<https://leetcode.com/problems/regular-expression-matching/>

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '\*' where:

'.' Matches any single character.

'\*' Matches zero or more of the preceding element.

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 = "a\*"

Output: true

Explanation: '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

**Example 3:**

Input: s = "ab", p = ".\*"

Output: true

Explanation: ".\*" means "zero or more (\*) of any character (.)".

**Constraints:**

1 <= s.length <= 20

1 <= p.length <= 20

s contains only lowercase English letters.

p contains only lowercase English letters, '.', and '\*'.

It is guaranteed for each appearance of the character '\*', there will be a previous valid character to match.

**Attempt 1: 2023-10-17**

**Solution 1:  Native DFS (360 min)**

**Wrong Solution**

**The problem happens on terminate condition:**

**Since \* in p can match 0 of previous char, so empty string(i == s.length()) may match p**

**简单说p是pattern串，由于\*的存在，完全可以匹配多个s中的字符或者0个字符，当s结束了即空串的时候依然可以对应到p中的\*表示0个字符的情况，换句话说，s结束p并不一定结束，s结束并且p结束是s和p等价的充分不必要条件，p结束并且s结束是s和p等价的必要条件**

class Solution {

public boolean isMatch(String s, String p) {

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

}

// '.' Matches any single character.​​​​

// '\*' Matches zero or more of the preceding element.

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

// Wrong condition:

// Test out by

// Input: s = "a", p = "ab\*"

// Output: false Expected: true

// Since \* in p can match 0 of previous char, so empty

// string(i == s.length()) may match p

// 简单说p是pattern串，由于\*的存在，完全可以匹配多个s中的字符

// 或者0个字符，当s结束了即空串的时候依然可以对应到p中的\*表示0

// 个字符的情况，换句话说，s结束p并不一定结束，s结束并且p结束是

// s和p等价的充分不必要条件，p结束并且s结束是s和p等价的必要条件，

// p结束但s没结束则这两个字符串不可能等价

if(i == s.length() && j == p.length()) {

return true;

}

if(i == s.length() || j == p.length()) {

return false;

}

if(j + 1 < p.length() && p.charAt(j + 1) == '\*') {

if(helper(s, i, p, j + 2)) {

return true;

}

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

if(helper(s, ++i, p, j + 2)) {

return true;

}

}

} else {

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

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

}

}

return false;

}

}

**Correct solution:**

class Solution {

public boolean isMatch(String s, String p) {

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

}

// '.' Matches any single character.​​​​

// '\*' Matches zero or more of the preceding element.

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

// Since \* in p can match 0 of previous char, so empty

// string(i == s.length()) may match p

// 简单说p是pattern串，由于\*的存在，完全可以匹配多个s中的字符

// 或者0个字符，当s结束了即空串的时候依然可以对应到p中的\*表示0

// 个字符的情况，换句话说，s结束p并不一定结束，s结束并且p结束是

// s和p等价的充分不必要条件，p结束并且s结束是s和p等价的必要条件，

// p结束但s没结束则这两个字符串不可能等价

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

return i == s.length();

}

if(j + 1 < p.length() && p.charAt(j + 1) == '\*') {

// i-1 i i+1 i+2

// s: <-- b

// j-1 j j+1 j+2

// p: <-- a \* b

// 第一种情况：\*星号的作用是让前面的字符出现0次，意味着位置在j+1的\*配上

// 位置在j的a代表a出现了0次，所以[a\*]实际相当于空字符串，直接从p[0,j+2]

// 中可以剥离出来并且等价于p[0,j-1]和p[j+2]组合成一个字符串，而同时我们

// 需要的就是将去掉[a\*]之后的p字符串p[0,j+2]与s[0,i]作比较

if(helper(s, i, p, j + 2)) {

return true;

}

// i-1 i i+1 i+2 i-1 i i+1 i+2 i+3 i-1 i i+1 i+...

// s: <-- a b s: <-- a a a b s: <-- a ... b

// j-1 j j+1 j+2 or j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a \* b p: <-- a \* b p: <-- . \* b

// (1) \* = 'a' X 1 (2) \* = 'a' X (> 1) (3) \* = '.' X (>=1)

// 第二种情况：\*星号的作用是让前面的字符出现1 ~ n次，意味着位置在j+1的\*

// 配上位置在j的a代表a出现了1 ~ n次:

// (1) 当[a\*]代表1个a时，满足p[j,j+1](='a')等于a[i](='a')，只需要继续

// 比较p[j+2,plen)和s[i+1,slen)

// (2) 当[a\*]代表n个a时，满足p[j,j+1](='aa...a')，此时如果s[i]到s[i+n]

// 都为'a'时，即可一直满足等于[a\*]，可以采用while循环的方式寻找s中从位置i

// 开始后的所有连续n个'a'，直到第一个不为'a'的字符出现在位置i+n+1为止，

// 只需要继续比较p[j+2,plen]和s[i+n+1,slen]

// 本质上(1)是(2)的一种特殊情况，所以逻辑可以合并成一种

// (3) 当p的位置j为'.'时，p[j,j+1]构成'.\*'可以代表任意字符重复任意次，可

// 以采用while循环的方式尝试匹配每一种可能，具体来说一种可能是以s中位置i

// 之后的某一个字符等于p[j+2]来开启的，毕竟要满足两个字符串相等，第一字符

// 必须相等，比如s[i+3]=p[j+2]='b'和s[i+5]=p[j+2]='b'代表了两种可能，

// 如果选择s[i+3]=p[j+2]='b'作为入口，其中s[i,i+2]都可以被p[j,j+1]='.\*'

// 覆盖，如果选择s[i+5]=p[j+2]='b'作为入口，其中s[i,i+4]都可以被p[j,j+1]

// ='.\*'覆盖，这样话，就只需要比较s[i+3,slen)和p[j+2,plen)或者s[i+5,slen)

// 和p[j+2,plen)是否相等

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

// The 'i = i + 1' must happen before recursion, because we suppose

// to compare s[i + 1] with p[j + 2] only, not s[i] with p[j + 2],

// since we the condition in while loop already compared s[i] and p[j]

i = i + 1;

if(helper(s, i, p, j + 2)) {

return true;

}

}

} else {

// 若p的下一个字符不为'\*'，若此时s为空返回 false，否则判断当前字符是否匹配

// (或者p的当前字符为'.')，且从各自的下一个字符开始调用递归函数匹配

// i-1 i i+1 i+2 i-1 i i+1 i+2

// s: <-- a b s: <-- a b

// j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a b p: <-- . b

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 is roughly O(m + n choose n).

Say n is length of s, m is length of p. In the worst case,

T(n,m) = T(n,m-2)+T(n-1,m-2)+T(n-2,m-2)+...+T(1, m-2)

T(n-1,m) = T(n-1,m-2)+T(n-2,m-2)+...+T(1,m-2)

=> T(n,m) = T(n-1,m) + T(n,m-2)

Initial conditions are T(0,m) = m, T(n,0) = 1

**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 zero or more of the preceding element.

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

// Since \* in p can match 0 of previous char, so empty

// string(i == s.length()) may match p

// 简单说p是pattern串，由于\*的存在，完全可以匹配多个s中的字符

// 或者0个字符，当s结束了即空串的时候依然可以对应到p中的\*表示0

// 个字符的情况，换句话说，s结束p并不一定结束，s结束并且p结束是

// s和p等价的充分不必要条件，p结束并且s结束是s和p等价的必要条件

// p结束但s没结束则这两个字符串不可能等价

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

return i == s.length();

}

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

return memo[i][j];

}

if(j + 1 < p.length() && p.charAt(j + 1) == '\*') {

// i-1 i i+1 i+2

// s: <-- b

// j-1 j j+1 j+2

// p: <-- a \* b

// 第一种情况：\*星号的作用是让前面的字符出现0次，意味着位置在j+1的\*配上

// 位置在j的a代表a出现了0次，所以[a\*]实际相当于空字符串，直接从p[0,j+2]

// 中可以剥离出来并且等价于p[0,j-1]和p[j+2]组合成一个字符串，而同时我们

// 需要的就是将去掉[a\*]之后的p字符串p[0,j+2]与s[0,i]作比较

if(helper(s, i, p, j + 2, memo)) {

return memo[i][j] = true;

}

// i-1 i i+1 i+2 i-1 i i+1 i+2 i+3 i-1 i i+1 i+...

// s: <-- a b s: <-- a a a b s: <-- a ... b

// j-1 j j+1 j+2 or j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a \* b p: <-- a \* b p: <-- . \* b

// (1) \* = 'a' X 1 (2) \* = 'a' X (> 1) (3) \* = '.' X (>=1)

// 第二种情况：\*星号的作用是让前面的字符出现1 ~ n次，意味着位置在j+1的\*

// 配上位置在j的a代表a出现了1 ~ n次:

// (1) 当[a\*]代表1个a时，满足p[j,j+1](='a')等于a[i](='a')，只需要继续

// 比较p[j+2,plen)和s[i+1,slen)

// (2) 当[a\*]代表n个a时，满足p[j,j+1](='aa...a')，此时如果s[i]到s[i+n]

// 都为'a'时，即可一直满足等于[a\*]，可以采用while循环的方式寻找s中从位置i

// 开始后的所有连续n个'a'，直到第一个不为'a'的字符出现在位置i+n+1为止，

// 只需要继续比较p[j+2,plen]和s[i+n+1,slen]

// 本质上(1)是(2)的一种特殊情况，所以逻辑可以合并成一种

// (3) 当p的位置j为'.'时，p[j,j+1]构成'.\*'可以代表任意字符重复任意次，可

// 以采用while循环的方式尝试匹配每一种可能，具体来说一种可能是以s中位置i

// 之后的某一个字符等于p[j+2]来开启的，毕竟要满足两个字符串相等，第一字符

// 必须相等，比如s[i+3]=p[j+2]='b'和s[i+5]=p[j+2]='b'代表了两种可能，

// 如果选择s[i+3]=p[j+2]='b'作为入口，其中s[i,i+2]都可以被p[j,j+1]='.\*'

// 覆盖，如果选择s[i+5]=p[j+2]='b'作为入口，其中s[i,i+4]都可以被p[j,j+1]

// ='.\*'覆盖，这样话，就只需要比较s[i+3,slen)和p[j+2,plen)或者s[i+5,slen)

// 和p[j+2,plen)是否相等

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

// The 'i = i + 1' must happen before recursion, because we suppose

// to compare s[i + 1] with p[j + 2] only, not s[i] with p[j + 2],

// since we the condition in while loop already compared s[i] and p[j]

i = i + 1;

if(helper(s, i, p, j + 2, memo)) {

return memo[i][j] = true;

}

}

} else {

// 若p的下一个字符不为'\*'，若此时s为空返回 false，否则判断当前字符是否匹配

// (或者p的当前字符为'.')，且从各自的下一个字符开始调用递归函数匹配

// i-1 i i+1 i+2 i-1 i i+1 i+2

// s: <-- a b s: <-- a b

// j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a b p: <-- . b

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;

}

}

**Solution 3: DP (120 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];

// In Native DFS top = [0][0] bottom = [...][pLen] and [sLen][...]

// not directly mention bottom = [sLen][pLen] because in DFS

// recursion the terminate condition NOT at [sLen][pLen] only but

// [...][pLen]

// (1) When pLen reached, sLen reached, terminated, [=sLen][=pLen] = true

// (2) When pLen reached, sLen not reached, terminated, [<sLen][=pLen] = false

// (3) When pLen not reached, sLen reached, it may terminated with true

// only when p ending with 'char + \*' or '.\*' because '\*' in p can

// represent preceding char at 0 times

// p结束且s结束则这两个字符串等价

// p结束但s没结束则这两个字符串不可能等价

// p没结束但s结束则这两个字符串可能等价(此时p可能剩余某个字符配上\*或者.\*)

dp[sLen][pLen] = true;

// For condition (2) default as false no need initialize

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(j + 1 < pLen && p.charAt(j + 1) == '\*') {

// i-1 i i+1 i+2

// s: <-- b

// j-1 j j+1 j+2

// p: <-- a \* b

// 第一种情况：\*星号的作用是让前面的字符出现0次，意味着位置在j+1的\*配上

// 位置在j的a代表a出现了0次，所以[a\*]实际相当于空字符串，直接从p[0,j+2]

// 中可以剥离出来并且等价于p[0,j-1]和p[j+2]组合成一个字符串，而同时我们

// 需要的就是将去掉[a\*]之后的p字符串p[0,j+2]与s[0,i]作比较

dp[i][j] = dp[i][j + 2];

// i-1 i i+1 i+2 i-1 i i+1 i+2 i+3 i-1 i i+1 i+...

// s: <-- a b s: <-- a a a b s: <-- a ... b

// j-1 j j+1 j+2 or j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a \* b p: <-- a \* b p: <-- . \* b

// (1) \* = 'a' X 1 (2) \* = 'a' X (> 1) (3) \* = '.' X (>=1)

// 第二种情况：\*星号的作用是让前面的字符出现1 ~ n次，意味着位置在j+1的\*

// 配上位置在j的a代表a出现了1 ~ n次:

// (1) 当[a\*]代表1个a时，满足p[j,j+1](='a')等于a[i](='a')，只需要继续

// 比较p[j+2,plen)和s[i+1,slen)

// (2) 当[a\*]代表n个a时，满足p[j,j+1](='aa...a')，此时如果s[i]到s[i+n]

// 都为'a'时，即可一直满足等于[a\*]，可以采用while循环的方式寻找s中从位置i

// 开始后的所有连续n个'a'，直到第一个不为'a'的字符出现在位置i+n+1为止，

// 只需要继续比较p[j+2,plen]和s[i+n+1,slen]

// 本质上(1)是(2)的一种特殊情况，所以逻辑可以合并成一种

// (3) 当p的位置j为'.'时，p[j,j+1]构成'.\*'可以代表任意字符重复任意次，可

// 以采用while循环的方式尝试匹配每一种可能，具体来说一种可能是以s中位置i

// 之后的某一个字符等于p[j+2]来开启的，毕竟要满足两个字符串相等，第一字符

// 必须相等，比如s[i+3]=p[j+2]='b'和s[i+5]=p[j+2]='b'代表了两种可能，

// 如果选择s[i+3]=p[j+2]='b'作为入口，其中s[i,i+2]都可以被p[j,j+1]='.\*'

// 覆盖，如果选择s[i+5]=p[j+2]='b'作为入口，其中s[i,i+4]都可以被p[j,j+1]

// ='.\*'覆盖，这样话，就只需要比较s[i+3,slen)和p[j+2,plen)或者s[i+5,slen)

// 和p[j+2,plen)是否相等

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

// 如果和Native DFS的写法比较，其实下面的公式其实基于：

// dp[i][j] = (dp[i][j + 2] || dp[i + 1][j])

// 但因为之前已经在第一种情况中有了dp[i][j] = dp[i][j + 2]，所以合并

// 得到：dp[i][j] = (dp[i][j] || dp[i + 1][j])，也就是下面的最终写法

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

}

} else {

// 若p的下一个字符不为'\*'，若此时s为空返回 false，否则判断当前

// 字符是否匹配(或者p的当前字符为'.')，如果匹配则dp当前状态

// [i][j]取决于上一个状态[i + 1][j + 1]

// i-1 i i+1 i+2 i-1 i i+1 i+2

// s: <-- a b s: <-- a b

// j-1 j j+1 j+2 or j-1 j j+1 j+2

// p: <-- a b p: <-- . b

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), M is s length, N is p length

Space Complexity: O(M\*N)

**Refer to**

<https://leetcode.com/problems/regular-expression-matching/solutions/5847/evolve-from-brute-force-to-dp/>

The point I want to show is once you figure out the naive approach (#1 or #2), it is effortless to transform to dp. Mostly copy and replace.

Recursion. This is the most straight forward idea. We match the current char in s and p, then solve the problem recursively. Following are the keys

1) Termination condition. Since p can match empty string, s is empty (i=sn) cannot terminate the matching. Only p is empty(j==pn) terminates the recursion.

2) If a char is followed by \*, then we only need to honor the \*. No need to do 1 to 1 match. 1 to 1 match is only needed if a char is not followed by \*.

3) When matching \*, we check if the next char is \*. If we process \* as the current char, it means 1 to 1 match is already done for the previous char. The logic is not clean.

Java

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) { // since \* in p can match 0 of previous char, so empty string(i==sn) may match p

return i==sn;

}

char pj = p.charAt(j);

if(j+1<pn && p.charAt(j+1)=='\*') { //match \*, needs to look at the next char to repeate current char

if(isMatch(i,s,j+2,p)) {

return true;

}

while(i<sn && (pj == '.'||pj==s.charAt(i))) {

if(isMatch(++i,s,j+2,p)) {

return true;

}

}

} else if(i<sn && (s.charAt(i) == pj || //match char

pj=='.')) { //match dot

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

}

return false;

}

Time complexity is roughly O(m+n choose n).

Say n is length of s, m is length of p. In the worst case,

T(n,m) = T(n,m-2)+T(n-1,m-2)+T(n-2,m-2)+...+T(1, m-2)

T(n-1,m) = T(n-1,m-2)+T(n-2,m-2)+...+T(1,m-2)

=> **T(n,m) = T(n-1,m) + T(n,m-2)**

Initial conditions are T(0,m) = m, T(n,0) = 1

To solve the 2 variable recursion, see [here](http://math.stackexchange.com/questions/206158/solving-recurrence-relation-in-2-variables)

Recursion The highlighted recursive equation indicates a more concise way to handle the case when \* matches 1 or more characters. It does not change the time complexity of the recursion but it transforms to a better run time dp solution compared with the original equation.

Java

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) { // since \* in p can match 0 of previous char, so empty string(i==sn) may match p

return i==sn;

}

char pj = p.charAt(j);

if(j+1<pn && p.charAt(j+1)=='\*') { //match \*, needs to look at the next char to repeate current char

if(isMatch(i,s,j+2,p)) {

return true;

}

if(i<sn && (pj == '.'||pj==s.charAt(i))) {

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

return true;

}

}

} else if(i<sn && (s.charAt(i) == pj || //match char

pj=='.')) { //match dot

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

}

return false;

}

Recursion with memoization O(mn), add a vector, the rest is the same as #2.

Java

Boolean[][] mem;

public boolean isMatch(String s, String p) {

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

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) { // since \* in p can match 0 of previous char, so empty string(i==sn) may match p

return i==sn;

}

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

return mem[i][j];

}

char pj = p.charAt(j);

if(j+1<pn && p.charAt(j+1)=='\*') { //match \*, needs to look at the next char to repeate current char

if(isMatch(i,s,j+2,p)) {

return mem[i][j]=true;

}

if(i<sn && (pj == '.'||pj==s.charAt(i))) {

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

return mem[i][j]=true;

}

}

} else if(i<sn && (s.charAt(i) == pj || //match char

pj=='.')) { //match dot

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

}

return mem[i][j]=false;

}

Bottom up dp to be consistent with #3, O(mn), use for loop instead of recursion, the rest is the same as #2 and #3.

Idea is the same as top down dp.

Java

public boolean isMatch(String s, String p) {

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

boolean[][] dp=new boolean[sn+1][pn+1];

dp[sn][pn]=true;

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

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

if(j+1<pn&&p.charAt(j+1)=='\*') {

dp[i][j]=dp[i][j+2];

if(i<sn&&(p.charAt(j)=='.'||s.charAt(i)==p.charAt(j)))

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

} else if(i<sn&&(p.charAt(j)=='.'||s.charAt(i)==p.charAt(j)))

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

return dp[0][0];

}

**Refer to**

<https://grandyang.com/leetcode/10/>

这道求正则表达式匹配的题和那道 [Wildcard Matching](http://www.cnblogs.com/grandyang/p/4401196.html) 的题很类似，不同点在于*的意义不同，在之前那道题中，*表示可以代替任意个数的字符，而这道题中的*表示之前那个字符可以有0个，1个或是多个，就是说，字符串 a*b，可以表示b或是 aaab，即a的个数任意，这道题的难度要相对之前那一道大一些，分的情况的要复杂一些，需要用递归 Recursion 来解，大概思路如下：

- 若p为空，若s也为空，返回 true，反之返回 false。

- 若p的长度为1，若s长度也为1，且相同或是p为 ‘.’ 则返回 true，反之返回 false。

- 若p的第二个字符不为\*，若此时s为空返回 false，否则判断首字符是否匹配，且从各自的第二个字符开始调用递归函数匹配。

- 若p的第二个字符为\*，进行下列循环，条件是若s不为空且首字符匹配（包括 p[0] 为点），调用递归函数匹配s和去掉前两个字符的p（这样做的原因是假设此时的星号的作用是让前面的字符出现0次，验证是否匹配），若匹配返回 true，否则s去掉首字母（因为此时首字母匹配了，我们可以去掉s的首字母，而p由于星号的作用，可以有任意个首字母，所以不需要去掉），继续进行循环。

- 返回调用递归函数匹配s和去掉前两个字符的p的结果（这么做的原因是处理星号无法匹配的内容，比如 s=”ab”, p=”a*b”，直接进入 while 循环后，我们发现 “ab” 和 “b” 不匹配，所以s变成 “b”，那么此时跳出循环后，就到最后的 return 来比较 “b” 和 “b” 了，返回 true。再举个例子，比如 s=””, p=”a*“，由于s为空，不会进入任何的 if 和 while，只能到最后的 return 来比较了，返回 true，正确）。

解法一：

class Solution {

public:

bool isMatch(string s, string p) {

if (p.empty()) return s.empty();

if (p.size() == 1) {

return (s.size() == 1 && (s[0] == p[0] || p[0] == '.'));

}

if (p[1] != '\*') {

if (s.empty()) return false;

return (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));

}

while (!s.empty() && (s[0] == p[0] || p[0] == '.')) {

if (isMatch(s, p.substr(2))) return true;

s = s.substr(1);

}

return isMatch(s, p.substr(2));

}

};

上面的方法可以写的更加简洁一些，但是整个思路还是一样的，先来判断p是否为空，若为空则根据s的为空的情况返回结果。当p的第二个字符为*号时，由于*号前面的字符的个数可以任意，可以为0，那么我们先用递归来调用为0的情况，就是直接把这两个字符去掉再比较，或者当s不为空，且第一个字符和p的第一个字符相同时，再对去掉首字符的s和p调用递归，注意p不能去掉首字符，因为*号前面的字符可以有无限个；如果第二个字符不为*号，那么就老老实实的比较第一个字符，然后对后面的字符串调用递归，参见代码如下：

解法二：

class Solution {

public:

bool isMatch(string s, string p) {

if (p.empty()) return s.empty();

if (p.size() > 1 && p[1] == '\*') {

return isMatch(s, p.substr(2)) || (!s.empty() && (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p));

} else {

return !s.empty() && (s[0] == p[0] || p[0] == '.') && isMatch(s.substr(1), p.substr(1));

}

}

};

我们也可以用 DP 来解，定义一个二维的 DP 数组，其中 dp[i][j] 表示 s[0,i) 和 p[0,j) 是否 match，然后有下面三种情况(下面部分摘自[这个帖子](https://discuss.leetcode.com/topic/17852/9-lines-16ms-c-dp-solutions-with-explanations))：

1.  P[i][j] = P[i - 1][j - 1], if p[j - 1] != ‘*‘ && (s[i - 1] == p[j - 1] || p[j - 1] == ‘.’);2.  P[i][j] = P[i][j - 2], if p[j - 1] == ‘*‘ and the pattern repeats for 0 times;3.  P[i][j] = P[i - 1][j] && (s[i - 1] == p[j - 2] || p[j - 2] == ‘.’), if p[j - 1] == ‘\*’ and the pattern repeats for at least 1 times.

解法三：

class Solution {

public:

bool isMatch(string s, string p) {

int m = s.size(), n = p.size();

vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));

dp[0][0] = true;

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

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

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

dp[i][j] = dp[i][j - 2] || (i > 0 && (s[i - 1] == p[j - 2] || p[j - 2] == '.') && dp[i - 1][j]);

} else {

dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s[i - 1] == p[j - 1] || p[j - 1] == '.');

}

}

}

return dp[m][n];

}

};