<https://leetcode.com/problems/wildcard-matching/description/>

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 '\*'.

**Attempt 1: 2023-10-20**

**Solution 1: Native DFS (10 min, TLE 1638/1811)**

class Solution {

public boolean isMatch(String s, String p) {

return helper(s, 0, p, 0);

}

// '?' Matches any single character.

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

private boolean helper(String s, int i, String p, int j) {

if(j == p.length()) {

return i == s.length();

}

// If the current pattern character is '\*' then we have two options,

// either to move j forward and don't use it for matching or we can

// match and move the string index and keep the pattern index at j only

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

// '\*' means empty sequence, we can ignore char '\*' at position j in p

if(helper(s, i, p, j + 1)) {

return true;

}

// we match s[i] and p[j] but only move i forward and keep j as is

// since '\*' can match any sequence of characters

if(i < s.length() && helper(s, i + 1, p, j)) {

return true;

}

} else {

// If p[j] not '\*', the current charcter of pattern and string are

// equal or not, they would be equal if either s[i]==p[j] or p[j]='?'

// Note: The 'i < s.length()' is mandatory, during recursion, i has

// chance to be larger than j because of in certain recursion we will

// keep j no change but increase i -> helper(s, i + 1, p, j)

// Test out: s = "acdcb", p = "a\*c?b"

// No need to check 'j < p.length()' because in base condition when

// 'j == p.length()' we will directly return, no further logic happen

if(i < s.length() && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '?')) {

return helper(s, i + 1, p, j + 1);

}

}

return false;

}

}

Time Complexity: Recursion O(2^N), \* matches 0 or more chars.

Space Complexity: O(2^N)

**Solution 2: DFS + Memoization (10 min)**

class Solution {

public boolean isMatch(String s, String p) {

Boolean[][] memo = new Boolean[s.length() + 1][p.length() + 1];

return helper(s, 0, p, 0, memo);

}

// '?' Matches any single character.

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

private boolean helper(String s, int i, String p, int j, Boolean[][] memo) {

if(j == p.length()) {

return i == s.length();

}

if(memo[i][j] != null) {

return memo[i][j];

}

// If the current pattern character is '\*' then we have two options,

// either to move j forward and don't use it for matching or we can

// match and move the string index and keep the pattern index at j only

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

// '\*' means empty sequence, we can ignore char '\*' at position j in p

if(helper(s, i, p, j + 1, memo)) {

return memo[i][j] = true;

}

// we match s[i] and p[j] but only move i forward and keep j as is

// since '\*' can match any sequence of characters

if(i < s.length() && helper(s, i + 1, p, j, memo)) {

return memo[i][j] = true;

}

} else {

// If p[j] not '\*', the current charcter of pattern and string are

// equal or not, they would be equal if either s[i]==p[j] or p[j]='?'

// Note: The 'i < s.length()' is mandatory, during recursion, i has

// chance to be larger than j because of in certain recursion we will

// keep j no change but increase i -> helper(s, i + 1, p, j)

// Test out: s = "acdcb", p = "a\*c?b"

// No need to check 'j < p.length()' because in base condition when

// 'j == p.length()' we will directly return, no further logic happen

if(i < s.length() && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '?')) {

return helper(s, i + 1, p, j + 1, memo);

}

}

return memo[i][j] = false;

}

}

Time Complexity:

https://leetcode.com/problems/wildcard-matching/solutions/477823/recursive-dfs-solution-with-memoization-top-down-approach/comments/918686

What is the time complexity of this solution? I always find it hard to analyze recursive + memo solutions. Any help on this part, please?

It is similar to bottom-up that is O(M\*N) where M = len(input\_string) N = len(pattern).

to find time complexity of top down approach try to figure out maximum possible states. here (i,j) is a state and there are MN different possiblities of (i,j) where 0<=i<=M, 0<=j<=N.

why total possible states as time complexity?

because, if recursively any state is repeated we use memorized answer and return it in O(1).

**Solution 3: DP (10 min, 标准顶底之术)**

class Solution {

public boolean isMatch(String s, String p) {

int sLen = s.length();

int pLen = p.length();

boolean[][] dp = new boolean[sLen + 1][pLen + 1];

// Since \* in p can match any sequence of characters

// (including the empty sequence), so empty

// string(i == s.length()) may match p, so we have

// -> if(j == pLen) {return i == sLen}

// (1) p end, s end, true

dp[sLen][pLen] = true;

// (2) p end, s not end, false

for(int i = 0; i < sLen; i++) {

dp[i][pLen] = false;

}

for(int i = sLen; i >= 0; i--) {

for(int j = pLen - 1; j >= 0; j--) {

// If the current pattern character is '\*' then we have two options,

// either to move j forward and don't use it for matching or we can

// match and move the string index and keep the pattern index at j only

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

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

if(i < sLen) {

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

}

} else {

// If p[j] not '\*', the current charcter of pattern and string are

// equal or not, they would be equal if either s[i]==p[j] or p[j]='?'

// Note: The 'i < s.length()' is mandatory, during recursion, i has

// chance to be larger than j because of in certain recursion we will

// keep j no change but increase i -> helper(s, i + 1, p, j)

// Test out: s = "acdcb", p = "a\*c?b"

// No need to check 'j < p.length()' because in base condition when

// 'j == p.length()' we will directly return, no further logic happen

if(i < sLen && (s.charAt(i) == p.charAt(j) || p.charAt(j) == '?')) {

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

}

}

}

}

return dp[0][0];

}

}

Time Complexity: O(M\*N), where M = len(input\_string) N = len(pattern).

Space Complexity: O(M\*N)

=========================================================================

One example:

P = \*a\*b

S = adceb

[P] \* a \* b -

[S]

a T T T F F

d F F T F F

c F F T F F

e F F T F F

b F F T T F

- F F F F T

**Refer to**

<https://leetcode.com/problems/wildcard-matching/solutions/17859/evolve-from-brute-force-to-optimal/>

* Recursion O(2^n), \* matches 0 or more chars

public boolean isMatch(String s, String p) {

return isMatch(0, s, 0, p);

}

private boolean isMatch(int i, String s, int j, String p) {

int sn = s.length(), pn = p.length();

if(j==pn) {

return i==sn;

}

char pj = p.charAt(j);

if(i<sn && pj == '?') {

return isMatch(i+1, s, j+1, p);

} else if(pj == '\*') {

return isMatch(i,s,j+1,p) || i<sn && isMatch(i+1,s,j,p);

} else if(i<sn && pj == s.charAt(i)) {

return isMatch(i+1, s, j+1, p);

}

return false;

}

* Memorization O(n^2), memorization turns out to be faster than dp. I think it is because dfs terminates as soon as a match is found but dp is always n^2.

Boolean[][] mem;

public boolean isMatch(String s, String p) {

mem = new Boolean[s.length()+1][p.length()+1];

return isMatch(0, s, 0, p);

}

private boolean isMatch(int i, String s, int j, String p) {

int sn = s.length(), pn = p.length();

if(j==pn) {

return i==sn;

}

if(mem[i][j] != null) {

return mem[i][j];

}

char pj = p.charAt(j);

if(i<sn && pj == '?') {

return mem[i][j] = isMatch(i+1, s, j+1, p);

} else if(pj == '\*') {

return mem[i][j] = isMatch(i,s,j+1,p) || i<sn && isMatch(i+1,s,j,p);

} else if(i<sn && pj == s.charAt(i)) {

return mem[i][j] = isMatch(i+1, s, j+1, p);

}

return mem[i][j] = false;

}

* dp O(n^2)

bool isMatch(string s, string p) {

int sn = s.size(), pn = p.size();

vector<vector<bool>> dp(sn+1,vector<bool>(pn+1));

dp[sn][pn]=1;

for(int i=sn;i>=0;i--)

for(int j=pn-1;j>=0;j--)

if(p[j]=='\*') dp[i][j] = dp[i][j+1]||(i<sn && dp[i+1][j]);

else dp[i][j] = i<sn && (p[j]=='?'|| s[i]==p[j]) && dp[i+1][j+1];

return dp[0][0];

}

For most recursion to dp problems, we are done. However, we can still do better in this problem. For each star, we match it incrementally with 0, 1, 2 ... chars. If a path fails, we only need to backtrack from the last star. Backtracking from earlier stars eats more chars in s and leaves a shorter string for the last star. This does not create any more choices for the last star. More formally,

*Say we use #1 and have 2 stars in p separated by characters. When we reach the 2nd star for the first time, there is a match right before it between s(0...i) and p(0......j). s(0...i) is the first/shortest substring that matches p(0...j) because we match to chars incrementally. Matching the 2nd star starts from s[i+1]. If we backtrack the 1st star and match it with more characters then the next time when s(0...k) matche s p(0...j), k must be larger than i. At this point, matching the 2nd star starts from s[k+1]. Since k>i, so it is covered by just backtracking the 2nd star. Therefore backtracking the 1st star does not create more opportunities and we can ignore it.*

int lastStar;

public boolean isMatch(String s, String p) {

lastStar = -1;

return isMatch(0, s, 0, p);

}

private boolean isMatch(int i, String s, int j, String p) {

int sn = s.length(), pn = p.length();

if(j==pn) {

return i==sn;

}

char pj = p.charAt(j);

if(i<sn && pj == '?') {

return isMatch(i+1, s, j+1, p);

} else if(pj == '\*') {

lastStar = j;

return isMatch(i,s,j+1,p) || i<sn && j==lastStar && isMatch(i+1,s,j,p);

} else if(i<sn && pj == s.charAt(i)) {

return isMatch(i+1, s, j+1, p);

}

return false;

}

**Refer to**

<https://leetcode.com/problems/wildcard-matching/solutions/752350/recursion-brute-force-to-top-down-dp-and-bottom-up/>

We can solve this problem using recursion. The basic conditions that we need to put here is when j reaches the end of pattern length, then we need to check if the i has also reached the end or not, if not then it will return false, otherwise true.

If the i reaches the end of the string i.e. i=s.length(), then only when p[j]=' \* ' since \* can be equal to the empty sequence as well.

We will now check if the current charcter of pattern and string are equal or not. They would be equal if either s[i]==p[j] or p[j]='?'.

if the current pattern character is ' \* ' then we have two options either to move j forward and don't use it for matching or we can match and move the string index and keep the pattern index at j only.

if the current character is not ' \* ', then we need to check only if the first\_match is true and move both the i and j index by 1.

class Solution {

public:

bool isMatch(string s, string p) {

return helper(s,p,0,0);

}

bool helper(string s, string p, int i, int j)

{

if(j==p.length())

return i==s.length();

if(i==s.length())

return (p[j]=='\*' && helper(s,p,i,j+1));

bool first\_match=(i<s.length() && (p[j]==s[i] || p[j]=='?'));

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

{

return (helper(s,p,i+1,j) || helper(s,p,i,j+1));

}

else

{

return (first\_match && helper(s,p,i+1,j+1));

}

}

};

Top down DP solution :

We are solving the same subproblems many times instead we can save those problems and resuse them. We can initialize the dp array with -1 so that if it becomes postive then that means it has been solved for that i and j.

class Solution {

public:

bool isMatch(string s, string p) {

if(p.length()==0){

return (s.length()==0);

}

vector<vector<int>> v(s.length()+1,vector<int> (p.length()+1,-1));

return helper(s,p,0,0,v);

}

bool helper(string s, string p,int i,int j,vector<vector<int>> &v)

{

if(j==p.length())

return (i==s.length());

if(v[i][j]<0){

if(i==s.length())

v[i][j]= (p[j]=='\*' && helper(s,p,i,j+1,v));

else if(i<s.length() && (p[j]==s[i] || p[j]=='?'))

{

v[i][j]= helper(s,p,i+1,j+1,v);

}

else if(p[j]=='\*')

{

v[i][j]= (helper(s,p,i,j+1,v) || helper(s,p,i+1,j,v));

}

else

v[i][j]= false;

}

return v[i][j];

}

};

Bottom up solution

We can use bottom up approach to solve this problem.

dp[0][0]=true or 1. it is because if the length of the pattern and matching string is 0 then, they are equal or they are a match.

We can fill the first row of the dp. First row of DP tells us that the matching string length is zero, then uptill which column the pattern matches the empty string. So we know that it can only happen if the pattern character at that point is ' \* ' and if anything else comes other then a ' \* ' then we break.

Now, we can start filling the second row of dp, if the pattern is ' \* ' at j-1, then either we can use it to match in that case it would be equal to dp[i-1][j] and if we use the empty string for ' \* ' then it is equal to dp[i][j-1].

If the pattern at j-1 is not ' \* ' then we need check if the characters are equal or pattern character at j-1 is ' ? ' then dp[i][j] =dp[i-1][j-1];

class Solution {

public:

bool isMatch(string s, string p) {

if(p.length()==0)

return (s.length()==0);

vector<vector<int>> dp(s.length()+1,vector<int>(p.length()+1,0));

dp[0][0]=1;

for(int i=1;i<=p.length();i++)

{

if(p[i-1]=='\*')

dp[0][i]=1;

else

break;

}

for(int i=1;i<=s.length();i++)

{

for(int j=1;j<=p.length();j++)

{

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

{

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

}

else if(p[j-1]==s[i-1] || p[j-1]=='?')

{

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

}

}

}

return dp[s.length()][p.length()];

}

};