Wildcard Pattern Matching - GeeksforGeeks

Source: https://www.geeksforgeeks.org/wildcard-pattern-matching/

Courses Tutorials Practice Jobs DSA Tutorial Interview Questions Quizzes Must Do Advanced DSA System Design Aptitude Puzzles Interview Corner DSA Python Technical Scripter 2026 Explore DSA Fundamentals Logic Building Problems Analysis of Algorithms Data Structures Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Algorithms Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Tutorial Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Advanced Segment Tree Binary Indexed Tree or Fenwick Tree Square Root (Sqrt) Decomposition Algorithm Binary Lifting Geometry Interview Preparation Interview Corner GfG160 Practice Problem GeeksforGeeks Practice - Leading Online Coding Platform Problem of The Day - Develop the Habit of Coding DSA Course 90% Refund Wildcard Pattern Matching Last Updated : 10 Nov, 2025 Given two strings pat and txt which may be of different lengths. Check if the wildcard pattern(pat) matches with txt or not. The wildcard pattern pat can include the characters ' ? ' and ' * '. ' ? ' – can match to any single character. ' * ' – can match to any sequence of characters (including the empty sequence). Note: After processing, the wildcard pattern pat must completely match the entire text txt. Input: txt = "abcde", pat = "a?c*" Output: true Explanation: ? matches the character b in the text and * matches the substring de in the text. Input: txt = "baaabab", pat = "a*ab" Output: false Explanation : The pattern starts with a, but the text starts with b, so the pattern does not match the text. Input: txt = "abc", pat = "*" Output: true Explanation: The pattern * matches to the entire string "abc". Try it on GfG Practice Table of Content [Naive Approach] Using Recursion - O(2^(n+m)) Time and O(n) Space [Better Approach 1] Using Memoization - O(n*m) Time and O(n*m) Space [Better Approach 2] Using Bottom-Up DP (Tabulation) - O(n*m) Time and O(n*m) Space [Efficient Approach] Using Space Optimized DP - O(n*m) Time and O(m) Space [Expected Approach] Simple Traversal - O(n * m) Time and O(1) Space [Naive Approach] Using Recursion - O(2 (n+m)) Time and O(n) Space The idea is to recursively check if the given pattern can match the text by comparing characters from the end of both strings. If both strings become empty at the same time, it means the match is successful. If the text becomes empty but the pattern still has characters, it can only match if all remaining pattern characters are *. When the current characters are the same, or the pattern has a ?, we move one step back in both strings. If the current character in the pattern is *, it can either match no character (skip *) or one or more characters (move left in the text and keep *). By exploring these possibilities recursively, we can determine whether the entire pattern matches the text. C++ //Driver Code Starts #include <iostream> #include <string> using namespace std ; //Driver Code Ends bool wildCardRec (string & txt , string & pat , int n , int m) { // Empty pattern can match with a empty text only if (m == 0) return (n == 0); // Empty text can match with a pattern consisting // of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) if (pat [i] != '*') return false ; return true ; } // Either the characters match or pattern has '?' // move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') return wildCardRec (txt , pat , n - 1 , m - 1); // if the current character of pattern is '*' // first case: It matches with zero character // second case: It matches with one or more characters if (pat [m - 1] == '*') return wildCardRec (txt , pat , n , m - 1) || wildCardRec (txt , pat , n - 1 , m); return false ; } bool wildCard (string & txt , string & pat) { int n = txt . size (); int m = pat . size (); return wildCardRec (txt , pat , n , m); } //Driver Code Starts int main () { string txt = "abcde" ; string pat = "a?c*" ; cout << (wildCard (txt , pat) ? "true" : "false"); return 0 ; } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static boolean wildCardRec (char [] txt , char [] pat , int n , int m) { // Empty pattern can match with an empty text only if (m == 0) return (n == 0); // Empty text can match with a pattern consisting // of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) if (pat [i] != '*') return false ; return true ; } // Either the characters match or pattern has '?' // move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') return wildCardRec (txt , pat , n - 1 , m - 1); // if the current character of pattern is '*' // first case: It matches with zero character // second case: It matches with one or more characters if (pat [m - 1] == '*') return wildCardRec (txt , pat , n , m - 1) || wildCardRec (txt , pat , n - 1 , m); return false ; } static boolean wildCard (String txt , String pat) { int n = txt . length (); int m = pat . length (); return wildCardRec (txt . toCharArray (), pat . toCharArray (), n , m); } //Driver Code Starts public static void main (String [] args) { String txt = "abcde" ; String pat

= "a?c*" ; System . out . println (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends Python def wildCardRec (txt , pat , n , m): # Empty pattern can match with an empty text only if m == 0 : return n == 0 # Empty text can match with a pattern consisting # of '*' only. if n == 0 : for i in range (m): if pat [i] != '*' : return False return True # Either the characters match or pattern has '?' # move to the next in both text and pattern if txt [n - 1] == pat [m - 1] or pat [m - 1] == '?' : return wildCardRec (txt , pat , n - 1 , m - 1) # if the current character of pattern is '*' # first case: It matches with zero character # second case: It matches with one or more characters if pat [m - 1] == '*' : return wildCardRec (txt , pat , n , m - 1) or \ wildCardRec (txt , pat , n - 1 , m) return False def wildCard (txt , pat): n = len (txt) m = len (pat) return wildCardRec (txt , pat , n , m) if __name__ == "__main__" : #Driver Code Starts txt = "abcde" pat = "a?c*" print ("true" if wildCard (txt , pat) else "false") #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static bool wildCardRec (string txt , string pat , int n , int m) { // Empty pattern can match with an empty text only if (m == 0) return (n == 0); // Empty text can match with a pattern consisting // of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) if (pat [i] != '*') return false ; return true ; } // Either the characters match or pattern has '?' // move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') return wildCardRec (txt , pat , n - 1 , m - 1); // if the current character of pattern is '*' // first case: It matches with zero character // second case: It matches with one or more characters if (pat [m - 1] == '*') return wildCardRec (txt , pat , n , m - 1) || wildCardRec (txt , pat , n - 1 , m); return false ; } static bool wildCard (string txt , string pat) { int n = txt . Length ; int m = pat . Length ; return wildCardRec (txt , pat , n , m); } //Driver Code Starts static void Main (string [] args) { string txt = "abcde" ; string pat = "a?c*" ; Console . WriteLine (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends JavaScript function wildCardRec (txt , pat , n , m) { // Empty pattern can match with an empty text only if (m === 0) return (n === 0); // Empty text can match with a pattern consisting // of '*' only. if (n === 0) { for (let i = 0 ; i < m ; i ++) if (pat [i] !== '*') return false ; return true ; } // Either the characters match or pattern has '?' // move to the next in both text and pattern if (txt [n - 1] === pat [m - 1] || pat [m - 1] === '?') return wildCardRec (txt , pat , n - 1 , m - 1); // if the current character of pattern is '*' // first case: It matches with zero character // second case: It matches with one or more characters if (pat [m - 1] === '*') return wildCardRec (txt , pat , n , m - 1) || wildCardRec (txt , pat , n - 1 , m); return false ; } function wildCard (txt , pat) { let n = txt . length ; let m = pat . length ; return wildCardRec (txt , pat , n , m); } // Driver code //Driver Code Starts let txt = "abcde" ; let pat = "a?c*" ; console . log (wildCard (txt , pat) ? "true" : "false"); //Driver Code Ends Output true [Better Approach 1] Using Memoization - O(n*m) Time and O(n*m) Space We can observe that while matching a pattern with wildcards, many overlapping subproblems naturally occur. For instance, when checking if pattern[0..i] matches text[0..j], we might again need to evaluate smaller cases like pattern[0..i-1] with text[0..j] or pattern[0..i] with text[0..j-1], especially when '*' can represent zero or more characters. To handle this efficiently, we use memoization a technique that stores the result of each state (i, j), representing whether the prefixes up to i and j match. If the same state is encountered again, we simply reuse the stored result instead of recomputing it, thus avoiding redundant calculations and improving efficiency. C++ //Driver Code Starts #include <iostream> #include <vector> #include <string> using namespace std ; //Driver Code Ends bool wildCardRec (string & txt , string & pat , int n , int m , vector < vector < int >> & dp) { // Empty pattern can match with a empty text only if (m == 0) return (n == 0); // If result for this sub problem has been // already computed, return it if (dp [n][m] != -1) return dp [n][m]; // Empty text can match with a pattern consisting // of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) if (pat [i] != '*') return dp [n][m] = false ; return dp [n][m] = true ; } // Either the characters match or pattern has '?' // move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') return dp [n][m] = wildCardRec (txt , pat , n - 1 , m - 1 , dp); // if the current character of pattern is '*' // first case: It matches with zero character // second case: It matches with one or more characters if (pat [m - 1] == '*') return dp [n][m] = wildCardRec (txt , pat , n , m - 1 , dp) || wildCardRec (txt , pat , n - 1 , m , dp); return dp [n][m] = false ; } bool wildCard (string & txt , string & pat) { int n = txt . size (); int m = pat . size (); vector < vector < int >> dp (n + 1 , vector < int > (m + 1 , -1)); return wildCardRec (txt , pat , n , m , dp); } //Driver Code Starts int main () { string txt = "abcde" ; string pat = "a?c*" ; cout << (wildCard (txt , pat) ? "true" : "false"); return 0 ; } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static boolean wildCardRec (char [] txt , char [] pat , int n , int m , int [][] dp) { // Empty pattern can match with an empty text only if (m == 0) return (n == 0); // If result for this subproblem has been // already computed, return it if (dp [n][m] != - 1) return dp [n][m] == 1 ; // Empty text can match with a // pattern consisting of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) { if (pat [i] != '*') { dp [n][m] = 0 ; return false ; } } dp [n][m] = 1 ; return true ; } // Either

the characters match or pattern has '?' // Move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') { dp [n][m] = wildCardRec (txt , pat , n - 1 , m - 1 , dp) ? 1 : 0 ; return dp [n][m] == 1 ; } // If the current character of pattern is '*' // First case: It matches with zero character // Second case: It matches with one or more characters if (pat [m - 1] == '*') { dp [n][m] = (wildCardRec (txt , pat , n , m - 1 , dp) || wildCardRec (txt , pat , n - 1 , m , dp)) ? 1 : 0 ; return dp [n][m] == 1 ; } dp [n][m] = 0 ; return false ; } //Driver Code Starts static boolean wildCard (String txt , String pat) { int n = txt . length (); int m = pat . length (); int [][] dp = new int [n + 1][m + 1] ; for (int [] row : dp) Arrays . fill (row , - 1); return wildCardRec (txt . toCharArray (), pat . toCharArray (), n , m , dp); } public static void main (String [] args) { String txt = "abcde" ; String pat = "a?c*" ; System . out . println (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends Python def wildCardRec (txt , pat , n , m , dp): # Empty pattern can match with an empty text only if m == 0 : return n == 0 # If result for this subproblem has been # already computed, return it if dp [n][m] != - 1 : return dp [n][m] # Empty text can match with a pattern consisting # of '*' only. if n == 0 : for i in range (m): if pat [i] != '*' : dp [n][m] = False return False dp [n][m] = True return True # Either the characters match or pattern has '?' # move to the next in both text and pattern if txt [n - 1] == pat [m - 1] or pat [m - 1] == '?' : dp [n][m] = wildCardRec (txt , pat , n - 1 , m - 1 , dp) return dp [n][m] # if the current character of pattern is '*' # first case: It matches with zero character # second case: It matches with one or more characters if pat [m - 1] == '*' : dp [n][m] = wildCardRec (txt , pat , n , m - 1 , dp) \ or wildCardRec (txt , pat , n - 1 , m , dp) return dp [n][m] dp [n][m] = False return False def wildCard (txt , pat): n = len (txt) m = len (pat) dp = [[- 1 for _ in range (m + 1)] for _ in range (n + 1)] return wildCardRec (txt , pat , n , m , dp) if __name__ == "__main__" : #Driver Code Starts txt = "abcde" pat = "a?c*" print ("true" if wildCard (txt , pat) else "false") #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static bool wildCardRec (string txt , string pat , int n , int m , int [,] dp) { // Empty pattern can match with an empty // text only if (m == 0) return (n == 0); // If result for this subproblem has been // already computed, return it if (dp [n , m] != - 1) return dp [n , m] == 1 ; // Empty text can match with a pattern // consisting of '*' only. if (n == 0) { for (int i = 0 ; i < m ; i ++) { if (pat [i] != '*') { dp [n , m] = 0 ; return false ; } } dp [n , m] = 1 ; return true ; } // Either the characters match or pattern has '?' // Move to the next in both text and pattern if (txt [n - 1] == pat [m - 1] || pat [m - 1] == '?') { dp [n , m] = wildCardRec (txt , pat , n - 1 , m - 1 , dp) ? 1 : 0 ; return dp [n , m] == 1 ; } // If the current character of pattern is '*' // First case: It matches with zero character // Second case: It matches with one or more characters if (pat [m - 1] == '*') { dp [n , m] = (wildCardRec (txt , pat , n , m - 1 , dp) || wildCardRec (txt , pat , n - 1 , m , dp)) ? 1 : 0 ; return dp [n , m] == 1 ; } dp [n , m] = 0 ; return false ; } static bool wildCard (string txt , string pat) { int n = txt . Length ; int m = pat . Length ; int [,] dp = new int [n + 1 , m + 1]; // Initialize dp array with -1 for (int i = 0 ; i <= n ; i ++) { for (int j = 0 ; j <= m ; j ++) dp [i , j] = - 1 ; } return wildCardRec (txt , pat , n , m , dp); } //Driver Code Starts static void Main (string [] args) { string txt = "abcde" ; string pat = "a?c*" ; Console . WriteLine (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends JavaScript function wildCardRec (txt , pat , n , m , dp) { // Empty pattern can match with an empty text only if (m === 0) return (n === 0); // If result for this subproblem has been // already computed, return it if (dp [n][m] !== - 1) return dp [n][m] === 1 ; // Empty text can match with a // pattern consisting of '*' only. if (n === 0) { for (let i = 0 ; i < m ; i ++) { if (pat [i] !== '*') { dp [n][m] = 0 ; return false ; } } dp [n][m] = 1 ; return true ; } // Either the characters match or pattern has '?' // Move to the next in both text and pattern if (txt [n - 1] === pat [m - 1] || pat [m - 1] === '?') { dp [n][m] = wildCardRec (txt , pat , n - 1 , m - 1 , dp) ? 1 : 0 ; return dp [n][m] === 1 ; } // If the current character of pattern is '*' // First case: It matches with zero character // Second case: It matches with one or more characters if (pat [m - 1] === '*') { dp [n][m] = (wildCardRec (txt , pat , n , m - 1 , dp) || wildCardRec (txt , pat , n - 1 , m , dp)) ? 1 : 0 ; return dp [n][m] === 1 ; } dp [n][m] = 0 ; return false ; } function wildCard (txt , pat) { const n = txt . length ; const m = pat . length ; const dp = Array . from ({ length : n + 1 }, () => Array (m + 1). fill (- 1)); return wildCardRec (txt , pat , n , m , dp); } // Driver code //Driver Code Starts const txt = "abcde" ; const pat = "a?c*" ; console . log (wildCard (txt , pat) ? "true" : "false"); //Driver Code Ends Output true [Better Approach 2] Using Bottom Up DP (Tabulation) - O(n*m) Time and O(n*m) Space The idea is to convert the recursive relation into a bottom-up table form. We create a 2D DP table dp[i][j], where each cell represents whether the first i characters of the pattern match the first j characters of the text. The base case is that an empty pattern matches an empty text, so dp[0][0] = true. If the pattern begins with *, it can match an empty string, so we propagate that condition along the first column. Then, we fill the table iteratively: When the current pattern character matches the current text character or is a ?, we take the diagonal value dp[i-1][j-1]. When it is *, we can either ignore it (take dp[i-1][j]) or let it match one more character (take dp[i][j-1]). By the end,

dp[n][m] gives the final answer whether the entire pattern matches the text. C++ //Driver Code Starts #include <iostream> #include <string> #include <vector> using namespace std ; //Driver Code Ends bool wildCard (string & txt , string & pat) { int n = txt . size (); int m = pat . size (); vector < vector < bool >> dp (n + 1 , vector < bool > (m + 1 , false)); // empty text and pattern dp [0][0] = true ; // if pattern is empty, // text has to be empty in order to match for (int i = 0 ; i <= n ; i ++) { for (int j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { dp [i][j] = ((j > 0) ? dp [i][j - 1] : true) && (pat [j - 1] == '*'); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { dp [i][j] = dp [i - 1][j - 1]; } else if (pat [j - 1] == '*') { dp [i][j] = dp [i - 1][j] || dp [i][j - 1]; } } } return dp [n][m]; } //Driver Code Starts int main () { string txt = "abcde" ; string pat = "a?c*" ; bool match = wildCard (txt , pat); cout << (match ? "true" : "false"); } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static boolean wildCard (String txt , String pat) { int n = txt . length (); int m = pat . length (); boolean [][] dp = new boolean [n + 1][m + 1] ; // empty text and pattern dp [0][0] = true ; for (int i = 0 ; i <= n ; i ++) { for (int j = 1 ; j <= m ; j ++) { // empty text matches with string of '*'s if (i == 0) { dp [i][j] = (pat . charAt (j - 1) == '*') && dp [i][j - 1] ; } // if char at both index matches or '?' else if (pat . charAt (j - 1) == '?' || txt . charAt (i - 1) == pat . charAt (j - 1)) { dp [i][j] = dp [i - 1][j - 1] ; } // if pattern char is '*' else if (pat . charAt (j - 1) == '*') { dp [i][j] = dp [i - 1][j] || dp [i][j - 1] ; } } } return dp [n][m] ; } //Driver Code Starts public static void main (String [] args) { String txt = "abcde" ; String pat = "a?c*" ; boolean match = wildCard (txt , pat); System . out . println (match ? "true" : "false"); } } //Driver Code Ends Python def wildCard (txt , pat): n = len (txt) m = len (pat) dp = [[False] * (m + 1) for _ in range (n + 1)] # empty text and pattern dp [0][0] = True # if pattern is empty, # text has to be empty in order to match for i in range (n + 1): for j in range (1 , m + 1): # empty text matches with strng of '*'s if i == 0 : dp [i][j] = ((j > 0 and dp [i][j - 1]) or j == 0) \ and (pat [j - 1] == '*') # if char at both index matches elif pat [j - 1] == '?' or txt [i - 1] == pat [j - 1]: dp [i][j] = dp [i - 1][j - 1] elif pat [j - 1] == '*' : dp [i][j] = dp [i - 1][j] or dp [i][j - 1] return dp [n][m] if __name__ == "__main__" : #Driver Code Starts txt = "abcde" pat = "a?c*" match = wildCard (txt , pat) print ("true" if match else "false") #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static bool wildCard (char [] txt , char [] pat) { int n = txt . Length ; int m = pat . Length ; bool [,] dp = new bool [n + 1 , m + 1]; // empty text and pattern dp [0 , 0] = true ; // if pattern is empty, // text has to be empty in order to match for (int i = 0 ; i <= n ; i ++) { for (int j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { dp [i , j] = ((j > 0) ? dp [i , j - 1] : true) && (pat [j - 1] == '*'); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { dp [i , j] = dp [i - 1 , j - 1]; } else if (pat [j - 1] == '*') { dp [i , j] = dp [i - 1 , j] || dp [i , j - 1]; } } } return dp [n , m]; } //Driver Code Starts public static void Main (string [] args) { string txt = "abcde" ; string pat = "a?c*" ; bool match = wildCard (txt . ToCharArray (), pat . ToCharArray ()); Console . WriteLine (match ? "true" : "false"); } } //Driver Code Ends JavaScript function wildCard (txt , pat) { const n = txt . length ; const m = pat . length ; const dp = Array . from ({ length : n + 1 }, () => Array (m + 1). fill (false)); // empty text and pattern dp [0][0] = true ; // if pattern is empty, // text has to be empty in order to match for (let i = 0 ; i <= n ; i ++) { for (let j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { dp [i][j] = ((j > 0 ? dp [i][j - 1] : true) && (pat [j - 1] == '*')); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { dp [i][j] = dp [i - 1][j - 1]; } else if (pat [j - 1] == '*') { dp [i][j] = dp [i - 1][j] || dp [i][j - 1]; } } } return dp [n][m]; } // Driver code //Driver Code Starts const txt = "abcde" ; const pat = "a?c*" ; const match = wildCard (txt , pat); console . log (match ? "true" : "false"); //Driver Code Ends Output true [Efficient Approach] Using Space Optimized DP - O(n*m) Time and O(m) Space In the above approach, we can optimize further by observing that dp[i][j] depends only on the previous row (i-1) and the current row. This means we don’t need to store the entire 2D DP table. We use two 1D arrays — one (prev) for the previous row and another (curr) for the current row. For each character in the pattern and text, we fill curr[j] using values from prev and already computed entries in curr. The base cases handle situations like matching an empty text or patterns that consist only of *. After completing one row, we assign prev = curr and move on to the next pattern character. Finally, the result for the entire match is available in prev[m]. C++ //Driver Code Starts #include <iostream> #include <string> #include <vector> using namespace std ; //Driver Code Ends bool wildCard (string & txt , string & pat) { int n = txt . size (); int m = pat . size (); vector < bool > prev (m + 1 , false); for (int i = 0 ; i <= n ; i ++) { vector < bool > curr (m + 1 , false); // empty text and pattern if (i == 0) curr [0] = true ; for (int j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { curr [j] = ((j > 0) ? (curr [j - 1]) : true) && (pat [j - 1] == '*'); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { curr [j] = prev [j - 1]; } else if (pat [j - 1] == '*') { curr [j] = prev [j] || curr [j - 1]; } } prev = curr ; } return prev [m]; } //Driver Code

Starts int main () { string txt = "abcde" ; string pat = "a?c*" ; bool match = wildCard (txt , pat); cout << (match ? "true" : "false"); } //Driver Code Ends Java //Driver Code Starts import java.util.Arrays ; class GFG { //Driver Code Ends static boolean wildCard (String txt , String pat) { int n = txt . length (); int m = pat . length (); boolean [] prev = new boolean [m + 1] ; for (int i = 0 ; i <= n ; i ++) { boolean [] curr = new boolean [m + 1] ; // empty text and pattern if (i == 0) curr [0] = true ; for (int j = 1 ; j <= m ; j ++) { // empty text matches with string of '*'s if (i == 0) { curr [j] = (pat . charAt (j - 1) == '*') && curr [j - 1] ; } // if char at both index matches or '?' else if (pat . charAt (j - 1) == '?' || txt . charAt (i - 1) == pat . charAt (j - 1)) { curr [j] = prev [j - 1] ; } // if pattern char is '*' else if (pat . charAt (j - 1) == '*') { curr [j] = prev [j] || curr [j - 1] ; } } prev = curr ; } return prev [m] ; } //Driver Code Starts public static void main (String [] args) { String txt = "abcde" ; String pat = "a?c*" ; boolean match = wildCard (txt , pat); System . out . println (match ? "true" : "false"); } } //Driver Code Ends Python def wildCard (txt , pat): n = len (txt) m = len (pat) prev = [False] * (m + 1) for i in range (n + 1): curr = [False] * (m + 1) # empty text and pattern if i == 0 : curr [0] = True for j in range (1 , m + 1): # empty text matches with strng of '*'s if i == 0 : curr [j] = ((j > 0 and curr [j - 1]) or j == 0) \ and (pat [j - 1] == '*') # if char at both index matches elif pat [j - 1] == '?' or txt [i - 1] == pat [j - 1]: curr [j] = prev [j - 1] elif pat [j - 1] == '*' : curr [j] = prev [j] or curr [j - 1] prev = curr return prev [m] if __name__ == "__main__" : #Driver Code Starts txt = "abcde" pat = "a?c*" match = wildCard (txt , pat) print ("true" if match else "false") #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static bool wildCard (char [] txt , char [] pat) { int n = txt . Length ; int m = pat . Length ; bool [] prev = new bool [m + 1]; for (int i = 0 ; i <= n ; i ++) { bool [] curr = new bool [m + 1]; // empty text and pattern if (i == 0) curr [0] = true ; for (int j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { curr [j] = ((j > 0) ? (curr [j - 1]) : true) && (pat [j - 1] == '*'); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { curr [j] = prev [j - 1]; } else if (pat [j - 1] == '*') { curr [j] = prev [j] || curr [j - 1]; } } prev = curr ; } return prev [m]; } //Driver Code Starts public static void Main (string [] args) { string txt = "abcde" ; string pat = "a?c*" ; bool match = wildCard (txt . ToCharArray (), pat . ToCharArray ()); Console . WriteLine (match ? "true" : "false"); } } //Driver Code Ends JavaScript function wildCard (txt , pat) { const n = txt . length ; const m = pat . length ; let prev = new Array (m + 1). fill (false); for (let i = 0 ; i <= n ; i ++) { let curr = new Array (m + 1). fill (false); // empty text and pattern if (i == 0) curr [0] = true ; for (let j = 1 ; j <= m ; j ++) { // empty text matches with strng of '*'s if (i == 0) { curr [j] = ((j > 0 ? curr [j - 1] : true) && (pat [j - 1] == '*')); } // if char at both index matches else if (pat [j - 1] == '?' || txt [i - 1] == pat [j - 1]) { curr [j] = prev [j - 1]; } else if (pat [j - 1] == '*') { curr [j] = prev [j] || curr [j - 1]; } } prev = curr ; } return prev [m]; } // Driver code //Driver Code Starts const txt = "abcde" ; const pat = "a?c*" ; const match = wildCard (txt , pat); console . log (match ? "true" : "false"); //Driver Code Ends Output true [Expected Approach] Simple Traversal - O(n * m) Time and O(1) Space In this approach, we match the text and pattern using two pointers, handling * and ? without extra space. The * can represent any sequence of characters, while ? matches exactly one. When we encounter a *, we record its position in the pattern and the corresponding position in the text. If a mismatch occurs later, we backtrack by returning to the last * — we move the pattern pointer to just after * and advance the text pointer by one from the last recorded match position. This effectively treats the * as matching one more character in the text, allowing us to explore all valid possibilities efficiently. C++ //Driver Code Starts #include <iostream> using namespace std ; //Driver Code Ends bool wildCard (string & txt , string & pat) { int n = txt . length (); int m = pat . length (); int i = 0 , j = 0 , startIndex = -1 , match = 0 ; while (i < n) { // Characters match or '?' in // pattern matches any character. if (j < m && (pat [j] == '?' || pat [j] == txt [i])) { i ++ ; j ++ ; } else if (j < m && pat [j] == '*') { // Wildcard character '*', mark the current // position in the pattern and the text as a // proper match. startIndex = j ; match = i ; j ++ ; } else if (startIndex != -1) { // No match, but a previous wildcard was found. // Backtrack to the last '*' character position // and try for a different match. j = startIndex + 1 ; match ++ ; i = match ; } else { // If none of the above cases comply, the // pattern does not match. return false ; } } // Consume remaining '*' // characters in given pattern. while (j < m && pat [j] == '*') { j ++ ; } // If we have reached the end of both the pattern and // the text, the pattern matches the text. return j == m ; } //Driver Code Starts int main () { string txt = "abcde" ; string pat = "a?c*" ; cout << (wildCard (txt , pat) ? "true" : "false"); } //Driver Code Ends Java //Driver Code Starts class GFG { //Driver Code Ends static boolean wildCard (String txt , String pat) { int n = txt . length (); int m = pat . length (); int i = 0 , j = 0 , startIndex = - 1 , match = 0 ; while (i < n) { // Characters match or '?' in // pattern matches any character. if (j < m && (pat . charAt (j) == '?' || pat . charAt (j) == txt . charAt (i))) { i ++ ; j ++ ; } else if (j < m && pat . charAt (j) == '*') { // Wildcard character '*', mark the current // position in the pattern and the text as a // proper match. startIndex = j ; match = i ; j ++ ; } else

if (startIndex != - 1) { // No match, but a previous wildcard was found. // Backtrack to the last '*' character position // and try for a different match. j = startIndex + 1 ; match ++ ; i = match ; } else { // If none of the above cases comply, the // pattern does not match. return false ; } } // Consume remaining '*' // characters in given pattern. while (j < m && pat . charAt (j) == '*') { j ++ ; } // If we have reached the end of both the pattern and // the text, the pattern matches the text. return j == m ; } //Driver Code Starts public static void main (String [] args) { String txt = "abcde" ; String pat = "a?c*" ; System . out . println (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends Python def wildCard (txt , pat): n = len (txt) m = len (pat) i = 0 j = 0 startIndex = - 1 match = 0 while i < n : # Characters match or '?' in # pattern matches any character. if j < m and (pat [j] == '?' or pat [j] == txt [i]): i += 1 j += 1 elif j < m and pat [j] == '*' : # Wildcard character '*', mark the current # position in the pattern and the text as a # proper match. startIndex = j match = i j += 1 elif startIndex != - 1 : # No match, but a previous wildcard was found. # Backtrack to the last '*' character position # and try for a different match. j = startIndex + 1 match += 1 i = match else : # If none of the above cases comply, the # pattern does not match. return False # Consume remaining '*' # characters in given pattern. while j < m and pat [j] == '*' : j += 1 # If we have reached the end of both the pattern and # the text, the pattern matches the text. return j == m if __name__ == "__main__" : #Driver Code Starts txt = "abcde" pat = "a?c*" print ("true" if wildCard (txt , pat) else "false") #Driver Code Ends C# //Driver Code Starts using System ; class GFG { //Driver Code Ends static bool wildCard (string txt , string pat) { int n = txt . Length ; int m = pat . Length ; int i = 0 , j = 0 , startIndex = - 1 , match = 0 ; while (i < n) { // Characters match or '?' in // pattern matches any character. if (j < m && (pat [j] == '?' || pat [j] == txt [i])) { i ++ ; j ++ ; } else if (j < m && pat [j] == '*') { // Wildcard character '*', mark the current // position in the pattern and the text as a // proper match. startIndex = j ; match = i ; j ++ ; } else if (startIndex != - 1) { // No match, but a previous wildcard was found. // Backtrack to the last '*' character position // and try for a different match. j = startIndex + 1 ; match ++ ; i = match ; } else { // If none of the above cases comply, the // pattern does not match. return false ; } } // Consume remaining '*' // characters in given pattern. while (j < m && pat [j] == '*') { j ++ ; } // If we have reached the end of both the pattern and // the text, the pattern matches the text. return j == m ; } //Driver Code Starts static void Main (string [] args) { string txt = "abcde" ; string pat = "a?c*" ; Console . WriteLine (wildCard (txt , pat) ? "true" : "false"); } } //Driver Code Ends JavaScript function wildCard (txt , pat) { let n = txt . length ; let m = pat . length ; let i = 0 , j = 0 , startIndex = - 1 , match = 0 ; while (i < n) { // Characters match or '?' in // pattern matches any character. if (j < m && (pat [j] === '?' || pat [j] === txt [i])) { i ++ ; j ++ ; } else if (j < m && pat [j] === '*') { // Wildcard character '*', mark the current // position in the pattern and the text as a // proper match. startIndex = j ; match = i ; j ++ ; } else if (startIndex !== - 1) { // No match, but a previous wildcard was found. // Backtrack to the last '*' character position // and try for a different match. j = startIndex + 1 ; match ++ ; i = match ; } else { // If none of the above cases comply, the // pattern does not match. return false ; } } // Consume remaining '*' // characters in given pattern. while (j < m && pat [j] === '*') { j ++ ; } // If we have reached the end of both the pattern and // the text, the pattern matches the text. return j === m ; } // Driver code //Driver Code Starts let txt = "abcde" ; let pat = "a?c*" ; console . log (wildCard (txt , pat) ? "true" : "false"); //Driver Code Ends Output true Comment Article Tags: Article Tags: Strings Dynamic Programming Pattern Searching DSA Microsoft Amazon Walmart Zoho InMobi United Health Group Ola Cabs + 7 More