**The problem of the day – 2**

**Problem:**

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '\*' where:

'?' Matches any single character.

'\*' Matches any sequence of characters (including the empty sequence).

The matching should cover the entire input string (not partial).

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 '\*'.

**Logic code:**

class Solution {

public boolean isMatch(String s, String p) {

boolean dp[][] = new boolean[p.length()+1][s.length()+1];

for(int i = dp.length-1; i >= 0; i--) {

for(int j = dp[0].length-1; j >= 0; j--) {

if(i == dp.length-1 && j == dp[0].length-1) {

dp[i][j] = true;

} else if(i == dp.length-1) {

dp[i][j] = false;

} else if(j == dp[0].length-1) {

if(p.charAt(i) == '\*') {

dp[i][j] = dp[i+1][j];

} else {

dp[i][j] = false;

}

} else {

if(p.charAt(i) == '?') {

dp[i][j] = dp[i+1][j+1];

} else if(p.charAt(i) == '\*') {

dp[i][j] = dp[i+1][j] || dp[i][j+1];

} else if(p.charAt(i) == s.charAt(j)) {

dp[i][j] = dp[i+1][j+1];

} else {

dp[i][j] = false;

}

}

}

}

return dp[0][0];

}

}

**Approach:**

Here's a step-by-step explanation of your dynamic programming approach for wildcard pattern matching:

Create a 2D boolean array dp with dimensions (p.length()+1) x (s.length()+1). dp[i][j] will represent whether the pattern p starting from index i matches the input string s starting from index j.

Initialize the base cases:

For i = p.length() (when pattern is empty), and j = s.length() (when input is empty), dp[i][j] is set to true since an empty pattern matches an empty string.

For i = p.length() (when pattern is empty) and any valid j, dp[i][j] is set to false since a non-empty input string can't match an empty pattern.

For j = s.length() (when input is empty) and any valid i, you need to check if the pattern consists of only \*. If yes, then it matches the empty string, otherwise, it doesn't match.

Fill in the dp array by considering the characters of the pattern and the input string one by one, starting from the last characters of each:

If p.charAt(i) is a ?, the result depends on whether the rest of the pattern (p[i+1:]) matches the rest of the input string (s[j+1:]). Therefore, dp[i][j] is set to dp[i+1][j+1].

If p.charAt(i) is a \*, you have two choices:

You can consider the \* as matching zero characters in the input string, i.e., dp[i][j] depends on dp[i+1][j].

You can consider the \* as matching one or more characters in the input string, i.e., dp[i][j] depends on dp[i][j+1].

Therefore, dp[i][j] is set to dp[i+1][j] || dp[i][j+1].

If p.charAt(i) is a regular character and matches s.charAt(j), then the result depends on whether the rest of the pattern and input string match (dp[i][j] is set to dp[i+1][j+1]).

If p.charAt(i) is a regular character but doesn't match s.charAt(j), then dp[i][j] is set to false.

The final answer is stored in dp[0][0], which represents whether the entire pattern p matches the entire input string s.

Complexity

Time complexity:

The time complexity of your code is O(m \* n), where m is the length of the pattern string p and n is the length of the input string s. This is because you are using a nested loop to iterate over all possible combinations of characters in p and s, and for each combination, you perform constant time operations.

Space complexity:

The space complexity of your code is also O(m \* n), where m is the length of the pattern string p and n is the length of the input string s. This is because you are using a 2D boolean array dp with dimensions (m+1) x (n+1) to store intermediate results for dynamic programming. Each cell in the array requires constant space, and since you have (m+1) x (n+1) cells, the overall space complexity is O(m \* n).