Wildcard Matching

Implement wildcard pattern matching with support for `'?'` and `'*'`.

```
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty sequence).

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

The function prototype should be:
bool isMatch(const char *s, const char *p)

Some examples:
isMatch("aa","a") → false
isMatch("aa","aa") → true
isMatch("aaa","aa") → false
isMatch("aa", "*") → true
isMatch("aa", "a*") → true
isMatch("ab", "?*") → true
isMatch("aab", "c*a*b") → false
```

Solution 1

I found this solution from http://yucoding.blogspot.com/2013/02/leetcode-question-123-wildcard-matching.html

---

The basic idea is to have one pointer for the string and one pointer for the pattern. This algorithm iterates at most length(string) + length(pattern) times, for each iteration, at least one pointer advance one step.

Here is Yu's elegant solution in C++

```
bool isMatch(const char *s, const char *p) {
        const char* star=NULL;
        const char* ss=s;
        while (*s){
            //advancing both pointers when (both characters match) or ('?' found
in pattern)
            //note that *p will not advance beyond its length
            if ((*p=='?')||(*p==*s)){s++;p++;continue;}

            // * found in pattern, track index of *, only advancing pattern point
er
            if (*p=='*'){star=p++; ss=s;continue;}

            //current characters didn't match, last pattern pointer was *, curren
t pattern pointer is not *
            //only advancing pattern pointer
            if (star){ p = star+1; s=++ss;continue;}

           //current pattern pointer is not star, last patter pointer was not *
           //characters do not match
            return false;
        }

       //check for remaining characters in pattern
        while (*p=='*'){p++;}

        return !*p;
    }
```

Here is my re-write in Java

```
boolean comparison(String str, String pattern) {
        int s = 0, p = 0, match = 0, starIdx = -1;
        while (s < str.length()){
            // advancing both pointers
            if (p < pattern.length()  && (pattern.charAt(p) == '?' || str.charAt(
s) == pattern.charAt(p))){
                s++;
                p++;
            }
            // * found, only advancing pattern pointer
            else if (p < pattern.length() && pattern.charAt(p) == '*'){
                starIdx = p;
                match = s;
                p++;
            }
           // last pattern pointer was *, advancing string pointer
            else if (starIdx != -1){
                p = starIdx + 1;
                match++;
                s = match;
            }
           //current pattern pointer is not star, last patter pointer was not *
          //characters do not match
            else return false;
        }

        //check for remaining characters in pattern
        while (p < pattern.length() && pattern.charAt(p) == '*')
            p++;

        return p == pattern.length();
}
```

written by pandora111 original link here

Solution 2

```
class Solution:
# @return a boolean
def isMatch(self, s, p):
    length = len(s)
    if len(p) - p.count('*') > length:
        return False
    dp = [True] + [False]*length
    for i in p:
        if i != '*':
            for n in reversed(range(length)):
                dp[n+1] = dp[n] and (i == s[n] or i == '?')
        else:
            for n in range(1, length+1):
                dp[n] = dp[n-1] or dp[n]
        dp[0] = dp[0] and i == '*'
    return dp[-1]
```

dp[n] means the substring s[:n] if match the pattern i

dp[0] means the empty string '' or s[:0] which only match the pattern '*'

use the reversed builtin because for every dp[n+1] we use the previous 'dp'

add Java O(m*n) version code

```
public boolean isMatch(String s, String p) {
    int count = 0;
    for (char c : p.toCharArray()) {
        if (c == '*')
            count++;
    }
    if (p.length() - count > s.length())
        return false;
    boolean[][] dp = new boolean[p.length() + 1][s.length() + 1];
    dp[0][0] = true;
    for (int j = 1; j <= p.length(); j++) {
        char pattern = p.charAt(j - 1);
        dp[j][0] = dp[j - 1][0] && pattern == '*';
        for (int i = 1; i <= s.length(); i++) {
            char letter = s.charAt(i - 1);
            if (pattern != '*') {
                dp[j][i] = dp[j - 1][i - 1] && (pattern == '?' || pattern == lett
er);
            } else
                dp[j][i] = dp[j][i - 1] || dp[j - 1][i];
        }
    }
    return dp[p.length()][s.length()];
}
```

written by tusizi original link here

Solution 3

Updated: Since the OJ has relaxed the time constraint, the following DP solution is now accepted without the trick :-)

Well, so many people has tried to solve this problem using DP. And almost all of them get TLE (if you see a **C++** DP solution that gets accepted, please let me know ^_^). Well, this post aims at providing an **accpted** DP solution which uses a **trick** to get around the largest test case, insteaed of a solution that is **fully correct**. So please do not give me down votes for that :-)

Let's briefly summarize the idea of DP. We define the state `P[i][j]` to be whether `s[0..i)` matches `p[0..j)`. The state equations are as follows:

1. `P[i][j] = P[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '?')`, if `p[j - 1] != '*'`;
2. `P[i][j] = P[i][j - 1] || P[i - 1][j]`, if `p[j - 1] == '*'`.

If you feel confused with the second equation, you may refer tothis link. There is an explanation in the comments.

We optimize the DP code to `O(m)` space by recording `P[i - 1][j - 1]` using a single variable `pre`.

The trick to avoid TLE is to hard-code the result for the largest test case by

```
if (n > 30000) return false;
```

The complete code is as follows.

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length(), n = p.length();
        if (n > 30000) return false; // the trick
        vector<bool> cur(m + 1, false);
        cur[0] = true;
        for (int j = 1; j <= n; j++) {
            bool pre = cur[0]; // use the value before update
            cur[0] = cur[0] && p[j - 1] == '*';
            for (int i = 1; i <= m; i++) {
                bool temp = cur[i]; // record the value before update
                if (p[j - 1] != '*')
                    cur[i] = pre && (s[i - 1] == p[j - 1] || p[j - 1] == '?');
                else cur[i] = cur[i - 1] || cur[i];
                pre = temp;
            }
        }
        return cur[m];
    }
};
```

For those interested in a fully correct solution, this link has a nice Greedy solution.

And I have rewritten the code below to fit the new C++ interface (changed from `char*` to `string`).

```
class Solution {
public:
    bool isMatch(string s, string p) {
        int m = s.length(), n = p.length();
        int i = 0, j = 0, asterisk = -1, match;
        while (i < m) {
            if (j < n && p[j] == '*') {
                match = i;
                asterisk = j++;
            }
            else if (j < n && (s[i] == p[j] || p[j] == '?')) {
                i++;
                j++;
            }
            else if (asterisk >= 0) {
                i = ++match;
                j = asterisk + 1;
            }
            else return false;
        }
        while (j < n && p[j] == '*') j++;
        return j == n;
    }
};
```

written by jianchao.li.fighter original link here