291. Word Pattern II

Medium | Hash Table | String | Backtracking

Leetcode Link

Problem Description

In this problem, we're given a `pattern`, which is a string consisting of characters, and a string `s`. Our goal is to determine if the string `s` matches the given `pattern`.

A string `s` is considered to match a `pattern` if we can associate each character in `pattern` with a non-empty string in `s` such that the concatenation of the strings, in the order they appear in `pattern`, exactly makes up `s`. The mapping must be bijective, which means two conditions must be met: first, no two different characters in `pattern` can map to the same substring in `s`, and second, each character must map to exactly one substring (and not multiple different substrings at different instances).

For example, if `pattern` is "abab" and `s` is "redblueredblue" then we could map 'a' to "red" and 'b' to "blue", which matches the pattern.

Intuition

Approaching the solution requires a way to test different mappings from pattern characters to substrings of `s`. Since the problem's constraints don't offer an obvious pattern or formula, we opt for a strategy that tries out different possibilities—this is a typical case for using a backtracking algorithm.

The intuition behind the solution is to recursively attempt to build a mapping between each character in `pattern` and the substrings of `s`. We start by considering the first character in the `pattern` and try to map it to all possible prefixes of the string `s`. For each mapping we consider, we recursively attempt to extend the mapping to the rest of the characters in the `pattern`. If at any point, we find a complete and valid mapping, we return `true`.

If we ever reach a situation where a character is already mapped to a different string, or if we try to associate a substring with a pattern character that is already taken by another substring, we backtrack. We also stop exploring the current recursive branch if we exhaust the characters in `pattern` or `s`, or if the remaining length of `s` is too short to cover the remaining characters in `pattern`. This process continues until either a full mapping is found or all possibilities have been exhausted, which would lead us to conclude that no valid mapping exists.

Solution Approach

The provided solution uses a recursive backtracking approach to implement the logic described in the intuition section. Let's examine the implementation in detail:

1. **Depth-First Search (DFS) with Backtracking**:
   - The `dfs` function is a recursive function that keeps track of the current position `i` in the `pattern` and the current position `j` in the string `s`.
   - If `i` equals the length of `pattern` (`m`) and `j` equals the length of `s` (`n`), it means we have successfully mapped the entire pattern to the string `s`, and we return `True`.
   - If either `i` equals `m` or `j` equals `n` before the other, or if there aren't enough characters left in `s` to match the remaining pattern (`n - j < m - i`), we return `False` as it signifies that the current mapping does not cover `s` or `pattern` correctly.
2. **Attempt to Match Substrings**:
   - The `dfs` function iterates over the substring of `s` starting from index `j` and ending at various points `k` within the string. This loop essentially tries to map the current pattern character to various lengths of substrings in `s`.
   - We extract the current substring of `s` from index `j` to `k` and store it in a variable `t`.
   - If the current pattern character at index `i` already has a mapping in the dictionary `d` and it is equal to `t`, we recursively call `dfs` for the next character in the pattern and the next part of the string. If this returns `True`, then the current mapping is part of a solution, and we propagate this success back up the recursion chain.
3. **Adding New Mappings**:
   - If the current pattern character at index `i` is not yet mapped in the dictionary `d` and the substring `t` is not already used (checked using set `vis`), we add the mapping `pattern[i] -> t` to `d` and add `t` to `vis`.
   - We then recurse to see if this new mapping could lead to a solution.
   - Whether the recursive call returns `True` or `False`, we then backtrack by removing the current mapping from `d` and `t` from `vis`, allowing the next iteration to try a different mapping for the same pattern character.
4. **Data Structures**:
   - A dictionary `d` is used to keep track of the current mapping from pattern characters to substrings in `s`.
   - A set `vis` is used to keep track of the substrings of `s` that have already been mapped to some pattern characters. This helps ensure the bijective nature of the mapping.
5. **Initiating the Recursive Search**:
   - Before starting the backtracking process, we initialize the variables `m` and `n` to hold the lengths of `pattern` and `s`, respectively. The dictionary `d` and the set `vis` are initialized to be empty.
   - The `dfs` function is then initiated with `i` and `j` both set to 0, which signifies the start of `pattern` and `s`.

In conclusion, the recursive backtracking approach is quite fitting for this problem, as it allows the solution to explore all possible mappings and backtrack as soon as it detects a mapping scenario that cannot lead to a successful pattern match. This approach is particularly powerful for problems related to pattern matching, permutations, and combinations where all potential scenarios need to be considered.

Example Walkthrough

Let's illustrate the solution approach using a small example. Consider the `pattern` "aba" and the string `s` "catdogcat".

1. **Start with the First Character in Pattern**:
   - We examine the first character in the `pattern`, which is 'a'.
   - We try to map 'a' to each possible prefix of the string `s`. This means we initially consider 'c', 'ca', 'cat', 'catd', 'catdo', 'catdog', 'catdogc', 'catdogca', and 'catdogcat'.
2. **First Recursive Branch - 'a' Mapped to 'c'**:
   - On the first attempt, we associate 'a' with 'c' and then recurse to the next character in the `pattern`.
   - Now, our pattern looks like "c_b_c" with 'b' unassigned, and the remaining part of the string is "atdogcat".
   - Next, we attempt to map 'b' to 'a', but this leaves us with "c_ac_c", and the remaining string is "tdogcat", which no longer has a viable place to map the next 'a', as the next 'a' in the pattern would be mapped to 't'. Since 't' is not equal to our existing mapping of 'a', we must backtrack.
3. **Second Recursive Branch - 'a' Mapped to 'ca'**:
   - Backtracking from the previous step, we now try to map 'a' to 'ca'.
   - Following a similar pattern to the step above, we look at the next character 'b' and attempt to map it to 't', and then 'd', and then 'do', and so on until 'tdog'.
   - Every time, we check if the subsequent character in the pattern ('a') can continue with the already established mapping ('ca'). If not, we continue.
4. **Successful Recursive Branch - 'a' Mapped to 'cat'**:
   - Continuing to explore, we finally map 'a' to 'cat'.
   - Our pattern now looks like "cat_b_cat", and the remaining string is "dog".
   - We map 'b' to 'dog', as it's the only substring left that fits.
   - The full string according to our pattern and mapping is "catdogcat", which matches `s`.
   - Since we've successfully mapped the entire pattern to the string `s` without any conflicting mappings, the algorithm would return `True`.

Throughout the process, the algorithm uses a dictionary to maintain the mapping of pattern characters to substrings in `s` and a set to ensure that no two characters in the pattern map to the same substring. If at any point the mapping is not consistent or a character cannot be mapped to a remaining substring without conflict, the algorithm backtracks and tries a different mapping. In this example, we eventually found a successful mapping that satisfies the requirement of the problem.

Python Solution

```python
class Solution:
    def wordPatternMatch(self, pattern: str, string: str) -> bool:
        # Recursive function to check pattern match
        def backtrack(pattern_index, string_index):
            # Base case: when both pattern and string are completely matched
            if pattern_index == pattern_length and string_index == string_length:
                return True
            # If one is finished before the other, or if the remaining string is
            # shorter than the remaining pattern, it's not a match
            if pattern_index == pattern_length or string_index == string_length or \
               string_length - string_index < pattern_length - pattern_index:
                return False

            for end_index in range(string_index, string_length):
                # Get the current substring
                current_substring = string[string_index : end_index + 1]
                # Check if it matches the pattern's current token
                if pattern_to_string.get(pattern[pattern_index]) == current_substring:
                    # Continue to the next token
                    if backtrack(pattern_index + 1, end_index + 1):
                        return True
                # If it's a new pattern token and substring, try to map them
                if pattern[pattern_index] not in pattern_to_string and \
                   current_substring not in used_substrings:
                    # Add the new mapping
                    pattern_to_string[pattern[pattern_index]] = current_substring
                    used_substrings.add(current_substring)
                    # Continue to the next token
                    if backtrack(pattern_index + 1, end_index + 1):
                        return True
                    # Backtrack: remove the added mapping if it didn't lead to a solution
                    pattern_to_string.pop(pattern[pattern_index])
                    used_substrings.remove(current_substring)
            # If no solution was found after trying all options
            return False

        # Lengths of the input pattern and string
        pattern_length, string_length = len(pattern), len(string)
        # Dictionary to keep track of the mapping from pattern to string
        pattern_to_string = {}
        # Set to keep track of already used substrings
        used_substrings = set()
        # Start the recursion
        return backtrack(0, 0)

```

Java Solution

```java
class Solution {
    // A set to maintain the unique substrings that are already mapped.
    private Set<String> visited;

    // A map to maintain the mapping from a pattern character to its corresponding substring.
    private Map<Character, String> patternMapping;

    // The pattern string that needs to be matched.
    private String pattern;

    // The string that we are trying to match against the pattern.
    private String str;

    // The length of the pattern string.
    private int patternLength;

    // The length of the string str.
    private int stringLength;

    public boolean wordPatternMatch(String pattern, String str) {
        visited = new HashSet<>();
        patternMapping = new HashMap<>();
        this.pattern = pattern;
        this.str = str;
        patternLength = pattern.length();
        stringLength = str.length();
        // Initiate the depth-first search from the beginning of both the pattern and the str.
        return depthFirstSearch(0, 0);
    }

    // A helper function to perform the depth-first search.
    private boolean depthFirstSearch(int patternIndex, int strIndex) {
        // If we reach the end of both the pattern and the str, we have found a match.
        if (patternIndex == patternLength && strIndex == stringLength) {
            return true;
        }
        // If we reach the end of one without the other, or there are more characters to match in the pattern than the remaining le
        if (patternIndex == patternLength || strIndex == stringLength || patternLength - patternIndex > stringLength - strIndex) {
            return false;
        }
        // Get the current character from the pattern.
        char currentPatternChar = pattern.charAt(patternIndex);
        // Try all possible substrings of str starting at strIndex.
        for (int end = strIndex + 1; end <= stringLength; ++end) {
            String currentSubstring = str.substring(strIndex, end);
            // Check if the current pattern character is already mapped to this substring.
            if (patternMapping.getOrDefault(currentPatternChar, "").equals(currentSubstring)) {
                if (depthFirstSearch(patternIndex + 1, end)) {
                    return true;
                }
            } else if (!patternMapping.containsKey(currentPatternChar) && !visited.contains(currentSubstring)) {
                // If the current pattern character is not mapped and the current substring has not been visited, try this new patt
                patternMapping.put(currentPatternChar, currentSubstring);
                visited.add(currentSubstring);
                if (depthFirstSearch(patternIndex + 1, end)) {
                    return true;
                }
                // Backtrack and try different mappings for the current pattern character.
                visited.remove(currentSubstring);
                patternMapping.remove(currentPatternChar);
            }
        }
        // If no valid mapping was found, return false.
        return false;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Main function to check if the pattern matches the string
    bool wordPatternMatch(string pattern, string s) {
        unordered_set<string> visited; // Holds unique mappings
        unordered_map<char, string> dict; // Maps pattern chars to strings
        // Start recursive depth-first search for pattern matching
        return dfs(0, 0, pattern, s, visited, dict);
    }

    // Helper function to perform a depth-first search
    bool dfs(int patternIndex, int strIndex, string& pattern, string& str,
             unordered_set<string>& visited, unordered_map<char, string>& dict) {
        int patternSize = pattern.size(), strSize = str.size();
        // If both pattern and string indices are at the end, we have a match
        if (patternIndex == patternSize && strIndex == strSize) return true;
        // If one of them is at the end or if there aren't enough characters
        // left in str for the remaining pattern, there's no match
        if (patternIndex == patternSize || strIndex == strSize || patternSize - patternIndex > strSize - strIndex) return false;

        // Current pattern character
        char currentChar = pattern[patternIndex];

        // Try every possible string partition
        for (int k = strIndex + 1; k <= strSize; ++k) {
            // Get a substring from the current str index up to k
            string t = str.substr(strIndex, k - strIndex);
            // If current pattern char is already associated with that substring
            if (dict.count(currentChar) && dict[currentChar] == t) {
                // Continue the search for the rest of the string and pattern
                if (dfs(patternIndex + 1, k, pattern, str, visited, dict)) return true;
            }
            // If current pattern char isn't associated yet and t is a new word
            if (!dict.count(currentChar) && !visited.count(t)) {
                // Create new associations and continue search
                dict[currentChar] = t;
                visited.insert(t);
                if (dfs(patternIndex + 1, k, pattern, str, visited, dict)) return true;
                // Backtracking: undo the association if it doesn't lead to a solution
                visited.erase(t);
                dict.erase(currentChar);
            }
        }
        // If no partitioning leads to a solution, return false
        return false;
    }
};

```

Typescript Solution

```typescript
// Importing Set and Map classes from ES6 for usage
import { Set } from "typescript-collections";
import { Map } from "typescript-collections";

// Tracks unique mappings of strings to characters
const visited: Set<string> = new Set();
// Maps pattern characters to strings
const dict: Map<string, string> = new Map();

// Main function to check if the pattern matches the string
function wordPatternMatch(pattern: string, s: string): boolean {
    // Start recursive depth-first search for pattern matching
    return dfs(0, 0, pattern, s);
}

// Helper function to perform a depth-first search
function dfs(patternIndex: number, strIndex: number, pattern: string, str: string): boolean {
    // The sizes of the pattern and the string
    const patternSize: number = pattern.length;
    const strSize: number = str.length;

    // If both pattern and string indices are at the end, we have a match
    if (patternIndex === patternSize && strIndex === strSize) return true;
    // If one of them is at the end, or if there aren't enough characters
    // left in str to match the remaining pattern, there's no match
    if (patternIndex === patternSize || strIndex === strSize || patternSize - patternIndex > strSize - strIndex) return false;

    // Current pattern character
    const currentChar: string = pattern.charAt(patternIndex);

    // Attempt every possible partition of the string
    for (let k: number = strIndex + 1; k <= strSize; k++) {
        // Get a substring from the current str index up to k
        const t: string = str.substring(strIndex, k);
        // If current pattern character is already associated with that substring
        if (dict.containsKey(currentChar) && dict.getValue(currentChar) === t) {
            // Continue the search for the rest of the string and pattern
            if (dfs(patternIndex + 1, k, pattern, str)) return true;
        }
        // If current pattern character isn't associated yet, and t is a new substring
        if (!dict.containsKey(currentChar) && !visited.contains(t)) {
            // Create new associations and continue search
            dict.setValue(currentChar, t);
            visited.add(t);
            if (dfs(patternIndex + 1, k, pattern, str)) return true;
            // Backtracking: undo the association if it doesn't lead to a solution
            visited.remove(t);
            dict.remove(currentChar);
        }
    }

    // If no partitioning leads to a solution, return false
    return false;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function is dependent on the number of recursive calls made by the `dfs` function. In each call, the function tries to map `pattern[i]` to a substring of `s` starting at index `j`. The for loop in `dfs` can run up to `O(n)` times for each call, where `n` is the size of string `s`.

In the worst case, the pattern can be bijectively mapped onto a prefix of `s`, leading to a situation where each character in `pattern` maps to a different substring of `s`. This means that there could be up to `n` levels in our recursion tree, each having `n` branches if we assume that the length of `s` is `n`.

Therefore, the worst-case time complexity is `O(n^m)`, where `m` is the length of the pattern and `n` is the length of the string `s`. However, since the depth of the recursion is also restricted by the size of `m` with the condition `n - j < m - i`, the actual time complexity could potentially be lower, but the upper bound still holds as the worst-case scenario.

Space Complexity

The space complexity is influenced by the storage of the mappings (`d`) and the visited substrings (`vis`), in addition to the recursion call stack.

The maximum size of the dictionary `d` is bounded by the length of the pattern `m`, since at most each character in `pattern` can map to a unique substring of `s`. Therefore, the space taken by `d` is `O(m)`.

The `vis` set can potentially contain all possible substrings of `s` that are mapped to pattern characters. In the worst case, this could result in `O(n^2)` space complexity since there can be `n` choices for the starting point and `n` choices for the ending point of the substring.

The recursion stack can go as deep as the number of characters in `pattern`, so the maximum depth is `O(m)`.

Combining these considerations, the total space complexity becomes `O(m + n^2)`, with `n^2` usually dominating unless m is significantly larger than n.