

        Map<Character, String> map = new HashMap<Character, String>();

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

        return helper(map, set, pattern, 0, str, 0);

    }

    private boolean helper(Map<Character, String> map, Set<String> set, String pattern, int i, String str, int j) {

        if(i == pattern.length() && j == str.length()) {

            return true;

        }

        if(i == pattern.length() || j == str.length() || pattern.length() - i > str.length() - j) {

            return false;

        }

        char c = pattern.charAt(i);

        if(map.containsKey(c)) {

            String section = map.get(c);

            if(!str.startsWith(section, j)) {

                return false;

            }

            return helper(map, set, pattern, i + 1, str, j + section.length());

        } else {

            for(int k = j; k < str.length(); k++) {

                String candidate = str.substring(j, k + 1);

                if(set.contains(candidate)) {

                    continue;

                }

                map.put(c, candidate);

                set.add(candidate);

                if(helper(map, set, pattern, i + 1, str, k + 1)) {

                    return true;

                }

                map.remove(c);

                set.remove(candidate);

            }

        }

        return false;

    }

}

Time Complexity: O(N^M), or C(N^M) to be exact. Pattern length is M, str length is N

Space Complexity: O(M), Pattern length is M, str length is N. We use a map and a set to store the lookup,

but at one time, the map should not exceed the pattern size, so is the set

**Refer to**

<https://protegejj.gitbook.io/algorithm-practice/leetcode/backtracking/291-word-pattern-ii>

class Solution {

    public boolean wordPatternMatch(String pattern, String str) {

        Map<Character, String> map = new HashMap<>();

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

        return isMatched(pattern, 0, str, 0, set, map);

    }

    public boolean isMatched(String pattern, int patternIndex, String str, int strIndex, Set<String> set, Map<Character, String> map) {

        if (patternIndex == pattern.length() && strIndex == str.length()) {

            return true;

        }

        // 没有足够的部分完成匹配

        if (patternIndex == pattern.length() || strIndex == str.length()

            || str.length() - strIndex < pattern.length() - patternIndex) {

            return false;

        }

        char c = pattern.charAt(patternIndex);

        // 如果当前字符c已经有匹配的string

        if (map.containsKey(c)) {

            String matchedStr = map.get(c);

            // 如果string剩余的部分并非以已匹配的string开头的话，则pattern不match

            if (!str.startsWith(matchedStr, strIndex)) {

                return false;

            }

            return isMatched(pattern, patternIndex + 1, str, strIndex + matchedStr.length(), set, map);

        }

        // 如果当前字符c目前没有匹配的string, 尝试match各种可能的string

        else {

            for (int i = strIndex; i < str.length(); i++) {

                String candidate = str.substring(strIndex, i + 1);

                // 利用HashSet防止不同的字符对应一样的string

                if (set.contains(candidate)) {

                    continue;

                }

                set.add(candidate);

                map.put(c, candidate);

                if (isMatched(pattern, patternIndex + 1, str, i + 1, set, map)) {

                    return true;

                }

                map.remove(c);

                set.remove(candidate);

            }

            return false;

        }

    }

}

时间复杂度O(n \* C(m, n)),n是str的长度, m为pattern的长度，这个问题相当于将str分成m份，而分的方法是C(n, m),每部分都要O(n)时间验证, 空间复杂度O(m + n + n) => O(m + n), 递归的所需空间最多为n, set的空间最多为n, map的空间最多为n

**Refer to**

<https://blog.baozitraining.org/2019/07/leetcode-solution-291-word-pattern-ii.html>

**Thought Process**

It is quite similar to [Word Pattern](https://blog.baozitraining.org/2019/06/leetcode-solution-290-word-pattern.html) and [Isomorphic String](https://blog.baozitraining.org/2019/07/leetcode-solution-205-isomorphic-strings.html) problem, where we would keep a mapping from char to a string while also ensure there would be a one to one mapping, i.e., bijection mapping. The tricky part is it seems there are way many combinations of the mapping, how can we efficiently solve them?

Maybe we could list all the combinations? Maybe we could use DP since it is string related and only ask for true/false result?

**How to list all combinations? Think about this way, let's say you have pattern = "aba" and str = "redbluered", since one char in pattern can map to any string length >= 1 in str, it is equivalent to divide up str into 3 parts (length of pattern) and check all cases.**

For instance, the cut of the words is like below:

r | e | d b l u e r e d

r | e d | b l u e r e d

r | e d b | l u e r e d

r | e d b l | u e r e d

r | e d b l u | e r e d

r | e d b l u e | r e d

r | e d b l u e r | e d

r | e d b l u e r e | d

r e | d | b l u e r e d

r e | d b | l u e r e d

r e | d b l | u e r e d

r e | d b l u | e r e d

r e | d b l u e | r e d

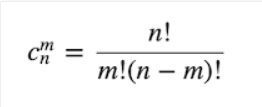
r e | d b l u e r | e d

r e | d b l u e r e | d

r e d | b | l u e r e d

.....

In general, if the length of pattern is M, the str length is N, **we try to divide N length string into M parts, the time complexity of this brute force method is O(N^M), more accurately, it should be C(n, m)**

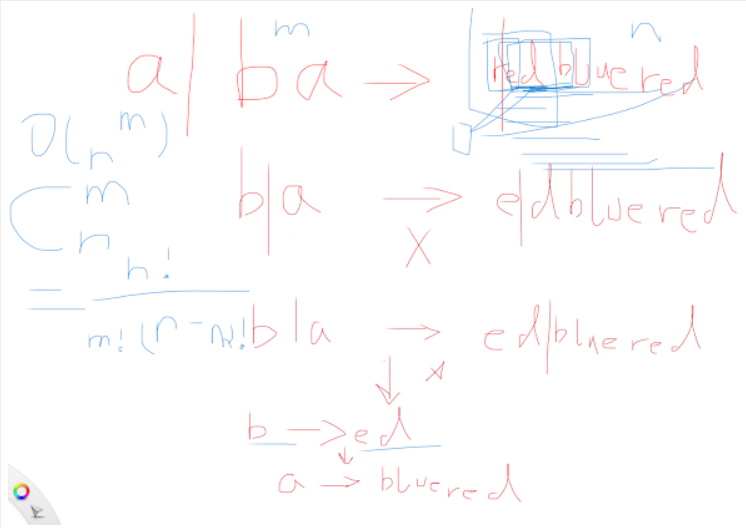


DP solution does not work since we cannot easily get a deduction formula :(

**Solutions**

**Brute force list all the combos**

For each character in pattern, try to map any possible remaining strings in str from length 1 to the end. During this process, need to make sure the string mapping is bijection (no two chars in pattern map to the same string in str) and if a mapping has been seen before, continue use that mapping.



A DFS recursion would be the implementation. A few caveats in implementation

Remember to reset the map and set after recursion returned false

When there is a bijection mapping, should continue instead of directly break

Time Complexity: O(N^M), or C(N^M) to be exact. Pattern length is M, str length is N

Space Complexity: O(M), Pattern length is M, str length is N. We use a map and a set to store the lookup, but at one time, the map should not exceed the pattern size, so is the set

**Style 2: Void return but with global variable**

public class Solution {

    /\*\*

    \* @param pattern: a string,denote pattern string

    \* @param str: a string, denote matching string

    \* @return: a boolean

    \*/

    boolean result = false;

    public boolean wordPatternMatch(String pattern, String str) {

        Map<Character, String> map = new HashMap<Character, String>();

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

        helper(map, set, pattern, 0, str, 0);

        return result;

    }

    private void helper(Map<Character, String> map, Set<String> set, String pattern, int i, String str, int j) {

        if(i == pattern.length() && j == str.length()) {

            result = true;

            return;

        }

        if(i == pattern.length() || j == str.length() || pattern.length() - i > str.length() - j) {

            return;

        }

        char c = pattern.charAt(i);

        if(map.containsKey(c)) {

            String section = map.get(c);

            if(!str.startsWith(section, j)) {

                return;

            }

            helper(map, set, pattern, i + 1, str, j + section.length());

        } else {

            for(int k = j; k < str.length(); k++) {

                String candidate = str.substring(j, k + 1);

                if(set.contains(candidate)) {

                    continue;

                }

                map.put(c, candidate);

                set.add(candidate);

                helper(map, set, pattern, i + 1, str, k + 1);

                map.remove(c);

                set.remove(candidate);

            }

        }

    }

}

Time Complexity: O(N^M), or C(N^M) to be exact. Pattern length is M, str length is N

Space Complexity: O(M), Pattern length is M, str length is N. We use a map and a set to store the lookup,

but at one time, the map should not exceed the pattern size, so is the set

**Refer to**

<https://segmentfault.com/a/1190000003827151>

**回溯法**

**复杂度**

**Time Complexity**

O(NM) Where

N is total number of characters in an input pattern

M is total number of characters in an input string

**Space Complexity**

O(N + M) Where

N is total number of characters in an input pattern

M is total number of characters in an input string

**思路**

因为目标字符串可以任意划分，所以我们不得不尝试所有可能性。这里通过深度优先搜索的回溯法，对于pattern中每个字母，在str中尝试所有的划分方式，如果划分出来的子串可以用这个字母映射，或者可以建立一个新的字母和字符串的映射关系，我们就继续递归判断下一个pattern中的字母。

**代码**

public class Solution {

    Map<Character, String> map;

    Set<String> set;

    boolean res;

    public boolean wordPatternMatch(String pattern, String str) {

        // 和I中一样，Map用来记录字符和字符串的映射关系

        map = new HashMap<Character, String>();

        // Set用来记录哪些字符串被映射了，防止多对一映射

        set = new HashSet<String>();

        res = false;

        // 递归回溯

        helper(pattern, str, 0, 0);

        return res;

    }

    public void helper(String pattern, String str, int i, int j){

        // 如果pattern匹配完了而且str也正好匹配完了，说明有解

        if(i == pattern.length() && j == str.length()){

            res = true;

            return;

        }

        // 如果某个匹配超界了，则结束递归

        if(i >= pattern.length() || j >= str.length()){

            return;

        }

        char c = pattern.charAt(i);

        // 尝试从当前位置到结尾的所有划分方式

        for(int cut = j + 1; cut <= str.length(); cut++){

            // 拆出一个子串

            String substr = str.substring(j, cut);

            // 如果这个子串没有被映射过，而且当前pattern的字符也没有产生过映射

            // 则新建一对映射，并且继续递归求解

            if(!set.contains(substr) && !map.containsKey(c)){

                map.put(c, substr);

                set.add(substr);

                helper(pattern, str, i + 1, cut);

                map.remove(c);

                set.remove(substr);

            // 如果已经有映射了，但是是匹配的，也继续求解

            } else if(map.containsKey(c) && map.get(c).equals(substr)){

                helper(pattern, str, i + 1, cut);

            }

            // 否则跳过该子串，尝试下一种拆分

        }

    }

}