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

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

Here **follow** means a full match, such that there is a bijection between a letter in pattern and a **non-empty** word in s.

**Example 1:**

Input: pattern = "abba", s = "dog cat cat dog"

Output: true

**Example 2:**

Input: pattern = "abba", s = "dog cat cat fish"

Output: false

**Example 3:**

Input: pattern = "aaaa", s = "dog cat cat dog"

Output: false

**Constraints:**

1 <= pattern.length <= 300

pattern contains only lower-case English letters.

1 <= s.length <= 3000

s contains only lowercase English letters and spaces ' '.

s **does not contain** any leading or trailing spaces.

All the words in s are separated by a **single space**.

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

**Wrong Solution**

class Solution {

    public boolean wordPattern(String pattern, String s) {

        String[] strs = s.split("\\s+");

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

        for(int i = 0; i < pattern.length(); i++) {

            char c = pattern.charAt(i);

            // If no current {key, value} mapping, the {key, value} mapping should be the first

            if(!map.containsKey(c)) {

                // The wrong condition is here, if pattern = "abba",

                // str = "dog dog dog dog", when put ('b', "dog") on

                // map, it will surely return null, but "dog" is

                // occupied by another projection ('a', "dog"), the

                // right expression should reflect miss projection

                // such as map.containsValue(a[i])

                if(map.put(c, strs[i]) != null) {

                    return false;

                }

            // If already have current {key, value} mapping, the {key, value} mapping must has the same value

            } else {

                if(!map.put(c, strs[i]).equals(strs[i])) {

                    return false;

                }

            }

        }

        return true;

    }

}

**Solution 1:  HashTable (10 min)**

class Solution {

    public boolean wordPattern(String pattern, String s) {

        String[] strs = s.split("\\s+");

        if(strs.length != pattern.length()) {

            return false;

        }

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

        for(int i = 0; i < pattern.length(); i++) {

            char c = pattern.charAt(i);

            // If no current {key, value} mapping, the {key, value} mapping should be the first

            if(!map.containsKey(c)) {

                // But if already contains the projection value of this key,

                // which means the value match another key not relate as

                // one on one projection.

                // E.g pattern = "abba", str = "dog dog dog dog",

                // First putting ('a', "dog") on map, key = 'b' not contain

                // in map now, when we try to put ('b', "dog") on map, we

                // find "dog" already on map and is projection of 'a', this

                // violate one(key) on one(value) projection, so return false

                if(map.containsValue(strs[i])) {

                    return false;

                }

                map.put(c, strs[i]);

            // If already have current {key, value} mapping, the {key, value} mapping must has the same value

            } else {

// Don't write condition as 'if(map.get(c).equals(strs[i]))'

// Test out by:

// pattern = "abba", s = "dog cat cat dog"

// Output = false, Expected = true

// Why need map.put(key, value) method ?

// Because map.put(key, value) will return the previous value associated with key,

// which means previous value for this key equals current value, then confirm the

// {key, value} pair is unique

                if(!map.put(c, strs[i]).equals(strs[i])) {

                    return false;

                }

            }

        }

        return true;

    }

}

Time Complexity : O(N)

Space Complexity : O(N)

**Refer to Deepseek**

To solve Leetcode 290, "Word Pattern," we need to determine if a given string of words follows a specific pattern where each character in the pattern corresponds to a unique word in the string and vice versa. This is similar to the isomorphic strings problem but involves words instead of individual characters.

**Two Maps**

**Approach**

Split the String: First, split the input string into an array of words using the split method.

Check Lengths: Ensure the number of words matches the length of the pattern. If not, return false immediately.

Use Two Maps: Utilize two hash maps to maintain the mappings:

One map (charToWord) to map each character in the pattern to its corresponding word.

Another map (wordToChar) to map each word back to its corresponding character in the pattern.

\*\*Check Mappings\*\*: Iterate through each character in the pattern and the corresponding word:

If the character is not in charToWord and the word is not in wordToChar, add both mappings.

If either the character or the word is already mapped but does not match the current pair, return false.

\*\*Return Result\*\*: If all characters and words match correctly, return true.

**Solution Code**

import java.util.HashMap;

import java.util.Map;

class Solution {

public boolean wordPattern(String pattern, String s) {

String[] words = s.split(" ");

if (words.length != pattern.length()) {

return false;

}

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

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

for (int i = 0; i < pattern.length(); i++) {

char c = pattern.charAt(i);

String word = words[i];

if (!charToWord.containsKey(c)) {

if (wordToChar.containsKey(word)) {

return false;

}

charToWord.put(c, word);

wordToChar.put(word, c);

} else {

if (!charToWord.get(c).equals(word)) {

return false;

}

}

}

return true;

}

}

**Explanation**

Splitting the String: The input string s is split into an array of words using split(" "), which splits the string at each space.

Length Check: The lengths of the pattern and the array of words must match; otherwise, they cannot follow the same pattern.

Mapping Characters to Words: The charToWord map ensures each character in the pattern maps to exactly one word.

Mapping Words to Characters: The wordToChar map ensures each word maps back to exactly one character in the pattern, maintaining bijection.

Validation: During iteration, if a character or word is already mapped but does not correspond to the current pair, the function returns false immediately. If all checks pass, it returns true.

This approach efficiently checks for the bijective relationship between the pattern and the words, ensuring correctness with optimal performance.

**Single Map**

Yes, we can solve the problem using just one map, but we need to ensure that the mapping is bijective (both one-to-one and onto). Here's how we can do it:

**Approach**

Split the String: Split the input string s into an array of words.

Check Lengths: Ensure the number of words matches the length of the pattern. If not, return false.

Single Map for Mapping: Use a single map to map characters in the pattern to words in the string.

Check for Bijection: While mapping, ensure that:

No two different characters in the pattern map to the same word.

No two different words are mapped by the same character.

This can be done by checking if the word is already mapped to any character other than the current one.

**Solution Code**

import java.util.HashMap;

import java.util.Map;

class Solution {

public boolean wordPattern(String pattern, String s) {

String[] words = s.split(" ");

if (words.length != pattern.length()) {

return false;

}

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

for (int i = 0; i < pattern.length(); i++) {

char c = pattern.charAt(i);

String word = words[i];

if (charToWord.containsKey(c)) {

if (!charToWord.get(c).equals(word)) {

return false;

}

} else {

if (charToWord.containsValue(word)) {

return false;

}

charToWord.put(c, word);

}

}

return true;

}

}

**Explanation**

Splitting the String: The input string s is split into an array of words using split(" ").

Length Check: The lengths of the pattern and the array of words must match; otherwise, they cannot follow the same pattern.

Single Map Usage: The charToWord map is used to store the mapping from characters in the pattern to words in the string.

Bijection Check:

If a character is already mapped, we check if the current word matches the mapped word.

If a character is not mapped, we check if the current word is already mapped to any other character using containsValue. If it is, we return false to maintain bijection.

This approach efficiently checks for the bijective relationship using a single map, ensuring correctness with optimal performance. The time complexity is O(n), where n is the length of the pattern, and the space complexity is O(n) to store the mappings.

**Refer to**

[L205.Isomorphic Strings (Ref.L290,L890)](note://WEB87f5bda440967963b486a4c5368709a7)

[L890.Find and Replace Pattern (Ref.L205,L290)](note://WEBfbeaed37e3a76872fd1d3ab511ca5882)