# String matching where one string contains wildcard characters

Given two strings **wild** and **pattern** where wild string may contain wild card characters and pattern string is a normal string. Determine if the two strings match. The following are the allowed wild card characters in first string :-

\* --> Matches with 0 or more instances of any

character or set of characters.

? --> Matches with any one character.

**Example 1:**

**Input:** wild = ge\*ks

pattern = geeks

**Output:** Yes

**Explanation:** Replace the '\*' with 'e' to obtain

the string.

**Example 2:**

**Input:** wild =ge?ks\*

pattern = geeksforgeeks

**Output:** Yes

**Explanation:** Replace '?' with 'e' and '\*' with

'forgeeks' and it will be same as pattern.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **match()**which takesthe string wild and pattern as input parameters and returns true if the string wild can be made equal to the string pattern, otherwise, returns false.

**Expected Time Complexity:** O(length of wild string + length of pattern string)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1<=length of the two string<=10^3

## Solution:

// A C program to match wild card characters

#include <stdio.h>

#include <stdbool.h>

// The main function that checks if two given strings

// match. The first string may contain wildcard characters

bool match(char \*first, char \* second)

{

// If we reach at the end of both strings, we are done

if (\*first == '\0' && \*second == '\0')

return true;

// Make sure that the characters after '\*' are present

// in second string. This function assumes that the first

// string will not contain two consecutive '\*'

if (\*first == '\*' && \*(first+1) != '\0' && \*second == '\0')

return false;

// If the first string contains '?', or current characters

// of both strings match

if (\*first == '?' || \*first == \*second)

return match(first+1, second+1);

// If there is \*, then there are two possibilities

// a) We consider current character of second string

// b) We ignore current character of second string.

if (\*first == '\*')

return match(first+1, second) || match(first, second+1);

return false;

}

// A function to run test cases

void test(char \*first, char \*second)

{ match(first, second)? puts("Yes"): puts("No"); }

// Driver program to test above functions

int main()

{

test("g\*ks", "geeks"); // Yes

test("ge?ks\*", "geeksforgeeks"); // Yes

test("g\*k", "gee"); // No because 'k' is not in second

test("\*pqrs", "pqrst"); // No because 't' is not in first

test("abc\*bcd", "abcdhghgbcd"); // Yes

test("abc\*c?d", "abcd"); // No because second must have 2

// instances of 'c'

test("\*c\*d", "abcd"); // Yes

test("\*?c\*d", "abcd"); // Yes

return 0;

}

**Output:**

Yes

Yes

No

No

Yes

No

Yes

Yes

**Approach using DP:**

Each occurrence of ‘?’ character in wildcard pattern can be replaced with any other character and each occurrence of ‘\*’ with a sequence of characters such that the wildcard pattern becomes identical to the input string after replacement.

Let’s consider any character in the pattern.

**Case 1: The character is ‘\*’**   
Here two cases arise

1. We can ignore ‘\*’ character and move to next character in the Pattern.
2. ‘\*’ character matches with one or more characters in Text. Here we will move to next character in the string.

**Case 2: The character is ‘?’**   
We can ignore current character in Text and move to next character in the Pattern and Text.

**Case 3: The character is not a wildcard character**   
If current character in Text matches with current character in Pattern, we move to next character in the Pattern and Text. If they do not match, wildcard pattern and Text do not match.  
We can use Dynamic Programming to solve this problem –   
Let **T[i][j]** is true if first i characters in given string matches the first j characters of pattern.

**DP Initialization:**

// both text and pattern are null

T[0][0] = true;

// pattern is null

T[i][0] = false;

// text is null

T[0][j] = T[0][j - 1] if pattern[j – 1] is '\*'

**DP relation :**

// If current characters match, result is same as

// result for lengths minus one. Characters match

// in two cases:

// a) If pattern character is '?' then it matches

// with any character of text.

// b) If current characters in both match

if ( pattern[j – 1] == ‘?’) ||

(pattern[j – 1] == text[i - 1])

T[i][j] = T[i-1][j-1]

// If we encounter ‘\*’, two choices are possible-

// a) We ignore ‘\*’ character and move to next

// character in the pattern, i.e., ‘\*’

// indicates an empty sequence.

// b) '\*' character matches with ith character in

// input

else if (pattern[j – 1] == ‘\*’)

T[i][j] = T[i][j-1] || T[i-1][j]

else // if (pattern[j – 1] != text[i - 1])

T[i][j] = false

Below is the implementation of the above Dynamic Programming approach.

// C++ program to implement wildcard

// pattern matching algorithm

#include <bits/stdc++.h>

using namespace std;

// Function that matches input str with

// given wildcard pattern

bool strmatch(char str[], char pattern[], int n, int m)

{

// empty pattern can only match with

// empty string

if (m == 0)

return (n == 0);

// lookup table for storing results of

// subproblems

bool lookup[n + 1][m + 1];

// initialize lookup table to false

memset(lookup, false, sizeof(lookup));

// empty pattern can match with empty string

lookup[0][0] = true;

// Only '\*' can match with empty string

for (int j = 1; j <= m; j++)

if (pattern[j - 1] == '\*')

lookup[0][j] = lookup[0][j - 1];

// fill the table in bottom-up fashion

for (int i = 1; i <= n; i++) {

for (int j = 1; j <= m; j++) {

// Two cases if we see a '\*'

// a) We ignore ‘\*’ character and move

// to next character in the pattern,

// i.e., ‘\*’ indicates an empty sequence.

// b) '\*' character matches with ith

// character in input

if (pattern[j - 1] == '\*')

lookup[i][j]

= lookup[i][j - 1] || lookup[i - 1][j];

// Current characters are considered as

// matching in two cases

// (a) current character of pattern is '?'

// (b) characters actually match

else if (pattern[j - 1] == '?'

|| str[i - 1] == pattern[j - 1])

lookup[i][j] = lookup[i - 1][j - 1];

// If characters don't match

else

lookup[i][j] = false;

}

}

return lookup[n][m];

}

int main()

{

char str[] = "baaabab";

char pattern[] = "\*\*\*\*\*ba\*\*\*\*\*ab";

// char pattern[] = "ba\*\*\*\*\*ab";

// char pattern[] = "ba\*ab";

// char pattern[] = "a\*ab";

// char pattern[] = "a\*\*\*\*\*ab";

// char pattern[] = "\*a\*\*\*\*\*ab";

// char pattern[] = "ba\*ab\*\*\*\*";

// char pattern[] = "\*\*\*\*";

// char pattern[] = "\*";

// char pattern[] = "aa?ab";

// char pattern[] = "b\*b";

// char pattern[] = "a\*a";

// char pattern[] = "baaabab";

// char pattern[] = "?baaabab";

// char pattern[] = "\*baaaba\*";

if (strmatch(str, pattern, strlen(str),

strlen(pattern)))

cout << "Yes" << endl;

else

cout << "No" << endl;

return 0;

}

**Output**

Yes

**Time complexity:** O(m x n)   
**Auxiliary space:** O(m x n)

**DP Memoization solution:-**

// C++ program to implement wildcard

// pattern matching algorithm

#include <bits/stdc++.h>

using namespace std;

// Function that matches input str with

// given wildcard pattern

vector<vector<int> > dp;

int finding(string& s, string& p, int n, int m)

{

// return 1 if n and m are negative

if (n < 0 && m < 0)

return 1;

// return 0 if m is negative

if (m < 0)

return 0;

// return n if n is negative

if (n < 0)

{

// while m is positive

while (m >= 0)

{

if (p[m] != '\*')

return 0;

m--;

}

return 1;

}

// if dp state is not visited

if (dp[n][m] == -1)

{

if (p[m] == '\*')

{

return dp[n][m] = finding(s, p, n - 1, m)

|| finding(s, p, n, m - 1);

}

else

{

if (p[m] != s[n] && p[m] != '?')

return dp[n][m] = 0;

else

return dp[n][m]

= finding(s, p, n - 1, m - 1);

}

}

// return dp[n][m] if dp state is previsited

return dp[n][m];

}

bool isMatch(string s, string p)

{

dp.clear();

// resize the dp array

dp.resize(s.size() + 1, vector<int>(p.size() + 1, -1));

return dp[s.size()][p.size()]

= finding(s, p, s.size() - 1, p.size() - 1);

}

// Driver code

int main()

{

string str = "baaabab";

string pattern = "\*\*\*\*\*ba\*\*\*\*\*ab";

// char pattern[] = "ba\*\*\*\*\*ab";

// char pattern[] = "ba\*ab";

// char pattern[] = "a\*ab";

// char pattern[] = "a\*\*\*\*\*ab";

// char pattern[] = "\*a\*\*\*\*\*ab";

// char pattern[] = "ba\*ab\*\*\*\*";

// char pattern[] = "\*\*\*\*";

// char pattern[] = "\*";

// char pattern[] = "aa?ab";

// char pattern[] = "b\*b";

// char pattern[] = "a\*a";

// char pattern[] = "baaabab";

// char pattern[] = "?baaabab";

// char pattern[] = "\*baaaba\*";

if (isMatch(str, pattern))

cout << "Yes" << endl;

else

cout << "No" << endl;

return 0;

}

**Output**

Yes

**Time complexity**: O(m x n).   
**Auxiliary space:**O(m x n).

**Further Improvements:**   
We can improve space complexity by making use of the fact that we only uses the result from last row.   
One more improvement is you can merge consecutive ‘\*’ in the pattern to single ‘\*’ as they mean the same thing. For example for pattern “\*\*\*\*\*ba\*\*\*\*\*ab”, if we merge consecutive stars, the resultant string will be “\*ba\*ab”. So, value of m is reduced from 14 to 6.

**Linear Time and Constant Space solution:**

For applying the optimization, we will at the first note the **BASE CASE** which says, if the length of the pattern is zero then answer will be true only if the length of the text with which we have to match the pattern is also zero.  
**Algorithm:** 

1. Let i be the marker to point at the current character of the text.   
   Let j be the marker to point at the current character of the pattern.   
   Let index\_txt be the marker to point at the character of text on which we encounter ‘\*’ in the pattern.   
   Let index\_pat be the marker to point at the position of ‘\*’ in the pattern.
2. At any instant, if we observe that txt[i] == pat[j], then we increment both i and j as no operation needs to be performed in this case.
3. If we encounter pat[j] == ‘?’, then it resembles the case mentioned in step – (2) as ‘?’ has the property to match with any single character.
4. If we encounter pat[j] == ‘\*’, then we update the value of index\_txt and index\_pat as ‘\*’ has the property to match any sequence of characters (including the empty sequence) and we will increment the value of j to compare next character of pattern with the current character of the text. (As character represented by i has not been answered yet).
5. Now if txt[i] == pat[j], and we have encountered a ‘\*’ before, then it means that ‘\*’ included the empty sequence, else if txt[i] != pat[j], a character needs to be provided by ‘\*’ so that current character matching takes place, then i needs to be incremented as it is answered now but the character represented by j still needs to be answered, therefore, j = index\_pat + 1, i = index\_txt + 1 (as ‘\*’ can capture other characters as well), index\_txt++ (as current character in text is matched).
6. If step – (5) is not valid, that means txt[i] != pat[j], also we have not encountered a ‘\*’ that means it is not possible for the pattern to match the string. (return false).
7. Check whether j reached its final value or not, then return the final answer.

**Let us see the above algorithm in action, then we will move to the coding section:**  
text = “baaabab”   
pattern = “\*\*\*\*\*ba\*\*\*\*\*ab”  
**NOW APPLYING THE ALGORITHM**  
Step – (1) : i = 0 (i –> ‘b’)   
j = 0 (j –> ‘\*’)   
index\_txt = -1   
index\_pat = -1  
**NOTE: LOOP WILL RUN TILL i REACHES ITS FINAL**  
**VALUE OR THE ANSWER BECOMES FALSE MIDWAY.**  
**FIRST COMPARISON :-**  
As we see here that pat[j] == ‘\*’, therefore directly jumping on to step – (4).   
Step – (4) : index\_txt = i (index\_txt –> ‘b’)   
index\_pat = j (index\_pat –> ‘\*’)   
j++ (j –> ‘\*’)  
After four more comparisons : i = 0 (i –> ‘b’)   
j = 5 (j –> ‘b’)   
index\_txt = 0 (index\_txt –> ‘b’)   
index\_pat = 4 (index\_pat –> ‘\*’)  
**SIXTH COMPARISON :-**  
As we see here that txt[i] == pat[j], but we already encountered ‘\*’ therefore using step – (5).   
Step – (5) : i = 1 (i –> ‘a’)   
j = 6 (j –> ‘a’)   
index\_txt = 0 (index\_txt –> ‘b’)   
index\_pat = 4 (index\_pat –> ‘\*’)  
**SEVENTH COMPARISON :-**  
Step – (5) : i = 2 (i –> ‘a’)   
j = 7 (j –> ‘\*’)   
index\_txt = 0 (index\_txt –> ‘b’)   
index\_pat = 4 (index\_pat –> ‘\*’)  
**EIGTH COMPARISON :-**  
Step – (4) : i = 2 (i –> ‘a’)   
j = 8 (j –> ‘\*’)   
index\_txt = 2 (index\_txt –> ‘a’)   
index\_pat = 7 (index\_pat –> ‘\*’)  
After four more comparisons : i = 2 (i –> ‘a’)   
j = 12 (j –> ‘a’)   
index\_txt = 2 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**THIRTEENTH COMPARISON :-**  
Step – (5) : i = 3 (i –> ‘a’)   
j = 13 (j –> ‘b’)   
index\_txt = 2 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**FOURTEENTH COMPARISON :-**  
Step – (5) : i = 3 (i –> ‘a’)   
j = 12 (j –> ‘a’)   
index\_txt = 3 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**FIFTEENTH COMPARISON :-**  
Step – (5) : i = 4 (i –> ‘b’)   
j = 13 (j –> ‘b’)   
index\_txt = 3 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**SIXTEENTH COMPARISON :-**  
Step – (5) : i = 5 (i –> ‘a’)   
j = 14 (j –> end)   
index\_txt = 3 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**SEVENTEENTH COMPARISON :-**  
Step – (5) : i = 4 (i –> ‘b’)   
j = 12 (j –> ‘a’)   
index\_txt = 4 (index\_txt –> ‘b’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**EIGHTEENTH COMPARISON :-**  
Step – (5) : i = 5 (i –> ‘a’)   
j = 12 (j –> ‘a’)   
index\_txt = 5 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**NINETEENTH COMPARISON :-**  
Step – (5) : i = 6 (i –> ‘b’)   
j = 13 (j –> ‘b’)   
index\_txt = 5 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**TWENTIETH COMPARISON :-**  
Step – (5) : i = 7 (i –> end)   
j = 14 (j –> end)   
index\_txt = 5 (index\_txt –> ‘a’)   
index\_pat = 11 (index\_pat –> ‘\*’)  
**NOTE : NOW WE WILL COME OUT OF LOOP TO RUN STEP – 7.**  
Step – (7) : j is already present at its end position, therefore answer is true.

Below is the implementation of the above approach:

// C++ program to implement wildcard

// pattern matching algorithm

#include <bits/stdc++.h>

using namespace std;

// Function that matches input text

// with given wildcard pattern

bool strmatch(char txt[], char pat[],

int n, int m)

{

// empty pattern can only

// match with empty string.

// Base Case :

if (m == 0)

return (n == 0);

// step-1 :

// initialize markers :

int i = 0, j = 0, index\_txt = -1,

index\_pat = -1;

while (i < n)

{

// For step - (2, 5)

if (j < m && txt[i] == pat[j])

{

i++;

j++;

}

// For step - (3)

else if (j < m && pat[j] == '?')

{

i++;

j++;

}

// For step - (4)

else if (j < m && pat[j] == '\*')

{

index\_txt = i;

index\_pat = j;

j++;

}

// For step - (5)

else if (index\_pat != -1)

{

j = index\_pat + 1;

i = index\_txt + 1;

index\_txt++;

}

// For step - (6)

else

{

return false;

}

}

// For step - (7)

while (j < m && pat[j] == '\*')

{

j++;

}

// Final Check

if (j == m)

{

return true;

}

return false;

}

// Driver code

int main()

{

char str[] = "baaabab";

char pattern[] = "\*\*\*\*\*ba\*\*\*\*\*ab";

// char pattern[] = "ba\*\*\*\*\*ab";

// char pattern[] = "ba\*ab";

// char pattern[] = "a\*ab";

if (strmatch(str, pattern,

strlen(str), strlen(pattern)))

cout << "Yes" << endl;

else

cout << "No" << endl;

char pattern2[] = "a\*\*\*\*\*ab";

if (strmatch(str, pattern2,

strlen(str), strlen(pattern2)))

cout << "Yes" << endl;

else

cout << "No" << endl;

return 0;

}

**Output:**

Yes

No

**Complexity Analysis:**

* **Time Complexity:** O(m + n), where ‘m’ and ‘n’ are the lengths of text and pattern respectively.
* **Auxiliary Space:** O(1).   
  No use of any data structure for storing values