Wildcard Matching Documentation

Overview

This documentation provides an explanation of the wildcard matching problem along with a solution that implements wildcard pattern matching with support for '?' and '*'. The goal is to determine whether an input string matches a given pattern, where '?' matches any single character and '*' matches any sequence of characters, including the empty sequence. The matching should cover the entire input string, not just a partial match.

Problem Description

Given an input string `s` and a pattern `p`, the task is to determine whether the pattern `p` matches the entire input string `s`.

Examples

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "*"

Output: true

Explanation: '*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"

Output: false

Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

Constraints

- 0 <= s.length, p.length <= 2000
- `s` contains only lowercase English letters.
- `p` contains only lowercase English letters, '?' or '*'.

Solution Approach

The solution employs dynamic programming to solve the problem efficiently. It utilizes a 2D array `dp` to store whether substrings of `s` and `p` match up to a certain index. The algorithm iterates through each character of both `s` and `p`, updating `dp` based on the following conditions:

- If the current character in `p` is '*', `dp[i][j]` is updated based on the preceding characters in `s` and `p`.
- If the current character in `p` is '?' or matches the corresponding character in `s`, `dp[i][j]` is updated based on the previous state of `dp`.
- Otherwise, `dp[i][j]` remains `False`.

Finally, the algorithm returns `dp[m][n]`, where `m` and `n` are the lengths of `s` and `p`, respectively.

Class Structure

The solution is encapsulated within a class named `Solution`, containing a method `isMatch` which takes `s` and `p` as input parameters and returns a boolean value indicating whether the input string matches the pattern.

Time Complexity

The time complexity of the solution is O(m * n), where m and n are the lengths of the input strings `s` and `p`, respectively. This is because the algorithm involves iterating through each character of both strings once to populate the dynamic programming array `dp`.

Space Complexity

The space complexity of the solution is O(m * n), where m and n are the lengths of the input strings `s` and `p`, respectively. This space is required to store the dynamic programming array `dp`.