换了个新的排版主题，哈哈，感觉如何?

738.单调递增的数字

给定一个非负整数 N，找出小于或等于 N 的最大的整数，同时这个整数需要满足其各个位数上的数字是单调递增。

（当且仅当每个相邻位数上的数字 x 和 y 满足 x <= y 时，我们称这个整数是单调递增的。）

示例 1:
输入: N = 10
输出: 9

示例 2:
输入: N = 1234
输出: 1234

示例 3:
输入: N = 332
输出: 299

说明: N 是在 [0, 10^9] 范围内的一个整数。

举个例子:
对于98，一旦出现strNum[i-1]>strNum[i]的情况(非单调递增),首先让strNum[i-1]--,然后strNum[i]给为9.
局部最优:遇到strNum[i-1]>strNum[i]的情况，**让strNum[i-1]--**,然后**strNum[i]给为9,**可以保证这两位变成**最大单调递增整数**

此时是从前向后遍历还是从后向前遍历呢?

从前向后遍历的话,遇到strNum[i-1]>strNum[i]的情况，让strNum[i-1]--,让strNum[i]给为9,此时可能strNum[i-1]小于strNum[i-2]
所以从前向后遍历更为合理.
思路:从后向前遍历时要记录最后一个满足strNum[i-1]>strNum[i]的情况的i，然后对x>=i且x<strNum.size()的位值赋为9

```
class Solution {
public:
    int monotoneIncreasingDigits(int N) {
 //首先想将整数值转换为string
 string strNum=to_string(N);
 int flag=strNum.size();
```

```
7   for(int i=strNum.size()-1;i>0;i--){
8   if(strNum[i-1]>strNum[i]){
9   flag=i;
10   strNum[i-1]--;
11   }
12   }
13   //把flag后面的所有位赋值为'9'，如果flag==strNum.size说明原来的数值也满足相关性质
14   for(int i=flag;i<strNum.size();i++)
15   strNum[i]='9';
16   return stoi(strNum);
17      }
18  };
```

难点:
1、贪心思想，重点是什么情况下是局部最优
2、**遍历方向**是从前向后遍历还是从后向前遍历
3、代码实现用一个flag来标记从哪里开始赋值9

都知道买卖股票系列是动态规划长项，那么我来给大家拓展一下贪心的解法

714. 买卖股票的最佳时机含手续费

给定一个整数数组 prices，其中第 i 个元素代表了第 i 天的股票价格 ；非负整数 fee 代表了交易股票的手续费用。

你可以无限次地完成交易，但是你每笔交易都需要付手续费。如果你已经购买了一个股票，在卖出它之前你就不能再继续购买股票了。

返回获得利润的最大值。

注意：这里的一笔交易指买入持有并卖出股票的整个过程，每笔交易你只需要为支付一次手续费。

示例 1: 输入: prices = [1, 3, 2, 8, 4, 9], fee = 2 输出: 8

解释: 能够达到的最大利润:
在此处买入 prices[0] = 1
在此处卖出 prices[3] = 8
在此处买入 prices[4] = 4
在此处卖出 prices[5] = 9
总利润: ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

> 注意：这里的一笔交易是指**买入持有**并**卖出股票**的整个过程，每笔交易只需要为支付**一次手续费**

引用:在[贪心算法：122.买卖股票的最佳时机II]中使用贪心策略不用关心具体什么时候买卖，只要收集每天的正利润，最后稳稳的就是最大利润了。

> 由于本题目具有手续费，就要关系什么时候买卖了，**因为计算所获得利润，要考虑买卖利润可能不足以手续费的情况**

> 买入日期:遇到更低点就记录一下
> 卖出日期:只要当价格大于(买入价格+手续费),就可以收获利润。
> 所以有以下几种情况：

- 情况一：收获利润的这一天并不是收获利润区间里的最后一天（不是真正的卖出，相当于持有股票），所以后面要继续收获利润。
- 情况二：前一天是收获利润区间里的最后一天（相当于真正的卖出了），今天要重新记录最小价格了。
- 情况三：不作操作，保持原有状态（买入，卖出，不买不卖）

如何判断是否利润区间的最后一天呢?

并不实际去判断是否是最后一天而是把minPrice=Prices[i]-fee,此时如果还在收获利润区间里，表示并不是真正卖出，到最后真正收获利润时不会多减一次手续费.
此时把minPrice=prices[i]-fee时只要i后面的价格值大于prices[i]就会被计算入result即不考虑手续费因素了。

什么时候是利润区间最后一天呢?

当存在一个价格小于minPrice时，那么就会新的买入，此时就得出利润区间的最后一天了即最后一次收获利润的那一次。

```
class Solution {
public:
    int maxProfit(vector<int>& prices, int fee) {
 int result=0;
 int minPrice=prices[0];
 for(int i=1;i<prices.size();i++){
 //第一种情况遇到比minPrice还低的价格
 if(minPrice>prices[i])
 minPrice=prices[i];
  //第三种情况,现在收获所得利润不够fee
  if(prices[i]-minPrice<=fee) continue;
  //第二种情况，所得利润大于fee
  result+=prices[i]-minPrice-fee;
  //非常关键的一步，此后在新买入前的所有卖出所得利润都不用考虑手续费
  minPrice=prices[i]-fee;
  }
```

```
17   return result;
18       }
19  };
```

968.监控二叉树

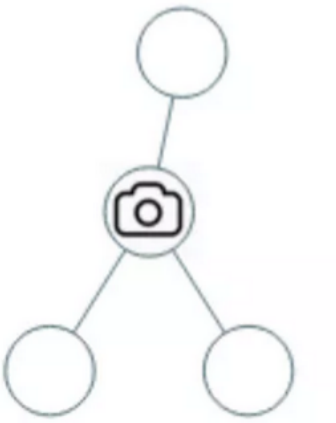
题目地址 : https://leetcode-cn.com/problems/binary-tree-cameras/

给定一个二叉树，我们在树的节点上安装摄像头。

节点上的每个摄影头都可以监视其父对象、自身及其直接子对象。

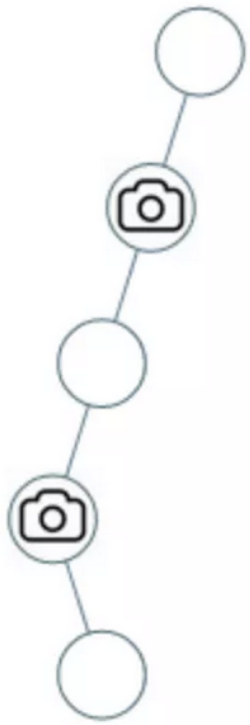
计算监控树的所有节点所需的最小摄像头数量。

示例 1：



输入：[0,0,null,0,0]
输出：1
解释：如图所示，一台摄像头足以监控所有节点。

示例 2：



输入：[0,0,null,0,null,0,null,null,0]

输出：2

解释：需要至少两个摄像头来监视树的所有节点。上图显示了摄像头放置的有效位置之

> **为什么摄像头都没有放在叶子节点上?**
> 首先我们可以发现摄像头可以监控到上中下三层，如果放在叶子节点上那么就浪费了一层，所以把摄像头放在叶子节点的父节点上。
> **为什么不从头结点开始看起呢，为啥要从叶子节点看呢?**
> 因为头结点放不放摄像头也就省下一个摄像头，叶子节点放不放摄像头省下了的摄像头数量是指数阶别的
> **局部最优:让叶子节点的父节点安装摄像头,所用摄像头最少，整体最优:全部摄像头数量所用最少!**

大体思路:先给叶子节点的父节点安装摄像头，然后隔两个节点放一个摄像头，直至到二叉树头节点

难点:

1、二叉树的遍历

> 可以采用**后序遍历算法，也就是左右中的顺序**，这样就可以在回溯过程中**从下到上**进行推导了

2、如何隔两个节点放一个摄像头

> 此时用到**状态转移公式,我们分别用三个数字表示:0表示无覆盖，1表示有摄像头，2表示有覆盖**
> **我们定义空节点为有覆盖状态即2(如果空节点为无覆盖状态那么叶子节点必须要有摄像头，如果空节点为有摄像头状态那么上一个摄像头则安在叶子节点的父节点父节点)**

```
1  递归关系：
```

```
if(cur==NULL) return 2;//返回表示空节点为有覆盖状态
单层逻辑关系:一共有4种情况:
1:左右节点都有覆盖,那么此时中间节点应该就是无覆盖状态
2:左右节点至少有一个无覆盖情况,则中间节点(父节点)应该放摄像头:
3:左右节点至少有一个有摄像头且没有无覆盖情况,那中间节点就是有覆盖状态
left == 1 && right == 2 左节点有摄像头,右节点有覆盖
left == 2 && right == 1 左节点有覆盖,右节点有摄像头
left == 1 && right == 1 左右节点都有摄像头
 4:头结点没有覆盖
 还要判断根节点,如果没有覆盖,result++
```

这个是贪心算法的最后一章总结网站

https://mp.weixin.qq.com/s/ltyoYNr0moGEYeRtcjZL3Q