

# 和小浩学算法

## 关于本书

小浩算法是我在疫情期间完成的一部图解算法题典！目前共完成 **105道** 高频面试算法题目，全部采用漫画图解的方式。该教程目前共有 **11w** 人阅读。面向 **算法小白** 和 **初中阶读者**。所有代码均在 leetcode 上测试运行。

### 图解算法100篇 【0523版】

小浩算法

小浩截止到5月10号，共写了100篇 算法题

文章 100 阅读 11.8万

在看 114

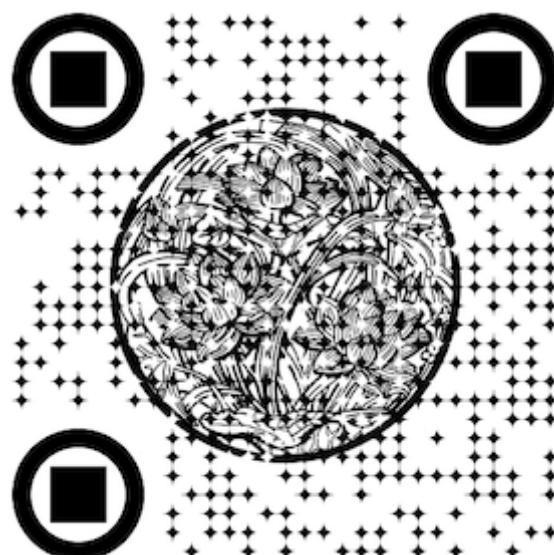
目前该教程**全部免费**，正因如此被一些不良商家拿去卖钱！

所以我把全部的内容都同步到了 [github](#)

如果可以的话，希望大家可以帮我点一个 star，防止有更多的人上当受骗！

同时，为了大家更好的交流，我创建了 [万人刷题群](#)。

我承诺群里无广告，不套路，仅作算法交流学习使用！下方扫码回复 **【进群】** 即可



当然，大家也可以直接到 [小浩算法网站](#) 进行学习：

The screenshot displays two detailed algorithm explanations on the 'Little Xiao's Algorithm' website:

- Topic Analysis (202):** This section shows a step-by-step analysis of a problem involving three boxes labeled 'a', 'b', and 'c'. It includes a diagram illustrating the movement of stones between boxes. The text states: "比如: a = 1, b = 2, c = 5".
- LRU Usage (Explanation) (146):** This section provides a detailed explanation of LRU cache usage. It includes a diagram of a cache with two slots, labeled 'key-1' and 'key-2', each with its value. The text states: "LRUCache cache = new LRUCache(); cache.put(1,1); cache.put(2,2); cache.get(1);". It also includes a step-by-step process:
  - 第一步：我们声明一个 LRUCache，长这样为 2。
  - 第二步：我们分别向 cache 里边 put(1,1) 和 put(2,2)，这说明因为最近使用的是 2 (put 之后) 所以在后，1 在后。
  - 第三步：我们 get(1)，也就是我们使用了 1，所以需要将 1 移到前面。
  - 第四步：此时我们 put(3,3)，因为 2 是最近很少使用的，所以我们需要将 2 进行应用。此时我们调用 get(2)，就应该返回 1。
  - 第五步：我们需要 put(4,4)，同理我们将其作废。此时如果 get(1)，也是返回 -1。
  - 第六步：此时我们 get(3)，实际为删除 3 的位置。

## 学习指南

### 为什么要这样做这样的一个算法图解合集

网上的算法教程杂乱且分散，质量层次不齐，浪费了大家大量宝贵的时间。很多题解，在我掌握题目后去看都费劲，更何况对于一些初学者。

### 本教程阅读门槛

本教程基本没有学习门槛。因为在每道题目中，我都会尽量去串基础知识，以达到学以致用的效果。

### 学完本教程期望达到什么样的目的

- 掌握基本的数据结构与算法
- 掌握各类型高频面试算法题

### 本教程有何特色

每一道算法题都配有完整图解！仅此一家！

## 题解是围绕什么编写的

掌握！所有的题解都以掌握二字为前提。不会追求过多的奇淫技巧，毕竟我们不是专门研究算法的人。我见过太多算法初学者，一个题解看不懂，转头又去看第二个题解，第二个看不懂，又去看第三个，直到最后放弃掉。浪费了时间，题目还是不会做，这图什么呢？所以本教程所有的题解都是以掌握为目标，尽量把每一道题的思路都讲的明明白白的。

## 题解是否严谨

绝对严谨，所有的题解都在leetcode上进行过测试运行。

## 没学过 java、go 是否可以学习

当然可以。我期望大家更多的是去关注算法的本身，而不是语言层面的东西。所以本教程，其实各语言都会使用一些，并不局限于 java、go。但是，我绝对不会使用任何语法特性！我希望你不要被语言所束缚！

## 是否可以按照本教程顺序来刷题

当然可以。一般刷题我们有两种策略，一种就是刷 leetcode 前一百道题目，另一种就是根据分类刷题。刷 leetcode 前一百道题，是因为这些题目都是经典题目。而根据分类，更适合算法小白和初中阶段读者。所以我在这里选择了根据分类来汇编，这样我们还可以在做一些题目的时候，与前面同类型的题目进行比较。

## 这些题目刷完能达到什么效果

刷完再说！

## 你需要做什么

开干！奥利给！

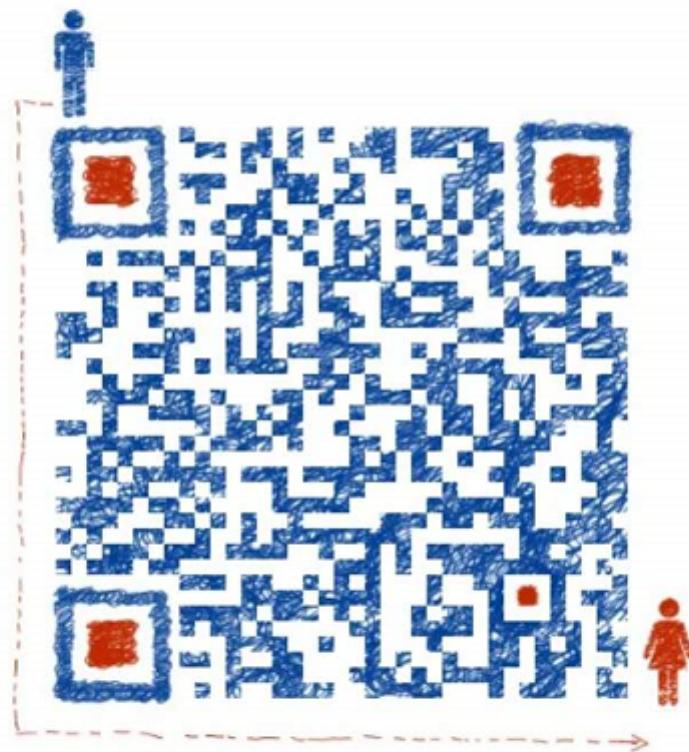
**看完题目还是不懂怎么办？**

下方扫码，加我微信，我拉你到刷题群。和大家一起交流学习！（备注：进群）



小浩

中国大陆



扫一扫上面的二维码图案，加我微信

## 数组系列

**两个数组的交集(350)**

## 01、题目分析

我们先来看一道题目：

### 第350题：两个数组的交集

给定两个数组，编写一个函数来计算它们的交集。

#### 示例 1:

```
1 | 输入: nums1 = [1,2,2,1], nums2 = [2,2]
2 |
3 | 输出: [2,2]
```

#### 示例 2:

```
1 | 输入: nums1 = [4,9,5], nums2 = [9,4,9,8,4]
2 |
3 | 输出: [4,9]
```

说明：

- 输出结果中每个元素出现的次数，应与元素在两个数组中出现的次数一致。
- 我们可以不考虑输出结果的顺序。

进阶：

- 如果给定的数组已经排好序呢？将如何优化你的算法呢？

思路：设定两个为0的指针，比较两个指针的元素是否相等。如果指针的元素相等，我们将两个指针一起向后移动，并且将相等的元素放入空白数组。

## 02、题解分析

首先拿到这道题，我们基本马上可以想到，此题可以看成是一道传统的映射题（map映射），为什么可以这样看呢，因为我们需要找出两个数组的交集元素，同时应与两个数组中出现的次数一致。这样就导致了我们需要知道每个值出现的次数，所以映射关系就成了<元素,出现次数>。剩下的就是顺利成章的解题。

由于该种解法过于简单，我们不做进一步分析，直接给出题解：

```

1 //GO
2 func intersect(nums1 []int, nums2 []int) []int {
3     m0 := map[int]int{}
4     for _, v := range nums1 {
5         //遍历nums1, 初始化map
6         m0[v] += 1
7     }
8     k := 0
9     for _, v := range nums2 {
10        //如果元素相同, 将其存入nums2中, 并将出现次数减1
11        if m0[v] > 0 {
12            m0[v] -=1
13            nums2[k] = v
14            k++
15        }
16    }
17    return nums2[0:k]
18 }
```

这个方法比较简单，相信大家都能看的懂！

### 03、题目进阶

题目在进阶问题中问道：如果给定的数组已经排好序呢？你将如何优化你的算法？我们分析一下，假如两个数组都是有序的，分别为：arr1 = [1,2,3,4,4,13]，arr2 = [1,2,3,9,10]

1	3	4	4	13
1	4	9	10	

浩仔讲算法

对于两个已经排序好数组的题，我们可以很容易想到使用双指针的解法~

解题步骤如下：

<1> 设定两个为0的指针，**比较两个指针的元素是否相等**。如果指针的元素相等，我们将两个指针一起向后移动，并且将相等的元素放入空白数组。下图中我们的指针分别指向第一个元素，判断元素相等之后，将相同元素放到空白的数组。

1				
1	3	4	4	13
1	4	9	10	

浩仔讲算法

<2> 如果两个指针的元素不相等，**我们将小的一个指针后移**。图中我们指针移到下一个元素，判断不相等之后，将元素小的指针向后移动，继续进行判断。

1				
1	3	4	4	13
1	4	9	10	

浩仔讲算法

<3> 反复以上步骤。

1	4			
1	3	4	4	13
1	4	9	10	

浩仔讲算法

<4> 直到任意一个数组终止。

1	4			
1	3	4	4	13
1	4	9	10	

浩仔讲算法

## 04、题目解答

根据分析，我们很容易得到下面的题解：

```

1 //GO
2 func intersect(nums1 []int, nums2 []int) []int {
3     i, j, k := 0, 0, 0
4     sort.Ints(nums1)
5     sort.Ints(nums2)

```

```
6  for i < len(nums1) && j < len(nums2) {
7      if nums1[i] > nums2[j] {
8          j++
9      } else if nums1[i] < nums2[j] {
10         i++
11     } else {
12         nums1[k] = nums1[i]
13         i++
14         j++
15         k++
16     }
17 }
18 return nums1[:k]
19 }
```

提示：解答中我们并没有创建空白数组，因为遍历后的数组其实就没用了。我们可以**将相等的元素放入用过的数组中，就为我们节省下了空间。**

## 最长公共前缀(14)

### 01、题目分析

首先还是看下题目：

#### 题目14: 最长公共前缀

编写一个函数来查找字符串数组中的最长公共前缀。如果不存在公共前缀，则返回""

#### 示例1:

```
1 输入: ["flower", "flow", "flight"]
2 输出: "fl"
```

#### 示例 2:

```
1 输入: ["dog", "racecar", "car"]
2 输出: ""
```

解释：

- 输入不存在公共前缀。

说明：

- 所有输入只包含小写字母 a-z

## 02、题解分析

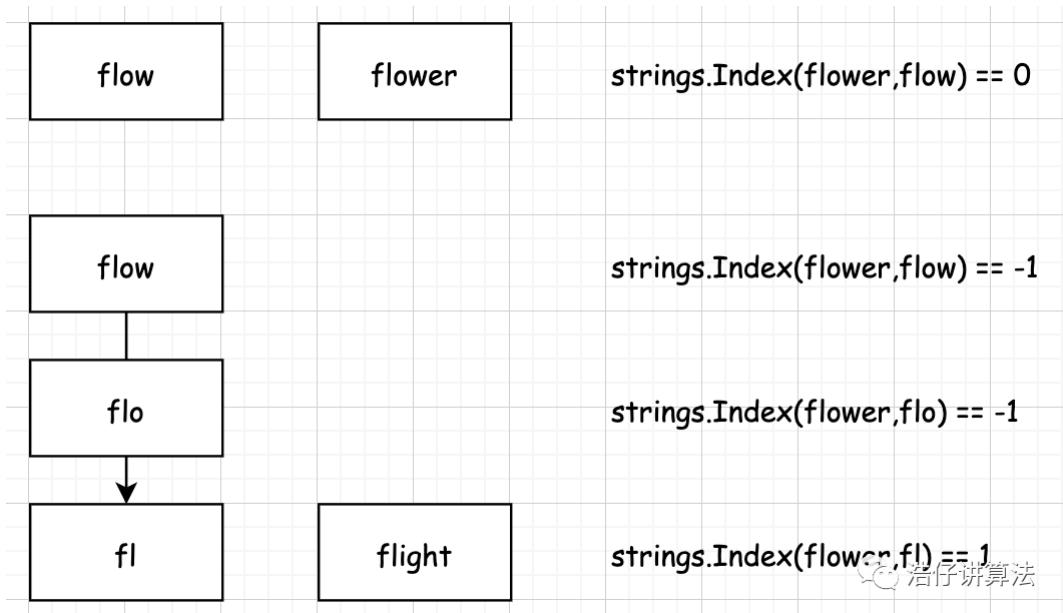
我们要想寻找最长公共前缀，那么首先这个前缀是公共的，我们可以从任意一个元素中找到它。假定我们现在就从一个数组中寻找最长公共前缀，那么首先，我们可以将第一个元素设置为基准元素 $x_0$ 。假如数组为["flow", "flower", "flight"]，flow就是我们的基准元素 $x_0$ 。

然后我们只需要依次将基准元素和后面的元素进行比较（假定后面的元素依次为 $x_1, x_2, x_3, \dots$ ），不断更新基准元素，直到基准元素和所有元素都满足最长公共前缀的条件，就可以得到最长公共前缀。

具体比对过程如下：

- 如果 $\text{strings.Index}(x_1, x) == 0$ ，则直接跳过（因为此时 $x$ 就是 $x_1$ 的最长公共前缀），对比下一个元素。（如flower和flow进行比较）
- 如果 $\text{strings.Index}(x_1, x) != 0$ ，则截取掉基准元素 $x$ 的最后一个元素，再次和 $x_1$ 进行比较，直至满足 $\text{string.Index}(x_1, x) == 0$ ，此时截取后的 $x$ 为 $x$ 和 $x_1$ 的最长公共前缀。（如flight和flow进行比较，依次截取出flow-flo-fl，直到fl被截取出，此时fl为flight和flow的最长公共前缀）

具体过程如下图所示：



我们需要注意的是，在处理基准元素的过程中，如果基准元素和任一个元素无法匹配，则说明不存在最长公共元素。

最后，我们记得处理一下临界条件。如果给定数组是空，也说明没有最长公共元素。

然后我们就可以开始写我们的代码了。

## 03、代码分析

根据分析，我们很容易得到下面的题解：

```
1 //GO
2 func longestCommonPrefix(strs []string) string {
3     if len(strs) < 1 {
4         return ""
5     }
6     prefix := strs[0]
7     for _,k := range strs {
8         for strings.Index(k,prefix) != 0 {
9             if len(prefix) == 0 {
10                 return ""
11             }
12             prefix = prefix[:len(prefix) - 1]
13         }
14     }
15     return prefix
16 }
```

运行结果：

执行结果： 通过 显示详情 >

执行用时： 0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗： 2.4 MB，在所有 Go 提交中击败了 61.56% 的用户

炫耀一下：



浩仔讲算法

当然，我们也可以用分治法或者其他方法来解答这道题目。你可以自己尝试尝试哈。我们下期见！

## 买卖股票的最佳时机(122)

# 01、题目分析

在leetcode上，股票相关的题目有8道之多：

#	题名	题解	通过率	难度
901	股票价格跨度	26	37.1%	中等
✓ 121	买卖股票的最佳时机	241	51.9%	简单
✓ 122	买卖股票的最佳时机 II	211	57.0%	简单
✓ 123	买卖股票的最佳时机 III	52	40.1%	困难
✓ 309	最佳买卖股票时机含冷冻期	47	50.7%	中等
✓ 188	买卖股票的最佳时机 IV	32	28.5%	困难
✓ 714	买卖股票的最佳时机含手续费	25	59.7%	中等
502	IPO	14	35.5%	困难

而且这一类型的题，面试时出现的频率非常的高。稍微改一改条件，就让我们防不胜防。那我们如何攻克这一类题型呢？我们从最简单的一道开始看起：

## 第122题：买卖股票的最佳时机 II

给定一个数组，它的第  $i$  个元素是一支给定股票第  $i$  天的价格。

如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。注意你不能在买入股票前卖出股票。

### 示例 1:

1 输入： [7,1,5,3,6,4]  
2 输出： 7

解释：在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 =  $5 - 1 = 4$ 。随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 =  $6 - 3 = 3$ 。

### 示例 2:

1 输入： [1,2,3,4,5]  
2 输出： 4

解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 =  $5 - 1 = 4$ 。注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

### 示例 3:

1 | 输入: [7, 6, 4, 3, 1]  
2 | 输出: 0

| 解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

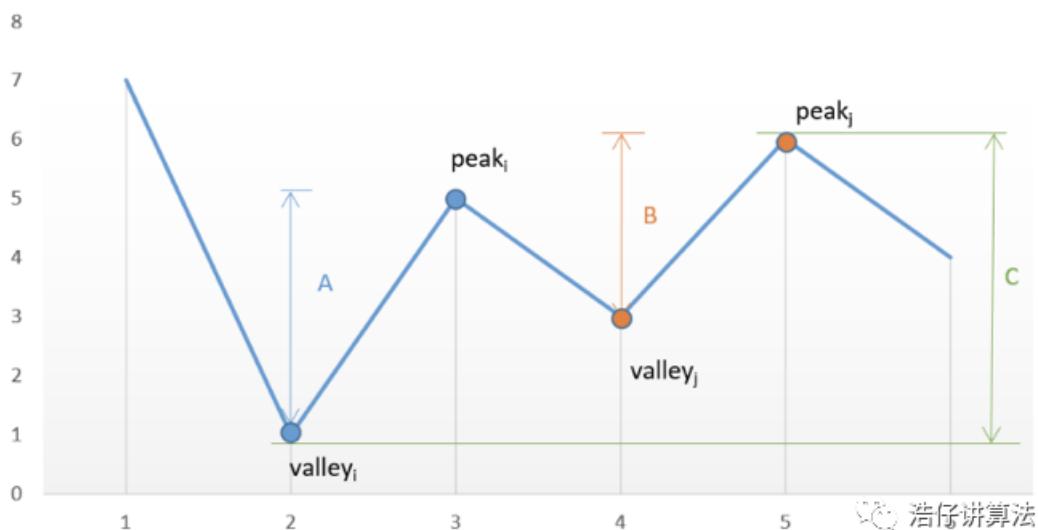
题目分析: 首先我们看一下题目中给出的两个条件:

- 1、**不能参与多笔交易**。换句话讲, 我们只能在手上没有股票的时候买入, 也就是**必须在再次购买前出售掉之前的股票**。像我们平时买股票时的追涨杀跌是不可以的。
- 2、**尽可能地多进行交易**。这个非常好理解。像是黄金, 一年基本上都有2-3次涨跌。我们只要把握住机会, 在每一次涨跌的时候, 低价买入高价卖出, 就可以使利益达到最大化。这个条件也是相当重要的, 如果我们把这里变成, 最多完成两笔交易, 就变成另一道题。

现在题目搞清楚了, 我们来思考一下。

## 02、题解分析

假设给定的数组为: [7, 1, 5, 3, 6, 4] 我们将其绘制成折线图, 大概是下面这样:



如我们上面分析, 我们要在满足1和2的条件下获取最大利益, 其实就是尽可能多的低价买入高价卖出。而每一次上升波段, 其实就是一次低价买入高价卖出。而我们没有限制交易次数, 也就是我们需要求出所有的上升波段的和。上图里就是A+B, 也就是  $(5-1) + (6-3) = 7$ , 就是我们能获取到的最大利益。

其实也就是尽可能多的低价买入, 高价卖出啦。

## 03、代码分析

根据以上分析，我们很容易得到下面的题解：

```
1 //GO
2 func maxProfit(prices []int) int {
3     if len(prices) < 2 {
4         return 0
5     }
6     dp := make([][2]int, len(prices))
7     dp[0][0] = 0
8     dp[0][1] = -prices[0]
9     for i := 1; i < len(prices); i++ {
10        dp[i][0] = max(dp[i-1][0], dp[i-1][1]+prices[i])
11        dp[i][1] = max(dp[i-1][0]-prices[i], dp[i-1][1])
12    }
13    return dp[len(prices)-1][0]
14 }
15
16 func max(a, b int) int {
17     if a > b {
18         return a
19     }
20     return b
21 }
```

## 04、题目扩展

图解的方式其实在各种算法题中，屡见不鲜。而我们通过图解的方式，也可以抽丝剥茧一样，一层一层剥掉算法题目的外壳，寻找到最直观的解题思路，直捣黄....咳咳，直奔核心。那我们又如何用图解的观察方式，来对本系列的其他题目寻找到一种通用解法，来规避题目中的陷阱呢？浩仔讲算法，我们下期再见喽！

## 旋转数组(189)

### 01、题目分析

### 题目189: 旋转数组

给定一个数组，将数组中的元素向右移动  $k$  个位置，其中  $k$  是非负数。

#### 示例 1:

- |   |                               |
|---|-------------------------------|
| 1 | 输入: [1,2,3,4,5,6,7] 和 $k = 3$ |
| 2 | 输出: [5,6,7,1,2,3,4]           |

解释:

- 向右旋转 1 步: [7,1,2,3,4,5,6]
- 向右旋转 2 步: [6,7,1,2,3,4,5]
- 向右旋转 3 步: [5,6,7,1,2,3,4]

#### 示例 2:

- |   |                              |
|---|------------------------------|
| 1 | 输入: [-1,-100,3,99] 和 $k = 2$ |
| 2 | 输出: [3,99,-1,-100]           |

解释:

- 向右旋转 1 步: [99,-1,-100,3]
- 向右旋转 2 步: [3,99,-1,-100]

说明:

尽可能想出更多的解决方案，至少有三种不同的方法可以解决这个问题。

**要求使用空间复杂度为  $O(1)$  的原地 算法。**

这道题如果不要求原地翻转的话，其实相当简单。但是原地翻转的方法却并不容易想到，我们直接看题解。

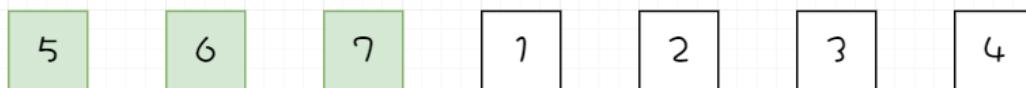
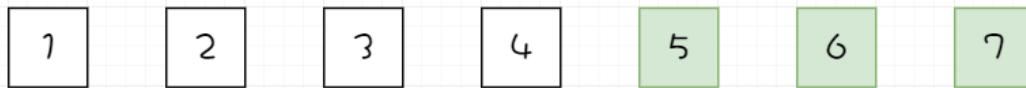
## 02、题目图解

这个方法基于这个事实：若我们需要将数组中的元素向右移动  $k$  个位置，那么  $k \% l$  ( $l$  为数组长度) 的尾部元素会被移动到头部，剩下的元素会被向后移动。

假设我们现在数组为 [1,2,3,4,5,6,7]， $l = 7$  且  $k = 3$ 。

如下图可以看到 5, 6, 7 被移动到数组头部。

$k = 3 \quad l = 7$

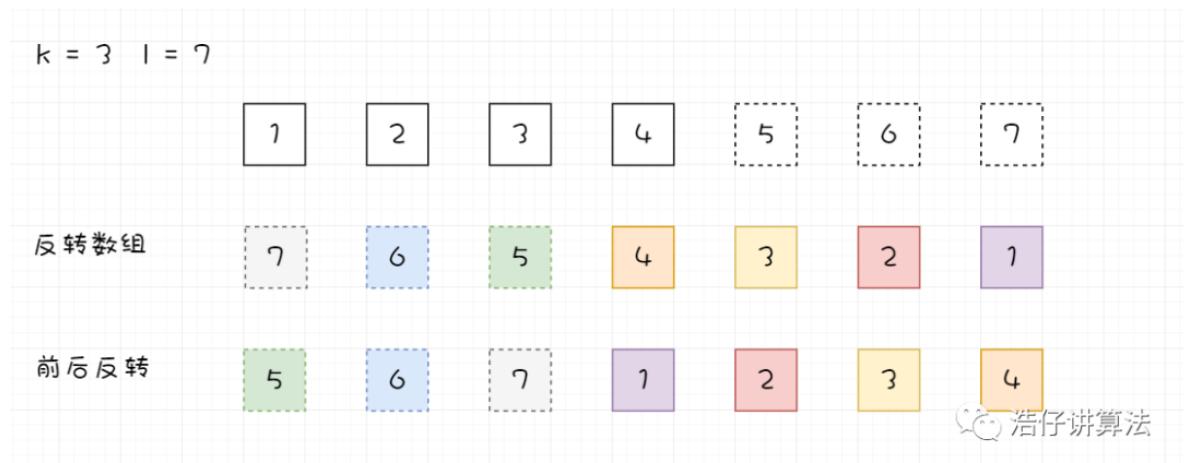


$k \% l = 3$  个元素被移动到数组头部



通过观察我们可以得到，我们要得到最终的结果。我们只需要将所有元素反转，然后反转前  $k$  个元素，再反转后面  $l-k$  个元素，就能得到想要的结果。

如下图：



### 03、题目解答

根据分析，我们可以得到下面的题解：

```
1 //GO
2 func rotate(nums []int, k int) {
3     reverse(nums)
4     reverse(nums[:k%len(nums)])
5     reverse(nums[k%len(nums):])
6 }
7
8 func reverse(arr []int) {
9     for i := 0; i < len(arr)/2; i++ {
10         arr[i], arr[len(arr)-i-1] = arr[len(arr)-i-1], arr[i]
11     }
12 }
```

## 原地删除(27)

### 01、题目分析

#### 题目27：移除元素

给定一个数组 `nums` 和一个值 `val`，你需要原地移除所有数值等于 `val` 的元素，返回移除后数组的新长度。

不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。元素的顺序可以改变。你不需要考虑数组中超出新长度后面的元素。

#### 示例 1：

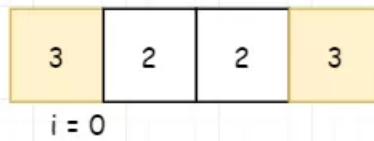
- 1 给定 `nums = [3,2,2,3]`，`val = 3`，
- 2 函数应该返回新的长度 2，并且 `nums` 中的前两个元素均为 2。
- 3 你不需要考虑数组中超出新长度后面的元素。

这道题比较简单哦，只要把握好“原地删除”这个关键字，就可以顺利求解啦！

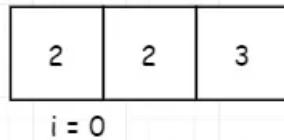
具体过程如下图所示：

nums = [3, 2, 2, 3] val = 3

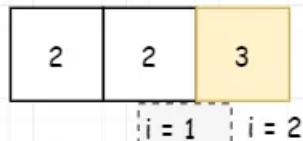
1、准备遍历数组，当当前值等于目标值时，移除目标值



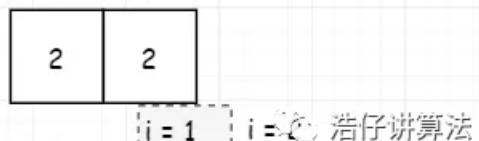
2、当当前值不等于目标值时。  
i++



3、重复第1、2步



4、遍历完整个数组



根据分析，我们可以得到下面的题解：

```
1 //GO
2 func removeElement(nums []int, val int) int {
3     for i := 0; i < len(nums); {
4         if nums[i] == val {
5             nums = append(nums[:i], nums[i+1:]...)
6         } else{
7             i++
8         }
9     }
10    return nums
11 }
```

和这道题类似的还有LeetCode 26题，大家可以尝试自己先做一做，然后再看答案哦。

## 02、类似题目分析

### 题目26：删除排序数组中的重复项

给定一个排序数组，你需要在原地删除重复出现的元素，使得每个元素只出现一次。

返回移除后数组的新长度。不要使用额外的数组空间，你必须在原地修改输入数组并在使用 O(1) 额外空间的条件下完成。

### 示例 1:

- 1 给定数组 `nums = [1,1,2]`,
- 2 函数应该返回新的长度 2，并且原数组 `nums` 的前两个元素被修改为 1, 2。
- 3 你不需要考虑数组中超出新长度后面的元素。

### 示例 2:

- 1 给定 `nums = [0,0,1,1,1,2,2,3,3,4]`,
- 2 函数应该返回新的长度 5，并且原数组 `nums` 的前五个元素被修改为 0, 1, 2, 3, 4。
- 3 你不需要考虑数组中超出新长度后面的元素。

这道题的重点是原地两个字，也就是要求必须在 O(1) 的空间下完成。并且题中已经告知了数组为有序数组，这样重复的元素一定是连在一起的，我们只需要一个一个移除重复的元素即可，具体方案方案怎么做，我们看看下面就会明白了。

1、遍历元素，定义当前元素为  $i$ ，  
下一个元素为  $i+1$

0	1	1	2	2	3
	$i$		$i+1$		

2、当发现第  $i$  个元素和  $i+1$  个元素相等时，  
移除第  $i+1$  个元素

0	1	1	2	2	3
	$i$				

3、当第  $i$  个元素和  $i+1$  个元素不等时，  
 $i++$

0	1	2	2	3
	$i$		$i+1$	

4、重复第 2、3 步骤

0	1	2	2	3
	$i$		$i+1$	

5、当  $i+1$  指向最后一个元素，跳出  
循环

0	1	2	3
	$i$		$i+1$

浩仔讲算法

根据分析，我们可以得到下面的题解：

```
1 //GO
2 func removeDuplicates(nums []int) int {
3     for i := 0; i+1 < len(nums);{
4         if nums[i] == nums[i+1]{
5             nums = append(nums[:i],nums[i+1:]...)
6         }else{
7             i++
8         }
9     }
10    return len(nums)
11 }
```

好啦，关于数组原地操作的两道题就讲到这里啦，如果大家有兴趣的话，可以参考做一下 LeetCode 283题（移动O），也是一样的做法哦！

## 加一(66)

看到这个标题，大家肯定会觉得，不就是“加1”嘛，这么简单的问题我可以！但是就是这么简单的“加1”可是面试的高频题哦，所以我们就一起来看看吧。按照往例，我们还是从一道LeetCode题开始吧。

### 01、题目分析

#### 第66题：加一

给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

最高位数字存放在数组的首位，数组中每个元素只存储单个数字。你可以假设除了整数 0 之外，这个整数不会以零开头。

#### 示例 1:

- 1 输入: [1,2,3]
- 2 输出: [1,2,4]
- 3 解释: 输入数组表示数字 123。

#### 示例 2:

- 1 输入: [4,3,2,1]
- 2 输出: [4,3,2,2]
- 3 解释: 输入数组表示数字 4321。

#### 题目分析:

根据题目，我们需要加一！没错，加一很重要。因为它只是加一，所以我们会考虑到两种情况：

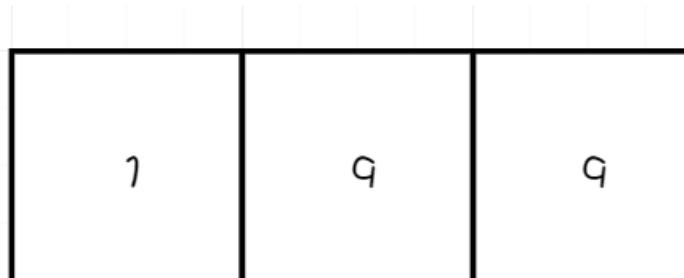
- 普通情况，除9之外的数字加1。
- 特殊情况，9加1。 (因为9加1需要进位)

所以我们只需要模拟这两种运算，就可以顺利进行求解！

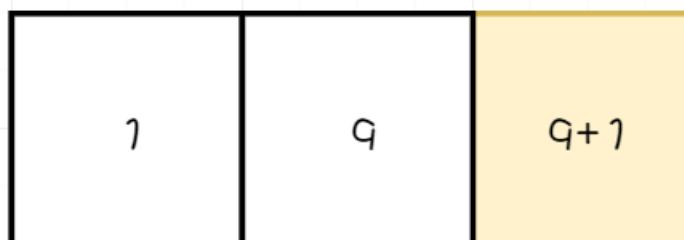
## 02、题目图解

假设我们的数为 [1,9,9]

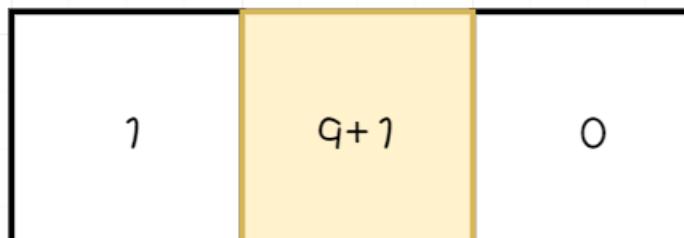
大概是下面这样： (这个图解...真的有点太简单了...)



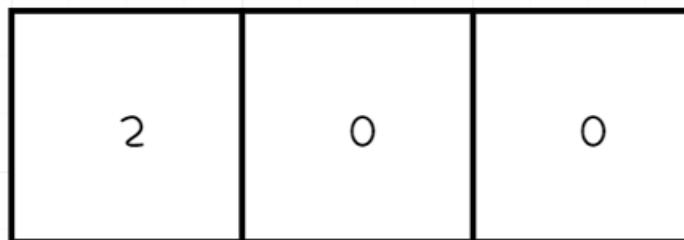
加1当然从个位加：



进位

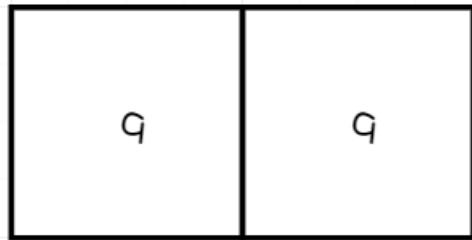


继续进位

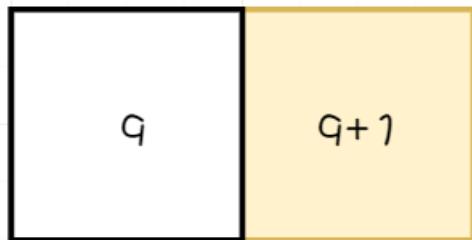


© 浩仔讲算法

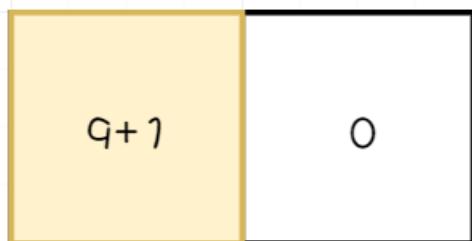
当然，这里我们需要考虑一种特殊情况，就是类似99，或者999，我们需要进行拼接数组。具体如下图：



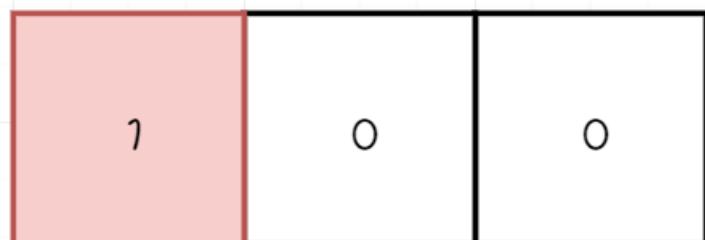
加1当然从个位加:



进位



特殊情况：继续进位，需要拼接数组



浩仔讲算法

通过以上分析，我们最后只需要将其转换成代码即可！这样看来，“加1”是不是也不像想象中的那么简单？

### 03、GO语言示例

根据以上分析，我们可以得到下面的题解：

```
1 func plusOne(digits []int) []int {
2     var result []int
3     addon := 0
4     for i := len(digits) - 1; i >= 0; i-- {
5         digits[i] += addon
6         addon = 0
7         if i == len(digits) - 1 {
8             digits[i]++
9         }
10        if digits[i] == 10 {
11            addon = 1
12            digits[i] = digits[i] % 10
13        }
14    }
15    if addon == 1 {
16        result = make([]int, 1)
17        result[0] = 1
18        result = append(result, digits...)
19    } else {
20        result = digits
21    }
22    return result
23 }
```

**提示:**

1 | `append(a,b...)` 的含义是: 将b切片中的元素追加到a中。

## 两数之和(1)

### 01、题目分析

#### 第1题：两数之和

给定一个整数数组 `nums` 和一个目标值 `target`, 请你在该数组中找出和为目标值的那 两个 整数, 并返回他们的数组下标。

你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

**示例:**

```
1 | 给定 nums = [2, 7, 11, 15], target = 9
2 | 因为 nums[0] + nums[1] = 2 + 7 = 9
3 | 所以返回 [0, 1]
```

题目分析

首先我们拿到题目一看，马上可以想到暴力题解。我们只需要“遍历每个元素  $x$ ，并查找是否存在一个值与  $\text{target} - x$  相等的目标元素。”

由于该种解题思路过于简单，直接上代码（如果有问题请留言..）：

```
1 func twoSum(nums []int, target int) []int {
2     for i, v := range nums {
3         for k := i + 1; k < len(nums); k++ {
4             if target-v == nums[k] {
5                 return []int{i, k}
6             }
7         }
8     }
9     return []int{}
10 }
```

执行结果：

执行结果：通过 [显示详情 >](#)

执行用时：**52 ms**，在所有 Go 提交中击败了 **16.31%** 的用户

内存消耗：**2.9 MB**，在所有 Go 提交中击败了 **85.60%** 的用户

运行成功，但是该种解题方式的时间复杂度过高，达到了  $O(n^2)$ 。为了对运行时间复杂度进行优化，我们需要一种更有效的方法来检查数组中是否存在目标元素。我们可以想到用哈希表的方式，通过以空间换取时间的方式来进行。

## 02、题目图解

假设  $\text{nums} = [2, 7, 11, 15]$ ,  $\text{target} = 9$

<1> 首先，我们还是先遍历数组  $\text{nums}$ ,  $i$  为当前下标。我们需要将每一个遍历的值放入  $\text{map}$  中作为 key。

2	7	11	15
i			

key	value
2	0
7	浩仔讲算法

<2> 同时，对每个值都判断 map 中是否存在 `target-nums[i]` 的 key 值。在这里就是  $9-7=2$ 。我们可以看到 2 在 map 中已经存在。

2	7	11	15
i			

key	value
2	0

浩仔讲算法

<3> 所以，2 和 7 所在的 key 对应的 value，也就是 [0,1]。就是我们要找的两个数组下标。

2	7	11	15
i			

key	value
2	0
7	1

浩仔讲算法

### 03、Go语言示例

根据以上分析，我们可以得到下面的题解：

```
1 func twoSum(nums []int, target int) []int {
2     result := []int{}
3     m := make(map[int]int)
4     for i,k := range nums {
5         if value,exist := m[target-k];exist {
6             result = append(result,value)
7             result = append(result,i)
8         }
9         m[k] = i
10    }
11    return result
12 }
```

执行结果：

执行结果： 通过 [显示详情 >](#)

执行用时：**4 ms**，在所有 Go 提交中击败了 **96.78%** 的用户

内存消耗：**3.8 MB**，在所有 Go 提交中击败了 **13.49%** 的用户

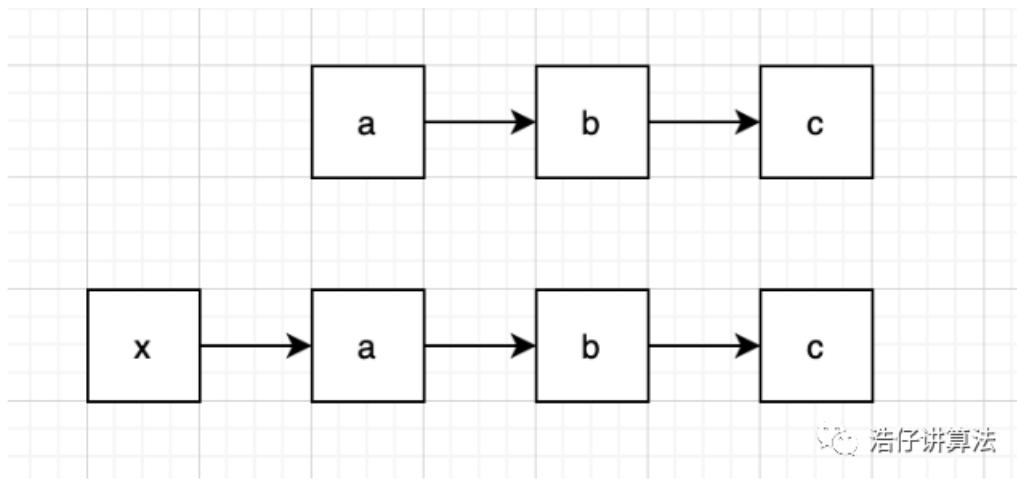
## 链表系列

### 删除链表倒数第N个节点(19)

#### 01、哨兵节点

在链表的题目中，十道有九道会用到**哨兵节点**，所以我们先讲一下什么是哨兵节点。

哨兵节点，其实就是一个附加在原链表最前面用来简化边界条件的附加节点，它的值域不存储任何东西，只是为了操作方便而引入。比如原链表为  $a \rightarrow b \rightarrow c$ ，则加了哨兵节点的链表即为  $x \rightarrow a \rightarrow b \rightarrow c$ ，如下图：



那我们为什么需要引入哨兵节点呢？举个例子，比如我们要删除某链表的第一个元素，**常见的删除链表的操作是找到要删元素的前一个元素**，假如我们记为 pre。我们通过：

`pre.Next = pre.Next.Next`

来进行删除链表的操作。但是此时若是删除第一个元素的话，你就很难进行了，因为按道理来讲，此时第一个元素的前一个元素就是 nil（空的），如果使用 pre 就会报错。那如果此时你设置了哨兵节点的话，此时的 pre 就是哨兵节点了。这样对于链表中的任何一个元素，你要删除都可以通过 `pre.Next = pre.Next.Next` 的方式来进行，这就是哨兵节点的作用。

下面我们看一道题目，看一下哨兵节点的应用

## 02、题目讲解

### 第19题：删除链表倒数第N个节点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

- 1 给定一个链表：1->2->3->4->5，和  $n = 2$ .
- 2 当删除了倒数第二个节点后，链表变为 1->2->3->5.

说明：

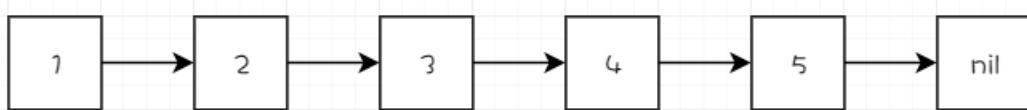
- 给定的 n 保证是有效的。

进阶：

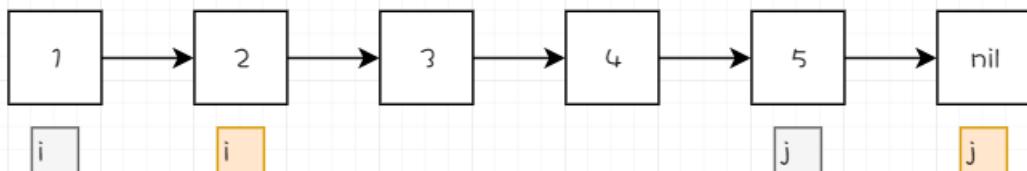
- 你能尝试使用一趟扫描实现吗？

思路分析：

首先我们思考，让我们删除倒数第N个元素，那我们只要找到倒数第N个元素就可以了，那怎么找呢？我们只需要设置两个指针变量，中间间隔N-1元素。当后面的指针遍历完所有元素指向nil时，前面的指针就指向了我们要删除的元素。如下图所示：

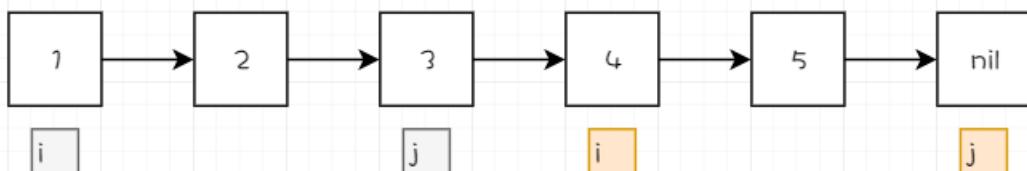


假如我们要删除倒数第4个元素：



此时2就是我们要删除的元素

假如我们要删除倒数第2个元素：



此时4就是我们要删除的元素

浩仔讲算法

接下来，我们只要同时定位到要删除的元素的前1个元素，通过前面讲过的删除操作，就可以很顺利的完成这道题目啦。

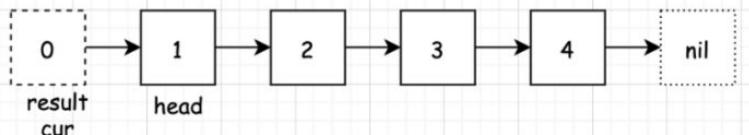
### 03、解题过程

现在我们来完整捋一遍解题过程：

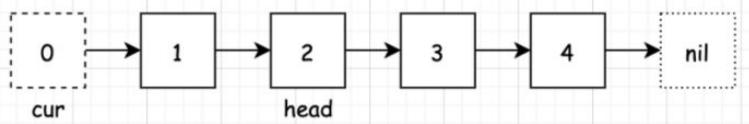
1. 首先我们定义好哨兵节点result，指向哨兵节点的目标元素指针cur，以及目标指针cur的前一个指针pre，此时pre指向nil。
2. 接下来我们开始遍历整个链表。
3. 当head移动到距离目标元素cur的距离为N-1时，同时开始移动cur。
4. 当链表遍历完之后，此时head指向nil，这时的cur就是我们要找的待删除的目标元素。
5. 最后我们通过pre.Next = pre.Next.Next完成删除操作，就完成了整个解题过程。

下面是解题过程图，可以看得更清楚哦。

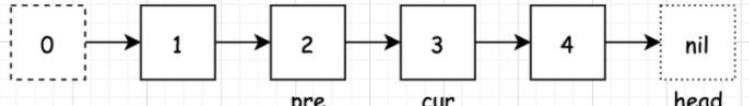
```
result.Next = head  
pre = nil
```



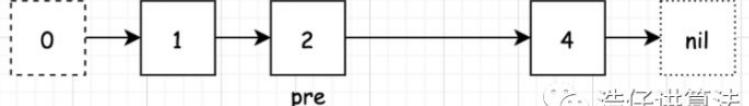
当head遍历至第n个元素时，准备移动cur元素



当head遍历至nil后，cur即为要删除的节点



```
pre.Next = pre.Next.Next
```



浩仔讲算法

## 04、题目解答

根据以上分析，我们可以得到下面的题解：

```
1 func removeNthFromEnd(head *ListNode, n int) *ListNode {  
2     result := &ListNode{}  
3     result.Next = head  
4     var pre *ListNode  
5     cur := result  
6     i := 1  
7     for head != nil {  
8         if i >= n {  
9             pre = cur  
10            cur = cur.Next  
11        }  
12        head = head.Next  
13        i++  
14    }  
15    pre.Next = pre.Next.Next  
16    return result.Next  
17 }
```

## 合并两个有序链表(21)

## 01、题目分析

### 第21题：合并两个有序链表

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

1	输入: 1->2->4, 1->3->4
2	输出: 1->1->2->3->4->4

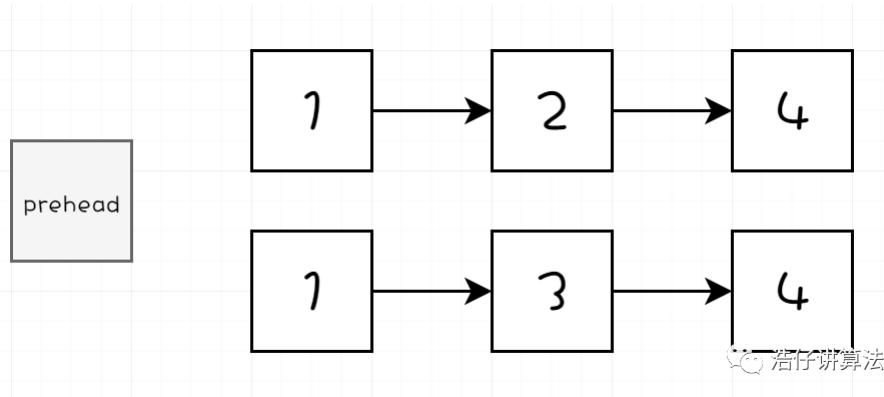
首先我们拿到题目乍眼一看，类似这种**链表的合并问题**。基本上马上可以想到需要设置一个**哨兵节点**，这可以在最后让我们比较容易地返回**合并后的链表**。（不懂哨兵节点的同学，可以先移驾到 [06.删除链表倒数第N个节点\(19\)](#) 进行学习）

假设我们的链表分别为：

$l_1 = [1, 2, 4]$

$l_2 = [1, 3, 4]$

同时我们设定一个 "prehead" 的哨兵节点，大概是下面这样：

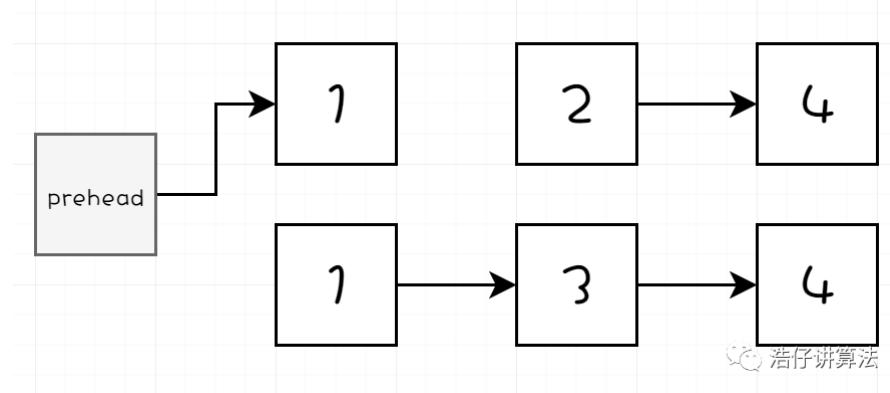


## 02、题目图解

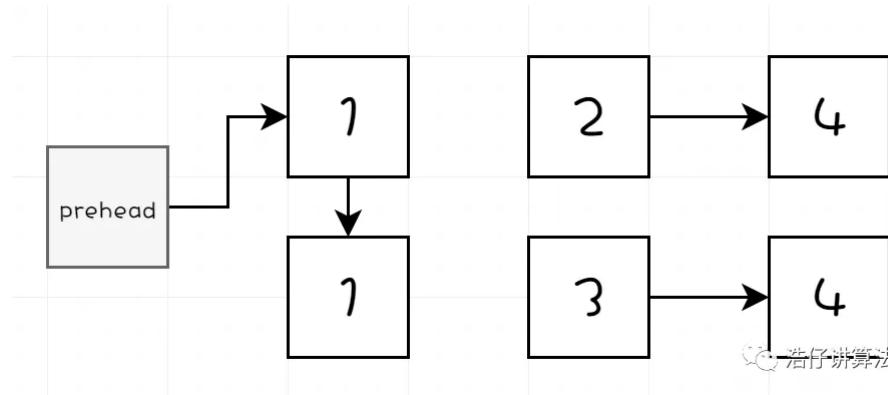
如上图所示，首先我们维护一个 **prehead** 的哨兵节点。我们其实只需要调整它的 **next 指针**。让它总是指向  $l_1$  或者  $l_2$  中较小的一个，直到  $l_1$  或者  $l_2$  任一指向 **null**。这样到了最后，如果  $l_1$  还是  $l_2$  中任意一方还有余下元素没有用到，那余下的这些元素一定大于 **prehead** 已经合并完的链表（因为是有序链表）。

我们只需要将这些元素全部追加到 prehead 合并完的链表后，最终就得到了我们需要的链表。大概流程如下：

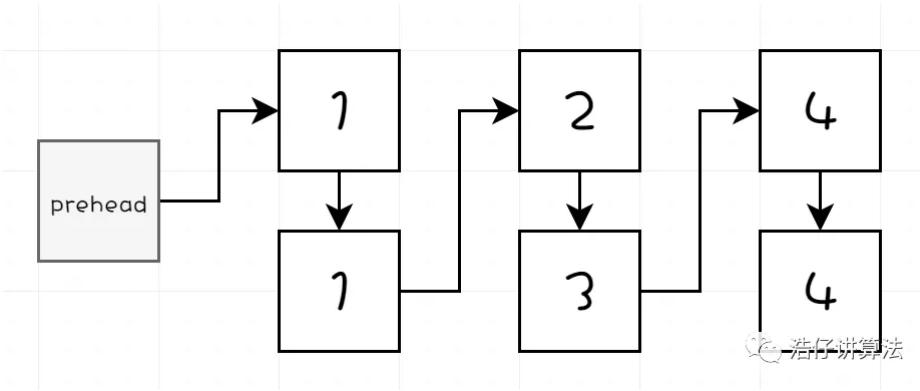
1. 首先我们将 prehead 指向 l1 或者 l2 中比较小的一个。如果相等，则任意一个都可以。此时的 l1 为 [2,4], l2 为 [1,3,4]



2. 我们继续上面的步骤。将 prehead 的链表指向 l1 和 l2 中较小的一个。现在这里就是指向1。



3. 反复上图步骤。



4. 现在 prehead.Next 就是我们需要的链表。

### 03、Go语言示例

根据以上分析，我们可以得到下面的题解：

```

1 func mergeTwoLists(l1 *ListNode, l2 *ListNode) *ListNode {
2     prehead := &ListNode{}
3     result := prehead
4     for l1 != nil && l2 != nil {
5         if l1.Val < l2.Val {
6             prehead.Next = l1
7             l1 = l1.Next
8         } else {
9             prehead.Next = l2
10            l2 = l2.Next
11        }
12        prehead = prehead.Next
13    }
14    if l1 != nil {
15        prehead.Next = l1
16    }
17    if l2 != nil {
18        prehead.Next = l2
19    }
20    return result.Next
21 }

```

## 环形链表(21)

今天为大家带来，**链表检测成环**的经典题目。如果你觉得你会了，请你不妨耐心些认真看下去，我相信会有一些不一样的收获！还是先从一道题目开始哟，准备好了吗？Let's go！

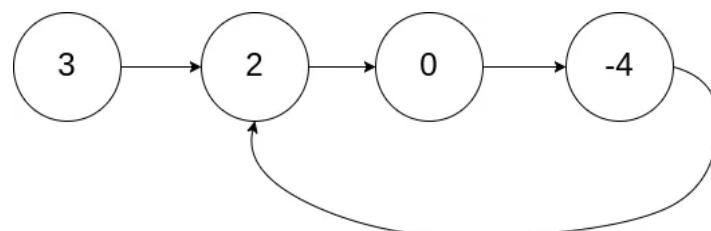
### 01、题目分析

#### 第141题：环形链表

给定一个链表，判断链表中是否有环。为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。如果 pos 是 -1，则在该链表中没有环。

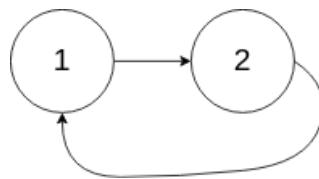
#### 示例 1：

- 1 输入: head = [3,2,0,-4], pos = 1
- 2 输出: true
- 3 解释: 链表中有一个环, 其尾部连接到第二个节点。



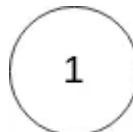
#### 示例 2：

```
1 | 输入: head = [1,2], pos = 0  
2 | 输出: true  
3 | 解释: 链表中有一个环, 其尾部连接到第一个节点。
```



### 示例 3:

```
1 | 输入: head = [1], pos = -1  
2 | 输出: false  
3 | 解释: 链表中没有环。
```



题目可能你会觉得过于简单！但是不妨耐心看完！

则一定会有收获！

## 02、题目分析

### 题解一：哈希表判定

思路：通过hash表来检测节点之前是否被访问过，来判断链表是否成环。这是最容易想到的一种题解了。过于简单，直接上代码：

```
1 | func hasCycle(head *ListNode) bool {  
2 |     m := make(map[*ListNode]int)  
3 |     for head != nil {  
4 |         if _,exist := m[head];exist {  
5 |             return true  
6 |         }  
7 |         m[head] = 1  
8 |         head = head.Next  
9 |     }  
10 |    return false  
11 | }
```

### 题解二：JS特殊解法

相信对于 JS 中的 JSON.stringify() 方法大家都用过，主要用于将 JS 对象 转换为 JSON 字符串。基本使用如下：

```
1 | var car = {  
2 |   name: '小喵',  
3 |   age: 20,  
4 | }  
5 | var str = JSON.stringify(car);  
6 | console.log(str)  
7 | //=> {"name": "小喵", "age": 20}
```

大家想一下，如果是自己实现这样的一个函数，我们需要处理什么样的特殊情况？对，就是**循环引用**。因为对于循环引用，我们很难通过 JSON 的结构将其进行展示！比如下面：

```
1 | var a = {}  
2 | var b = {  
3 |   a: a  
4 | }  
5 | a.b = b  
6 | console.log(JSON.stringify(a))  
7 | //=> TypeError: Converting circular structure to JSON
```

那我们思考，对于环形链表，是不是就是一个循环结构呢？当然是！因为只要是环形链表，它一定存在类似以下代码：

```
a.Next = b  
b.Next = a
```

所以我们可以利用 JSON.stringify() 的特性进行求解：

```
1 | var hasCycle = function(head) {  
2 |   try{  
3 |     JSON.stringify(head)  
4 |   }catch(e){  
5 |     return true  
6 |   }  
7 |   return false  
8 | };
```

当然，这种解法并不是建议的标准题解！在此列出是为了拓宽思维！（大家如有兴趣，可以自己去看下 JSON.stringify 内部的实现，是如何检测循环引用的。）

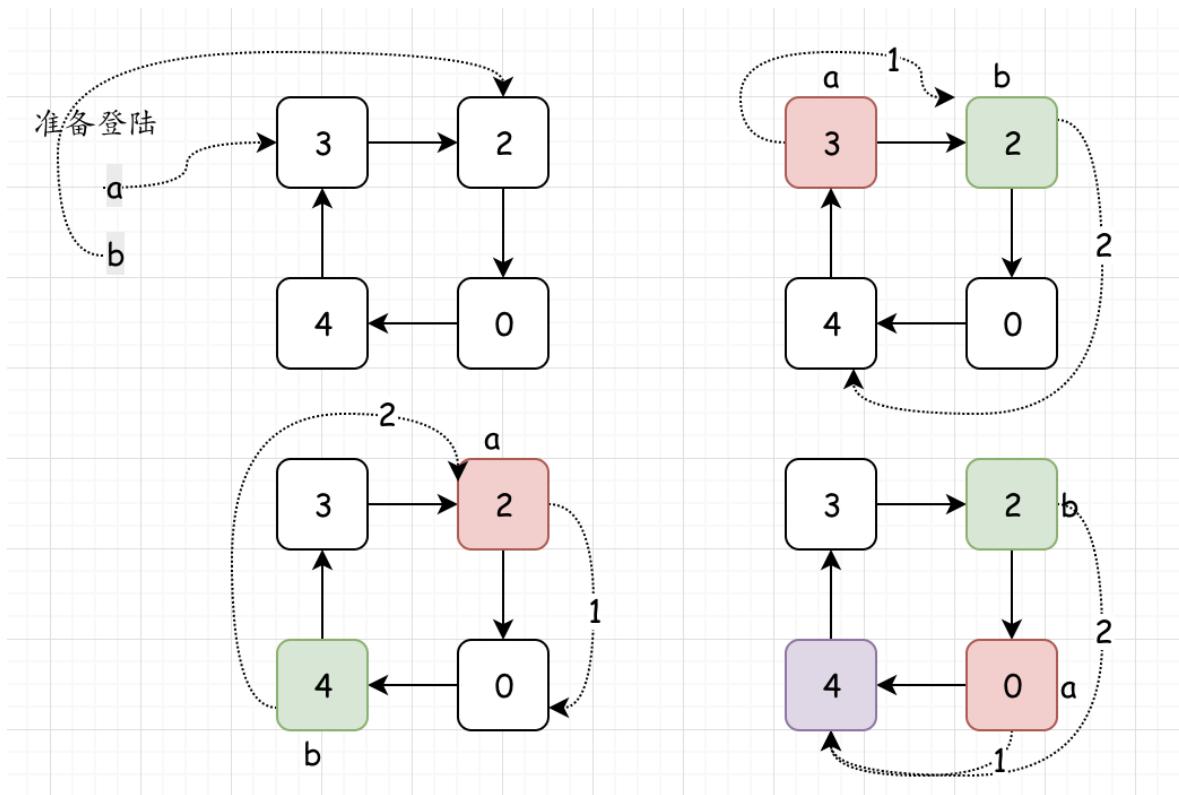
### 题解三：双指针解法

本题标准解法！常识内容，**必须掌握**！

思路来源：先想象一下，两名运动员以不同速度在跑道上进行跑步会怎么样？相遇！好了，这道题你会了。

解题方法：通过使用具有不同速度的快、慢两个指针遍历链表，空间复杂度可以被降低至  $O(1)$ 。慢指针每次移动一步，而快指针每次移动两步。

假设链表为 ，其步骤如下：



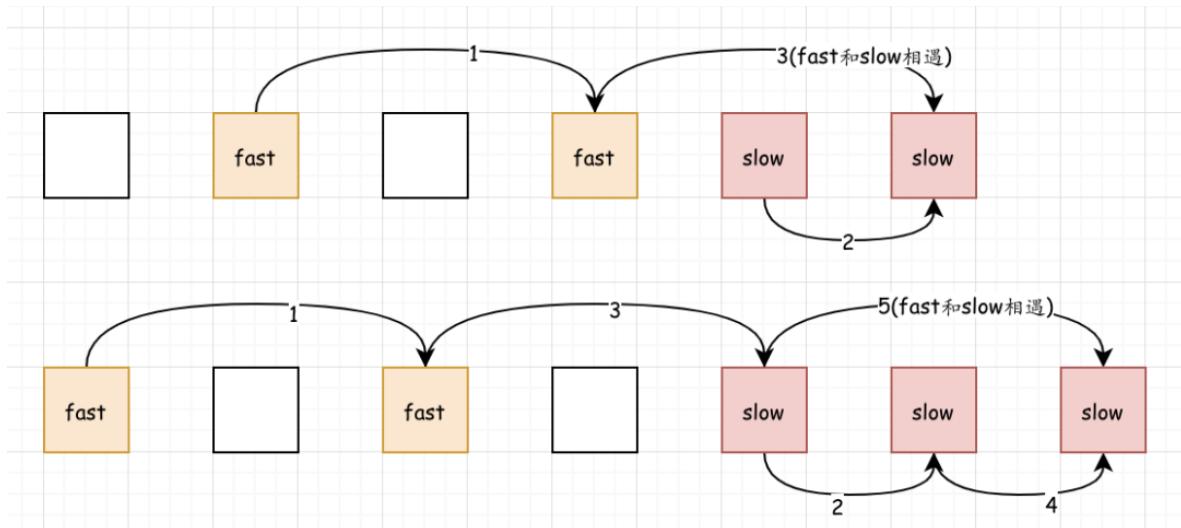
分析完毕，直接上代码：

```
1 func hasCycle(head *ListNode) bool {
2     if head == nil {
3         return false
4     }
5     fast := head.Next      // 快指针，每次走两步
6     for fast != nil && head != nil && fast.Next != nil {
7         if fast == head {    // 快慢指针相遇，表示有环
8             return true
9         }
10        fast = fast.Next.Next
11        head = head.Next      // 慢指针，每次走一步
12    }
13    return false
14 }
```

这里我们要特别说明一下，为什么慢指针的步长设置为 1，而快指针步长设置为 2。

首先，慢指针步长为 1，很容易理解，因为我们需要让慢指针步行至每一个元素。而快指针步长为 2，通俗点可以理解为他们的相对速度只差 1，快的只能一个一个格子的去追慢的，必然在一个格子相遇。

如果没看懂，我们来分析：在快的快追上慢的时，他们之间一定是只差 1 个或者 2 个格子。如果落后 1 个，那么下一次就追上了。如果落后 2 个，那么下一次就落后 1 个，再下一次就能追上！如下图：



所以我们的快指针的步长可以设置为 2。

### 03、特别说明

我们常会遇到一些所谓的“简单题目”，然后用着前人留下来的那些“经典题解”迅速作答。在解题的过程中，追求公式化、模板化。当然，这个过程是好的，因为社会、工作、学业要求我们如此！但是，我希望我们也可以留下一些自己的思考，纵然不是最优解，但是是我们自己想到的、创造的！真正在算法题中去收获快乐～

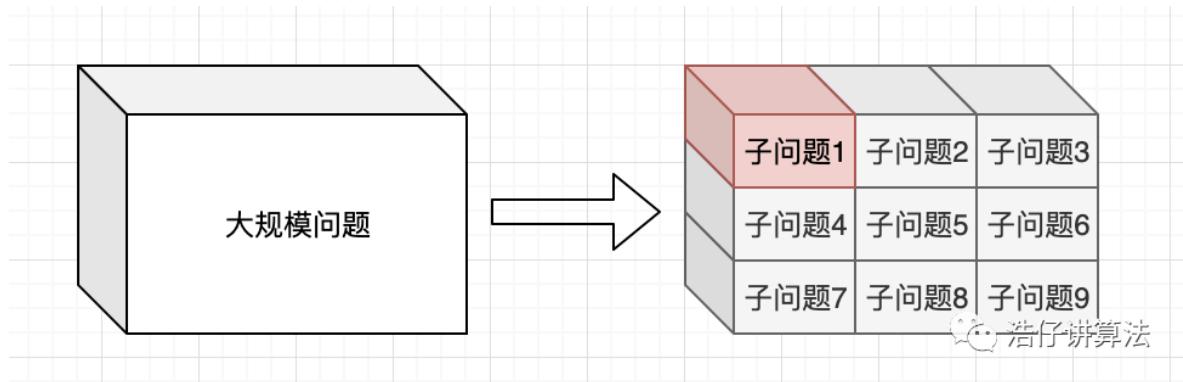
## 动态规划系列

### 爬楼梯(70)

#### 01、概念讲解

关于动态规划的资料很多，官方的定义是指把多阶段过程转化为一系列单阶段问题，利用各阶段之间的关系，逐个求解。概念中的各阶段之间的关系，其实指的就是状态转移方程。很多人觉得DP难（下文统称动态规划为DP），根本原因是由于DP跟一些固定形式的算法不同（比如DFS、二分法、KMP），它没有实际的步骤规定第一步、第二步来做什么，所以准确来说，**DP其实是一种解决问题的思想**。

这种思想的本质是：**一个规模比较大的问题**（可以用两三个参数表示的问题），**可以通过若干规模较小的问题的结果来得到的**（通常会寻求到一些特殊的计算逻辑，如求最值等），如下图所示，一个大规模的问题由若干个子问题组成。



那么我们应该如何通过子问题去得到大规模问题呢？这就用到了**状态转移方程**（上面有介绍状态转移方程哦，不懂的请往上翻哦），我们一般看到的状态转移方程，基本都是这样：

```
1 opt : 指代特殊的计算逻辑，通常为 max or min。  
2  
3 i,j,k 都是在定义DP方程中用到的参数。  
4  
5 dp[i] = opt(dp[i-1])+1  
6  
7 dp[i][j] = w(i,j,k) + opt(dp[i-1][k])  
8  
9 dp[i][j] = opt(dp[i-1][j] + xi, dp[i][j-1] + yj, ...)
```

每一个状态转移方程，多少都有一些细微的差别。这个其实很容易理解，世间的关系多了去了，不可能抽象出完全可以套用的公式。所以我个人其实**不建议去死记硬背各种类型的状态转移方程**。但是DP的题型真的就完全无法掌握，无法归类进行分析吗？我认为不是的。在本系列中，我将由简入深为大家讲解动态规划这个主题。

## 02、题目分析

我们先看一道最简单的DP题目，熟悉DP的概念：

### 第70题：爬楼梯

假设你正在爬楼梯。需要  $n$  阶你才能到达楼顶。每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？**注意：**给定  $n$  是一个正整数。

#### 示例 1：

1	输入： 2      输出： 2      解释： 有两种方法可以爬到楼顶。
2	1. 1 阶 + 1 阶
3	2. 2 阶

#### 示例 2：

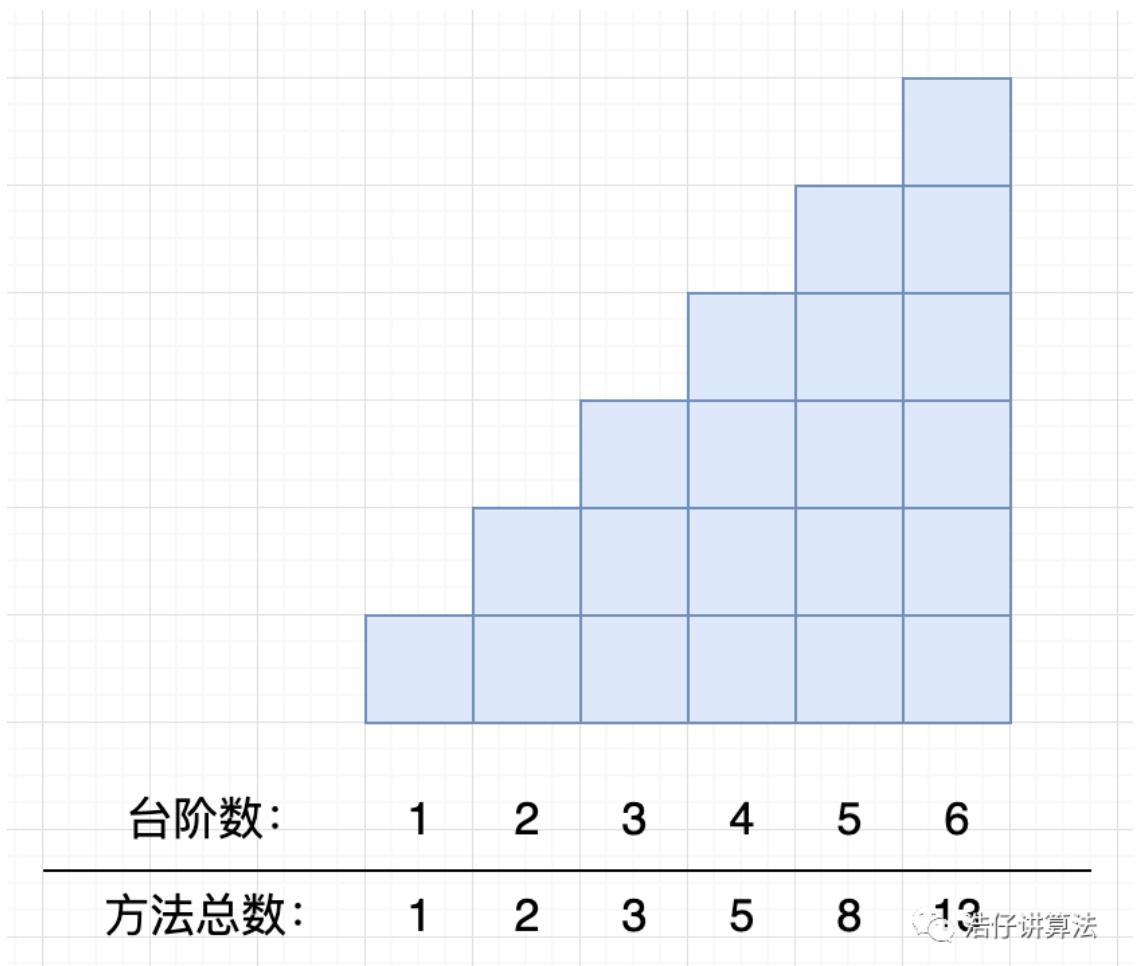
1	输入： 3      输出： 3      解释： 有三种方法可以爬到楼顶。
2	1. 1 阶 + 1 阶 + 1 阶
3	2. 1 阶 + 2 阶
4	3. 2 阶 + 1 阶

## 03、图解分析

通过分析我们可以明确，该题可以被分解为一些包含最优子结构的子问题，即它的**最优解可以从其子问题的最优解来有效地构建**。满足“**将大问题分解为若干个规模较小的问题**”的条件。所以我们令  $dp[n]$  表示能到达第  $n$  阶的方法总数，可以得到如下状态转移方程：

$$dp[n] = dp[n-1] + dp[n-2]$$

- 上 1 阶台阶：有 1 种方式。
- 上 2 阶台阶：有 1+1 和 2 两种方式。
- 上 3 阶台阶：到达第 3 阶的方法总数就是到第 1 阶和第 2 阶的方法数之和。
- 上  $n$  阶台阶，到达第  $n$  阶的方法总数就是到第  $(n-1)$  阶和第  $(n-2)$  阶的方法数之和。



## 04、GO语言示例

根据以上分析，可以得到代码如下：

```

1 func climbStairs(n int) int {
2     if n == 1 {
3         return 1
4     }
5     dp := make([]int, n+1)
6     dp[1] = 1
7     dp[2] = 2
8     for i := 3; i <= n; i++ {
9         dp[i] = dp[i-1] + dp[i-2]
10    }
11    return dp[n]
12 }
```

## 最大子序和(53)

在上一篇文章[011.动态规划系列—第一讲\(70\)](#)中，我们讲解了DP的概念并且通过示例了解了什么是动态规划。本篇中，我们将继续通过1道简单题型，进一步学习动态规划的思想。

## 01、题目分析

### 第53题：最大子序和

给定一个整数数组  $\text{nums}$ ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

示例：

- 1 输入：  $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ ，
- 2 输出： 6
- 3 解释： 连续子数组  $[4, -1, 2, 1]$  的和最大，为 6。

拿到题目请不要直接看下方题解，先自行思考2-3分钟....

## 02、题目图解

首先我们分析题目，一个连续子数组一定要以一个数作为结尾，那么我们可以将状态定义成如下：

$\text{dp}[i]$ ：表示以  $\text{nums}[i]$  结尾的连续子数组的最大和。

那么为什么这么定义呢？因为这样定义其实是最容易想到的！在上一节中我们提到，状态转移方程其实是通过1-3个参数的方程来描述小规模问题和大规模问题间的关系。

当然，如果你没有想到，其实也非常正常！因为该问题最早于 1977 年提出，但是直到 1984 年才被发现了线性时间的最优解法。

根据状态的定义，我们继续进行分析：如果要得到  $\text{dp}[i]$ ，那么  $\text{nums}[i]$  一定会被选取。并且  $\text{dp}[i]$  所表示的连续子序列与  $\text{dp}[i-1]$  所表示的连续子序列很可能就差一个  $\text{nums}[i]$ 。即：

$$\text{dp}[i] = \text{dp}[i-1] + \text{nums}[i], \text{ if } (\text{dp}[i-1] \geq 0)$$

但是这里我们遇到一个问题，**很有可能  $\text{dp}[i-1]$  本身是一个负数**。那这种情况的话，**如果  $\text{dp}[i]$  通过  $\text{dp}[i-1]+\text{nums}[i]$  来推导，那么结果其实反而变小了**，因为我们  $\text{dp}[i]$  要求的是最大和。所以在这种情况下，**如果  $\text{dp}[i-1] < 0$ ，那么  $\text{dp}[i]$  其实就是  $\text{nums}[i]$  的值**。即

$$\text{dp}[i] = \text{nums}[i], \text{ if } (\text{dp}[i-1] < 0)$$

综上分析，我们可以得到：

$$\text{dp}[i] = \max(\text{nums}[i], \text{dp}[i-1] + \text{nums}[i])$$

得到了状态转移方程，但是我们还需要通过一个已有的状态的进行推导，我们可以想到  $\text{dp}[0]$  一定是以  $\text{nums}[0]$  进行结尾，所以

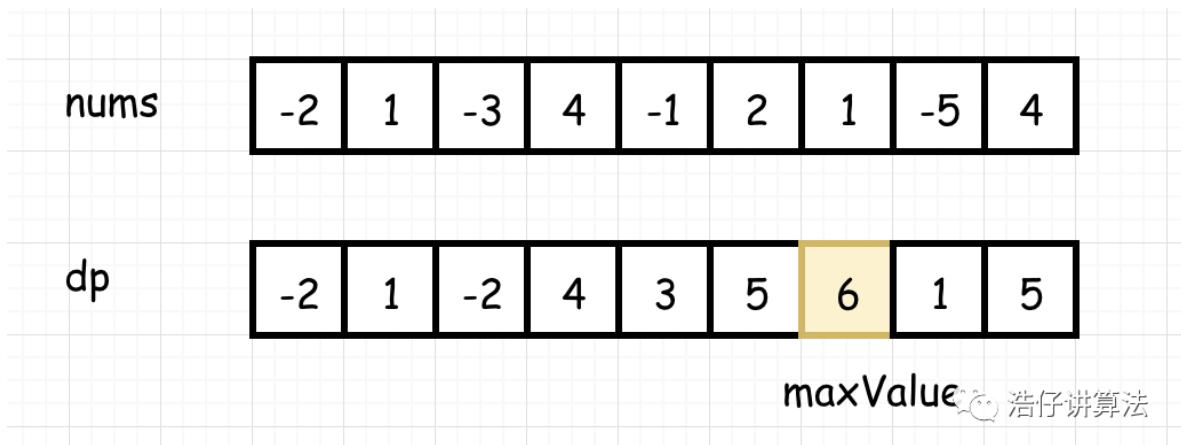
$$\text{dp}[i] = \text{dp}[i-1] + \text{nums}[i], \text{ if } (\text{dp}[i-1] \geq 0)$$

```
dp[0] = nums[0]
```

在很多题目中，因为  $dp[i]$  本身就定义成了题目中的问题，所以  $dp[i]$  最终就是要的答案。但是这里状态中的定义，并不是题目中要的问题，不能直接返回最后的一个状态（这一步经常有初学者会摔跟头）。所以最终的答案，其实我们是寻找：

```
max(dp[0], dp[1], ..., dp[i-1], dp[i])
```

分析完毕，我们绘制成图（图中假定  $nums$  为  $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$ ）：



## 03、Go语言示例

根据以上分析，可以得到代码如下：

```
1 //Go
2 func maxSubArray(nums []int) int {
3     if len(nums) < 1 {
4         return 0
5     }
6     dp := make([]int, len(nums))
7     //设置初始化值
8     dp[0] = nums[0]
9     for i := 1; i < len(nums); i++ {
10        //处理 dp[i-1] < 0 的情况
11        if dp[i-1] < 0 {
12            dp[i] = nums[i]
13        } else {
14            dp[i] = dp[i-1] + nums[i]
15        }
16    }
17    result := -1 << 31
18    for _, k := range dp {
19        result = max(result, k)
20    }
21    return result
22 }
23
24 func max(a, b int) int {
25     if a > b {
26         return a
27     }
28     return b
}
```

我们可以进一步精简代码为：

```

1 //Go
2 func maxSubArray(nums []int) int {
3     if len(nums) < 1 {
4         return 0
5     }
6     dp := make([]int, len(nums))
7     result := nums[0]
8     dp[0] = nums[0]
9     for i := 1; i < len(nums); i++ {
10        dp[i] = max(dp[i-1]+nums[i], nums[i])
11        result = max(dp[i], result)
12    }
13    return result
14 }
15
16 func max(a, b int) int {
17     if a > b {
18         return a
19     }
20     return b
21 }
```

复杂度分析：时间复杂度： $O(N)$ 。空间复杂度： $O(N)$

## 最长上升子序列(300)

在上一篇中，我们了解了什么是DP（动态规划），并且通过DP中的经典问题“最大子序和”，学习了状态转移方程应该如何定义。在本节中，我们将沿用之前的分析方法，通过一道例题，进一步巩固之前的内容！

### 01、题目分析

#### 第300题：最长上升子序列

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：

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

#### 说明：

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。

**这道题有一定难度哦！如果没有思路请回顾上一篇的学习内容！  
不建议直接看题解！**

## 02、题目图解

首先我们分析题目，要找的是最长上升子序列（Longest Increasing Subsequence, LIS）。因为题目中没有要求连续，所以LIS可能是连续的，也可能是非连续的。同时，LIS符合可以从其子问题的最优解来进行构建的条件。所以我们可以尝试用动态规划来进行求解。首先我们定义状态：

dp[i]：表示以nums[i]结尾的最长上升子序列的长度

我们假定nums为[1, 9, 5, 9, 3]，如下图：

nums	1	9	5	9	3
					浩仔讲算法

我们分两种情况进行讨论：

- 如果nums[i]比前面的所有元素都小，那么dp[i]等于1（即它本身）（该结论正确）
- 如果nums[i]前面存在比他小的元素nums[j]，那么dp[i]就等于dp[j]+1（该结论错误，比如 nums[3]>nums[0]，即9>1，但是dp[3]并不等于dp[0]+1）

我们先初步得出上面的结论，但是我们发现了一些问题。因为dp[i]前面比他小的元素，不一定只有一个！

可能除了nums[j]，还包括nums[k], nums[p]等等等。所以dp[i]除了可能等于dp[j]+1，还有可能等于dp[k]+1, dp[p]+1等等等。所以我们求dp[i]，需要找到dp[j]+1, dp[k]+1, dp[p]+1等等等中的最大值。（我在3个等等等上都进行了加粗，主要是因为初学者非常容易在这里摔跟斗！这里强调的目的是希望能记住这道题型！）即：

$$dp[i] = \max(dp[j]+1, dp[k]+1, dp[p]+1, \dots)$$

只要满足：

$$nums[i] > nums[j]$$

$$nums[i] > nums[k]$$

$$nums[i] > nums[p]$$

....

最后，我们只需要找到dp数组中的最大值，就是我们要找的答案。

分析完毕，我们绘制成图：

nums	<table border="1"><tr><td>1</td><td>9</td><td>5</td><td>9</td><td>3</td></tr></table>	1	9	5	9	3	
1	9	5	9	3			
dp[0]	<table border="1"><tr><td>1</td></tr></table>	1	dp[0]=1				
1							
dp[1]	<table border="1"><tr><td>1</td><td>9</td></tr></table>	1	9	dp[1]=dp[0]+1=2			
1	9						
dp[2]	<table border="1"><tr><td>1</td><td>5</td></tr></table>	1	5	dp[2]=dp[0]+1=2			
1	5						
dp[3]	<table border="1"><tr><td>1</td><td>5</td><td>9</td></tr></table>	1	5	9	dp[2]=max(dp[0]+1,dp[2]+1)=3		
1	5	9					
dp[4]	<table border="1"><tr><td>1</td><td>3</td></tr></table>	1	3	dp[2]=dp[0]+1=2			
1	3						

浩仔讲算法

### 03、Go语言示例

根据以上分析，可以得到代码如下：

```

1 func lengthofLIS(nums []int) int {
2     if len(nums) < 1 {
3         return 0
4     }
5     dp := make([]int, len(nums))
6     result := 1
7     for i := 0; i < len(nums); i++ {
8         dp[i] = 1
9         for j := 0; j < i; j++ {
10            if nums[j] < nums[i] {
11                dp[i] = max(dp[j]+1, dp[i])
12            }
13        }
14        result = max(result, dp[i])
15    }
16    return result
17 }
18
19 func max(a, b int) int {
20     if a > b {
21         return a
22     }
23     return b
24 }
```

### 三角形最小路径和(120)

在上一篇中，我们通过题目“最长上升子序列”以及“最大子序和”，学习了DP（动态规划）在线性关系中的分析方法。这种分析方法，也在运筹学中被称为“线性动态规划”，具体指的是“目标函数为特定变量的线性函数，约束是这些变量的线性不等式或等式，目的是求目标函数的最大值或最小值”。这点大家作为了解即可，不需要死记，更不要生搬硬套！

在本节中，我们将继续分析一道略微区别于之前的题型，希望可以由此题与之前的题目进行对比论证，进而顺利求解！

## 01、题目分析

### 第120题：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。

例如，给定三角形：

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

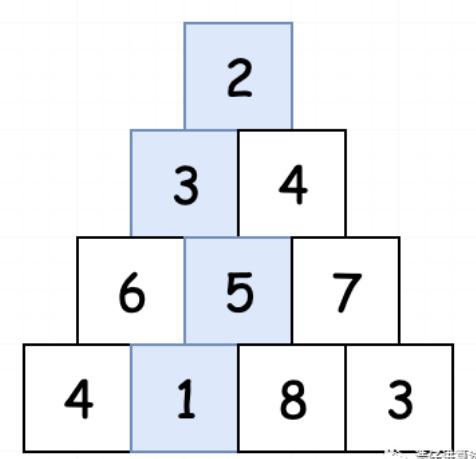
则自顶向下的最小路径和为 11（即， $2 + 3 + 5 + 1 = 11$ ）。

这道题有一定难度哦！如果没有思路请回顾上一篇的学习内容！

不建议直接看题解！

## 02、题目图解

首先我们分析题目，要找的是**三角形最小路径和**，这是个啥意思呢？假设我们有一个三角形：[[2], [3,4], [6,5,7], [4,1,8,3]]



那从上到下的最小路径和就是2-3-5-1，等于11。

由于我们是使用数组来定义一个三角形，所以便于我们分析，我们将三角形稍微进行改动：

2			
3	4		
6	5	7	
4	1	8	3

浩仔讲算法

这样相当于我们将整个三角形进行了拉伸。这时候，我们根据题目中给出的条件：每一步只能移动到下一行中相邻

的结点上。其实也就等同于，**每一步我们只能往下移动一格或者右下移动一格**。将其转化成代码，假如2所在的元

素位置为[0,0]，那我们往下移动就只能移动到[1,0]或者[1,1]的位置上。假如5所在的位置为[2,1]，同样也只能移动

到[3,1]和[3,2]的位置上。如下图所示：

2			
3	4		
6	5	7	
4	1	8	3

2			
3	4		
6	5	7	
4	1	8	3

浩仔讲算法

题目明确了之后，现在我们开始进行分析。题目很明显是一个**找最优解的问题，并且可以从子问题的最优解进**

**行构建**。所以我们通过动态规划进行求解。首先，我们定义状态：

$dp[i][j]$  : 表示包含第*i*行*j*列元素的最小路径和

我们很容易想到可以自顶向下进行分析。并且，无论最后的路径是哪一条，它一定要经过最顶上的元素，即 [0,0]。所以我们需要对  $dp[0][0]$  进行初始化。

$dp[0][0] = [0][0]$  位置所在的元素值

继续分析，如果我们要求  $dp[i][j]$ ，那么其一定会从自己头顶上的两个元素移动而来。

2			
3	4		
6	5	7	
4	1	8	3

浩仔讲算法

如5这个位置的最小路径和，要么是从2-3-5而来，要么是从2-4-5而来。然后取两条路径和中较小的一个即可。进

而我们得到状态转移方程：

$$dp[i][j] = \min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j]$$

但是，我们这里会遇到一个问题！除了最顶上的元素之外，

2			
3	4		
6	5	7	
4	1	8	3

浩仔讲算法

最左边的元素只能从自己头顶而来。 (2-3-6-4)

2			
3	4		
6	5	7	
4	1	8	3

浩仔讲算法

最右边的元素只能从自己左上角而来。 (2-4-7-3)

然后，我们观察发现，位于第2行的元素，都是特殊元素（因为都只能从[0,0]的元素走过来）

	2			
3	4			
6	5	7		
4	1	8	3	

动态规划算法

我们可以直接将其特殊处理，得到：

```
dp[1][0] = triangle[1][0] + triangle[0][0]
dp[1][1] = triangle[1][1] + triangle[0][0]
```

最后，我们只要找到最后一行元素中，路径和最小的一个，就是我们的答案。即：

```
I: dp数组长度
result = min(dp[I-1,0], dp[I-1,1], dp[I-1,2]....)
```

综上我们就分析完了，我们总共进行了4步：

1. 定义状态
2. 总结状态转移方程
3. 分析状态转移方程不能满足的特殊情况。
4. 得到最终解

## 03、Go语言示例

根据以上分析，可以得到代码如下：

```
1 func minimumTotal(triangle [][]int) int {
2     if len(triangle) < 1 {
3         return 0
4     }
5     if len(triangle) == 1 {
6         return triangle[0][0]
7     }
8     dp := make([][]int, len(triangle))
9     for i, arr := range triangle {
10        dp[i] = make([]int, len(arr))
11    }
12    result := 1<<31 - 1
13    dp[0][0] = triangle[0][0]
14    dp[1][1] = triangle[1][1] + triangle[0][0]
15    dp[1][0] = triangle[1][0] + triangle[0][0]
16
17    for i := 2; i < len(triangle); i++ {
18        for j := 0; j < len(triangle[i]); j++ {
19            if j == 0 {
```

```
20             dp[i][j] = dp[i-1][j] + triangle[i][j]
21     } else if j == (len(triangle[i]) - 1) {
22         dp[i][j] = dp[i-1][j-1] + triangle[i][j]
23     } else {
24         dp[i][j] = min(dp[i-1][j-1], dp[i-1][j]) + triangle[i][j]
25     }
26 }
27 for _, k := range dp[len(dp)-1] {
28     result = min(result, k)
29 }
30 return result
31 }
32
33 func min(a, b int) int {
34     if a > b {
35         return b
36     }
37     return a
38 }
```

运行结果：

执行用时：**4 ms**，在所有 Go 提交中击败了 **96.51%** 的用户

内存消耗：**3.3 MB**，在所有 Go 提交中击败了 **26.34%** 的用户

运行上面的代码，我们发现使用的内存过大。我们有没有什么办法可以压缩内存呢？通过观察我们发现，在我们

自顶向下的过程中，其实我们只需要使用到上一层中已经累积计算完毕的数据，并且不会再次访问之前的元素数

据。绘制成图如下：

<1>

2			
3	4		
6	5	7	
4	1	8	3

<2>

2			
5	6		
6	5	7	
4	1	8	3

<3>

2			
5	6		
11	10	13	
4	1	8	3

<4>

2			
5	6		
11	10	13	
15	11	18	16

告子讲算法

优化后的代码如下：

```

1 func minimumTotal(triangle [][]int) int {
2     l := len(triangle)
3     if l < 1 {
4         return 0
5     }
6     if l == 1 {
7         return triangle[0][0]
8     }
9     result := 1<<31 - 1
10    triangle[0][0] = triangle[0][0]
11    triangle[1][1] = triangle[1][1] + triangle[0][0]
12    triangle[1][0] = triangle[1][0] + triangle[0][0]
13    for i := 2; i < l; i++ {
14        for j := 0; j < len(triangle[i]); j++ {
15            if j == 0 {
16                triangle[i][j] = triangle[i-1][j] + triangle[i][j]
17            } else if j == (len(triangle[i]) - 1) {
18                triangle[i][j] = triangle[i-1][j-1] + triangle[i][j]
19            } else {
20                triangle[i][j] = min(triangle[i-1][j-1], triangle[i-1][j])
21                + triangle[i][j]
22            }
23        }
24    }
25    return result ^ triangle[l-1][l-1]
26 }
```

```
23     }
24     for _, k := range triangle[1-1] {
25         result = min(result, k)
26     }
27     return result
28 }
29
30 func min(a, b int) int {
31     if a > b {
32         return b
33     }
34     return a
35 }
```

运行结果：

执行用时：**4 ms**，在所有 Go 提交中击败了 **96.51%** 的用户  
内存消耗：**3.1 MB**，在所有 Go 提交中击败了 **100.00%** 的用户

## 最小路径和(64)

在上一篇中，我们通过分析，顺利完成了“**三角形最小路径和**”的动态规划题解。在本节中，我们继续看一道相似题型，以求能完全掌握这种“路径和”的问题。话不多说，先看题目：

### 01、题目分析

#### 第64题：最小路径和

给定一个包含非负整数的  $m \times n$  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：

```
1 输入：
2 [
3     [1,3,1],
4     [1,5,1],
5     [4,2,1]
6 ]
7 输出： 7
8 解释：因为路径 1→3→1→1→1 的总和最小。
```

这道题有一定难度哦！如果没有思路请回顾上一篇的学习内容！

不建议直接看题解！

## 02、题目图解

首先我们分析题目，要找的是 **最小路径和**，这是个啥意思呢？假设我们有一个  $m * n$  的矩形： [[1,3,1], [1,5,1], [4,2,1]]

1	3	1
1	5	1
4	2	1

那从**左上角到右下角**的最小路径和，我们可以很容易看出就是 1-3-1-1-1，这一条路径，结果等于 7。

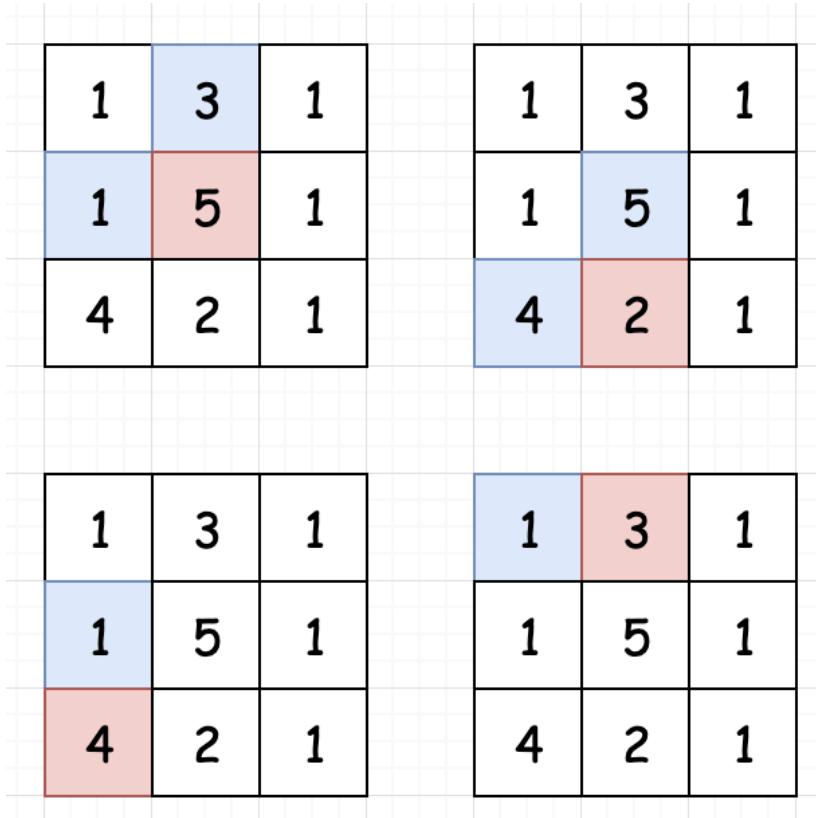
题目明确了，我们继续进行分析。该题与上一道求三角形最小路径和一样，题目明显符合可以**从子问题的最优解进行构建**，所以我们考虑使用动态规划进行求解。首先，我们定义状态：

dp[i][j]：表示包含第i行j列元素的最小路径和

同样，因为任何一条到达右下角的路径，都会经过 [0,0] 这个元素。所以我们需要对 dp[0][0] 进行初始化。

dp[0][0] = [0][0]位置所在的元素值

继续分析，根据题目给的条件，如果我们要求 dp[i][j]，那么它一定是从自己的上方或者左边移动而来。如下图所示：



5, 只能从3或者1移动而来  
2, 只能从5或者4移动而来  
4, 从1移动而来

3, 从1移动而来  
(红色位置必须从蓝色位置移动而来)

进而我们得到状态转移方程：

$$dp[i][j] = \min(dp[i-1][j], dp[i][j-1]) + grid[i][j]$$

同样我们需要考虑两种特殊情况：

- 最上面一行，只能由左边移动而来 (1-3-1)
- 最左边一列，只能由上面移动而来 (1-1-4)

1	3	1
1	5	1
4	2	1

1	3	1
1	5	1
4	2	1

最后，因为我们的目标是**从左上角走到右下角，整个网格的最小路径和其实就是包含右下角元素的最小路径和**。即：

设：dp的长度为l  
最终结果就是： $dp[l-1][len(dp[l-1])-1]$

综上我们就分析完了，我们总共进行了4步：

1. 定义状态
2. 总结状态转移方程
3. 分析状态转移方程不能满足的特殊情况。
4. 得到最终解

## 03、GO语言示例

根据以上分析，可以得到代码如下：

```
1 func minPathSum(grid [][]int) int {
2     l := len(grid)
3     if l < 1 {
4         return 0
5     }
6     dp := make([][]int, l)
7     for i, arr := range grid {
8         dp[i] = make([]int, len(arr))
9     }
10    dp[0][0] = grid[0][0]
11    for i := 0; i < l; i++ {
12        for j := 0; j < len(grid[i]); j++ {
13            if i == 0 && j != 0 {
14                dp[i][j] = dp[i][j-1] + grid[i][j]
15            } else if j == 0 && i != 0 {
16                dp[i][j] = dp[i-1][j] + grid[i][j]
17            } else if i != 0 && j != 0 {
```

```
18         dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]
19     }
20 }
21 return dp[l-1][len(dp[l-1])-1]
22 }
23 }
24
25 func min(a, b int) int {
26     if a > b {
27         return b
28     }
29     return a
30 }
```

运行结果：

执行用时：**8 ms**，在所有 Go 提交中击败了 **91.47%** 的用户

内存消耗：**4.4 MB**，在所有 Go 提交中击败了 **42.02%** 的用户

同样，运行上面的代码，我们发现使用的内存过大。有没有什么办法可以压缩内存呢？通过观察我们发现，在我们自左上角到右下角计算各个节点的最小路径和的过程中，我们只需要使用到之前已经累积计算完毕的数据，并且不会再次访问之前的元素数据。绘制成图如下：(大家看这个过程像不像扫雷，其实如果大家研究扫雷外挂的话，就会发现在扫雷的核心算法中，就有一处颇为类似这种分析方法，这里就不深究了)

1	3	1
1	5	1
4	2	1

<1>

1	4	1
5	5	1
4	2	1

<2>

1	4	1
5	9	1
4	2	1

<3>

1	4	5
5	9	1
4	2	1

<4>

1	4	5
5	9	6
4	2	1

<5>

1	4	5
5	9	6
9	2	1

<6>

1	4	5
5	9	6
9	11	1

<7>

1	4	5
5	9	6
9	11	7

<8>

优化后的代码如下：

```

1 func minPathSum(grid [][]int) int {
2     l := len(grid)
3     if l < 1 {
4         return 0
5     }
6     for i := 0; i < l; i++ {
7         for j := 0; j < len(grid[i]); j++ {
8             if i == 0 && j != 0 {
9                 grid[i][j] = grid[i][j-1] + grid[i][j]
10            } else if j == 0 && i != 0 {
11                grid[i][j] = grid[i-1][j] + grid[i][j]
12            } else if i != 0 && j != 0 {
13                grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j]
14            }
15        }
16    }
17    return grid[l-1][len(grid[l-1])-1]
18 }
19
20 func min(a, b int) int {
21     if a > b {

```

```
22     return b  
23 }  
24 return a  
25 }
```

运行结果：

执行用时：**4 ms**，在所有 Go 提交中击败了 **98.91%** 的用户

内存消耗：**3.9 MB**，在所有 Go 提交中击败了 **100.00%** 的用户

课后思考：路径和类问题和之前的子序列类问题有何区别？

## 打家劫舍(198)

在前两篇中，我们分别学习了“**三角形最小路径和**”以及“**矩形最小路径和**”的问题，相信已经掌握了这类题型的解题方式。我们只要**明确状态的定义**，基本上都可以顺利求解。

在本节中，我们将回归一道简单点的题目，目的是剖析一下**状态定义的过程**，并且举例说明如果状态定义错误，会对我们带来多大困扰！希望大家不要轻视！

### 01、题目分析

#### 第198题：打家劫舍

你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

#### 示例 1：

```
1 输入： [1,2,3,1]  
2 输出： 4  
3 解释： 偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。  
4     偷窃到的最高金额 = 1 + 3 = 4 。
```

#### 示例 2：

```
1 输入： [2,7,9,3,1]  
2 输出： 12  
3 解释： 偷窃 1 号房屋（金额 = 2），偷窃 3 号房屋（金额 = 9），接着偷窃 5 号房屋（金额 = 1）。  
4     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

本题主要剖析状态定义的过程！强烈建议先进行前面5节内容的学习！

以达到最好的学习效果！

## 02、题目图解

假设有 $i$ 间房子，我们可能会定义出两种状态：

- $dp[i]$ ：偷盗含第 $i$ 个房子时，所获取的最大利益
- $dp[i]$ ：偷盗至第 $i$ 个房子时，所获取的最大利益

如果我们定义为状态一，因为我们没办法知道**获取最高金额时**，小偷到底偷盗了哪些房屋。所以我们需要找到**所有状态中的最大值**，才能找到我们的最终答案。即：

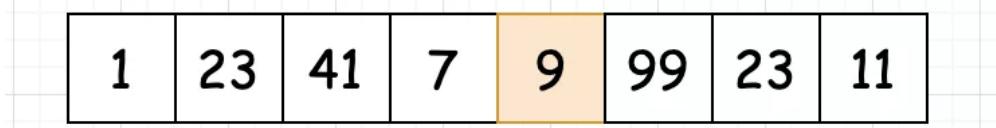
$$\max(dp[0], dp[1], \dots, dp[\text{len}(dp)-1])$$

如果我们定义为状态二，因为小偷一定会**从前偷到最后**（强调：偷盗至第 $i$ 个房间，不代表小偷要从第 $i$ 个房间中获取财物）。所以我们的最终答案很容易确定。即：

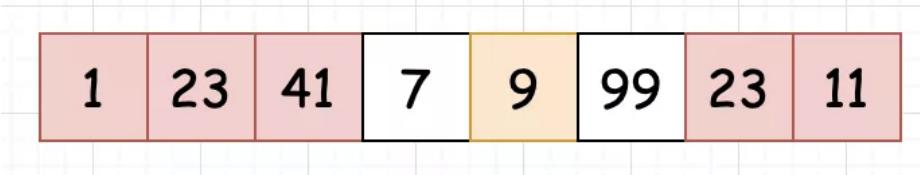
$$dp[i]$$

现在我们分析这两种状态定义下的状态转移方程：

如果是状态一，偷盗含第 $i$ 个房间时能获取的最高金额，我们相当于要**找到偷盗每一间房子时可以获取到的最大金额**。比如下图，我们要找到 $dp[4]$ ，也就是偷盗9这间房子时，能获取到的最大金额。



那我们就需要找到与9不相邻的前后两段中能获取到的最大金额。



我们发现题目进入恶性循环，因为我们若要找到与9不相邻的两端中能偷盗的最大金额，根据 $dp[i]$ 的定义，我们就又需要分析在这两段中盗取每一间房子时所能获取的最大利益！想想都很可怕！所以我们放弃掉这种状态的定义。

如果是状态二，偷盗至第 $i$ 个房子时，所能获取的最大利益。那我们可以想到，**由于不可以再相邻的房屋闯入，所以至 $i$ 房屋可盗窃的最大值，要么就是至 $i-1$ 房屋可盗窃的最大值，要么就是至 $i-2$ 房屋可盗窃的最大值加上当前房屋的值，二者之间取最大值**，即：

$$dp[i] = \max(dp[i-2]+nums[i], dp[i-1])$$

如果不能理解可以看下图：

（相当于小贼背了个背包，里边装了之前偷来的财物，每到达下一个房间门口，来选择是偷还是不偷。）

盗贼偷盗的房屋：

9	1	1	10
---	---	---	----

9	1	1	10
---	---	---	----

9	1	1	10
---	---	---	----

9	1	1	10
---	---	---	----

偷至每个房屋所能获取的最大值：

9	1	1	10
---	---	---	----

9	9	1	10
---	---	---	----

9	9	10	10
---	---	----	----

9	9	10	19
---	---	----	----

$$DP[0]=9$$

$$DP[1]=\max(DP[0], DP[1])=9$$

$$DP[2]=\max(DP[1], DP[0]+1)=10$$

$$DP[3]=\max(DP[2], DP[1]+10)=19$$

### 03、GO语言示例

分析完毕，我们根据第二种状态定义进行求解：

```
1 func rob(nums []int) int {
2     if len(nums) < 1 {
3         return 0
4     }
5     if len(nums) == 1 {
6         return nums[0]
7     }
8     if len(nums) == 2 {
9         return max(nums[0], nums[1])
10    }
11    dp := make([]int, len(nums))
12    dp[0] = nums[0]
13    dp[1] = max(nums[0], nums[1])
14    for i := 2; i < len(nums); i++ {
15        dp[i] = max(dp[i-2]+nums[i], dp[i-1])
16    }
17    return dp[len(dp)-1]
18}
19
20 func max(a, b int) int {
21     if a > b {
22         return a
23     }
24     return b
25}
```

```
23     }
24     return b
25 }
```

运行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2 MB，在所有 Go 提交中击败了 27.72% 的用户

同样，运行上面的代码，我们发现使用的内存过大。有没有什么办法可以压缩内存呢？我们很容易想到，在小贼偷盗的过程中，不可能转回头去到自己已经偷过的房间！（太蠢）小偷只需要每次将财物搬到下一个房间就行！

根据上面思路，优化后的代码如下：

```
1 func rob(nums []int) int {
2     if len(nums) < 1 {
3         return 0
4     }
5     if len(nums) == 1 {
6         return nums[0]
7     }
8     if len(nums) == 2 {
9         return max(nums[0], nums[1])
10    }
11    nums[1] = max(nums[0], nums[1])
12    for i := 2; i < len(nums); i++ {
13        nums[i] = max(nums[i-2]+nums[i], nums[i-1])
14    }
15    return nums[len(nums)-1]
16 }
17
18 func max(a, b int) int {
19     if a > b {
20         return a
21     }
22     return b
23 }
```

运行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2 MB，在所有 Go 提交中击败了 100.00% 的用户

## 字符串系列

### 反转字符串(301)

## 01、题目分析

### 第344题：反转字符串

编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

#### 示例 1：

1 | 输入: ["h", "e", "l", "l", "o"]  
2 | 输出: ["o", "l", "l", "e", "h"]

#### 示例 2：

1 | 输入: ["H", "a", "n", "n", "a", "h"]  
2 | 输出: ["h", "a", "n", "n", "a", "H"]

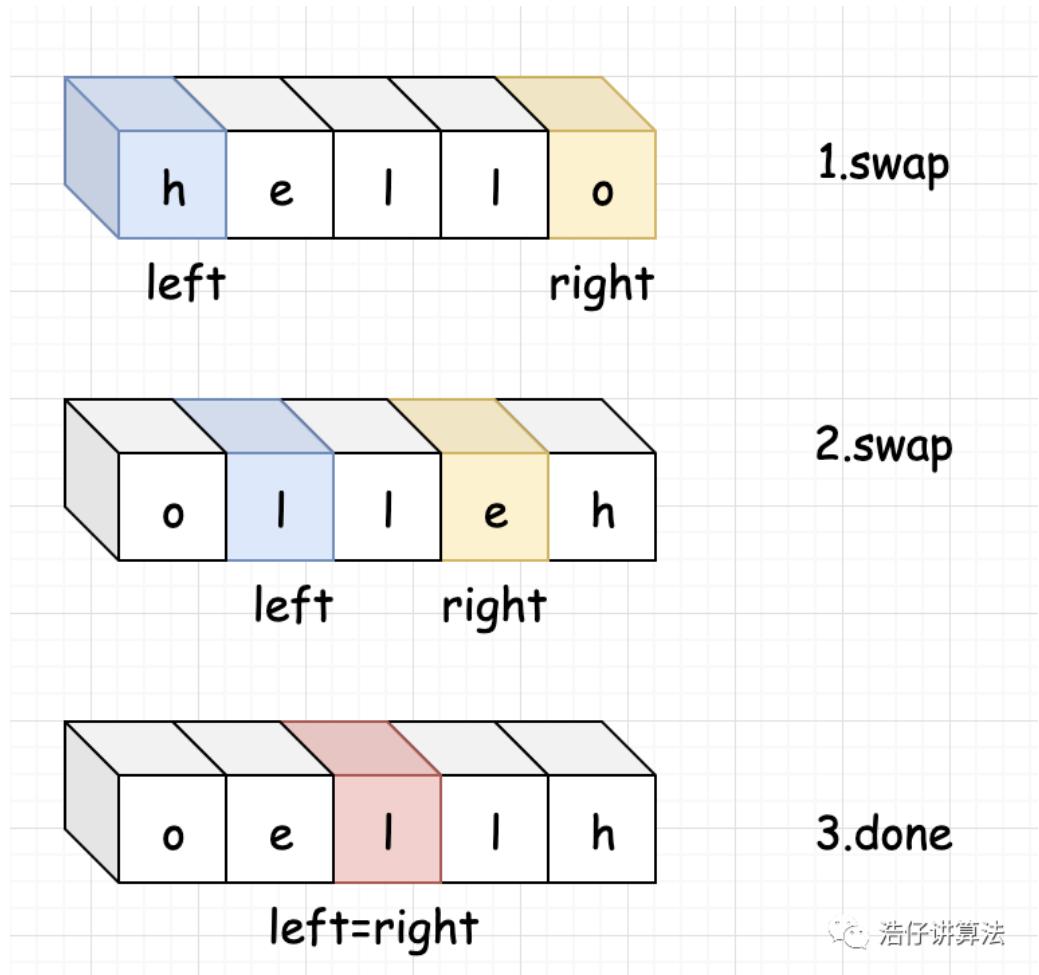
## 02、题目图解

这是一道相当简单的经典题目，直接上题解：使用双指针进行反转字符串。

假设输入字符串为["h","e","l","l","o"]

- 定义left和right分别指向首元素和尾元素
- 当left < right，进行交换。
- 交换完毕，left++，right--
- 直至left == right

具体过程如下图所示：



浩仔讲算法

### 03、Go语言示例

根据以上分析，我们可以得到下面的题解：

```

1 //Go
2 func reverseString(s []byte) {
3     left := 0
4     right := len(s) - 1
5     for left < right {
6         s[left], s[right] = s[right], s[left]
7         left++
8         right--
9     }
10 }
```

### 字符串中的第一个唯一字符(387)

#### 01、题目分析

第387题：字符串中的第一个唯一字符

给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。

案例：

```
1 | s = "leetcode"
2 | 返回 0.
3 |
4 | s = "loveleetcode",
5 | 返回 2.
```

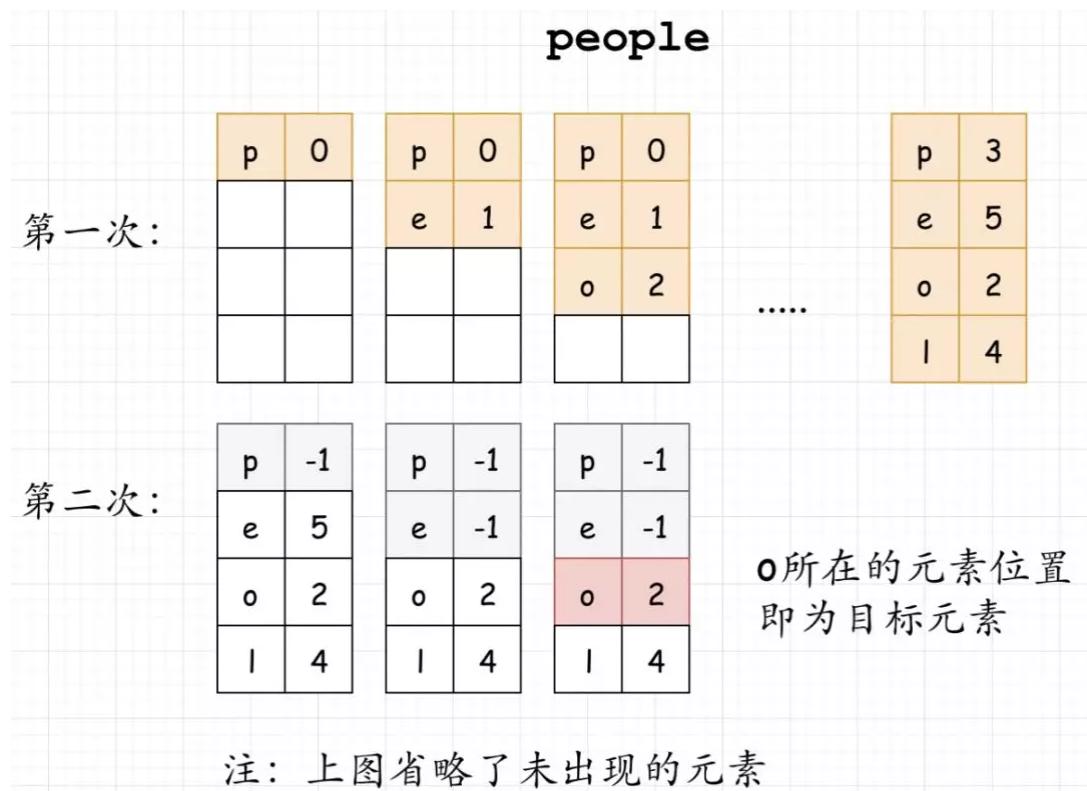
注意事项：您可以假定该字符串只包含小写字母。

常考题目，建议自行思考 1-2 分钟先~

## 02、题目图解

题目不难，直接进行分析。由于字母共有 26 个，所以我们可以声明一个 26 个长度的数组（该种方法在本类题型很常用）因为字符串中字母可能是重复的，所以我们可以先进行第一次遍历，在数组中记录 **每个字母的最后一次出现的所在索引**。然后再通过一次循环，**比较各个字母第一次出现的索引是否为最后一次的索引**。如果是，我们就找到了我们的目标，如果不是我们将其设为 -1（标示该元素非目标元素）如果第二次遍历最终没有找到目标，直接返回 -1 即可。

图解如下：



## 03、GO语言示例

根据以上分析，可以得到代码如下：

```
1 | func firstUniqChar(s string) int {
2 |     var arr [26]int
3 |     for i, k := range s {
```

```

4         arr[k - 'a'] = i
5     }
6     for i,k := range s {
7         if i == arr[k - 'a']{
8             return i
9         }else{
10            arr[k - 'a'] = -1
11        }
12    }
13    return -1
14 }
```

## 二叉树系列

### 最大深度与DFS(104)

在计算机科学中，二叉树是每个结点最多有两个子树的树结构。通常子树被称作“左子树”（left subtree）和“右子树”（right subtree）。二叉树常被用于实现二叉查找树和二叉堆。树比链表稍微复杂，因为链表是线性数据结构，而树不是。树的问题很多都可以由广度优先搜索或深度优先搜索解决。

在本系列中，我们将通过一些例题，学习关于二叉树的经典操作！

#### 01、题目分析

##### 第104题：二叉树的最大深度

给定一个二叉树，找出其最大深度。二叉树的深度为根节点到最远叶子节点的最长路径上的节点数

说明：叶子节点是指没有子节点的节点。

示例：

```

1 给定二叉树 [3,9,20,null,null,15,7],
2     3
3     / \
4     9   20
5     /   \
6     15   7
```

本系列内容均为必须掌握！

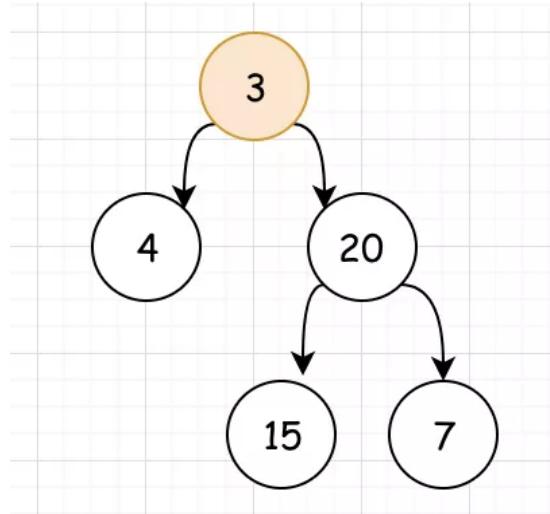
#### 02、递归求解

我们知道，每个节点的深度与它左右子树的深度有关，且等于其左右子树最大深度值加上 1。即：

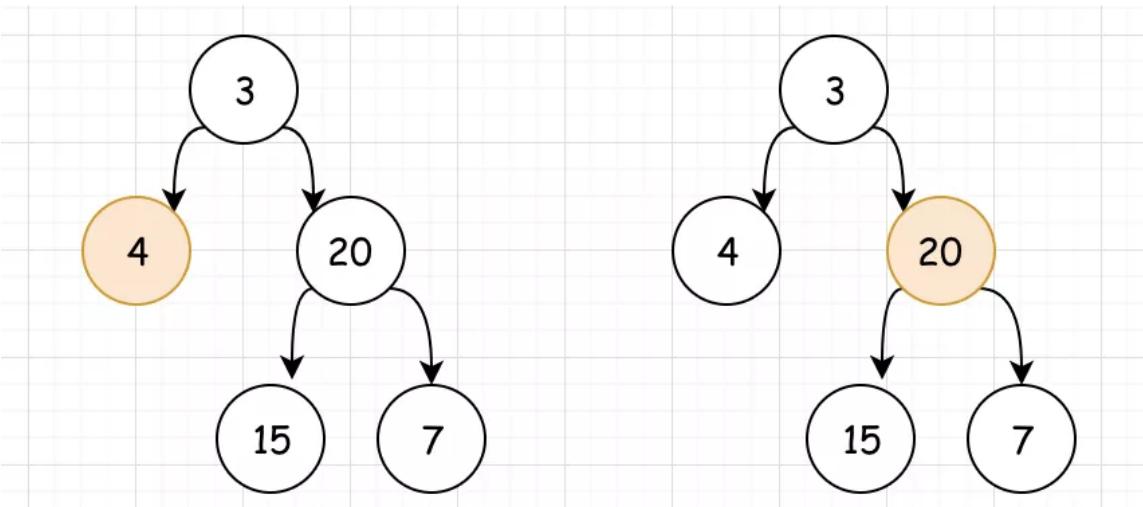
$$\begin{aligned} \text{maxDepth}(\text{root}) &= \max(\text{maxDepth}(\text{root.left}), \\ &\quad \text{maxDepth}(\text{root.right})) + 1 \end{aligned}$$

以 [3,4,20,null,null,15,7] 为例：

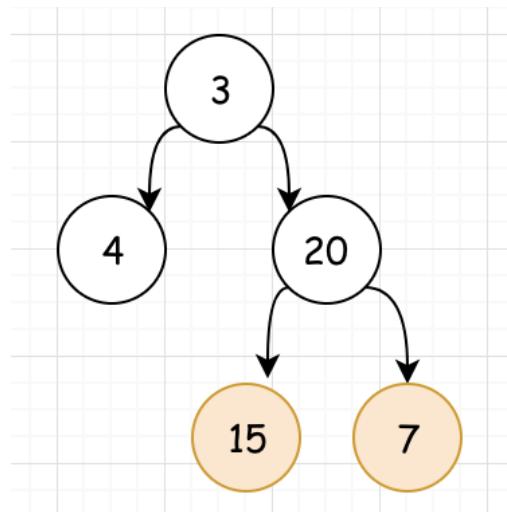
<1>我们要对根节点的最大深度求解，就要对其左右子树的深度进行求解



<2>我们看出。以4为根节点的子树没有左右节点，其深度为1。而以20为根节点的子树的深度，同样取决于它的左右子树深度。



<3>对于15和7的子树，我们可以一眼看出其深度为1。



<4>由此我们可以得到根节点的最大深度为：

```

1 maxDepth(root-3)
2 =max(**maxDepth***(sub-4), **maxDepth***(sub-20))+1
3 =max(1,max(**maxDepth***(sub-15), **maxDepth***(sub-7))+1)+1
4 =max(1,max(1,1)+1)+1
5 =max(1,2)+1
6 =3

```

根据分析，我们通过递归进行求解代码如下：

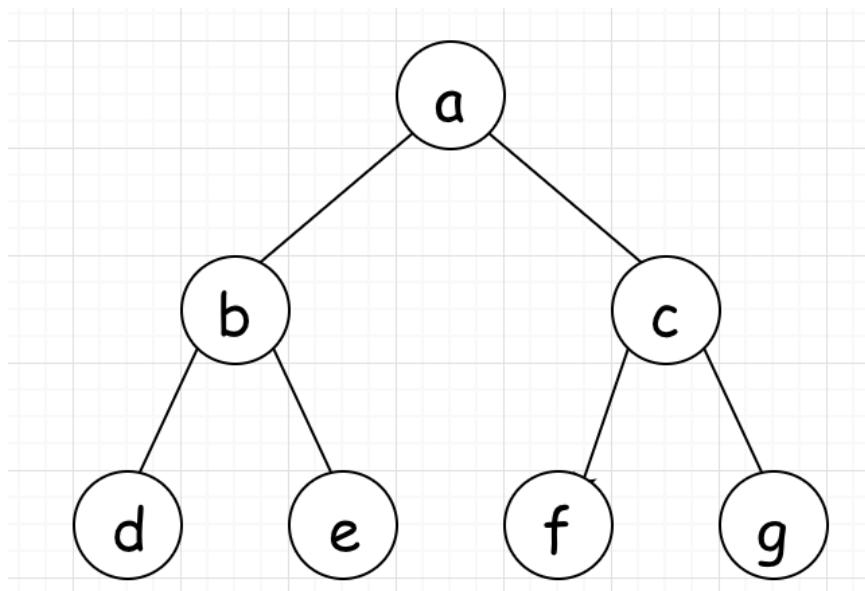
```

1 func maxDepth(root *TreeNode) int {
2     if root == nil {
3         return 0
4     }
5     return max(maxDepth(root.Left), maxDepth(root.Right)) + 1
6 }
7
8 func max(a int, b int) int {
9     if a > b {
10         return a
11     }
12     return b
13 }

```

## 03、DFS

其实我们上面用的递归方式，本质上是使用了DFS的思想。先介绍一下DFS：深度优先搜索算法（Depth First Search），对于二叉树而言，它沿着树的深度遍历树的节点，尽可能深的搜索树的分支，这一过程一直进行到已发现从源节点可达的所有节点为止。



A-B-D-E-C-F-G

A-B-D-E-C-F-G

到这里，我们思考一个问题？虽然我们用递归的方式根据DFS的思想顺利完成了题目。但是这种方式的缺点却显而易见。**因为在递归中，如果层级过深，我们很可能保存过多的临时变量，导致栈溢出。**这也是为什么我们一般不在后台代码中使用递归的原因。如果不理解，下面我们详细说明：

事实上，函数调用的参数是通过栈空间来传递的，在调用过程中会**占用线程的栈资源**。而递归调用，**只有走到最后的结束点后函数才能依次退出**，而未到达最后的结束点之前，占用的栈空间一直没有释放，如果递归调用次数过多，就可能导致占用的栈资源超过线程的最大值，从而导致栈溢出，导致程序的异常退出。

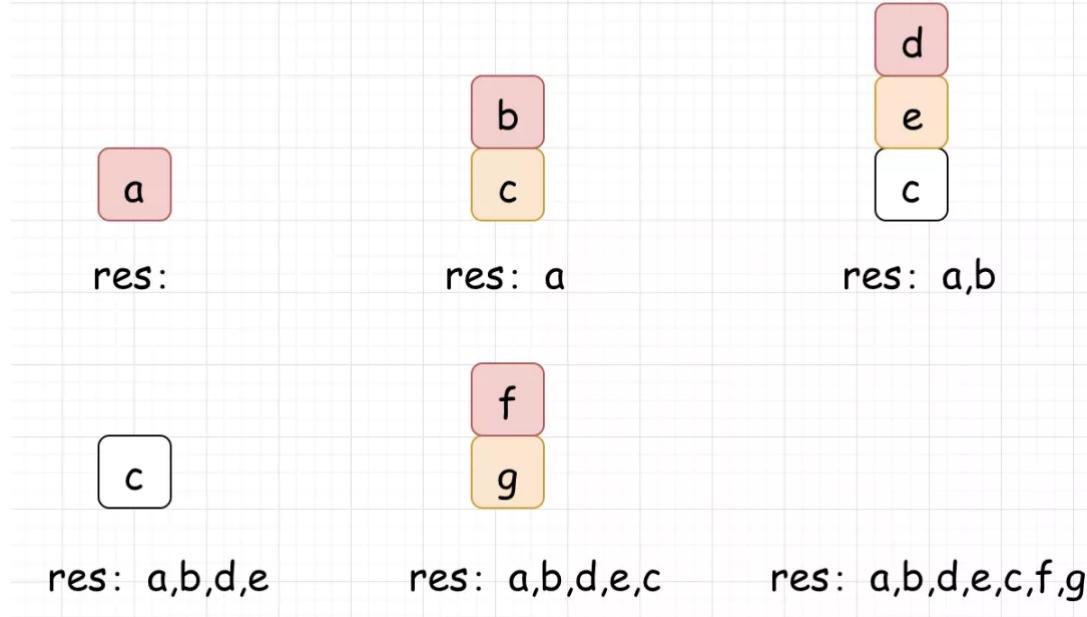
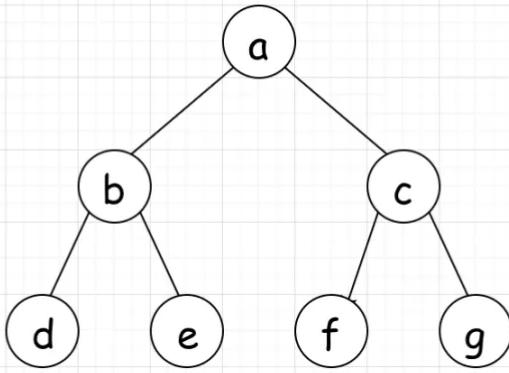
所以，我们引出下面的话题：如何将递归的代码转化成非递归的形式。这里请记住，**99%的递归转非递归，都可以通过栈来进行实现。**

非递归的DFS，代码如下：

```
1 private List<TreeNode> traversal(TreeNode root) {
2     List<TreeNode> res = new ArrayList<>();
3     Stack<TreeNode> stack = new Stack<>();
4     stack.add(root);
5     while (!stack.empty()) {
6         TreeNode node = stack.peek();
7         res.add(node);
8         stack.pop();
9         if (node.right != null) {
10             stack.push(node.right);
11         }
12         if (node.left != null) {
13             stack.push(node.left);
14         }
15     }
16     return res;
17 }
```

上面的代码，唯一需要强调的是，为什么需要先右后左压入数据？是因为我们需要将先访问的数据，后压入栈（请思考栈的特点）。

如果不理解代码，请看下图：



1: 首先将a压入栈

2: a弹栈，将c、b压入栈（注意顺序）

3: b弹栈，将e、d压入栈

4: d、e、c弹栈，将g、f压入栈

5: f、g弹栈

至此，非递归的DFS就讲解完毕了。那我们如何通过非递归DFS的方式，来进行本题求解呢？相信已经很简单了，留下课后作业，请自行实践！

## 层次遍历与BFS(102)

在上一节中，我们通过例题学习了二叉树的DFS（深度优先搜索），其实就是在沿着一个方向一直向下遍历。那我们可不可以按照高度一层一层的访问树中的数据呢？当然可以，就是本节中我们要讲的BFS（宽度优先搜索），同时也被称为广度优先搜索。

我们仍然通过例题进行讲解。

### 01、题目分析

## 第102题：二叉树的层次遍历

给定一个二叉树，返回其按层次遍历的节点值。（即逐层地，从左到右访问所有节点）。

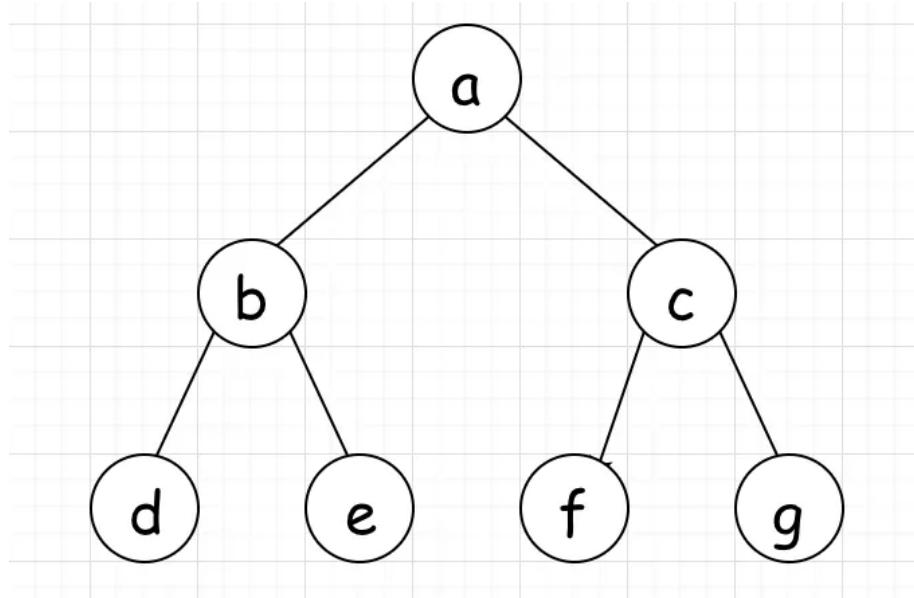
示例：

```
1 | 给定二叉树 [3,9,20,null,null,15,7],  
2 |     3  
3 |     / \  
4 |     9   20  
5 |     /   \  
6 |     15   7  
7 | 返回其层次遍历结果: [[3],[9,20],[15,7]]
```

本系列内容均为必须掌握！

## 02、BFS介绍

BFS，广度/宽度优先。其实就是从上到下，先把每一层遍历完之后再遍历下一层。假如我们的树如下：



按照BFS，访问顺序如下：

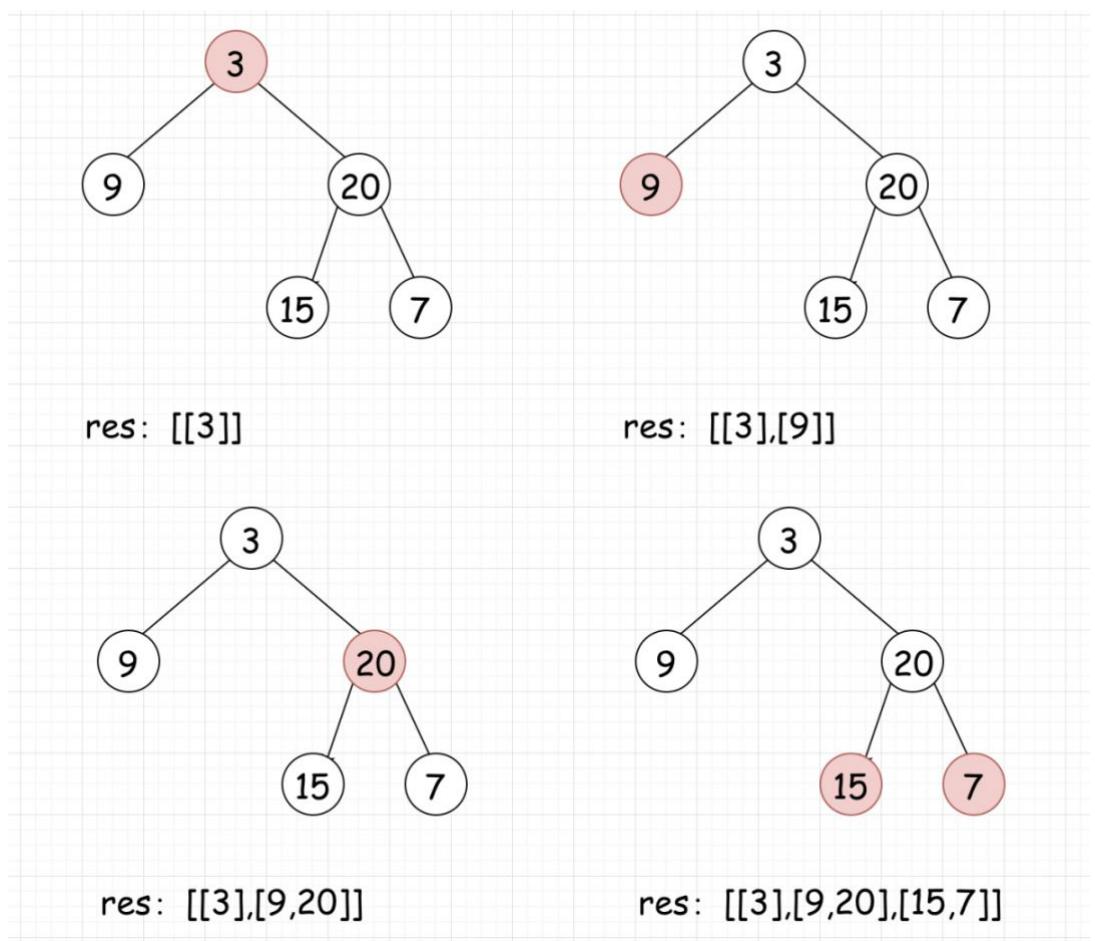
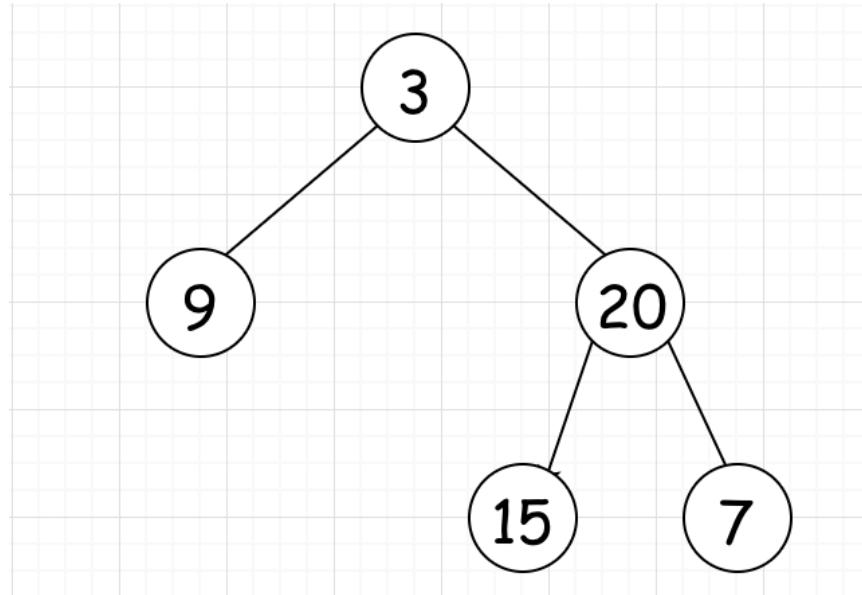
a->b->c->d->e->f->g

了解了BFS，我们开始对本题进行分析。

## 03、递归求解

同样，我们先考虑本题的递归解法。想到递归，我们一般先想到DFS。我们可以对该二叉树进行**先序遍历（根左右的顺序）**，同时，记录节点所在的层次level，并且对每一层都定义一个数组，然后将访问到的节点值放入对应层的数组中。

假设给定二叉树为[3,9,20,null,null,15,7]，图解如下：



根据以上分析，代码如下：

```
1 func levelOrder(root *TreeNode) [][]int {
2     return dfs(root, 0, [][]int{})
3 }
4
5 func dfs(root *TreeNode, level int, res [][][]int) [][]int {
6     if root == nil {
7         return res
8     }
9     if len(res) == level {
10        res = append(res, []int{root.Val})
11    }
12    if root.Left != nil {
13        res = dfs(root.Left, level+1, res)
14    }
15    if root.Right != nil {
16        res = dfs(root.Right, level+1, res)
17    }
18    return res
19 }
```

```

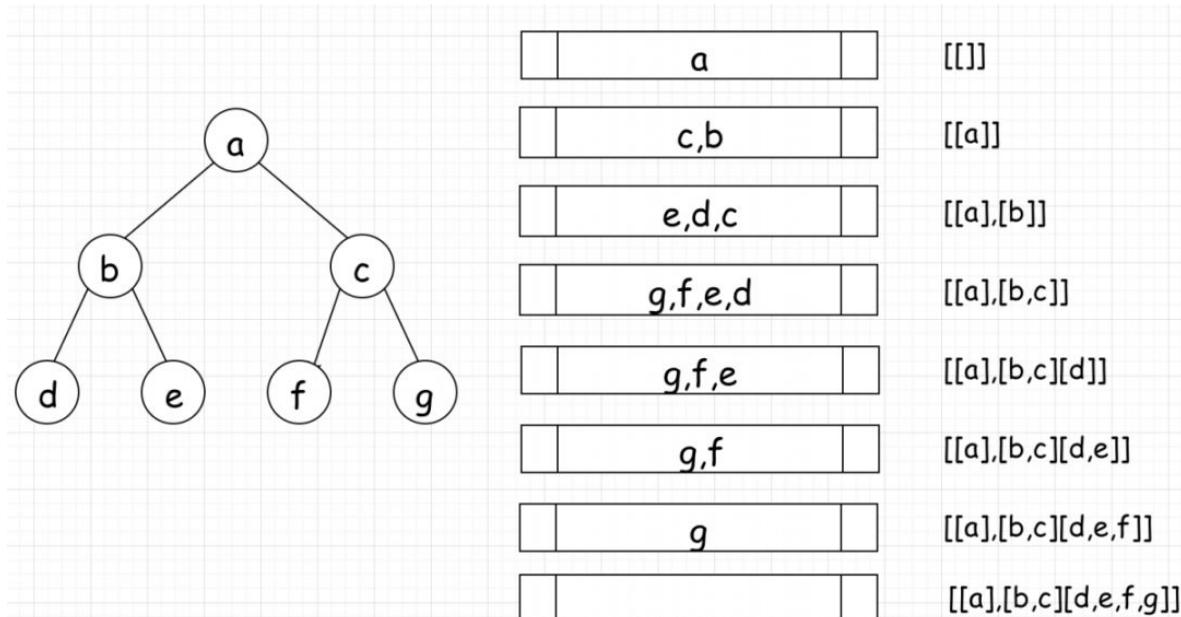
11     } else {
12         res[level] = append(res[level], root.Val)
13     }
14     res = dfs(root.Left, level+1, res)
15     res = dfs(root.Right, level+1, res)
16
17 } 

```

## 04、BFS求解

上面的解法，其实相当于是用DFS的方法实现了二叉树的BFS。那我们能不能直接使用BFS的方式进行解题呢？当然，我们可以使用Queue的数据结构。我们将root节点初始化进队列，通过**消耗尾部，插入头部**的方式来完成BFS。

具体步骤如下图：



根据以上分析，代码如下：

```

1 func levelOrder(root *TreeNode) [][]int {
2     var result [][]int
3     if root == nil {
4         return result
5     }
6     // 定义一个双向队列
7     queue := list.New()
8     // 头部插入根节点
9     queue.PushFront(root)
10    // 进行广度搜索
11    for queue.Len() > 0 {
12        var current []int
13        listLength := queue.Len()
14        for i := 0; i < listLength; i++ {
15            // 消耗尾部
16            // queue.Remove(queue.Back()).(*TreeNode): 移除最后一个元素并将其转
化为TreeNode类型
17            node := queue.Remove(queue.Back()).(*TreeNode)

```

```

18         current = append(current, node.Val)
19         if node.Left != nil {
20             //插入头部
21             queue.PushFront(node.Left)
22         }
23         if node.Right != nil {
24             queue.PushFront(node.Right)
25         }
26     }
27     result = append(result, current)
28 }
29 return result
30 }
```

## BST与其验证(98)

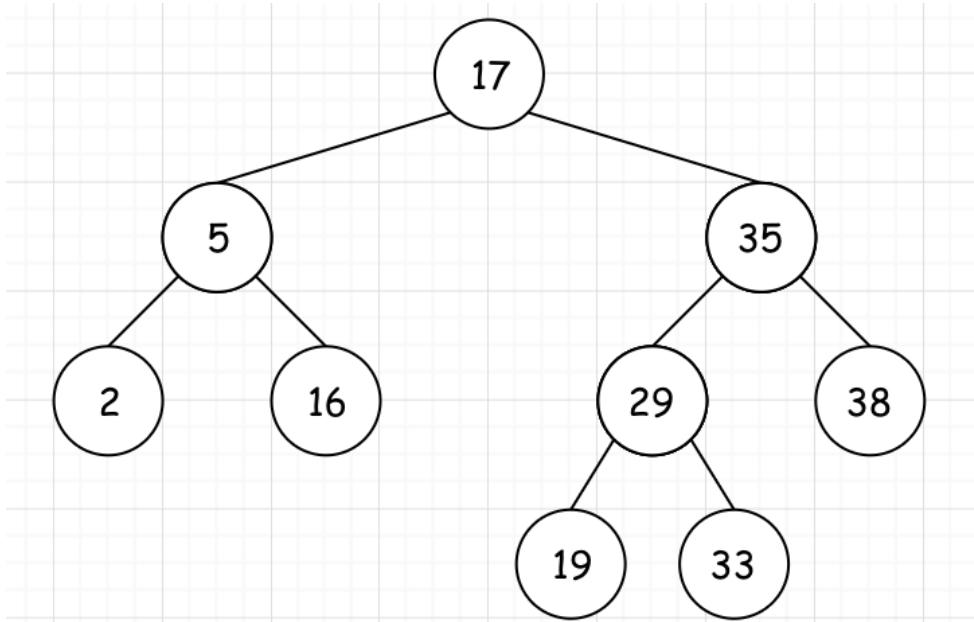
在上一节中，我们分别学习了DFS与BFS。在本节中，我们将继续学习一种特殊的二叉树结构——二叉搜索树（BST）。

### 01、二叉搜索树

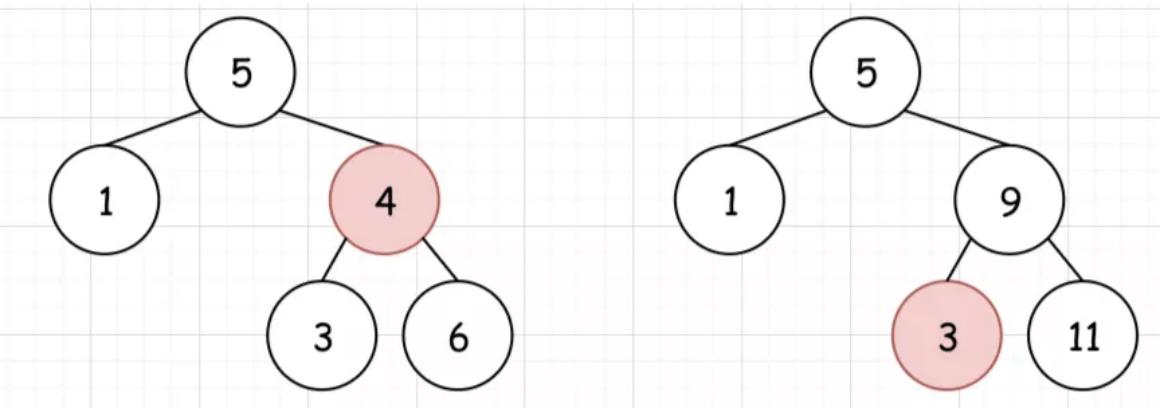
先看定义：二叉搜索树（Binary Search Tree），（又：二叉查找树，二叉排序树）它或者是一棵空树，或者是具有下列性质的二叉树：若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；它的左、右子树也分别为二叉搜索树。

这里强调一下子树的概念：设T是有根树，a是T中的一个顶点，由a以及a的所有后裔（后代）导出的子图称为有向树T的子树。具体来说，子树就是树的其中一个节点以及其下面的所有节点所构成的树。

比如下面这就是一颗二叉搜索树：



下面这两个都不是：



<1>图中4节点位置的数值应该大于根节点

<2>图中3节点位置的数值应该大于根节点

那我们如何来验证一颗二叉搜索树？我们看题。

## 02、题目分析

### 第98题：验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

#### 示例 1：

```

1 | 输入：
2 |   5
3 |   / \
4 |   1   4
5 |   / \
6 |   3   6
7 | 输出: false
8 | 解释：输入为：[5,1,4,null,null,3,6]。
9 |       根节点的值为 5 ，但是其右子节点值为 4 。

```

#### 示例 2：

```

1 | 输入：
2 |   5
3 |   / \
4 |   1   4
5 |   / \
6 |   3   6
7 | 输出: false
8 | 解释：输入为：[5,1,4,null,null,3,6]。
9 |       根节点的值为 5 ，但是其右子节点值为 4 。

```

首先看完题目，我们很容易想到遍历整棵树，比较所有节点，通过左节点值<节点值，右节点值>节点值的方式来进行求解。但是这种解法是错误的，因为对于任意一个节点，我们不光需要左节点值小于该节点，并且左子树上的所有节点值都需要小于该节点。（右节点一致）所以我们在此引入上界与下界，用以保存之前的节点中出现的最大值与最小值。

## 03、递归求解

明确了题目，我们直接使用递归进行求解。这里需要强调的是，在每次递归中，我们除了进行左右节点的校验，还需要与上下界进行判断。由于该递归分析有一定难度，所以我们先展示代码：

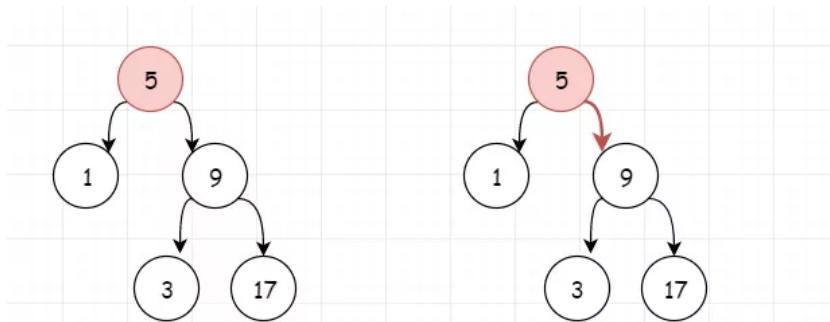
```
1 func isValidBST(root *TreeNode) bool {
2     if root == nil{
3         return true
4     }
5     return isBST(root,math.MinInt64,math.MaxInt64)
6 }
7
8 func isBST(root *TreeNode,min, max int) bool{
9     if root == nil{
10        return true
11    }
12    if min >= root.Val || max <= root.Val{
13        return false
14    }
15    return isBST(root.Left,min,root.Val) && isBST(root.Right,root.Val,max)
16 }
```

运行结果：

执行用时：5 ms，在所有 Go 提交中击败了 95.38 的用户

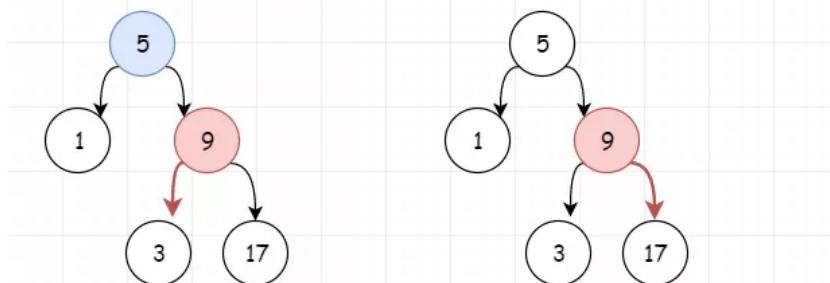
内存消耗：5.4 MB，在所有 Go 提交中击败了 77.00% 的用户

如果觉得上文中的递归不太容易理解，可以通过下图理解：



`isBST (root, min, max)  
isBST (5, -99999, 99999)`

`isBST (root.Right, root.Val, max)  
isBST (9, 5, 99999)`



`isBST (root.Left, min, root.Val)  
isBST (3, 5, 9)`

`isBST (root.Right, root.Val, max)  
isBST (17, 9, 99999)`

**注：该图略掉了对根节点左子树的分析过程**

## BST 的查找(700)

在上一节中，我们学习了二叉搜索树。那我们如何在二叉搜索树中查找一个元素呢？和普通的二叉树又有何不同？我们将在本节内容中进行学习！

下面我们仍然通过例题进行讲解。

### 01、题目分析

#### 第700题：二叉搜索树中的搜索

给定二叉搜索树 (BST) 的根节点和一个值。你需要在 BST 中找到节点值等于给定值的节点。返回以该节点为根的子树。如果节点不存在，则返回 NULL。

**示例：**

```

1 | 给定二叉搜索树：
2 |
3 |     4
4 |     / \
5 |     2   7
6 |     / \
7 |     1   3
8 |
9 | 和值： 2

```

你应该返回如下子树：

1		2
2		/ \
3		1    3

在上述示例中，如果要找的值是 5，但因为没有节点值为 5，我们应该返回 NULL。

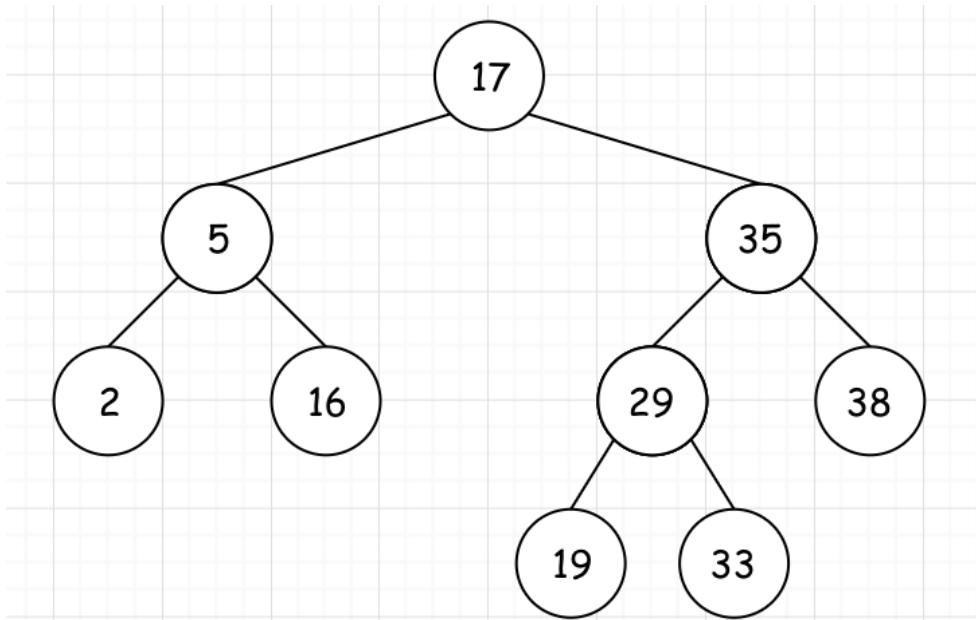
本题为必须掌握! 需要非常熟悉  
为后续题目打基础!

## 02、复习巩固

先复习一下，二叉搜索树（BST）的特性：

- 若它的左子树不为空，则所有左子树上的值均小于其根节点的值
- 若它的右子树不为空，则所有右子树上的值均大于其根节点得值
- 它的左右子树也分别为二叉搜索树

如下图就是一棵典型的BST：

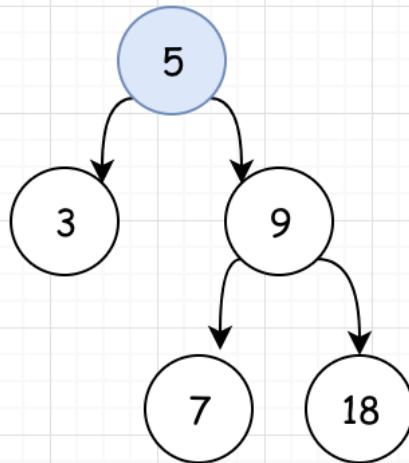


## 03、图解分析

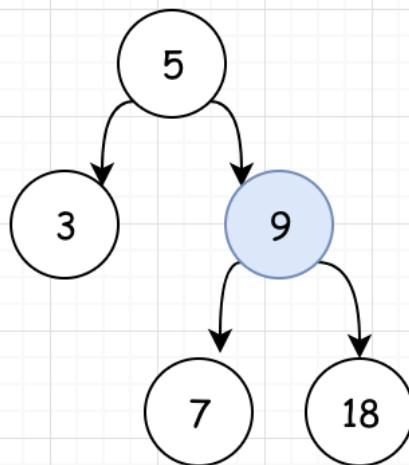
假设目标值为 val，根据BST的特性，我们可以很容易想到查找过程

- 如果val小于当前结点的值，转向其左子树继续搜索；
- 如果val大于当前结点的值，转向其右子树继续搜索；
- 如果已找到，则返回当前结点。

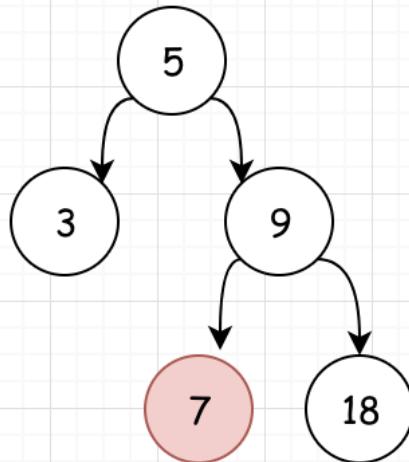
7>5, to Right



7<9, to Left



7=7, done



很简单，不是吗？

## 04、代码示例

给出递归和迭代两种解法：

```
1 //递归
2 public TreeNode searchBST(TreeNode root, int val) {
3     if (root == null)
```

```

4         return null;
5     if (root.val > val) {
6         return searchBST(root.left, val);
7     } else if (root.val < val) {
8         return searchBST(root.right, val);
9     } else {
10        return root;
11    }
12}
13//迭代
14public TreeNode searchBST(TreeNode root, int val) {
15    while (root != null) {
16        if (root.val == val) {
17            return root;
18        } else if (root.val > val) {
19            root = root.left;
20        } else {
21            root = root.right;
22        }
23    }
24    return null;
25}

```

## 递归与迭代的区别

递归：重复调用函数自身实现循环称为递归；

迭代：利用变量的原值推出新值称为迭代，或者说迭代是函数内某段代码实现循环；

## 特别说明

本题确实很简单！专门拉出来讲解的目的有二。第一BST确实很重要，第二希望通过重复，能加深大家对BST的印象，不至于和后面的内容出现交叉感染！

## BST 的删除(450)

在两节中，我们了解了BST（二叉搜索树）的概念，并且知道了如何在BST中查找一个元素。那我们又如何在BST中去删除一个元素呢？我们将通过本节的例题进行学习！

下面我们仍然通过例题进行讲解。

## 01、题目分析

### 第450题：删除二叉搜索树中的节点

给定一个二叉搜索树的根节点 root 和一个值 key，删除二叉搜索树中的 key 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

- 首先找到需要删除的节点；
- 如果找到了，删除它。

说明：要求算法时间复杂度为  $O(h)$ ,  $h$  为树的高度。

示例：

```
1 root = [5,3,6,2,4,null,7]
2 key = 3
3
4     5
5     / \
6     3   6
7     / \   \
8     2   4   7
9
10 给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。
11
12 一个正确的答案是 [5,4,6,2,null,null,7]，如下图所示。
13     5
14     / \
15     4   6
16    /     \
17    2       7
18
19 另一个正确答案是 [5,2,6,null,4,null,7]。
20     5
21     / \
22     2   6
23     \   \
24     4   7
```

强烈建议先学习之前两节内容！

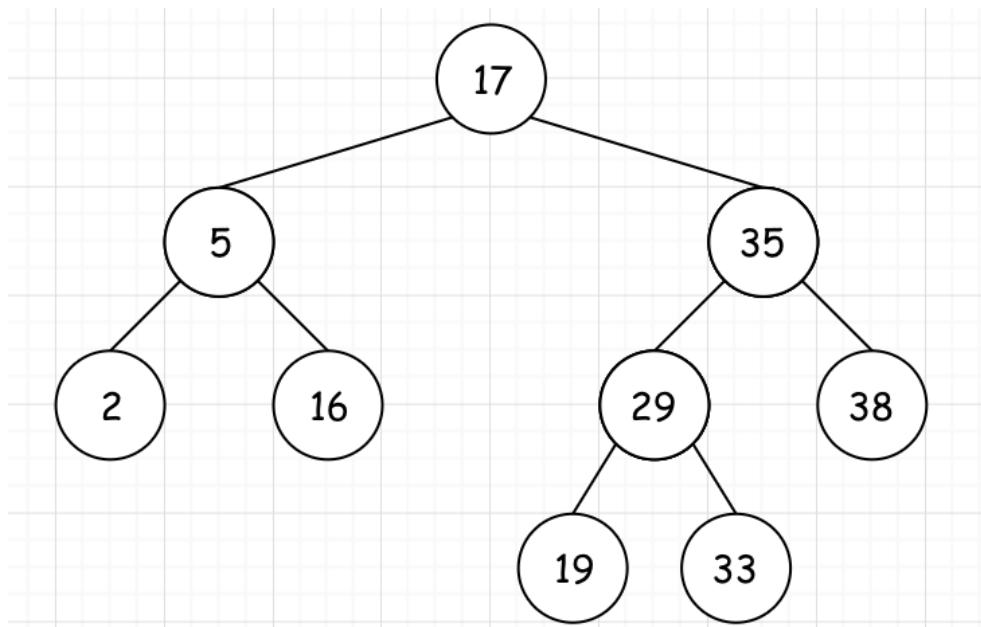
以达到最好的学习效果！

## 02、复习巩固

先复习一下，**二叉搜索树**（BST）的特性：

- 若它的左子树不为空，则所有左子树上的值均小于其根节点的值
- 若它的右子树不为空，则所有右子树上的值均大于其根节点得值
- 它的左右子树也分别为二叉搜索树

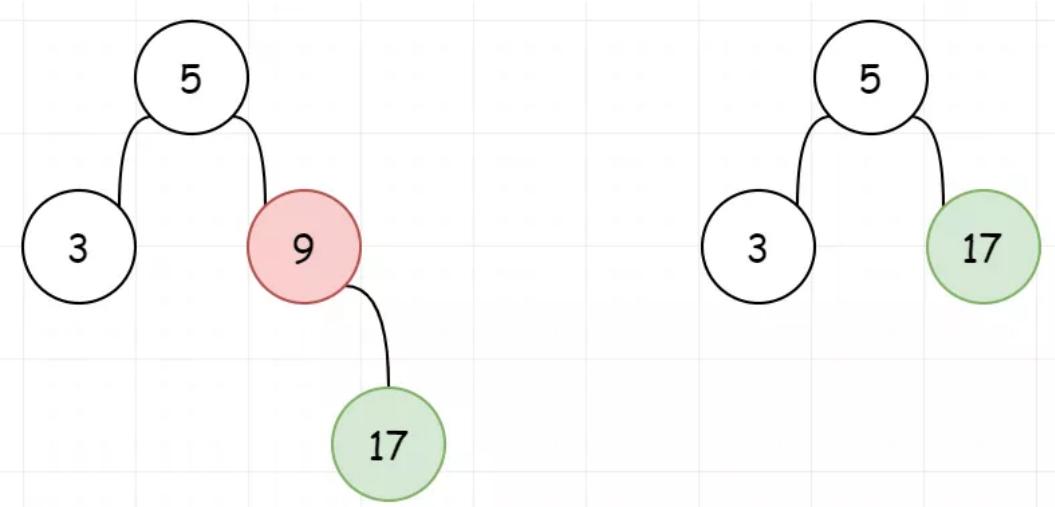
如下图就是一棵典型的BST：



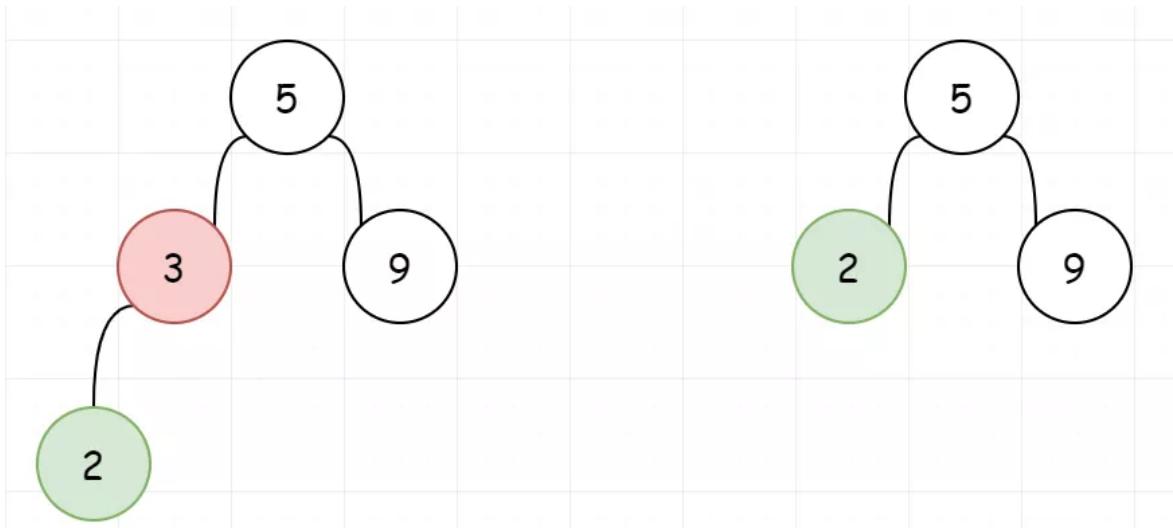
### 03、图解分析

明确了概念，我们进行分析。我们要删除BST的一个节点，首先需要**找到该节点**。而找到之后，会出现三种情况。

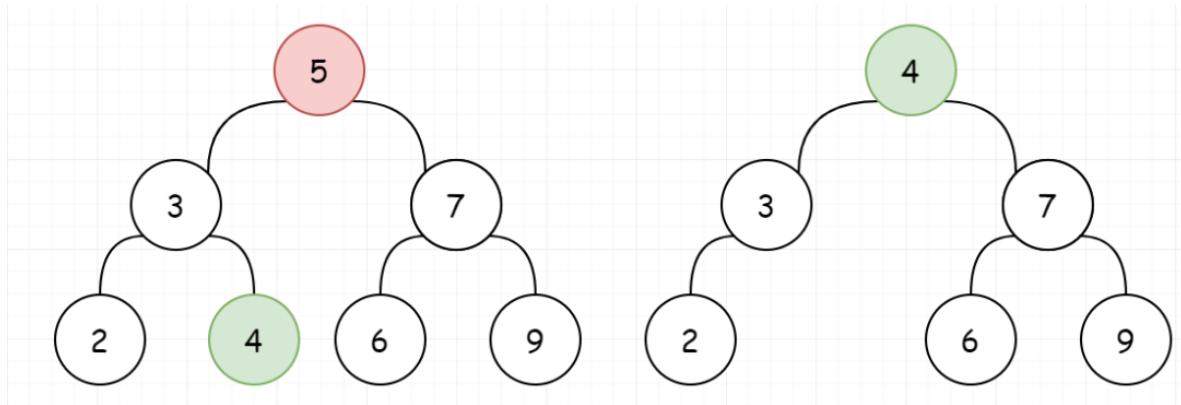
1、待删除的节点左子树为空，让待删除节点的右子树替代自己。



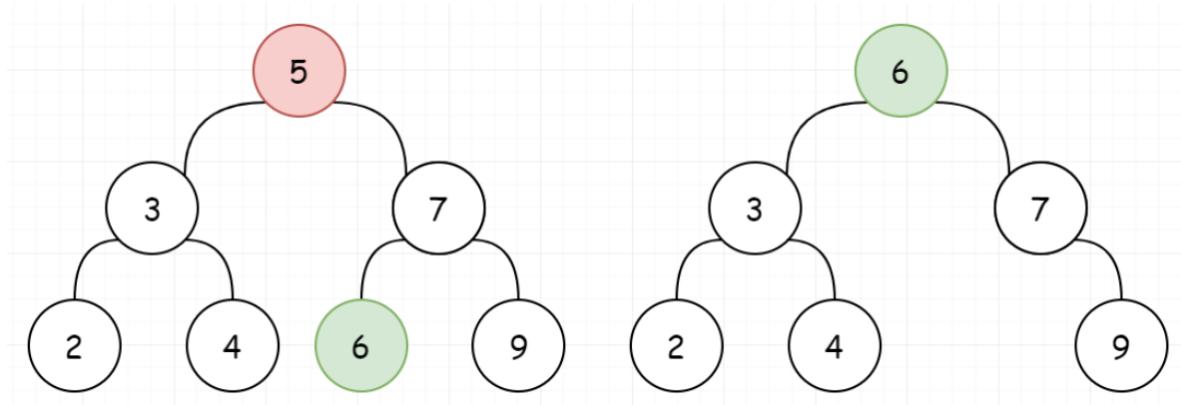
2、待删除的节点右子树为空，让待删除节点的左子树替代自己。



3、如果待删除的节点的左右子树都不为空。我们需要找到**比当前节点小的最大节点（前驱）**，来替换自己



或者**比当前节点大的最小节点（后继）**，来替换自己。



分析完毕，我们一起看代码怎么实现吧。

## 04、GO语言示例

这里我们给出通过**后继节点**来替代自己的方案（请后面自行动手实现另一种方案）：

```

1 func deleteNode(root *TreeNode, key int) *TreeNode {
2     if root == nil {
3         return nil
4     }
5     if key < root.Val {
6         root.Left = deleteNode( root.Left, key )
7     } else if key > root.Val {
8         root.Right = deleteNode( root.Right, key )
9     } else {
10        // 找到后继节点
11        if root.Right != nil {
12            var cur = root.Right
13            for cur.Left != nil {
14                cur = cur.Left
15            }
16            root.Val = cur.Val
17            root.Right = deleteNode( root.Right, cur.Val )
18        } else {
19            // 叶子结点
20            if root.Left != nil {
21                root = root.Left
22            } else {
23                root = nil
24            }
25        }
26    }
27    return root
28}
```

```

7         return root
8     }
9     if key > root.Val {
10        root.Right = deleteNode( root.Right, key )
11        return root
12    }
13    //到这里意味已经查找到目标
14    if root.Right == nil {
15        //右子树为空
16        return root.Left
17    }
18    if root.Left == nil {
19        //左子树为空
20        return root.Right
21    }
22    minNode := root.Right
23    for minNode.Left != nil {
24        //查找后继
25        minNode = minNode.Left
26    }
27    root.Val = minNode.Val
28    root.Right = deleteMinNode( root.Right )
29    return root
30 }
31
32
33 func deleteMinNode( root *TreeNode ) *TreeNode {
34     if root.Left == nil {
35         pRight := root.Right
36         root.Right = nil
37         return pRight
38     }
39     root.Left = deleteMinNode( root.Left )
40     return root
41 }
```

执行结果：

执行用时：16 ms，在所有 Go 提交中击败了 86.52% 的用户

内存消耗：6.7 MB，在所有 Go 提交中击败了 65.52% 的用户

## 平衡二叉树(110)

在之前的系列中，我们已经学习了二叉树的深度以及DFS，如果不会可以先查看之前的文章。今天我们将对其进行应用，直接看题目。

### 01、题目分析

**第110题：平衡二叉树**

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过1。

### 示例 1：

```
1 | 给定二叉树 [3,9,20,null,null,15,7]
2 |
3 |     3
4 |     / \
5 |     9   20
6 |     /   \
7 |     15   7
8 |
9 | 返回 true 。
```

### 示例 2：

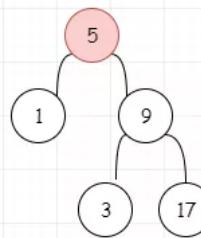
```
1 | 给定二叉树 [1,2,2,3,3,null,null,4,4]
2 |
3 |     1
4 |     / \
5 |     2   2
6 |     / \
7 |     3   3
8 |     / \
9 |     4   4
10|
11| 返回 false 。
```

## 02、图解分析

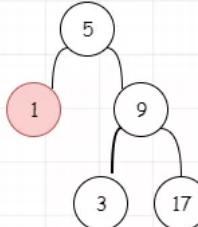
首先分析题目，这道题思路很简单，我们想判断一棵树是否满足平衡二叉树，无非就是判断当前结点的两个孩子是否满足平衡，同时两个孩子的高度差是否超过1。那只要我们可以得到高度，再基于高度进行判断即可。

我们先复习一下之前对于树高度的求解：

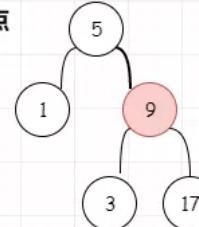
对任意节点高度求解，  
需要对其左右节点的  
高度求解



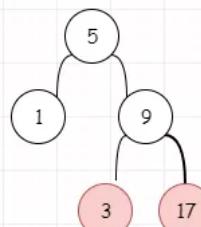
左节点



右节点

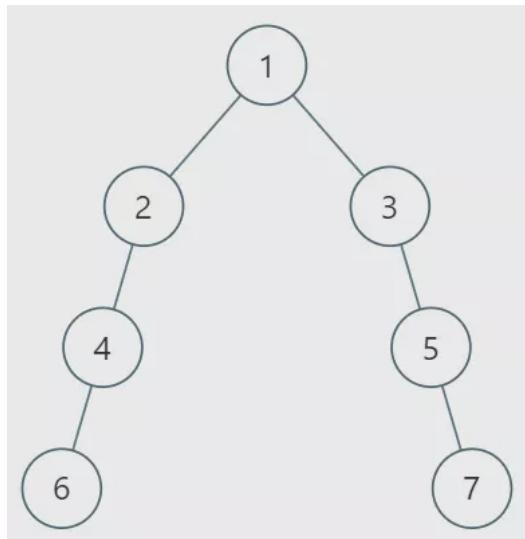


通过递归，  
继续重复上述过程



这里唯一要注意的是，当我们判定其中任意一个节点如果不满足平衡二叉树时，那说明整棵树已经不是一颗平衡二叉树，我们可以对其进行阻断，不需要继续递归下去。

另外，需要注意的是，下面这棵并不是平衡二叉树：



### 03、代码分析

根据分析，逻辑非常清晰，顺利得出代码：

```
1 func isBalanced(root *TreeNode) bool {
2     if root == nil {
3         return true
4     }
5     if !isBalanced(root.Left) || !isBalanced(root.Right) {
6         return false
7     }
8     if abs(height(root.Left), height(root.Right)) > 1 {
9         return false
10    }
11    return true
12 }
```

```

7 }
8 leftH := maxDepth(root.Left) + 1;
9 rightH := maxDepth(root.Right) + 1;
10 if abs(leftH-rightH) > 1 {
11     return false
12 }
13 return true
14 }

15 func maxDepth(root *TreeNode) int {
16     if root == nil {
17         return 0
18     }
19     return max(maxDepth(root.Left),maxDepth(root.Right)) + 1
20 }
21 }

22 func max(a,b int) int {
23     if a > b {
24         return a
25     }
26     return b
27 }

28 }

29 func abs(a int) int {
30     if a < 0 {
31         return -a
32     }
33     return a
34 }

35 }
```

执行结果：

执行用时：4 ms，在所有 Go 提交中击败了 99.76% 的用户

内存消耗：5.7 MB，在所有 Go 提交中击败了 98.50% 的用户

## 完全二叉树(222)

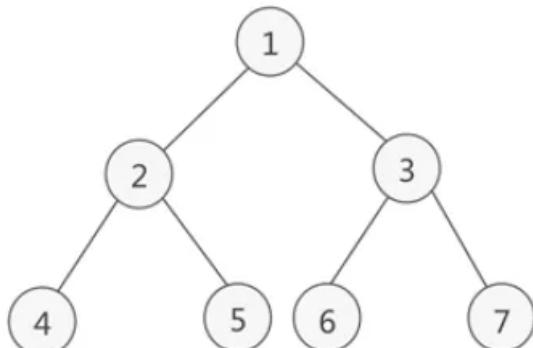
在上一篇中，我们学习了解了**平衡二叉树**，并且利用DFS进行了验证。在本节中，我们将继续学习**完全二叉树**的相关内容。首先了解一下什么是完全二叉树。

### 01、完全二叉树

完全二叉树由满二叉树引出，先来了解一下什么是满二叉树：

如果二叉树中除了叶子结点，每个结点的度都为 2，则此二叉树称为**满二叉树**。（**二叉树的度**代表某个结点的孩子或者说直接后继的个数。对于二叉树而言，1度是只有一个孩子或者说单子树,2度是有两个孩子或者说左右子树都有。）

比如下面这颗：

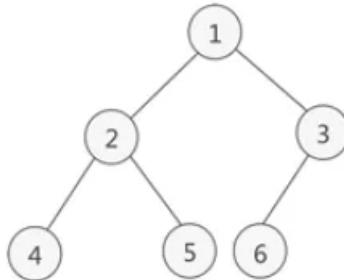


满二叉树示意图

那什么又是完全二叉树呢：

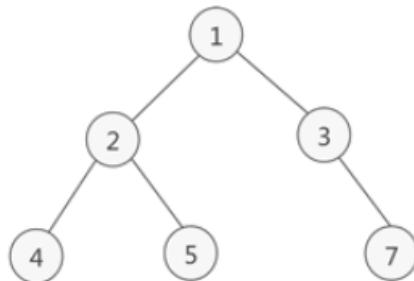
如果二叉树中除去最后一层节点为满二叉树，且最后一层的结点依次从左到右分布，则此二叉树被称为完全二叉树。

比如下面这颗：



a ) 完全二叉树

而这颗就不是：



b ) 非完全二叉树

熟悉了概念，我们还是一起来看题目吧。

## 02、题目分析

第222题：完全二叉树的节点个数

给出一个完全二叉树，求出该树的节点个数。

### 说明:

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第  $h$  层，则该层包含  $1 \sim 2^h$  个节点。

### 示例:

```
1 | 输入：
2 |     1
3 |     / \
4 |     2   3
5 |     / \   /
6 |     4   5   6
7 |
8 | 输出： 6
```

## 03、递归求解

首先分析题目，我们很容易可以想到通过递归，来求解节点个数。

```
1 | func countNodes(root *TreeNode) int {
2 |     if root != nil {
3 |         return 1 + countNodes(root.Right) + countNodes(root.Left)
4 |     }
5 |     return 1 + countNodes(root.Right) + countNodes(root.Left)
6 | }
```

执行结果：

执行用时：**20 ms**，在所有 Go 提交中击败了 **78.79%** 的用户

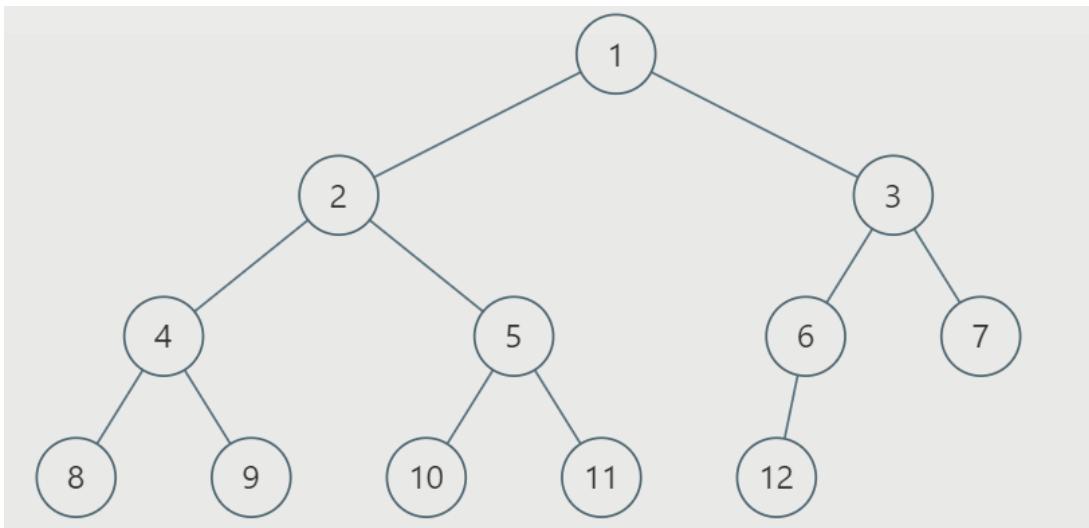
内存消耗：**6.6 MB**，在所有 Go 提交中击败了 **57.38%** 的用户

但是很明显，出题者肯定不是要这种答案。因为这种答案和完全二叉树一毛钱关系都没有。所以我们继续思考。

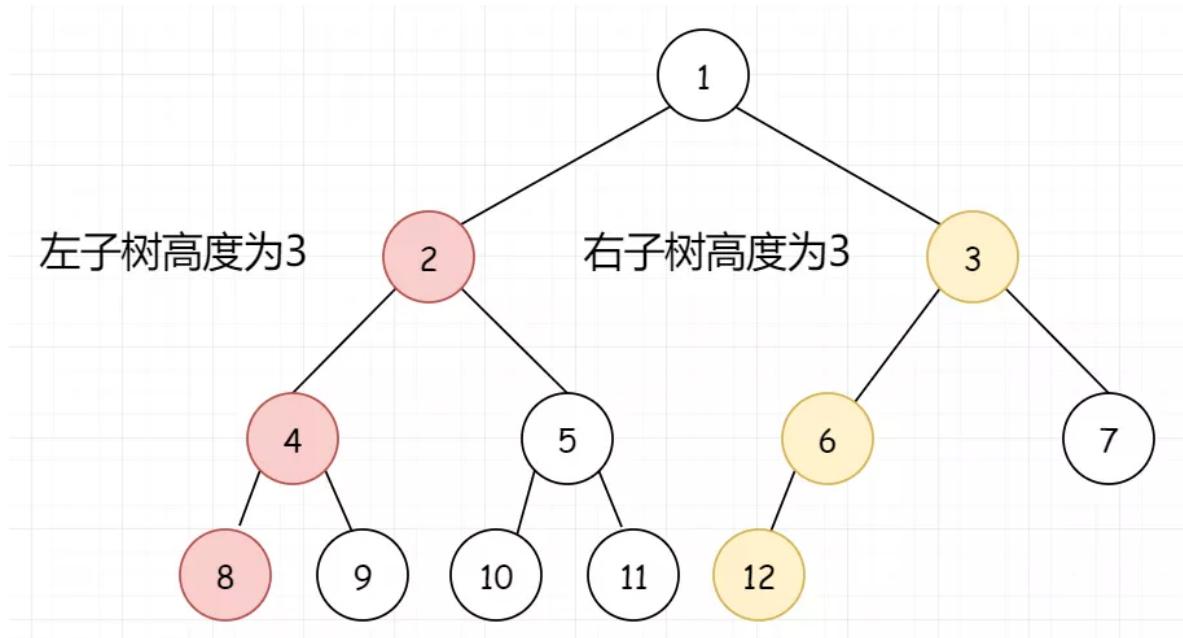
## 04、经典解法

由于题中已经告诉我们这是一颗完全二叉树，我们又已知了完全二叉树除了最后一层，其他层都是满的，并且最后一层的节点全部靠向了左边。那我们可以想到，可以将该完全二叉树可以分割成**若干满二叉树和完全二叉树**，**满二叉树直接根据层高h计算出节点为2^h-1**，然后**继续计算子树中完全二叉树节点**。那如何分割成若干满二叉树和完全二叉树呢？**对任意一个子树，遍历其左子树层高left，右子树层高right，相等左子树则是满二叉树，否则右子树是满二叉树**。这里可能不容易理解，我们看图。

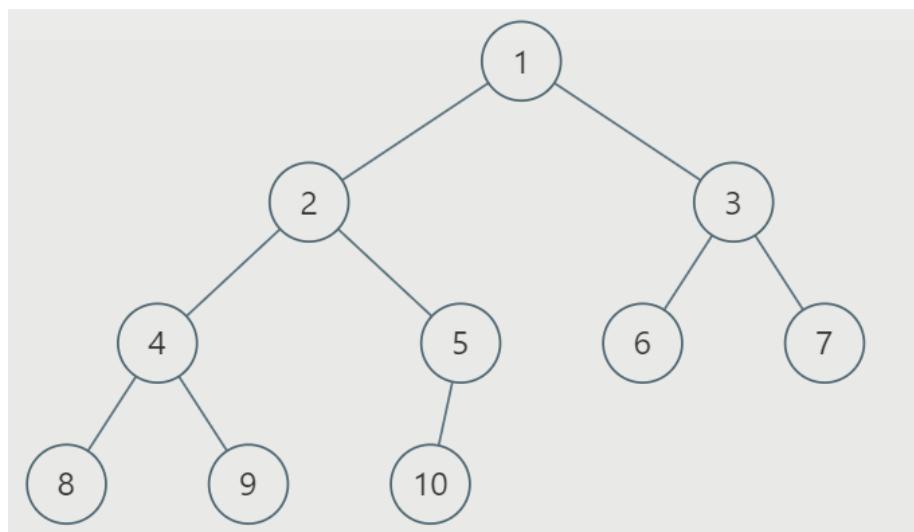
假如我们有树如下：



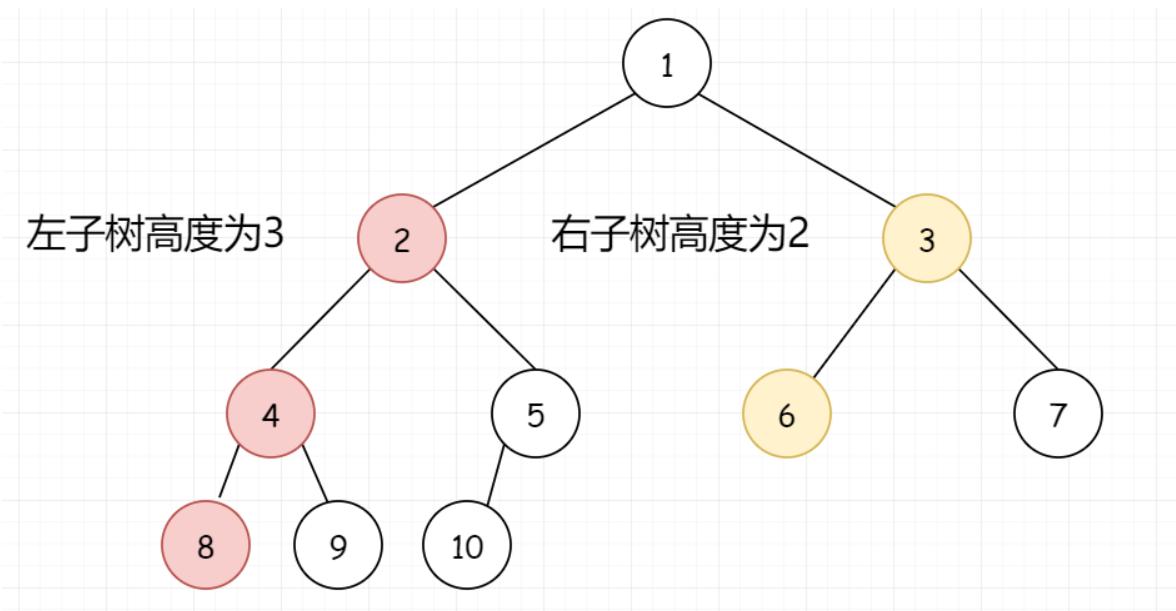
我们看到根节点的左右子树高度都为3，那么说明左子树是一颗满二叉树。因为节点已经填充到右子树了，左子树必定已经填满了。所以左子树的节点总数我们可以直接得到，是 $2^{\text{left}} - 1$ ，加上当前这个root节点，则正好是 $2^3$ ，即8。然后只需要再对右子树进行递归统计即可。



那假如我们的树是这样：



我们看到左子树高度为3，右子树高度为2。说明此时最后一层不满，但倒数第二层已经满了，可以直接得到右子树的节点个数。同理，右子树节点+root节点，总数为 $2^{\text{right}}$ ，即 $2^2$ 。再对左子树进行递归查找。



根据分析，得出代码：

```

1 class Solution {
2     public int countNodes(TreeNode root) {
3         if (root == null) {
4             return 0;
5         }
6         int left = countLevel(root.left);
7         int right = countLevel(root.right);
8         if (left == right) {
9             return countNodes(root.right) + (1 << left);
10        } else {
11            return countNodes(root.left) + (1 << right);
12        }
13    }
14
15    private int countLevel(TreeNode root) {
16        int level = 0;
17        while (root != null) {
18            level++;
19            root = root.left;
20        }
21        return level;
22    }
23 }
```

运行结果：

执行用时：0 ms，在所有 Java 提交中击败了 100.00% 的用户

内存消耗：40.3 MB，在所有 Java 提交中击败了 70.13% 的用户

## 二叉树的剪枝(814)

在之前的系列中。我们学习了DFS、BFS，也熟悉了平衡二叉树，满二叉树，完全二叉树，BST（二叉搜索树）等概念。在本节中，我们将学习一种二叉树中常用的操作——剪枝。这里额外说一点，就本人而言，对这个操作以及其衍化形式的使用会比较频繁。因为我是做规则引擎的，在规则引擎中，我们会有一个概念叫做决策树，那如果一颗决策树完全生长，就会带来比较大的过拟合问题。因为完全生长的决策树，每个节点只会包含一个样本。所以我们就需要对决策树进行剪枝操作，来提升整个决策模型的泛化能力（ML概念）… 听不懂也没关系，简单点讲，就是我觉得这个很重要，或者每道算法题都很重要。如果你在工作中没有用到，不是说明算法不重要，而可能是你还不够重要。

## 01、剪枝概述

假设有一棵树，最上层的是root节点，而父节点会依赖子节点。如果现在有一些节点已经标记为无效，我们要删除这些无效节点。如果无效节点的依赖的节点还有效，那么不应该删除，如果无效节点和它的子节点都无效，则可以删除。剪掉这些节点的过程，称为剪枝，目的是用来处理二叉树模型中的依赖问题。

我们还是通过一道题目来进行具体学习。

## 02、题目分析

### 第814题：二叉树的剪枝

给定二叉树根结点 root，此外树的每个结点的值要么是 0，要么是 1。返回移除了所有不包含 1 的子树的原二叉树。（节点 X 的子树为 X 本身，以及所有 X 的后代。）

示例1：

```
1 | 输入： [1,null,0,0,1]
2 | 输出： [1,null,0,null,1]
```

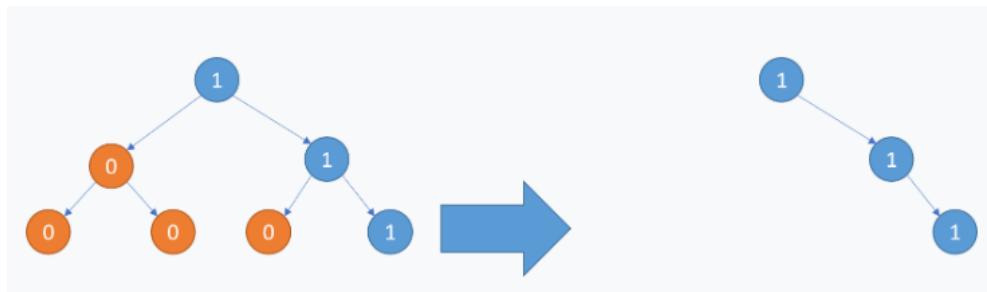


解释：

- 只有红色节点满足条件“所有不包含 1 的子树”。
- 右图为返回的答案。

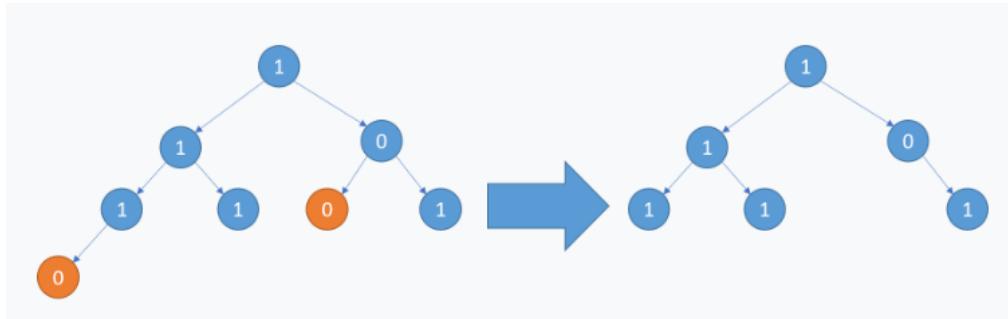
示例2：

```
1 | 输入： [1,0,1,0,0,0,1]
2 | 输出： [1,null,1,null,1]
```



示例3：

1	输入： [1,1,0,1,1,0,1,0]
2	输出： [1,1,0,1,1,null,1]



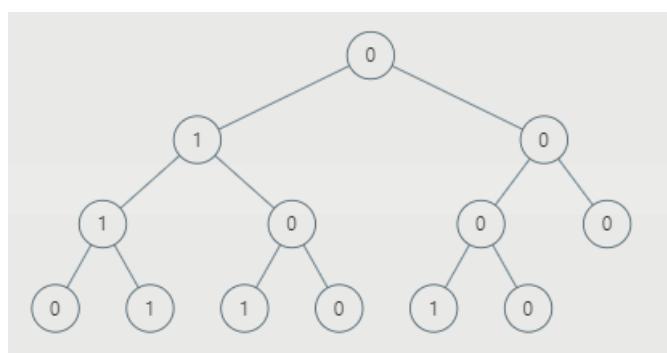
说明：

- 给定的二叉树最多有 100 个节点。
- 每个节点的值只会为 0 或 1。

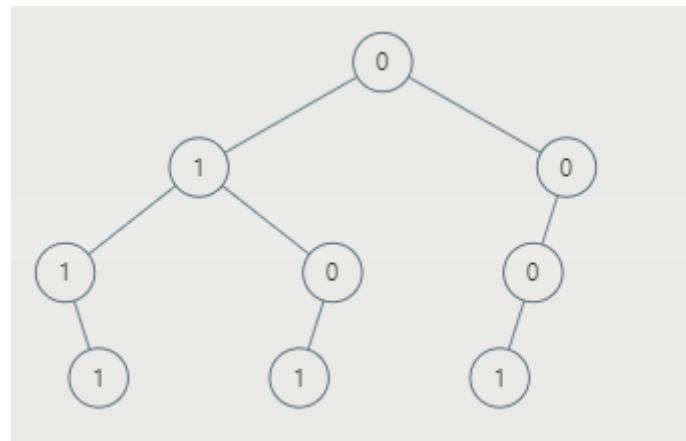
### 03、递归求解

二叉树的问题，大多都可以通过递归进行求解。我们直接进行分析。假设我们有二叉树如下：  
[0,1,0,1,0,0,0,1,1,0,1,0]

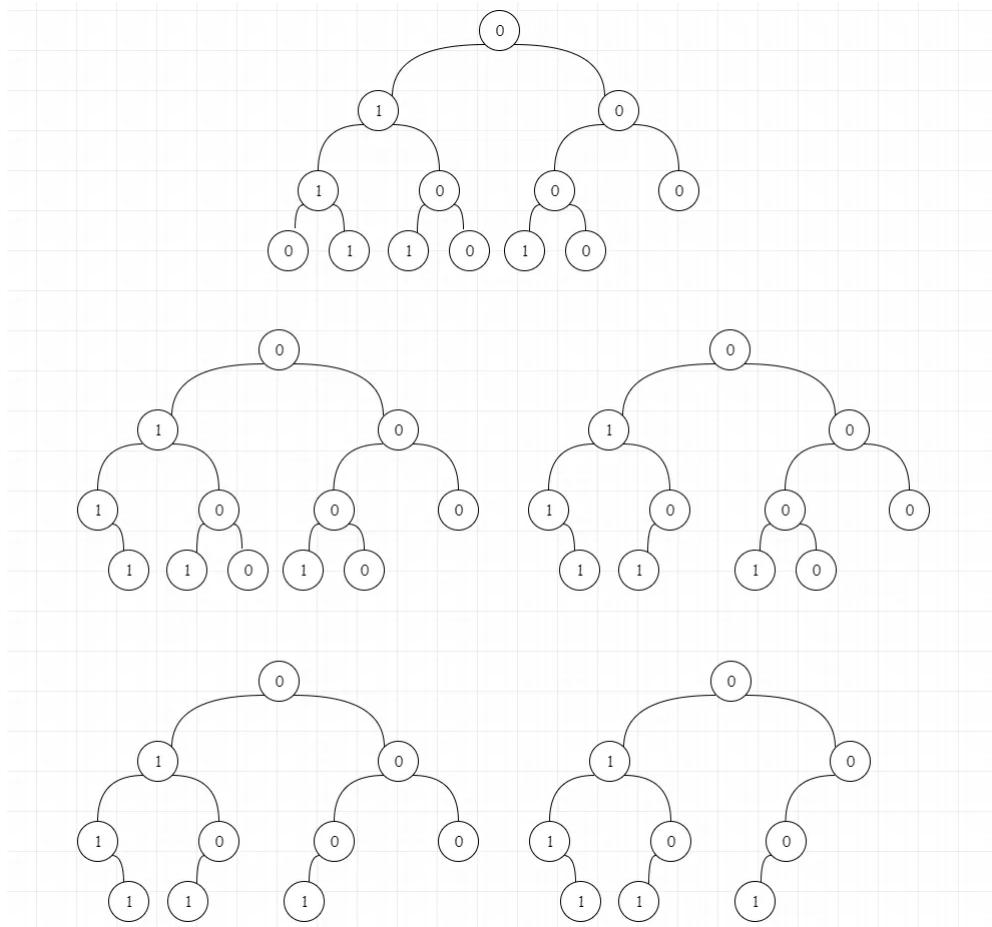
就是下面这样：



剪枝之后是这样：



剪什么大家应该都能理解。那关键是怎么剪？过程也很简单，在递归的过程中，如果当前结点的左右节点皆为空，且当前结点为0，我们就将当前节点剪掉即可。



根据分析，很自然得出代码：

```

1 func pruneTree(root *TreeNode) *TreeNode {
2     return deal(root)
3 }
4
5 func deal(node *TreeNode) *TreeNode {
6     if node == nil {
7         return nil
8     }
9     node.Left = deal(node.Left)
10    node.Right = deal(node.Right)
11    if node.Left == nil && node.Right == nil && node.Val == 0 {
12        return nil
13    }

```

```
14     return node  
15 }
```

运行结果：

执行用时：**0 ms**，在所有 Go 提交中击败了 **100.00%** 的用户

内存消耗：**2.3 MB**，在所有 Go 提交中击败了 **70.83%** 的用户

我一度认为算法题的意义绝不简简单单只是为了面试，一度想把这种观念传递给认识我的朋友们。至少在工作中，对于栈，优先队列，红黑树，图等知识或多或少都是有机会能用到。不记得是不是李开复说的了，**算法是提高智商少数为之可行的手段**。真心希望大家可以在这个过程中为之成长，我们大家一起努力啊。

## 滑动窗口系列

### 滑动窗口最大值（239）

有读者小伙伴建议讲一下**滑动窗口**相关题型，因为经常面试会被问到。所以就开了这个系列（所以如果大家有想让分享的题型都可以留言区告诉我，任何事情我觉得都需要有反馈。比如一个错误，你不反馈，我不知道..那就只能这样过去了..）闲话不啰嗦，直接看题！

#### 01、题目分析

##### 第239题：滑动窗口最大值

给定一个数组  $\text{nums}$ ，有一个大小为  $k$  的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的  $k$  个数字。滑动窗口每次只向右移动一位。返回滑动窗口中的最大值。

给定一个数组  $\text{nums}$ ，有一个大小为  $k$  的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的  $k$  个数字。滑动窗口每次只向右移动一位。

返回滑动窗口中的最大值**所构成的数组**。

示例：

```

1 | 输入: nums = [1,3,-1,-3,5,3,6,7], 和 k = 3
2 | 输出: [3,3,5,5,6,7]
3 | 解释:
4 |
5 | 滑动窗口的位置           最大值
6 |   --
7 | [1 3 -1] -3 5 3 6 7      3
8 | 1 [3 -1 -3] 5 3 6 7      3
9 | 1 3 [-1 -3 5] 3 6 7      5
10 | 1 3 -1 [-3 5 3] 6 7      5
11 | 1 3 -1 -3 [5 3 6] 7      6
12 | 1 3 -1 -3 5 [3 6 7]      7

```

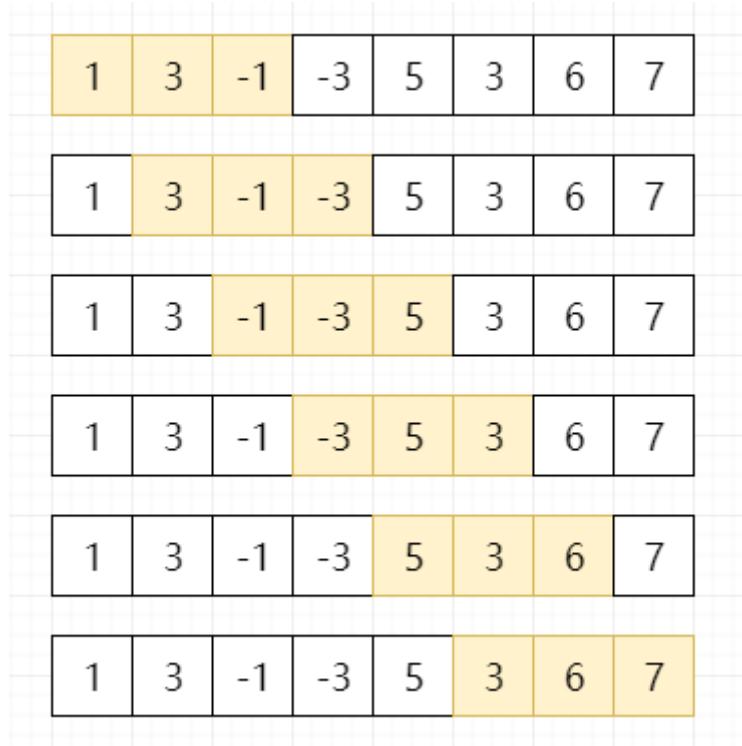
本题有一定难度！建议认真阅读

并课后练习~

## 02、题目分析

本题对于题目没有太多需要额外说明的，应该都能理解，直接进行分析。我们很容易想到，可以通过遍历所有的滑动窗口，找到每一个窗口的最大值，来进行暴力求解。那一共有多少个滑动窗口呢，小学题目，可以得到共有  $L-k+1$  个窗口。

假设  $\text{nums} = [1,3,-1,-3,5,3,6,7]$ ，和  $k = 3$ ，窗口数为6



根据分析，直接完成代码：

```

1 | class Solution {
2 |     public int[] maxSlidingWindow(int[] nums, int k) {
3 |         int len = nums.length;
4 |         if (len * k == 0) return new int[0];
5 |         int [] win = new int[len - k + 1];

```

```

6     //遍历所有的滑动窗口
7     for (int i = 0; i < len - k + 1; i++) {
8         int max = Integer.MIN_VALUE;
9         //找到每一个滑动窗口的最大值
10        for(int j = i; j < i + k; j++) {
11            max = Math.max(max, nums[j]);
12        }
13        win[i] = max;
14    }
15    return win;
16 }
17 }
```

运行结果：

执行用时：**67 ms**，在所有 Java 提交中击败了 **6.89%** 的用户

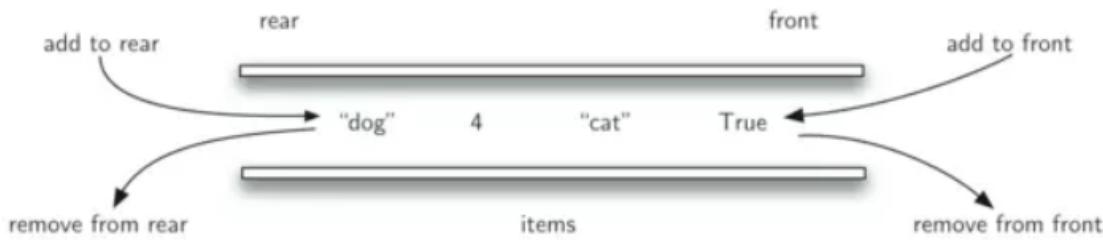
内存消耗：**56 MB**，在所有 Java 提交中击败了 **5.04%** 的用户

It's Bullshit! 结果令我们很不满意，时间复杂度达到了 $O(LK)$ ，如果面试问到这道题，基本上只写出这样的代码，一定就挂掉了。那我们怎么样优化时间复杂度呢？有没有可以 $O(L)$ 的实现呢？=\_=

## 03、线性题解

这里不卖关子，其实这道题比较经典，我们可以采用队列，DP，堆等方式进行求解，所有思路的主要源头应该都是在窗口滑动的过程中，如何更快的完成查找最大值的过程。但是最典型的解法还是使用双端队列。具体怎么来求解，一起看一下。

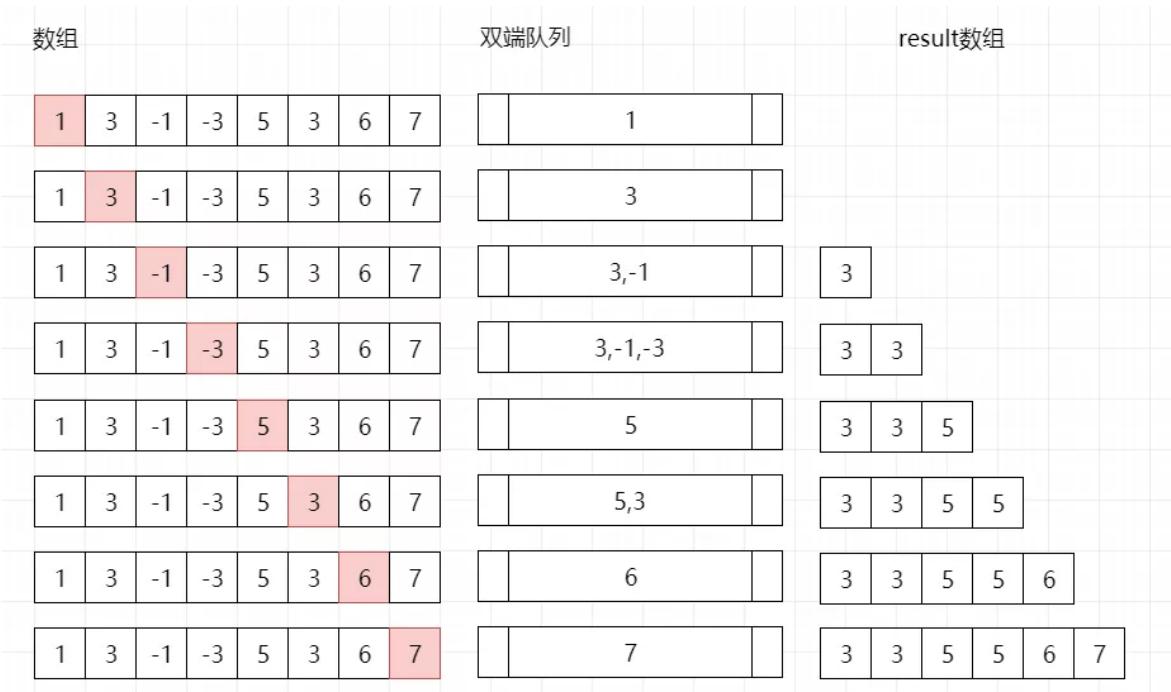
首先，我们了解一下，什么是双端队列：是一种具有队列和栈的性质的数据结构。双端队列中的元素可以从两端弹出或者插入。



我们可以利用双端队列来实现一个窗口，目的是让该窗口可以做到**张弛有度**（汉语博大精深，也就是长度动态变化。其实用游标或者其他解法的目的都是一样的，就是去维护一个可变长的窗口）

然后我们再做一件事，只要遍历该数组，同时**在双端队列的头去维护当前窗口的最大值**（在遍历过程中，发现当前元素比队列中的元素大，就将原来队列中的元素祭天），在整个遍历的过程中我们再记录下每一个窗口的最大值到结果数组中。最终结果数组就是我们想要的，整体图解如下。

假设  $\text{nums} = [1, 3, -1, -3, 5, 3, 6, 7]$ ，和  $k = 3$



(小浩os：我觉得自己画的这个图对于双端队列的解法还是介绍的比较清晰的，大家好好看一下哦，这样我的努力也就没有白费呢。)

根据分析，得出代码：

```

1 func maxSlidingWindow(nums []int, k int) []int {
2     if len(nums) == 0 {
3         return []int{}
4     }
5     //用切片模拟一个双端队列
6     queue := []int{}
7     result := []int{}
8     for i := range nums {
9         for i > 0 && (len(queue) > 0) && nums[i] > queue[len(queue)-1] {
10             //将比当前元素小的元素祭天
11             queue = queue[:len(queue)-1]
12         }
13         //将当前元素放入queue中
14         queue = append(queue, nums[i])
15         if i >= k && nums[i-k] == queue[0] {
16             //维护队列，保证其头元素为当前窗口最大值
17             queue = queue[1:]
18         }
19         if i >= k-1 {
20             //放入结果数组
21             result = append(result, queue[0])
22         }
23     }
24     return result
25 }
```

执行结果：

执行用时：20 ms，在所有 Go 提交中击败了 98.41% 的用户

内存消耗：6.3 MB，在所有 Go 提交中击败了 88.18% 的用户

Perfact题目完成！看着一下子超越百分之99的用户，是不是感觉很爽呢

## 无重复字符的最长子串（3）

在上一节中，我们使用**双端队列**完成了滑动窗口的一道颇为困难的题目，以此展示了什么是滑动窗口。在本节中我们将继续深入分析，探索滑动窗口题型一些具有模式性的解法。

### 01、滑动窗口介绍

对于大部分滑动窗口类型的题目，一般是**考察字符串的匹配**。比较标准的题目，会给出一个**模式串B**，以及一个**目标串A**。然后提出问题，找到**A**中符合对**B**一些**限定规则的子串**或者对**A**一些**限定规则的结果**，最终**再将搜索出的子串完成题意中要求的组合或者其他**。

比如：给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

又或者：给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

再如：给定一个字符串 s 和一些长度相同的单词 words。找出 s 中恰好可以由 words 中所有单词串联形成的子串的起始位置。

都是属于这一类的标准题型。而对于这一类题目，我们常用的解题思路，**是去维护一个可变长度的滑动窗口**。无论是使用**双指针**，还是使用**双端队列**，又或者用**游标**等其他奇技淫巧，目的都是一样的。

Now，我们通过一道题目来进行具体学习吧

### 02、题目分析

#### 第3题：无重复字符的最长子串

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1：

- 输入: "abcabcbb"
- 输出: 3
- 解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3

#### 示例 2:

- 输入: "bbbbbb"
- 输出: 1
- 解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

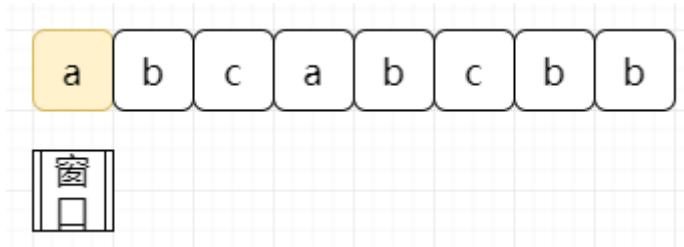
#### 示例 3:

- 输入: "pwwkew"
- 输出: 3
- 解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。

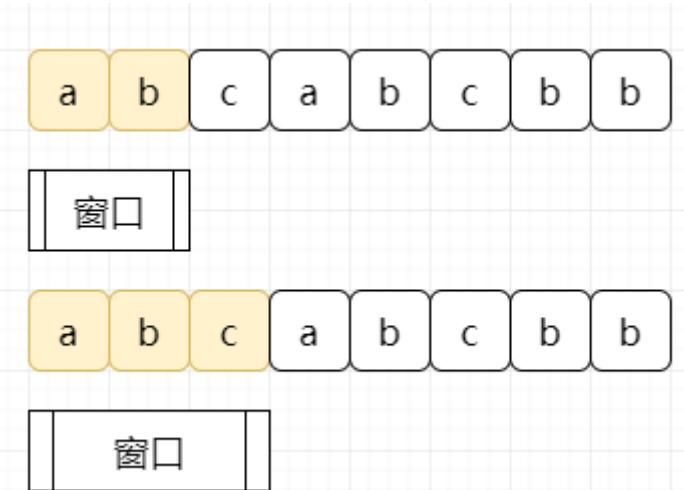
请注意，你的答案必须是子串的长度，“pwke”是一个子序列，不是子串。

## 03、题解分析

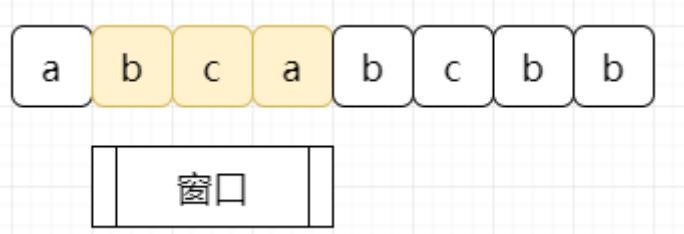
直接分析题目，假设我们的输入为“abcabcbb”，我们只需要维护一个窗口在输入字符串中进行移动。如下图：



当下一个元素在窗口没有出现过时，我们扩大窗口。



当下一个元素在窗口中出现过时，我们缩小窗口，将出现过的元素以及其左边的元素统统移出：



在整个过程中，我们记录下窗口出现过的最大值即可。而我们唯一要做的，只需要尽可能扩大窗口。

那我们代码中通过什么来维护这样的一个窗口呢？anyway~ 不管是队列，双指针，甚至通过map来做，都可以。

我们演示一个双指针的做法：

```

1 public class Solution {
2     public static int lengthOfLongestSubstring(String s) {
3         int n = s.length();
4         Set<Character> set = new HashSet<>();
5         int result = 0, i = 0, j = 0;
6         while (i < n && j < n) {
7             //charAt: 返回指定位置处的字符
8             if (!set.contains(s.charAt(j))) {
9                 set.add(s.charAt(j));
10                j++;
11                result = Math.max(result, j - i);
12            } else {
13                set.remove(s.charAt(i));
14                i++;
15            }
16        }
17        return result;
18    }
19 }
```

执行结果：

执行用时：19 ms，在所有 Java 提交中击败了 29.10% 的用户

内存消耗：46.7 MB，在所有 Java 提交中击败了 5.00% 的用户

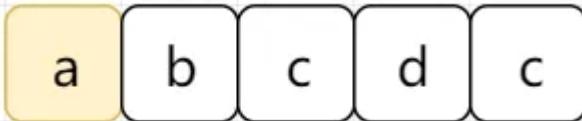
通过观察，我们能看出。如果是最坏情况的话，我们每一个字符都可能会访问两次，left一次，right一次，时间复杂度达到了 $O(2N)$ ，这是不可饶恕的。不理解的话看下图：

假设我们的字符串为“abcdc”，对于abc我们都访问了2次。

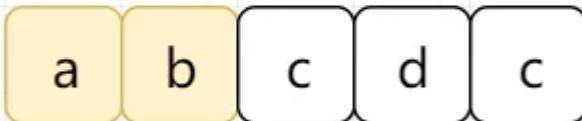


那如何来进一步优化呢？

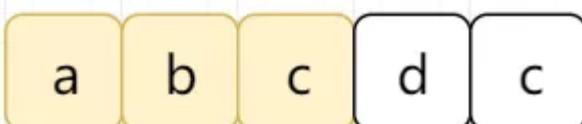
其实我们可以定义字符到索引的映射，而不是简单通过一个集合来判断字符是否存在。这样的话，当我们找到重复的字符时，我们可以立即跳过该窗口，而不需要对之前的元素进行再次访问。



a	1
---	---



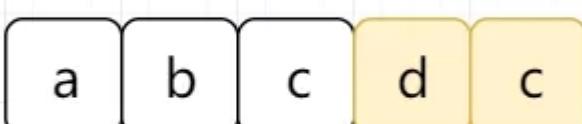
a	1
b	2



a	1
b	2
c	3



a	1
b	2
c	3
d	4



a	1
b	2
c	5
d	4

优化代码如下：

```
1 public class Solution {
2     public static int lengthOfLongestSubstring(String s) {
3         int n = s.length(), result = 0;
4         Map<Character, Integer> map = new HashMap<>();
5         for (int right = 0, left = 0; right < n; right++) {
6             if (map.containsKey(s.charAt(right))) {
7                 left = Math.max(map.get(s.charAt(right)), left);
8             }
9             result = Math.max(result, right - left + 1);
10            map.put(s.charAt(right), right + 1);
11        }
12        return result;
13    }
14 }
```

执行结果：

执行用时：**12 ms**，在所有 Java 提交中击败了 **56.06%** 的用户

内存消耗：**46.6 MB**，在所有 Java 提交中击败了 **5.00%** 的用户

修改之后，我们发现虽然时间复杂度有了一定提高，但是还是比较慢！如何更进一步的优化呢？我们可以使用一个256位的数组来替代hashmap，以进行优化。（因为ASCII码表里的字符总共有128个。ASCII码的长度是一个字节，8位，理论上可以表示256个字符，但是许多时候只谈128个。具体原因可以下去自行学习~）

进一步优化代码：

```
1 class Solution {
2     public int lengthOfLongestSubstring(String s) {
3         int n = s.length();
4         int result = 0;
5         int[] charIndex = new int[256];
6         for (int left = 0, right = 0; right < n; right++) {
7             char c = s.charAt(right);
8             left = Math.max(charIndex[c], left);
9             result = Math.max(result, right - left + 1);
10            charIndex[c] = right + 1;
11        }
12
13        return result;
14    }
15 }
```

执行结果：

执行用时：**3 ms**，在所有 Java 提交中击败了 **96.75%** 的用户

内存消耗：**37.5 MB**，在所有 Java 提交中击败了 **64.18%** 的用户

我们发现优化后时间复杂度有了极大的改善！这里简单说一下原因，对于数组和hashmap访问时，两个谁快谁慢不是一定的，需要思考 hashmap 的底层实现，以及数据量大小。但是在这里，因为已知了待访问数据的下标，可以**直接寻址**，所以极大的缩短了查询时间。

## 04、总结

本题基本就到这里。最后要说的，一般建议如果要分析一道题，我们要压缩压缩再压缩，抽茧剥丝一样走到最后，尽可能的完成对题目的优化。不一定非要自己想到最优解，但绝对不要局限于单纯的完成题目，那样将毫无意义！

# 找到字符串中所有字母异位词（438）

之前的两节讲解了滑动窗口类问题的模式解法，相信大家对该类题型已不陌生。今天将继续完成一道题目，来进行巩固学习。

## 01、题目分析

### 第438. 找到字符串中所有字母异位词

给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

字符串只包含小写英文字母，并且字符串 s 和 p 的长度都不超过 20100。

**说明：**

- 字母异位词指字母相同，但排列不同的字符串。
- 不考虑答案输出的顺序。

**示例 1：**

```
1 输入:s: "cbaebabacd" p: "abc"
2
3 输出:[0, 6]
4
5 解释:
6 起始索引等于 0 的子串是 "cba"，它是 "abc" 的字母异位词。
7 起始索引等于 6 的子串是 "bac"，它是 "abc" 的字母异位词。
```

**示例 2：**

```
1 输入:s: "abab" p: "ab"
2
3 输出:[0, 1, 2]
4
5 解释:
6 起始索引等于 0 的子串是 "ab", 它是 "ab" 的字母异位词。
7 起始索引等于 1 的子串是 "ba", 它是 "ab" 的字母异位词。
8 起始索引等于 2 的子串是 "ab", 它是 "ab" 的字母异位词。
```

#### 提示:

建议先完成上节内容的学习!

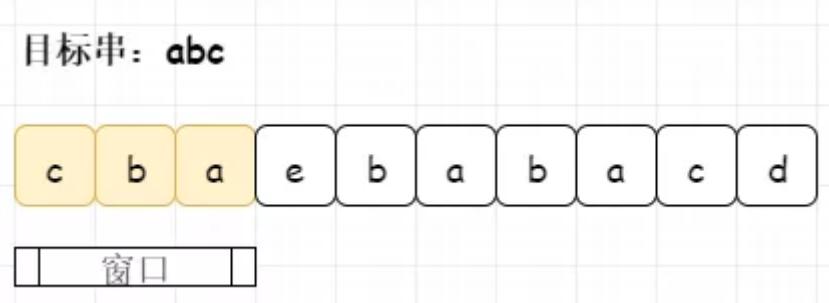
否则可能理解会有一定困难!

## 02、题解分析

直接套用之前的模式，使用双指针来模拟一个滑动窗口进行解题。分析过程如下：

假设我们有字符串为“cbaebabacd”，目标串为“abc”

我们通过双指针维护一个窗口，由于我们只需要判断字母异位词，我们可以将窗口初始化大小和目标串保持一致。（当然，你也可以初始化窗口为1，逐步扩大）

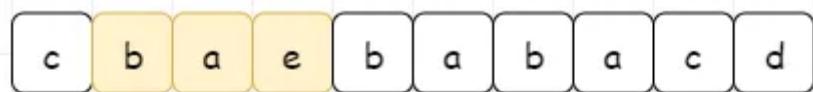


而判断字母异位词，我们需要保证窗口中的字母出现次数与目标串中的字母出现次数一致。这里因为字母只有26个，直接使用数组来替代map进行存储（和上一讲中的ASCII使用256数组存储思想一致）。

pArr为目标串数组，sArr为窗口数组。我们发现初始化数组，本身就满足，记录下来。（**这里图示用map模拟数组，便于理解**）

sArr	pArr
a 1	a 1
b 1	b 1
c 1	c 1

然后我们通过移动窗口，来更新窗口数组，进而和目标数组匹配，匹配成功进行记录。每一次窗口移动，**左指针前移，原来左指针位置处的数值减1，表示字母移出；同时右指针前移，右指针位置处的数值加1，表示字母移入**。详细过程如下：



窗口

sArr

a	1
b	1
c	0
d	0
e	1

pArr

a	1
b	1
c	1



窗口

sArr

a	1
b	1
c	0
d	0
e	1

pArr

a	1
b	1
c	1



窗口

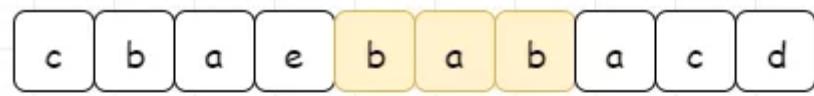
sArr

a	1
b	1
c	0

pArr

a	1
b	1
c	1

d	0						
e	1						



窗口

sArr

a	1
b	2
c	0
d	0
e	0

pArr

a	1
b	1
c	1



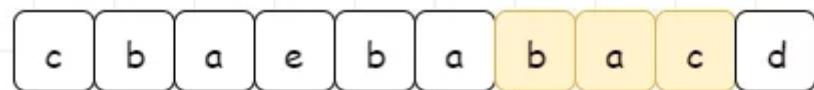
窗口

sArr

a	2
b	1
c	0
d	0
e	0

pArr

a	1
b	1
c	1



窗口

sArr

a	1
b	1
c	1

pArr

a	1
b	1
c	1



最终，当右指针到达边界，意味着匹配完成。

## 04、代码展示

根据分析，完成代码：(下面pSize相关的忽略，调试忘删了)

```

1 class Solution {
2
3     public List<Integer> findAnagrams(String s, String p) {
4
5         if (s == null || p == null || s.length() < p.length()) return new
6         ArrayList<>();
7
8         List<Integer> list = new ArrayList<>();
9
10        int[] pArr = new int[26];
11        int pSize = p.length();
12        int[] sArr = new int[26];
13
14        for (int i = 0; i < p.length(); i++) {
15            sArr[s.charAt(i) - 'a']++;
16            pArr[p.charAt(i) - 'a']++;
17        }
18
19        for (int i = 0; i < p.length(); i++) {
20            int index = p.charAt(i) - 'a';
21            if (pArr[index] == sArr[index])
22                pSize--;
23        }
24
25        int i = 0;
26        int j = p.length();
27
28        // 窗口大小固定为p的长度
29        while (j < s.length()) {
30            if (isSame(pArr, sArr))
31                list.add(i);
32            //sArr[s.charAt(i) - 'a']-- 左指针位置处字母减1
33            sArr[s.charAt(i) - 'a']--;
34            i++;
35            //sArr[s.charAt(j) - 'a']++ 右指针位置处字母加1
36            sArr[s.charAt(j) - 'a']++;
37            j++;
38        }
39
40        if (isSame(pArr, sArr))
41            list.add(i);
42
43        return list;
}

```

```
44
45     public boolean isSame(int[] arr1, int[] arr2) {
46         for (int i = 0; i < arr1.length; ++i)
47             if (arr1[i] != arr2[i])
48                 return false;
49         return true;
50     }
51 }
```

执行结果：

执行用时：21 ms，在所有 Java 提交中击败了 49.31% 的用户

内存消耗：45.4 MB，在所有 Java 提交中击败了 25.02% 的用户

PS：本题属于模式解法，非最优解！这里留下思考内容，如何在本解法的基础上进行优化，进一步降低时间复杂度？

## 博弈论系列

### 囚徒困境

本系列将为大家带来一整套的**博弈论问题**。因为在面试的过程中，除了常规的算法题目，我们经常也会被问到一些趣味题型来考察思维，而这类问题中，很多都有博弈论的影子存在。这些公司里以FLAG (Facebook, LinkedIn, Amazon, Google) 为典型，特别喜欢考察本类题型。同时，本系列将不一定都是算法问题，不是IT行业的小伙伴也可以进行学习，来提高分析问题的能力~

### 01、什么是“博弈论”

古语有云，“笑人情似纸，世事如棋”。生活中每个人如同棋手，其每一个行为如同在一张看不见的棋盘上布子，精明慎重的棋手们相互揣摩、牵制、争赢，下出诸多精彩纷呈、变化多端的棋局。而什么是博弈论？就是研究棋手们的“出棋”过程，从中抽象出可逻辑化的部分，并将其系统化的一门科学，也是运筹学的一个重要学科。

我们从最简单的一道“囚徒困境”来进行学习~

### 02、囚徒困境

#### 囚徒困境

一件严重的纵火案发生后，警察在现场抓到两个犯罪嫌疑人。事实上，正是他们一起放火烧了这座仓库。但是，警方没有掌握足够的证据，只得把他们分开囚禁起来，要求他们坦白交代。

在分开囚禁后，警察对其分别告知：

如果你坦白，而对方不坦白，则将你释放，判对方8年。

如果你不坦白，而对方坦白，则将对方释放，而判你8年。

如果你两都坦白了，则判你两各自4年。

那么两个囚犯应该如何做，是互相背叛还是一起合作？

### 题目分析：

从表面上看，其实囚犯最应该的就是一起合作，都不坦白，这样因为证据不足，会将两人都进行释放。但是！因为事实确实是两人放的火，所以他们**不得不进行思考，另一人采取了什么样的行为？**

犯人甲当然不傻，他根本无法相信同伙不会向警方提供任何信息！因为如果同伙一旦坦白，而自己这边如果什么都没说的话，就可以潇洒而去。但他同时也意识到，他的同伙也不傻，也会同样来这样设想他。

所以犯人甲的结论是，**唯一理性的选择就是背叛同伙**，把一切都告诉警方！这样的话，如果他的同伙笨得只会保持沉默，那么他就会是那个离开的人。而如果他的同伙也根据这个逻辑向警方交代了，那么也没有关系，起码他不必服最重的刑！

	甲供认	甲不供认
乙供认	甲 <b>4年</b> 乙 <b>4年</b>	甲 <b>8年</b> 乙释放
乙不供认	甲释放 乙 <b>8年</b>	无罪释放

## 03、囚徒困境与纳什均衡

这场博弈的过程，**显然不是顾及团体利益的最优解决方案**。以全体利益而言，如果两个参与者都合作保持沉默，两人都可以无罪释放，总体利益更高！但根据假设（人性），**二人为理性的个人**，且只追求自己的个人利益。均衡状况会是两个囚徒都选择背叛，这就是“困境”所在！

事实上，这种**两人都选择坦白的策略以及因此被判4年的结局被称作“纳什均衡”**（也叫非合作均衡），换言之，在此情况下，**无一参与者可以“独自行动”（即单方面改变决定）而增加收获。**

我们看一下官方释意是多么难懂“所谓纳什均衡，指的是参与人的一种策略组合，在该策略组合上，**任何参与人单独改变策略都不会得到好处。**”简单点讲，如果在一个策略组合上，当所有其他人都不改变策略时，没有人会改变自己的策略，则该策略组合就是一个纳什均衡。

## 辛普森悖论

本系列主要为大家带来一整套的**博弈论问题（广义）**。因为在面试的过程中，除了常规的算法题目，我们经常也会被问到一些趣味题型来考察思维，而这类问题中，很多都有博弈论的影子存在。这些公司里以FLAG (Facebook, LinkedIn, Amazon, Google) 为典型，特别喜欢考察本类题型。同时，本系列将不一定都是算法问题，不是IT行业的小伙伴也可以进行学习，来提高分析问题的能力~

### 01、辛普森悖论

#### 辛普森悖论

羊羊医院里统计了两种胆结石治疗方案的治愈率。在统计过程中，医生将病人分为大膽结石和小膽结石两组。统计结果如下：

	手术A	手术B
小胆结石	93%(81/87)	87%(234/270)
大膽结石	73%(192/263)	69%(55/80)

- 对于小胆结石而言，手术A的治愈率（93%）高于手术B（87%）
- 对于大膽结石而言，手术A的治愈率（73%）高于手术B（69%）

**羊羊医院的医生得出结论：**

无论是对于大小胆结石，手术A的治愈率都胜过手术B。

但是真的是这样吗？当然不是，我们根据样本统计出大小胆结石总计的治愈率，发现**手术B(治愈率83%)其实是要高于手术A(治愈率78%)**。

	手术A	手术B
小胆结石	93%(81/87)	87%(234/270)
大胆结石	73%(192/263)	69%(55/80)
总计	78%(273/350)	83%(289/350)

为什么会出现这样的结果？这就是著名的辛普森悖论。

## 02、直觉的缺陷

得到了结论，我们来思考背后的东西。在我们的直觉里有这样一个逻辑：**如果一个事物的各部分都分别大于另一个事物的各部分，那么这个事物大于另一个事物。**比如：我们的直觉告诉我们如果手术A在两组病人中都更好，那么在所有病人中也应该更好。

我们可以将其公式化（该公式错误），假设：

$$A = A_1 + A_2 + \dots + A_n$$

$$B = B_1 + B_2 + \dots + B_n$$

如果对  $i=1,2,\dots,n$  都有  $A_i > B_i$ ，则  $A > B$

乍一看，我们觉得该公式没有问题~所以这个公式也就代表了我们大部分人的思维工作。其实在这个公式中，隐藏掉了一个很重要的条件：**A1、A2、An 以及 B1、B2、Bn 并不能简单的通过“加”来得到 A 或者 B。**这就是**可加性**的前提。在大脑的思维过程中，因为我们很难直接看到这个前提，进而就导致了我们错误的思考！

## 03、辛普森悖论举例

下面我们举一些在生活中常见的辛普森悖论例子：

- 打麻将的时候，把把都赢小钱，造成赢钱的假象，其实不如别人赢一把大的。
- 在苹果和安卓的竞争中，你听见身边的人都在逃离苹果，奔向安卓。但是其实苹果的流入率还是要高于安卓。（有数据证明，很经典的案例）
- 你男票，这里比别人差，那里比别人差，但是其实他真的比别的男生差吗？（这个纯属本人胡扯了..）

好啦，这篇内容就到这里了，你学会了吗？

## 红眼睛和蓝眼睛

在面试的过程中，除了常规的算法题目，我们经常也会被问到一些趣味题型来考察思维，尤其以FLAG (Facebook, LinkedIn, Amazon, Google) 等公司为典型。而这类问题的背后，很多都有博弈论的影子。所以在本系列，我将为大家分享一整套需要掌握的**博弈论**相关知识，希望大家可以喜欢。

PS：本系列将不一定都是算法问题，不是IT行业的小伙伴也可以进行学习，来提高自身分析问题的能力。

## 01、红眼睛和蓝眼睛

### 红眼睛和蓝眼睛

一个岛上有100个人，其中有5个红眼睛，95个蓝眼睛。这个岛有三个奇怪的宗教规则。

- 1.他们不能照镜子，不能看自己眼睛的颜色。
- 2.他们不能告诉别人对方的眼睛是什么颜色。
- 3.一旦有人知道了自己是红眼睛，他就必须在当天夜里自杀。

某天，有个旅行者到了这个岛上。由于不知道这里的规矩，所以他在和全岛人一起狂欢的时候，不留神就说了一句话：【你们这里有红眼睛的人。】

问题：假设这个岛上的人每天都可以看到其他所有人，每个人都可以做出缜密的逻辑推理，请问岛上会发生什么？

## 02、题目分析

题目乍看之下，没有任何逻辑可言！以目测条件，基本无法完成任何正常的推理。但是在仔细推敲之后，我们可以将问题简化，从假设只有1个红眼睛开始分析。

我们假设岛上只有1个红眼睛的人，还有99个都是蓝眼睛。因为这个旅行者说了“这里有红眼睛的人”，**那么在第一天的时候，这个红眼睛会发现其他的人都是蓝眼睛**（与此同时，其他人因为看到了这个红眼睛的人，所以都确认了自己的安全）**那么这天晚上，这个红眼睛的人一定会自杀！**

继续分析，假设这个岛上有2个红眼睛，那么当旅行者说“这里有红眼睛的人”之后的第一天，这两个红眼睛分别发现还有别的红眼睛存在，所以他们当天晚上认为自己是安全的。但是到了第二天，红眼睛惊讶的发现，**另一个红眼睛的人竟然没有自杀（说明岛上有不止一个红眼睛），并且当天他们也没有发现有别的红眼睛存在（说明另一个红眼睛就是自己）**WTF，那肯定另一个红眼睛就是自己了，所以在**第二天夜里，两个红眼睛的人会同时自杀！**

继续分析，假如岛上红眼睛有3个。那么在第一天，红眼睛发现了岛上还有另外两个红眼睛，红眼睛呵呵一笑，“反正不是我”。到了第二天，红眼睛仍然看到了另外两个红眼睛，红眼睛心想，“这下你两该完蛋了吧”，毕竟你两都知道了自己是红眼睛，晚上回去统统自杀吧！（根据上面的推论得出）但是惊奇的是，到了第三天，红眼睛发现另外两个红眼睛竟然都没有自杀。（说明岛上红眼睛的人不止两个）并且当天红眼睛也没发现新的红眼睛（说明还有一个红眼睛就是自己）所以在第三天的夜里，三个红眼睛会同时自杀。

根据上面的推论，**假设有N个红眼睛，那么到了第N天，这N个红眼睛就会自杀**。所以最终这个岛上红眼睛的人会统统自杀！这就是答案，生活就是这么朴实无华，且枯燥。

## 03、旅客的挽回

上面的分析大家应该都看懂了。但若是在旅客说完这句话后，其并没有离开这个岛。同时他也看到了周围人眼里的惊慌和失措，这个时候，旅客为自己的行为感到了懊恼和悔恨！旅客决定对自己的话进行挽回，旅客又该怎么做呢？

这里我提供一种思路，**旅客可以在第N次集会上杀掉N个红眼睛**，让这N个红眼睛“GO TO SLEEP”，就可以中断事件的推理。事实上，基于人道主义，旅客并不需要手动杀人，她只需要在第N天的时候告诉这N个人，你们是红眼睛，那么这天晚上，这N个人就会自杀。“All RETURN”，一切将回归秩序~

## 海盗分金币

在面试的过程中，除了常规的算法题目，我们经常也会被问到一些趣味题型来考察思维，尤其以FLAG (Facebook, LinkedIn, Amazon, Google) 等公司为典型。而这类问题的背后，很多都有博弈论的影子。所以在本系列，我将为大家分享一整套需要掌握的**博弈论**相关知识，希望大家可以喜欢。

PS：本系列将不一定都是算法问题，不是IT行业的小伙伴也可以进行学习，来提高自身分析问题的能力。

## 01、海盗分金币问题

### 海盗分金币

在大海上，有5个海盗抢得100枚金币，他们决定每一个人按顺序依次提出自己的分配方案，如果提出的方案没有获得半数或半数以上的人的同意，则这个提出方案的人就被扔到海里喂鲨鱼。那么第一个提出方案的人要怎么做，才能使自己的利益最大化？

海盗们有如下特点：

- 1.足智多谋，总是采取最优策略。

2. 贪生怕死，尽量保全自己性命。
3. 贪得无厌，希望自己得到越多宝石越好
4. 心狠手辣，在自己利益最大的情况下希望越多人死越好。
5. 疑心多虑，不信任彼此，尽量确保自身利益不寄希望与别人给自己更大利益。

## 02、题目分析

首先我们很容易会觉得，抽签到第一个提方案的海盗会很吃亏！因为只要死的人够多，那么平均每个人获取的金币就最多，而第一个提方案的人是最容易死的。但是事实是，在满足海盗特点的基础上，**第一个提方案的海盗是最赚的**，我们一起来分析一下。

假如我们设想只有两个海盗。那么不管第一个说什么，只要第二个人不同意，第二个人就可以得到全部的金币！所以**第一个海盗必死无疑**，这个大家都能理解。（当然，这样的前提是二号提出方案后不可以马上自己同意，不然如果自己提出给自己全部金币的方案，然后自己支持，这样就是二号必死无疑）

假如现在我们加入第三个海盗，这时候原来的一号成为了二号，二号成为了三号。这时候现在的二号心里会清楚，**如果他投死了一号，那么自己必死无疑！** 所以根据贪生怕死的原则，二号肯定会让一号存活。而此时一号心理也清楚，无论自己提出什么样的方案，二号都会让自己存活，而这时只要加上自己的一票，就有半数通过，所以一号提出方案：把金币都给我。

现在又继续加入了新的海盗！原来的1,2,3号，成为了现在的2,3,4号。这时候新的一号海盗洞悉了奥秘，知道了**如果自己死了，二号就可以获取全部的金币**，所以提出给三号和四号一人一个金币，一起投死2号。而与此同时，现在的3号和4号获取的要比三个人时多（三个人时自己获取不了任何金币），所以他们会同意这个方案！

现在加入我们的大Boss，最后一个海盗。根据分析，大Boss海盗1号推知出2号的方案后就可以提出(97,0,1,2,0)或者(97,0,1,0,2)的方案。这样的分配方案对现在的3号海盗相比现在的2号的分配方案还多了1枚金币，就会投赞成票，4号或者5号因为得到了2枚金币，相比2号的一枚多，也会支持1号，加上1号自己的赞成票，方案就会通过，即1号提出(97,0,1,2,0)或(97,0,1,0,2)的分配方案，大Boss成功获得了97枚金币。

## 03、思考

最终，**大Boss一号海盗得到97枚金币，投死了老二和老五**，这竟然是我们分析出的最佳方案！这个答案明显是反直觉的，如果你是老大，你敢这样分金币，必死无疑。可是，推理过程却非常严谨，无懈可击，那么问题出在哪里呢？

其实，在“海盗分赃”模型中，任何“分配者”想让自己的方案获得通过的关键是，**事先考虑清楚“对手”的分配方案是什么**，\*\*并用最小的代价获取最大收益，拉拢“对手”分配方案中最不得意的人们\*\*。1号看起来最有可能喂鲨鱼，但他牢牢地把握住先发优势，结果不但消除了死亡威胁，还收益最大。而5号，看起来最安全，没有死亡的威胁，甚至还能坐收渔人之利，却因不得不看别人脸色行事而只能分得一小杯羹。

不过，模型任意改变一个假设条件，最终结果都不一样。而现实世界远比模型复杂。**因为假定所有人都理性，本身就是不理性的。**回到“海盗分金”的模型中，只要3号、4号或5号中有一个人偏离了绝对聪明的假设，海盗1号无论怎么分都可能会被扔到海里去了。所以，1号首先要考虑的就是他的海盗兄弟们的聪明和理性究竟靠得住靠不住，否则先分者必定倒霉。

如果某人和一号本身不对眼，就想丢他喂鲨鱼。果真如此，1号自以为得意的方案岂不成了自掘坟墓。再就是俗语所说的“人心隔肚皮”。由于信息不对称，谎言和虚假承诺就大有用武之地，而阴谋也会像杂草般疯长，并借机获益。如果2号对3、4、5号大放烟幕弹，宣称对于1号所提出任何分配方案，他一定会再多加上一个金币给他们。这样，结果又当如何？

通常，现实中人人都有自认的公平标准，因而时常会嘟囔：“**谁动了我的奶酪？**”可以料想，一旦1号所提方案和其所想的不符，就会有人大闹。当大家都闹起来的时候，1号能拿着97枚金币毫发无损、镇定自若地走出去吗？最大的可能就是，海盗们会要求修改规则，然后重新分配。当然，大家也可以讲清楚下次再得100枚金币时，先由2号海盗来分...然后是3号.....颇有点像美国总统选举，轮流主政。说白了，其实是民主形式下的分赃制。

最可怕的是其他四人形成一个反1号的大联盟并制定出新规则：四人平分金币，将1号扔进大海。这就颇有点阿Q式的革命理想：高举平均主义的旗帜，将富人扔进死亡深渊。

最后，这里也提供一份代码实现，供有兴趣的同学参考（该代码我大概看了一下，但是因为时间的关系，没有跑单测进行验证，特此说明！）

1 | 以上代码输出：5人时分配方案：[97, 0, 1, 0, 2]

看懂了吗？如果看懂了，这里提出一个问题：假如我们将**人性**考虑在内，同时也进行**理性的**分析，如果你是老大，又该如何提出这个方案呢？大家在留言区留下自己的回答吧！

## 排序类题目

### 按奇偶排序数组 (905)

在本系列中，将为大家讲解**排序算法**相关内容。同时，由于网上排序相关的教程太多了，我会尽可能的讲解一些不一样的内容。而不是按照 排序讲解 标准Title，什么“十大排序算法”，“经典排序算法”，“排序算法必知必会”之类的一个一个来进行讲解。所以，如果内容引起不适，概不负责...

## 01、排序的重要性

在leetcode中，直接搜索**排序标签**出现的题目有80余道，这是与排序直接相关的题目，不包括其他一些用到排序思想的题目。

#	题名	题解	通过率	难度
280	摆动排序	19	68.8%	中等
912	排序数组	69	52.6%	中等
面试题 16.16	部分排序 <span style="color:red">新</span>	3	43.2%	中等
面试题 03.05	栈排序	1	41.7%	中等
969	煎饼排序	36	62.6%	中等
324	摆动排序 II	28	34.7%	中等
148	排序链表	140	63.8%	中等
面试题 10.09	排序矩阵查找	1	48.3%	中等
33	搜索旋转排序数组	289	36.4%	中等
869	重新排序得到 2 的幂	14	48.5%	中等
面试题 10.01	合并排序的数组 <span style="color:red">新</span>	14	51.5%	简单
26	删除排序数组中的重复项	689	48.5%	简单
791	自定义字符串排序	34	65.3%	中等
451	根据字符出现频率排序	84	61.9%	中等
768	最多能完成排序的块 II	17	45.0%	困难
80	删除排序数组中的重复项 II	139	54.4%	中等
81	搜索旋转排序数组 II	78	34.9%	中等
1329	将矩阵按对角线排序	26	75.1%	中等
922	按奇偶排序数组 II	129	66.9%	简单
1122	数组的相对排序	147	65.1%	简单
905	按奇偶排序数组	170	68.1%	简单
147	对链表进行插入排序	89	62.6%	中等
23	合并K个排序链表	281	48.8%	困难
面试题25	合并两个排序的链表	30	77.5%	简单
769	最多能完成排序的块	17	53.7%	中等
面试题53 - I	在排序数组中查找数字 I	24	52.8%	简单
82	删除排序链表中的重复元素 II	218	45.8%	中等
83	删除排序链表中的重复元素	228	49.0%	简单
153	寻找旋转排序数组中的最小值	119	50.2%	中等
154	寻找旋转排序数组中的最小值 II	71	46.2%	困难
34	在排序数组中查找元素的第一个和最后一个位置	342	39.0%	中等
426	将二叉搜索树转化为排序的双向链表	24	60.6%	中等

同时，各个公司在面试的过程中，或多或少都直接或间接问到过排序相关的内容（毕竟面试官不知道问什么时，都会用排序算法来救救场。不要问我是怎么知道的...），尤其是**快排、堆排序、全排列**等Topic，在面试中屡试不爽。

百度：堆排序

百度 (20张照片)  
Baidu 百度  
互联网 | 北京 | 10000人以上  
全球最大的中文搜索引擎、最大的中文网站 [more](#)

主页 问答 点评 面试 工资 招聘

百度软件开发面试经验详情

匿名用户 软件开发的面试经验 两年前  
问了对排序的问题自己...  
面试地点：百度-北京  
问了对排序的问题自己不会  
面试官的问题：  
问 堆排序  
面试结果：面试未通过 面试难度：有难度 面试感受：不好

### 滴滴：全排列

滴滴现场三面面经



总体问了很多，都是关于之前在滴滴实习项目的问题的思考和具体场景的优化，很多：

算法：

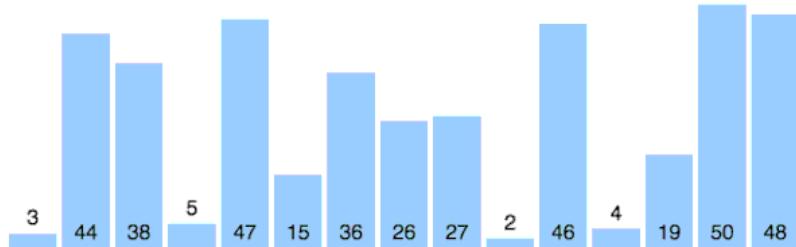
- 1.层序遍历
- 2.下一个排列
- 3.全排列
- 4.手撕ArrayList

综上，得出结论：**为了offer~排序很重要，我们需要进行掌握。**

## 02、从“插入排序”说起

为什么要先讲**插入排序**的原因，是因为我觉得插入排序是最容易理解的一个，而且插入这个词有一定的神秘感（好吧，反正我不觉得冒泡最容易理解，谁没事一天去观察吐泡泡？）

插入排序：就是炸金花的时候，你接一个同花顺的过程。（标准定义：在要排序的一组数中，假定前n-1个数已经排好序，现在将第n个数插到前面的有序数列中，使得这n个数也是排好顺序的）



代码示例：

```

1 func main() {
2     arr := []int{5, 4, 3, 2, 1}
3     insert_sort(arr)
4 }
5 func insert_sort(arr []int) {
6     for i := 1; i < len(arr); i++ {
7         for j := i; j > 0; j-- {
8             if arr[j] < arr[j-1] {
9                 arr[j], arr[j-1] = arr[j-1], arr[j]
10            }
11        }
12        fmt.Println(arr)
13    }
14 }
```

输入：

```

[4 5 3 2 1]
[3 4 5 2 1]
[2 3 4 5 1]
[1 2 3 4 5]
```

讲解完了插入排序，我们根据其思想，完成下面这道题目吧

## 03、题目分析

### 第905题：按奇偶排序数组

给定一个非负整数数组 A，返回一个数组，在该数组中，A 的所有偶数元素之后跟着所有奇数元素。你可以返回满足此条件的任何数组作为答案。

示例：

- 1 输入： [3,1,2,4]
- 2 输出： [2,4,3,1]
- 3 输出 [4,2,3,1], [2,4,1,3] 和 [4,2,1,3] 也会被接受。

**提示：**

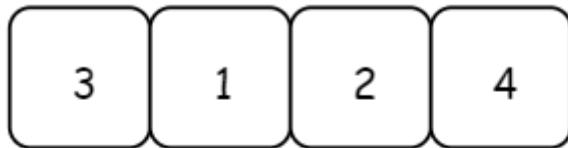
```
1 | 1 <= A.length <= 5000  
2 | 0 <= A[i] <= 5000
```

## 04、题目图解

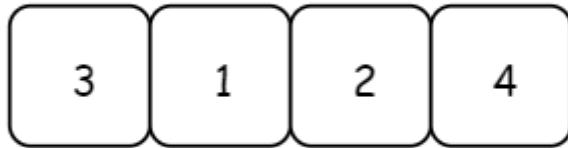
这道题，按照插入排序的思想，很容易可以想到题解。我们只需要遍历数组，当我们遇到偶数时，**将其插入到数组前最近的一个为奇数的位置，\*\*与该位置的奇数元素交换\*\***。为了达成该目的，我们引入一个指针 j，来维持这样一个奇数的位置。

假设我们的数组为：[3,1,2,4]

开始遍历



j i

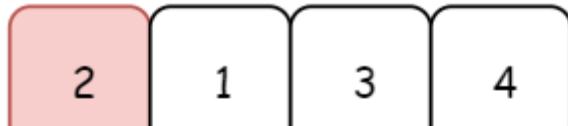


j i



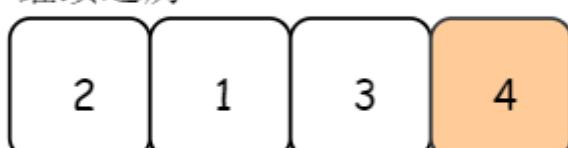
j i

发现偶数，插入到前排，移动 j



j i

继续遍历



j i

发现偶数，插入到前排，移动 j



j i

根据以上分析，得到代码：

```
1 func sortArrayByParity(A []int) []int {
2     j := 0
3     for i := range A {
4         if A[i]%2 == 0 {
5             A[j], A[i] = A[i], A[j]
6             j++
7         }
8     }
9     return A
10 }
```

执行结果：

执行用时：8 ms，在所有 Go 提交中击败了 98.33% 的用户

内存消耗：4.7 MB，在所有 Go 提交中击败了 100.00% 的用户

## 位运算系列

### 使用位运算求和

今天为大家分享一道本应很简单的题目，但是却因增加了特殊条件，而大幅增加了难度。话不多说，直接看题。

#### 01、题目标示例

该题很容易出现在各大厂的面试中，属于必须掌握的题型。

##### 连续n个数的和

求  $1 + 2 + \dots + n$ ，要求不能使用乘除法、for、while、if、else、switch、case等关键字及条件判断语句 (A?B:C)。

##### 示例 1：

1 | 输入：n = 3 输出：6

##### 示例 2：

1 | 输入：n = 9 输出：45

##### 限制：

## 02、题目分析

这道题目出自《贱指offer》，因为比较有趣，就拿来分享给大家。

题目上手，因为不能使用公式直接计算（公式中包含乘除法），所以考虑使用递归进行求解，但是**递归中一般又需要使用if来指定返回条件（这里不允许使用if）**，所以没办法使用普通的递归思路。那该怎么办呢？这里我们直接上代码（本题将展示多个语言），再进行分析。

```

1 //JAVA
2 class Solution {
3     public int sumNums(int n) {
4         boolean b = n > 0 && ((n -= sumNums(n - 1)) > 0);
5         return n;
6     }
7 }
```

首先我们了解一下 `&&` 的特性，比如有 `A&&B`

- 如果A为true，返回B的布尔值（继续往下执行）
- 如果A为false，直接返回false（相当于短路）

利用这一特性，我们将递归的返回条件取非然后作为 `&&` 的第一个条件，递归主体转换为第二个条件语句。我知道肯定有人又会懵圈了，所以我们绘图说明。假若这里`n=3`，大概就是下面这样：

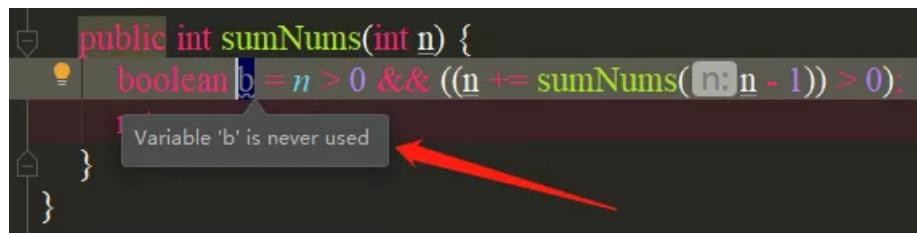
```

1 //java
2 public int sumNums(int n) {
3     System.out.println("start:" + n);
4     boolean b = n > 0 && ((n += sumNums(n - 1)) > 0);
5     System.out.println("end:" + n);
6     return n;
7 }
```

```

start:3
start:2
start:1
start:0
end:0
end:1
end:3
end:6
```

这里还有一点要强调的就是，受制于各语言的语法规则，我们需要做一些额外的处理。比如Java，这里如果去掉前面的变量申明，就会直接报错。



但是如果是C 就没有这样的问题：

```
1 //c
2 int sumNums(int n) {
3     n && (n = sumNums(n-1));
4     return n;
5 }
```

python就是下面这样：

```
1 //py3
2 class Solution:
3     def sumNums(self, n: int) -> int:
4         return n and n + self.sumNums(n-1)
```

Go怎么搞？

```
1 //go
2 func plus(a *int, b int) bool {
3     *a = b
4     return true
5 }
6 func sumNums(n int) int {
7     _ = n > 0 && plus(&n, sumNums(n - 1))
8     return n10}
```

什么，还要我给一个PHP的？惹不起..惹不起...大佬请拿走...

```
1 //PHP
2 class Solution {
3     function sumNums($n) {
4         $n > 0 && $n = $this->sumNums($n - 1);
5         return $n;
6     }
7 }
```

## 03、额外福利

另外，我还看到这样一个解法，感觉很有趣（思想很简单）。因为不是自己写的，所以这里得额外说明，咱不能白嫖，对不？（所以你们这些白嫖的不去给我点个star嘛...）

```
1 //go
2 func sumNums(n int) int {
3     return (int(math.Pow(float64(n), float64(2)))) n>>1
4 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：1.9 MB，在所有 Go 提交中击败了 100.00% 的用户

## 2的幂(231)

今天给大家分享一道比较简单但是很经典的题目。话不多说，直接看题。

### 01、题目示例

这道题，大家先想一想是用什么思路进行求解？

#### 第231题：2的幂

给定一个整数，编写一个函数来判断它是否是 2 的幂次方。

示例 1：

1	输入： 1
2	输出： true
3	解释： $2^0 = 1$

示例 2：

1	输入： 16
2	输出： true
3	解释： $2^4 = 16$

示例 3：

1	输入： 218
2	输出： false

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

### 02、题目分析

这道题是通过位运算来进行求解的非常典型的题目。当然，其他的题解也有很多：比如暴力求解，又或者是不停除以2通过递归的方式求解，等等。但是并不是今天我想说的。

先观察一些是2的幂的二进制数：

1						1
2					1	0
4				1	0	0
8			1	0	0	0
16		1	0	0	0	0
32	1	0	0	0	0	0

可以发现这些数，都是最高位为1，其他位为0。所以我们把问题转化为“判断一个数的二进制，除了最高位为1，是否还有别的1存在”。然后我们再观察下面这样的一组数，对应着上面的数减去1：

0						
1						1
3					1	1
7				1	1	1
15			1	1	1	1
31		1	1	1	1	1

我们对两组数求“&”运算：

2	0	0	0	0	1	0
1	0	0	0	0	0	1
2&1	0	0	0	0	0	0

8	0	0	1	0	0	0
7	0	0	0	1	1	1
8&7	0	0	0	0	0	0

4	0	0	0	1	0	0
3	0	0	0	0	1	1
4&3	0	0	0	0	0	0

16	0	1	0	0	0	0
15	0	0	1	1	1	1
16&15	0	0	0	0	0	0

可以看到，对于N为2的幂的数，都有  $N \& (N-1) = 0$ ，所以这就是我们的判断条件。（这个技巧可以记忆下来，在一些别的位运算的题目中也是会用到的）

根据分析，完成代码：

```

1 //go
2 func isPowerOfTwo(n int) bool {
3     return n > 0 && n&(n-1) == 0
4 }
```

执行结果：

执行用时：4 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2.1 MB，在所有 Go 提交中击败了 100.00% 的用户

### 03、证明过程

“下里巴人”和“阳春白雪”是古代楚国的歌曲名，屈原的大弟子宋玉曾著有《对楚王问》：“客有歌于郢中者，其始曰下里巴人，国中属而和者数千人……其为阳春白雪，国中属而和者不过数十人。”“下里巴人”和“阳春白雪”一词后来被用来泛指通俗和高雅的文艺作品。古琴十大名曲之一。

“阳春白雪，下里巴人”这个比喻虽然有点牵强，但是却难掩位运算的重要性。位运算在整个算法体系里，不少人可能会觉得有点食之无味、弃之可惜的意思。但其实，完全不是这样！有这种想法的，大多是初学者。对于这点，应该C系的玩家，会深有感触。万丈高楼平地起，暂且不说位运算在底层运算中占据了多大比重，单是整个leetcode列表里，打着位运算标签的题目就超过80余道，我想已经说明了问题。**至少，在面试这块，你必须对位运算了如指掌！**所以，今天的题目算是一个引子，后面我会出一个位运算的专题，希望尽我所能，帮助大家攻克这一类型的问题。

#	题名	题解	通过率	难度
401	<a href="#">二进制手表</a>	88	52.0%	简单
405	<a href="#">数字转换为十六进制数</a>	67	49.9%	简单
411	<a href="#">最短特异单词缩写</a>	9	45.0%	困难
421	<a href="#">数组中两个数的最大异或值</a>	28	58.5%	中等
461	<a href="#">汉明距离</a>	217	75.2%	简单
476	<a href="#">数字的补数</a>	113	68.2%	简单
477	<a href="#">汉明距离总和</a>	27	49.3%	中等
693	<a href="#">交替位二进制数</a>	61	60.5%	简单
751	<a href="#">IP 到 CIDR</a>	11	60.9%	简单
756	<a href="#">金字塔转换矩阵</a>	22	53.9%	中等
762	<a href="#">二进制表示中质数个计算置位</a>	42	67.0%	简单
784	<a href="#">字母大小写全排列</a>	105	62.9%	简单
898	<a href="#">子数组按位或操作</a>	12	27.4%	中等

所以，今天的问题你学会了吗？评论区留下你的想法！

## 返回一个数二进制中1的个数(191)

今天继续分享一道和位运算有关的题型，同样在难度上属于简单。我们还是从一道题开始吧

### 01、题目示例

这道题，大家先想一想是用什么思路进行求解？

#### 第191题：位1的个数

编写一个函数，输入是一个无符号整数，返回其二进制表达式中数字位数为‘1’的个数（也被称为汉明重量）。

示例 1：

1 | 输入: 000000000000000000000000000000001011  
2 | 输出: 3  
3 | 解释: 输入的二进制串 000000000000000000000000000000001011 中, 共有三位为 '1'。

### 示例 2:

1 | 输入: 0000000000000000000000000000000010000000  
2 | 输出: 1  
3 | 解释: 输入的二进制串 0000000000000000000000000000000010000000 中, 共有一位为 '1'。

### 示例 3:

1 | 输入: 1111111111111111111111111111111101  
2 | 输出: 31  
3 | 解释: 输入的二进制串 1111111111111111111111111111111101 中, 共有 31 位为 '1'。

### 提示:

- 请注意, 在某些语言(如 Java)中, 没有无符号整数类型。在这种情况下, 输入和输出都将被指定为有符号整数类型, 并且不应影响您的实现, 因为无论整数是有符号的还是无符号的, 其内部的二进制表示形式都是相同的。
- 在 Java 中, 编译器使用二进制补码记法来表示有符号整数。因此, 在上面的示例 3 中, 输入表示有符号整数 -3。

PS: 建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

## 02、题目分析

这道题仍然是通过位运算来进行求解的非常典型的题目。掩码是指使用一串二进制代码对目标字段进行位与运算, 屏蔽当前的输入位。

首先最容易想到的方法是: 我们直接把目标数转化成二进制数, 然后遍历每一位看看是不是1, 如果是1就记录下来。通过这种比较暴力的方式, 来进行求解。比如java中, int类型是32位, 我们只要能计算出当前是第几位, 就可以顺利进行求解。

那如何计算当前是第几位呢, 我们可以构造一个掩码来进行, 说掩码可能大家听着有点懵逼, 其实就是弄个1出来, 1的二进制是这样:

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

我们只需要让这个掩码每次向左移动一位, 然后与目标值求“&”, 就可以判断目标值的当前位是不是1。比如目标值为21, 21的二进制是这样:

0	0	0	1	0	1	0	1
---	---	---	---	---	---	---	---

然后每次移动掩码, 来和当前位进行计算:

The diagram illustrates the step-by-step process of calculating the Hamming weight of the binary number 10000000000000000000000000000000. It consists of 16 rows of 8-bit binary numbers. The process involves shifting the bits of the input number to the right and checking each bit against a mask of 1. The bits that are 1 and align with the mask are highlighted in blue. The count of these highlighted bits is the Hamming weight.

根据分析，完成代码：

```

1 //java
2 public class Solution {
3     public int hammingWeight(int n) {
4         int result = 0;
5         //初始化掩码为1
6         int mask = 1;
7         for (int i = 0; i < 32; i++) {
8             if ((n & mask) != 0) {
9                 result++;
10            }
11            mask = mask << 1;
12        }
13        return result;
14    }
15 }
```

执行结果：

执行用时：1 ms，在所有 Java 提交中击败了 99.76% 的用户

内存消耗：36.7 MB，在所有 Java 提交中击败了 5.04% 的用户

注意：这里判断  $n \& mask$  的时候，千万不要错写成  $(n \& mask) == 1$ ，因为这里你对比的是十进制数。  
(恰好这个题我之前面试别人的时候问到过，对方就直接这么写了...)

## 03、继续优化

位运算小技巧：对于任意一个数，将  $n$  和  $n-1$  进行  $\&$  运算，我们都可以把  $n$  中最低位的 1 变成 0

大家是否还记得昨天学会的技巧，昨天的题目我们通过计算  $n \& n-1$  的值，来判断是否是 2 的幂。今天我们继续使用这个技巧，观察一下，**对于任意一个数，将  $n$  和  $n-1$  进行  $\&$  运算，我们都可以把  $n$  中最低位的 1 变成 0**。比如下面这两对数：

31	0	0	0	1	1	1	1	1
30	0	0	0	1	1	1	1	0
28	0	0	0	1	1	1	0	0
27	0	0	0	1	1	0	1	1

那下面就简单了，只需要不断进行这个操作就可以了。（翻CPP牌子，有没有好评的？）

```
1 //c
2 class Solution {
3 public:
4     int hammingWeight(uint32_t n) {
5         int count = 0;
6         while(n > 0)
7         {
8             n &= (n - 1);
9             ++count;
10        }
11        return count;
12    }
13};
```

肯定有人又是看的一脸懵逼，我们拿 11 举个例子：（注意最后一位1变成0的过程）

11	0	0	0	0	1	0	1	1
10	0	0	0	0	1	0	1	0
11&10=10	0	0	0	0	1	0	1	0
9	0	0	0	0	1	0	0	1
10&9=8	0	0	0	0	1	0	0	0
7	0	0	0	0	0	1	1	1
8&7=0	0	0	0	0	0	0	0	0

所以，今天的问题你学会了吗？评论区留下你的想法！

## 只出现一次的数字(136)

今天仍然分享一道关于位运算颇为简单的题型，同时，从明天开始将会提高难度，大家做好准备。

### 01、题目示例

这道题，大家先想一想是用什么思路进行求解？

#### 第136题：只出现一次的数字

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

#### 说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

#### 示例 1：

1	输入： [2, 2, 1]
2	输出： 1

#### 示例 2：

1 | 输入： [4,1,2,1,2]  
2 | 输出： 4

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

## 02、题目分析

位运算的题目我们已经讲了好几道了，这道也不例外，也是其中一个非常典型的例子！属于必须掌握的题型。

直接分析，我们要找只出现一次的数字，并且已知了其他的数字都只出现了两次。那么这种一听其实就应该想到需使用位运算来进行求解。最好的，就是在读完题目的瞬间，直接条件反射！（当然，如果你现在第一反应是想到通过遍历统计，或者其他如使用hashmap等方式来进行求解，那我觉得你的位运算这块，是有必要加强练习力度的。如果你第一反应，连思路都没有，那我觉得对于整个算法的能力这块，都是比较欠缺的，需要下苦功！）

回到题目，如何使用位运算进行求解呢？对于任意两个数a和b，我们对其使用“异或”操作，应该有以下性质：

- 任意一个数和0异或仍然为自己：

$$a \oplus 0 = a$$

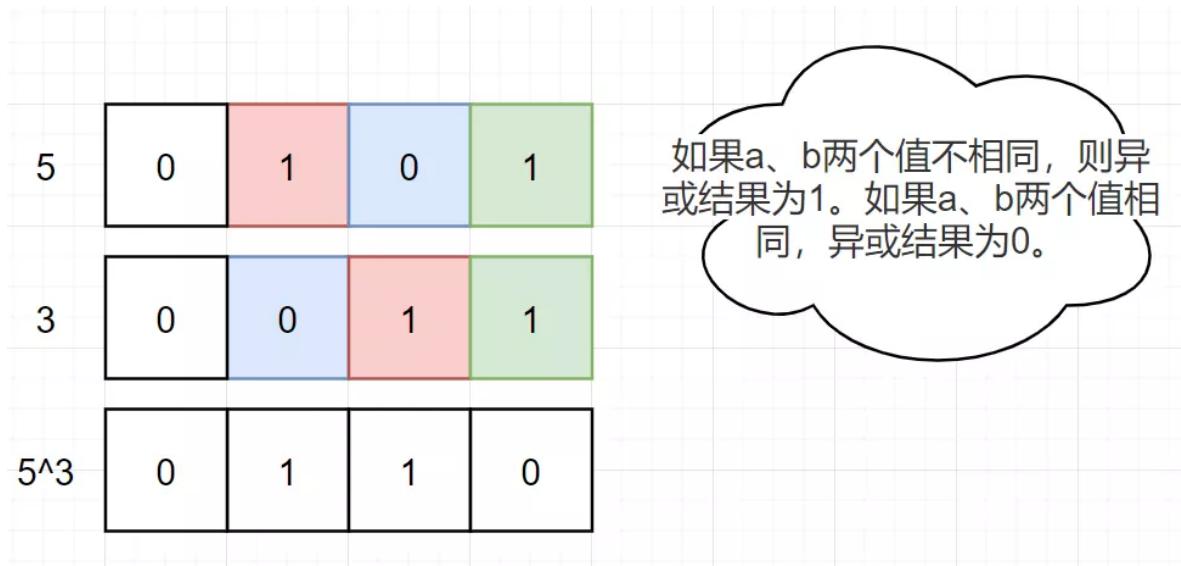
- 任意一个数和自己异或是0：

$$a \oplus a = 0$$

- 异或操作满足交换律和结合律：

$$a \oplus b \oplus a = (a \oplus a) \oplus b = 0 \oplus b = b$$

可能有人直接都不知道异或是什么，所以还是举个例子，比如5异或3，也就是 $5 \oplus 3$ ，也就是 $5^3$ ，是下面这样：



根据分析，得出代码：(c 版本)

```
1 //CPP
2 class Solution {
3 public:
4     int singleNumber(vector<int>& nums) {
5         int ans = 0;
6         for (int num : nums) {
7             ans ^= num;
8         }
9         return ans;
10    }
11};
```

(java版本)

```
1 //JAVA
2 class Solution {
3     public int singleNumber(int[] nums) {
4         int ans = nums[0];
5         for (int i = 1; i < nums.length; i++) {
6             ans = ans ^ nums[i];
7         }
8         return ans;
9     }
10}
```

(python版本)

```
1 //py
2 class Solution:
3     def singleNumber(self, nums: List[int]) -> int:
4         res = 0
5         for i in range(len(nums)):
6             res ^= nums[i]
7         return res
```

执行结果：

执行用时：**40 ms**，在所有 Python3 提交中击败了 **95.42%** 的用户

内存消耗：**15.1 MB**，在所有 Python3 提交中击败了 **51.30%** 的用户

## 03、题目进阶

如果修改上面的题目，除了某个元素只出现一次，其余元素都出现了3次以上，那么该如何求解？

修改一个条件之后，本题的难度大幅度提升！“异或”的方式看起来似乎没办法运用在“其余数出现3次以上”的条件中。那对于这种问题又该如何求解？我这里给出几种思路，大家下去分析一下，明天我会公布这道衍化题型的解决方案：

- 思路1：使用hashmap，统计每个数字出现的次数，最后返回次数为1的数字。。。然后等待一段时间，接到很遗憾的通知。
- 思路2：上面的题目，对于相同的两个数，进行异或运算，我们可以进行“抵消”，那是否可以找到一种方式，来让相同的三个数进行相互抵消呢？
- 思路3：是不是可以通过数学的方式来进行计算？

所以，今天的问题你学会了吗？评论区留下你的想法！

## 只出现一次的数字II(137)

昨天我们在“除了某个元素只出现一次以外，其余每个元素均出现二次”的条件下，通过使用“异或”的操作，找到了只出现一次的元素。那对于其余每个元素均出现三次的case，我们应该如何解决呢？一起来看下吧。

看之前强烈建议复习昨天的文章：

[只出现一次的数字\(136\)](#)

### 01、题目标示例

这种通过改变题中条件，进而增加难度的方式，其实是出题者惯用的一种手段！

#### 第137题：只出现一次的数字II

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现了三次。找出那个只出现了一次的元素。说明：你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

#### 说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

#### 示例 1：

1 | 输入： [2,2,3,2]  
2 | 输出： 3

#### 示例 2：

1 | 输入： [0,1,0,1,0,1,99]  
2 | 输出： 99

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

## 02、HashMap求解

很简单就能想到，说白了就是统计每个元素出现的次数，最终再返回次数为1的元素。但是使用了额外空间。

直接上代码：(go版本)

```
1 func singleNumber(nums []int) int {
2     m := make(map[int]int)
3     for _, k := range nums {
4         //如果是其他语言，请注意对应的判空操作！
5         m[k]
6     }
7     for k, v := range m {
8         if v == 1 {
9             return k
10        }
11    }
12    return 0
13 }
```

执行结果：

执行用时：**11 ms**，在所有 Java 提交中击败了 **6.95%** 的用户

内存消耗：**42.3 MB**，在所有 Java 提交中击败了 **5.19%** 的用户

## 03、数学方式

这个题目曾经在Google很火~目前国内应该也有很多厂子会问到。

原理：[A,A,A,B,B,B,C,C,C] 和 [A,A,A,B,B,B,C]，差了两个C。即：

$$3 \times (a b c) - (a a a b b b c) = 2c$$

也就是说，如果把原数组去重、再乘以3得到的值，刚好就是要找的元素的2倍。举个例子：

sum

14	1	1	1	2	3	3	3
----	---	---	---	---	---	---	---

18	1	1	1	2	2	2	3	3
----	---	---	---	---	---	---	---	---

18-14	2	2	4/2	2				
-------	---	---	-----	---	--	--	--	--

利用这个性质，进行求解：（python代码如下，这里要注意的是，使用int可能会因为超出界限报错）

```
1 class Solution:  
2     def singleNumber(self, nums: List[int]) -> int:  
3         return int((sum(set(nums)) * 3 - sum(nums)) / 2)
```

执行结果：

执行用时：36 ms，在所有 Python3 提交中击败了 97.27% 的用户

内存消耗：14.6 MB，在所有 Python3 提交中击败了 20.75% 的用户

## 04、位运算

对于“每个其余元素，均出现了二次”之所以可以使用“异或”进行求解，原因是因为“异或”操作可以让两数相同归 0。那对于其余元素出现三次的，是不是只要可以让其三者相同归 0，就能达到我们的目的呢？

这个思想可能比较简单，但是要让大家理解，还是有一定难度。如果大家准备好了，可以开始往下看。我看过了leetcode上的题解，很多都是直接扔出来一个公式，其实讲的我认为并不是特别清楚。所以我打算先把本题退化到“每个其余元素，均出现二次”的case来进行分析一下。

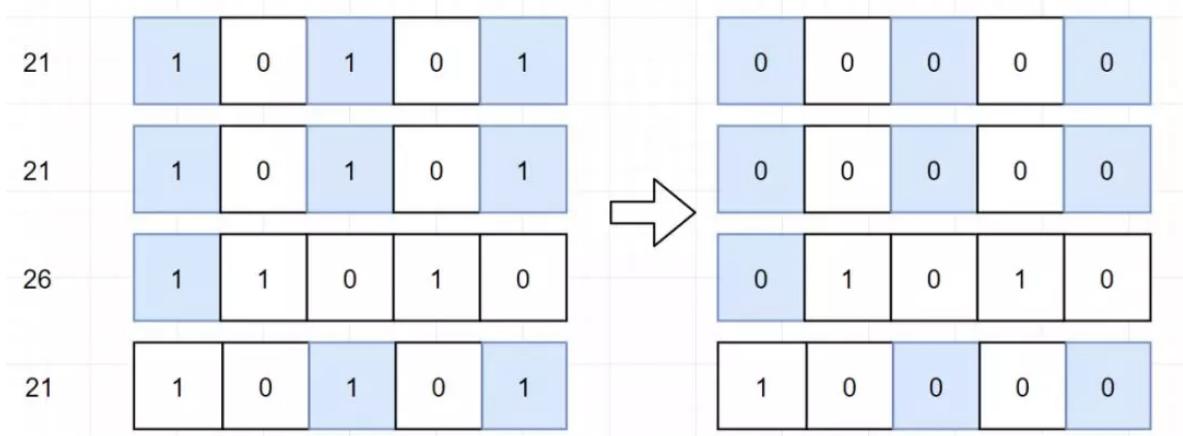
假如我们有 [21,21,26] 三个数，是下面这样：

21	1	0	1	0	1
21	1	0	1	0	1
26	1	1	0	1	0

回想一下，之所以能用“异或”，其实我们是完成了一个 **同一位上有2个1清零** 的过程。上面的图看起来可能容易，如果是这样（下图应为 $26 \wedge 21$ ）：

21	1	0	1	0	1
26	1	1	0	1	0
26&21=15	0	1	1	1	1
21	1	0	1	0	1
26	1	1	0	1	0

那对于“每个其余元素，均出现了三次”也是一样，如果我们可以完成一个**同一位上的三个1清零的过程**，也就是  $a ? a ? a = 0$ ，问题则迎刃而解。那因为各语言中都没有这样一个现成的方法可以使用，所以我们需要构造一个。（想象一下，位运算也是造出来的对不对？）



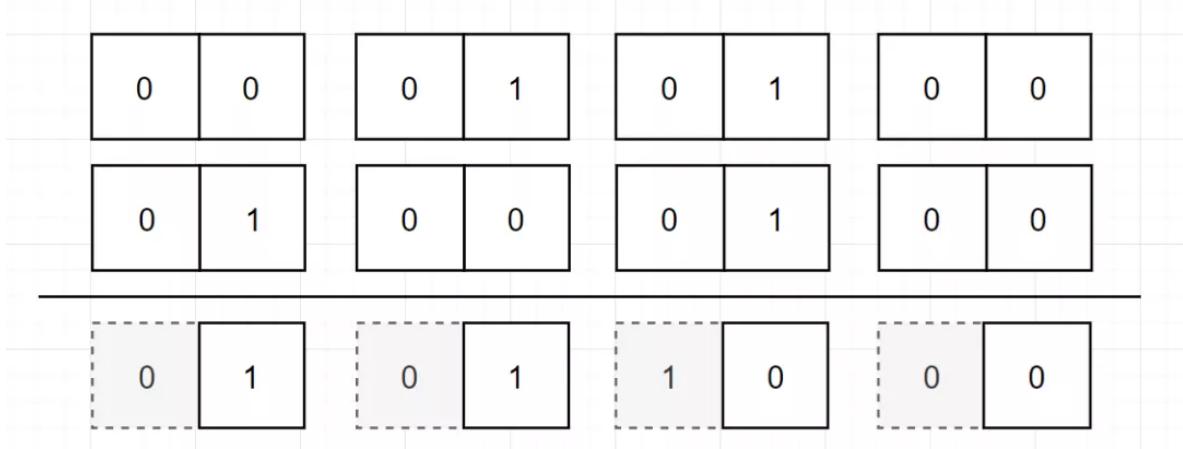
如何构造，这里先说第一种方法（注意，到这里我们的问题已经转化成了定义一种  $a ? a ? a = 0$  的运算），观察一下“异或”运算：

$$1 \wedge 1 = 0$$

$$1 \wedge 0 = 1$$

$$0 \wedge 1 = 1$$

是不是可以理解为，其实就是二进制的加法，然后砍掉进位呢？

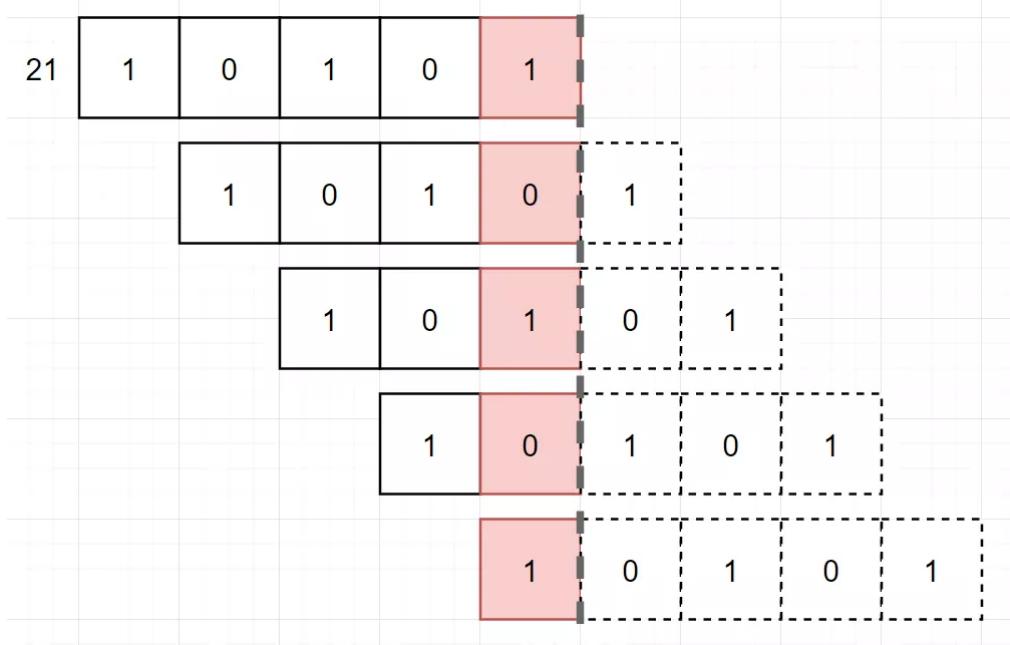


砍掉进位的过程，是不是又可以理解为对 2 进行取模，也就是取余。到了这里，问题已经非常非常明确了。那我们要完成一个  $a ? a ? a = 0$  的运算，是不是其实就是让其二进制的每一位数都相加，最后再对 3 进行一个取模的过程呢？（一样，如果要定义一个  $a ? a ? a ? a = 0$  的运算，那就最后对 4 进行取模就可以了）

```

1 //go
2 func singleNumber(nums []int) int {
3     number, res := 0, 0
4     for i := 0; i < 64; i++ {
5         //初始化每一位1的个数为0
6         number = 0
7         for _, k := range nums {
8             //通过右移i位的方式，计算每一位1的个数
9             number += (k >> i) & 1
10        }
11        //最终将抵消后剩余的1放到对应的位数上
12        res |= (number) % 3 << i
13    }
14    return res
15 }
```

如果对上面的代码不能理解，可以看看这个图，假设只有一个数 [21]，我们通过不断右移的方式，获取其每一位上的1。当然，这里因为余数都是1，所以肯定都保留了下来，然后与 1 进行“与”运算，最终再将其放入到对应的位数上。



执行结果：

执行用时：8 ms，在所有 Go 提交中击败了 50.29% 的用户

内存消耗：3.6 MB，在所有 Go 提交中击败了 100.00% 的用户

在上面的代码中，**我们通过一个number，来记录每一位数出现的次数**。但是缺点是，我们记录了64位（Go语言中，int为32位以上）

`int` is a signed integer type that is at least 32 bits in size. It is a distinct type, however, and not an alias for, say, `int 32`.

那如果我们可以同时对所有位进行计数，是不是就可以简化过程。因为我们的目的是把每一位与3取模进行运算，是不是就可以理解为其实是一个**三进制**。如果大家听不懂三进制的话，可以简单理解为3次一循环，也就是 00 - 01 - 10 - 11。但是又因为对于 11 这种情况，我们需要砍掉（上面已经说过了，相当于 11 - 00 的转化），所以我们就只有3个状态，00 - 01 - 10，所以我们采用 `a` 和 `b` 来记录状态。其中的状态转移过程如下：

<b>a</b>	<b>b</b>	<b>next</b>	<b>a'</b>	<b>b'</b>
0	0	1	0	1
0	1	1	1	0
1	0	1	0	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0

这里 `a'` 和 `b'` 的意思代表着 `a` 和 `b` 下一次的状态。`next` 代表着下一个 bit 位对应的值。然后这是什么，不就是状态机嘛。。。我们通过 `a` 和 `b` 的状态变化，来完成次数统计。

然后因为此图复杂，将其分别简化成 a 和 b 的卡诺图（卡诺图是逻辑函数的一种图形表示。两逻辑相邻项，合并为一项，保留相同变量，消去不同变量。先  $b'$  后  $a'$ ）

$next \setminus a, b$	00	01	11	10
1	1	0	X	0
0	0	1	X	0

$next \setminus a, b$	00	01	11	10
1	0	1	X	0
0	0	0	X	1

然后我们根据卡诺图（这个卡诺图其实并不难看。。如果学习一下话，还是挺简单的。）写出关系式：

$$a' = (a \& \sim next) | (b \& next)$$

$$b' = (\sim a \& \sim b \& next) | (b \& \sim next)$$

然后就是套公式：（Java代码，注意Go语言中是不天然支持  $\sim$  这种运算的）

```

1 class Solution {
2     public int singleNumber(int[] nums) {
3         int a = 0, b = 0, tmp = 0;
4         for (int next : nums) {
5             tmp = (a & ~next) | (b & next);
6             b = (~a & ~b & next) | (b & ~next);
7             a = tmp;
8         }
9         return b;
10    }
11 }
```

当然，其实题解还可以再进一步优化，其实就是化简上一步中的公式：

```

1 class Solution {
2     public int singleNumber(int[] nums) {
3         int a = 0, b = 0;
4         for (int next : nums) {
5             b = (b ^ next) & ~a;
6             a = (a ^ next) & ~b; 7
7         }
8     }
9 }
```

当然，这个解法就相当牛皮了，反正我第一次做肯定想不到。。。我看了一下，第一个给出这个解法的人，应该是一位国外的工程师（某扣上面有很多人其实都是把题解翻译过来的，当然我有时也会哈哈哈，我觉得这某种意义上讲也是一个好的现象，挺好）不过毕竟非原创，还是得说明一下！

## 137. Single Number II

Hot Newest to Oldest Most Votes Most Posts Recent Activity Oldest to Newest Search topics or comments... New +

Detailed explanation and generalization of the bitwise operation method for single numbers  
fun4LeetCode created at: April 13, 2015 10:06 AM | Last Reply: faanged February 18, 2020 2:20 AM ▲ 1.0K 82.3K

Challenge me , thx  
against1 created at: June 25, 2014 3:14 PM | Last Reply: krishnan09 January 8, 2020 10:33 AM ▲ 797 135.7K

Java O(n) easy to understand solution, easily extended to any times of occurance  
yfcheng created at: April 21, 2016 2:22 AM | Last Reply: Tjianik February 4, 2020 4:24 PM ▲ 492 37.3K

An General Way to Handle All this sort of questions.  
ziyihao created at: August 29, 2015 1:34 PM | Last Reply: majestyhao November 20, 2019 3:48 AM ▲ 445 57.7K

Accepted code with proper Explanation. Does anyone have...  
Deepalaxmi created at: August 24, 2014 4:06 PM | Last Reply: JustKeepCodinggg February 12, 2020 5:39 AM ▲ 163 19.1K

A general C++ solution for these type problems  
HelloW0rld created at: September 7, 2015 5:51 PM | Last Reply: sm\_c January 10, 2020 12:42 PM ▲ 116 13.4K

总之，今天的题目，有一定的难度！希望大家动脑动手动脚，认真想想。

所以，今天的问题你学会了吗？评论区留下你的想法！

## 缺失数字(268)

上一篇题目的难度可能对很多同学引起了不适，今天将回归一道比较简单的题目，大概耗时2-3分钟即可学习！

有兴趣回顾上一篇题目的：

[只出现一次的数字II\(137\)](#)

### 01、题目示例

本题比较简单哈~尽可能多的给出解法吧！

#### 第268题：缺失数字

给定一个包含  $0, 1, 2, \dots, n$  中  $n$  个数的序列，找出  $0 \dots n$  中没有出现在序列中的那个数。

示例 1：

1 | 输入： [3,0,1]  
2 | 输出： 2

### 示例 2：

1 | 输入： [9,6,4,2,3,5,7,0,1]  
2 | 输出： 8

### 说明：

你的算法应具有线性时间复杂度。你能否仅使用额外常数空间来实现？

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

## 02、题目分析

说高斯公式，估计大家听着懵逼，其实也就是那个  $1 + 2 + 3 + \dots + n = \frac{(1 + n) * n}{2}$ ，即：

$$\sum_{i=0}^n i = \frac{n(n + 1)}{2}$$

首先求出数组的和，然后再利用公式求出前 $n$ 项之和，最终求差值，即为缺失的值！比如下面长度为4的数组，缺失4。

2	3	1	5
---	---	---	---

- $2 + 3 + 1 + 5 = 11$
- $(1 + 5) * 5 / 2 = 15$
- $15 - 11 = 4$

根据分析完成题解：（翻一个CPP牌子吧）

```
1 //CPP
2 class Solution {
3     public:
4     int missingNumber(vector<int>& nums) {
5         int length=nums.size();
6         int result=(length + 1)*length/2;
7         for(int e:nums)
8             result-=e;
9         return result;
10    }
11 }
```

时间复杂度O(N), 空间复杂度O(1)

执行结果：

执行用时：24 ms，在所有 C++ 提交中击败了 66.37% 的用户

内存消耗：18.9 MB，在所有 C++ 提交中击败了 5.09% 的用户

## 03、位运算求解

位运算的方式，本质和数学法一样，都是通过与无序序列抵消，然后找到缺失值。所以不能说哪个更好，都掌握最好~

直接使用“异或”进行求解。这个其实讲了好多次了，就是利用“**两个相同的数，使用异或可以相消除**”的原理。

先给一个Go语言的示例：

```
1 //Go
2 func missingNumber(nums []int) int {
3     result := 0
4     for i,k := range nums {
5         result ^= k ^ i
6     }
7     return result ^ len(nums)
8 }
```

再给一个Java的版本：

```
1 //java
2 class Solution {
3     public int missingNumber(int[] nums) {
4         int res = 0;
5         for(int i = 0; i < nums.length; i++)
6             res ^= nums[i] ^ i;
7         return res ^ nums.length;
8     }
9 }
```

为了照顾各语言大爷们的情绪，我还是会尽可能的多给出几种语言示例，但是，请记住：**算法思想才是最重要的。**

最后再补一个python的吧，毕竟这种语言，对于这种短短的题目，往往都可以弄出来一行代码求解的骚操作....

```
1 #python
2 class Solution:
3     def missingNumber(self, nums: List[int]) -> int:
4         return sum(range(len(nums) + 1)) - sum(nums)
```

所以，今天的问题你学会了吗？评论区留下你的想法！

## 二分法系列

### 爱吃香蕉的珂珂 (875)

#### 01、题目标示例

不知道为什么叫做爱吃香蕉的阿珂，难道不应该是爱吃香蕉的猴子么...或者爱吃队友的露娜么？

##### 第875题：阿珂喜欢吃香蕉

这里总共有  $N$  堆香蕉，第  $i$  堆中有  $piles[i]$  根香蕉。警卫已经离开了，将在  $H$  小时后回来。阿珂可以决定她吃香蕉的速度  $K$ （单位：根/小时），每个小时，她将选择一堆香蕉，从中吃掉  $K$  根。

如果这堆香蕉少于  $K$  根，她将吃掉这堆的所有香蕉，然后这一小时内不会再吃更多的香蕉。

珂珂喜欢慢慢吃，但仍然想在警卫回来前吃掉所有的香蕉。返回她可以在  $H$  小时内吃掉所有香蕉的最小速度  $K$ （ $K$  为整数）。

##### 示例 1：

```
1 | 输入: piles = [3,6,7,11], H = 8  
2 | 输出: 4
```

### 示例 2:

```
1 | 输入: piles = [30,11,23,4,20], H = 5  
2 | 输出: 30
```

### 示例 3:

```
1 | 输入: piles = [30,11,23,4,20], H = 6  
2 | 输出: 23
```

### 提示:

```
1 | 1 <= piles.length <= 10^4  
2 | piles.length <= H <= 10^9  
3 | 1 <= piles[i] <= 10^9
```

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

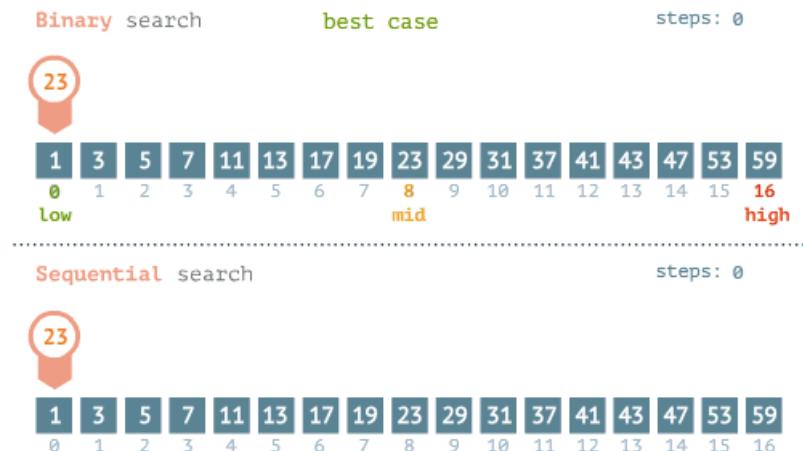
## 02、二分查找

十个二分九个错，该算法被形容 "思路很简单，细节是魔鬼"。第一个二分查找算法于 1946 年出现，然而第一个完全正确的二分查找算法实现直到 1962 年才出现。下面的二分查找，其实是二分查找里最简单的一个模板，在后面的文章系列里，我将逐步为大家讲解二分查找的其他变形形式。

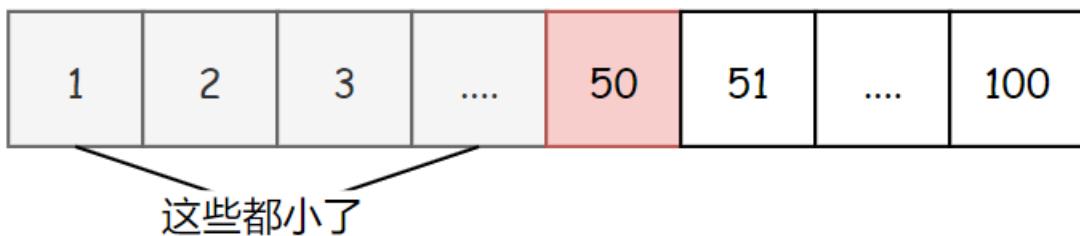
二分查找是计算机科学中最基本、最有用的算法之一。它描述了**在有序集合中搜索特定值的过程**。一般二分查找由以下几个术语构成：

- 目标 Target —— 你要查找的值
- 索引 Index —— 你要查找的当前位置
- 左、右指示符 Left, Right —— 我们用来维持查找空间的指标
- 中间指示符 Mid —— 我们用来应用条件来确定我们应该向左查找还是向右查找的索引

在最简单的形式中，二分查找对具有指定左索引和右索引的**连续序列**进行操作。我们也称之为**查找空间**。二分查找维护查找空间的左、右和中间指示符，并比较查找目标；如果条件不满足或值不相等，则清除目标不可能存在的那一半，并在剩下的一半上继续查找，直到成功为止。



举例说明：比如你需要找1-100中的一个数字，你的目标是用最少的次数猜到这个数字。你每次猜测后，我会说大了或者小了。而你只需要每次猜测中间的数字，就可以将余下的数字排除一半。



不管我心里想的数字如何，你在7次之内都能猜到，这就是一个典型的二分查找。每次筛选掉一半数据，所以我们也称之为**折半查找**。一般而言，对于包含n个元素的列表，用二分查找最多需要 $\log_2 n$ 步。



当然，一般题目不太可能给你一个如此现成的题型，让你上手就可以使用二分，所以我们需要思考，如何来构造一个成功的二分查找。大部分的二分查找，基本都由以下三步组成：

- 预处理过程（大部分场景就是对未排序的集合进行排序）
- 二分查找过程（找到合适的循环条件，每一次将查找空间一分为二）
- 后处理过程（在剩余的空间中，找到合适的目标值）

了解了二分查找的过程，我们对二分查找进行**一般实现**（这里给出一个Java版本，比较正派的代码，没有用一些缩写形式）

```
1 //JAVA
2 public int binarySearch(int[] array, int des) {
3     int low = 0, high = array.length - 1;
4     while (low <= high) {
```

```
5     int mid = low + (high - low) / 2;
6     if (des == array[mid]) {
7         return mid;
8     } else if (des < array[mid]) {
9         high = mid - 1;
10    } else {
11        low = mid + 1;
12    }
13}
14return -1;
15}
```

注意：上面的代码，mid 使用  $low + (high - low)/2$  的目的，是防止 high low 溢出内存。

为什么说是一般实现？

1、根据边界的不同（开闭区间调整），有时需要弹性调整low与high的值，以及循环的终止条件。

2、根据元素是否有重复值，以及是否需要找到重复值区间，有时需要对原算法进行改进。

那上面我们说了，一般二分查找的过程分为：预处理 - 二分查找 - 后处理，上面的代码，就没有后处理的过程，因为在每一步中，你都检查了元素，如果到达末尾，也已经知道没有找到元素。

总结一下一般实现的几个条件：

- 初始条件：left = 0, right = length-1
- 终止：left > right
- 向左查找：right = mid-1
- 向右查找：left = mid +1

请大家记住这个模板原形，在后面的系列中，我们将介绍二分查找其他的模板类型。

## 03、题目分析

简单复习了二分查找，我们来看本题。

注意，绝大部分「在递增递减区间中搜索目标值」的问题，都可以转化为二分查找问题。并且，二分查找的题目，基本逃不出三种：找特定值，找大于特定值的元素（上界），找小于特定值的元素（下界）。

而根据这三种，代码又最终会转化为以下这些问题：

- low、high 要初始化为 0、n-1 还是 0、n 又或者 1、n？
  - 循环的判定条件是 low < high 还是 low <= high？
  - if 的判定条件应该怎么写？
  - if 条件正确时，应该移动哪边的边界？
  - 更新 low 和 high 时，mid 如何处理？

处理好了上面的问题，自然就可以顺利解决问题。将上面的思想代入到本题，我们要找“阿珂在 H 小时吃掉所有香蕉的最小速度 K”。那最笨的就是阿珂吃的特别慢，每小时只吃掉 1 根香蕉，然后我们逐渐递增阿珂吃香蕉的速度到 i，刚好满足在 H 小时可以吃掉所有香蕉，此时 i 就是我们要找的最小速度。当然，我们没有这么笨，所以可以想到使用二分的思想来进行优化。

然后就简单了，我们寻找二分查找模板中初始条件和终止条件（注意，这里的 high、low、mid 都代表的是速度）：

```
1 //这里我把最小速度定义成了1，可能大家会觉得奇怪，模板里不是0吗？
2 //所以这里我其实是想说，算法千变万化，大家不要生搬硬套。
3 //从字面理解，如果定义成0，意味着阿珂会选择一个香蕉都不吃，难道阿珂傻？
4 var low = 1
5 //最大的速度，当然等于吃掉最大一堆的香蕉，毕竟一小时只能吃一堆，再大也没有意义
6 var high = maxArr(piles)
7 //中间速度
8 var mid = (low + high) / 2
```

```
1 //JAVA
2 public class Solution {
3     public int minEatingSpeed(int[] piles, int H) {
4         int maxVal = 1;
5         for (int pile : piles) {
6             maxVal = Math.max(maxVal, pile);
7         }
8         int left = 1;
9         int right = maxVal;
10        while (left < right) {
11            int mid = (left + right) >> 1;
12            if (canEat(piles, mid, H)) {
13                left = mid + 1;
14            } else {
15                right = mid;
16            }
17        }
18        return left;
19    }
20
21    private boolean canEat(int[] piles, int speed, int H) {
22        int sum = 0;
23        for (int pile : piles) {
24            //向上取整
25            sum += Math.ceil(pile * 1.0 / speed);
26        }
27        return sum > H;
28    }
29 }
```

```
28 }  
29 }
```

执行结果:

执行用时: 10 ms , 在所有 Java 提交中击败了 83.68% 的用户

内存消耗: 42.2 MB , 在所有 Java 提交中击败了 57.26% 的用户

额外补充 (昨天有人问我的问题) :

- 第一: 就是不需要再对原数组进行排序了, 因为我们是把这样一个问题转化为二分查找的问题, 而通过 canEat, 计算在 mid 速度下吃完 piles 共需要多少小时。已经将 piles 利用进去了, 所以此时并不需要对 piles 排序。
- 第二: 就是昨天有人私下问我, 对  $(pile \ speed - 1)/speed$  不能理解。这个其实就是一个向上取整的小技巧, 相当于  $\text{Math.ceil}(pile * 1.0 / speed)$ 。

留下一个问题, 假如我们的阿珂就是笨笨的, 将 low 初始化成了 0, 此时的循环条件应该如何写? if 条件如果成立, low 和 high 又该如何进行调整? 大家可以尝试一下! (一百个人有一百个二分, 不要妄图和别人写出一模一样的代码, 这是没有意义的。只有自己理解了, 一步步的分析问题, 写出自己的代码, 才是真正属于你的)

所以, 今天的问题你学会了吗? 评论区留下你的想法!

## x的平方根 (69)

今天继续为大家分享二分法系列篇的内容, 看一道比较简单的题目。

### 01、题目分析

这道题目是比较简单的, 但我认为同时也是非常经典的, 建议大家掌握!

#### 第69题: x的平方根

计算并返回 x 的平方根, 其中 x 是非负整数。由于返回类型是整数, 结果只保留整数的部分, 小数部分将被舍去。

PS: 建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

## 02、二分查找

使用二分法来完成平方根还是比较容易被想到的，在有限的“区间”中，每次通过筛选一半的元素，到最终只剩下一个数（收敛），这个数就是题目要求的取整的平方根整数。

根据之前说过的二分法模板，要使用二分法，我们当然要找到Left, Right, Mid，那在这里，Mid自然被作为最终我们要找的平方根的值（不像上一道题，Mid是作为速度，不太容易被想到），而Left和Right，我们采用1和 $x/2$ 。

Left设置为1比较容易理解，因为我们可以直接处理掉 $x$ 为0的情况（当然，也可以把Left初始化为2，然后我们额外处理0和1的情况，我之前说过，二分法一万个人有一万种写法，只要能解释清楚，那就是你自己的）。但是为什么Right是 $x/2$ 呢？

我们看一下下面这些数的值：

x	$\sqrt{x}$	整数平方根	$x/2$
2	1.414	1	1
3	1.732	1	1.5
4	2	2	2
5	2.236	2	2.5

很容易观察出，当 $x > 2$ 时，它的整数平方根一定小于等于 $x/2$ 。即有 $0 < \text{整数平方根} \leq x/2$ 。所以我们的问题转化为在 $[0, x/2]$ 中找一个特定值，满足二分查找的条件。（当然，如果没有想到使用 $x/2$ 作为Right而直接使用 $x$ ，其实也是可以的）

剩下的逻辑就很简单了，我们不停缩小mid的范围，如果最终平方大于 $x$ 就放回它前面一个值，否则就正常返回，直到两边的边界完全收敛。

```
1 public class Solution {
2     public int mySqrt(int x) {
3         if (x == 0) return 0;
4         long left = 1;
5         long right = x / 2;
6         while (left < right) {
7             //注意这一行代码
8             long mid = (right + left) / 2 + 1;
9             if (mid * mid > x) {
10                 right = mid - 1;
11             } else {
12                 left = mid;
13             }
14         }
15         return (int) left;
16     }
17 }
```

上面的代码，有三处需要进行讲解：

- 第一，就是这里将 left 和 right 都设置为了 long，这是因为担心超出界限。同时，也正是因为设置为了 long，所以下面我可以直接使用 right - left，而不用担心报错。
- 第二，还是这行代码，大家肯定会疑惑，为什么要在  $(right - left) / 2$  后面再加1。这其实是一种技巧，一般人我不告诉他。因为在面试的时候，我们往往需要快速写出freebug 的代码，但是如果遇到二分的题目，你很可能会不停的纠结 mid 到底如何设置，是左边界还是右边界。其实，面试官大多时候，并不需要你写出一个非常非常标准的二分，找到绝对的中值。那这里我们是不是就可以偷懒了？我们通过略微增大搜索空间，来降低自己代码的难度，并且因为代码完美的通过，别人还会觉得你牛逼。
- 第三，这点本来不需要额外说明的，正是在第二的基础上，我们通过不停的缩小搜索空间，最终 left 就变成我们要找的 mid 值，所以直接返回 left 就可以了。（因为昨天的题目有人问我，问为什么最后是返回 left，而不是 mid）其实这也勉强算是一种技巧，一般熟悉二分的人，都不会多余的去写一个mid，而是通过这种返回边界的方式，来找到目标值。这有2个好处，第一是可以让代码更加简洁，第二是不容易出错。

这里，我给出上面代码的三种 mid 衍化形式，大家琢磨琢磨：

```

1 public class Solution {
2     public int mySqrt(int x) {
3         if (x == 0) return 0;
4         long left = 1;
5         long right = x / 2;
6         while (left < right) {
7             #1 long mid = (right + left) / 2 + 1;
8             #2 long mid = left + (right - left + 1) / 2;
9             #3 long mid = (left + right + 1) >> 1
10            if (mid > x / mid) {
11                right = mid - 1;
12            } else {
13                left = mid;
14            }
15        }
16        return (int) left;
17    }
18 }
```

同时，我也再给出一个没想到将 Right 设置为  $x/2$  的解法，这个解法是非常正派的，特别的适合新手。

```

1 //java
2 public class Solution {
3
4     public int mySqrt(int x) {
5         int left = 0;
6         int right = x;
7         while (left <= right) {
8             long mid = (left + right) / 2;
9             if (mid * mid == x)
10                 return (int) mid;
11             else if (mid * mid < x)
12                 left = (int) (mid + 1);
13             else
14                 right = (int) (mid - 1);
15     }
16 }
```

```
16     return right;
17 }
18
19 }
```

读算法文章的目的，是跟着对方的思路走，而不是说这个我会了，就不需要学习了，这样恐怕进步很难。本题自然可以通过 **牛顿法**，**递归** 等多种方式求解，但并不是我想说的。

## 03、一点建议

我拉出来讲这道题的原因，绝对不是说你会了，知道怎么样做了就可以了。我是希望通过本题，各位去深度思考二分法中几个元素的建立过程，比如 **Left** 和 **Right** 我们应该如何去设置，如本题中 **Right** 既可以设置为  $x$  也可以设置为  $x/2$ ；又比如 **mid** 值该如何计算。大家一定要明确 **mid** 的真正含义有两层，第一：大部分题目最后的 **mid** 值就是我们要找的目标值 第二：我们通过 **mid** 值来收敛搜索空间。

那么问题来了，如何可以彻底掌握二分法？初期我并不建议大家直接去套模板，这样意义不是很大，因为套模板很容易边界值出现错误（当然，也可能我的理解还不够深入，网上有很多建议是去直接套模板的）我的建议是：**去思考二分法的本质，了解其通过收敛来找到目标的内涵\*\***，对每一个二分的题目都进行深度剖析，多分析别人的答案。**你得知道**，每一个答案，背后都是对方的思考过程\*\*。从这些过程中抽茧剥丝，最终留下的，才是二分的精髓。也只有到这一刻，我认为才可以真正的说一句掌握了二分。毕竟模板的目的，也是让大家去思考模板背后的东西，而不是模板本身。

所以，今天的问题你学会了吗？评论区留下你的想法！

## 第一个错误的版本 (287)

如果你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的，所以我们需要回滚代码，那如何能找到错误的版本呢？

### 01、题目示例

#### 第278题：第一个错误的版本

假设你有  $n$  个版本  $[1, 2, \dots, n]$ ，你想找出导致之后所有版本出错的第一个错误的版本。

你可以通过调用 `bool isBadVersion(version)` 接口来判断版本号 `version` 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的次数。

## 示例:

```
1 | 给定 n = 5, 并且 version = 4 是第一个错误的版本。
2 |
3 | 调用 isBadVersion(3) -> false
4 | 调用 isBadVersion(5) -> true
5 | 调用 isBadVersion(4) -> true
6 |
7 | 所以, 4 是第一个错误的版本。
```

## 02、推导过程

这个题目还是相当简单的....我拿出来讲的原因，是因为我的开发生涯中，真的遇到过这样一件事。当时我们做一套算薪系统，算薪系统主要复杂在业务上，尤其是销售的薪资，设计到数百个变量，并且还需要考虑异动（比如说销售A是团队经理，但是下调到B团队成为一名普通销售，然后就需要根据A异动的时间，来切分他的业绩组成。同时，最恶心的是，普通销售会影响到其团队经理的薪资，团队经理又会影响到营业部经理的薪资，一直到最上层，影响到整个大区经理的薪资构成）要知道，当时我司的销售有近万名，每个月异动的人就有好几千，这是非常非常复杂的。然后我们遇到的问题，就是同一个月，有几十个团队找上来，说当月薪资计算不正确（放在个人来讲，有时候差个几十块，别人也是会来找的）最后，在一阵漫无目的的排查之后，我们采用二分的思想，通过切变量，最终切到错误的异动逻辑上，进行了修正。

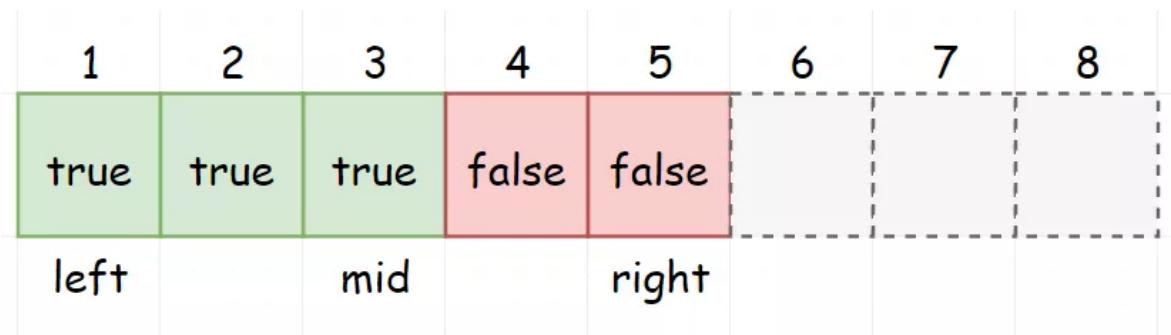
回到本题，我们当然可以一个版本一个版本的进行遍历，直到找到最终的错误版本。但是如果是这样，还讲毛线呢。。。

```
1 //JAVA
2 public int firstBadVersion(int n) {
3     for (int i = 1; i < n; i++ ) {
4         if (isBadVersion(i)) {
5             return i;
6         }
7     }
8     return n;
9 }
```

我们自然是采用二分的思想，来进行查找。举个例子，比如我们版本号对应如下：

1	2	3	4	5	6	7	8
true	true	true	false	false	false	false	false
left				mid			right

如果中间的mid如果是错误版本，那我们就知道 mid 右侧都不可能是第一个错误的版本。那我们就令 right = mid，把下一次搜索空间变成[left, mid]，然后自然我们很顺利查找到目标。



根据分析，代码如下：

```

1 //JAVA
2 public int firstBadVersion(int n) {
3     int left = 1;
4     int right = n;
5     while (left < right) {
6         int mid = left + (right - left) / 2;
7         if (isBadVersion(mid)) {
8             right = mid;
9         } else {
10            left = mid + 1;
11        }
12    }
13    return left;
14 }
```

额外补充：

- 请大家习惯这种返回left的写法，保持代码简洁的同时，也简化了思考过程，何乐而不为呢。

当然，代码也可以写成下面这个样子（是不是感觉差点意思？）

```

1 //JAVA
2 public class Solution extends VersionControl {
3     public int firstBadVersion(int n) {
4         int left = 1;
5         int right = n;
6         int res = n;
7         while (left <= right) {
8             int mid = left + ((right - left) >> 1);
9             if (isBadVersion(mid)) {
10                 res = mid;
11                 right = mid - 1;
12             } else {
13                 left = mid + 1;
14             }
15         }
16         return res;
17     }
18 }
```

据查，医书有服用响豆的方法，响豆就是槐树果实在夜里爆响的，这种豆一棵树上只有一个，辨认不出来。取这种豆的方法是，在槐树刚开花时，就用丝网罩在树上，以防鸟雀啄食。结果成熟后，缝制许多布囊贮存豆荚。夜里用来当枕头，没有听到声音，便扔掉。就这么轮着枕，肯定有一个囊里有爆响声。然后把这一囊的豆类又分成几个小囊装好，夜里再枕着听。听到响声再一分二，装进囊中枕着听。这么分下去到最后只剩下两颗，再分开枕听，就找到响豆了。

前三章的题目，都是比较简单的，目的是让大家对二分能有一些深层次的思考。下面我就会增大难度，为大家讲解一些，不那么容易可以直接想到使用二分法进行求解的题目，希望大家支持！

## 其他题目

### 螺旋矩阵（54）

今天为大家分享一道关于螺旋矩阵的问题。话不多说，直接看题目吧。

#### 01、题目分析

##### 第54题：螺旋矩阵

定一个包含  $m \times n$  个元素的矩阵（ $m$  行,  $n$  列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

##### 示例 1：

```
1 | 输入：  
2 | [  
3 |   [ 1, 2, 3 ],  
4 |   [ 4, 5, 6 ],  
5 |   [ 7, 8, 9 ]  
6 | ]  
7 | 输出： [1,2,3,6,9,8,7,4,5]
```

##### 示例 2：

```
1 | 输入：  
2 | [  
3 |   [1, 2, 3, 4 ],  
4 |   [5, 6, 7, 8 ],  
5 |   [9,10,11,12]  
6 | ]  
7 | 输出： [1,2,3,4,8,12,11,10,9,5,6,7]  
8 |  
9 |
```

## 02、题目分析

本题的思路，在于**模拟螺旋的移动轨迹**。

问题的难点，在于**想明白模拟过程中会遇到什么问题**。

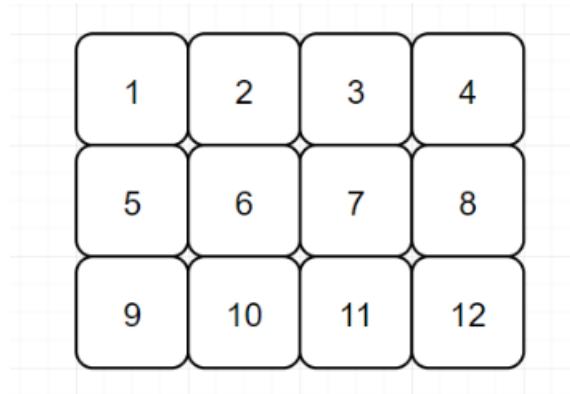
那模拟的过程中会遇到什么样的问题？**边界处理**。

因为只有我们能找到边界（边界包括：1、数组的边界 2、已经访问过的元素），才可以通过“**右，下，左，上**”的方向来进行移动。同时，每一次**碰壁**，就可以调整到下一个方向。

思路明确了，我们看一下整个过程。假如我们的数组为：

```
[  
    [1, 2, 3, 4],  
    [5, 6, 7, 8],  
    [9, 10, 11, 12]  
]
```

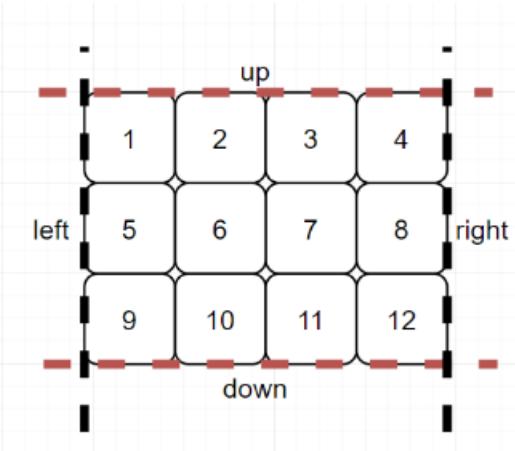
数组如下图所示：



我们首先对其设置好四个边界：

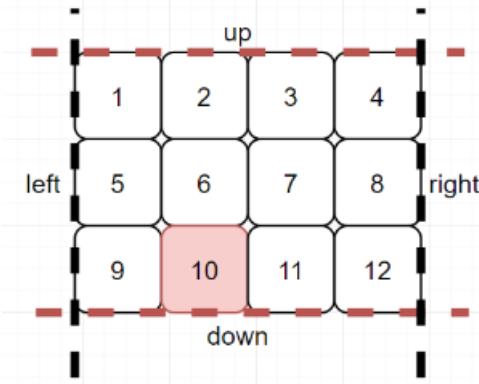
```
1 up := 0  
2 down := len(matrix) - 1  
3 left := 0  
4 right := len(matrix[0]) - 1
```

如下图所示：

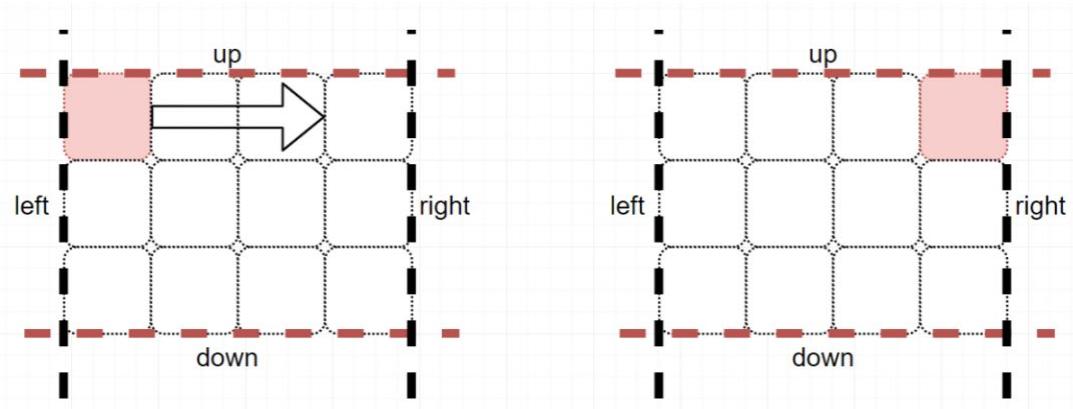


同时，我们定义x和y，来代表行和列。

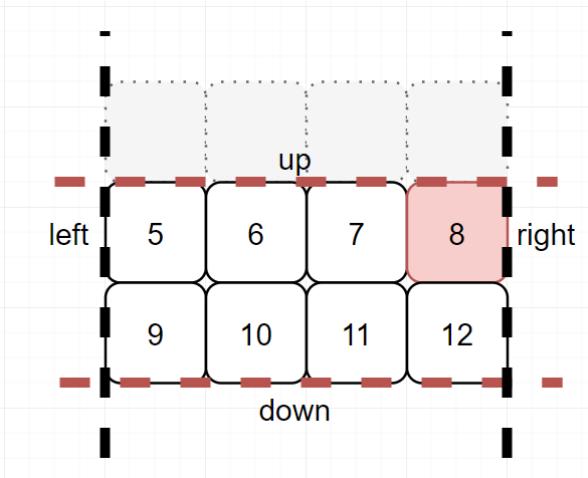
如 $x=2$ ,  $y=1$ , 则  $\text{arr}[2][1]=10$  (第3行第2列)



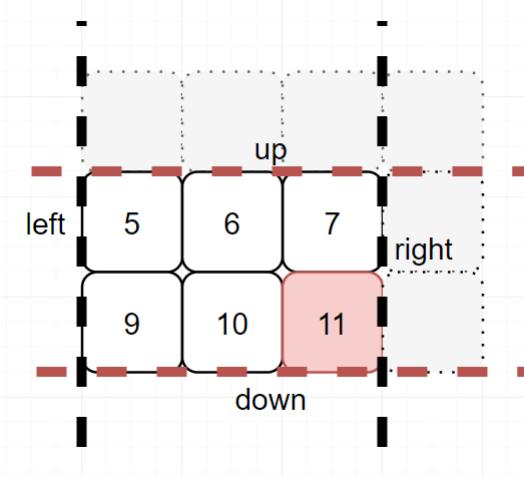
然后我们从第一个元素开始行军 ( $y=\text{left}$ ) , 完成第一行的遍历, 直到碰壁。 ( $y \leq \text{right}$ )



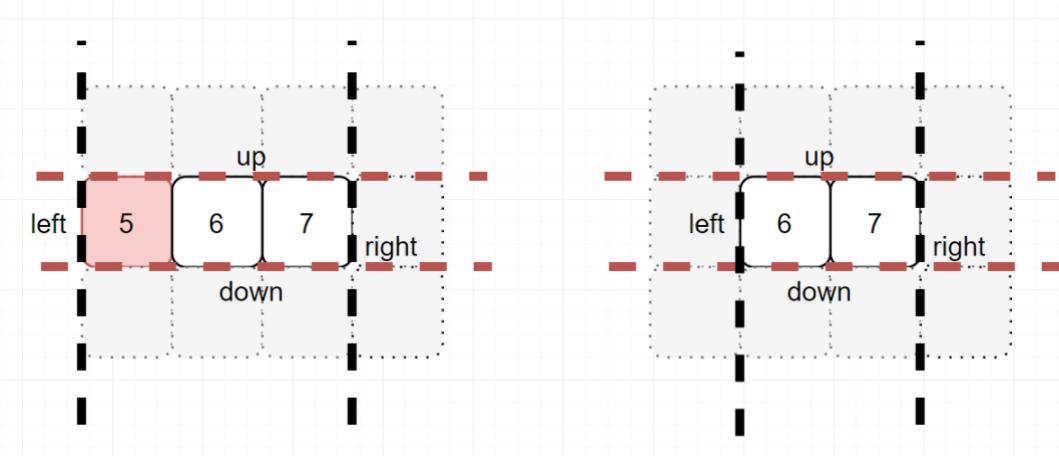
下面关键的一步来了, 因为第一行已经走过了, 我们将上界下调\*\* ( $\text{up}++$ ) \*\*, 同时转弯向下走。



直到碰到底部时 ( $x \leq down$ ) , 我们将**右界左调 (right--)** , 转弯向左走。



后面向左和向上，分别完成**下界上调 (down--)** 和**左界右调 (left++)**。



最后，对剩下的矩阵重复整个过程，直到上下、左右的壁与壁碰在一起 (**up <= down && left <= right**, 这是避免碰壁的条件)。

### 03、Go语言示例

所以这道题很简单，只要会碰壁，就可以顺利得到代码（很漂亮，不是吗？），代码如下：

```
1 func spiralOrder(matrix [][]int) []int {
2     var result []int
3     if len(matrix) == 0 {
4         return result
5     }
6     left, right, up, down := 0, len(matrix[0])-1, 0, len(matrix)-1
7
8     var x, y int
9     for left <= right && up <= down {
10        for y = left; y <= right && avoid(left, right, up, down); y++ {
11            result = append(result, matrix[x][y])
12        }
13        y--
14        up++
15        for x = up; x <= down && avoid(left, right, up, down); x++ {
16            result = append(result, matrix[x][y])
17        }
18        x--
```

```
19     right--  
20     for y = right; y >= left && avoid(left, right, up, down); y-- {  
21         result = append(result, matrix[x][y])  
22     }  
23     y++  
24     down--  
25     for x = down; x >= up && avoid(left, right, up, down); x-- {  
26         result = append(result, matrix[x][y])  
27     }  
28     x++  
29     left++  
30 }  
31 return result  
32 }  
33  
34 func avoid(left, right, up, down int) bool {  
35     return up <= down && left <= right  
36 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2 MB，在所有 Go 提交中击败了 98.29% 的用户

## 只有两个键的键盘（650）

今天为大家分享一道关于“复制” + “粘贴”的题目。话不多说，直接看题吧。

### 01、题目分析

#### 第650题：只有两个键的键盘

最初在一个记事本上只有一个字符 'A'。你每次可以对这个记事本进行两种操作：Copy All (复制全部)：你可以复制这个记事本中的所有字符(部分的复制是不允许的)。Paste (粘贴)：你可以粘贴你上一次复制的字符。

给定一个数字 n。你需要使用最少的操作次数，在记事本中打印出恰好 n 个 'A'。输出能够打印出 n 个 'A' 的最少操作次数。

示例 1：

```
1 | 输入: 3
2 | 输出: 3
3 | 解释:
4 | 最初, 我们只有一个字符 'A'。
5 | 第 1 步, 我们使用 Copy All 操作。
6 | 第 2 步, 我们使用 Paste 操作来获得 'AA'。
7 | 第 3 步, 我们使用 Paste 操作来获得 'AAA'。
```

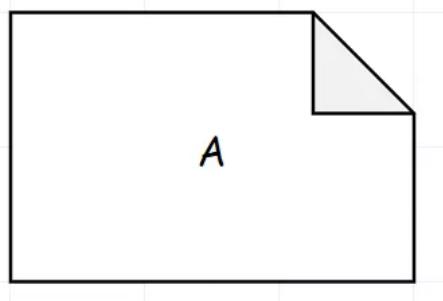
#### 说明:

n 的取值范围是 [1, 1000]。

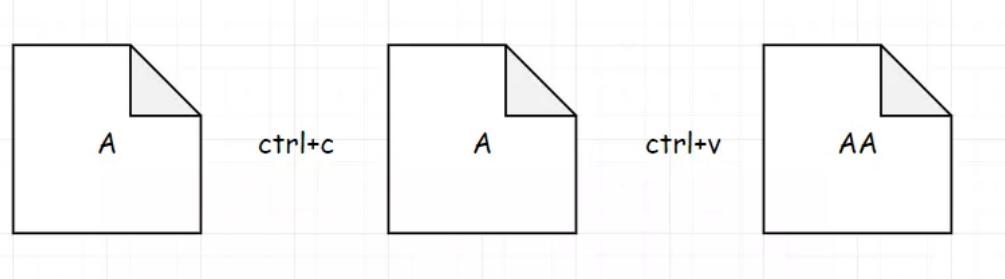
## 02、题目分析

本题的思路, 在于想明白复制和粘贴过程中的规律, 找到如何组成N个A的最小操作数。

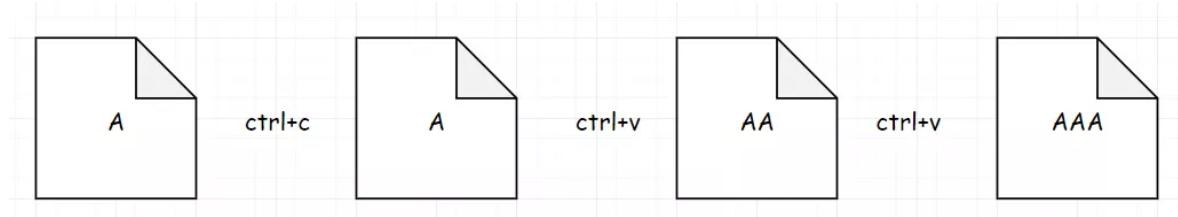
我们从最简单的开始分析, 假如我们给定数字为1, 那啥也不用做, 因为面板上本来就有一个A。



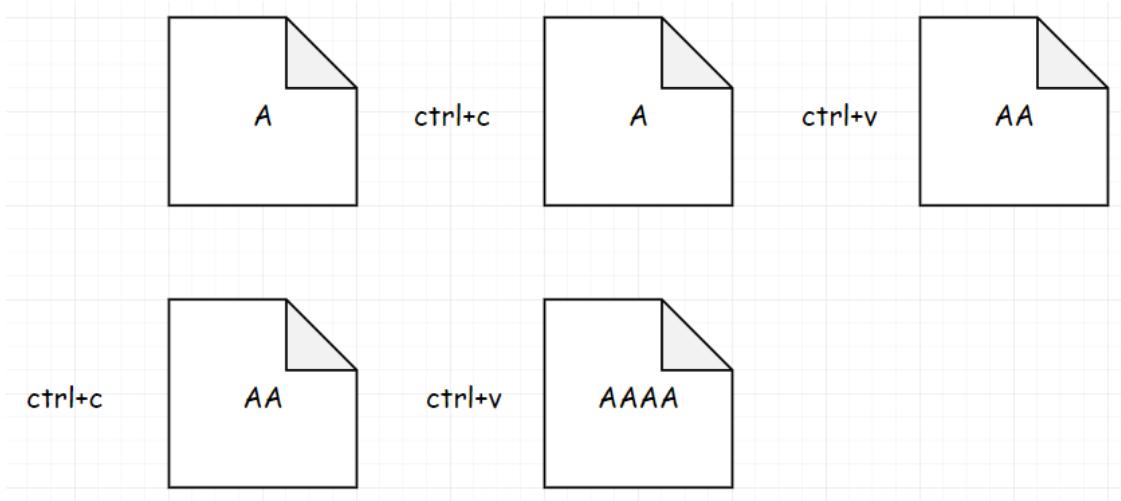
假如我们给定数字为2, 那我们需要做C-P, 共计2次操作来得到。



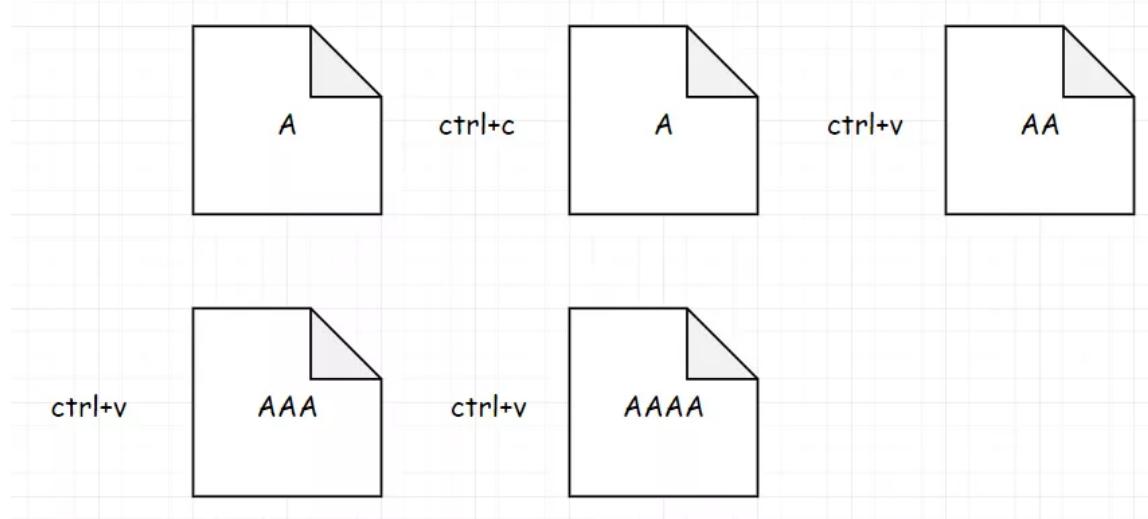
假如我们给定数字为3, 那我们需要做C-P-P, 共计3次操作来得到。



假如我们给定数字为4, 我们发现好像变得不一样了。因为我们有两种方法都可以得到目标。 (C-P-C-P)



或者 (C-P-P-P)



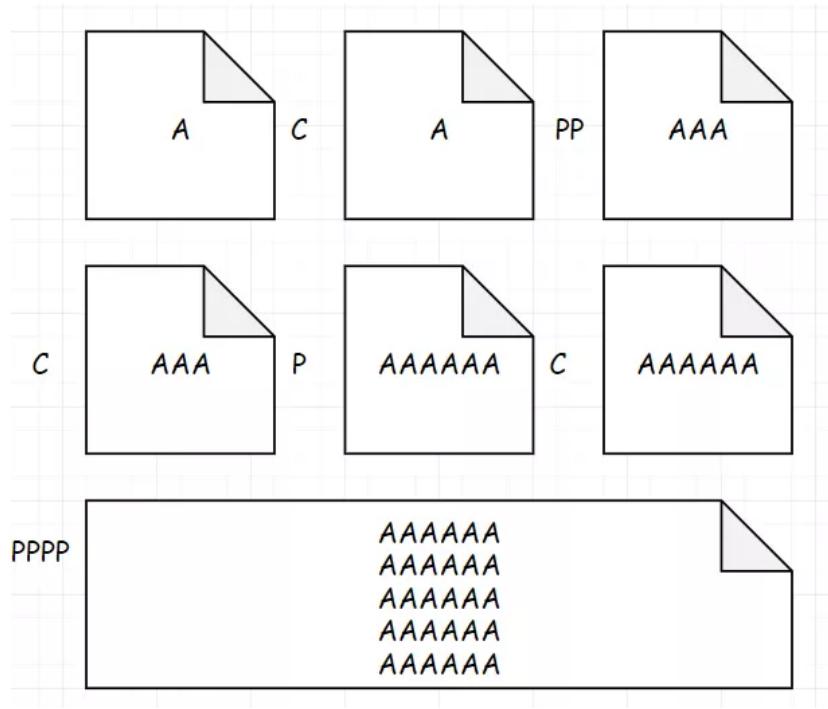
但是需要的步骤还是一样。

好了，到这里为止，STOP！通过上面的分析，我们至少可以观察出：**如果 i 为质数，那么 i 是多少，就需要粘贴多少次**。即：素数次数为本身的结论。如 两个A = 2， 三个A = 3， 五个A = 5。

那对于合数又该如何分析呢？（自然数中除能被1和本身整除外,还能被其他的数整除的数）这里我们直接给出答案：合数的次数为**将其分解质因数的操作次数的和**。解释一下，这是个啥意思？举个例子：

比如30，可以分解为：325。什么意思呢？我们演示一遍：首先复制1，进行2次粘贴得到3。然后复制3，进行1次粘贴得到6。然后复制6，进行4次粘贴得到30。总共需要 (CPPCPC

####P)



注意：这里由于每一次都需要进行一次复制，所以直接就等于分解质因数的操作次数的和。并且分解的顺序，不会影响到结果。

综合上面的分析，我们得出分析结果：

- 1、质数次数为其本身。
- 2、合数次数为将其分解到所有不能再分解的质数的操作次数的和。

### 03、Go语言示例

分析完毕，代码如下所示：

```

1 func minsteps(n int) int {
2     res := 0
3     for i := 2; i <= n; i++ {
4         for n%i == 0 {
5             res += i
6             n /= i
7         }
8     }
9     return res
10 }
```

执行结果：

执行用时：**0 ms**，在所有 Go 提交中击败了 **100.00%** 的用户

内存消耗：**1.9 MB**，在所有 Go 提交中击败了 **100.00%** 的用户

# 24点游戏 (679)

“24点”是一种数学游戏，正如象棋、围棋一样是一种人们喜闻乐见的娱乐活动。它始于何年何月已无从考究，但它以自己独具的数学魅力和丰富的内涵正逐渐被越来越多的人们所接受。今天就为大家分享一道关于“24点”的算法题目。

话不多说，直接看题。

## 01、题目分析

### 第679题：24点游戏

你有 4 张写有 1 到 9 数字的牌。你需要判断是否能通过  $*$ ,  $/$ ,  $+$ ,  $-$ ,  $(\ )$  的运算得到 24。

#### 示例 1:

- ```
1 | 输入: [4, 1, 8, 7]
2 | 输出: True
3 | 解释: (8-4) * (7-1) = 24
```

#### 示例 2:

- ```
1 | 输入: [1, 2, 1, 2]
2 | 输出: False
```

注意:

- 1、除法运算符 / 表示实数除法，而不是整数除法。例如  $4 / (1 - 2/3) = 12$ 。
- 2、每个运算符对两个数进行运算。特别是我们不能用 - 作为一元运算符。例如， $[1, 1, 1, 1]$  作为输入时，表达式  $-1 - 1 - 1 - 1$  是不允许的。
- 3、你不能将数字连接在一起。例如，输入为  $[1, 2, 1, 2]$  时，不能写成  $12 + 12$ 。

## 02、题目分析

拿到题目，第一反应就可以想到暴力求解。如果我们要判断给出的4张牌是否可以通过组合得到24，那我们只需找出所有的可组合的方式进行遍历。

4个数字，3个操作符，外加括号，基本目测就能想到组合数不会大到超出边界。所以，我们只要把他们统统列出来，不就可以进行求解了吗？说干就干！

我们首先定义个方法，用来判断两个数的所有操作符组合是否可以得到24。

```

1 func judgePoint24_2(a, b float64) bool {
2     return a+b == 24 || a*b == 24 || a-b == 24 || b-
3     a == 24 || a/b == 24 || b/a == 24

```

但是这个方法写的正确吗？其实不对！因为在计算机中，实数在计算和存储过程中会有一些微小的误差，对于一些与零作比较的语句来说，有时会因误差而导致原本是等于零但结果却小于或大于零之类的情况发生，所以常用一个很小的数 **1e-6** 代替 0，进行判读！

(**1e-6**: 表示1乘以10的负6次方。Math.abs(x)<1e-6 其实相当于x==0。**1e-6**(也就是0.000001)叫做 **epsilon**，用来抵消浮点运算中因为误差造成的相等无法判断的情况。这个知识点需要掌握！)

举个例子：

```

1 func main() {
2     var a float64
3     var b float64
4     b = 2.0
5     //math.Sqrt: 开平方根
6     c := math.Sqrt(2)
7     a = b - c*c
8     fmt.Println(a == 0)           //false
9     fmt.Println(a < 1e-6 && a > -(1e-6)) //true
10 }

```

这里直接用 **a==0** 就会得到**false**，但是通过 **a < 1e-6 && a > -(1e-6)** 却可以进行准确的判断。

所以我们将上面的方法改写：

```

1 //go语言
2 //judgePoint24_2: 判断两个数的所有操作符组合是否可以得到24
3 func judgePoint24_2(a, b float64) bool {
4     return (a+b < 24+1e-6 && a+b > 24-1e-6) ||
5            (a*b < 24+1e-6 && a*b > 24-1e-6) ||
6            (a-b < 24+1e-6 && a-b > 24-1e-6) ||
7            (b-a < 24+1e-6 && b-a > 24-1e-6) ||
8            (a/b < 24+1e-6 && a/b > 24-1e-6) ||
9            (b/a < 24+1e-6 && b/a > 24-1e-6)
}

```

完善了通过两个数来判断是否可以得到24的方法，现在我们加一个判断三个数是否可以得到24的方法。

```

1 //硬核代码，不服来辩！
2 func judgePoint24_3(a, b, c float64) bool { 3
3     return judgePoint24_2(a+b, c) ||
4            judgePoint24_2(a-b, c) ||
5            judgePoint24_2(a*b, c) ||
6            judgePoint24_2(a/b, c) ||
7            judgePoint24_2(b-a, c) ||
8            judgePoint24_2(b/a, c) ||
9            judgePoint24_2(a+c, b) ||

```

```
10     judgePoint24_2(a-c, b) ||  
11     judgePoint24_2(a*c, b) ||  
12     judgePoint24_2(a/c, b) ||  
13     judgePoint24_2(c-a, b) ||  
14     judgePoint24_2(c/a, b) ||  
15     judgePoint24_2(c+b, a) ||  
16     judgePoint24_2(c-b, a) ||  
17     judgePoint24_2(c*b, a) ||  
18     judgePoint24_2(c/b, a) ||  
19     judgePoint24_2(b-c, a) ||  
20     judgePoint24_2(b/c, a)  
21 }
```

好了。三个数的也出来了，我们再加一个判断4个数为24点的方法：（排列组合，我想大家都会....）

```
1 //硬核代码，不服来辩！
2 func judgePoint24(nums []int) bool {
3     return judgePoint24_3(float64(nums[0])+float64(nums[1]),
4         float64(nums[2]), float64(nums[3])) ||
5             judgePoint24_3(float64(nums[0])-float64(nums[1]), float64(nums[2]),
6         float64(nums[3])) ||
7             judgePoint24_3(float64(nums[0])*float64(nums[1]), float64(nums[2]),
8         float64(nums[3])) ||
9             judgePoint24_3(float64(nums[0])/float64(nums[1]), float64(nums[2]),
10        float64(nums[3])) ||
11        judgePoint24_3(float64(nums[1])-float64(nums[0]), float64(nums[2]),
12        float64(nums[3])) ||
13        judgePoint24_3(float64(nums[1])/float64(nums[0]), float64(nums[2]),
14        float64(nums[3])) ||
15        judgePoint24_3(float64(nums[0])+float64(nums[2]), float64(nums[1]),
16        float64(nums[3])) ||
17        judgePoint24_3(float64(nums[0])-float64(nums[2]), float64(nums[1]),
18        float64(nums[3])) ||
19        judgePoint24_3(float64(nums[0])*float64(nums[2]), float64(nums[1]),
20        float64(nums[3])) ||
21        judgePoint24_3(float64(nums[0])/float64(nums[2]), float64(nums[1]),
22        float64(nums[3])) ||
23        judgePoint24_3(float64(nums[2])-float64(nums[0]), float64(nums[1]),
24        float64(nums[3])) ||
25        judgePoint24_3(float64(nums[2])/float64(nums[0]), float64(nums[1]),
26        float64(nums[3])) ||
27        judgePoint24_3(float64(nums[3])-float64(nums[0]), float64(nums[1]),
28        float64(nums[2])) ||
29        judgePoint24_3(float64(nums[3])/float64(nums[0]), float64(nums[1]),
30        float64(nums[2])) ||
```

```

24     judgePoint24_3(float64(nums[2])+float64(nums[3]), float64(nums[0]),
25     float64(nums[1])) ||
26     judgePoint24_3(float64(nums[2])-float64(nums[3]), float64(nums[0]),
27     float64(nums[1])) ||
28     judgePoint24_3(float64(nums[2])*float64(nums[3]), float64(nums[0]),
29     float64(nums[1])) ||
30     judgePoint24_3(float64(nums[2])/float64(nums[3]), float64(nums[0]),
31     float64(nums[1])) ||
32     judgePoint24_3(float64(nums[3])-float64(nums[2]), float64(nums[0]),
33     float64(nums[1])) ||
34     judgePoint24_3(float64(nums[3])/float64(nums[2]), float64(nums[0]),
35     float64(nums[1])) ||
36     judgePoint24_3(float64(nums[1])+float64(nums[2]), float64(nums[0]),
37     float64(nums[3])) ||
38     judgePoint24_3(float64(nums[1])-float64(nums[2]), float64(nums[0]),
39     float64(nums[3])) ||
40     judgePoint24_3(float64(nums[1])*float64(nums[2]), float64(nums[0]),
41     float64(nums[3])) ||
42     judgePoint24_3(float64(nums[1])/float64(nums[2]), float64(nums[0]),
43     float64(nums[3])) ||
44 }

```

## 03、Go语言示例

我们整合全部代码如下：

```

1 func judgePoint24(nums []int) bool {
2     return judgePoint24_3(float64(nums[0])+float64(nums[1]),
3     float64(nums[2]), float64(nums[3])) ||
4     judgePoint24_3(float64(nums[0])-float64(nums[1]), float64(nums[2]),
5     float64(nums[3])) ||
6     judgePoint24_3(float64(nums[0])*float64(nums[1]), float64(nums[2]),
7     float64(nums[3])) ||
8     judgePoint24_3(float64(nums[0])/float64(nums[1]), float64(nums[2]),
9     float64(nums[3])) ||
10    judgePoint24_3(float64(nums[1])-float64(nums[0]), float64(nums[2]),
11    float64(nums[3])) ||
12 }

```

```

7     judgePoint24_3(float64(nums[1])/float64(nums[0]), float64(nums[2]),
8     float64(nums[3])) ||
9
10    judgePoint24_3(float64(nums[0])+float64(nums[2]), float64(nums[1]),
11    float64(nums[3])) ||
12    judgePoint24_3(float64(nums[0])-float64(nums[2]), float64(nums[1]),
13    float64(nums[3])) ||
14    judgePoint24_3(float64(nums[0])*float64(nums[2]), float64(nums[1]),
15    float64(nums[3])) ||
16    judgePoint24_3(float64(nums[0])/float64(nums[2]), float64(nums[1]),
17    float64(nums[3])) ||
18    judgePoint24_3(float64(nums[2])-float64(nums[0]), float64(nums[1]),
19    float64(nums[3])) ||
20    judgePoint24_3(float64(nums[2])/float64(nums[0]), float64(nums[1]),
21    float64(nums[3])) ||
22
23    judgePoint24_3(float64(nums[0])+float64(nums[3]), float64(nums[2]),
24    float64(nums[1])) ||
25    judgePoint24_3(float64(nums[0])-float64(nums[3]), float64(nums[2]),
26    float64(nums[1])) ||
27    judgePoint24_3(float64(nums[0])*float64(nums[3]), float64(nums[2]),
28    float64(nums[1])) ||
29
30    judgePoint24_3(float64(nums[1])+float64(nums[2]), float64(nums[0]),
31    float64(nums[3])) ||
32    judgePoint24_3(float64(nums[1])-float64(nums[2]), float64(nums[0]),
33    float64(nums[3])) ||
34    judgePoint24_3(float64(nums[1])*float64(nums[2]), float64(nums[0]),
35    float64(nums[3])) ||
36    judgePoint24_3(float64(nums[1])-float64(nums[2]), float64(nums[0]),
37    float64(nums[3])) ||
38
39    judgePoint24_3(float64(nums[1])+float64(nums[3]), float64(nums[2]),
40    float64(nums[0])) ||

```

```

38         judgePoint24_3(float64(nums[1])-float64(nums[3]), float64(nums[2]),
39         float64(nums[0])) ||
40         judgePoint24_3(float64(nums[1])*float64(nums[3]), float64(nums[2]),
41         float64(nums[0])) ||
42         judgePoint24_3(float64(nums[1])/float64(nums[3]), float64(nums[2]),
43         float64(nums[0])) ||
44         judgePoint24_3(float64(nums[3])-float64(nums[1]), float64(nums[2]),
45         float64(nums[0])) ||
46         judgePoint24_3(float64(nums[3])/float64(nums[1]), float64(nums[2]),
47         float64(nums[0]))
48     }
49
50 func judgePoint24_3(a, b, c float64) bool {
51     return judgePoint24_2(a+b, c) ||
52         judgePoint24_2(a-b, c) ||
53         judgePoint24_2(a*b, c) ||
54         judgePoint24_2(a/b, c) ||
55         judgePoint24_2(b-a, c) ||
56         judgePoint24_2(b/a, c) ||
57
58         judgePoint24_2(a+c, b) ||
59         judgePoint24_2(a-c, b) ||
60         judgePoint24_2(a*c, b) ||
61         judgePoint24_2(a/c, b) ||
62         judgePoint24_2(c-a, b) ||
63         judgePoint24_2(c/a, b) ||
64
65         judgePoint24_2(c+b, a) ||
66         judgePoint24_2(c-b, a) ||
67         judgePoint24_2(c*b, a) ||
68         judgePoint24_2(c/b, a) ||
69         judgePoint24_2(b-c, a) ||
70         judgePoint24_2(b/c, a)
71     }
72
73 func judgePoint24_2(a, b float64) bool {
74     return (a+b < 24+1e-6 && a+b > 24-1e-6) ||
75         (a*b < 24+1e-6 && a*b > 24-1e-6) ||
76         (a-b < 24+1e-6 && a-b > 24-1e-6) ||
77         (b-a < 24+1e-6 && b-a > 24-1e-6) ||
78         (a/b < 24+1e-6 && a/b > 24-1e-6) ||
79         (b/a < 24+1e-6 && b/a > 24-1e-6)
80     }

```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：1.9 MB，在所有 Go 提交中击败了 100.00% 的用户

由于代码过于硬核，我们直接击败100%的对手：（没想到吧！代码还可以这么写~）

## 飞机座位分配概率 (1227)

坐汽车、火车、飞机的时候，大家不知道有没有想过这样一个问题？如果自己的票弄丢了，那么自己屁股随机这么一蹲，坐到自己位置的概率有多大？今天就为大家分析一下这个问题。

## 01、题目分析

### 第1227题：飞机座位分配概率

有  $n$  位乘客即将登机，飞机正好有  $n$  个座位。第一位乘客的票丢了，他随便选了一个座位坐下。

剩下的乘客将会：

- 如果他们自己的座位还空着，就坐到自己的座位上，
- 当他们自己的座位被占用时，随机选择其他座位

第  $n$  位乘客坐在自己的座位上的概率是多少？

#### 示例 1：

- 1 输入：  $n = 1$
- 2 输出： 1.00000
- 3 解释： 第一个人只会坐在自己的位置上。

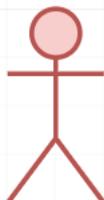
#### 示例 2：

- 1 输入：  $n = 2$
- 2 输出： 0.50000
- 3 解释： 在第一个人选好座位坐下后，第二个人坐在自己的座位上的概率是 0.5。

## 02、题目图解

对于这道题，不卖关子，直接分析：

一个位置一个人，一屁股蹲下，概率100%，这没啥可说的。



座位

两个位置两个人，第一个人已经坐下，要么坐对了，要么坐错了。所以第二个人坐在自己位置上的概率是50%。



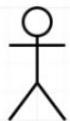
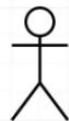
座位

座位

座位

座位

重点来了，三个位置三个人，第一个一屁股坐下，有三种坐法。



座位

座位

座位

座位

座位

座位

座位

座位

座位

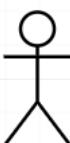
如果恰好第一个人坐到了自己的座位上（1/3），那这种情况下，第二个人也就可以直接坐在自己的座位上，第三个人一样。所以此时第三人坐在自己座位上的可能性是 100%。



座位

座位

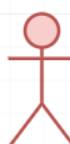
座位



座位

座位

座位



座位

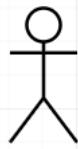
座位

座位

如果第一个人占掉了第二个人的位置（1/3）。此时第二人上来之后，要么坐在第一人的位置上，要么坐在第三人的位置上。（1/2）所以，在这种情况下，第三人的座位被占的可能性是  $1/3 \times 1/2 = 1/6$ 。



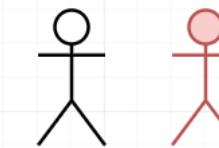
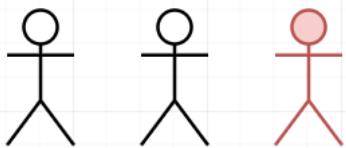
座位	座位	座位
----	----	----



座位	座位	座位
----	----	----

座位	座位	座位
----	----	----

那假如第一人直接一屁股坐在第三人的座位上，此时第三人的座位被占的可能性就是第一人选择第三人座位的可能性。 (1/3)



座位	座位	座位
----	----	----

所以，如果三个座位三个人，第三个人坐到自己位置上的概率就是： $1 - \frac{1}{6} - \frac{1}{3} = \frac{1}{2}$ 。当然，也可以通过  $\frac{1}{3} + \frac{1}{6} = \frac{1}{2}$  来正向计算。

而对于  $n > 3$  的情况，我们参照 3 个座位时进行分析：

- 如果第 1 个乘客选择第 1 个座位，那么第  $n$  个人选择到第  $n$  个座位的可能性就是 100%。 $(1/n)$
- 如果第 1 个乘客选择了第  $n$  个座位，那么第  $n$  个人选择第  $n$  个座位的可能性就是 0。 $(0)$
- 而对于第 1 个乘客选择除了第一个和第  $n$  个座位之外的座位  $k$  ( $1 < k < n$ )，就会导致有可能出现，前  $n-1$  位乘客占第  $n$  位乘客的概率出现。

第一二种情况都好说，对于第三种情况。因为此时第  $k$  个座位被占用，于第  $k$  个乘客而言，他又会面临和第一个乘客一样的选择。相当于乘客 1 将问题转移到了第  $k$  个乘客身上，等同于\*\*本次选择无效！且这个过程会一直持续到没有该选项。于是乎，对于第  $n$  个人，他最后将只有两个选项：1、自己的 2、第一个人。所以对于  $n \geq 3$  的情况，等同于  $n=2$ ，全部的概率都为  $1/2^{**}$ 。

如果还是不能理解的小伙伴，可以这样想。登机时座位被占的乘客，其实相当于和上一位坐错的乘客交换了身份。直到完成终止条件（坐对位置或者坐到最后一个位置），否则该交换将一直进行下去。所以第  $n$  位乘客，坐到第  $n$  个位置，自然还是  $1/2$ 。

## 03、Go语言示例

根据分析，完成代码：

```
1 func nthPersonGetsNthSeat(n int) float64 {
2     if n == 1 {
3         return 1
4     }
5     return 0.5
6 }
```

小伙伴都看懂了吗？

这里留下一个疑问，假如共有200个座位，平均有多少人没有坐到自己的位置呢？

评论区留下你的想法吧！

## 水分子的产生

今天为大家分享一道看起来“高大上”\*\*\*的题目。

话不多说，直接看题吧。

### 01、水分子的产生

#### 水分子的产生

现在有两种线程，氢 oxygen 和氧 hydrogen，你的目标是组织这两种线程来产生水分子。

存在一个屏障（barrier）使得每个线程必须等候直到一个完整水分子能够被产生出来。

氢和氧线程会被分别给予 releaseHydrogen 和 releaseOxygen 方法来允许它们突破屏障。

这些线程应该三三成组突破屏障并能立即组合产生一个水分子。

你必须保证产生一个水分子所需线程的结合必须发生在下一个水分子产生之前。

换句话说：

如果一个氧线程到达屏障时没有氢线程到达，它必须等候直到两个氢线程到达。

如果一个氢线程到达屏障时没有其它线程到达，它必须等候直到一个氧线程和另一个氢线程到达。

书写满足这些限制条件的氢、氧线程同步代码。

### 示例 1:

```
1 | 输入: "HOH"
2 | 输出: "HHO"
3 | 解释: "HOH" 和 "OHH" 依然都是有效解。
```

### 示例 2:

```
1 | 输入: "OOHHHH"
2 | 输出: "HHOHHO"
3 | 解释: "HOHHHO", "OHHHHO", "HHOH OH", "HOHHOH", "OHHHOH", "HHOOHH", "HOHOHH" 和
    "OHHOHH" 依然都是有效解。
```

限制条件:

- 输入字符串的总长将会是  $3n$ ,  $1 \leq n \leq 50$ ;
- 输入字符串中的 "H" 总数将会是  $2n$ ;
- 输入字符串中的 "O" 总数将会是  $n$ .

代码模板:

```
1 | class H2O {
2 |     public H2O() {
3 |
4 |     }
5 |
6 |     public void hydrogen(Runnable releaseHydrogen) throws InterruptedException {
7 |         // releaseHydrogen.run() outputs "H". Do not change or remove this
8 |         // line.
9 |         releaseHydrogen.run();
10 |
11 |     }
12 |
13 |     public void oxygen(Runnable releaseOxygen) throws InterruptedException {
14 |
15 |         // releaseOxygen.run() outputs "O". Do not change or remove this line.
16 |         releaseOxygen.run();
17 |
18 |     }
19 | }
```

## 02、JAVA分析

乍看之下，题目貌似很高大上，不少同学一听多线程直接就慌了神。我们一起分析一下。一个氧消耗两个氢，两个氢供给一个氧。我们只要可以模拟氢和氧的供给关系，就可以顺利进行求解。

这里先介绍一下Java中的Semaphore: Semaphore是 synchronized 的加强版，作用是**控制线程的并发数量**。可以通过 acquire 和 release 来进行类似 lock 和 unlock 的操作。

```
1 //请求一个信号量，这时候信号量个数-1，当减少到0的时候，下一次acquire不会再执行，只有当执行
2 //一个release()的时候，信号量不为0的时候才可以继续执行acquire
3 void acquire()
4 //释放一个信号量，这时候信号量个数+1,
5 void release();
```

什么？听不懂！大白话就是叫做 Semaphore 的这个东东，可以控制同时有多少线程可以进去，比一般的锁要稍微高级那么一点点。

由于题目中给的限制条件，已经明确说明了H是 $2n$ ，O是n，所以我们不需要考虑无法构成水分子的情况。我们分别定义H和O的信号量，都初始化为2个信号量。

在每一次产生O的过程中，都需要等待产生了两个H。

```
1 import java.util.concurrent.Semaphore;
2
3 class H2O {
4     public H2O() { 6
5     }
6     private Semaphore h = new Semaphore(2);
7     private Semaphore o = new Semaphore(2);
8
9     public void hydrogen(Runnable releaseHydrogen) throws InterruptedException {
10         h.acquire(1);
11         releaseHydrogen.run();
12         o.release(1);
13     }
14
15     public void oxygen(Runnable releaseOxygen) throws InterruptedException {
16         o.acquire(2);
17         releaseOxygen.run();
18         h.release(2);
19     }
20 }
```

## 03、C++代码分析

如果没有原生的信号量支持怎么办？其实也是一样的。我们可以通过锁来模拟信号量。这里加一个C++版本的实现。

```
1 class H2O {
2 private:
3     int countOxygen;
4     pthread_mutex_t lockHy;
5     pthread_mutex_t lockOx;
6 public:
7     H2O() {
```

```

8     pthread_mutex_init(&lockOx,NULL);
9     pthread_mutex_init(&lockHy,NULL);
10    pthread_mutex_lock(&lockOx);
11    countOxygen = 2;
12 }
13 void hydrogen(function<void()> releaseHydrogen) {
14     pthread_mutex_lock(&lockHy);
15     releaseHydrogen();
16     countOxygen--;
17     if(countOxygen > 0){
18         pthread_mutex_unlock(&lockHy);
19     }else{
20         pthread_mutex_unlock(&lockOx);
21     }
22 }
23 void oxygen(function<void()> releaseOxygen) {
24     pthread_mutex_lock(&lockOx);
25     releaseOxygen();
26     countOxygen = 2;
27     pthread_mutex_unlock(&lockHy);
28 }
29 };

```

## 03、Python代码分析

好吧，其他语言都有并发。但是我PY竟然连并发都没有（杠精勿扰，我知道有 threading 库可以用。并且里边也已经提供了现成的信号量可以用）这种情况下怎么办？

还是可以解决，我们可以用队列模拟进行实现

```

1 class H2O:
2     def __init__(self):
3         self.h, self.o = [], []
4     def hydrogen(self, releaseHydrogen: 'Callable[[], None]') -> None:
5         self.h.append(releaseHydrogen) 7
6         self.res()
7     def oxygen(self, releaseOxygen: 'Callable[[], None]') -> None:
8         self.o.append(releaseOxygen)
9         self.res()
10    def res(self):
11        if len(self.h) > 1 and len(self.o) > 0:
12            self.h.pop(0)()
13            self.h.pop(0)()
14            self.o.pop(0)()

```

## 04、GO语言版本

已经提供了PY, JAVA, C++的版本。对于GO而言，不管你是通过channel来模拟信号量的方式，还是参考PY的方式进行实现，我觉得应该都可以完成。

# 救生艇 (881)

小浩算法改版了，大家看一下风格怎么样，还喜欢吗？所有的排版，绘图，文案，题解都是由小浩一人完成哦~

今天为大家分享一道关于“**救生艇**”的题目。

话不多说，直接看题吧。

## 01、题目示例

### 第881题：救生艇

第  $i$  个人的体重为  $\text{people}[i]$ ，每艘船可以承载的最大重量为  $\text{limit}$ 。每艘船最多可同时载两人，但条件是这些人的重量之和最多为  $\text{limit}$ 。返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

#### 示例 1：

```
1 | 输入: people = [1,2], limit = 3
2 | 输出: 1
3 | 解释: 1 艘船载 (1, 2)
```

#### 示例 2：

```
1 | 输入: people = [3,2,2,1], limit = 3
2 | 输出: 3
3 | 解释: 3 艘船分别载 (1, 2), (2) 和 (3)
```

#### 示例 3：

```
1 | 输入: people = [3,5,3,4], limit = 5
2 | 输出: 4
3 | 解释: 4 艘船分别载 (3), (3), (4), (5)
```

#### 提示：

- $1 \leq \text{people.length} \leq 50000$
- $1 \leq \text{people}[i] \leq \text{limit} \leq 30000$

## 02、题目分析

这不是一道算法题，这是一个脑筋急转弯。

一个船最多可以装两个人，并且不能把船压垮。同时要求把这些人可以统统装下的最小船数。用脚趾头也可以想到，我们需要**尽最大努力的去维持一个船上得有两个人**。哦，不，船上。这是什么思想？Bingo，贪心。

思路很简单：

1. 我们首先需要让这些人根据体重进行排序。
2. 同时维护两个指针，每次让最重的一名上船，同时让最轻的也上船。（因为最重的要么和最轻的一起上船。要么就无法配对，只能自己占用一艘船的资源）

## 03、JAVA示例

根据分析，得到代码：

```
1 class Solution {
2     public int numRescueBoats(int[] people, int limit) {
3         Arrays.sort(people);
4         int i = 0, j = people.length - 1;
5         int ans = 0;
6
7         while (i <= j) {
8             ans++;
9             if (people[i] + people[j] <= limit)
10                 i++;
11             j--;
12         }
13         return ans;
14     }
15 }
```

## 04、GO示例

GO代码其实也一样：

```
1 func numRescueBoats(people []int, limit int) int {
2     sort.Ints(people)
3     ans := 0
4     i, j := 0, len(people) - 1
5     for i <= j {
6         if people[i] + people[j] <= limit {
7             i++
8         } else{
9             j--
10        }
11        ans++
12    }
13    return ans
14 }
```

## 05、题目扩展

这里肯定马上就有细心的读者会问！为什么你每次是让最瘦的和最胖的来凑一对。而不是放弃掉这个最瘦的，去找一个逼近limit体重的人来乘船呢？这里要注意题目，因为题中已经告诉了，一艘船仅能坐两人。所以去找一个逼近limit体重的人是没有意义的。

但是，这里并不妨碍我们将此题扩展进行思考。这里留下疑问，如果我们不对船上的人数进行限制，那么应该如何来完成本题呢？大家可以尝试代码练习一下。

## 救生艇（881）

小浩算法改版了，大家看一下风格怎么样，还喜欢吗？所有的排版，绘图，文案，题解都是由小浩一人完成哦~

今天为大家分享一道关于“救生艇\*\*\*”的题目。

话不多说，直接看题吧。

### 01、题目标示例

#### 第881题：救生艇

第 i 个人的体重为 `people[i]`，每艘船可以承载的最大重量为 `limit`。每艘船最多可同时载两人，但条件是这些人的重量之和最多为 `limit`。返回载到每一个人所需的最小船数。(保证每个人都能被船载)。

#### 示例 1：

```
1 输入: people = [1,2], limit = 3
2 输出: 1
3 解释: 1 艘船载 (1, 2)
```

#### 示例 2：

```
1 输入: people = [3,2,2,1], limit = 3
2 输出: 3
3 解释: 3 艘船分别载 (1, 2), (2) 和 (3)
```

#### 示例 3：

```
1 输入: people = [3,5,3,4], limit = 5
2 输出: 4
3 解释: 4 艘船分别载 (3), (3), (4), (5)
```

**提示：**

- $1 \leq \text{people.length} \leq 50000$
- $1 \leq \text{people}[i] \leq \text{limit} \leq 30000$

## 02、题目分析

这不是一道算法题，这是一个脑筋急转弯。

一个船最多可以装两个人，并且不能把船压垮。同时要求把这些人可以统统装下的最小船数。用脚趾头也可以想到，我们需要**尽最大努力的去维持一个船上得有两个人**。。哦，不，船上。这是什么思想？Bingo，贪心。

思路很简单：

1. 我们首先需要让这些人**根据体重进行排序**。
2. **同时维护两个指针，每次让最重的一名上船，同时让最轻的也上船。**（因为最重的要么和最轻的一起上船。要么就无法配对，只能自己占用一艘船的资源）

## 03、JAVA示例

根据分析，得到代码：

```
1 class Solution {
2     public int numRescueBoats(int[] people, int limit) {
3         Arrays.sort(people);
4         int i = 0, j = people.length - 1;
5         int ans = 0;
6
7         while (i <= j) {
8             ans++;
9             if (people[i] + people[j] <= limit)
10                 i++;
11                 j--;
12             }
13             return ans;
14         }
15 }
```

## 04、GO示例

GO代码其实也一样：

```
1 func numRescueBoats(people []int, limit int) int {
2     sort.Ints(people)
3     ans := 0
4     i, j := 0, 0, len(people) - 1
```

```
5     for i <= j {
6         if people[i] + people[j] <= limit {
7             i++
8         } else{
9             j--
10        }
11        ans++
12    }
13    return ans
14 }
```

## 05、题目扩展

这里肯定马上就有细心的读者会问！为什么你每次是让最瘦的和最胖的来凑一对。而不是放弃掉这个最瘦的，去找一个逼近limit体重的人来乘船呢？这里要注意题目，**因为题中已经告诉了，一艘船仅能坐两人**。所以去找一个逼近limit体重的人是没有意义的。

但是，这里并不妨碍我们将此题扩展进行思考。这里留下疑问，**如果我们不对船上的人数进行限制，那么应该如何来完成本题呢？**大家可以尝试代码练习一下。

## 灯泡开关（319）

今天为大家分享一道关于“电灯泡”的题目。

话不多说，直接看题。

### 01、题目标示例

#### 第319题：开关灯泡

初始时有  $n$  个灯泡关闭。第 1 轮，你打开所有的灯泡。第 2 轮，每两个灯泡关闭一次。第 3 轮，每三个灯泡切换一次开关（如果关闭则开启，如果开启则关闭）。第  $i$  轮，每  $i$  个灯泡切换一次开关。对于第  $n$  轮，你只切换最后一个灯泡的开关。找出  $n$  轮后有多少个亮着的灯泡。

**示例：**

```
1 输入： 3
2 输出： 1
3 解释：
4 初始时，灯泡状态 [关闭, 关闭, 关闭].
5 第一轮后，灯泡状态 [开启, 开启, 开启].
6 第二轮后，灯泡状态 [开启, 关闭, 开启].
7 第三轮后，灯泡状态 [开启, 关闭, 关闭].
8
9 你应该返回 1，因为只有一个灯泡还亮着。
```

## 02、题目图解

这是一道难度评定为**困难**的题目。但是，其实这并不是一道算法题，而是一个脑筋急转弯。只要我们模拟一下开关灯泡的过程，大家就会瞬间get，一起来分析一下：

我们模拟一下n从1到12的过程。在第一轮，你打开了12个灯泡：

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

因为对于大于n的灯泡你是不care的，所以我们用黑框框表示：

第1轮	1	2	3	4	5	6	7	8	9	10	11	12
-----	---	---	---	---	---	---	---	---	---	----	----	----

然后我们列出n从1-12的过程中所有的灯泡示意图：

第1轮	1	2	3	4	5	6	7	8	9	10	11	12
第2轮	1	2	3	4	5	6	7	8	9	10	11	12
第3轮	1	2	3	4	5	6	7	8	9	10	11	12
第4轮	1	2	3	4	5	6	7	8	9	10	11	12
第5轮	1	2	3	4	5	6	7	8	9	10	11	12
第6轮	1	2	3	4	5	6	7	8	9	10	11	12
第7轮	1	2	3	4	5	6	7	8	9	10	11	12
第8轮	1	2	3	4	5	6	7	8	9	10	11	12
第9轮	1	2	3	4	5	6	7	8	9	10	11	12
第10轮	1	2	3	4	5	6	7	8	9	10	11	12
第11轮	1	2	3	4	5	6	7	8	9	10	11	12
第12轮	1	2	3	4	5	6	7	8	9	10	11	12

可以得到如下表格：

灯泡数 N	N 轮后亮着的灯泡
1	1
2	1
3	1
4	2
5	2
6	2
7	2
8	2
9	3
10	3
11	3
12	3

观察一下，这是什么？观察不出来，咱们看看这个：

```

1 //go
2 func main() {
3     for n := 1; n <= 12; n++ {
4         fmt.Println("n=", n, "灯泡数\t", math.Sqrt(float64(n)))
5     }
6 }
```

```

1 //print
2 n= 1 灯泡数 1
3 n= 2 灯泡数 1.4142135623730951
4 n= 3 灯泡数 1.7320508075688772
5 n= 4 灯泡数 2
6 n= 5 灯泡数 2.23606797749979
7 n= 6 灯泡数 2.449489742783178
8 n= 7 灯泡数 2.6457513110645907
9 n= 8 灯泡数 2.8284271247461903
10 n= 9 灯泡数 3
11 n= 10 灯泡数 3.1622776601683795
12 n= 11 灯泡数 3.3166247903554
13 n= 12 灯泡数 3.4641016151377544
```

没错，只要我们对n进行开方，就可以得到最终的灯泡数。根据分析，得出代码：

```

1 //给一个c++版本的
2 class Solution {
3 public:
4     int bulbSwitch(int n) {
5         return sqrt(n);
6     }
7 };
```

执行结果：

执行用时：0 ms，在所有 C++ 提交中击败了 100.00% 的用户

内存消耗：8.2 MB，在所有 C++ 提交中击败了 73.08% 的用户

## 03、证明

我不服，没有证明，你说毛线！证明如下：

约数，又称因数。整数a除以整数b( $b \neq 0$ )除得的商正好是整数而没有余数，我们就说a能被b整除，或b能整除a。a称为b的倍数，b称为a的约数。

从我们观察可以发现，如果一个灯泡有奇数个约数，那么最后这个灯泡一定会亮着。

什么，你问我奇数是什么？奇数（odd）指不能被2整除的整数，数学表达形式为： $2k+1$ ，奇数可以分为正奇数和负奇数。

所以其实我们是求，**从1-n有多少个数的约数有奇数个。而有奇数个约数的数一定是完全平方数。**这是因为，对于数n，如果m是它的约数，则 $n/m$ 也是它的约数，若 $m \neq n/m$ ，则它的约数是以m、 $n/m$ 的形式成对出现的。而 $m = n/m$ 成立且 $n/m$ 是正整数时，n是完全平方数而它有奇数个约数。

我们再次转化问题，**求1-n有多少个数是完全平方数。**

什么，你又不知道什么是完全平方数了？完全平方指用一个整数乘以自己例如11，22， $3^2$ 等，依此类推。若一个数能表示成某个整数的平方的形式，则称这个数为完全平方数。

到这里，基本就很明朗了。剩下的，我想不需要再说了吧！

## 三门问题

三门问题（Monty Hall problem）亦称为蒙提霍尔问题、蒙特霍问题或蒙提霍尔悖论，出自美国的电视游戏节目Let's Make a Deal。今天为大家进行完整分析。

话不多说，直接看题目。

## 01、题目标示例

### 三门问题

参赛者的面前有三扇关闭着的门，其中一扇的后面是天使，选中后天使会达成你的一个愿望，而另外两扇门后面则是恶魔，选中就会死亡。

当你选定了一扇门，但未去开启它的时候，上帝会开启剩下两扇门中的一扇，露出其中一只恶魔。（上帝是全能的，必会打开恶魔门）随后上帝会问你要不要更换选择，选另一扇仍然关着的门。

## 02、题目分析

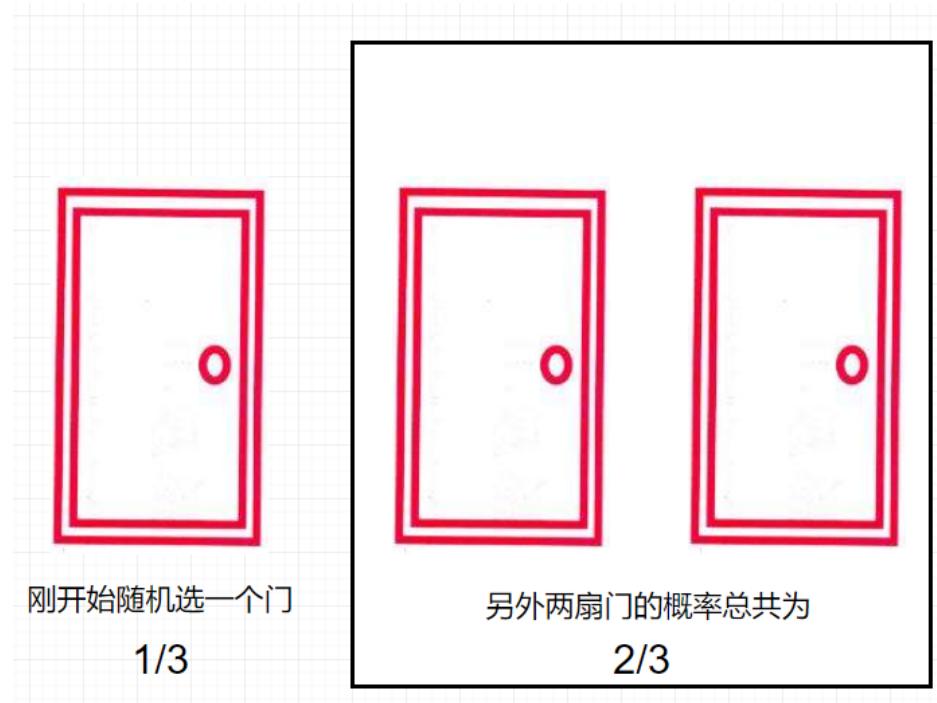
按照常理，参赛者在做出最开始的决定时，对三扇门后面的事情一无所知，因此他选择正确的概率是 $1/3$ ，这个应该大家都想到。

接下来，主持人排除掉了一个错误答案（有恶魔的门），于是剩下的两扇门必然是一扇是天使，一扇是恶魔，那么此时无论选择哪一扇门，胜率都是 $1/2$ ，依然合乎直觉。

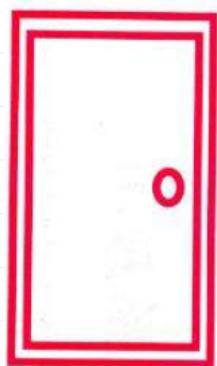
所以你作为参赛者，你会认为换不换都无必要，获胜概率均为 $1/2$ 。但是，真的是这样吗？

正确的答案是，**如果你选择了换，碰见天使的概率会高达 $2/3$ ，而不换的话，碰见天使的概率只有 $1/3$ 。**怎么来的？

我们用一个很通俗的方法，能让你一听就懂。首先刚开始选择的一扇门的概率为 $1/3$ ，而另外两扇门的总概率为 $2/3$ 。



现在上帝打开了其中一扇为恶魔的门，我们知道这个门后面不会再有天使，所以相当于这部分概率被第三个门持有。



刚开始随机选一个门

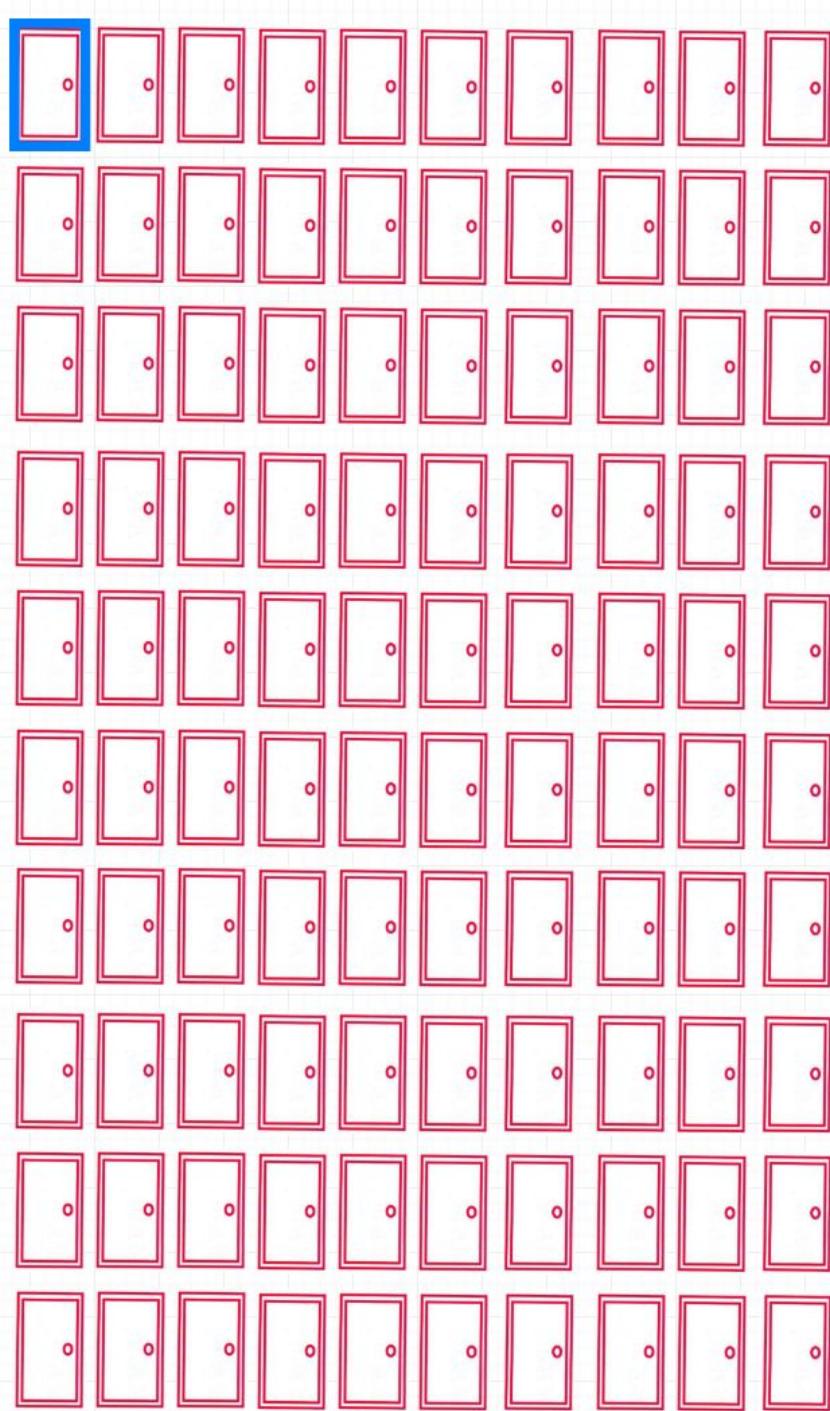
1/3



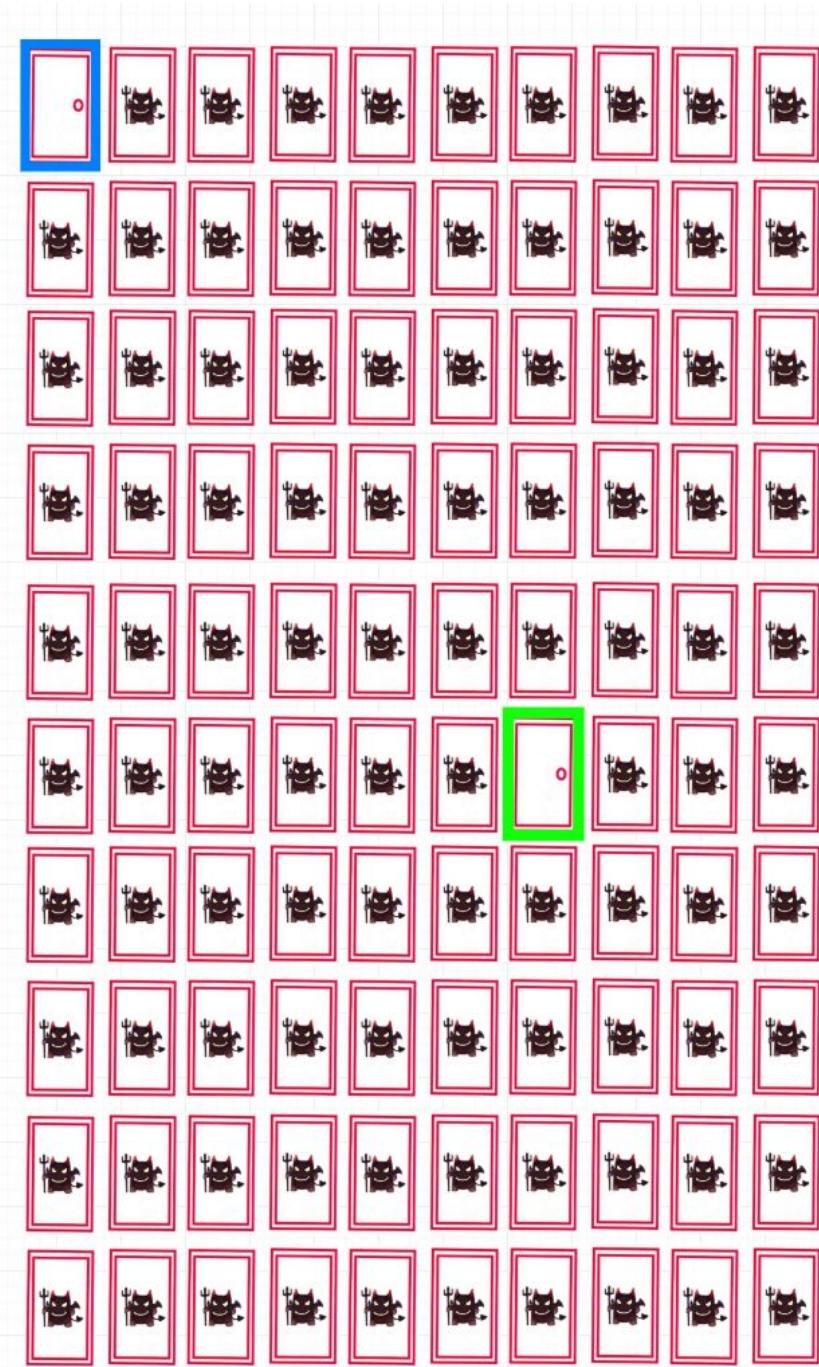
这部分概率  
将被该门持有  
2/3

剩下的那扇门的概率 (2/3) 相当于刚开始选择的门 (1/3) 的二倍。所以我们得换。

如果还没有听懂。我们可以假设有一百扇门，里边有99只都是恶魔。现在你随机选择一扇门，选择到天使的概率是1/100。



这时，上帝打开其中的98扇，里边都是恶魔。这时候就相当于99/100的概率都集中在了另一扇门里。自然，我们需要选择换。



## 04、代码证明

为了验证结果，我用代码跑了一百万次。什么？用贝叶斯分析分析！太俗（请隔壁去找李永乐老师），咱们还是直接上代码。

```
1 func main() {
2     //换门遇见天使的次数和不换门遇见天使的次数
3     changeAngelCount, unchangeAngelCount := 0, 0
4     for i := 0; i < 1000000; i++ {
5         //门的总数
6         doors := []int{0, 1, 2}
7         //天使门和选中的门
8         angelDoor, selectedDoor := rand.Intn(3), rand.Intn(3)
9         //上帝移除一扇恶魔门
10        for j := 0; j < len(doors); j++ {
11            if doors[j] != selectedDoor && doors[j] != angelDoor {
```

```

9         doors = append(doors[:j], doors[j+1:]...)
10        break
11    }
12  }
13 //统计
14 if selectedDoor == angelDoor {
15     unchangeAngelCount++
16 } else {
17     changeAngelCount++
18 }
19 }
20 fmt.Println("不换门遇见天使次数:", unchangeAngelCount, "比
例: ", (float32(unchangeAngelCount) / 1000000))
21 fmt.Println("换门遇见天使次数:", changeAngelCount, "比
例: ", (float32(changeAngelCount) / 1000000))
22 }

```

跑了一百万次，结果当然不让我们失望！执行结果为：

不换门遇见天使次数: 333535	比例: 0.333535
换门遇见天使次数: 666465	比例: 0.666465

所以，今天的问题你听明白了吗？评论区留下你的想法吧！

## 猜数字游戏（299）

今天为大家分享一道非常经典的题目，**猜数字**。话不多说，直接看题。

### 01、题目分析

#### 第299题：猜数字（Bulls and Cows）游戏

你写下一个数字让你的朋友猜。每次他猜测后，你给他一个提示，告诉他有多少位数字和确切位置都猜对了（称为“Bulls”，公牛），有多少位数字猜对了但是位置不对（称为“Cows”，奶牛）。你的朋友将会根据提示继续猜，直到猜出秘密数字。

请写出一个根据秘密数字和朋友的猜测数返回提示的函数，用 A 表示公牛，用 B 表示奶牛。

请注意秘密数字和朋友的猜测数都可能含有重复数字。

#### 示例 1：

```

1 输入: secret = "1807", guess = "7810"
2 输出: "1A3B"
3 解释: 1 公牛和 3 奶牛。公牛是 8，奶牛是 0, 1 和 7。

```

#### 示例 2：

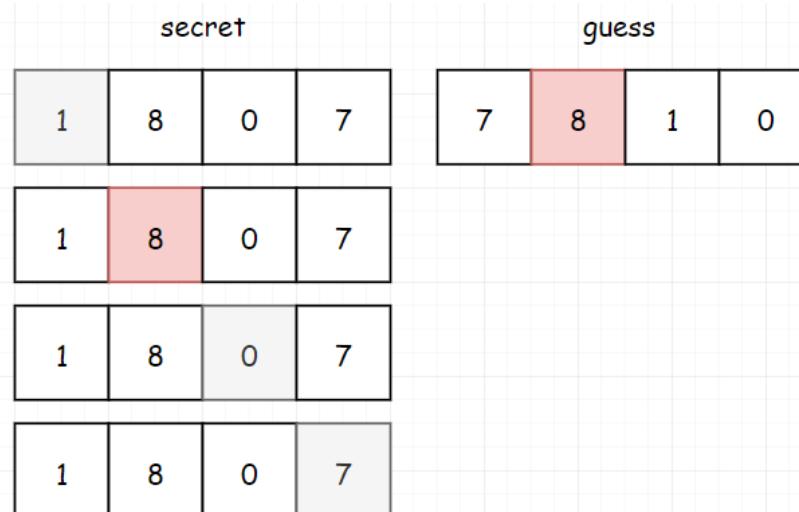
```
1 | 输入: secret = "1123", guess = "0111"
2 | 输出: "1A1B"
3 | 解释: 朋友猜测数中的第一个 1 是公牛, 第二个或第三个 1 可被视为奶牛。
```

**说明:** 你可以假设秘密数字和朋友的猜测数都只包含数字, 并且它们的长度永远相等。

## 02、题目图解

这道题, 虽然被评定为“简单”, 但是其实非常有趣。基本拿到题目, 我们就能想到可以使用hashmap进行求解, 一起来分析一下。

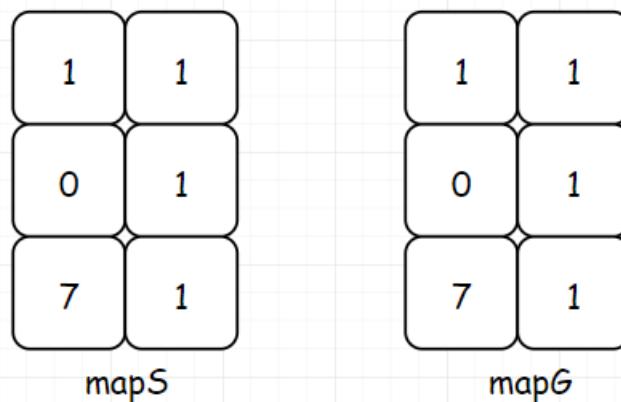
- 因为secret数字和guess数字长度相等, 所以我们遍历secret数字。
- 如果当前索引两个数字相同, 就将公牛数加1。



- **如果不相同, 我们将secret和guess当前索引位置处的数字通过map记录下来, 统计他们出现的次数。**当然, 之前我们讲过。有限的map, 比如数字 0-10, 字母 a-z, 都可以通过**数组**来进行替换, 用以压缩空间。

`mapS`: 记录`secret`中不相等数字的出现次数

`mapG`: 记录`guess`中不相等数字出现的次数



- 最后, 如果记录的两个map中, **数字出现重叠** (可以通过最小值来判断), 则意味着该数字在两边都出现过, 就将母牛数加一 (我就想说是母牛, 不服来辩)

## 03、GO语言示例

根据分析，完成代码（这次翻Go的牌子）：

```
1 func getHint(secret string, guess string) string {
2     a, b := 0, 0
3     mapS, mapG := make([]int, 10), make([]int, 10)
4     for i := range secret {
5         //注意：这里是获取对应数字的ASCII码
6         tmp, charGuess := secret[i], guess[i]
7         if tmp == guess[i] {
8             a++
9         } else {
10            mapS[tmp-'0']++
11            mapG[charGuess-'0']++
12        }
13    }
14    for i := 0; i < 10; i++ {
15        //找到重叠的
16        b += min(mapS[i], mapG[i])
17    }
18    //strconv.Itoa : 整数转字符串
19    return strconv.Itoa(a) + "A" + strconv.Itoa(b) + "B"
20 }
21
22 func min(a, b int) int {
23     if a > b {
24         return b
25     }
26     return a
27 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2.2 MB，在所有 Go 提交中击败了 100.00% 的用户

## 04、奇怪的知识

**奶牛包不包括公牛？**为了研究这个问题，我google了好一会儿。首先，国际定义，**奶牛包括公牛**。那公奶牛能不能产奶呢？答案是**不能**。那现在就有意思了，为什么公牛不产奶，还可以被称为奶牛？这是因为公奶牛是用来交配的，他们要保证所有的母奶牛都在哺乳期，所以他们需要不停的交配。一般一个养殖场，公母的比例大约是8：100。母牛当然舒服了，挤挤奶就成。但是这些公牛，却是相当辛苦。正所谓，“吃水不忘挖井人”，如此含辛茹苦的公牛，凭什么就不能被称为奶牛呢？

所以，今天的问题你听明白了吗？评论区留下你的想法吧！

# LRU缓存机制 (146)

今天为大家分享很出名的 LRU 算法，第一讲共包括 4 节。

- LRU概述
- LRU使用
- LRU实现
- Redis近LRU概述

## 01、LRU 概述

LRU 是 Least Recently Used 的缩写，译为最近最少使用。它的理论基础为“**最近使用的数据会在未来一段时期内仍然被使用，已经很久没有使用的数据大概率在未来很长一段时间仍然不会被使用**”由于该思想非常契合业务场景，并且可以解决很多实际开发中的问题，所以我们经常通过 LRU 的思想来作缓存，一般也将其称为**LRU缓存机制**。因为恰好 leetcode 上有这道题，所以我干脆把题目贴这里。但是对于 LRU 而言，希望大家不要局限于本题（大家不用担心学不会，我希望能做一个全网最简单的版本，希望可以坚持看下去！）下面，我们一起学习一下。

### 第146题：LRU缓存机制

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作：获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1 。

写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最近最少使用的数据值，从而为新的数据值留出空间。

进阶:你是否可以在 O(1) 时间复杂度内完成这两种操作？

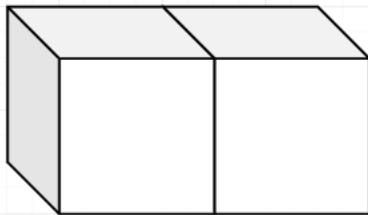
### 示例:

```
1 LRUcache cache = new LRUcache( 2 /* 缓存容量 */ );
2 cache.put(1, 1);
3 cache.put(2, 2);
4 cache.get(1);      // 返回 1
5 cache.put(3, 3);      // 该操作会使得密钥 2 作废
6 cache.get(2);      // 返回 -1 (未找到)
7 cache.put(4, 4);      // 该操作会使得密钥 1 作废
8 cache.get(1);      // 返回 -1 (未找到)
9 cache.get(3);      // 返回 3
10 cache.get(4);     // 返回 4
```

## 02、LRU 使用 (解释)

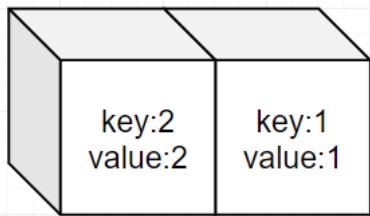
由于我实在担心部分同学完全懵逼零基础，所以我把上面的LRUCache的示例解释一下。

第一步：我们申明一个 LRUCache，长度为 2。



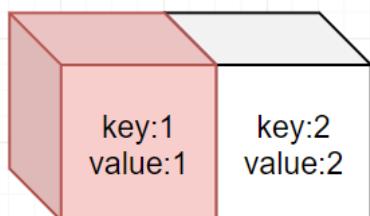
```
LRUCache cache = new LRUCache()
```

第二步：我们分别向 cache 里边 put(1,1) 和 put(2,2)，这里因为最近使用的是 2 (put 也算作使用) 所以2在前，1 在后。



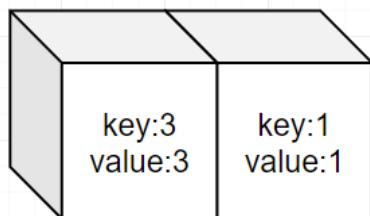
```
LRUCache cache = new LRUCache()  
cache.put(1,1)  
cache.put(2,2)
```

第三步：我们 get(1)，也就是我们使用了 1，所以需要将 1 移到前面。



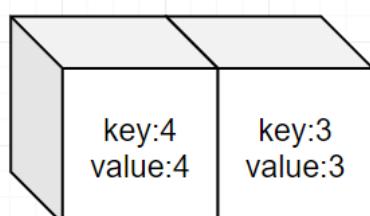
```
LRUCache cache = new LRUCache()  
cache.put(1,1)  
cache.put(2,2)  
cache.get(1)
```

第四步：此时我们 put(3,3)，因为 2 是最近最少使用的，所以我们需要将 2 进行作废。此时我们再 get(2)，就会返回 -1。



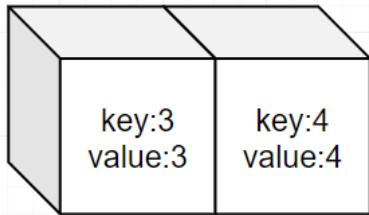
```
LRUCache cache = new LRUCache()  
cache.put(1,1)  
cache.put(2,2)  
cache.get(1)  
cache.put(3,3)  
cache.get(2)
```

第五步：我们继续 put(4,4)，同理我们将 1 作废。此时如果 get(1)，也是返回 -1。



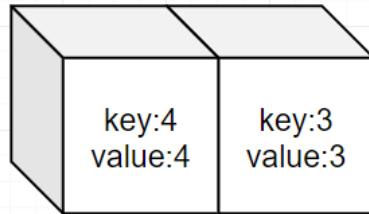
```
LRUCache cache = new LRUCache()  
cache.put(1,1)  
cache.put(2,2)  
cache.get(1)  
cache.put(3,3)  
cache.get(2)  
cache.put(4,4)  
cache.get(1)
```

第六步：此时我们 get(3)，实际为调整 3 的位置。



```
LRUCache cache = new LRUCache()
cache.put(1,1)
cache.put(2,2)
cache.get(1)
cache.put(3,3)
cache.get(2)
cache.put(4,4)
cache.get(1)
cache.get(3)
```

第七步：同理，get(4)，继续调整 4 的位置。



```
LRUCache cache = new LRUCache()
cache.put(1,1)
cache.put(2,2)
cache.get(1)
cache.put(3,3)
cache.get(2)
cache.put(4,4)
cache.get(1)
cache.get(3)
cache.get(4)
```

### 03、LRU 实现（层层剖析）

上面的图搞了我半小时，基本不是弱智的话，应该都能理解 LRU 的使用了。现在我们聊一下实现。LRU 一般来讲，我们是使用**双向链表**实现。基本上在面试的时候，能写出来双向链表的实现，已经可以打 9 分了。但是这里我要强调的是，其实在项目中，并不绝对是这样。比如 Redis 源码里，LRU 的淘汰策略，就没有使用双向链表，而是使用一种模拟链表的方式。因为 Redis 大多是内存中用（我知道可以持久化），如果再在内存中去维护一个链表，就平添了一些复杂性，同时也会多耗掉一些内存，后面我会单独拉出来 Redis 的源码给大家分析，这里不细说。

回到题目，为什么我们要选择双向链表来实现呢？看看上面的使用步骤图，大家会发现，在整个 LRUCache 的使用中，我们需要**频繁的去调整首尾元素的位置**。而双向链表的结构，刚好满足这一点（再啰嗦一下，前几天我刚好看过了 groupcache 的源码，里边就是用双向链表来做的 LRU，当然它里边做了一些改进。groupcache 是 memcache 作者实现的 go 版本，如果有 go 的读者，可以去看看源码，还是有一些收获。）

下面，我们采用 hashmap+ 双向链表的方式进行实现。

首先，我们定义一个 `LinkNode`，用以存储元素。因为是双向链表，自然我们要定义 `pre` 和 `next`。同时，我们需要存储下元素的 `key` 和 `value`。`val` 大家应该都能理解，关键是因为需要存储 `key`？举个例子，比如当整个 cache 的元素满了，此时我们需要删除 map 中的数据，需要通过 `LinkNode` 中的 `key` 来进行查询，否则无法获取到 `key`。

```
1 type LinkNode struct {
2     key, val int
3     pre, next *LinkNode
4 }
```

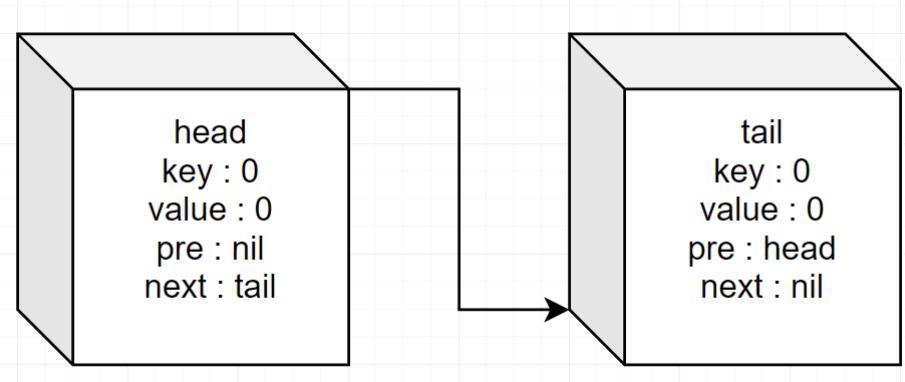
现在有了 LinkNode，自然需要一个 Cache 来存储所有的 Node。我们定义 cap 为 cache 的长度，m 用来存储元素。head 和 tail 作为 Cache 的首尾。

```
1 type LRUCache struct {
2     m          map[int]*LinkNode
3     cap        int
4     head, tail *LinkNode
5 }
```

接下来我们对整个 Cache 进行初始化。在初始化 head 和 tail 的时候将它们连接在一起。

```
1 func Constructor(capacity int) LRUCache {
2     head := &LinkNode{0, 0, nil, nil}
3     tail := &LinkNode{0, 0, nil, nil}
4     head.next = tail
5     tail.pre = head
6     return LRUCache{make(map[int]*LinkNode), capacity, head, tail}
7 }
```

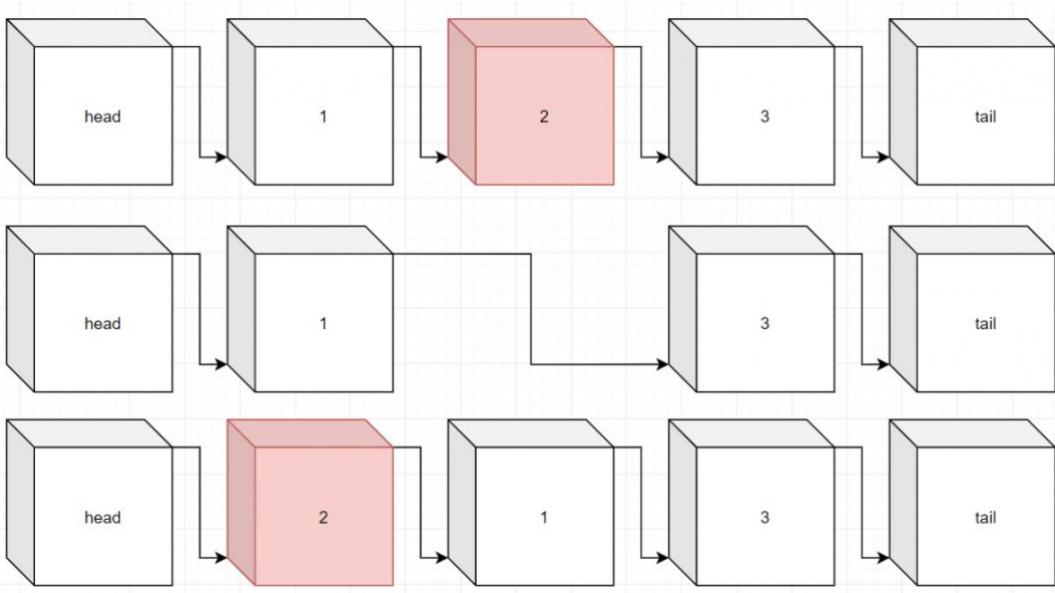
大概是这样：



现在我们已经完成了 Cache 的构造，剩下的就是添加它的 API 了。因为 Get 比较简单，我们先完成 Get 方法。这里分两种情况考虑，如果没有找到元素，我们返回 -1。如果元素存在，我们需要把这个元素移动到首位置上去。

```
1 func (this *LRUCache) Get(key int) int {
2     head := this.head
3     cache := this.m
4     if v, exist := cache[key]; exist {
5         v.pre.next = v.next
6         v.next.pre = v.pre
7         v.next = head.next
8         head.next.pre = v
9         head.next = v
10        v.pre = head
11        return v.val
12    } else {
13        return -1
14    }
15 }
```

大概就是下面这个样子（假若 2 是我们 get 的元素）



我们很容易想到这个方法后面还会用到，所以将其抽出。

```

1 func (this *LRUCache) AddNode(node *LinkNode) {
2     head := this.head
3     //从当前位置删除
4     node.pre.next = node.next
5     node.next.pre = node.pre
6     //移动到首位置
7     node.next = head.next
8     head.next.pre = node
9     node.pre = head
10    head.next = node
11 }
12
13 func (this *LRUCache) Get(key int) int {
14     cache := this.m
15     if v, exist := cache[key]; exist {
16         this.MoveToHead(v)
17         return v.val
18     } else {
19         return -1
20     }
21 }
```

现在我们开始完成 Put。实现 Put 时，有两种情况需要考虑。假若元素存在，其实相当于做一个 Get 操作，也是移动到最前面（**但是需要注意的是，这里多了一个更新值的步骤**）。

```

1 func (this *LRUCache) Put(key int, value int) {
2     head := this.head
3     tail := this.tail
4     cache := this.m
5     //假若元素存在
6     if v, exist := cache[key]; exist {
7         //1.更新值
8         v.val = value
9         //2.移动到最前
10        this.MoveToHead(v)
11    } else {
12        //TODO
13    }
14 }
```

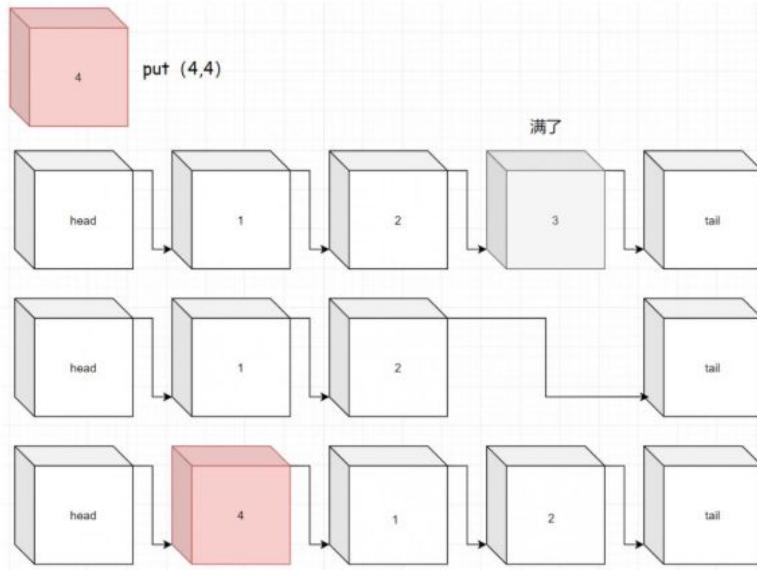
```
13     }
14 }
```

假若元素不存在，我们将其插入到元素首，并把该元素值放入到 map 中。

```
1 func (this *LRUCache) Put(key int, value int) {
2     head := this.head
3     tail := this.tail
4     cache := this.m
5     //存在
6     if v, exist := cache[key]; exist {
7         //1.更新值
8         v.val = value
9         //2.移动到最前
10        this.MoveToHead(v)
11    } else {
12        v := &LinkNode{key, value, nil, nil}
13        v.next = head.next
14        v.pre = head
15        head.next.pre = v
16        head.next = v
17        cache[key] = v
18    }
19 }
```

但是我们漏掉了一种情况，**如果恰好此时Cache中元素满了，需要删掉最后的元素**。处理完毕，附上 Put 函数完整代码。

```
1 func (this *LRUCache) Put(key int, value int) {
2     head := this.head
3     tail := this.tail
4     cache := this.m
5     //存在
6     if v, exist := cache[key]; exist {
7         //1.更新值
8         v.val = value
9         //2.移动到最前
10        this.MoveToHead(v)
11    } else {
12        v := &LinkNode{key, value, nil, nil}
13        if len(cache) == this.cap {
14            //删除最后元素
15            delete(cache, tail.pre.key)
16            tail.pre.pre.next = tail
17            tail.pre = tail.pre.pre
18        }
19        v.next = head.next
20        v.pre = head
21        head.next.pre = v
22        head.next = v
23        cache[key] = v
24    }
25 }
```



最后，我们完成所有代码：

```

1 type LinkNode struct {
2     key, val int
3     pre, next *LinkNode
4 }
5
6 type LRUCache struct {
7     m          map[int]*LinkNode
8     cap        int
9     head, tail *LinkNode
10 }
11
12 func Constructor(capacity int) LRUCache {
13     head := &LinkNode{0, 0, nil, nil}
14     tail := &LinkNode{0, 0, nil, nil}
15     head.next = tail
16     tail.pre = head
17     return LRUCache{make(map[int]*LinkNode), capacity, head, tail}
18 }
19
20 func (this *LRUCache) Get(key int) int {
21     cache := this.m
22     if v, exist := cache[key]; exist {
23         this.MoveToHead(v)
24         return v.val
25     } else {
26         return -1
27     }
28 }
29 func (this *LRUCache) AddNode(node *LinkNode) {
30     head := this.head
31     //从当前位置删除
32     node.pre.next = node.next
33     node.next.pre = node.pre
34     //移动到首位置
35     node.next = head.next
36     head.next.pre = node
37     node.pre = head
38     head.next = node

```

```

39 }
40 func (this *LRUCache) Put(key int, value int) {
41     head := this.head
42     tail := this.tail
43     cache := this.m
44     //存在
45     if v, exist := cache[key]; exist {
46         //1.更新值
47         v.val = value
48         //2.移动到最前
49         this.MoveToHead(v)
50     } else {
51         v := &LinkNode{key, value, nil, nil}
52         if len(cache) == this.cap {
53             //删除最后元素
54             delete(cache, tail.pre.key)
55             tail.pre.pre.next = tail
56             tail.pre = tail.pre.pre
57         }
58         v.next = head.next
59         v.pre = head
60         head.next.pre = v
61         head.next = v
62         cache[key] = v
63     }
64 }
```

优化后：

```

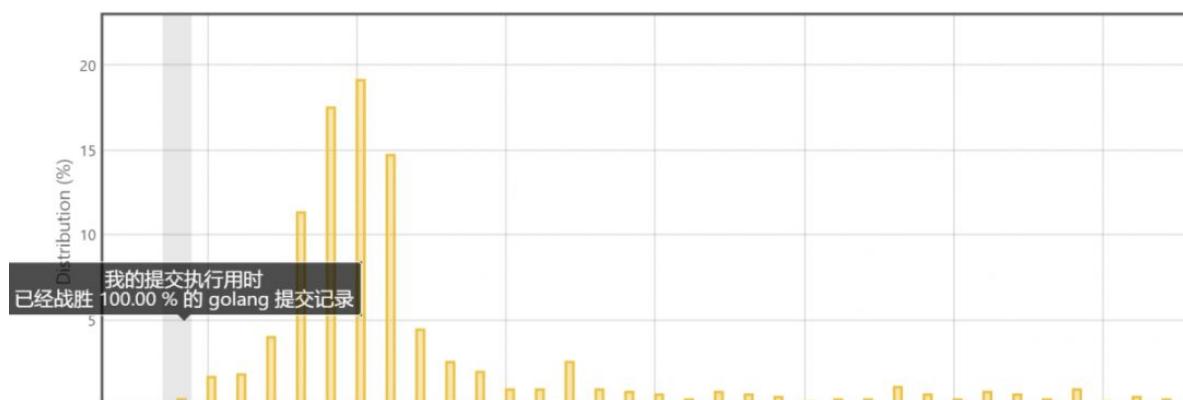
1 type LinkNode struct {
2     key, val int
3     pre, next *LinkNode
4 }
5
6 type LRUCache struct {
7     m          map[int]*LinkNode
8     cap        int
9     head, tail *LinkNode
10 }
11
12 func Constructor(capacity int) LRUCache {
13     head := &LinkNode{0, 0, nil, nil}
14     tail := &LinkNode{0, 0, nil, nil}
15     head.next = tail
16     tail.pre = head
17     return LRUCache{make(map[int]*LinkNode), capacity, head, tail}
18 }
19
20 func (this *LRUCache) Get(key int) int {
21     cache := this.m
22     if v, exist := cache[key]; exist {
23         this.MoveToHead(v)
24         return v.val
25     } else {
26         return -1
27     }
28 }
```

```

29
30     func (this *LRUCache) RemoveNode(node *LinkNode) {
31         node.pre.next = node.next
32         node.next.pre = node.pre
33     }
34
35     func (this *LRUCache) AddNode(node *LinkNode) {
36         head := this.head
37         node.next = head.next
38         head.next.pre = node
39         node.pre = head
40         head.next = node
41     }
42
43     func (this *LRUCache) MoveToHead(node *LinkNode) {
44         this.RemoveNode(node)
45         this.AddNode(node)
46     }
47
48     func (this *LRUCache) Put(key int, value int) {
49         tail := this.tail
50         cache := this.m
51         if v, exist := cache[key]; exist {
52             v.val = value
53             this.MoveToHead(v)
54         } else {
55             v := &LinkNode{key, value, nil, nil}
56             if len(cache) == this.cap {
57                 delete(cache, tail.pre.key)
58                 this.RemoveNode(tail.pre)
59             }
60             this.AddNode(v)
61             cache[key] = v
62         }
63     }

```

执行结果：



因为该算法过于重要，给一个 Java 版本：

```

1 //java版本
2 import java.util.Hashtable;

```

```
3 public class LRUCache {
4     class DlinkedNode {
5         int key;
6         int value;
7         LinkedNode prev;
8         LinkedNode next;
9     }
10
11    private void addNode(DlinkedNode node) {
12        node.prev = head;
13        node.next = head.next;
14        head.next.prev = node;
15        head.next = node;
16    }
17
18    private void removeNode(DlinkedNode node){
19        LinkedNode prev = node.prev;
20        LinkedNode next = node.next;
21        prev.next = next;
22        next.prev = prev;
23    }
24
25    private void moveToHead(DlinkedNode node){
26        removeNode(node);
27        addNode(node);
28    }
29
30    private DlinkedNode popTail() {
31        DlinkedNode res = tail.prev;
32        removeNode(res);
33        return res;
34    }
35
36    private Hashtable<Integer, DlinkedNode> cache =
37        new Hashtable<Integer, DlinkedNode>();
38    private int size;
39    private int capacity;
40    private DlinkedNode head, tail;
41
42    public LRUCache(int capacity) {
43        this.size = 0;
44        this.capacity = capacity;
45        head = new DlinkedNode();
46        tail = new DlinkedNode();
47        head.next = tail;
48        tail.prev = head;
49    }
50
51    public int get(int key) {
52        DlinkedNode node = cache.get(key);
53        if (node == null) return -1;
54        moveToHead(node);
55        return node.value;
56    }
57
58    public void put(int key, int value) {
59        DlinkedNode node = cache.get(key);
60
```

```
61     if(node == null) {
62         DLinkedNode newNode = new DLinkedNode();
63         newNode.key = key;
64         newNode.value = value;
65         cache.put(key, newNode);
66         addNode(newNode);
67         ++size;
68         if(size > capacity) {
69             DLinkedNode tail = popTail();
70             cache.remove(tail.key);
71             --size;
72         }
73     } else {
74         node.value = value;
75         moveToHead(node);
76     }
77 }
78 }
79 }
```

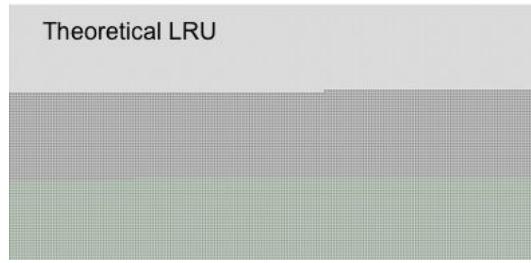
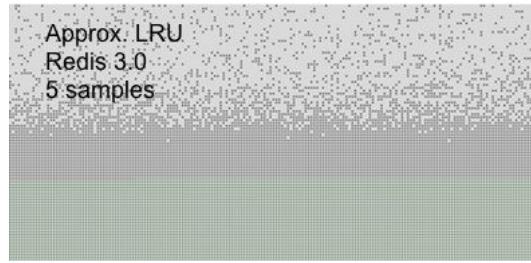
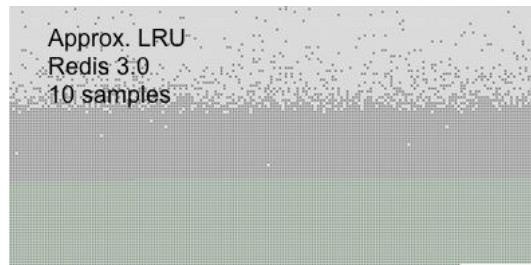
## 04、Redis 近LRU 介绍

上文完成了咱们自己的 LRU 实现，现在聊一聊 Redis 中的近似 LRU。由于**真实LRU需要过多的内存（在数据量比较大时）**，所以 Redis 是使用一种随机抽样的方式，来实现一个近似 LRU 的效果。说白了，LRU 根本只是一个预测键访问顺序的模型。

在 Redis 中有一个参数，叫做“maxmemory-samples”，是干嘛用的呢？

```
1 # LRU and minimal TTL algorithms are not precise algorithms but approximated
# algorithms (in order to save memory), so you can tune it for speed or
# accuracy. For default Redis will check five keys and pick the one that was
# used less recently, you can change the sample size using the following
# configuration directive. 6#
# The default of 5 produces good enough results. 10 Approximates very closely
# true LRU but costs a bit more CPU. 3 is very fast but not very accurate.
# 10maxmemory-samples 5
```

上面我们说过了，**近似LRU是用随机抽样的方式来实现一个近似的LRU效果**。这个参数其实就是作者提供了一种方式，可以让我们人为干预样本数大小，将其设的越大，就越接近真实 LRU 的效果，当然也就意味着越耗内存。（初始值为 5 是作者默认的最佳）



这个图解释一下，绿色的点是新增加的元素，深灰色的点是没有被删除的元素，浅灰色的是被删除的元素。最下面的这张图，是真实 LRU 的效果，第二张图是默认该参数为 5 的效果，可以看到浅灰色部分和真实的契合还是不错的。第一张图是将该参数设置为 10 的效果，已经基本接近真实 LRU 的效果了。

今天基本就说到这里。那 Redis 中的近似 LRU 是如何实现的呢？因为时间的关系，我打算做到下一期的内容。最后，评论区留下你的想法吧！

## 最小的k个数

今天分享一道比较简单的题目，希望大家可以5分钟掌握！

### 01、题目标例

#### 最小的k个数

输入整数数组 arr，找出其中最小的 k 个数。例如，输入4、5、1、6、2、7、3、8这8个数字，则最小的4个数字是1、2、3、4。

#### 示例 1：

- 1 | 输入: arr = [3,2,1], k = 2
- 2 | 输出: [1,2] 或者 [2,1]

#### 示例 2：

```
1 | 输入: arr = [0,1,2,1], k = 1  
2 | 输出: [0]
```

### 限制:

```
1 | 0 <= k <= arr.length <= 10000  
2 | 0 <= arr[i] <= 10000
```

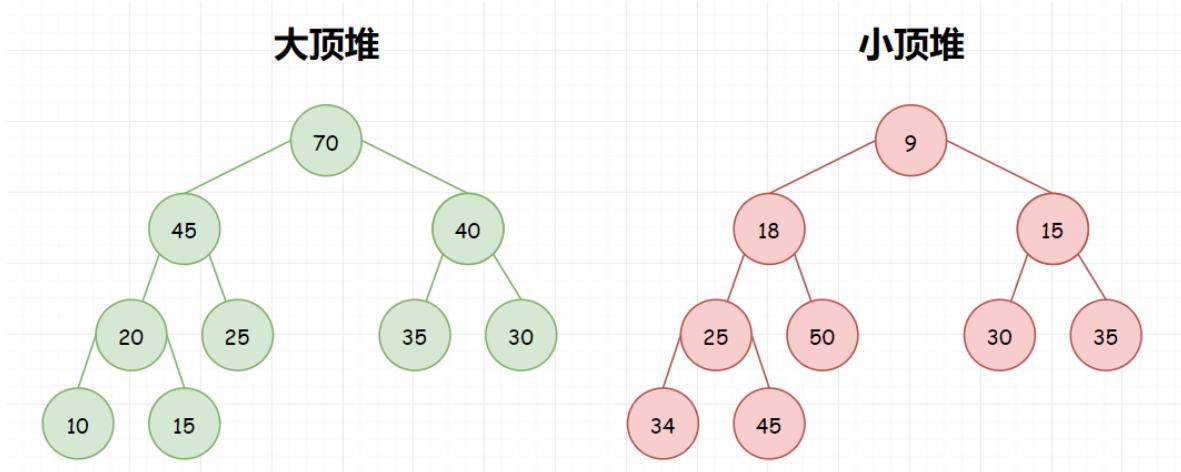
## 02、堆和大小顶堆

这道题出自《剑指offer》，是一道非常高频的题目。可以通过排序等多种方法求解。但是这里，我们使用较为经典的大顶堆（大根堆）解法进行求解。因为我知道有很多人可能一脸懵逼，所以，我们先复习一下大顶堆。

首先复习一下堆，堆(Heap)是计算机科学中一类特殊的数据结构的统称，我们通常是指一个可以被看做一棵完全二叉树的数组对象。如果不记得什么是完全二叉树，可以复习这篇：

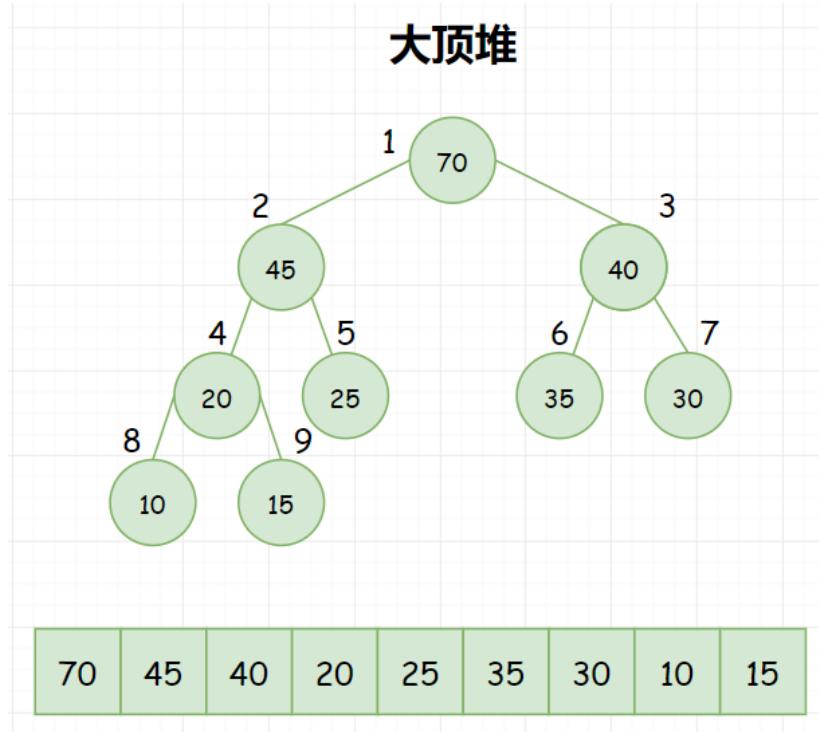
[二叉树第七讲 - 完全二叉树\(222\)](#)

堆的特性是父节点的值总是比其两个子节点的值大或小。如果父节点比它的两个子节点的值都要大，我们叫做**大顶堆**。如果父节点比它的两个子节点的值都要小，我们叫做**小顶堆**。



我们对堆中的结点按层进行编号，将这种逻辑结构映射到数组中就是下面这个样子。

## 大顶堆

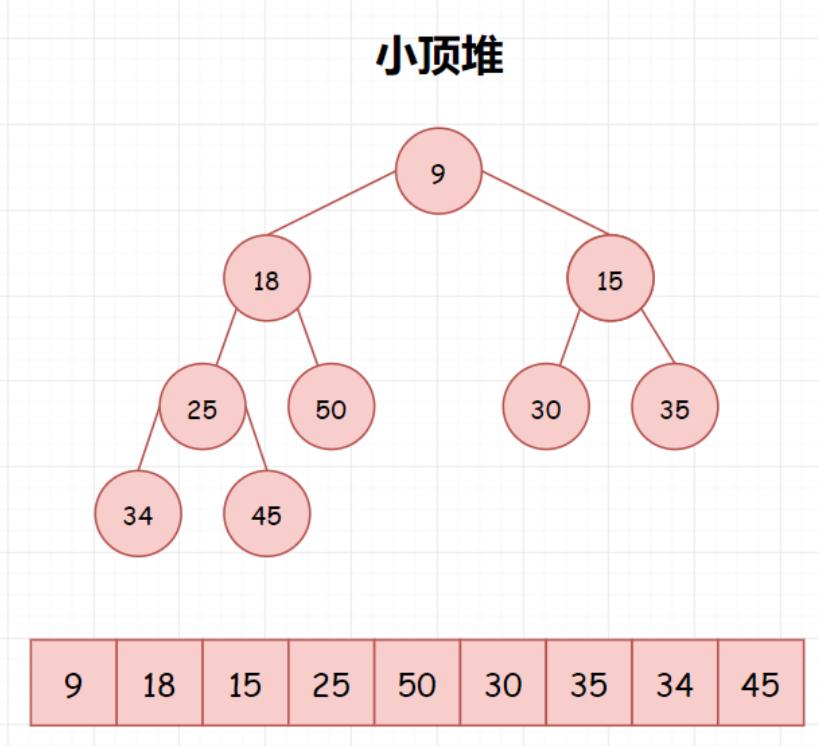


大顶堆，满足以下公式：

arr[i] >= arr[2i 1] && arr[i] >= arr[2i 2]

小顶堆也一样：

## 小顶堆



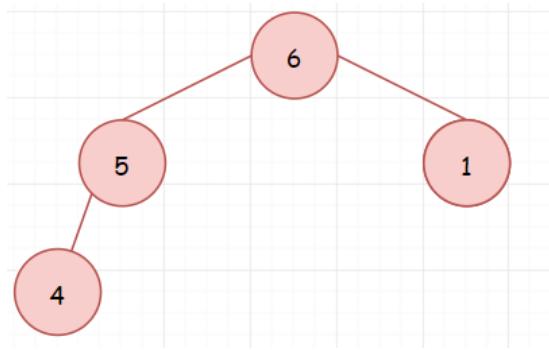
小顶堆，满足以下公式：

arr[i] <= arr[2i 1] && arr[i] <= arr[2i 2]

## 03、题目分析

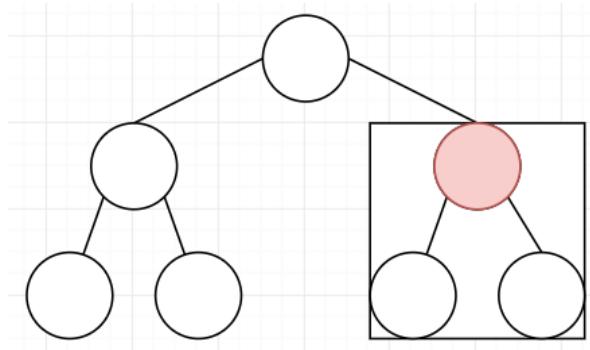
上面我们学习了大顶堆，现在考虑如何用大根堆进行求解。

首先，我们创建一个大小为k的大顶堆。假如数组为[4,5,1,6,2,7,3,8]， k=4。大概是下面这样：

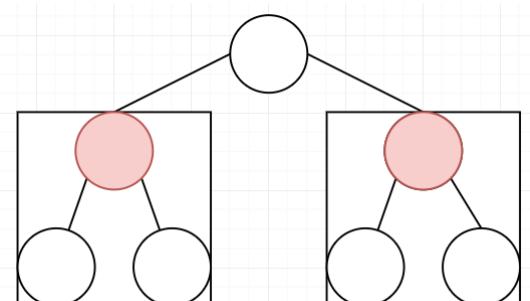


我想肯定这里有不知道如何建堆的同学。记住：对于一个没有维护过的堆（完全二叉树），我们可以从其最后一个节点的父节点开始进行调整。这个不需要死记硬背，其实就是一个层层调节的过程。

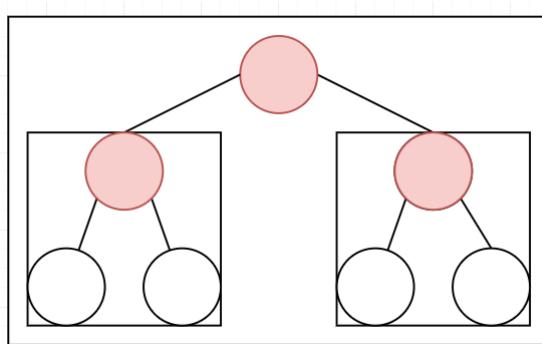
从最后一个节点的父节点调整



继续向上调整



继续向上调整



建堆 调整的代码大概就是这样：（翻Java牌子）

```

1 //建堆。对于一个还没维护过的堆，从他的最后一个节点的父节点开始进行调整。
2 private void buildHeap(int[] nums) {
3     //最后一个节点
4     int lastNode = nums.length - 1;
5     //记住：父节点 = (i - 1) / 2 左节点 = 2 * i + 1 右节点 = 2 * i + 2;

```

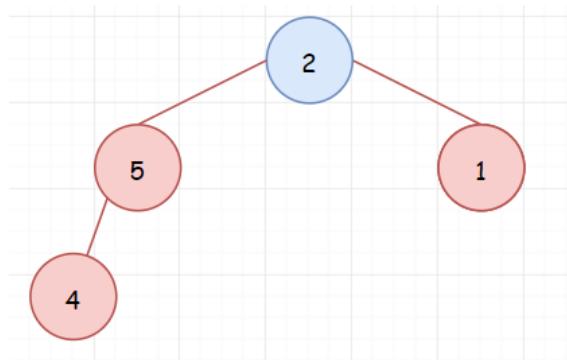
```

6 //最后一个节点的父节点 7
7 int startHeapify = (lastNode - 1) / 2;
8 while (startHeapify >= 0) {
9     //不断调整建堆的过程
10    heapify(nums, startHeapify--);
11 }
12 }
13 //调整大顶堆的过程
14 private void heapify(int[] nums, int i) {
15     //和当前节点的左右节点比较，如果节点中有更大的数，那么交换，并继续对交换后的节点进行维
16     //护
17     int len = nums.length;
18     if (i >= len)
19         return;
20     //左右子节点
21     int c1 = ((i << 1) - 1), c2 = ((i << 1) + 2);
22     //假定当前节点最大
23     int max = i;
24     //如果左子节点比较大，更新max = c1;
25     if (c1 < len && nums[c1] > nums[max]) max = c1;
26     //如果右子节点比较大，更新max = c2;
27     if (c2 < len && nums[c2] > nums[max]) max = c2;
28     //如果最大的数不是节点i的话，那么heapify(nums, max)，即调整节点i的子树。
29     if (max != i) {
30         swap(nums, max, i);
31         //递归处理
32         heapify(nums, max);
33     }
34     private void swap(int[] nums, int i, int j) {
35         nums[i] = nums[i] - (nums[j] = nums[i]);
36     }

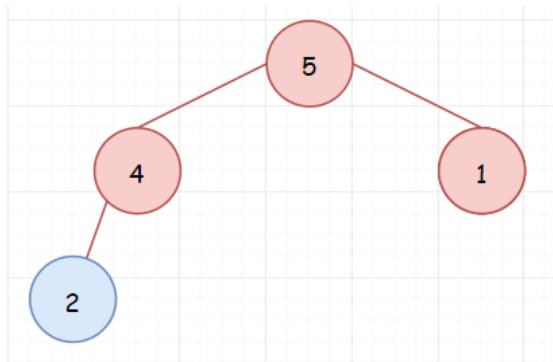
```

然后我们从下标 k 继续开始依次遍历数组的剩余元素。如果元素小于堆顶元素，那么取出堆顶元素，将当前元素入堆。在上面的示例中，因为2小于堆顶元素6，所以将2入堆。我们发现现在的完全二叉树不满足大顶堆，所以对其进行调整。

调整前

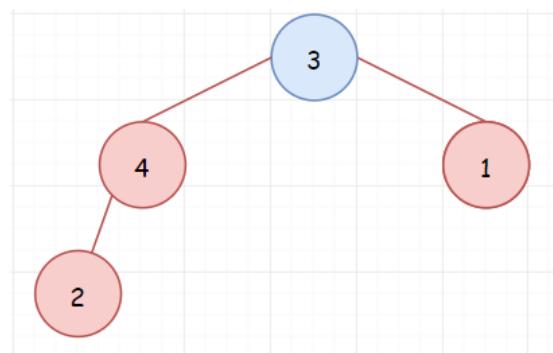


调整后

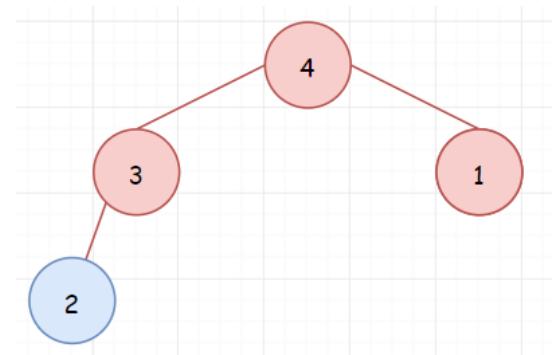


继续重复上述步骤，依次将7,3,8入堆。这里因为7和8都大于堆顶元素5，所以只有3会入堆。

调整前



调整后



最后得到的堆，就是我们想要的结果。由于堆的大小是 K，所以这里空间复杂度是  $O(K)$ ，时间复杂度是  $O(N \log K)$ 。

根据分析，完成代码：

```

1 //java
2 class Solution {
3     public int[] getLeastNumbers(int[] arr, int k) {
4         if (k == 0)
5             return new int[0];
6         int len = arr.length;
7         if (k == len)
8             return arr;
9         //对arr数组的前k个数建堆
10        int[] heap = new int[k];
11        System.arraycopy(arr, 0, heap, 0, k);
12        buildHeap(heap);
13    }
}

```

```

14     //对后面较小的树建堆
15     for (int i = k; i < len; i++) {
16         if (arr[i] < heap[0]) {
17             heap[0] = arr[i];
18             heapify(heap, 0);
19         }
20     }
21     //返回这个堆
22     return heap;
23 }
24 private void buildHeap(int[] nums) {
25     int lastNode = nums.length - 1;
26     int startHeapify = (lastNode - 1) / 2;
27     while (startHeapify >= 0) {
28         heapify(nums, startHeapify--);
29     }
30 }
31 private void heapify(int[] nums, int i) {
32     int len = nums.length;
33     if (i >= len)
34         return;
35     int c1 = ((i << 1) - 1), c2 = ((i << 1) - 2);
36     int max = i;
37     if (c1 < len && nums[c1] > nums[max]) max = c1;
38     if (c2 < len && nums[c2] > nums[max]) max = c2;
39     if (max != i) {
40         swap(nums, max, i);
41         heapify(nums, max);
42     }
43 }
44 private void swap(int[] nums, int i, int j) {
45     nums[i] = nums[i] - nums[j] + (nums[j] = nums[i]);
46 }
47 }
```

执行结果：

执行用时：5 ms，在所有 Java 提交中击败了 83.07% 的用户

内存消耗：42.9 MB，在所有 Java 提交中击败了 100.00% 的用户

## 不同路径

今天为大家分享一道BAT常考题目，不同路径。

### 01、题目示例

该题很容易出现在各大厂的面试中，一般会要求手写，所以需要完整掌握。

## 不同路径

一个机器人位于一个  $m \times n$  网格的左上角，起始点在下图中标记为“Start”。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角，在下图中标记为“Finish”。问：总共有多少条不同的路径？



例如，上图是一个 $7 \times 3$  的网格。有多少可能的路径？

**说明：**  $m$  和  $n$  的值均不超过 100。

### 示例 1:

```
1 | 输入: m = 3, n = 2
2 | 输出: 3
3 |
4 | 解释:
5 | 从左上角开始, 总共有 3 条路径可以到达右下角。
6 | \1. 向右 -> 向右 -> 向下
7 | \2. 向右 -> 向下 -> 向右
8 | \3. 向下 -> 向右 -> 向右
```

### 示例 2:

```
1 | 输入: m = 7, n = 3
2 | 输出: 28
```

## 02、题目分析

这道题属于相当标准的动态规划，虽然还有一些公式法等其他解法，但是如果面试官问到，基本就是想考察你的动态规划。

拿到题目，首先定义状态。因为有横纵坐标，明显属于二维DP。我们定义  $DP[i][j]$  表示到达  $i$  行  $j$  列的最多路径。同时，因为第0行和第0列都只有一条路径，所以需要初始化为1。

1	1	1	1	1	1	1
1						
1						

状态转移方程一目了然， $dp[i][j] = dp[i-1][j] + dp[i][j-1]$ 。（想象你站在一个十字路口，到达这个十字路口可能的所有路径，就是从东南西北四个方向过来可能出现的所有路径和。放在这道题里，其实就是砍掉东南。）


根据分析，完成代码：

```

1 //go
2 func uniquePaths(m int, n int) int {
3     dp := make([][]int, m)
4     for i := 0; i < m; i {
5         dp[i] = make([]int, n)
6     }
7     for i := 0; i < m; i {
8         dp[i][0] = 1
9     }
10    for j := 0; j < n; j {
11        dp[0][j] = 1
12    }
13    for i := 1; i < m; i {
14        for j := 1; j < n; j {
15            dp[i][j] = dp[i-1][j] + dp[i][j-1]
16        }
17    }
}

```

```
18     return dp[m-1][n-1]
19 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：2 MB，在所有 Go 提交中击败了 46.70% 的用户

### 03、代码优化

上面的答案，如果在面试时给出，可以给到7分，后面3分怎么拿，我们真的需要用一个二维数组来存储吗？一起看下！

在上文中，我们使用**二维数组**记录状态。但是这里观察一下，每一个格子可能的路径，**都是由左边的格子和上面的格子的总路径计算而来，对于之前更早的数据，其实已经用不到了**。如下图，计算第三行时，已经用不到第一行的数据了。

1	1	1	1	1	1	1

1	1	1	1	1	1	1
1	2	3	4	5	6	7

1	1	1	1	1	1	1
1	2	3	4	5	6	7
1	3	6	10	15	21	28

那我们只要能定义一个状态，同时可以表示左边的格子和上面的格子，是不是就可以解决问题？所以我们定义状态 $dp[j]$ ，用来表示**当前行到达第j列的最多路径**。这个“当前行”三个字很重要，比如我们要计算 $dp[3]$ ，因为还没有计算出，所以这时 $dp[3]$ 保存的其实是4（上一行的数据），而 $dp[2]$ 由于已经计算出了，所以保存的是6（当前行的数据）。理解了这个，就理解如何压缩状态。

1	1	1	1	1	1	1
1	2	3	4	5	6	7
1	3	6	10	15	21	28

最后，根据分析得出代码：

```

1 //go
2 func uniquePaths(m int, n int) int {
3     dp := make([]int, n)
4     for j := 0; j < n; j {
5         dp[j] = 1
6     }
7     for i := 1; i < m; i {
8         for j := 1; j < n; j {
9             //注意，这里dp[j-1]已经是新一行的数据了，而dp[j]仍然是上一行的数据
10            dp[j] = dp[j - 1]
11        }
12    }
13    return dp[n-1]
14 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：1.9 MB，在所有 Go 提交中击败了 99.23% 的用户

## 不同路径 - 障碍物

上一篇为大家分享了不同路径的DP解法之后，有很多小伙伴后台给我留言，说直接用公式法一步就可以得到答案。给你们点个赞！确实是这样，我没有用公式法的原因，是因为我的目的是想层层推进难度为大家分析不同路径这一类题型。后面我会单独拉出一系列，专门为给大家讲解公式法一类的题目。

如果还没有学习上一篇内容，建议先进行学习：

[不同路径](#)

## 01、题目示例

多了一点障碍物之后，题目会有何不同？（这可是困难题目哦~）

### 不同路径 - 障碍物

一个机器人位于一个  $m \times n$  网格的左上角，起始点在下图中标记为“Start”。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角，在下图中标记为“Finish”。现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？问总共有多少条不同的路径？



网格中的障碍物和空位置分别用 1 和 0 来表示。

**说明：**  $m$  和  $n$  的值均不超过 100。

#### 示例 1:

```
1 | 输入：  
2 | [  
3 |   [0,0,0],  
4 |   [0,1,0],  
5 |   [0,0,0]  
6 | ]  
7 | 输出： 2  
8 |  
9 | 解释：  
10| 3x3 网格的正中间有一个障碍物。  
11| 从左上角到右下角一共有 2 条不同的路径：  
12| \1. 向右 -> 向右 -> 向下 -> 向下  
13| \2. 向下 -> 向下 -> 向右 -> 向右
```

## 02、题目分析

因为只是多了一点障碍物，题目的本质并没什么不同，所以直接进行分析即可。

首先我们还是定义状态，用  $DPI[i][j]$  表示到达  $i$  行  $j$  列的最多路径。同时，因为第 0 行和第 0 列都只有一条路径，所以需要初始化为 1。但有一点不一样的就是：如果在 0 行 0 列中遇到障碍物，后面的就都是 0，意为 **此路不通**。

1	1	0	0	0	0	0
0						
0						

完成了初始化，下面就是状态转移方程。和没有障碍物的相比没什么特别的，仍然是 $dp[i][j] = dp[i-1][j]$   $dp[i][j-1]$ 。唯一需要处理的是：如果恰好 $[i][j]$ 位置上有障碍物，则 $dp[i][j]$ 为0。比如下图，有 $dp[1][2]$ 为0。

		0				

根据分析，得出代码：（今天翻java牌子）

```

1 //JAVA
2 class Solution {
3     public int uniquePathsWithObstacles(int[][] obstacleGrid) {
4         int m = obstacleGrid.length;
5         int n = obstacleGrid[0].length;
6         int[][] dp = new int[m][n];
7         if (obstacleGrid[0][0] != 1) {
8             dp[0][0] = 1;
9         }
10        for (int j = 1; j < n; j++) {
11            dp[0][j] = obstacleGrid[0][j] == 1 ? 0 : dp[0][j - 1];
12        }
13        for (int i = 1; i < m; i++) {
14            dp[i][0] = obstacleGrid[i][0] == 1 ? 0 : dp[i - 1][0];
15        }
16        for (int i = 1; i < m; i++) {
17            for (int j = 1; j < n; j++) {
18                dp[i][j] = obstacleGrid[i][j] == 1 ? 0 : dp[i - 1][j]
19                                + dp[i][j - 1];
20            }
21        }
22        return dp[m - 1][n - 1];
23    }
}

```

```
22 }  
23 }
```

执行结果：

执行用时：1 ms，在所有 Java 提交中击败了 85.23% 的用户

内存消耗：37.7 MB，在所有 Java 提交中击败了 58.26% 的用户

### 03、代码优化

不啰嗦，我们当然要继续**压缩内存**。

为了大家更好的理解代码，我们还是绘图说明。假若我们的网格如下，其中黑色表示障碍物。

0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0

因为计算每一个格子能到达的最多路径，只需要左边和上边的元素，所以我们定义状态**dp[j]**表示到达当前行第j位置的最多路径。这里有一个需要额外说的，就是我们把dp[0]初始化为1，因为在到达第一行的第一个元素时，路径只有一个。下面的图，左边的是当前网格，右边的是指网格中对应dp数组的值。

0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
1	1	1	0	0	0	0

0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
1	1	1	0	0	0	0
1	0	1	1	1	0	0

0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
1	1	1	0	0	0	0
1	0	1	1	1	0	0
1	1	2	3	4	4	4

根据分析，得出代码：

```

1 //JAVA
2 class Solution {
3     public int uniquePathsWithObstacles(int[][] obstacleGrid) {
4         int m = obstacleGrid.length;
5         int n = obstacleGrid[0].length;
6         int[] dp = new int[n];
7         dp[0] = 1;
8         for (int[] ints : obstacleGrid) {
9             for (int j = 0; j < n; j++) {
10                 if (ints[j] == 1) {
11                     dp[j] = 0;
12                 } else if (j > 0) {
13                     dp[j] = dp[j - 1];
14                 }
15             }
16         }
17         return dp[n - 1];
18     }
19 }
```

思考：上面的代码其实还可以优化内存，大家想想怎么做

## 伪蒙特卡洛

分享一道由群员“Melbourne”，外号“Paper Machine”，有数学小王子之称的小伙伴分享的题目！

**特别说明：本文非原创，经投稿者同意后发表。**

## 01、算法介绍

期望：在概率论和统计学中，数学期望(mean)（或均值，亦简称期望）是试验中每次可能结果的概率乘以其结果的总和，是最基本的数学特征之一。它反映随机变量平均取值的大小。

**题目：在1\*1的正方形中随机撒三个点，两两点都可构成长方形的一组对顶点，这样一共有三个长方形，需要求面积第二大的长方形的面积的期望。**

算法：每次随机三个点，计算第二大面积，最后统计期望。

## 02、蒙特卡洛

蒙特卡罗法也称统计模拟法、统计试验法。是把概率现象作为研究对象的数值模拟方法。是按抽样调查法求取统计值来推定未知特性量的计算方法。蒙特卡罗是摩纳哥的著名赌城，该法为表明其随机抽样的本质而命名。故适用于对离散系统进行计算仿真试验。在计算仿真中，通过构造一个和系统性能相近似的概率模型，并在数字计算机上进行随机试验，可以模拟系统的随机特性。

蒙特卡洛方法(Monte Carlo Method) 指的是一类使用随机变量解决概率问题的方法。比较常见的是计算积分、计算概率、计算期望等问题。

常见的蒙特卡洛方法依赖于随机变量的“随机性”，即未发生的事件无法根据已有信息进行预测，比如抛硬币、掷骰子等。在计算机中，常见的随机数是由一系列确定性算法进行生成的，通常称之为伪随机数(pseudo random number)。由于计算精度有限，且这些随机数在统计意义上“不够随机”，会出现可预测的重复序列，这些数在统计意义上收敛精度有限。

与常见的蒙特卡洛方法不同的是，伪蒙特卡洛使用了低差异序列(low discrepancy sequence，常见的有halton序列、sobol序列等)，不使用常见的（伪）随机数，其收敛速率更快（记  $N$  为样本数量，伪蒙特卡洛收敛速率可达  $O(\frac{1}{N})$ ），而普通蒙特卡洛方法收敛速率仅为  $O(\frac{1}{N^2})$ 。另一个最重要的性质是伪蒙特卡洛使用的低差异序列是可复现的(replicable)，即不会随环境改变而改变，没有随机种子；而普通蒙特卡洛使用的伪随机数会因随机种子不同而导致结果不同，收敛效果也不尽相同。

## 03、题目分析

本算法利用伪蒙特卡洛完成。

CPP代码如下：

```
1 #include <cmath>
```

```

2 #include <cstdio>
3 #include <vector>
4 #include <cassert>
5 #include <omp.h>
6 const int UP=100;
7 bool sieve[UP 100];
8 int primes[UP],top=0;
9 void init()
10 {
11     for (int i=2;i<=UP; i)
12         if (!sieve[i])
13         {
14             primes[top]=i;
15             for (int j=i;j<=UP/i; j)
16                 sieve[i*j]=true;
17         }
18 }
19 std::vector<double> halton(long long i,const int &dim)
20 {
21     assert(dim<=top);
22     std::vector<double> prime_inv(dim,0),r(dim,0);
23     std::vector<long long> t(dim,i);
24     for (int j=0;j<dim; j)
25         prime_inv[j]=1.0/primes[j];
26     auto f=[](const std::vector<long long> &t)->long long {
27         long long ret=0;
28         for (const auto &e:t)
29             ret =e;
30         return ret;
31     };
32     for (;f(t)>0;)
33         for (int j=0;j<dim; j)
34         {
35             long long d=t[j]%primes[j];
36             r[j] =d*prime_inv[j];
37             prime_inv[j]/=primes[j];
38             t[j]/=primes[j];
39         }
40     return r;
41 }
42 double experiment(long long idx){
43     std::vector<double> li=halton(idx,6);
44     double area1=fabs((li.at(0)-li.at(2))*(li.at(1)-li.at(3)));
45     double area2=fabs((li.at(0)-li.at(4))*(li.at(1)-li.at(5)));
46     double area3=fabs((li.at(2)-li.at(4))*(li.at(3)-li.at(5)));
47     double w=area1 area2 area3-std::max(std::max(area1,area2),area3)-
48     std::min(std::min(area1,area2),area3);
49     return w;
50 }
51 const int BATCH=100000;
52 const int THREADS=40;
53 int main()
54 {
55     init();
56     double total=0;
57     for (long long trial=0;;)
58     {
59         std::vector<double> li(THREADS,0);

```

```

59     omp_set_dynamic(0);
60     omp_set_num_threads(THREADS);
61     #pragma omp parallel for
62     for (long long thread=0;thread<THREADS; thread)
63     {
64         for (long long i=0;i<BATCH; i)
65             li.at(thread) =experiment(trial thread*BATCH i);
66     }
67     for (const auto &d:li)
68         total =d;
69     trial =THREADS*BATCH;
70     printf("%ld: %.10f\n",trial,total/trial),fflush(stdout);
71 }
72 return 0;
73 }
```

分析：使用了并行计算，批量跑随机实验，速度大大提升。其中halton函数会生成halton低差异序列，其值域为[0, 1]，参数i表示第i个抽样，dim表示生成数据的维度（本例中每次实验需要6个点，使用6维数据点即可），不同样本之间互不影响，故可使用并行计算提速。

#表示随机试验次数 $\times 10^7$ , Avg表示第二大面积的平均值, Err表示与真实值的绝对误差 $\times 10^{-10}$ 。

#	Avg	Err	#	Avg	Err
1	0.1017786804	55	2	0.1017786707	152
3	0.1017786905	46	4	0.1017786889	30
5	0.1017786809	50	6	0.1017786836	23
7	0.1017786849	10	8	0.1017786868	9
9	0.1017786799	60	10	0.1017786837	22
11	0.1017786845	14	12	0.1017786839	20
13	0.1017786874	15	14	0.1017786839	20
15	0.1017786848	11	16	0.1017786868	9
17	0.1017786851	8	18	0.1017786863	4
19	0.1017786854	5	20	0.1017786887	28
21	0.1017786858	1	22	0.1017786844	15
23	0.1017786841	18	24	0.1017786852	7
25	0.1017786849	10	26	0.101778684	19

#	Avg	Err	#	Avg	Err
27	0.1017786838	21	28	0.1017786852	7
29	0.1017786838	21	30	0.1017786846	13
31	0.1017786859	0	32	0.1017786862	3
33	0.1017786859	0	34	0.1017786853	6
35	0.1017786854	5	36	0.1017786859	0
37	0.101778685	9	38	0.1017786854	5
39	0.1017786853	6	40	0.1017786858	1
41	0.1017786848	11	42	0.1017786851	8
43	0.1017786847	12	44	0.1017786841	18
45	0.101778685	9	46	0.1017786842	17
47	0.1017786852	7	48	0.1017786848	11
49	0.1017786854	5	50	0.1017786851	8
51	0.1017786842	17	52	0.1017786844	15

可以看到，在实验次之后，收敛精度可达9位小数，非常精确。由于使用的随机数“不够随机”，普通的蒙特卡洛在同样的实验次数下仅能收敛至五位小数的精度。

上述方法可扩展至其他随机问题中，非常实用且高效，欢迎大家讨论！

所以，今天的问题你学会了吗？评论区留下你的想法！

## 盛最多水的容器

今天为大家分享一道鹅厂的面试题。话不多说，直接看题目。

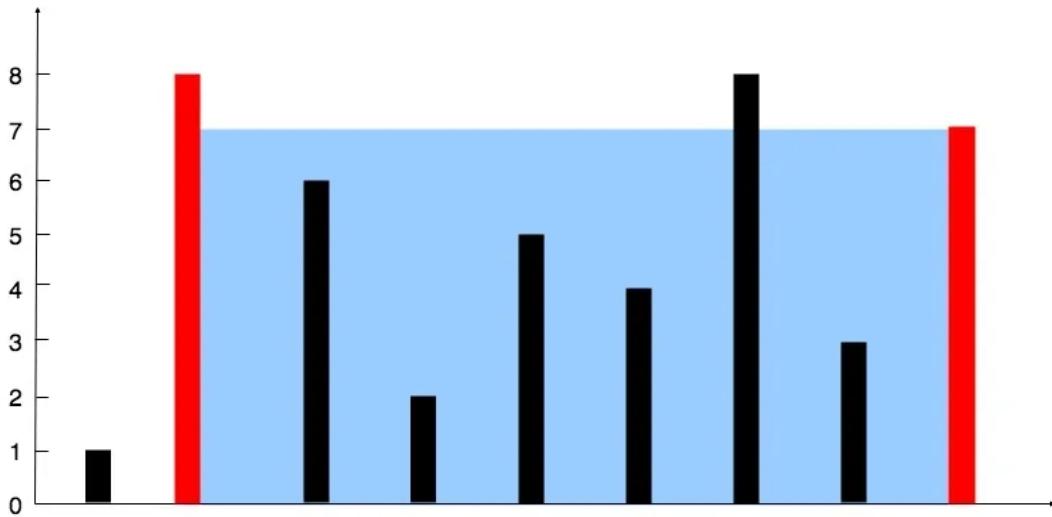
### 01、题目标例

这道题目会了的朋友可能觉得很简单，但是我觉得这题实在很经典，所以还是得拿出来讲讲。还有一个进阶版本“接雨水”，将在后面为大家讲解。

## 盛最多水的容器

给你  $n$  个非负整数  $a_1, a_2, \dots, a_n$ , 每个数代表坐标中的一个点  $(i, a_i)$ 。在坐标内画  $n$  条垂直线, 垂直线  $i$  的两个端点分别为  $(i, a_i)$  和  $(i, 0)$ 。找出其中的两条线, 使得它们与  $x$  轴共同构成的容器可以容纳最多的水。

说明: 你不能倾斜容器, 且  $n$  的值至少为 2。



图中垂直线代表输入数组  $[1, 8, 6, 2, 5, 4, 8, 3, 7]$ 。在此情况下, 容器能够容纳水 (表示为蓝色部分) 的最大值为 49。

示例:

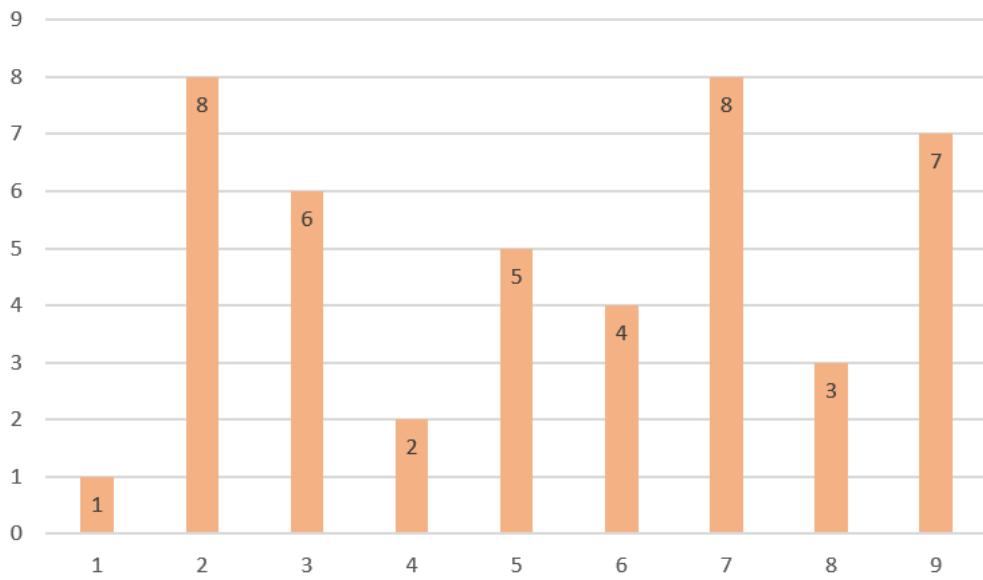
```
1 | 输入: [1, 8, 6, 2, 5, 4, 8, 3, 7]
2 | 输出: 49
```

## 02、题目分析

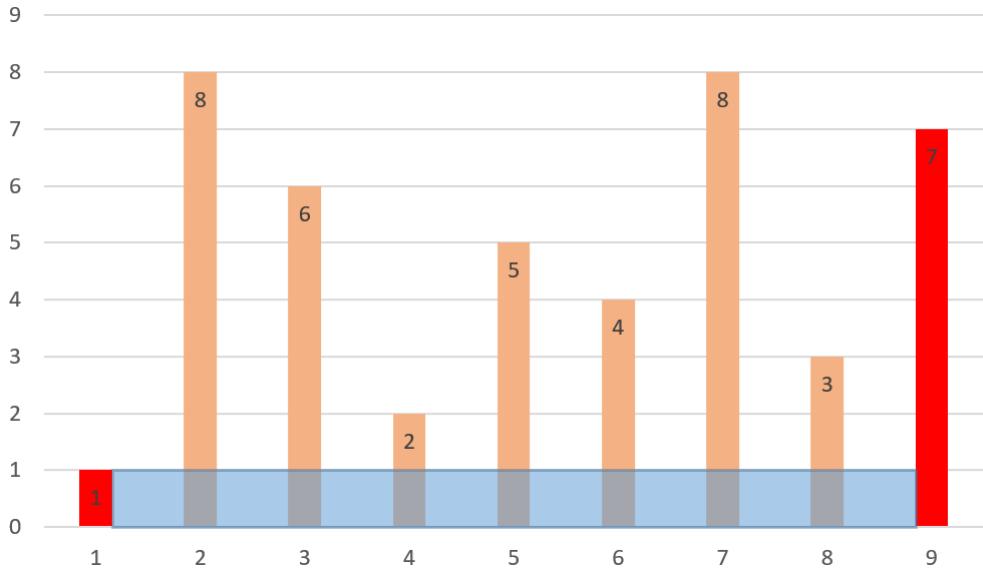
观察可得, 垂直的两条线段将会与坐标轴构成一个矩形区域, 较短线段的长度将会作为矩形区域的宽度, 两线间距将会作为矩形区域的长度, 我们求解容纳水的最大值, 实为找到该矩形最大化的区域面积。

首先, 本题自然可以暴力求解, 只要找到每对可能出现的线段组合, 然后找出这些情况下的最大面积。这种解法直接略过, 大家有兴趣可以下去自己尝试。这道题比较经典是使用双指针进行求解, 已经会的朋友不妨复习复习。

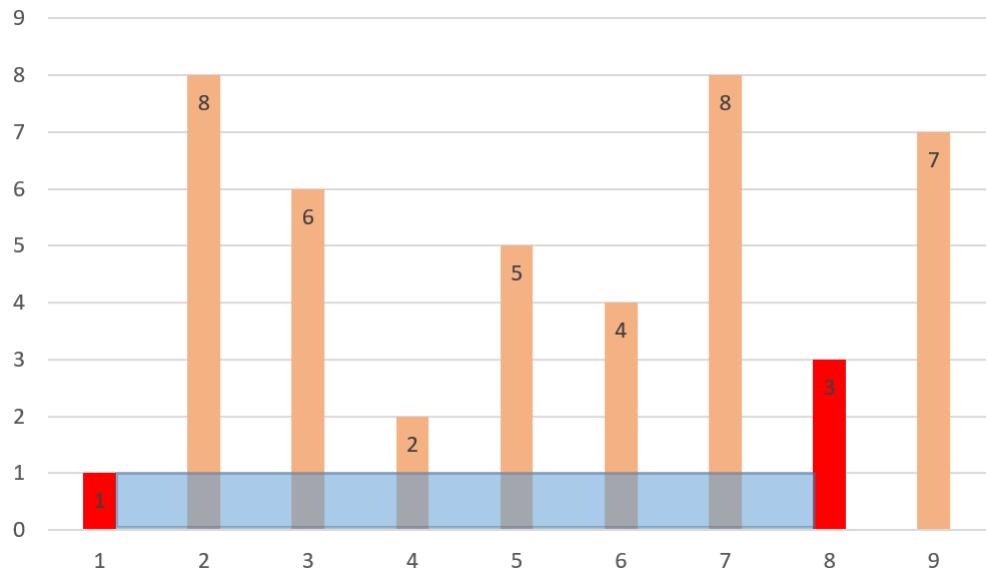
假若我们的数组为: [1 8 6 2 5 4 8 3 7], 长这样:



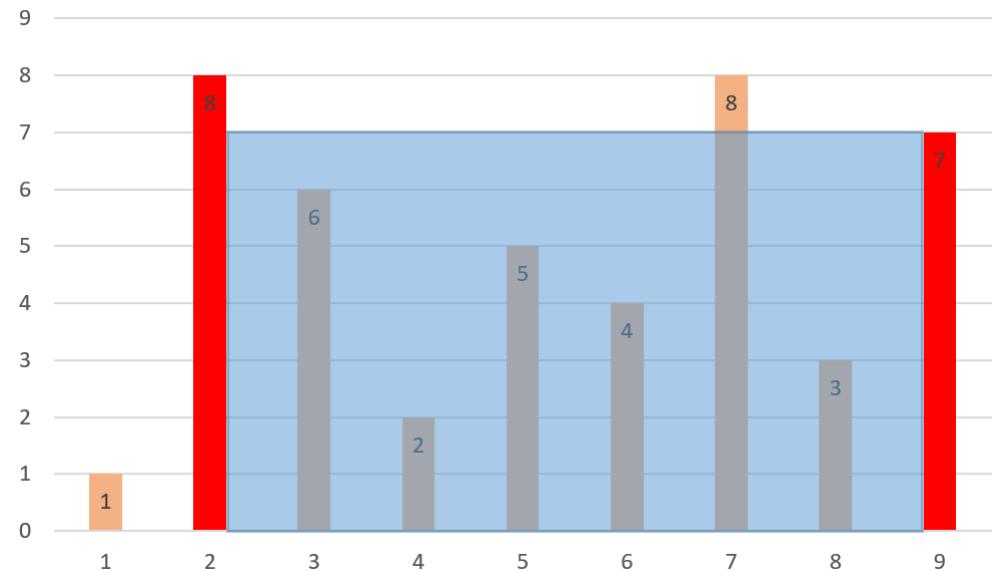
首先，我们初始化两个指针，分别指向两边，构成我们的第一个矩形区域。



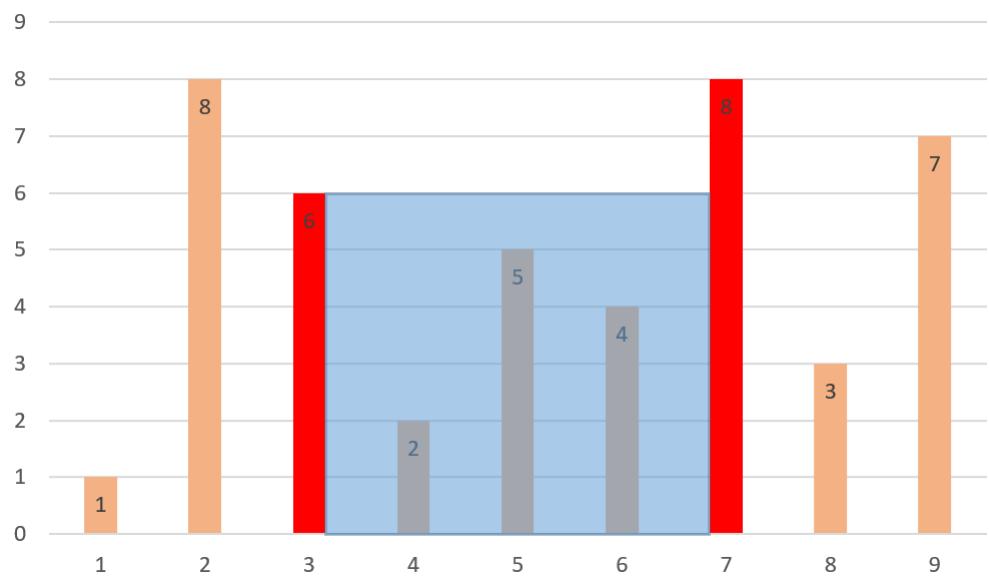
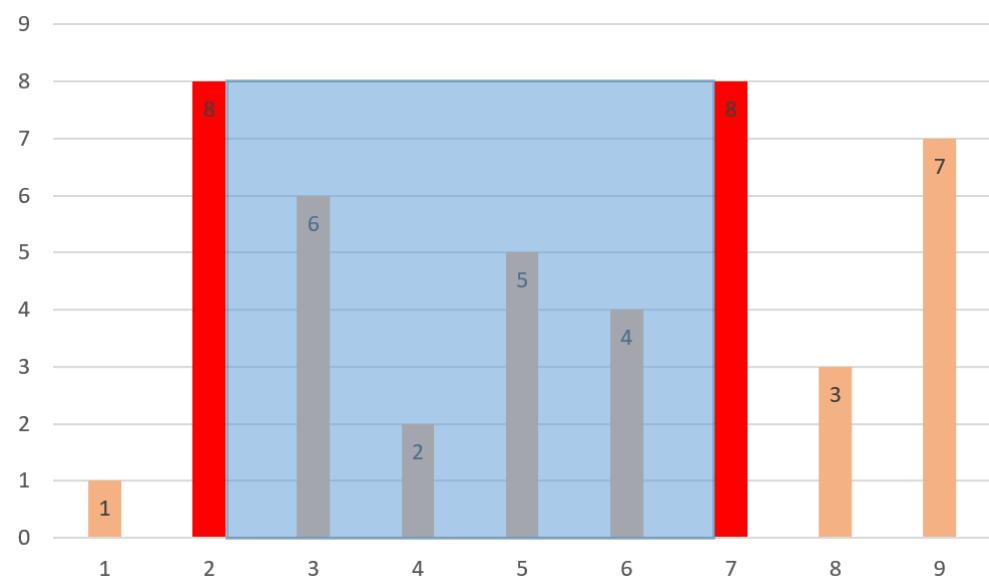
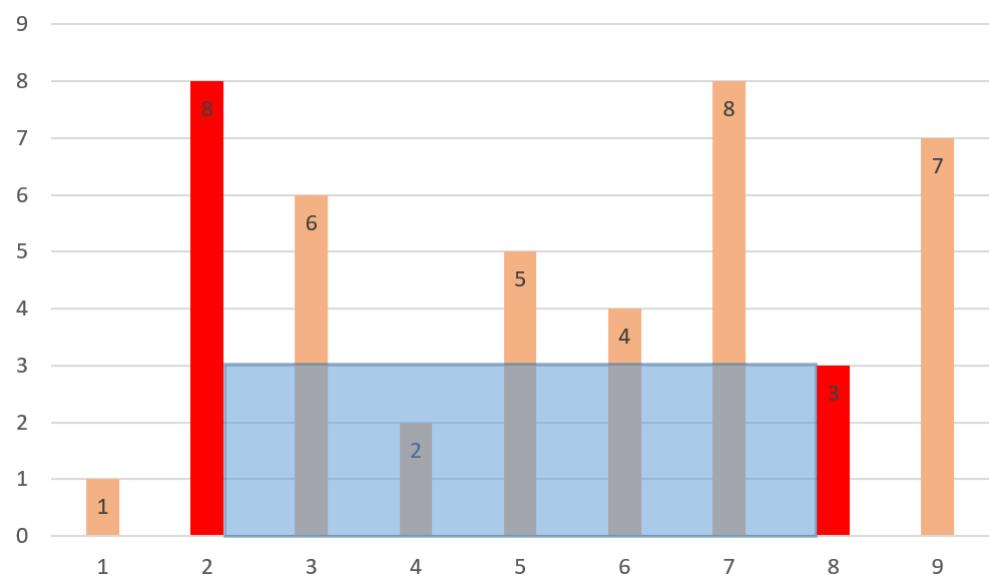
我们尝试将长的一侧向短的一侧移动，发现对于区域面积增加没有任何意义。比如下图：

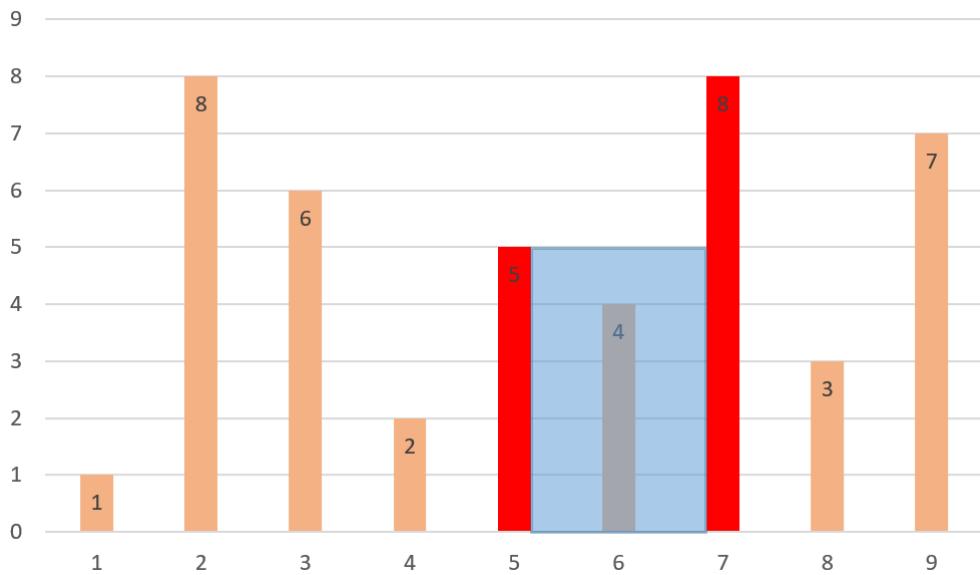
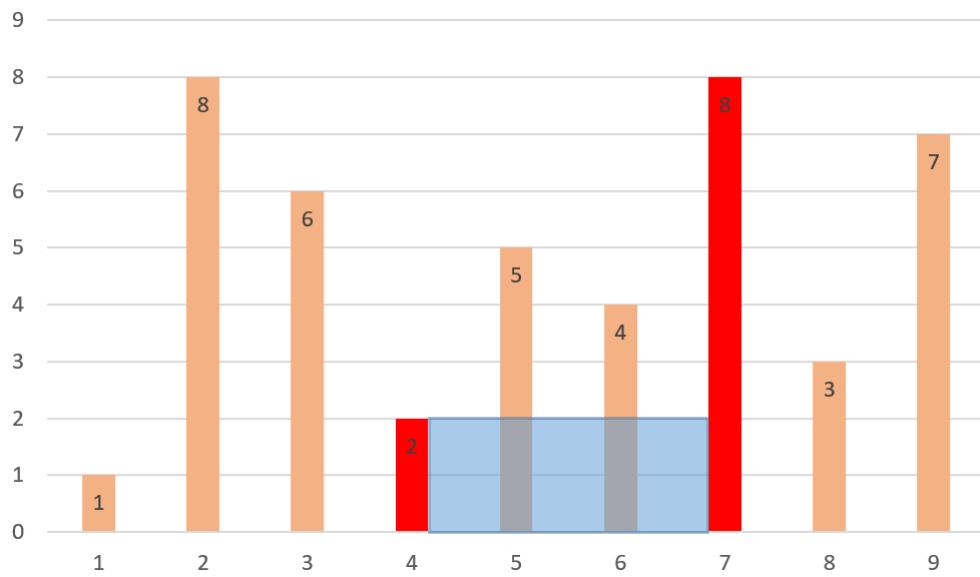


所以我们选择将短的一侧向长的一侧移动。根据木桶原理，水的高度取决于短的一侧。

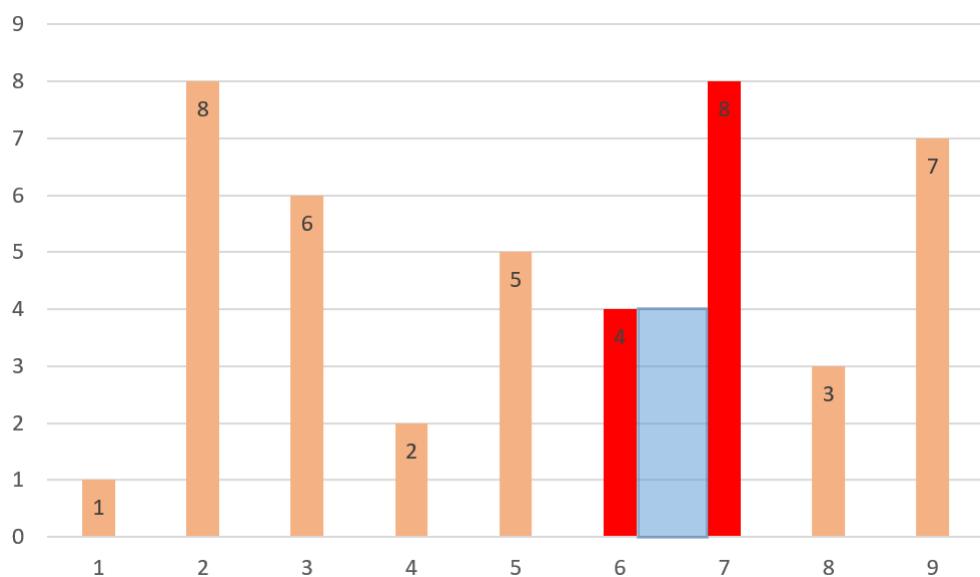


继续重复这个过程，我们总是**选择将短的一侧向长的一侧移动**。并且在每一次的移动中，我们记录下来当前面积大小。（下面这些图，都是我拿PPT一张张做的....）





一直到两个棒子撞在一起。



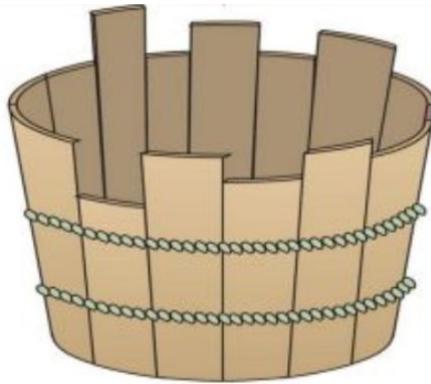
根据分析，得到代码：(翻Java牌子)

```

1 //JAVA
2 class Solution {
3     public int maxArea(int[] height) {
4         int i = 0, j = height.length - 1, res = 0;
5         while(i < j){
6             res = height[i] < height[j] ?
7                 Math.max(res, (j - i) * height[i++]):
8                 Math.max(res, (j - i) * height[j--]);
9         }
10    return res;
11 }
12 }
```

### 03、反证法证明

可能有的朋友想让我证明一下。其实我觉得，这就是个木桶原理。木桶原理：一只水桶能装多少水取决于它最短的那块木板。



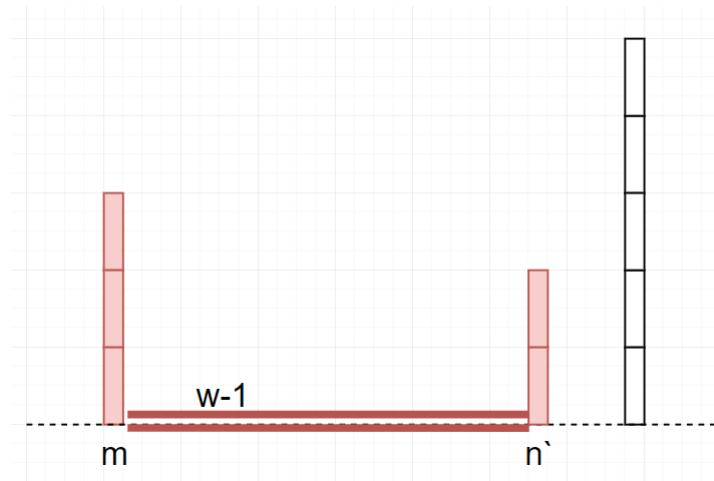
采用反证法进行证明：



移动n到n，如果n比m短，则有：

$$\text{area} = h(n) * (w-1)$$

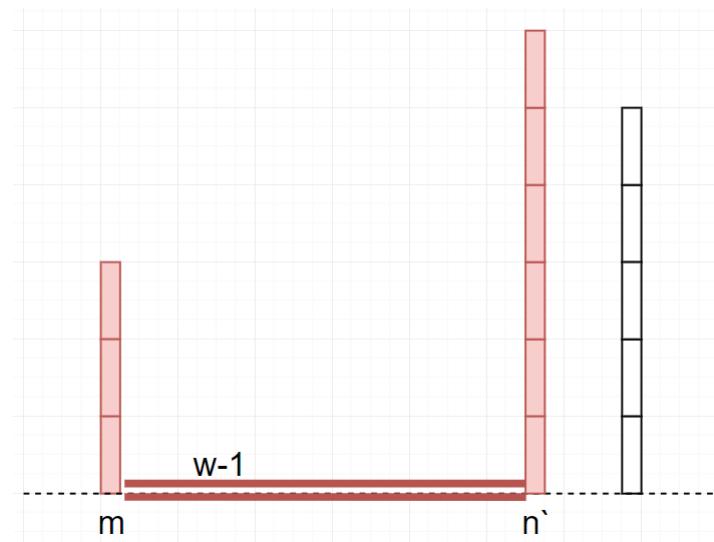
有  $\text{area} < \text{area}$



移动n到n'，如果n比m长，则有：

$$\text{area} = h(m) * (w-1)$$

有  $\text{area} < \text{area}$



所以，今天的问题你学会了吗？评论区留下你的想法！

## 扑克牌中的顺子容器

今天给大家分享一道比较简单的扑克牌题目。

### 01、题目标示例

拿到题目的小伙伴，可能觉得“我次奥”，这特么也能出一道题？不得不说《贱人offer》，嗯....不错！

#### 扑克牌中的顺子

从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大、小王为0，可以看成任意数字。A不能视为14。

### 示例 1:

```
1 | 输入: [1,2,3,4,5]  
2 | 输出: True
```

### 示例 2:

```
1 | 输入: [0,0,1,2,5]  
2 | 输出: True
```

### 限制:

```
1 | 数组长度为 5  
2 | 数组的数取值为 [0, 13]
```

## 02、题目分析

题目就是找一个顺子....啥？你不知道什么是顺子，看下面这个。

顺子长这样：



因为此题本身属于简单到要屎系列，所以直接给题解。数组长度限制了是5，非常省事，意味着我们不需要一些额外的处理。拿到牌，第一个想法是啥？排序！我想打过牌的人，都会知道这点（想象那个插插的过程）。**因为是5连，无论接没接到大小王，最小值和最大值之间，一定小于5。**

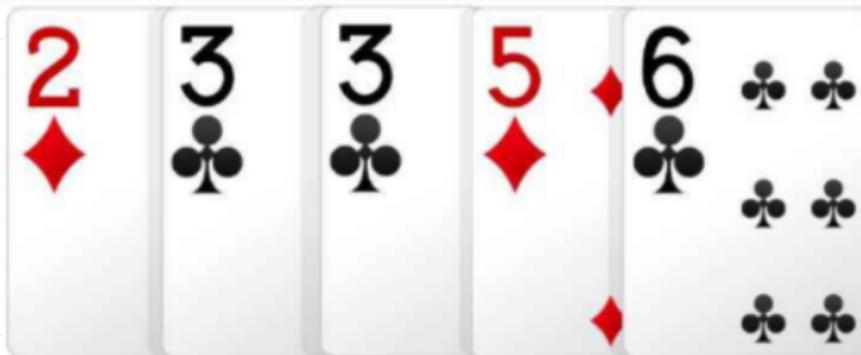


排序后，我们通过累积每两张牌之间的差值，来计算最小值和最大值中间的总差值。



拿到了王，就相当于拿到了通行证，直接跳过。

因为是排序的牌，如果接到对子，也就意味着不是五连，直接返回false。（怎么每次说到五连，我就想到“penta kill”）



根据分析，得出代码：（翻个go的牌子吧）

```
1 //go
2 func isStraight(nums []int) bool {
3     sort.Ints(nums)
4     sub := 0
5     for i := 0; i < 4; i++ {
6         if nums[i] == 0 {
7             continue
8         }
9         if nums[i] == nums[i+1] {
10            return false
11        }
12        sub = nums[i+1] - nums[i]
13    }
14    return sub < 5
15 }
```

执行结果:

执行用时: 0 ms , 在所有 Go 提交中击败了 100.00% 的用户

内存消耗: 2 MB , 在所有 Go 提交中击败了 100.00% 的用户

## 03、不排序咋整

“没吃过猪肉，也见过猪跑”这句话来源于《红楼梦》第十六回，有云：“偏你又怕他不在行了。谁都是在行的？孩子们这么大了，没吃过猪肉，也见过猪跑。”

没吃过猪肉还没见过猪跑么，一模一样的整法！和排序本质上没啥区别，还是通过计算最大值和最小值之间的差值，来判断是否为五连。唯一的区别，是需要记录一些数据。包括：用数组或者map记录下是否有重复牌，记录下最大值和最小值用来做最终差值计算。

直接给代码：（好奇c 有木有人看？）

```
1 //C
2 class Solution {
3     public:
4     bool isStraight(vector<int>& nums) {
5         vector<int> arr(14,0);
6         for(int i = 0; i < nums.size(); i){
7             arr[nums[i]]++;
8         }
9         for(int i = 1;i < 14; i){
10             if(arr[i] > 1)
11                 return false;
12         }
13         int min = 1,max = 13;
14         while(min < 14 && arr[min] == 0) min++;
15         while(max >= 0 && arr[max] == 0) max--;
16         return max - min <= 4;
17     }
18 }
```

执行结果:

执行用时: 0 ms , 在所有 C++ 提交中击败了 100.00% 的用户

内存消耗: 12.3 MB , 在所有 C++ 提交中击败了 100.00% 的用户

## 整数拆分（343）

能跟着看到现在，大家都有点疲惫了。为了提高各位积极性，我打算每天在文首放一张女神的图（不为别的，只为激励大家，毕竟美女对男女都是通杀的。祝大家早日拿到理想offer，实现人生赢家）话不多说，直接看题！

## 01、题目示例

这两天越来越多的读者私信小浩，说觉得只看题的话，不是很系统，想让我系统的讲一讲各类数据结构。对于这个问题，我统一回复一下，首先后面肯定是有系统的讲解各类数据结构的打算的，这个目前正在筹划中，所以大家请放心！另外对于看题，如果担心缺乏基础知识看不懂的朋友们，大家请一万个放心。老读者都知道，我讲题，一般都是会把这个题涉及到的基础知识都给你过一遍的。当然后面我也会用系列篇，把这些题目再串起来，所以大家还是耐心点的去看。记住，干就对了！

### 第343题：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：

1	输入： 2
2	输出： 1
3	解释： $2 = 1 + 1$ ， $1 \times 1 = 1$ 。

#### 示例 2：

1	输入： 10
2	输出： 36
3	解释： $10 = 3 + 4$ ， $3 \times 4 = 36$ 。

**说明：**你可以假设  $n$  不小于 2 且不大于 58。

## 02、题目分析

这个题理解了题意的话，其实还是比较简单的，一起看下。

要对一个整数进行拆分，并且要使这些拆分完后的因子的乘积最大。我们可以先尝试拆分几个数值，测试一下。

首先，拆分成1的因子组合毫无意义，所以我们省略				
n	拆分1	拆分2	拆分3	拆分4
2	1*1			
3	1*2			
4	2*2=4			
5	2*3=6			
6	2*4=8	2*2*2=8	3*3=9	
7	2*5=10	2*2*3=12	3*4=12	3*2*2=12
8	2*6=12	2*2*4=16	2*2*2*2=16	2*3*3=18
	3*5=15	3*2*3=18		
	4*4=16	2*2*2*2=16		

通过观察，首先肯定可以明确，**2 和 3 是没办法进行拆分的最小因子**。同时，我们好像能看出来：

- 只要把 n 尽可能的拆分成包含3的组合，就可以得到最大值。
- 如果没办法拆成 3 的组合，就退一步拆成 2 的组合。
- 对于 3 和 2，没办法再进行拆分。

根据分析，我们尝试使用**贪心**进行求解。因为一个数（假设为n）除以另一个数，总是包括整数部分(x) 和余数部分(y)。那刚才也得到了，**最优因子是3**，所以我们需要让  $n/3$ ，这样的话，余数可能是 1,2 两种可能性。

- 如果余数是 1，刚才我们也分析过，对于 1 的拆分是没有意义的，所以我们退一步，将最后一次的 3 和 1 的拆分，用 2 和 2 代替。
- 如果余数是 2，那不消多说，直接乘以最后的 2 即可。

根据分析，得出代码：

```

1 //JAVA
2 public static int integerBreak(int n) {
3     if (n <= 3) return n - 1;
4     int x = n / 3, y = n % 3;
5     //恰好整除，直接为3^x
6     if (y == 0) return (int) Math.pow(3, x);
7     //余数为1，退一步 3^(x-1)*2*2
8     if (y == 1) return (int) Math.pow(3, x - 1) * 4;
9     //余数为2，直接乘以2
10    return (int) Math.pow(3, x) * 2;
11 }
```

## 03、证明过程

答案是碰出来了，但是我们是通过观察，发现最优因子应该是 3。那如何来证明这个结论的正确性呢？

首先，通过均值不等式，很容易验证当每一个拆分值都相等的时候，才具有最大值，所以实际上就是将这个数均分。那么，对于整数，我们将其分解成份，每一份为则有

求的极值点为，最接近的也就是 3 了。（注意：这里是整数，如果是实数，该证明则有漏洞）

## 04、都看不懂

一力破万法，乱拳打死老师傅，使用万能的动态规划求解。

dp[i]代表 i 拆分之后得到的乘积的最大的元素，比如dp[4]就保存将4拆分后得到的最大的乘积。状态转移方程式为

$$dp[i] = \max(dp[i], (i-j) * \max(dp[j], j))$$

整体思路就是这样，将一个大的问题，分解成一个一个的小问题，然后完成一个**自底向上的**过程。举一个例子，比如计算 10，可以拆分 6 和 4，因为 6 的最大值  $3 \times 3$ ，以及 4 的最大值  $2 \times 2$  都已经得到，所以就替换成 9 和 4，也就是  $10 = 3 \times 3 \times 4$ 。

代码如下：（CPP听说很受欢迎？）

```
1 //C
2 class Solution {
3     public:
4     int integerBreak(int n)
5     {
6         vector<int> dp(n - 1, 0);
7         dp[1] = 1;
8         for (int i = 2; i <= n; i++)
9         {
10             for (int j = 1; j < i; j++)
11             {
12                 dp[i] = max(dp[i], max(dp[j], j) * (i - j));
13             }
14         }
15         return dp[n];
16     }
17 };
```

今天的题目可能有一定难度，建议大家自己写写画画，才能真正的做到理解和巩固。

## 移动石子直到连续 (1033)

今天为大家分享一个**脑筋急转弯**类型的算法题。leetcode这个脑筋急转弯的tag打的我措手不及...

For the minimum: We can always do it in at most 2 moves, by moving one stone next to another, then the third stone next to the other two. When can we do it in 1 move? 0 moves? For the maximum: Every move, the maximum position minus the minimum position must decrease by at least 1.

## 01、题目示例

分享这道题目的原因，是因为有很多同学，在拿到题目的一瞬间，如果发现题目很长，然后自己就慌了。其实，对于这种题目，如果认真的分析下去，非常简单。

### 第1033题：移动石子直到连续

三枚石子放置在数轴上，位置分别为  $a, b, c$ 。每一回合，我们假设这三枚石子当前分别位于位置  $x, y, z$  且  $x < y < z$ 。从位置  $x$  或者是位置  $z$  拿起一枚石子，并将该石子移动到某一整数位置  $k$  处，其中  $x < k < z$  且  $k \neq y$ 。当你无法进行任何移动时，即，这些石子的位置连续时，游戏结束。要使游戏结束，你可以执行的最小和最大移动次数分别是多少？以长度为 2 的数组形式返回答案：

`answer = [minimum_moves, maximum_moves]`

#### 示例1：

- 1 输入:  $a = 1, b = 2, c = 5$
- 2 输出:  $[1, 2]$
- 3 解释: 将石子从 5 移动到 4 再移动到 3，或者我们可以直接将石子移动到 3。

#### 示例 2:

- 1 输入:  $a = 4, b = 3, c = 2$
- 2 输出:  $[0, 0]$
- 3 解释: 我们无法进行任何移动。

#### 提示:

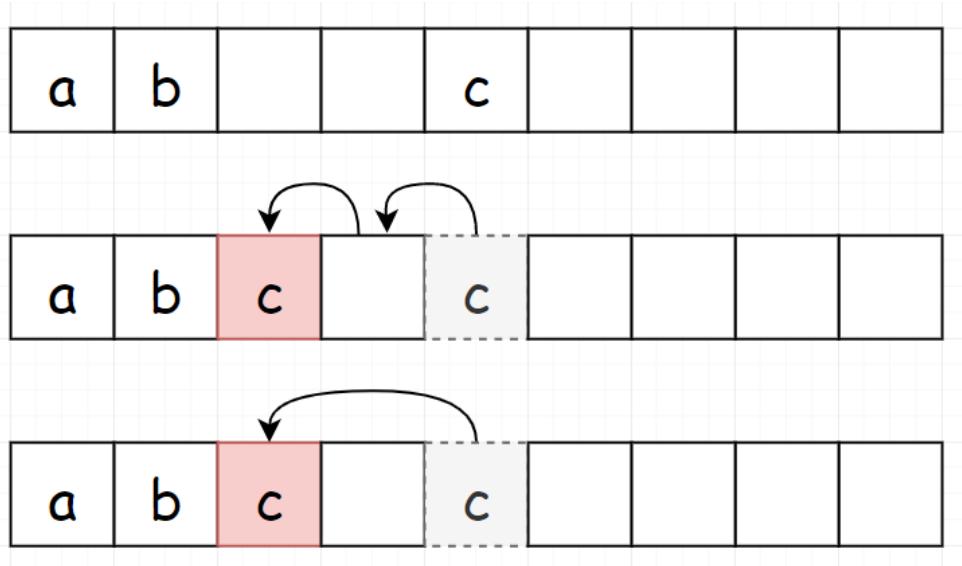
- 1  $1 \leq a \leq 100$
- 2  $1 \leq b \leq 100$
- 3  $1 \leq c \leq 100$
- 4  $a \neq b, b \neq c, c \neq a$

## 02、题目分析

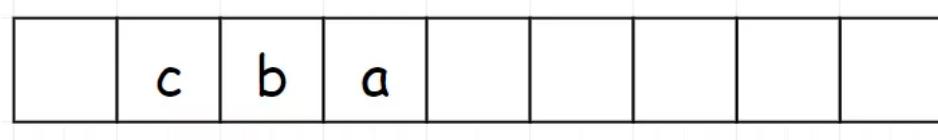
这种题，基本上不慌，就赢了一半。

通过分析题中的样例，就算再笨，画一画应该都能理解题意。

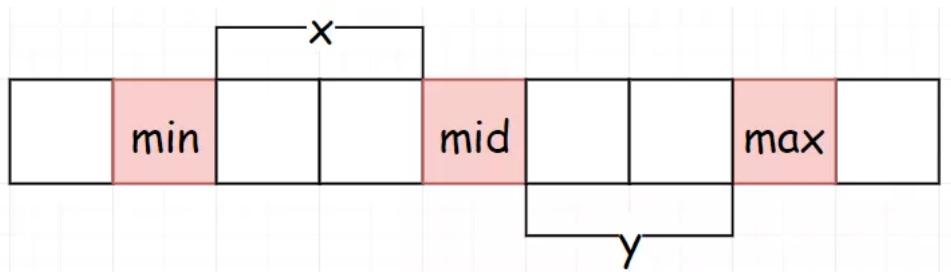
比如： $a = 1, b = 2, c = 5$



比如： $a = 4, b = 3, c = 2$



读懂了题意，开始进行分析。首先可以明确，每一次我们其实是从边上挑石子，然后往中间进行移动。所以，我们首先得找到min (左)，max (右) 以及mid (中) 三个值。我们设，min和mid中的距离为x，max和min中的距离为y。大概就是下面这样：



然后只需要计算x和y的和，就是我们要找的最大值。而最小值，就很容易了，只有0,1,2三种可能性。

根据分析，得到代码：

```
1 func numMovesStones(a int, b int, c int) []int {  
2     arr := []int{a, b, c}  
3     sort.Ints(arr)
```

```

4     x := arr[1] - arr[0] - 1
5     y := arr[2] - arr[1] - 1
6     max := x + y
7     min := 0
8     if x != 0 || y != 0 {
9         if x > 1 && y > 1 {
10            min = 2
11        } else {
12            min = 1
13        }
14    }
15    return []int{min, max}
16 }

```

## 03、C 代码

当然，也可以不用排序，把代码写漂亮一点。像是下面这样...

代码如下：

```

1 class Solution {
2 public:
3     vector<int> numMovesStones(int a, int b, int c) {
4         int max = a > b ? (a > c ? a : c) : (c > b ? c : b);
5         int min = a < b ? (a < c ? a : c) : (b < c ? b : c);
6         int med = a + b + c - max - min;
7         int maxMove = max - min - 2;
8         int minMove = 2;
9         if (max - med == 1 && med - min == 1) {
10             minMove = 0;
11         } else if (max - med == 1 || med - min == 1) {
12             minMove = 1;
13         } else if (max - med == 2 || med - min == 2) {
14             minMove = 1;
15         }
16         return vector{minMove, maxMove};
17     }
18 };

```

执行结果：

执行用时：0 ms，在所有 C++ 提交中击败了 100.00% 的用户

内存消耗：8 MB，在所有 C++ 提交中击败了 100.00% 的用户

## Nim 游戏 (292)

上一篇是为大家分享了一道打着“脑筋急转弯”tag的题目，然后我顺便就把这个类型的题目全部筛选出来看了看，发现总共没几个，所以就想的干脆一次全部讲完吧。

Brainteaser						
#	题名	题解	通过率	难度	出现频率	?
✓ 292	Nim 游戏	146	69.7%	简单	🔒	
✓ 1033	移动石子直到连续	48	37.3%	简单	🔒	
✓ 1227	飞机座位分配概率	42	63.7%	中等	🔒	
✓ 319	灯泡开关	38	45.0%	中等	🔒	
✓ 777	在LR字符串中交换相邻字符	16	31.9%	中等	🔒	

## 01、题目示例

这个类型的题目，其实除了废话多一点，好像没什么特别的。

### 第292题：Nim 游戏

你和你的朋友，两个人一起玩 Nim 游戏：桌子上有一堆石头，每次你们轮流拿掉 1 - 3 块石头。拿掉最后一块石头的人就是获胜者。你作为先手。你们是聪明人，每一步都是最优解。编写一个函数，来判断你是否可以在给定石头数量的情况下赢得游戏。

示例：

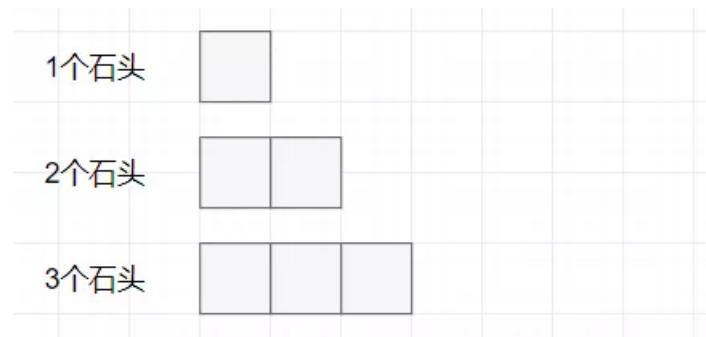
- 1 输入： 4
- 2 输出： `false`
- 3 解释： 如果堆中有 4 块石头，那么你永远不会赢得比赛；
- 4 因为无论你拿走 1 块、2 块 还是 3 块石头，最后一块石头总是会被你的朋友拿走。

PS：建议大家停留个两分钟先想一想...直接拉下去看题解就没什么意思了。

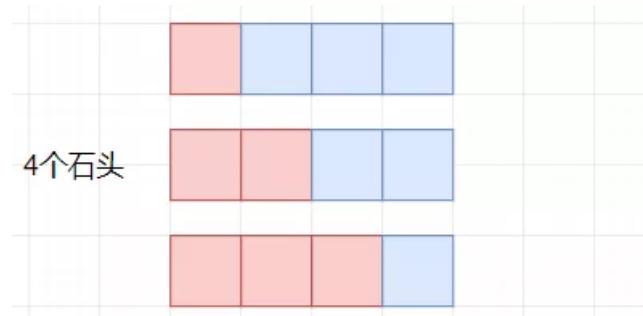
## 02、题目分析

这种问题，如果没有思路，可以先自己找个纸写写画画，找找规律

首先如果石头数小于4个，那么因为你是先手，一把拿走，肯定会赢。



而如果石头是4个，那不管你是拿了1,2,3个，最后一个都可以被你的对手拿走，所以怎么样都赢不了。



再继续分析到8个石头：对于5,6,7而言，你只需要对应的拿走1,2,3，然后留下4个，则对方必输。但是如果你要面对的是8，不管先拿（1,2,3）个，另一个人都可以通过  $8-(1,2,3)$ ，使得你面对4个石头，则你必输无疑。通过观察，我们发现，好像是只要N是4的倍数，我们就必输无疑。

石头数N	是否能赢
1	TRUE
2	TRUE
3	TRUE
4	FALSE
5	TRUE
6	TRUE
7	TRUE
8	FALSE

尝试性的写下代码：（这个，什么语言都无所谓吧....）

```

1 //go
2 func canWinNim(n int) bool {
3     return n % 4 != 0
4 }
```

执行结果：

执行用时：0 ms，在所有 Go 提交中击败了 100.00% 的用户

内存消耗：1.9 MB，在所有 Go 提交中击败了 100.00% 的用户

### 03、证明过程

脑筋急转弯的题目不是很多见，但是某些公司的某些人却钟情于此，如果是本着考察对方的思维能力，那我觉得还是挺好的。但若是为了寻找作为面试官的一丝丝优越感，那就只能是。。。呵，打扰了。。

首先需要说下的是，这个问题属于**博奕论**。NIM的意思就是“尼姆”，并不是什么高大上的英文缩写。所以，NIM游戏一般也称之为尼姆游戏。说白了，就是**设置两个对手，通过回合制的方式来玩的一种数学战略游戏**，在早期网络不发达的时候很火。毕竟那时候连梦幻都没有，更别说王者。（非戏说，很多回合制游戏，其实本质就是数学游戏。而对于王者这种多人实时竞技游戏更是如此，如果想玩好，数学学不好，基本就凉凉。有兴趣的，可以了解一下**游戏平衡师**）

回到本题，假若对于先手有N个石头，那么后手的可能性有N-1, N-2, N-3三种。**只有当后手的这三种可能性都必胜时，N才会必败。**因为题目说了，我们都是聪明人（一般博奕论的问题都会有这句话），那如果后手的三种可能性中，有哪一种必败，作为先手，我们一定会走出这种可能性。那这种可能性是什么，其实就是让对方去面对4的倍数。如果先手我们遇到一个不是4的倍数的值x，有：

$$4k > N > 4(k-1)$$

N一定处于两个4的倍数之间，因为N本身不是4的倍数，那N距离最近的4的倍数的值最大为3。所以，只要我们不是面对4的倍数，作为先手，我们一定可以取走(1,2,3)，使剩余的值变成4的倍数，则后手必输无疑。

所以，今天的问题你学会了吗？评论区留下你的想法！

## 后续

希望这本汇总的电子书对你有所帮助，因为是第一版，难免有所纰漏。例如错别字，语义混淆等，希望大家可以谅解！当然，大家遇到任何问题，都可以反馈给我。不管是通过公众号内留言的方式，还是加我个人微信：



小浩



中国大陆



扫一扫上面的二维码图案，加我微信

本书内容全部原创（包括图片），并首发于 **小浩算法** 公众号内，所有版权归作者所有！侵权必究！  
严禁任何单位和个人以商业为目的进行任何形式的复制！