290. Word Pattern

String

## **Problem Description**

Hash Table

Easy

letter in pattern corresponds to a non-empty word in s, and the relationship between the letters and the words must be a bijection. This means every character in the pattern should map to a uniquely associated word in s and vice versa, with no two letters mapping to the same word and no two words mapping to the same letter.

The problem presents the task of determining if a given string s follows the same pattern as given by a string pattern. Each

Intuition The key to solving this problem is to maintain a mapping between the characters of the pattern and the words in s. We can achieve this by using two hash tables: one table (d1) to map characters to words and another (d2) to map words to characters.

not, the pattern cannot be followed, and we can return false immediately. After checking this length criterion, we proceed to iterate over the characters and words in parallel. For every character-word pair, we check:

We start by splitting the string s into words. The number of words in s should match the number of characters in the pattern; if

2. Conversely, if the current word is already mapped to a different character in d2, the pattern is also broken.

**Solution Approach** 

The solution follows a straightforward approach using hash tables. Here's how it works, step by step:

1. If the current character is already mapped to a different word in d1, the pattern is broken.

meaning s does not follow the pattern, and we return false.

have a conflict, and we return false.

the string s does follow the given pattern.

### We begin by splitting the input string s using the split() method, which returns a list of words. This list is stored in a variable ws.

We immediately check if the number of words in ws is equal to the length of the pattern. If there's a mismatch, we return false as it's not possible for s to follow the pattern if the number of elements doesn't match.

We declare two dictionaries, d1 and d2. d1 will keep track of the letter-to-word mappings, and d2 will keep track of word-

- to-letter mappings. They help us verify that each letter in pattern maps to exactly one word in s and each word in s maps to exactly one letter in pattern. We use the built-in zip function to iterate over the characters in pattern and the words in ws simultaneously. For each
- character a and word b: We check if a is already a key in d1. If a is already mapped to a word, and that word is not b, we have a conflict,
- If neither of the above conflicts arises, we map character a to word b in d1 and word b to character a in d2. If we can iterate over all character-word pairs without encountering any conflicts, the function returns true, indicating that

Similarly, we check if b is already a key in d2. If b is already mapped to a character, and that character is not a, we again

- This approach cleverly uses the properties of hash tables: constant time complexity for insertions and lookups (on average). By maintaining two separate mappings and checking for existing mappings at each step, we ensure a bijection between the
- characters of pattern and the words in s. This algorithm has a time complexity of O(n), where n is the number of characters in pattern or the number of words in s, assuming that the hash operations are constant time.

To illustrate the solution approach, let's consider an example where the pattern string is "abba" and the string s is "dog cat cat dog".

2. Next, we check whether the length of ws matches the length of pattern. Here, they both have a length of 4, so we can

We begin by splitting the string s into words, which gives us ws = ["dog", "cat", "cat", "dog"].

## We initialize two dictionaries: $d1 = \{\}$ and $d2 = \{\}$ .

∘ (a, dog)

proceed with the mapping process.

o d1 = {'a': 'dog', 'b': 'cat'}

o d2 = {'dog': 'a', 'cat': 'b'}

Solution Implementation

words = str\_sequence.split()

if len(pattern) != len(words):

for char, word in zip(pattern, words):

word\_to\_char\_map[word] = char

// If no inconsistencies are found, return true

// Determines if a pattern matches the words in a string

// Utilize istringstream to split the string into words

// If the number of pattern characters and words do not match, return false

// Create mappings to keep track of the pattern to word relationships

bool wordPattern(string pattern, string str) {

// Splitting the string by whitespaces

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

unordered map<char, string> patternToWord;

unordered\_map<string, char> wordToPattern;

istringstream strStream(str);

while (strStream >> word) {

words.push\_back(word);

vector<string> words;

return false;

string word;

return False

**Example Walkthrough** 

∘ (b, cat) ∘ (b, cat) ∘ (a, dog)

For the third pair (b, cat), b is already in d1 and it maps to cat, and cat is in d2 and it maps to b, so no conflict occurs.

Since we have iterated through all character-word pairs without encountering any conflicts, we can conclude that the string s

Start iterating through the pairs generated by zip("abba", ["dog", "cat", "cat", "dog"]), which gives us:

For the first pair (a, dog), a is not a key in d1 and dog is not a key in d2, so we add them to our dictionaries:

o d1 = {'a': 'dog'} o d2 = {'dog': 'a'}

def wordPattern(self, pattern: str, str sequence: str) -> bool:

# Split the input string on whitespace to get individual words

# Iterate over the pattern and the corresponding words together

# If the character is already mapped to a different word,

# or the word is already mapped to a different character, return False

if (char in char to word map and char to word map[char] != word) or \

(word in word\_to\_char\_map and word\_to\_char\_map[word] != char):

# If the pattern length and word count are different, they don't match

Finally, for the fourth pair (a, dog), a is already in d1 and it maps to dog, and dog is in d2 and it maps to a, so again no conflict occurs.

does indeed follow the given pattern. Therefore, our function should return true for this example.

For the second pair (b, cat), b is not a key in d1 and cat is not a key in d2, so we likewise add them:

**Python** class Solution:

# Initialize dictionaries to store character-to-word map and word-to-character map char to word map =  $\{\}$ word\_to\_char\_map = {}

#### return False # Add the mappings to both dictionaries char to word map[char] = word

return true;

#include <sstream>

#include <vector>

#include <string>

class Solution {

#include <unordered\_map>

```
# If no mismatches are found, pattern and words match — return True
        return True
Java
class Solution {
    public boolean wordPattern(String pattern, String s) {
        // Split the s string into individual words
        String[] words = s.split(" ");
        // If the number of characters in the pattern does not match the number of words, return false
        if (pattern.length() != words.length) {
            return false;
        // Initialize two dictionaries to track the mappings from characters to words and vice versa
        Map<Character, String> charToWordMap = new HashMap<>();
        Map<String, Character> wordToCharMap = new HashMap<>();
        // Iterate over the pattern
        for (int i = 0; i < words.length; ++i) {</pre>
            char currentChar = pattern.charAt(i);
            String currentWord = words[i];
            // If the current mapping from char to word or word to char does not exist or is inconsistent, return false
            if (!charToWordMap.getOrDefault(currentChar, currentWord).equals(currentWord) || wordToCharMap.getOrDefault(currentWord,
                return false;
            // Update the mappings
            charToWordMap.put(currentChar, currentWord);
            wordToCharMap.put(currentWord, currentChar);
```

C++

public:

```
// Iterate through the pattern and corresponding words
        for (int i = 0; i < words.size(); ++i) {</pre>
            char patternChar = pattern[i];
            string currentWord = words[i];
            // Check if the current pattern character has already been mapped to a different word
            // or the current word has been mapped to a different pattern character
            if ((patternToWord.count(patternChar) && patternToWord[patternChar] != currentWord) ||
                (wordToPattern.count(currentWord) && wordToPattern[currentWord] != patternChar)) {
                return false;
            // Map the current pattern character to the current word and vice versa
            patternToWord[patternChar] = currentWord;
            wordToPattern[currentWord] = patternChar;
        // If all pattern characters and words match up, return true
        return true;
};
TypeScript
// Checks if a string pattern matches a given string sequence.
// Each letter in the pattern corresponds to a word in the s string.
function wordPattern(pattern: string, s: string): boolean {
    // Split the string s into an array of words.
    const words = s.split(' ');
    // If the number of elements in the pattern does not match the number of words, return false.
    if (pattern.length !== words.length) {
        return false;
    // Initialize two maps to store the character-to-word and word-to-character correspondences.
    const charToWordMap = new Map<string, string>();
    const wordToCharMap = new Map<string, string>();
    // Iterate over the pattern.
    for (let i = 0: i < pattern.length: ++i) {</pre>
        const char = pattern[i]; // Current character from the pattern.
        const word = words[i]; // Current word from the string.
        // Check if the current character is already associated with a different word.
        if (charToWordMap.has(char) && charToWordMap.get(char) !== word) {
            return false; // Mismatch found, return false.
        // Check if the current word is already associated with a different character.
        if (wordToCharMap.has(word) && wordToCharMap.get(word) !== char) {
            return false; // Mismatch found, return false.
        // Add the current character-to-word and word-to-character association to the maps.
        charToWordMap.set(char, word);
        wordToCharMap.set(word, char);
    // If no mismatch was found, return true.
    return true;
class Solution:
```

## The function wordPattern checks if a string follows a specific pattern. The time complexity and space complexity analysis is as follows:

Time and Space Complexity

return True

def wordPattern(self, pattern: str, str sequence: str) -> bool:

words = str\_sequence.split()

return False

char to word map = {}

word\_to\_char\_map = {}

if len(pattern) != len(words):

return False

for char. word in zip(pattern, words):

char to word map[char] = word

word\_to\_char\_map[word] = char

# Add the mappings to both dictionaries

# Split the input string on whitespace to get individual words

# Iterate over the pattern and the corresponding words together

# If the character is already mapped to a different word.

# If no mismatches are found, pattern and words match - return True

# If the pattern length and word count are different, they don't match

# Initialize dictionaries to store character-to-word map and word-to-character map

# or the word is already mapped to a different character, return False

if (char in char to word map and char to word map[char] != word) or \

(word in word\_to\_char\_map and word\_to\_char\_map[word] != char):

The time complexity of the code is O(N) where N is the length of the longer of the two inputs: the pattern and the s.split() list. s.split() operation itself is 0(n) where n is the length of the string s, as it must traverse the string once and create a list of words.

## The zip(pattern, ws) operation will iterate over the pairs of characters in pattern and words in ws, and the number of iterations will be the lesser of the two lengths. However, since we've already ensured both lengths are equal, it results in

**Time Complexity** 

min(len(pattern), len(ws)) operations, which in this case is len(pattern) or equivalently len(ws). Inside the loop, the operations involve checking membership and equality in two dictionaries, d1 and d2. Dictionary membership and assignment are average case 0(1) operations due to hash table properties. Combining these operations results in the total time complexity being O(N), where N = max(m, n) (the longer of the pattern's

length m and the split string list ws length n). But since the function only works when m == n, you can simplify the statement to just O(n) where n is the length of pattern or ws.

# **Space Complexity**

The space complexity of the function is also O(N) where N = m + n, the size of the input pattern plus the size of the list ws, which comes from the split() of string s.

Provided that both pattern and sisplit() are of the same length due to the early return conditional, the space complexity simplifies to O(n), where n is the length of pattern or ws.

The two dictionaries d1 and d2 will store at most m keys (unique characters in the pattern) and n keys (unique words in s).