10. Regular Expression Matching

Hard | Recursion | String | Dynamic Programming Leetcode Link

Problem Description

This problem asks you to implement a function that determines if the given input string `s` matches the given pattern `p`. The pattern `p` can include two special characters:

- A period/dot (`.`) which matches any single character.
- An asterisk (`*`) which matches zero or more of the element right before it.

The goal is to check if the pattern `p` matches the entire string `s`, not just a part of it. That means we need to see if we can navigate through the entire string `s` using the rules defined by the pattern.

Intuition

The intuition behind the provided solution is using dynamic programming to iteratively build up a solution. We create a 2D table `f` where `f[i][j]` will represent whether the first `i` characters of `s` match the first `j` characters of `p`.

The approach is as follows:

1. Initialize the table with `False`, and set `f[0][0]` to `True` because an empty string always matches an empty pattern.
2. Iterate over each character in the string `s` and the pattern `p`, and update the table based on the following rules:
   - If the current character in `p` is `*`, we check two things: a. If the pattern without this star and its preceding element matches the current string `s` up to `i`, i.e., `f[i][j] = f[i][j - 2]`. b. If the element before the star can be matched to the current character in `s` (either it's the same character or it's a `.`), and if the pattern `p` up to the current point matches the string `s` up until the previous character, i.e., `f[i - 1][j]`.
   - If the current character in `p` is `.` or it matches the current character in `s`, we just carry over the match state from the previous characters, i.e., `f[i][j] = f[i - 1][j - 1]`.
3. At the end, `f[m][n]` contains the result, which tells us if the whole string `s` matches the pattern `p`, where `m` and `n` are the lengths of `s` and `p`, respectively.

The key here is to realize that the problem breaks down into smaller subproblems. If we know how smaller parts of the string and pattern match, we can use those results to solve for larger parts. This is a classic dynamic programming problem where optimal substructure (the problem can be broken down into subproblems) and overlapping subproblems (calculations for subproblems are reused) are the main components.

Solution Approach

The solution involves dynamic programming – a method for solving complex problems by breaking them down into simpler subproblems. The key to this solution is a 2D table `f` with the dimensions `(m + 1) x (n + 1)`, where `m` is the length of the string `s` and `n` is the length of the pattern `p`. This table helps in storing the results of subproblems so they can be reused when necessary.

The algorithm proceeds as follows:

1. **Initialize the DP Table**: Create a boolean DP table `f` where `f[i][j]` is `True` if the first `i` characters of `s` (sub-s) match the first `j` characters of `p` (sub-p), and `False` otherwise. We initialize the table with `False` and set `f[0][0]` to `True` to represent that empty `s` matches empty `p`.
2. **Handle Empty Patterns**: Due to the nature of the `*` operator, a pattern like "a*" can match an empty sequence. We iterate over the pattern `p` and fill in `f[0][j]` (the case where `s` is empty). For example, if `p[j-1]` is `*`, then we check two characters back and if `f[0][j-2]` is `True`, then `f[0][j]` should also be `True`.
3. **Fill the Table**: The main part of the algorithm is to iterate over each character in `s` and `p` and decide the state of `f[i][j]` based on the last character of the sub-pattern `p[0...j]`:

   a. If the last character of sub-p is `*`, there are two subcases:

   - The star can be ignored (match 0 of the preceding element). This means if the pattern matches without the last two characters (`*` and its preceding element), the current state should be `True` (`f[i][j] = f[i][j-2]`).
   - The star contributes to the match (match 1 or more of the preceding element). This happens if the character preceding `*` is the same as the last character in sub-s or if it's a dot. If `f[i - 1][j]` is `True`, we can also set `f[i][j]` to `True` (`f[i][j] |= f[i - 1][j]`).

   b. If the last character of sub-p is not `*`, we check if it's a dot or a matching character:

   - If the characters match or if the character in `p` is `.` (which matches any character), the current state depends on the previous state without these two characters: `f[i][j] = f[i - 1][j - 1]`.
4. **Return the Result**: Once the table is filled, the answer to whether `s` matches `p` is stored in `f[m][n]`, because it represents the state of the entire string `s` against the entire pattern `p`.

In essence, the solution uses a bottom-up approach to fill the DP table, starting from an empty string/pattern and building up to the full length of `s` and `p`. The transition between the states is determined by the logic that considers the current and previous characters of `p` and their meaning based on regex rules.

Example Walkthrough

Let's take a small example to illustrate the approach described above. Consider `s = "xab"` and `p = "x*b."`. We want to determine if the pattern matches the string.

1. **Initialize the DP Table**: We create a table `f` where `f[i][j]` will be `True` if the first `i` characters of `s` (`sub-s`) match the first `j` characters of `p` (`sub-p`). The table has dimensions `(len(s) + 1) x (len(p) + 1)`, which is `(4 x 4)`:

	0	1	2	3
0	T	F	F	F
1	F			
2	F			
3	F			

   Here, `T` denotes `True`, and `F` denotes `False`. `f[0][0]` is `True` because an empty string matches an empty pattern.

2. **Handle Empty Patterns**: We iterate over `p` and update `f[0][j]`. Since `p[1]` is `*`, we can ignore "x*" for an empty `s`, so `f[0][2]` becomes `True`:

	0	1	2	3
0	T	F	T	F
1	F			
2	F			
3	F			

3. **Fill the Table**: Now, we iterate over the string `s` and pattern `p`.

   - For `i = 1` and `j = 1`, `s[0]` matches `p[0]` ('x' == 'x'). So `f[1][1] = f[0][0]` which is `True`.
   - For `i = 1` and `j = 2`, we have a `*`. As per the rules, we check `f[1][0]` (ignoring the star completely) which is `False`, so `f[1][2]` remains `False`.
   - However, since `p[1]` is `*`, and 'x' can match 'x', we also check `f[1 - 1][2]` which is `True`. Hence, `f[1][2]` is `True`.
   - For `i = 1` and `j = 3`, we move to the next character because `p[2]` is not a special character and it does not match 'x'. Hence, `f[1][3]` remains `False`.
   - For `i = 2` and `j = 2`, we have a `*`. The preceding letter 'x' can match 'x', so we check `f[2 - 1][2]` which is `True`, and hence `f[2][2]` is `True`.
   - For `i = 2` and `j = 3`, `p[2]` is `'.'` and it matches any character, while `f[1][2]` is `True`. Therefore, `f[2][3]` is `True`.
   - For `i = 3` and `j = 2`, we have a `*`. We consider matching zero or multiple 'x'. Since `f[2][2]` is `True`, and 'x' can match 'x', `f[3][2]` becomes `True`.
   - For `i = 3` and `j = 3`, `p[2]` is `'.'` and it matches any character, so `f[3][3] = f[2][2]`, hence `f[3][3]` is `True`.

   The final table looks as follows:

	0	1	2	3
0	T	F	T	F
1	F	T	T	F
2	F	F	T	T
3	F	F	T	T

4. **Return the Result**: The answer is stored in `f[m][n]`, which is `f[3][3]`. It is `True`, so `s` matches `p`.

By setting up this table and following the rules, we can confidently say that "xab" matches the pattern "x*b.".

Python Solution

```python
class Solution:
    def isMatch(self, text: str, pattern: str) -> bool:
        # Get lengths of text and pattern
        text_length, pattern_length = len(text), len(pattern)

        # Initialize DP table with False values
        dp = [[False] * (pattern_length + 1) for _ in range(text_length + 1)]

        # Empty pattern matches an empty text
        dp[0][0] = True

        # Iterate over text and pattern lengths
        for i in range(text_length + 1):
            for j in range(1, pattern_length + 1):
                # If the pattern character is '*', it could match zero or more of the previous element
                if pattern[j - 1] == "*":
                    # Check if zero occurrences of the character before '*' match
                    dp[i][j] = dp[i][j - 2]
                    # Additional check for one or more occurrences
                    if i > 0 and (pattern[j - 2] == "." or text[i - 1] == pattern[j - 2]):
                        dp[i][j] |= dp[i - 1][j]
                # If the current characters match or if pattern has '.', mark as true
                elif i > 0 and (pattern[j - 1] == "." or text[i - 1] == pattern[j - 1]):
                    dp[i][j] = dp[i - 1][j - 1]

        # The result is at the bottom right of the DP table
        return dp[text_length][pattern_length]

# Example usage:
# sol = Solution()
# result = sol.isMatch("aab", "c*a*b")
# print(result)  # Output: True

```

Java Solution

```java
class Solution {
    public boolean isMatch(String text, String pattern) {
        int textLength = text.length();
        int patternLength = pattern.length();

        // dp[i][j] will be true if the first i characters in the text match the first j characters of the pattern
        boolean[][] dp = new boolean[textLength + 1][patternLength + 1];

        // Base case: empty text and empty pattern are a match
        dp[0][0] = true;

        // Iterate over each position in the text and pattern
        for (int i = 0; i <= textLength; i++) {
            for (int j = 1; j <= patternLength; j++) {

                // If the current pattern character is '*', it will be part of a '*' pair with the prev char
                if (pattern.charAt(j - 1) == '*') {
                    // Check the position without the '*' pair (reduce pattern by 2)
                    dp[i][j] = dp[i][j - 2];
                    // If text character matches pattern character before '*' or if it's a '.'
                    if (i > 0 && (pattern.charAt(j - 2) == '.' || pattern.charAt(j - 2) == text.charAt(i - 1))) {
                        // 'OR' with the position above to see if any prev occurrences match
                        dp[i][j] |= dp[i - 1][j];
                    }
                } else {
                    // For '.' or exact match, current dp position is based on the prev diagonal position
                    if (i > 0 && (pattern.charAt(j - 1) == '.' || pattern.charAt(j - 1) == text.charAt(i - 1))) {
                        dp[i][j] = dp[i - 1][j - 1];
                    }
                }
            }
        }

        // The result is at the bottom-right corner, indicating if the entire text matches the entire pattern
        return dp[textLength][patternLength];
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Function to check if string 's' matches the pattern 'p'.
    bool isMatch(string s, string p) {
        int m = s.size(), n = p.size();
        vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

        // Base case: empty string matches with empty pattern
        dp[0][0] = true;

        // Fill the dp table
        for (int i = 0; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                // If the pattern character is '*', it can either eliminate the character and its predecessor
                // or if the string is not empty and the character matches, include it
                if (p[j - 1] == '*') {
                    dp[i][j] = dp[i][j - 2];
                    if (i > 0 && (p[j - 2] == '.' || p[j - 2] == s[i - 1])) {
                        dp[i][j] = dp[i][j] || dp[i - 1][j];
                    }
                }
                // If the current characters match (or the pattern has '.'), then the result
                // is determined by the previous states of both the string and pattern
                else if (i > 0 && (p[j - 1] == '.' || p[j - 1] == s[i - 1])) {
                    dp[i][j] = dp[i - 1][j - 1];
                }
            }
        }

        // Return the result at the bottom-right corner of the dp table
        return dp[m][n];
    }
};

```

Typescript Solution

```typescript
/**
 * Determine if the input string matches the pattern provided. The pattern may include '.' to represent any single character,
 * and '*' to denote zero or more of the preceding element.
 * @param {string} inputString - The input string to be matched.
 * @param {string} pattern - The pattern string, which may contain '.' and '*' special characters.
 * @returns {boolean} - Whether the input string matches the pattern.
 */
const isMatch = (inputString: string, pattern: string): boolean => {
    const inputLength: number = inputString.length;
    const patternLength: number = pattern.length;
    // Initialize DP table with all false values.
    const dp: boolean[][] = Array.from({ length: inputLength + 1 }, () => Array(patternLength + 1).fill(false));

    // Base case: empty string and empty pattern are a match.
    dp[0][0] = true;

    // Fill the DP table
    for (let i = 0; i <= inputLength; ++i) {
        for (let j = 1; j <= patternLength; ++j) {
            // If the pattern character is '*', we have two cases to check
            if (pattern[j - 1] === '*') {
                // Check if the pattern before '*' matches (zero occurrences of the preceding element).
                dp[i][j] = dp[i][j - 2];
                if (i && (pattern[j - 2] === '.' || pattern[j - 2] === inputString[i - 1])) {
                    // If one or more occurrences of the preceding element match, use the result from the row above.
                    dp[i][j] = dp[i][j] || dp[i - 1][j];
                }
            } else if (i && (pattern[j - 1] === '.' || pattern[j - 1] === inputString[i - 1])) {
                // If the current pattern character is '.', or it matches the current input character, follow the diagonal.
                dp[i][j] = dp[i - 1][j - 1];
            }
        }
    }
    // The final result will be in the bottom-right corner of the DP table.
    return dp[inputLength][patternLength];
};

// The function can be tested with an example call
// console.log(isMatch('string', 'pattern')); // Replace 'string' and 'pattern' with actual values to test.

```

Time and Space Complexity

The time complexity of the provided code is `O(m * n)`, where `m` is the length of the input string `s` and `n` is the length of the pattern `p`. This is because the solution iterates through all combinations of positions in `s` and `p` using nested loops.

In terms of space complexity, the code uses `O(m * n)` space as well due to the creation of a 2D array `f` that has `(m + 1) * (n + 1)` elements to store the state of matching at each step.