# LeetCode

## 10 Regular expression matching [Hard]

**题意**：源串s : “aaa”，模式串p : ”.\*”，能用p表示s否? 全匹配

**题解思路**：采用动态规划求解

令dp[i][j]代表s : 0 ~ i-1和p : 0 ~ j-1的匹配结果，即不包含s[i]和p[j]，这样原问题就转换为求解dp[sLen][pLen]。具体情况如下：

* 当p[j-1] != ’\*’时，那么p[j-1]必须和s[i-1]匹配，那就将匹配过程分成了两部分，即s : 0 ~ i-2 和p : 0 ~ j-2的匹配结果【即dp[i-1][j-1]】与p[j-1]和s[i-1]的匹配结果【p[j-1] == s[i-1] || p[j-1] == ‘.’】，所以dp[i][j] = dp[i-1][j-1] && (p[j-1] == s[i-1] || p[j-1] == ‘.’)
* 当p[j-1] == ‘\*’时,情况复杂一些，如下：

1. ‘\*’代表p[j-2]出现了0次，即s:0~i-1和p:0~j-3匹配【即dp[i][j-2]】
2. ‘\*’代表p[j-2]出现了1次或1次以上，那么s[i-1]一定要与p[j-1]匹配【p[j-1] == s[i-1] || p[j-1] == ‘.’】，因为p[j-1]至少出现了一次，例如：s:”aaaa”和p:”a\*”，’a’和’.’一定是匹配的。还有，s:0~i-2一定要与p:0~j-1匹配【即dp[i-1][j]】，可以用反证法，如果s:0~i-2不与p:0~ j-1匹配，那加上s[j-1]也肯定不匹配。正向思维就是只有在s:0~i-2和p:0~j-1匹配了，那我们再加上满足条件的s[i-1]也肯定匹配，因为p[j-1]为’\*’。
3. 这样，当p[j-1] == ‘\*’时，dp[i][j] = dp[i][j-2] || dp[i-1][j] && (p[j-1] == s[i-1] || p[j-1] == ‘.’)

**注意事项**：

* 审题，题目要求是全匹配
* 初始化dp[i][0]，不需要，除了dp[0][0]外，其余dp[i][0]=false默认
* 初始化dp[0][j]，当然dp[0][0] = true，当j>0时，dp[0][2K+1] = false,因为奇数位置是不可能匹配s为空字符串的，即s:””和p:”a”肯定是不能匹配的。出现匹配的情况只有是p:”a\*a\*a\*”，其中’a’可以换为任何字符。程序实现:

|  |
| --- |
| //初始化dp[0][2],dp[0][4],...,dp[0][2k],...  for(int i=2; i<=pLen; i+=2) {  if(!(p.charAt(i-1) == '\*' && dp[0][i-2])) {  break;  }else {  dp[0][i] = true;  }  } |

代码实现

|  |
| --- |
| **public** **class** Solution {  **public** **boolean** isMatch(String s, String p) {  **if**(s == **null** || p == **null**) **return** **false**;  **int** sLen = s.length();  **int** pLen = p.length();  **boolean**[][] dp = **new** **boolean**[sLen+1][pLen+1];  dp[0][0] = **true**;  //初始化dp[0][2],dp[0][4],...,dp[0][2k],...  **for**(**int** i=2; i<=pLen; i+=2) {  **if**(!(p.charAt(i-1) == '\*' && dp[0][i-2])) {  **break**;  }**else** {  dp[0][i] = **true**;  }  }  **for**(**int** i=1; i<=sLen; i++) {  **for**(**int** j=1; j<=pLen; j++) {  //dp[i][j] 代表s:0~i-1,p:0~j-1,而当j=1时,j-1=0,p不可能是以'\*'开头  **if**(p.charAt(j-1) == '\*') {  //dp[i][j] = dp[i][j-2] || dp[i][j-1] || dp[i-1][j-1] && (s.charAt(i-1) == p.charAt(j-2) || p.charAt(j-2) == '.');  dp[i][j] = dp[i][j-2] || i > 0 && dp[i-1][j] && (p.charAt(j-2) == s.charAt(i-1) || p.charAt(j-2) == '.');  }**else** {  dp[i][j] = dp[i-1][j-1] && (p.charAt(j-1) == s.charAt(i-1) || p.charAt(j-1) == '.');  }  }  }    **return** dp[sLen][pLen];  }  } |

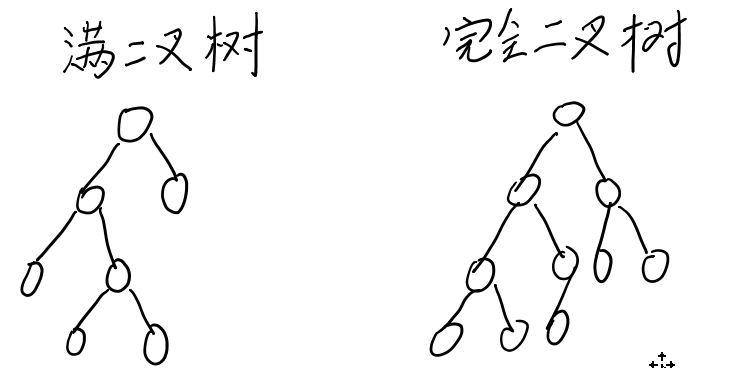
## 222 Count Complete Tree Nodes [Medium]

**题意**：计算一棵完全二叉树的结点数量

**题外话**：完全二叉树 vs 满二叉树

完全二叉树：只有最下面的两层结点度能够小于2，并且最下面一层的结点都集中在该层最左边的若干位置的二叉树。

满二叉树：除叶节点外，其他每个结点都有两个子节点。



**题解思路**：利用完全二叉树特点求解

首先，通过O(n)遍历所有结点计算结点数量，肯定是超时的，因为这种方法使用于所有二叉树，而没有利用完全二叉树的特点。

分析完全二叉树的特点有：完全二叉树的左右子树必有一个完全满二叉树，如上面的右子树，因为是一个完全满二叉树，所以结点数量num = 1^height – 1，加上根结点，num = 1 ^ height，这样，我们再计算左子树的结点数量，以此类推。

那么，怎么判断到底是左子树还是右子树是完全满二叉树呢？还有怎么计算树的高度呢？我们只需要对根结点的左右子结点循环访问他们的左子节点，每访问一次，树的高度加1，不妨令左子树高度为lh，右子树高度为rh，这样如果lh == rh，那证明左子树是完全满二叉树；如果lh > rh，那证明右子树是完全满二叉树；

**注意事项**：利用移位操作计算1^height，即1 << height。不要用Math.pow(2, height)会超时。

**代码实现**：

|  |
| --- |
| /\*\*  \* Definition for a binary tree node.  \* public class TreeNode {  \* int val;  \* TreeNode left;  \* TreeNode right;  \* TreeNode(int x) { val = x; }  \* }  \*/  public class Solution2 {  public int countNodes(TreeNode root) {  if(root == null) return 0;  int lh = getHeight(root.left);  int rh = getHeight(root.right);  if(lh == rh)  //直接用Math.pow会超时，移位操作最好  //return (int)Math.pow(2, lh) + countNodes(root.right);  //左子树为完全满二叉树  return (1 << lh) + countNodes(root.right);  //右子树为完全满二叉树  return (1 << rh) + countNodes(root.left);  }  private int getHeight(TreeNode root) {  int h = 0;  while(root != null) {  root = root.left;  h++;  }  return h;  }  } |