44. Wildcard Matching

Hard | Greedy | Recursion | String | Dynamic Programming

Problem Description

The problem is a classic example of pattern matching where we are given a string `s` and a pattern `p` that includes wildcard characters. We need to determine if the pattern `p` matches the entire string `s`. The wildcard characters are defined as follows:

- A question mark (`'?'`) matches any single character.
- An asterisk (`'*'`) matches any sequence of characters, including an empty sequence.

The goal is to check if there is a complete match between the entire string and the pattern, not just a partial match.

Intuition

For solving this problem, dynamic programming is a common approach because it allows us to break down the complex problem into smaller subproblems and then build up the solution from these.

The key insight is to realize that we can make a decision at each character in the string based on two conditions:

1. If the current character in the pattern is a `?` or matches the current character in the string, we refer to previous states where the match has been progressing without the current character and pattern.

2. If the current character in the pattern is a `*`, it can be complex because `*` can match an empty sequence or any sequence of characters. We need to consider multiple cases: either we use the `*` to match zero characters in the string (which means we look at the state of the match without the `*`), or we use the `*` to match at least one character in the string (which means we look at the state of the match without the current character in the string, but keep the `*`).

This method of breaking down the problem helps us to derive a solution using a 2D matrix `dp` where `dp[i][j]` represents whether the first `i` characters of the string `s` can be matched with the first `j` characters of the pattern `p`. The final answer at `dp[m][n]` (where `m` and `n` are the lengths of the string and pattern respectively) gives us the answer to whether the entire string matches the pattern.

By filling up the matrix by iterating over the string and the pattern, we use the previously solved subproblems to inform the solution of the current subproblem. Eventually, we derive the solution to the entire problem.

Solution Approach

The solution uses dynamic programming, a method where complex problems are broken into simpler subproblems and solved individually, with the solutions to the subproblems stored to avoid redundant calculations.

Here is a breakdown of the implementation steps:

1. Initialize a 2D matrix `dp` with dimensions `(m + 1) x (n + 1)`, where `m` is the length of the string `s` and `n` is the length of the pattern `p`. Each element `dp[i][j]` in this matrix will store a boolean value indicating if `s[0..i-1]` matches `p[0..j-1]`. The `+1` offset allows us to easily handle the empty string and pattern cases.

2. Set the first element `dp[0][0]` to `True` to represent that an empty string matches an empty pattern.

3. Pre-fill the first row of the matrix by setting `dp[0][j]` to `True` if `p[j-1]` is `*` and `dp[0][j-1]` is also `True`. This loop accounts for the situation where the pattern starts with one or multiple `*` characters, which can match the empty string.

```python
for j in range(1, n + 1):
    if p[j - 1] == '*':
        dp[0][j] = dp[0][j - 1]
```

4. Iterate through the matrix starting from `i = 1` and `j = 1`, and calculate the `dp[i][j]` value based on the following rules:

   - If the current character of `s[i - 1]` matches the current character of `p[j - 1]` or `p[j - 1]` is a `?` (which can match any single character), then set `dp[i][j]` to the value of `dp[i - 1][j - 1]` since we can carry forward the match status from previous characters.

     ```python
     if s[i - 1] == p[j - 1] or p[j - 1] == '?':
         dp[i][j] = dp[i - 1][j - 1]
     ```

   - If the current character of the pattern is `*`, there are two possibilities:

     - The `*` matches zero characters: carry forward the status from `dp[i][j - 1]`.
     - The `*` matches one or more characters: carry forward the status from `dp[i - 1][j]`.

   Hence:

   ````
   ```python
   elif p[j - 1] == '*':
       dp[i][j] = dp[i - 1][j] or dp[i][j - 1]
   ```
   ````

   - For all other cases where characters don't match and there is no wildcard, `dp[i][j]` will remain False, which is the default value after initialization.

5. After filling the entire matrix, the value at `dp[m][n]` will indicate whether the entire string `s` matches the pattern `p`.

This approach ensures that each subproblem is only calculated once and then reused, dramatically reducing the time complexity from exponential (which would be the case with a naive recursive approach) to polynomial time.

Example Walkthrough

Let's illustrate the solution approach using a small example:

Assume `s = "xaabyc"` and `p = "*a?b*"`.

1. **Initialize the 2D matrix `dp`**: Since `s` has a length of 6 and `p` has a length of 5, our matrix `dp` will be a 7×6 matrix (including the extra row and column for the empty string and pattern case). Initially, all values in `dp` are set to `False`.

2. **First element `dp[0][0]`**: We set `dp[0][0]` to `True` to denote that an empty pattern matches an empty string.

3. **First row pre-fill**: We then iterate over the first row of `dp` to account for a pattern that starts with `*`. In this case, `p[0]` is `*`, so `dp[0][1]` should be set to `True`. Following that logic, here's how the first row will look after pre-filling:

		*	a	?	b	*
	T	T	F	F	F	F

4. **Iterate and fill the matrix**:

a. For `i = 1` and `j = 1`, the pattern has `*` which matches zero characters from `s`. So, `dp[1][1]` is `True` because `dp[0][0]` was true.

b. When `j = 2` and `i = 1`, the pattern is `a` but our string is `x`. Since they don't match and the pattern is not a `?`, `dp[1][2]` stays `False`.

		*	a	?	b	*
	T	T	F	F	F	F
x	F	T	F	F	F	F

c. Eventually, by following the iteration rules while filling out the matrix, we will have:

		*	a	?	b	*
	T	T	F	F	F	F
x	F	T	F	F	F	F
a	F	T	T	F	F	F
a	F	T	T	T	F	F
b	F	T	F	T	T	F
y	F	T	F	F	T	T
c	F	T	F	F	F	T

5. **Final Check**: Our final cell `dp[6][5]` contains `True`. Therefore, we can deduce that with the given example, the string `s` matches the pattern `p`.

By processing the `dp` matrix according to the dynamic programming approach, we've avoided redundant calculations and determined the match between `s` and `p` efficiently.

Python Solution

```python
class Solution:
    def isMatch(self, string: str, pattern: str) -> bool:
        # Get the lengths of the input string and the pattern
        length_s, length_p = len(string), len(pattern)

        # Create a DP table with default values False
        dp = [[False] * (length_p + 1) for _ in range(length_s + 1)]

        # The empty pattern matches the empty string
        dp[0][0] = True

        # Initialize first row of the DP table, considering the pattern starting with '*'
        for j in range(1, length_p + 1):
            if pattern[j - 1] == '*':
                dp[0][j] = dp[0][j - 1]

        # Fill the DP table
        for i in range(1, length_s + 1):
            for j in range(1, length_p + 1):
                # If characters match or pattern has '?', we can move back diagonally in the table (match found)
                if string[i - 1] == pattern[j - 1] or pattern[j - 1] == '?':
                    dp[i][j] = dp[i - 1][j - 1]
                # If pattern has '*', we check two cases:
                # 1. '*' matches no character: move left in the table
                # 2. '*' matches at least one character: move up in the table
                elif pattern[j - 1] == '*':
                    dp[i][j] = dp[i - 1][j] or dp[i][j - 1]

        # Return the value at the bottom-right corner of the DP table
        return dp[length_s][length_p]

```

Java Solution

```java
class Solution {

    public boolean isMatch(String str, String pattern) {
        // Lengths of the input string and the pattern
        int strLen = str.length(), patternLen = pattern.length();

        // dp[i][j] will be true if the first i characters in given string
        // match the first j characters of the pattern
        boolean[][] dp = new boolean[strLen + 1][patternLen + 1];

        // Empty string and empty pattern are a match
        dp[0][0] = true;

        // Initialize the first row for the cases where pattern contains *
        // as they can match the empty string
        for (int j = 1; j <= patternLen; ++j) {
            if (pattern.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j - 1];
            }
        }

        // Build the dp matrix in bottom-up manner
        for (int i = 1; i <= strLen; ++i) {
            for (int j = 1; j <= patternLen; ++j) {
                // If the current characters match or pattern has '?',
                // we can propagate the diagonal value
                if (str.charAt(i - 1) == pattern.charAt(j - 1) || pattern.charAt(j - 1) == '?') {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                // If pattern contains '*', it can either match zero characters
                // in the string or it can match one character in the string
                // and continue matching
                else if (pattern.charAt(j - 1) == '*') {
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
                }
                // If the current pattern character is not a wildcard and the characters
                // don't match, dp[i][j] remains false, which is the default value.
            }
        }

        // The value in the bottom right corner will be our answer
        return dp[strLen][patternLen];
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>

class Solution {
public:
    bool isMatch(const std::string& s, const std::string& p) {
        int strSize = s.size(), patternSize = p.size();
        // Create a DP table with dimensions (m+1) x (n+1) initialized to false.
        // dp[i][j] will be true if the first i characters of s match the first j
        // characters of p.
        std::vector<std::vector<bool>> dp(strSize + 1, std::vector<bool>(patternSize + 1, false));

        // The empty pattern matches the empty string.
        dp[0][0] = true;

        // Initialize the first row of the DP table. If we find '*', it can match
        // an empty string, which is the state of dp[0][j-1].
        for (int j = 1; j <= patternSize; ++j) {
            if (p[j - 1] == '*') {
                dp[0][j] = dp[0][j - 1];
            }
        }

        // Fill the DP table.
        for (int i = 1; i <= strSize; ++i) {
            for (int j = 1; j <= patternSize; ++j) {
                // If the characters match or the pattern character is '?',
                // we can transition from the state dp[i-1][j-1].
                if (s[i - 1] == p[j - 1] || p[j - 1] == '?') {
                    dp[i][j] = dp[i - 1][j - 1];
                }
                // If the pattern character is '*', it can either match zero characters,
                // meaning we transition from dp[i][j-1], or it can match one character,
                // meaning we transition from dp[i-1][j].
                else if (p[j - 1] == '*') {
                    dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
                }
            }
        }

        // The final state dp[m][n] gives us the answer to whether the entire
        // strings s and p match with each other.
        return dp[strSize][patternSize];
    }
};

```

Typescript Solution

```typescript
function isMatch(s: string, p: string): boolean {
    let strLen: number = s.length;
    let patternLen: number = p.length;

    // Create a DP table with dimensions (strLen+1) x (patternLen+1) initialized to false.
    // dp[i][j] will be true if the first i characters of s match the first j characters of p.
    let dp: boolean[][] = Array.from({ length: strLen + 1 },
                                     () => Array<boolean>(patternLen + 1).fill(false));

    // The empty pattern matches the empty string.
    dp[0][0] = true;

    // Initialize the first row of the DP table. If we find '*', it can match
    // an empty string, thereby adopting the value from dp[0][j-1].
    for (let j = 1; j <= patternLen; j++) {
        if (p[j - 1] === '*') {
            dp[0][j] = dp[0][j - 1];
        }
    }

    // Fill the DP table.
    for (let i = 1; i <= strLen; i++) {
        for (let j = 1; j <= patternLen; j++) {
            // If the characters match or the pattern character is '?',
            // we can transition from the state dp[i-1][j-1].
            if (s[i - 1] === p[j - 1] || p[j - 1] === '?') {
                dp[i][j] = dp[i - 1][j - 1];
            }
            // If the pattern character is '*', it can either match zero characters,
            // meaning we transition from dp[i][j-1], or it can match one or more characters,
            // meaning we transition from dp[i-1][j].
            else if (p[j - 1] === '*') {
                dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
            }
        }
    }

    // The final state dp[strLen][patternLen] gives us the answer to whether the entire
    // strings s and p match with each other.
    return dp[strLen][patternLen];
}

```

Time and Space Complexity

The time complexity of the provided code is `O(m * n)`, where `m` is the length of the string `s` and `n` is the length of the pattern `p`. This is because the code involves a nested loop structure that iterates through the lengths of `s` and `p`. Each cell in the DP table `dp[i][j]` computes whether the first `i` characters of `s` match the first `j` characters of `p`, and the computation of each cell is constant time.

The space complexity of the code is also `O(m * n)`. This is due to the use of a two-dimensional list `dp`, which has `(m + 1) * (n + 1)` elements, to store the states of substring matches. Each element of this list represents a subproblem, with extra rows and columns to handle empty strings.