一行代码就能解决的算法题

相关推荐：

- 学习算法和数据结构的思路指南
- 我用四个命令概括了 Git 的所有套路

读完本文，你不仅学会了算法套路，还可以顺便去 LeetCode 上拿下如下题目：

292.Nim游戏

877.石子游戏

319.灯泡开关

下文是我在 LeetCode 刷题过程中总结的三道有趣的「脑筋急转弯」题目，可以使用算法编程解决，但只要稍加思考，就能找到规律，直接想出答案。

一、Nim 游戏

游戏规则是这样的：你和你的朋友面前有一堆石子，你们轮流拿，一次至少拿一颗，最多拿三颗，谁拿走最后一颗石子谁获胜。

假设你们都很聪明，由你第一个开始拿，请你写一个算法，输入一个正整数 n，返回你是否能赢（true 或 false）。

比如现在有 4 颗石子，算法应该返回 false。因为无论你拿 1 颗 2 颗还是 3 颗，对方都能一次性拿完，拿走最后一颗石子，所以你一定会输。

首先，这道题肯定可以使用动态规划，因为显然原问题存在子问题，且子问题存在重复。但是因为你们都很聪明，涉及到你和对手的博弈，动态规划会比较复杂。

我们解决这种问题的思路一般都是反着思考：

如果我能赢，那么最后轮到我取石子的时候必须要剩下 1~3 颗石子，这样我才能一把拿完。

如何营造这样的一个局面呢？显然，如果对手拿的时候只剩 4 颗石子，那么无论他怎么拿，总会剩下 1~3 颗石子，我就能赢。

如何逼迫对手面对 4 颗石子呢？要想办法，让我选择的时候还有 5~7 颗石子，这样的话我就有把握让对方不得不面对 4 颗石子。

如何营造 5~7 颗石子的局面呢？让对手面对 8 颗石子，无论他怎么拿，都会给我剩下 5~7 颗，我就能赢。

这样一直循环下去，我们发现只要踩到 4 的倍数，就落入了圈套，永远逃不出 4 的倍数，而且一定会输。所以这道题的解法非常简单：

```
bool canWinNim(int n) {
    // 如果上来就踩到 4 的倍数，那就认输吧
    // 否则，可以把对方控制在 4 的倍数，必胜
    return n % 4 != 0;
}
```

二、石头游戏

游戏规则是这样的：你和你的朋友面前有一排石头堆，用一个数组 piles 表示，piles[i] 表示第 i 堆石子有多少个。你们轮流拿石头，一次拿一堆，但是只能拿走最左边或者最右边的石头堆。所有石头被拿完后，谁拥有的石头多，谁获胜。

假设你们都很聪明，由你第一个开始拿，请你写一个算法，输入一个数组 piles，返回你是否能赢（true 或 false）。

注意，石头的堆的数量为偶数，所以你们两人拿走的堆数一定是相同的。石头的总数为奇数，也就是你们最后不可能拥有相同多的石头，一定有胜负之分。

举个例子，`piles=[2, 1, 9, 5]`，你先拿，可以拿 2 或者 5，你选择 2。

`piles=[1, 9, 5]`，轮到对手，可以拿 1 或 5，他选择 5。

`piles=[1, 9]` 轮到你拿，你拿 9。

最后，你的对手只能拿 1 了。

这样下来，你总共拥有 `2 + 9 = 11` 颗石头，对手有 `5 + 1 = 6` 颗石头，你是可以赢的，所以算法应该返回 true。

你看到了，并不是简单的挑数字大的选，为什么第一次选择 2 而不是 5 呢？因为 5 后面是 9，你要是贪图一时的利益，就把 9 这堆石头暴露给对手了，那你就要输了。

这也是强调双方都很聪明的原因，算法也是求最优决策过程下你是否能赢。

这道题又涉及到两人的博弈，也可以用动态规划算法暴力试，比较麻烦。但我们只要对规则深入思考，就会大惊失色：只要你足够聪明，你是必胜无疑的，因为你是先手。

```
boolean stoneGame(int[] piles) {
    return true;
}
```

这是为什么呢，因为题目有两个条件很重要：一是石头总共有偶数堆，石头的总数是奇数。这两个看似增加游戏公平性的条件，反而使该游戏成为了一个割韭菜游戏。我们以 `piles=[2, 1, 9, 5]` 讲解，假设这四堆石头从左到右的索引分别是 1，2，3，4。

如果我们把这四堆石头按索引的奇偶分为两组，即第 1、3 堆和第 2、4 堆，那么这两组石头的数量一定不同，也就是说一堆多一堆少。因为石头的总数是奇数，不能被平分。

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。

你最开始可以选择第 1 堆或第 4 堆。如果你想要偶数堆，你就拿第 4 堆，这样留给对手的选择只有第 1、3 堆，他不管怎么拿，第 2 堆又会暴露出来，你就可以拿。同理，如果你想拿奇数堆，你就拿第 1 堆，留给对手的只有第 2、4 堆，他不管怎么拿，第 3 堆又给你暴露出来了。

也就是说，你可以在第一步就观察好，奇数堆的石头总数多，还是偶数堆的石头总数多，然后步步为营，就一切尽在掌控之中了。知道了这个漏洞，可以整一整不知情的同学了。

三、电灯开关问题

这个问题是这样描述的：有 n 盏电灯，最开始时都是关着的。现在要进行 n 轮操作：

第 1 轮操作是把每一盏电灯的开关按一下（全部打开）。

第 2 轮操作是把每两盏灯的开关按一下（就是按第 2，4，6... 盏灯的开关，它们被关闭）。

第 3 轮操作是把每三盏灯的开关按一下（就是按第 3，6，9... 盏灯的开关，有的被关闭，比如 3，有的被打开，比如 6）...

如此往复，直到第 n 轮，即只按一下第 n 盏灯的开关。

现在给你输入一个正整数 n 代表电灯的个数，问你经过 n 轮操作后，这些电灯有多少盏是亮的?

我们当然可以用一个布尔数组表示这些灯的开关情况，然后模拟这些操作过程，最后去数一下就能出结果。但是这样显得没有灵性，最好的解法是这样的：

```
int bulbSwitch(int n) {
    return (int)Math.sqrt(n);
}
```

什么？这个问题跟平方根有什么关系？其实这个解法挺精妙，如果没人告诉你解法，还真不好想明白。

首先，因为电灯一开始都是关闭的，所以某一盏灯最后如果是点亮的，必然要被按奇数次开关。

我们假设只有 6 盏灯，而且我们只看第 6 盏灯。需要进行 6 轮操作对吧，请问对于第 6 盏灯，会被按下几次开关呢？这不难得出，第 1 轮会被按，第 2 轮，第 3 轮，第 6 轮都会被按。

为什么第 1、2、3、6 轮会被按呢？因为 `6=1*6=2*3`。一般情况下，因子都是成对出现的，也就是说开关被按的次数一般是偶数次。但是有特殊情况，比如说总共有 16 盏灯，那么第 16 盏灯会被按几次?

`16=1*16=2*8=4*4`

其中因子 4 重复出现，所以第 16 盏灯会被按 5 次，奇数次。现在你应该理解这个问题为什么和平方根有关了吧?

不过，我们不是要算最后有几盏灯亮着吗，这样直接平方根一下是啥意思呢？稍微思考一下就能理解了。

就假设现在总共有 16 盏灯，我们求 16 的平方根，等于 4，这就说明最后会有 4 盏灯亮着，它们分别是第 `1*1=1` 盏、第 `2*2=4` 盏、第 `3*3=9` 盏和第 `4*4=16` 盏。

就算有的 n 平方根结果是小数，强转成 int 型，也相当于一个最大整数上界，比这个上界小的所有整数，平方后的索引都是最后亮着的灯的索引。所以说我们直接把平方根转成整数，就是这个问题的答案。

==其他语言代码==