一 动态规划

1. 股票问题

[123. 买卖股票的最佳时机 III]

2 377. 组合总和IV 特殊的0-1背包问题

不同顺序算一个

个排列问题怎么解决。比如nums=[1,2,3] target = 8，要计算能构成8的所有排列有几个，那就可以计算构成5的排列有几个，但是到这我想的是，这个3应该有几种方法插入到构成5的排列里呢？就卡在这了。然后看了别人的代码，一时又难以理解，想了半天想明白了。

在计算构成8的排列有几个时即dp[8]，我们只关注分别以1,2,3为"屁股"的所有排列的个数，这仨排列个数之和就是构成8的所有排列了。为什么这么说呢？因为所有排列一定一定是以1,2,3其中一个为结尾的（废话），那么对于8而言，以3为结尾的排列个数就是dp[5]，以2为结尾的排列个数就是dp[6]，同理以1为结尾的排列个数就是dp[7]，把它仨加起来，就是dp[8]了。

这道题要注意一下初始化和overflow问题，初始化是应该dp[0]=1，详见代码注释。

零钱兑换问题中dp[i]为dp[i-nums[j]]的最小值，而这道题中dp[i]为dp[i-nums[j]]之和，这是它们的区别。

3 完全背包问题

518与377

```
class Solution {
    public int waysToChange(int n) {

        int coins[] = {1,5,10,25};
        int dp [] = new int[n+1];
        Arrays.fill(dp,0);
        //为0时 自动那个以为1
        dp[0] =1;
           for (int j =0;j<coins.length;j++){

               for (int i =coins[j]; i<=n;i++){
               if (i-coins[j]>=0){
                   dp[i] = (dp[i-coins[j]] +dp[i])% 1000000007;
               }
           }
       }
        return dp[n];
    }
}
```

4 0-1背包问题

416.:分割等和子集
474.:一和零
494

5 354. 俄罗斯套娃信封问题

给定一些标记了宽度和高度的信封，宽度和高度以整数对形式 (w, h) 出现。当另一个信封的宽度和高度都比这个信封大的时候，这个信封就可以放进另一个信封里，如同俄罗斯套娃一样。

请计算最多能有多少个信封能组成一组“俄罗斯套娃”信封（即可以把一个信封放到另一个信封里面）。

说明:
不允许旋转信封。

示例:

输入: envelopes = [[5,4],[6,4],[6,7],[2,3]]
输出: 3
解释: 最多信封的个数为 3, 组合为: [2,3] => [5,4] => [6,7]。

先排序，最长上升子序列问题

```
class Solution {
    public int maxEnvelopes(int[][] envelopes) {

            Arrays.sort(envelopes,(int a[], int b[])->{
                return a[0] == b[0] ? b[1]-a[1] : a[0] -b[0];
            });

      int n = envelopes.length;
      int val[] = new int [n];

      for(int i =0;i<n;i++){
           val[i] = envelopes[i][1];
      }
       return lengthOflsc(val);
    }

    private int lengthOflsc(int [] nums){

        if(nums.length == 0) return 0;
        int[] dp = new int[nums.length];
        int res = 1;
        Arrays.fill(dp, 1);
        for(int i = 1; i < nums.length; i++) {
            for(int j = 0; j < i; j++) {
                if(nums[j] < nums[i]) dp[i] = Math.max(dp[i], dp[j] + 1);
            }
            res = Math.max(res, dp[i]);
        }
        return res;
    }
}
```

6 1312 让字符串成为回文串的最少插入次数

给你一个字符串 s ， 每一次操作你都可以在字符串的任意位置插入任意字符。

请你返回让 s 成为回文串的 最少操作次数 。

示例 1：

输入：s = "zzazz"
输出：0
解释：字符串 "zzazz" 已经是回文串了，所以不需要做任何插入操作。
示例 2：

输入：s = "mbadm"
输出：2
解释：字符串可变为 "mbdadbm" 或者 "mdbabdm" 。
示例 3：

输入：s = "leetcode"
输出：5
解释：插入 5 个字符后字符串变为 "leetcodocteel" 。
示例 4：

输入：s = "g"
输出：0
示例 5：

输入：s = "no"
输出：1

与leetcode516相识,用总长度减去回文序列

```
class Solution {
    public int minInsertions(String s) {
      int n = s.length();
      int dp[][] = new int [n][n];
      for(int i=n-1;i>=0;i--){
          dp[i][i] =1;
         for(int j =i+1;j<n;j++){
           if(s.charAt(i) == s.charAt(j)){
           dp[i][j] = dp[i+1][j-1] +2;

           } else{
               dp[i][j] = Math.max(dp[i+1][j],dp[i][j-1]);
           }
         }
      }
      return n -dp[0][n-1];
    }
}
```

7 516最长回文子序列

给定一个字符串 s，找到其中最长的回文子序列，并返回该序列的长度。可以假设 s 的最大长度为 1000 。

示例 1:
输入:

"bbbab"
输出:

4
一个可能的最长回文子序列为 "bbbb"。

示例 2:
输入:

"cbbd"
输出:

2
一个可能的最长回文子序列为 "bb"。

```
class Solution {
    public int longestPalindromeSubseq(String s) {

      int n = s.length();
      int f[][] = new int [n][n];

        for (int i = n - 1; i >= 0; i--) {
            f[i][i] = 1;
            for (int j = i + 1; j < n; j++) {
                if (s.charAt(i) == s.charAt(j)) {
                    f[i][j] = f[i + 1][j - 1] + 2;
                } else {
                    f[i][j] = Math.max(f[i + 1][j], f[i][j - 1]);
                }
            }
        }
        return f[0][n - 1];
    }
}
```

8 1143. 最长公共子序列

9 718. 最长重复子数组 最大子串问题

11 300. 最长上升子序列

难度中等847收藏分享切换为英文关注反馈

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

dp[i] 代表 已i结尾的子序列

12 312. 戳气球

有 n 个气球，编号为0 到 n-1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。如果你戳破气球 i ，就可以获得 nums[left] * nums[i] * nums[right] 个硬币。 这里的 left 和 right 代表和 i 相邻的两个气球的序号。注意当你戳破了气球 i 后，气球 left 和气球 right 就变成了相邻的气球。

求所能获得硬币的最大数量。

说明:

你可以假设 nums[-1] = nums[n] = 1，但注意它们不是真实存在的所以并不能被戳破。
0 ≤ n ≤ 500, 0 ≤ nums[i] ≤ 100
示例:

输入: [3,1,5,8]
输出: 167
解释: nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 315 + 358 + 138 + 181 = 167

```
class Solution {
    public int maxCoins(int[] nums) {

      int n = nums.length;
      int rev[] = new int [n+2];
      int dp[][] = new int [n+2][n+2];

      rev[0] =1;
      rev[n+1] =1;

      for(int i =1;i<n+1;i++){
        rev[i] = nums[i-1];
      }

       for(int i = n-1;i>=0;i--){
        for(int j = i+2;j<n+2;j++){
           for(int k =i+1;k<j;k++){
            // 为什么不是 rev[k-1]] * rev[j] *rev[k+1] 模拟添加气球的
             int sum = rev[i] * rev[j] *rev[k];
             // 为什么包含k? i-k或者k-j
            sum = sum + dp[i][k] +dp[k][j];
             dp[i][j] = Math.max(dp[i][j],sum);
```

```
            }
        }}
         return dp[0][n+1];
    }
}
```

13 647. 回文子串

给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。

具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被计为是不同的子串。

示例 1:

输入: "abc"
输出: 3
解释: 三个回文子串: "a", "b", "c".
示例 2:

输入: "aaa"
输出: 6
说明: 6个回文子串: "a", "a", "a", "aa", "aa", "aaa".

```
class Solution {
        public int countSubstrings(String s) {

            int n = s.length();
            boolean dp [][] = new boolean[n+1][n+1];
            dp[0][0] = true;

            for(int j=1;j<n;j++){
                for(int i =j;i>=0;i--){
                    if(s.charAt(i) == s.charAt(j) &&(j-i<2||dp[i+1][j-1])){
                        dp[i][j] = true;
                    }
                }
            }

            int result =0;

            for(int i =0;i<n;i++){
                for(int j =i;j<n;j++){
                    if(dp[i][j]){
                        result++;

                    }
                }

            }
            return result;

        }
    }
```

14 编辑距离

给你两个单词 word1 和 word2，请你计算出将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

示例 1：

输入：word1 = "horse", word2 = "ros"
输出：3
解释：
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')

```
class Solution {
    public int minDistance(String word1, String word2) {

      int m = word1.length();
      int n= word2.length();
      int dp[][] = new int [m+1][n+1];

        //有初值问题
       // 初始化：当 word 2 长度为 0 时，将 word1 的全部删除
        for (int i = 1; i <= m; i++) {
            dp[i][0] = i;
        }
        // 当 word1 长度为 0 时，就插入所有 word2 的字符
        for (int j = 1; j <= n; j++) {
            dp[0][j] = j;
        }

       for(int i =0;i<m;i++){
         for(int j =0;j<n;j++){
           if(word1.charAt(i)==word2.charAt(j)){
               dp[i+1][j+1] = dp[i][j];
           } else{
                int insert = dp[i + 1][j] + 1;
                // 2、替换一个字符
                int replace = dp[i][j] + 1;
                // 3、删除一个字符
                int delete = dp[i][j + 1] + 1;
                dp[i + 1][j + 1] = Math.min(Math.min(insert, replace), delete);
           }
         }

       }
       return dp[m][n];

    }
}
```

15 97. 交错字符串

给定三个字符串 s1, s2, s3, 验证 s3 是否是由 s1 和 s2 交错组成的。

示例 1：

输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbcbcac"
输出：true
示例 2：

输入：s1 = "aabcc", s2 = "dbbca", s3 = "aadbbbaccc"
输出：false


```
class Solution {
    public boolean isInterleave(String s1, String s2, String s3) {

     int m = s1.length();
     int n = s2.length();
     int t = s3.length();

     if(m+n !=t){
         return false;
     }

     boolean dp[][] = new boolean[m+1][n+1];
```

```
        dp[0][0] = true;
        for(int i =1;i<=m;i++){
            if(s1.charAt(i-1) == s3.charAt(i-1)){
            dp[i][0] = true;
            } else{
                break;
            }
        }
        for(int i =1;i<=n;i++){
            if(s2.charAt(i-1) == s3.charAt(i-1)){
            dp[0][i] = true;
            } else{
                break;
            }
        }

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

           dp[i+1][j+1] = dp[i][j+1] && s1.charAt(i) ==s3.charAt(i+j+1) || dp[i+1]
[j] && s2.charAt(j) ==s3.charAt(i+j+1);

         }
        }
        return dp[m][n];
    }
}
```

16 45 跳跃游戏 II

给定一个非负整数数组，你最初位于数组的第一个位置。

数组中的每个元素代表你在该位置可以跳跃的最大长度。

你的目标是使用最少的跳跃次数到达数组的最后一个位置。

示例:

输入: [2,3,1,1,4]
输出: 2
解释: 跳到最后一个位置的最小跳跃数是 2。
    从下标为 0 跳到下标为 1 的位置，跳 1 步，然后跳 3 步到达数组的最后一个位置。

17 打家劫舍

198. 打家劫舍

337. 打家劫舍 III

在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。

计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

示例 1:

输入: [3,2,3,null,3,null,1]

```
  3
 / \
```

```
 2  3
 \  \
  3  1
```

输出: 7
解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public int rob(TreeNode root) {

      if(root==null){
          return 0;
      }
      int result =root.val;

      if(root.left != null){
          result  = result + rob(root.left.left) + rob(root.left.right);
      }
      if(root.right != null){
          result  = result + rob(root.right.left) + rob(root.right.right);
      }

      return Math.max(result,rob(root.left)+rob(root.right));

    }
```

18 139. 单词拆分

给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

说明:

拆分时可以重复使用字典中的单词。
你可以假设字典中没有重复的单词。
示例 1:

输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
示例 2:

输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
    注意你可以重复使用字典中的单词。
示例 3:

输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
     int len = s.length();
     boolean dp [] = new boolean[len+1];
     dp[0] =true;
        for(int j = 1; j <= s.length(); j++){
            for(int i = j-1; i >= 0; i--){
                dp[j] = dp[i] && check(wordDict,s.substring(i, j));
                if(dp[j])   break;
            }
        }
     return dp[len];
    }

    boolean check(List<String> wordDict,String s){

        return wordDict.contains(s);
    }
}
```

19 [140. 单词拆分 II]

20 [376. 摆动序列]

如果连续数字之间的差严格地在正数和负数之间交替，则数字序列称为摆动序列。第一个差（如果存在的话）可能是正数或负数。少于两个元素的序列也是摆动序列。

例如， [1,7,4,9,2,5] 是一个摆动序列，因为差值 (6,-3,5,-7,3) 是正负交替出现的。相反, [1,4,7,2,5] 和 [1,7,4,5,5] 不是摆动序列，第一个序列是因为它的前两个差值都是正数，第二个序列是因为它的最后一个差值为零。

给定一个整数序列，返回作为摆动序列的最长子序列的长度。 通过从原始序列中删除一些（也可以不删除）元素来获得子序列，剩下的元素保持其原始顺序。

示例 1:

输入: [1,7,4,9,2,5]
输出: 6
解释: 整个序列均为摆动序列。
示例 2:

输入: [1,17,5,10,13,15,10,5,16,8]
输出: 7
解释: 这个序列包含几个长度为 7 摆动序列，其中一个可为[1,17,10,13,10,16,8]。
示例 3:

输入: [1,2,3,4,5,6,7,8,9]
输出: 2
进阶:
你能否用 O(n) 时间复杂度完成此题?

```
class Solution {
    public int wiggleMaxLength(int[] nums) {

     int len = nums.length;
     if(len==0) {
         return 0;
     }

     //0 升序
     //1降序
     int dp[][] = new int [len+1][2];
     dp[0][0] =1;
     dp[0][1] =1;
     for(int i =1;i<len;i++) {
        if(nums[i]>nums[i-1]){
          dp[i][0] = dp[i-1][0];
          dp[i][1] = dp[i-1][0] +1;
        }
        else if(nums[i]<nums[i-1]){
          dp[i][0] = dp[i-1][1]+1;
          dp[i][1] = dp[i-1][1];
        } else{
          dp[i][0] = dp[i-1][0];
          dp[i][1] = dp[i-1][1];
        }
     }
     int max =1;

     for(int i =0;i<len;i++){
         max = Math.max(max,Math.max(dp[i][0],dp[i][1]));
     }



     return max;

    }
}
```

21 10. 正则表达式匹配

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

'.' 匹配任意单个字符
'*' 匹配零个或多个前面的那一个元素
所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

说明:

s 可能为空，且只包含从 a-z 的小写字母。
p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 *。
示例 1:

输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
示例 2:

输入:
s = "aa"
p = "a*"
输出: true
解释: 因为 '*' 代表可以匹配零个或多个前面的那一个元素, 在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。
示例 3:

输入:
s = "ab"
p = ".*"
输出: true
解释: ".*" 表示可匹配零个或多个（'*'）任意字符（'.'）。
示例 4:

输入:
s = "aab"
p = "c*a*b"
输出: true
解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。
示例 5:

输入:
s = "mississippi"
p = "mis*is*p*."
输出: false

```
class Solution {
    public boolean isMatch(String s, String p) {
     if(s==null||p==null) {
         return false;
     }
     int m = s.length();
     int n = p.length();
     boolean dp[][] = new boolean [m+1][n+1];

     char chars [] = s.toCharArray();
     char charP [] = p.toCharArray();
```

```
        if(m==0&&n==0){
            return true;
        }

        dp[0][0] = true;
        for (int j = 1; j < n + 1; j++) {
            if (charP[j - 1] == '*') {
             dp[0][j] = dp[0][j - 2];
            }
    }

       for(int i =1;i<=m;i++) {
            for(int j =1;j<=n;j++) {
            if(chars[i-1] ==charP[j-1]||charP[j-1] =='.'){
                dp[i][j] = dp[i-1][j-1];
            } else if(charP[j-1] =='*') {
                // 化为 0个或1个及以上
//dp[i-1][j] 1个及以上 前一个字母， abcd 与acd* 这个场景下 abcd 与acdd*,最后一个d约掉变成 adc和acd* 变成 dp[i-1][j]问题了
//
                if(chars[i-1] ==charP[j-2]||charP[j-2] =='.'){
                  dp[i][j] = dp[i][j-2] ||dp[i-1][j];
                }
                //*前面的字母只能选0个
                else{
                  dp[i][j] = dp[i][j-2];
                }
            } else{
                dp[i][j] = false;
            }
          }
         }
        return dp[m][n];
      }
}
```

22 65. 有效数字

验证给定的字符串是否可以解释为十进制数字。

例如:

"0" => true
" 0.1 " => true
"abc" => false
"1 a" => false
"2e10" => true
" -90e3   " => true
" 1e" => false
"e3" => false
" 6e-1" => true
" 99e2.5 " => false
"53.5e93" => true
" --6 " => false

"-+3" => false
"95a54e53" => false

说明: 我们有意将问题陈述地比较模糊。在实现代码之前，你应当事先思考所有可能的情况。这里给出一份可能存在于有效十进制数字中的字符列表：

数字 0-9
指数 - "e"
正/负号 - "+"/"-"
小数点 - "."
当然，在输入中，这些字符的上下文也很重要。

23 44. 通配符匹配

给定一个字符串 (s) 和一个字符模式 (p)，实现一个支持 '?' 和 '*' 的通配符匹配。

'?' 可以匹配任何单个字符。
'*' 可以匹配任意字符串（包括空字符串）。
两个字符串完全匹配才算匹配成功。

说明:

s 可能为空，且只包含从 a-z 的小写字母。
p 可能为空，且只包含从 a-z 的小写字母，以及字符 ? 和 *。
示例 1:

输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。
示例 2:

输入:
s = "aa"
p = "*"
输出: true
解释: * 可以匹配任意字符串。
示例 3:

输入:
s = "cb"
p = "?a"
输出: false
解释: '?' 可以匹配 'c', 但第二个 'a' 无法匹配 'b'。
示例 4:

输入:
s = "adceb"
p = "*a*b"
输出: true
解释: 第一个 * *可以匹配空字符串, 第二个* * 可以匹配字符串 "dce".
示例 5:

输入:
s = "acdcb"
p = "a*c?b"
输出: false

```
class Solution {
    public boolean isMatch(String s, String p) {

    if(s==null||p==null){
        return false;
    }

    int m = s.length();
    int n = p.length();

    boolean dp[][] = new boolean[m+1][n+1];
    char chars[] = s.toCharArray();
    char charp[] = p.toCharArray();

    dp[0][0] = true;

    for(int j =1;j<=n;j++){
        dp[0][j] = dp[0][j-1] &&(charp[j-1]=='*');
    }


    for(int i =1;i<=m;i++) {
         for(int j=1;j<=n;j++) {
          if(chars[i-1] == charp[j-1] || charp[j-1]== '?')  {
              dp[i][j] = dp[i-1][j-1];
          }
          else if (charp[j-1]=='*'){
              for(int w =i;w>=0;w--){
               dp[i][j] = dp[w][j-1]||dp[i][j];
               if(dp[i][j]){
                   break;
               }
              }
          }
         }
    }

     return dp[m][n];


    }
}
```

24 403. 青蛙过河

一只青蛙想要过河。 假定河流被等分为 x 个单元格，并且在每一个单元格内都有可能放有一石子（也有可能没有）。 青蛙可以跳上石头，但是不可以跳入水中。

给定石子的位置列表（用单元格序号升序表示），请判定青蛙能否成功过河（即能否在最后一步跳至最后一个石子上）。开始时，青蛙默认已站在第一个石子上，并可以假定它第一步只能跳跃一个单位（即只能从单元格1跳至单元格2）。

如果青蛙上一步跳跃了 k 个单位，那么它接下来的跳跃距离只能选择为 k - 1、k 或 k + 1个单位。另请注意，青蛙只能向前方（终点的方向）跳跃。

请注意：

石子的数量 ≥ 2 且 < 1100；
每一个石子的位置序号都是一个非负整数，且其 < 231；
第一个石子的位置永远是0。
示例 1:

[0,1,3,5,6,8,12,17]

总共有8个石子。
第一个石子处于序号为0的单元格的位置, 第二个石子处于序号为1的单元格的位置,
第三个石子在序号为3的单元格的位置，以此定义整个数组...
最后一个石子处于序号为17的单元格的位置。

返回 true。即青蛙可以成功过河，按照如下方案跳跃：
跳1个单位到第2块石子, 然后跳2个单位到第3块石子, 接着
跳2个单位到第4块石子, 然后跳3个单位到第6块石子,
跳4个单位到第7块石子, 最后，跳5个单位到第8个石子（即最后一块石子）。
示例 2:

[0,1,2,3,4,8,9,11]

返回 false。青蛙没有办法过河。
这是因为第5和第6个石子之间的间距太大，没有可选的方案供青蛙跳跃过去。

```
/*
思路①、使用二维数组的动态规划
        动态规划
        dp[i][k] 表示能否由前面的某一个石头 j 通过跳 k 步到达当前这个石头 i ，这个 j 的范围是 [1, i - 1]
        当然，这个 k 步是 i 石头 和 j 石头之间的距离
        那么对于 j 石头来说，跳到 j 石头的上一个石头的步数就必须是这个 k - 1 || k || k + 1
        由此可得状态转移方程：dp[i][k] = dp[j][k - 1] || dp[j][k] || dp[j][k + 1]
*/
class Solution {
    public boolean canCross(int[] stones) {

        int len = stones.length;

        if(stones[1] != 1){
            return false;
        }

        boolean[][] dp = new boolean[len][len + 1];
        dp[0][0] = true;
        for(int i = 1; i < len; i++){
            for(int j = 0; j < i; j++){
                int k = stones[i] - stones[j];
                /*
                    为什么有这么个判断？
```

```
                        因为其他石头跳到第 i 个石头跳的步数 k 必定满足 k <= i
                        这又是为什么？
                        1、比如 nums = [0,1,3,5,6,8,12,17]
                            那么第 0 个石头跳到第 1 个石头，步数肯定为 1，然后由于后续最大的步数是 k + 1，因此第 1 个石头最大只能跳 2 个单位
                            因此如果逐个往上加，那么第 2 3 4 5 ... 个石头最多依次跳跃的步数是 3 4 5 6...
                        2、 第 i 个石头能跳的最大的步数是 i + 1，那么就意味着其他石头 j 跳到第 i 个石头的最大步数只能是 i 或者 j + 1
                            而 这个 k 是其他石头跳到 i 石头上来的，因此 k 必须 <= i （或者是 k <= j + 1）
                    */
                    if(k <= i){
                        dp[i][k] = dp[j][k - 1] || dp[j][k] || dp[j][k + 1];
                        //提前结束循环直接返回结果
                        if(i == len - 1 && dp[i][k]){
                            return true;
                        }
                    }
                }
            }
            return false;
        }
    }
```

26 887. 鸡蛋掉落

你将获得 K 个鸡蛋，并可以使用一栋从 1 到 N 共有 N 层楼的建筑。

每个蛋的功能都是一样的，如果一个蛋碎了，你就不能再把它掉下去。

你知道存在楼层 F ，满足 0 <= F <= N 任何从高于 F 的楼层落下的鸡蛋都会碎，从 F 楼层或比它低的楼层落下的鸡蛋都不会破。

每次移动，你可以取一个鸡蛋（如果你有完整的鸡蛋）并把它从任一楼层 X 扔下（满足 1 <= X <= N）。

你的目标是确切地知道 F 的值是多少。

无论 F 的初始值如何，你确定 F 的值的最小移动次数是多少？

示例 1：

输入：K = 1, N = 2
输出：2
解释：
鸡蛋从 1 楼掉落。如果它碎了，我们肯定知道 F = 0 。
否则，鸡蛋从 2 楼掉落。如果它碎了，我们肯定知道 F = 1 。
如果它没碎，那么我们肯定知道 F = 2 。
因此，在最坏的情况下我们需要移动 2 次以确定 F 是多少。
示例 2：

输入：K = 2, N = 6
输出：3
示例 3：

输入：K = 3, N = 14
输出：4

27 413. 等差数列划分

数组 A 包含 N 个数，且索引从0开始。数组 A 的一个子数组划分为数组 (P, Q)，P 与 Q 是整数且满足 0<=P<Q<N 。

如果满足以下条件，则称子数组(P, Q)为等差数组：

元素 A[P], A[p + 1], ..., A[Q - 1], A[Q] 是等差的。并且 P + 1 < Q 。

函数要返回数组 A 中所有为等差数组的子数组个数。

示例:

A = [1, 2, 3, 4]

返回: 3, A 中有三个子等差数组: [1, 2, 3], [2, 3, 4] 以及自身 [1, 2, 3, 4]。

```
class Solution {
    public int numberOfArithmeticSlices(int[] A) {

        int len = A.length;
        if(len<3) {
            return 0;
        }
        boolean dp[][] = new boolean[len+1][len+1];

        for(int i=0;i<len-2;i++)  {
           for(int j=i+2;j<len;j++) {
               if(j-i==2){
                  dp[i][j] = A[j]-A[j-1]==A[j-1]-A[j-2];
               }
               if(dp[i][j-1]) {
                 dp[i][j] = dp[i][j]||A[j]-A[j-1]==A[j-1]-A[j-2];
               }
           }
        }
        int cunt =0;
         for(int i=0;i<len-2;i++)  {
           for(int j=i+2;j<len;j++) {
               if(dp[i][j]){
                 cunt++;
               }
           }
        }
        return cunt;

    }
}
```

28 517. 超级洗衣机

假设有 n 台超级洗衣机放在同一排上。开始的时候，每台洗衣机内可能有一定量的衣服，也可能是空的。

在每一步操作中，你可以选择任意 m （1 ≤ m ≤ n） 台洗衣机，与此同时将每台洗衣机的一件衣服送到相邻的一台洗衣机。

给定一个非负整数数组代表从左至右每台洗衣机中的衣物数量，请给出能让所有洗衣机中剩下的衣物的数量相等的最少的操作步数。如果不能使每台洗衣机中衣物的数量相等，则返回 -1。

示例 1：

输入: [1,0,5]

输出: 3

解释:
第一步: 1 0 <-- 5 => 1 1 4
第二步: 1 <-- 1 <-- 4 => 2 1 3
第三步: 2 1 <-- 3 => 2 2 2
示例 2：

输入: [0,3,0]

输出: 2

解释:
第一步: 0 <-- 3 0 => 1 2 0
第二步: 1 2 --> 0 => 1 1 1
示例 3:

输入: [0,2,0]

输出: -1

解释:
不可能让所有三个洗衣机同时剩下相同数量的衣物。

```
class Solution {
    public int findMinMoves(int[] machines) {

      int len = machines.length;
      if(len ==0) return 0;
      int sum =0;
      for(int i=0;i<len;i++) {
          sum += machines[i];
      }
      if(sum%len !=0) {
          return -1;
      }
      int mid = sum/len;

      //dp[][0]  多的的
      //dp[][1]   少的
      int dp[][] = new int[len+1][2];
      int cunt =0;
```

```
        for(int i=1;i<=len;i++) {
            int res =0;
            //往前移动和给后面移动
            int tem =0;
            if(machines[i-1]>mid){
                //一共多出来
               res = machines[i-1] - mid + dp[i-1][0];
               //还多
               if(res>=dp[i-1][1]) {
                  tem = res;
                  dp[i][0] = res -dp[i-1][1];
                  dp[i][1] =0;
               }
               // 还欠一点
               else{
                  tem = dp[i-1][1];
                  dp[i][0] =0;
                  dp[i][1]= dp[i-1][1] -res;
               }



            }else if (machines[i-1]<mid) {
                //一共欠
               res = mid -machines[i-1]+ dp[i-1][1];
               //还多
               if(res<=dp[i-1][0]) {
                     tem = dp[i - 1][0];
                     dp[i][1] = 0;
                     dp[i][0] = dp[i - 1][0] - res;
               }
               // 还欠一点
               else{
                     dp[i][1] = res - dp[i - 1][0];
                     dp[i][0] = 0;
                     tem = dp[i][1];

               }

            } else{
                dp[i][0] = dp[i-1][0];
                dp[i][1] = dp[i-1][1];
            }
            cunt = Math.max(cunt,tem);

        }

        return cunt;


    }
}
```

29 边界为1的正方形问题

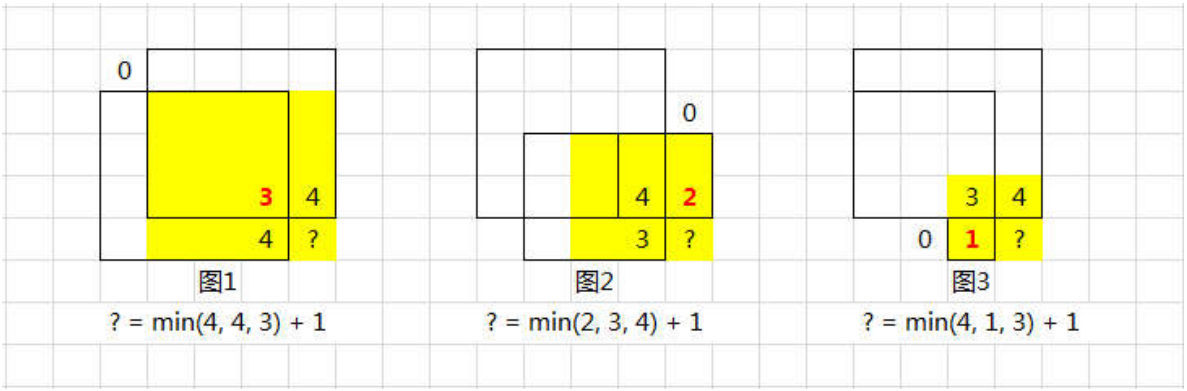
221. 最大正方形

难度中等532收藏分享切换为英文关注反馈

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

理解 min(上, 左, 左上) + 1

先来阐述简单共识



若形成正方形（非单 1），以当前为右下角的视角看，则需要：当前格、上、左、左上都是 1
可以换个角度：当前格、上、左、左上都不能受 0 的限制，才能成为正方形

上面详解了 三者取最小 的含义：

图 1：受限于左上的 0
图 2：受限于上边的 0
图 3：受限于左边的 0
数字表示：以此为正方形右下角的最大边长
黄色表示：格子 ? 作为右下角的正方形区域

输入：

```
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
```

输出：4

```
    public int maximalSquare(char[][] matrix) {
    // base condition
    if (matrix == null || matrix.length < 1 || matrix[0].length < 1) return 0;

    int height = matrix.length;
    int width = matrix[0].length;
    int maxSide = 0;

    // 相当于已经预处理新增第一行、第一列均为0
    int[][] dp = new int[height + 1][width + 1];
```

```
    for (int row = 0; row < height; row++) {
        for (int col = 0; col < width; col++) {
            if (matrix[row][col] == '1') {
                dp[row + 1][col + 1] = Math.min(Math.min(dp[row + 1][col],
dp[row][col + 1]), dp[row][col]) + 1;
                maxSide = Math.max(maxSide, dp[row + 1][col + 1]);
            }
        }
    }
    return maxSide * maxSide;
}
```

链接：https://leetcode-cn.com/problems/maximal-square/solution/li-jie-san-zhe-qu-zui-xiao-1-by-lzhlyle/

1139. 最大的以 1 为边界的正方形

给你一个由若干 0 和 1 组成的二维网格 grid，请你找出边界全部由 1 组成的最大 正方形 子网格，并返回该子网格中的元素数量。如果不存在，则返回 0。

示例 1：

输入：grid = [[1,1,1],[1,0,1],[1,1,1]]
输出：9
示例 2：

输入：grid = [[1,1,0,0]]
输出：1

提示：

1 <= grid.length <= 100
1 <= grid[0].length <= 100
grid[i][j] 为 0 或 1

```
class Solution {
// 使用3维数组dp[n + 1][m + 1][2](数组下标从1开始)
// dp[i][j][0]:表示第i行第j列的1往 左边 最长连续的1的个数
// dp[i][j][1]:表示第i行第j列的1往 上面 最长连续的1的个数
    public int largest1BorderedSquare(int[][] grid) {
     int m = grid.length;
     if(m==0){
         return 0;
     }
    int n = grid[0].length;
    int res =0;
    int dp[][][] = new int[m+1][n+1][2];
    for(int i =1;i<=m;i++) {
      for(int j =1;j<=n;j++) {
        int d=0;
        if(grid[i-1][j-1]==1) {
          dp[i][j][0] = dp[i-1][j][0]+1;
          dp[i][j][1] = dp[i][j-1][1]+1;

           d = Math.min(dp[i-1][j][0],dp[i][j-1][1]);
           while(d>0) {
             if(dp[i][j-d][0]>d&&dp[i-d][j][1]>d) {
```

```
                break;
            }
            d--;
          }

          res = Math.max(res,d+1);
        }

      }
    }

    return res*res;
    }
}
```

30 877. 石子游戏

31 120. 三角形最小路径和

给定一个三角形，找出自顶向下的最小路径和。每一步只能移动到下一行中相邻的结点上。

相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。

例如，给定三角形：

```
[
   [2],
  [3,4],
 [6,5,7],
[4,1,8,3]
]
```

自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。

说明：如果你可以只使用 O(n) 的额外空间（n 为三角形的总行数）来解决这个问题，那么你的算法会很加分。

32 343. 整数拆分

给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。

示例 1:

输入: 2
输出: 1
解释: 2 = 1 + 1, 1 × 1 = 1。
示例 2:

输入: 10
输出: 36
解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。

33 115. 不同的子序列

给定一个字符串 S 和一个字符串 T，计算在 S 的子序列中 T 出现的个数。

一个字符串的一个子序列是指，通过删除一些（也可以不删除）字符且不干扰剩余字符相对位置所组成的新字符串。（例如，"ACE" 是 "ABCDE" 的一个子序列，而 "AEC" 不是）

题目数据保证答案符合 32 位带符号整数范围。

示例 1：

输入：S = "rabbbit", T = "rabbit"
输出：3
解释：

如下图所示, 有 3 种可以从 S 中得到 "rabbit" 的方案。
(上箭头符号 ^ 表示选取的字母)

rabbbit
^^^^ ^^
rabbbit
^^ ^^^^
rabbbit
^^^ ^^^
示例 2：

输入：S = "babgbag", T = "bag"
输出：5
解释：

如下图所示, 有 5 种可以从 S 中得到 "bag" 的方案。
(上箭头符号 ^ 表示选取的字母)

babgbag
^^ ^
babgbag
^^ ^
babgbag
^ ^^
babgbag
 ^ ^^
babgbag
  ^^^

34 63. 不同路径 II

35 131. 分割回文串

36 132. 分割回文串 II

给定一个字符串 *s*，将 *s* 分割成一些子串，使每个子串都是回文串。

返回符合要求的最少分割次数。

示例:

```
输入: "aab"
输出: 1
解释: 进行一次分割就可将 s 分割成 ["aa","b"] 这样两个回文子串。
```

33 322. 零钱兑换

给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

示例 1:

输入: coins = [1, 2, 5], amount = 11
输出: 3
解释: 11 = 5 + 5 + 1
示例 2:

输入: coins = [2], amount = 3
输出: -1

```

```

34 279. 完全平方数

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

示例 1:

输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
示例 2:

输入: n = 13
输出: 2
解释: 13 = 4 + 9.

35 91. 解码方法

难度中等486收藏分享切换为英文关注反馈

一条包含字母 `A-Z` 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的**非空**字符串，请计算解码方法的总数。

示例 1:

```
输入: "12"
输出: 2
解释: 它可以解码为 "AB"（1 2）或者 "L"（12）。
```

示例 2:

```
输入: "226"
输出: 3
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。
```

```java
class Solution {
    public int numDecodings(String s) {
     int len = s.length();
     List<String> diction = new ArrayList<>();
     diction.add("1");
     diction.add("2");
     diction.add("3");
     diction.add("4");
     diction.add("5");
     diction.add("6");
     diction.add("7");
     diction.add("8");
     diction.add("9");
     diction.add("10");
     diction.add("11");
     diction.add("12");
     diction.add("13");
     diction.add("14");
     diction.add("15");
     diction.add("16");
     diction.add("11");
     diction.add("12");
     diction.add("13");
     diction.add("14");
     diction.add("15");
     diction.add("16");
     diction.add("17");
     diction.add("18");
     diction.add("19");
     diction.add("20");
     diction.add("21");
     diction.add("22");
     diction.add("23");
     diction.add("24");
     diction.add("25");
     diction.add("26");

     int dp[] = new int[len+1];
     dp[0] = 1;
     if(len>=1 && is(s.substring(0,1),diction)){
       dp[1]=1;
     }
```

```
        for(int i =2;i<=len;i++){
            if(is(s.substring(i-1,i),diction)){
                dp[i] += dp[i-1];
            }
            if(is(s.substring(i-2,i),diction)){
                dp[i] += dp[i-2];
            }
        }
         return dp[len];
        }

        private boolean is(String s,List<String> diction){
            return diction.contains(s);
        }
    }
```

36 279. 完全平方数

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

示例 1:

输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.
示例 2:

输入: n = 13
输出: 2
解释: 13 = 4 + 9.

37 343. 整数拆分

38 64最小路径和

给定一个包含非负整数的 m x n 网格，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。

说明：每次只能向下或者向右移动一步。

示例:

输入:
[
  [1,3,1],
  [1,5,1],
  [4,2,1]
]
输出: 7
解释: 因为路径 1→3→1→1→1 的总和最小。

39 673最长递增子序列的个数 最长递增子序列的个数

40 674最长连续递增序列

1.33. 搜索旋转排序数组](https://leetcode-cn.com/problems/search-in-rotated-sorted-array/)

难度中等836收藏分享切换为英文关注反馈

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 `[0,1,2,4,5,6,7]` 可能变为 `[4,5,6,7,0,1,2]`)。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 `-1` 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 *O*(log *n*) 级别。

示例 1:

```
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4
```

示例 2:

```
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

https://leetcode-cn.com/problems/search-in-rotated-sorted-array/solution/ji-bai-liao-9983de-javayong-hu-by-reedfan/

2.153题

3 287. 寻找重复数

给定一个包含 *n* + 1 个整数的数组 *nums*，其数字都在 1 到 *n* 之间（包括 1 和 *n*），可知至少存在一个重复的整数。假设只有一个重复的整数，找出这个重复的数。

示例 1:

```
输入: [1,3,4,2,2]
输出: 2
```

示例 2:

```
输入: [3,1,3,4,2]
输出: 3
```

抽屉原理 https://leetcode-cn.com/problems/find-the-duplicate-number/solution/er-fen-fa-si-lu-ji-dai-ma-python-by-liweiwei1419/

public class Solution {

```
public int findDuplicate(int[] nums) {
    int len = nums.length;
    int left = 1;
    int right = len - 1;
    while (left < right) {
        // 在 Java 里可以这么用，当 left + right 溢出的时候，无符号右移保证结果依然正确
        int mid = (left + right) >>> 1;

        int cnt = 0;
        for (int num : nums) {
            if (num <= mid) {
                cnt += 1;
            }
        }

        // 根据抽屉原理，小于等于 4 的个数如果严格大于 4 个
        // 此时重复元素一定出现在 [1, 4] 区间里
        if (cnt > mid) {
            // 重复元素位于区间 [left, mid]
            right = mid;
        } else {
            // if 分析正确了以后，else 搜索的区间就是 if 的反面
            // [mid + 1, right]
            left = mid + 1;
        }
    }
    return left;
}}
```

4 .88