labuladong 的算法小抄

Table of Contents

关于本小抄和作者

这份算法小抄整理自公众号 **labuladong** 的文章，旨在帮助公众号读者整理算法套路，助力面试，禁止商用！

本人于 2019 年 5 月开通公众号 labuladong，一直坚持原创算法文章，至今已经积累了几万读者。欢迎关注我的公众号 labuladong 交流，方便获得最新的优质文章：

目前作者 labuladong 的主要发文平台是微信公众号，很建议关注一下，公众号后台可以回复关键词【电子书】获得这份算法小抄的各种格式电子书版本。网页端可以访问作者的 Gitbook 查看所有文章：https://labuladong.gitbook.io/algo

PS：**另外建议读者在公众号选择 epub 格式的电子书下载，因为 pdf 格式无法显示部分文章的 GIF 动态图片**。

我讲解算法问题的风格是**结构化，模板化，力求辅助读者培养框架思维，举一反三**，相信大家能从这份算法小抄中有所收获！做成电子版是方便读者做笔记，我已经纠正了绝大多数格式和图片引用错误，就是 PDF 图片较多，所以体积较大，请担待。如果 Gitbook 或者电子书中还有图片加载之类的问题，可以在公众号后台具体发给我，我会修复。

动态规划系列

我们公众号最火的就是动态规划系列的文章，也许是动态规划问题有难度而且有意思，也许因为它是面试常考题型。不管你之前是否害怕动态规划系列的问题，相信这一章的内容足以帮助你消除对动态规划算法的恐惧。

具体来说，动态规划的一般流程就是三步：**暴力的递归解法 -> 带备忘录的递归解法 -> 迭代的动态规划解法**。

就思考流程来说，就分为一下几步：**找到状态和选择 -> 明确 dp 数组/函数的定义 -> 寻找状态之间的关系**。

这就是思维模式的框架，**本章都会按照以上的模式来解决问题，辅助读者养成这种模式思维**，有了方向遇到问题就不会抓瞎，足以解决一般的动态规划问题。

欢迎关注我的公众号 labuladong，方便获得最新的优质文章：

动态规划详解

这篇文章是我们号半年前一篇 200 多赞赏的成名之作「动态规划详解」的进阶版。由于账号迁移的原因，旧文无法被搜索到，所以我润色了本文，并添加了更多干货内容，希望本文成为解决动态规划的一部「指导方针」。

再说句题外话，我们的公众号开号至今写了起码十几篇文章拆解动态规划问题，我都整理到了公众号菜单的「文章目录」中，**它们都提到了动态规划的解题框架思维，本文就系统总结一下**。这段时间本人也从非科班小白成长到刷通半个 LeetCode，所以我总结的套路可能不适合各路大神，但是应该适合大众，毕竟我自己也是一路摸爬滚打过来的。

算法技巧就那几个套路，如果你心里有数，就会轻松很多，本文就来扒一扒动态规划的裤子，形成一套解决这类问题的思维框架。废话不多说了，上干货。

动态规划问题的一般形式就是求最值。动态规划其实是运筹学的一种最优化方法，只不过在计算机问题上应用比较多，比如说让你求**最长**递增子序列呀，**最小**编辑距离呀等等。

既然是要求最值，核心问题是什么呢？**求解动态规划的核心问题是穷举**。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。

动态规划就这么简单，就是穷举就完事了？我看到的动态规划问题都很难啊！

首先，动态规划的穷举有点特别，因为这类问题**存在「重叠子问题」**，如果暴力穷举的话效率会极其低下，所以需要「备忘录」或者「DP table」来优化穷举过程，避免不必要的计算。

而且，动态规划问题一定会**具备「最优子结构」**，才能通过子问题的最值得到原问题的最值。

另外，虽然动态规划的核心思想就是穷举求最值，但是问题可以千变万化，穷举所有可行解其实并不是一件容易的事，只有列出**正确的「状态转移方程」**才能正确地穷举。

以上提到的重叠子问题、最优子结构、状态转移方程就是动态规划三要素。具体什么意思等会会举例详解，但是在实际的算法问题中，**写出状态转移方程是最困难的**，这也就是为什么很多朋友觉得动态规划问题困难的原因，我来提供我研究出来的一个思维框架，辅助你思考状态转移方程：

明确「状态」 -> 定义 dp 数组/函数的含义 -> 明确「选择」 -> 明确 base case。

下面通过斐波那契数列问题和凑零钱问题来详解动态规划的基本原理。前者主要是让你明白什么是重叠子问题（斐波那契数列严格来说不是动态规划问题），后者主要举集中于如何列出状态转移方程。

请读者不要嫌弃这个例子简单，**只有简单的例子才能让你把精力充分集中在算法背后的通用思想和技巧上，而不会被那些隐晦的细节问题搞的莫名其妙**。想要困难的例子，历史文章里有的是。

一、斐波那契数列

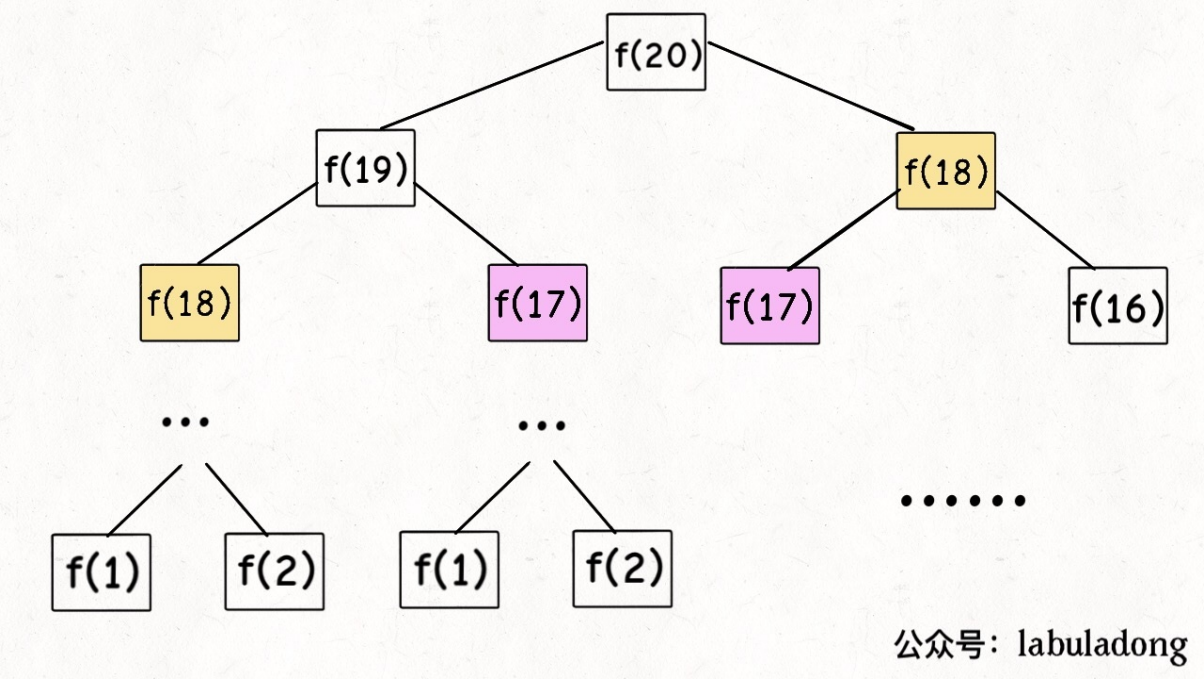
1、暴力递归

斐波那契数列的数学形式就是递归的，写成代码就是这样：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

这个不用多说了，学校老师讲递归的时候似乎都是拿这个举例。我们也知道这样写代码虽然简洁易懂，但是十分低效，低效在哪里？假设 n = 20，请画出递归树。

PS：但凡遇到需要递归的问题，最好都画出递归树，这对你分析算法的复杂度，寻找算法低效的原因都有巨大帮助。



这个递归树怎么理解？就是说想要计算原问题 `f(20)`，我就得先计算出子问题 `f(19)` 和 `f(18)`，然后要计算 `f(19)`，我就要先算出子问题 `f(18)` 和 `f(17)`，以此类推。最后遇到 `f(1)` 或者 `f(2)` 的时候，结果已知，就能直接返回结果，递归树不再向下生长了。

递归算法的时间复杂度怎么计算？子问题个数乘以解决一个子问题需要的时间。

子问题个数，即递归树中节点的总数。显然二叉树节点总数为指数级别，所以子问题个数为 O(2^n)。

解决一个子问题的时间，在本算法中，没有循环，只有 f(n - 1) + f(n - 2) 一个加法操作，时间为 O(1)。

所以，这个算法的时间复杂度为 O(2^n)，指数级别，爆炸。

观察递归树，很明显发现了算法低效的原因：存在大量重复计算，比如 `f(18)` 被计算了两次，而且你可以看到，以 `f(18)` 为根的这个递归树体量巨大，多算一遍，会耗费巨大的时间。更何况，还不止 `f(18)` 这一个节点被重复计算，所以这个算法及其低效。

这就是动态规划问题的第一个性质：**重叠子问题**。下面，我们想办法解决这个问题。

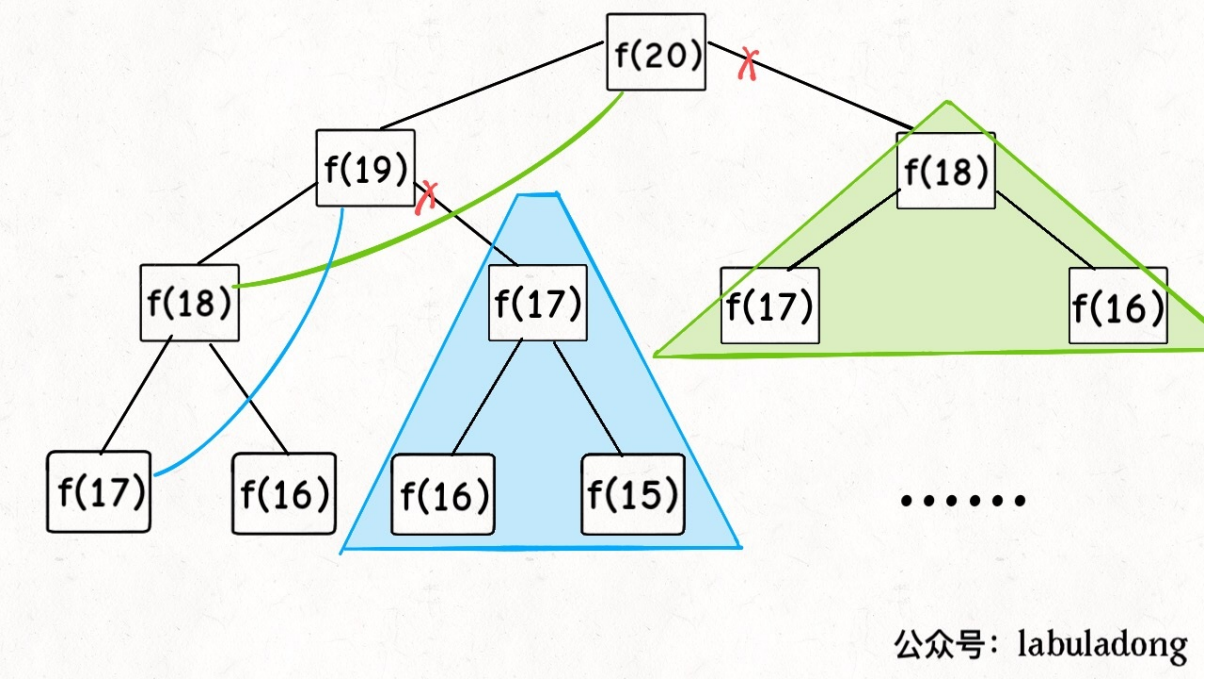
2、带备忘录的递归解法

明确了问题，其实就已经把问题解决了一半。即然耗时的原因是重复计算，那么我们可以造一个「备忘录」，每次算出某个子问题的答案后别急着返回，先记到「备忘录」里再返回；每次遇到一个子问题先去「备忘录」里查一查，如果发现之前已经解决过这个问题了，直接把答案拿出来用，不要再耗时去计算了。

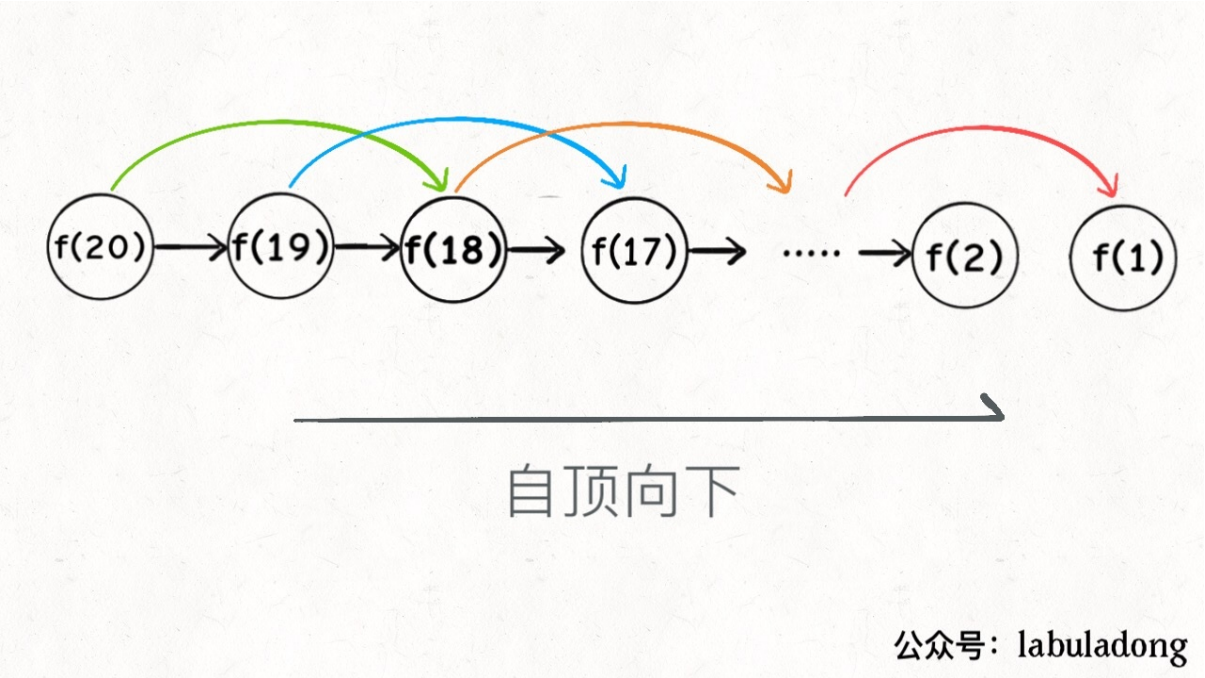
一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

```
int fib(int N) {
    if (N < 1) return 0;
    // 备忘录全初始化为 0
    vector<int> memo(N + 1, 0);
    // 初始化最简情况
    return helper(memo, N);
}

int helper(vector<int>& memo, int n) {
    // base case
    if (n == 1 || n == 2) return 1;
    // 已经计算过
    if (memo[n] != 0) return memo[n];
    memo[n] = helper(memo, n - 1) +
              helper(memo, n - 2);
    return memo[n];
}
```

现在，画出递归树，你就知道「备忘录」到底做了什么。



实际上，带「备忘录」的递归算法，把一棵存在巨量冗余的递归树通过「剪枝」，改造成了一幅不存在冗余的递归图，极大减少了子问题（即递归图中节点）的个数。



递归算法的时间复杂度怎么算？子问题个数乘以解决一个子问题需要的时间。

子问题个数，即图中节点的总数，由于本算法不存在冗余计算，子问题就是 `f(1)`，`f(2)`，`f(3)` ... `f(20)`，数量和输入规模 n = 20 成正比，所以子问题个数为 O(n)。

解决一个子问题的时间，同上，没有什么循环，时间为 O(1)。

所以，本算法的时间复杂度是 O(n)。比起暴力算法，是降维打击。

至此，带备忘录的递归解法的效率已经和迭代的动态规划解法一样了。实际上，这种解法和迭代的动态规划已经差不多了，只不过这种方法叫做「自顶向下」，动态规划叫做「自底向上」。

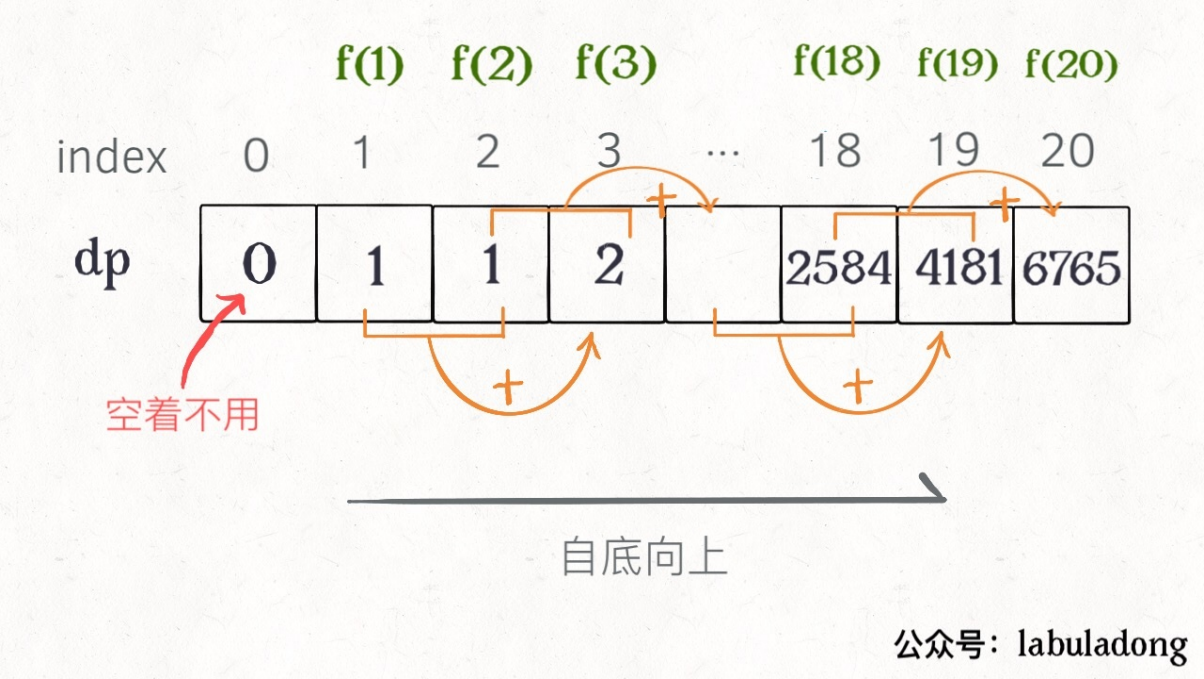
啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说 `f(20)`，向下逐渐分解规模，直到 `f(1)` 和 `f(2)` 触底，然后逐层返回答案，这就叫「自顶向下」。

啥叫「自底向上」？反过来，我们直接从最底下，最简单，问题规模最小的 `f(1)` 和 `f(2)` 开始往上推，直到推到我们想要的答案 `f(20)`，这就是动态规划的思路，这也是为什么动态规划一般都脱离了递归，而是由循环迭代完成计算。

3、dp 数组的迭代解法

有了上一步「备忘录」的启发，我们可以把这个「备忘录」独立出来成为一张表，就叫做 DP table 吧，在这张表上完成「自底向上」的推算岂不美哉！

```
int fib(int N) {
    vector<int> dp(N + 1, 0);
    // base case
    dp[1] = dp[2] = 1;
    for (int i = 3; i <= N; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[N];
}
```



画个图就很好理解了，而且你发现这个 DP table 特别像之前那个「剪枝」后的结果，只是反过来算而已。实际上，带备忘录的递归解法中的「备忘录」，最终完成后就是这个 DP table，所以说这两种解法其实是差不多的，大部分情况下，效率也基本相同。

这里，引出「状态转移方程」这个名词，实际上就是描述问题结构的数学形式：

$$f(n) = \begin{cases} 1, n = 1, 2 \\ f(n-1) + f(n-2), n > 2 \end{cases}$$

为啥叫「状态转移方程」？为了听起来高端。你把 f(n) 想做一个状态 n，这个状态 n 是由状态 n - 1 和状态 n - 2 相加转移而来，这就叫状态转移，仅此而已。

你会发现，上面的几种解法中的所有操作，例如 return f(n - 1) + f(n - 2)，dp[i] = dp[i - 1] + dp[i - 2]，以及对备忘录或 DP table 的初始化操作，都是围绕这个方程式的不同表现形式。可见列出「状态转移方程」的重要性，它是解决问题的核心。很容易发现，其实状态转移方程直接代表着暴力解法。

千万不要看不起暴力解，动态规划问题最困难的就是写出状态转移方程，即这个暴力解。优化方法无非是用备忘录或者 DP table，再无奥妙可言。

这个例子的最后，讲一个细节优化。细心的读者会发现，根据斐波那契数列的状态转移方程，当前状态只和之前的两个状态有关，其实并不需要那么长的一个 DP table 来存储所有的状态，只要想办法存储之前的两个状态就行了。所以，可以进一步优化，把空间复杂度降为 O(1)：

```
int fib(int n) {
    if (n == 2 || n == 1)
        return 1;
    int prev = 1, curr = 1;
    for (int i = 3; i <= n; i++) {
        int sum = prev + curr;
        prev = curr;
        curr = sum;
    }
    return curr;
}
```

有人会问，动态规划的另一个重要特性「最优子结构」，怎么没有涉及？下面会涉及。斐波那契数列的例子严格来说不算动态规划，因为没有涉及求最值，以上旨在演示算法设计螺旋上升的过程。下面，看第二个例子，凑零钱问题。

二、凑零钱问题

先看下题目：给你 `k` 种面值的硬币，面值分别为 `c1, c2 ... ck`，每种硬币的数量无限，再给一个总金额 `amount`，问你**最少**需要几枚硬币凑出这个金额，如果不可能凑出，算法返回 -1 。算法的函数签名如下：

```
// coins 中是可选硬币面值，amount 是目标金额
int coinChange(int[] coins, int amount);
```

比如说 `k = 3`，面值分别为 1，2，5，总金额 `amount = 11`。那么最少需要 3 枚硬币凑出，即 11 = 5 + 5 + 1。

你认为计算机应该如何解决这个问题？显然，就是把所有肯能的凑硬币方法都穷举出来，然后找找看最少需要多少枚硬币。

1、暴力递归

首先，这个问题是动态规划问题，因为它具有「最优子结构」的。**要符合「最优子结构」，子问题间必须互相独立**。啥叫相互独立？你肯定不想看数学证明，我用一个直观的例子来讲解。

比如说，你的原问题是考出最高的总成绩，那么你的子问题就是要把语文考到最高，数学考到最高…… 为了每门课考到最高，你要把每门课相应的选择题分数拿到最高，填空题分数拿到最高…… 当然，最终就是你每门课都是满分，这就是最高的总成绩。

得到了正确的结果：最高的总成绩就是总分。因为这个过程符合最优子结构，“每门科目考到最高”这些子问题是互相独立，互不干扰的。

但是，如果加一个条件：你的语文成绩和数学成绩会互相制约，此消彼长。这样的话，显然你能考到的最高总成绩就达不到总分了，按刚才那个思路就会得到错误的结果。因为子问题并不独立，语文数学成绩无法同时最优，所以最优子结构被破坏。

回到凑零钱问题，为什么说它符合最优子结构呢？比如你想求 `amount = 11` 时的最少硬币数（原问题），如果你知道凑出 `amount = 10` 的最少硬币数（子问题），你只需要把子问题的答案加一（再选一枚面值为 1 的硬币）就是原问题的答案，因为硬币的数量是没有限制的，子问题之间没有相互制，是互相独立的。

那么，既然知道了这是个动态规划问题，就要思考**如何列出正确的状态转移方程**？

先确定「状态」，也就是原问题和子问题中变化的变量。由于硬币数量无限，所以唯一的状态就是目标金额 `amount`。

然后确定 `dp` 函数的定义：当前的目标金额是 `n`，至少需要 `dp(n)` 个硬币凑出该金额。

然后确定「选择」并择优，也就是对于每个状态，可以做出什么选择改变当前状态。具体到这个问题，无论当的目标金额是多少，选择就是从面额列表 `coins` 中选择一个硬币，然后目标金额就会减少：

```
# 伪码框架
def coinChange(coins: List[int], amount: int):
    # 定义：要凑出金额 n，至少要 dp(n) 个硬币
    def dp(n):
        # 做选择，选择需要硬币最少的那个结果
        for coin in coins:
            res = min(res, 1 + dp(n - coin))
        return res
    # 我们要求的问题是 dp(amount)
    return dp(amount)
```

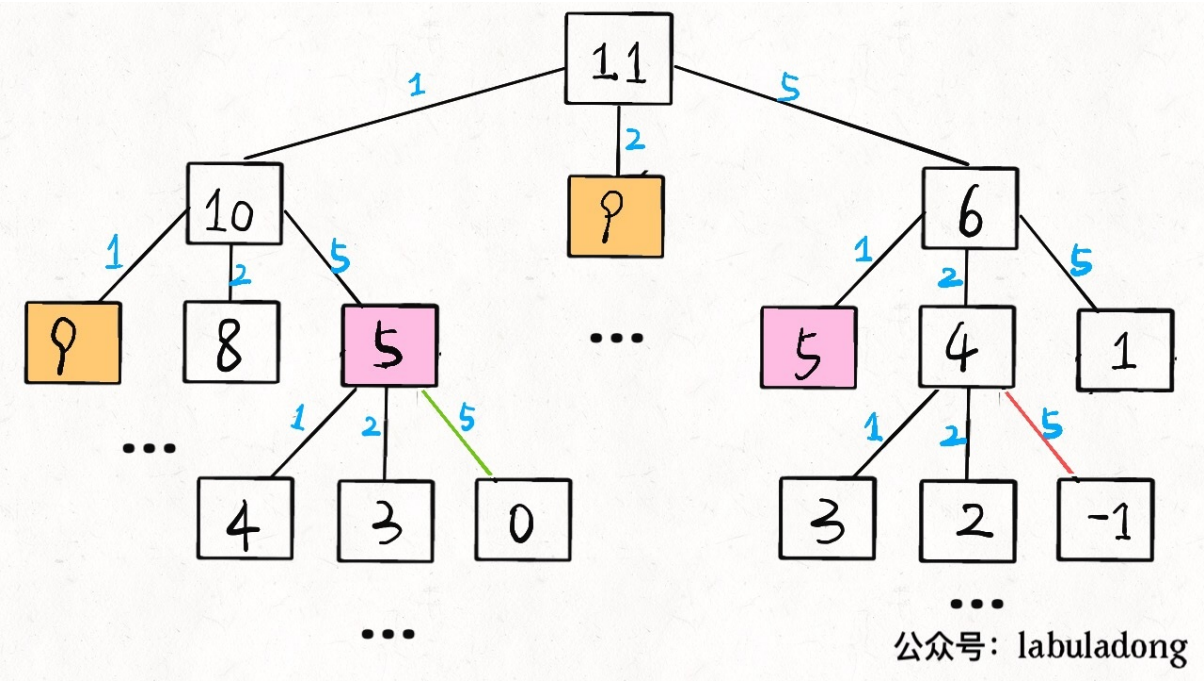
最后明确 base case，显然目标金额为 0 时，所需硬币数量为 0；当目标金额小于 0 时，无解，返回 -1：

```
def coinChange(coins: List[int], amount: int):

    def dp(n):
        # base case
        if n == 0: return 0
        if n < 0: return -1
        # 求最小值，所以初始化为正无穷
        res = float('INF')
        for coin in coins:
            subproblem = dp(n - coin)
            # 子问题无解，跳过
            if subproblem == -1: continue
            res = min(res, 1 + subproblem)

        return res if res != float('INF') else -1

    return dp(amount)
```

至此，状态转移方程其实已经完成了，以上算法已经是暴力解法了，以上代码的数学形式就是状态转移方程：

$$dp(n) = \begin{cases} 0, n = 0 \\ -1, n < 0 \\ min\{dp(n - coin) + 1 | coin \in coins\}, n > 0 \end{cases}$$

至此，这个问题其实就解决了，只不过需要消除一下重叠子问题，比如 `amount = 11, coins = {1,2,5}` 时画出递归树看看：



时间复杂度分析：子问题总数 x 每个子问题的时间。

子问题总数为递归树节点个数，这个比较难看出来，是 O(n^k)，总之是指数级别的。每个子问题中含有一个 for 循环，复杂度为 O(k)。所以总时间复杂度为 O(k * n^k)，指数级别。

2、带备忘录的递归

只需要稍加修改，就可以通过备忘录消除子问题：

```
def coinChange(coins: List[int], amount: int):
    # 备忘录
    memo = dict()
    def dp(n):
        # 查备忘录，避免重复计算
        if n in memo: return memo[n]

        if n == 0: return 0
```

```
        if n < 0: return -1
        res = float('INF')
        for coin in coins:
            subproblem = dp(n - coin)
            if subproblem == -1: continue
            res = min(res, 1 + subproblem)

        # 记入备忘录
        memo[n] = res if res != float('INF') else -1
        return memo[n]

    return dp(amount)
```

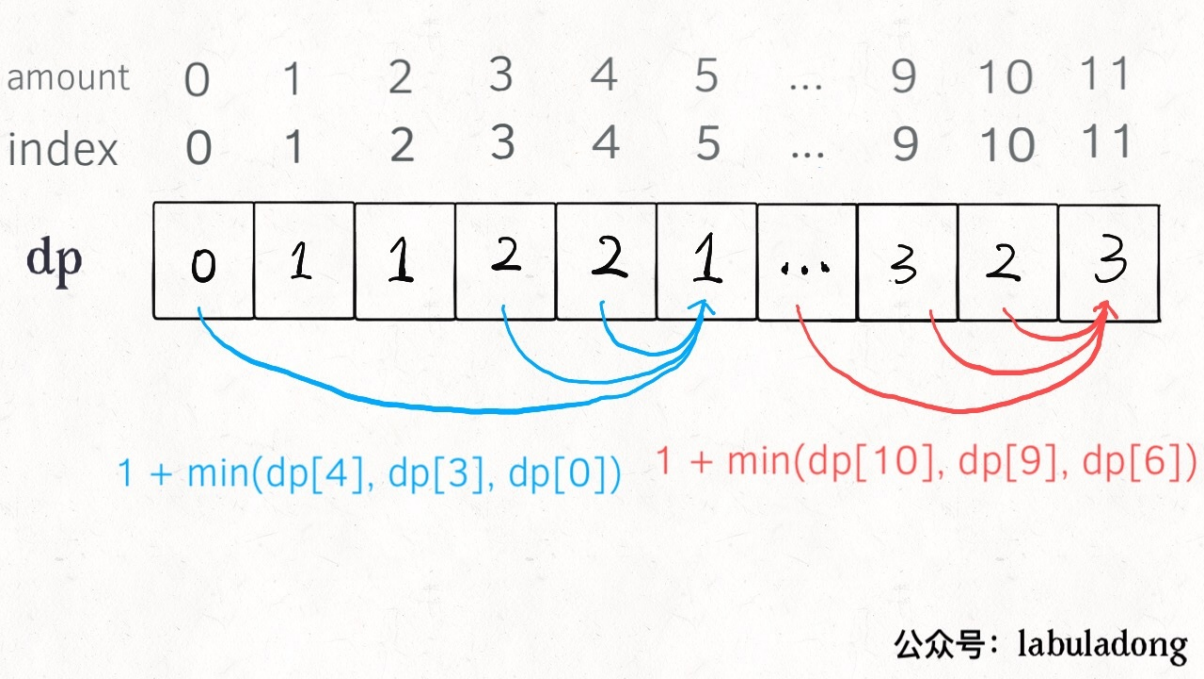
不画图了，很显然「备忘录」大大减小了子问题数目，完全消除了子问题的冗余，所以子问题总数不会超过金额数 n，即子问题数目为 O(n)。处理一个子问题的时间不变，仍是 O(k)，所以总的时间复杂度是 O(kn)。

3、dp 数组的迭代解法

当然，我们也可以自底向上使用 dp table 来消除重叠子问题，`dp` 数组的定义和刚才 `dp` 函数类似，定义也是一样的：

`dp[i] = x` 表示，当目标金额为 `i` 时，至少需要 `x` 枚硬币。

```
int coinChange(vector<int>& coins, int amount) {
    // 数组大小为 amount + 1，初始值也为 amount + 1
    vector<int> dp(amount + 1, amount + 1);
    // base case
    dp[0] = 0;
    for (int i = 0; i < dp.size(); i++) {
        // 内层 for 在求所有子问题 + 1 的最小值
        for (int coin : coins) {
            // 子问题无解，跳过
            if (i - coin < 0) continue;
            dp[i] = min(dp[i], 1 + dp[i - coin]);
        }
    }
    return (dp[amount] == amount + 1) ? -1 : dp[amount];
}
```



PS：为啥 `dp` 数组初始化为 `amount + 1` 呢，因为凑成 `amount` 金额的硬币数最多只可能等于 `amount`（全用 1 元面值的硬币），所以初始化为 `amount + 1` 就相当于初始化为正无穷，便于后续取最小值。

三、最后总结

第一个斐波那契数列的问题，解释了如何通过「备忘录」或者「dp table」的方法来优化递归树，并且明确了这两种方法本质上是一样的，只是自顶向下和自底向上的不同而已。

第二个凑零钱的问题，展示了如何流程化确定「状态转移方程」，只要通过状态转移方程写出暴力递归解，剩下的也就是优化递归树，消除重叠子问题而已。

如果你不太了解动态规划，还能看到这里，真得给你鼓掌，相信你已经掌握了这个算法的设计技巧。

计算机解决问题其实没有任何奇技淫巧，它唯一的解决办法就是穷举，穷举所有可能性。算法设计无非就是先思考“如何穷举”，然后再追求“如何聪明地穷举”。

列出动态转移方程，就是在解决“如何穷举”的问题。之所以说它难，一是因为很多穷举需要递归实现，二是因为有的问题本身的解空间复杂，不那么容易穷举完整。

备忘录、DP table 就是在追求“如何聪明地穷举”。用空间换时间的思路，是降低时间复杂度的不二法门，除此之外，试问，还能玩出啥花活？

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

动态规划答疑篇

这篇文章就给你讲明白两个问题：

1、到底什么才叫「最优子结构」，和动态规划什么关系。

2、为什么动态规划遍历 `dp` 数组的方式五花八门，有的正着遍历，有的倒着遍历，有的斜着遍历。

一、最优子结构详解

「最优子结构」是某些问题的一种特定性质，并不是动态规划问题专有的。也就是说，很多问题其实都具有最优子结构，只是其中大部分不具有重叠子问题，所以我们不把它们归为动态规划系列问题而已。

我先举个很容易理解的例子：假设你们学校有 10 个班，你已经计算出了每个班的最高考试成绩。那么现在我要求你计算全校最高的成绩，你会不会算？当然会，而且你不用重新遍历全校学生的分数进行比较，而是只要在这 10 个最高成绩中取最大的就是全校的最高成绩。

我给你提出的这个问题就**符合最优子结构**：可以从子问题的最优结果推出更大规模问题的最优结果。让你算**每个班**的最优成绩就是子问题，你知道所有子问题的答案后，就可以借此推出**全校**学生的最优成绩这个规模更大的问题的答案。

你看，这么简单的问题都有最优子结构性质，只是因为显然没有重叠子问题，所以我们简单地求最值肯定用不出动态规划。

再举个例子：假设你们学校有 10 个班，你已知每个班的最大分数差（最高分和最低分的差值）。那么现在我让你计算全校学生中的最大分数差，你会不会算？可以想办法算，但是肯定不能通过已知的这 10 个班的最大分数差推到出来。因为这 10 个班的最大分数差不一定就包含全校学生的最大分数差，比如全校的最大分数差可能是 3 班的最高分和 6 班的最低分之差。

这次我给你提出的问题就**不符合最优子结构**，因为你没办通过每个班的最优值推出全校的最优值，没办法通过子问题的最优值推出规模更大的问题的最优值。前文「动态规划详解」说过，想满足最优子结，子问题之间必须互相独立。全校的最大分数差可能出现在两个班之间，显然子问题不独立，所以这个问题本身不符合最优子结构。

那么遇到这种最优子结构失效情况，怎么办？策略是：改造问题。对于最大分数差这个问题，我们不是没办法利用已知的每个班的分数差吗，那我只能这样写一段暴力代码：

```
int result = 0;
for (Student a : school) {
    for (Student b : school) {
        if (a is b) continue;
        result = max(result, |a.score - b.score|);
    }
}
return result;
```

改造问题，也就是把问题等价转化：最大分数差，不就等价于最高分数和最低分数的差么，那不就是要求最高和最低分数么，不就是我们讨论的第一个问题么，不就具有最优子结构了么？那现在改变思路，借助最优子结构解决最值问题，再回过头解决最大分数差问题，是不是就高效多了？

当然，上面这个例子太简单了，不过请读者回顾一下，我们做动态规划问题，是不是一直在求各种最值，本质跟我们举的例子没啥区别，无非需要处理一下重叠子问题。

前文「不同定义不同解法」和「高楼扔鸡蛋进阶」就展示了如何改造问题，不同的最优子结构，可能导致不同的解法和效率。

再举个常见但也十分简单的例子，求一棵二叉树的最大值，不难吧（简单起见，假设节点中的值都是非负数）：

```
int maxVal(TreeNode root) {
    if (root == null)
```

```
        return -1;
    int left = maxVal(root.left);
    int right = maxVal(root.right);
    return max(root.val, left, right);
}
```

你看这个问题也符合最优子结构，以 `root` 为根的树的最大值，可以通过两边子树（子问题）的最大值推导出来，结合刚才学校和班级的例子，很容易理解吧。

当然这也不是动态规划问题，旨在说明，最优子结构并不是动态规划独有的一种性质，能求最值的问题大部分都具有这个性质；**但反过来，最优子结构性质作为动态规划问题的必要条件，一定是让你求最值的**，以后碰到那种恶心人的最值题，思路往动态规划想就对了，这就是套路。

动态规划不就是从最简单的 base case 往后推导吗，可以想象成一个链式反应，以小博大。但只有符合最优子结构的问题，才有发生这种链式反应的性质。

找最优子结构的过程，其实就是证明状态转移方程正确性的过程，方程符合最优子结构就可以写暴力解了，写出暴力解就可以看出有没有重叠子问题了，有则优化，无则 OK。这也是套路，经常刷题的朋友应该能体会。

这里就不举那些正宗动态规划的例子了，读者可以翻翻历史文章，看看状态转移是如何遵循最优子结构的，这个话题就聊到这，下面再来看另外个动态规划迷惑行为。

二、dp 数组的遍历方向

我相信读者做动态规问题时，肯定会对 `dp` 数组的遍历顺序有些头疼。我们拿二维 `dp` 数组来举例，有时候我们是正向遍历：

```
int[][] dp = new int[m][n];
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
        // 计算 dp[i][j]
```

有时候我们反向遍历：

```
for (int i = m - 1; i >= 0; i--)
    for (int j = n - 1; j >= 0; j--)
        // 计算 dp[i][j]
```

有时候可能会斜向遍历：

```
// 斜着遍历数组
for (int l = 2; l <= n; l++) {
    for (int i = 0; i <= n - l; i++) {
        int j = l + i - 1;
        // 计算 dp[i][j]
    }
}
```

甚至更让人迷惑的是，有时候发现正向反向遍历都可以得到正确答案，比如我们在「团灭股票问题」中有的地方就正反皆可。

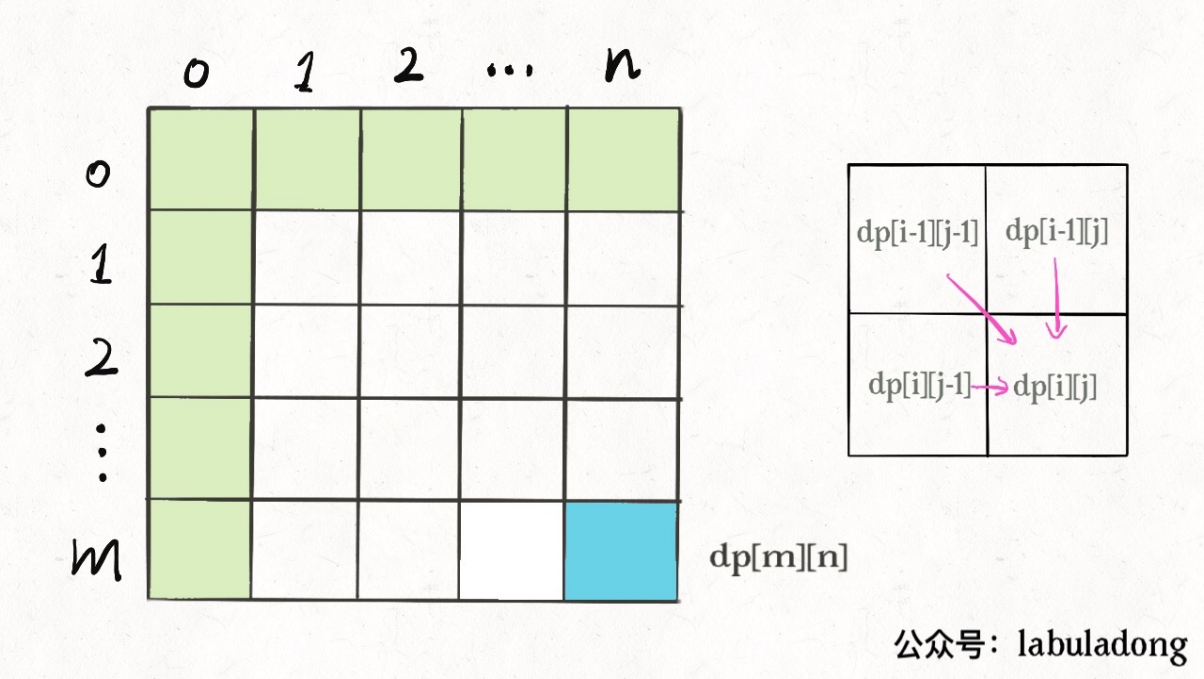
那么，如果仔细观察的话可以发现其中的原因的。你只要把住两点就行了：

1、遍历的过程中，所需的状态必须是已经计算出来的。

2、遍历的终点必须是存储结果的那个位置。

下面来距离解释上面两个原则是什么意思。

比如编辑距离这个经典的问题，详解见前文「编辑距离详解」，我们通过对 `dp` 数组的定义，确定了 base case 是 `dp[..][0]` 和 `dp[0][..]`，最终答案是 `dp[m][n]`；而且我们通过状态转移方程知道 `dp[i][j]` 需要从 `dp[i-1][j]`，`dp[i][j-1]`，`dp[i-1][j-1]` 转移而来，如下图：

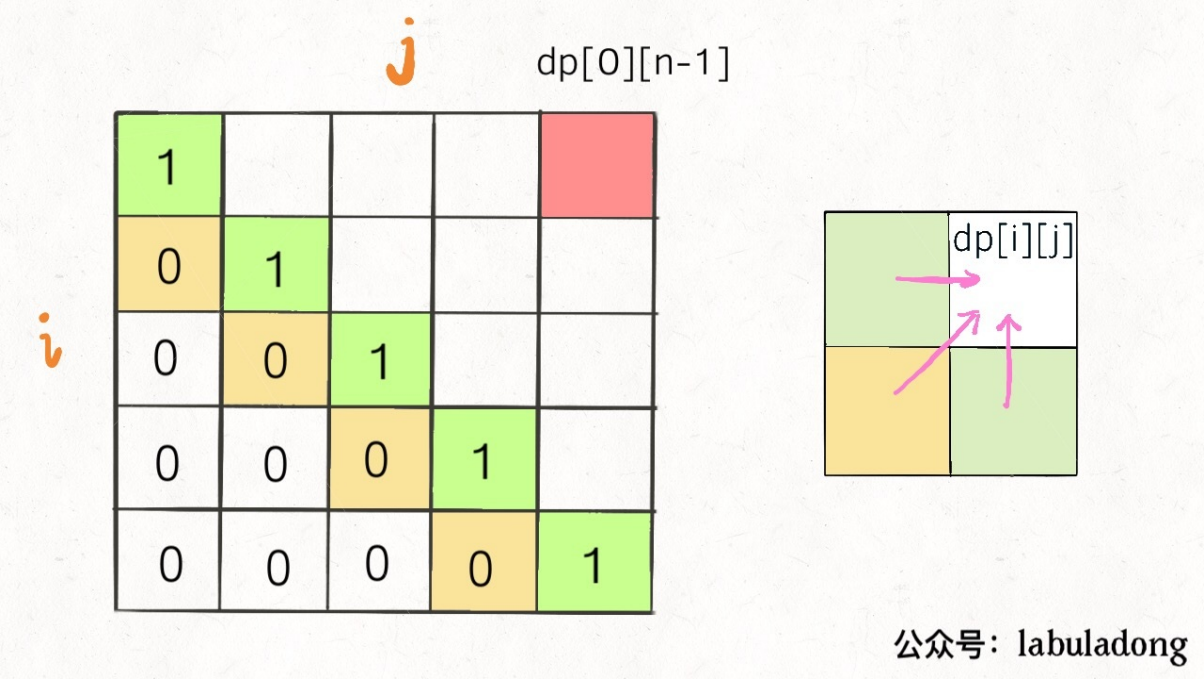


那么，参考刚才说的两条原则，你该怎么遍历 `dp` 数组？肯定是正向遍历：

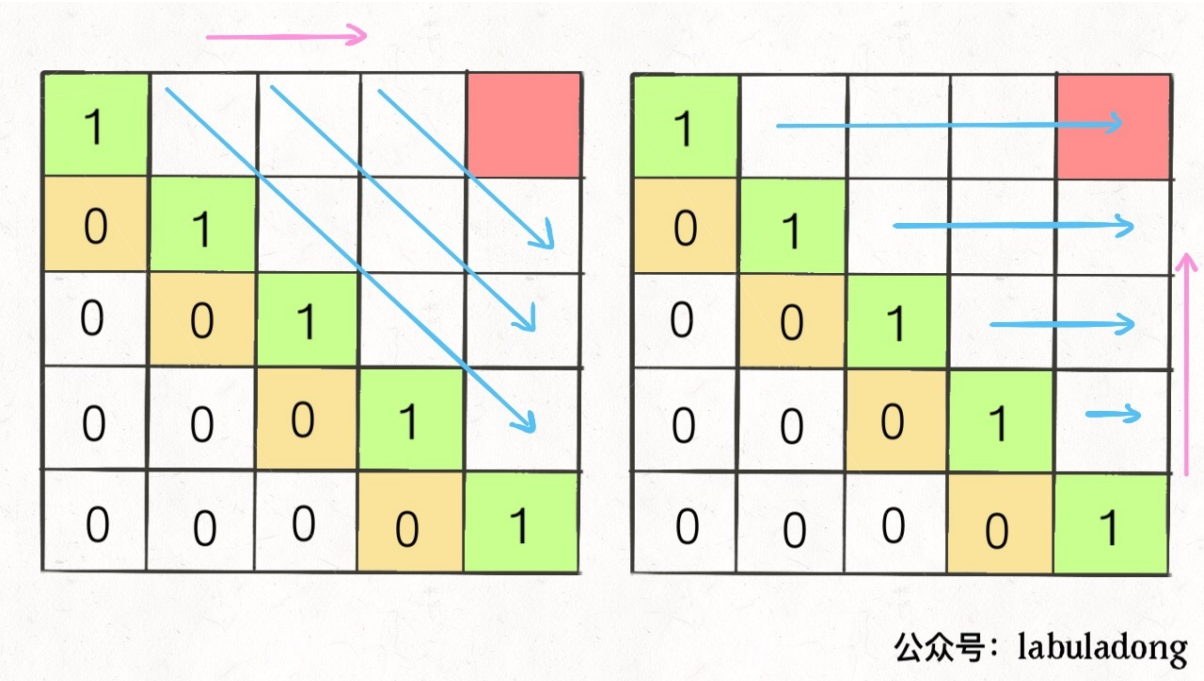
```
for (int i = 1; i < m; i++)
    for (int j = 1; j < n; j++)
        // 通过 dp[i-1][j], dp[i][j - 1], dp[i-1][j-1]
        // 计算 dp[i][j]
```

因为，这样每一步迭代的左边、上边、左上边的位置都是 base case 或者之前计算过的，而且最终结束在我们想要的答案 `dp[m][n]`。

再举一例，回文子序列问题，详见前文「子序列问题模板」，我们通过过对 `dp` 数组的定义，确定了 base case 处在中间的对角线，`dp[i][j]` 需要从 `dp[i+1][j]`，`dp[i][j-1]`，`dp[i+1][j-1]` 转移而来，想要求的最终答案是 `dp[0][n-1]`，如下图：



这种情况根据刚才的两个原则，就可以有两种正确的遍历方式：



要么从左至右斜着遍历，要么从下向上从左到右遍历，这样才能保证每次 `dp[i][j]` 的左边、下边、左下边已经计算完毕，得到正确结果。

现在，你应该理解了这两个原则，主要就是看 base case 和最终结果的存储位置，保证遍历过程中使用的数据都是计算完毕的就行，有时候确实存在多种方法可以得到正确答案，可根据个人口味自行选择。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

动态规划设计：最长递增子序列

很多读者反应，就算看了前文动态规划详解，了解了动态规划的套路，也不会写状态转移方程，没有思路，怎么办？本文就借助「最长递增子序列」来讲一种设计动态规划的通用技巧：数学归纳思想。

最长递增子序列（Longest Increasing Subsequence，简写 LIS）是比较经典的一个问题，比较容易想到的是动态规划解法，时间复杂度 O(N^2)，我们借这个问题来由浅入深讲解如何写动态规划。比较难想到的是利用二分查找，时间复杂度是 O(NlogN)，我们通过一种简单的纸牌游戏来辅助理解这种巧妙的解法。

先看一下题目，很容易理解：

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

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

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。
- 你算法的时间复杂度应该为 $O(n^2)$ 。

进阶: 你能将算法的时间复杂度降低到 $O(n \log n)$ 吗?

注意「子序列」和「子串」这两个名词的区别，子串一定是连续的，而子序列不一定是连续的。下面先来一步一步设计动态规划算法解决这个问题。

一、动态规划解法

动态规划的核心设计思想是数学归纳法。

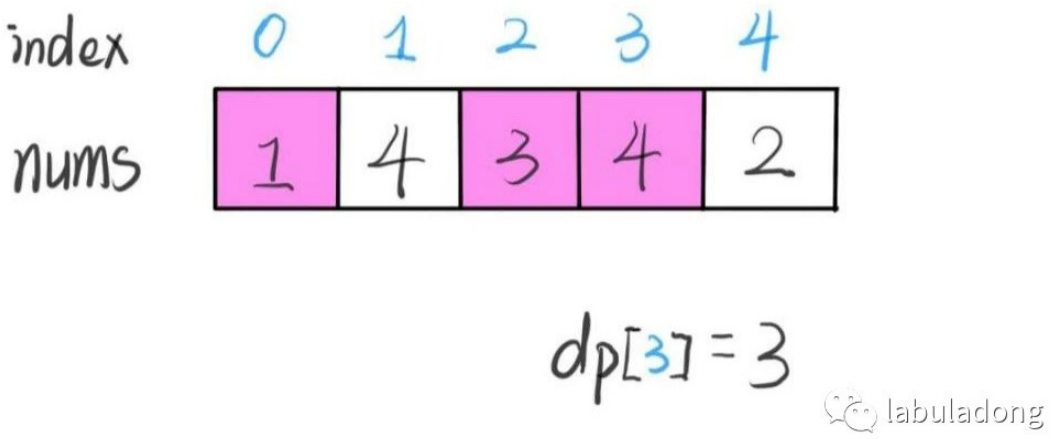
相信大家对数学归纳法都不陌生，高中就学过，而且思路很简单。比如我们想证明一个数学结论，那么我们先假设这个结论在 $k<n$ 时成立，然后想办法证明 $k=n$ 的时候此结论也成立。如果能够证明出来，那么就说明这个结论对于 k 等于任何数都成立。

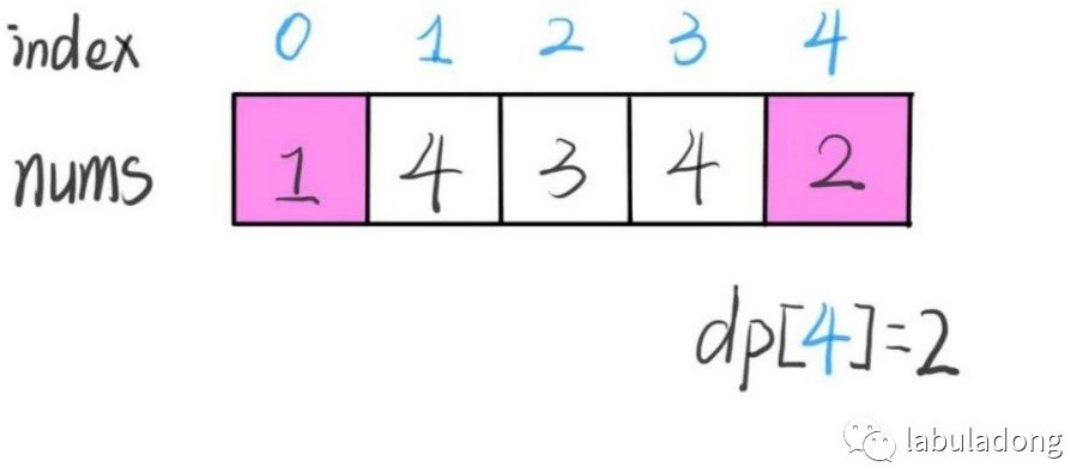
类似的，我们设计动态规划算法，不是需要一个 dp 数组吗？我们可以假设 $dp[0...i-1]$ 都已经被算出来了，然后问自己：怎么通过这些结果算出 dp[i]？

直接拿最长递增子序列这个问题举例你就明白了。不过，首先要定义清楚 dp 数组的含义，即 dp[i] 的值到底代表着什么？

我们的定义是这样的：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度。

举两个例子：





算法演进的过程是这样的，：

【pdf/mobi格式不支持 GIF:%E6%9C%80%E9%95%BF%E9%80%92%E5%A2%9E%E5%AD%90%E5%BA%8F%E5%88%97/gif1.gif】 请查看【关于本小抄及作者】章节的解决方案

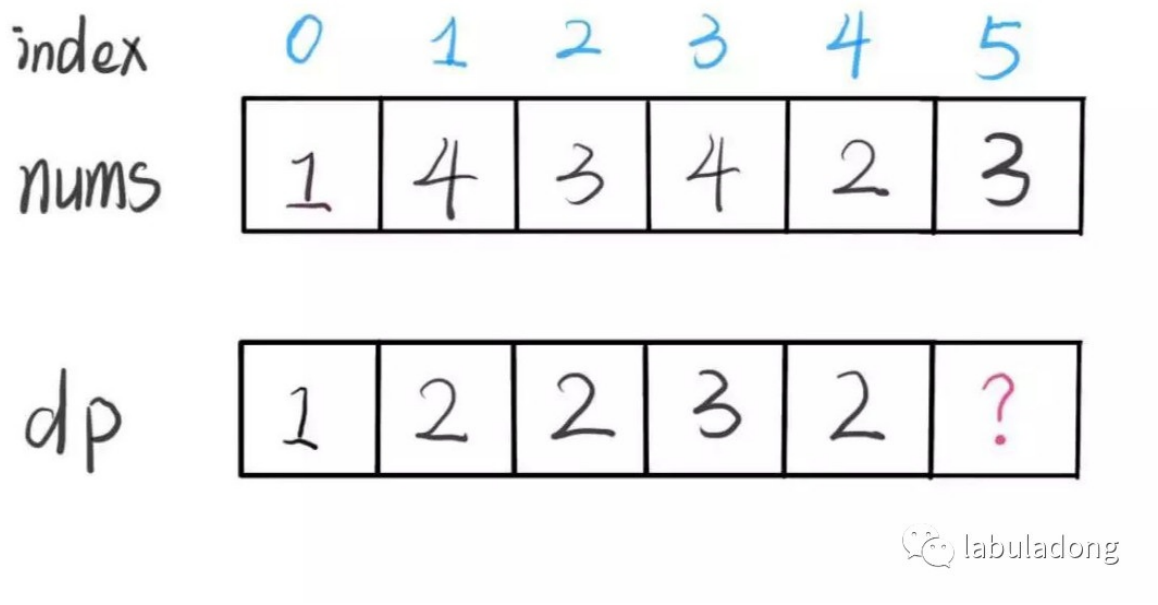
根据这个定义，我们的最终结果（子序列的最大长度）应该是 dp 数组中的最大值。

```
int res = 0;
for (int i = 0; i < dp.size(); i++) {
    res = Math.max(res, dp[i]);
}
return res;
```

读者也许会问，刚才这个过程中每个 dp[i] 的结果是我们肉眼看出来的，我们应该怎么设计算法逻辑来正确计算每个 dp[i] 呢？

这就是动态规划的重头戏了，要思考如何进行状态转移，这里就可以使用数学归纳的思想：

我们已经知道了 $dp[0...4]$ 的所有结果，我们如何通过这些已知结果推出 $dp[5]$ 呢？



根据刚才我们对 dp 数组的定义，现在想求 dp[5] 的值，也就是想求以 nums[5] 为结尾的最长递增子序列。

nums[5] = 3，既然是递增子序列，我们只要找到前面那些结尾比 3 小的子序列，然后把 3 接到最后，就可以形成一个新的递增子序列，而且这个新的子序列长度加一。

当然，可能形成很多种新的子序列，但是我们只要最长的，把最长子序列的长度作为 dp[5] 的值即可。

【pdf/mobi格式不支持 GIF:%E6%9C%80%E9%95%BF%E9%80%92%E5%A2%9E%E5%AD%90%E5%BA%8F%E5%88%97/gif2.gif】 请查看【关于本小抄及作者】章节的解决方案

```
for (int j = 0; j < i; j++) {
    if (nums[i] > nums[j])
        dp[i] = Math.max(dp[i], dp[j] + 1);
}
```

这段代码的逻辑就可以算出 dp[5]。到这里，这道算法题我们就基本做完了。读者也许会问，我们刚才只是算了 dp[5] 呀，dp[4], dp[3] 这些怎么算呢？

类似数学归纳法，你已经可以算出 dp[5] 了，其他的就都可以算出来：

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

还有一个细节问题，dp 数组应该全部初始化为 1，因为子序列最少也要包含自己，所以长度最小为 1。下面我们看一下完整代码：

```
public int lengthOfLIS(int[] nums) {
    int[] dp = new int[nums.length];
    // dp 数组全都初始化为 1
    Arrays.fill(dp, 1);
    for (int i = 0; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                dp[i] = Math.max(dp[i], dp[j] + 1);
        }
    }

    int res = 0;
    for (int i = 0; i < dp.length; i++) {
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

至此，这道题就解决了，时间复杂度 O(N^2)。总结一下动态规划的设计流程：

首先明确 dp 数组所存数据的含义。这步很重要，如果不得当或者不够清晰，会阻碍之后的步骤。

然后根据 dp 数组的定义，运用数学归纳法的思想，假设 $dp[0...i-1]$ 都已知，想办法求出 $dp[i]$，一旦这一步完成，整个题目基本就解决了。

但如果无法完成这一步，很可能就是 dp 数组的定义不够恰当，需要重新定义 dp 数组的含义；或者可能是 dp 数组存储的信息还不够，不足以推出下一步的答案，需要把 dp 数组扩大成二维数组甚至三维数组。

最后想一想问题的 base case 是什么，以此来初始化 dp 数组，以保证算法正确运行。

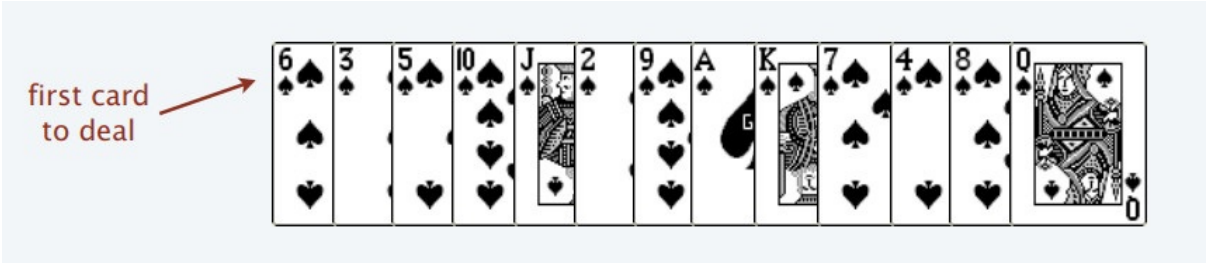
二、二分查找解法

这个解法的时间复杂度会将为 O(NlogN)，但是说实话，正常人基本想不到这种解法（也许玩过某些纸牌游戏的人可以想出来）。所以如果大家了解一下就好，正常情况下能够给出动态规划解法就已经很不错了。

根据题目的意思，我都很难想象这个问题竟然能和二分查找扯上关系。其实最长递增子序列和一种叫做 patience game 的纸牌游戏有关，甚至有一种排序方法就叫做 patience sorting（耐心排序）。

为了简单期间，后文跳过所有数学证明，通过一个简化的例子来理解一下思路。

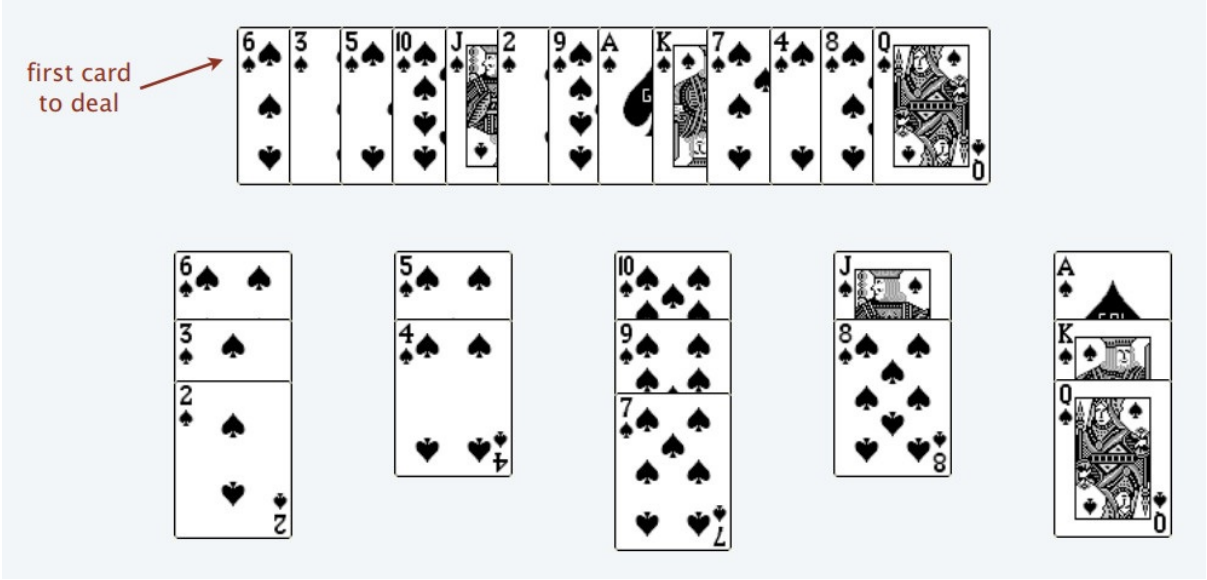
首先，给你一排扑克牌，我们像遍历数组那样从左到右一张一张处理这些扑克牌，最终要把这些牌分成若干堆。



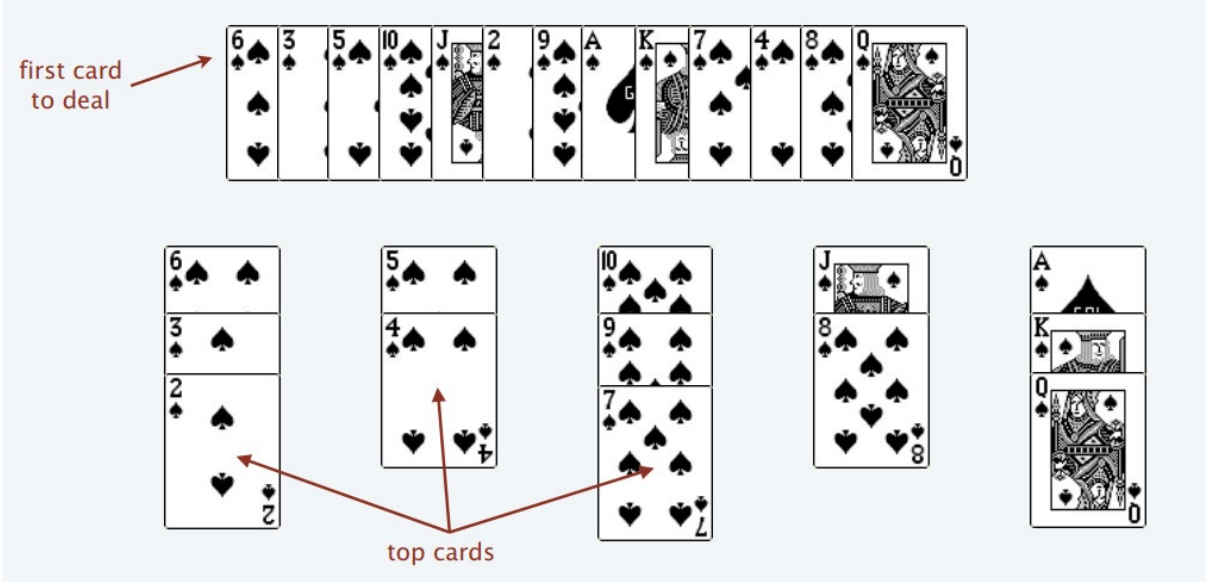
处理这些扑克牌要遵循以下规则：

只能把点数小的牌压到点数比它大的牌上。如果当前牌点数较大没有可以放置的堆，则新建一个堆，把这张牌放进去。如果当前牌有多个堆可供选择，则选择最左边的堆放置。

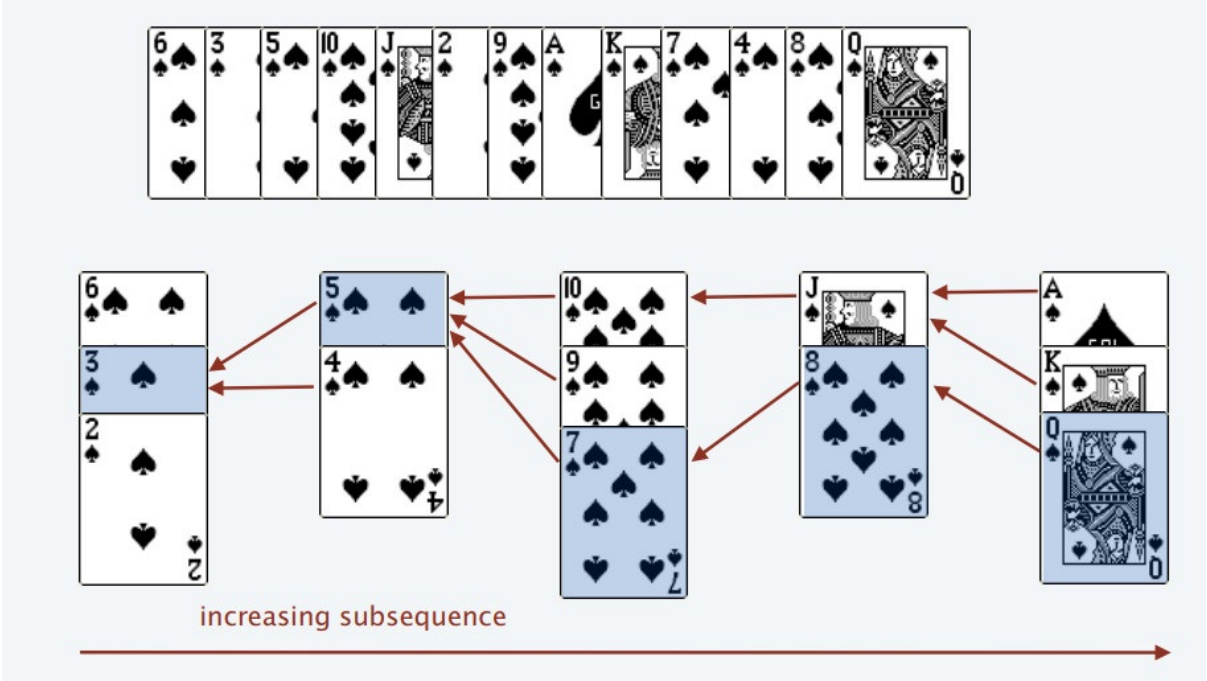
比如说上述的扑克牌最终会被分成这样 5 堆（我们认为 A 的值是最大的，而不是 1）。



为什么遇到多个可选择堆的时候要放到最左边的堆上呢？因为这样可以保证牌堆顶的牌有序（2, 4, 7, 8, Q），证明略。



按照上述规则执行，可以算出最长递增子序列，牌的堆数就是最长递增子序列的长度，证明略。



我们只要把处理扑克牌的过程编程写出来即可。每次处理一张扑克牌不是要找一个合适的牌堆顶来放吗，牌堆顶的牌不是有序吗，这就能用到二分查找了：用二分查找来搜索当前牌应放置的位置。

PS：旧文二分查找算法详解详细介绍了二分查找的细节及变体，这里就完美应用上了。如果没读过强烈建议阅读。

```
public int lengthOfLIS(int[] nums) {
    int[] top = new int[nums.length];
    // 牌堆数初始化为 0
    int piles = 0;
    for (int i = 0; i < nums.length; i++) {
        // 要处理的扑克牌
        int poker = nums[i];

        /***** 搜索左侧边界的二分查找 *****/
        int left = 0, right = piles;
        while (left < right) {
            int mid = (left + right) / 2;
            if (top[mid] > poker) {
                right = mid;
            } else if (top[mid] < poker) {
                left = mid + 1;
            } else {
                right = mid;
```

```
            }
        }
        /*********************************/

        // 没找到合适的牌堆，新建一堆
        if (left == piles) piles++;
        // 把这张牌放到牌堆顶
        top[left] = poker;
    }
    // 牌堆数就是 LIS 长度
    return piles;
}
```

至此，二分查找的解法也讲解完毕。

这个解法确实很难想到。首先涉及数学证明，谁能想到按照这些规则执行，就能得到最长递增子序列呢？其次还有二分查找的运用，要是对二分查找的细节不清楚，给了思路也很难写对。

所以，这个方法作为思维拓展好了。但动态规划的设计方法应该完全理解：假设之前的答案已知，利用数学归纳的思想正确进行状态的推演转移，最终得到答案。

编辑距离

前几天看了一份鹅场的面试题，算法部分大半是动态规划，最后一题就是写一个计算编辑距离的函数，今天就专门写一篇文章来探讨一下这个问题。

我个人很喜欢编辑距离这个问题，因为它看起来十分困难，解法却出奇得简单漂亮，而且它是少有的比较实用的算法（是的，我承认很多算法问题都不太实用）。下面先来看下题目：

给定两个字符串 **s1** 和 **s2**，计算出将 **s1** 转换成 **s2** 所使用的最少操作数 。

你可以对一个字符串进行如下三种操作：

1. 插入一个字符
2. 删除一个字符
3. 替换一个字符

示例 1:

```
输入: s1 = "horse", s2 = "ros"
输出: 3
解释:
horse -> rorse (将 'h' 替换为 'r')
rorse -> rose (删除 'r')
rose -> ros (删除 'e')
```

示例 2:

```
输入: s1 = "intention", s2 = "execution"
输出: 5
解释:
intention -> inention (删除 't')
inention -> enention (将 'i' 替换为 'e')
enention -> exention (将 'n' 替换为 'x')
exention -> exection (将 'n' 替换为 'c')
exection -> execution (插入 'u')
```

为什么说这个问题难呢，因为显而易见，它就是难，让人手足无措，望而生畏。

为什么说它实用呢，因为前几天我就在日常生活中用到了这个算法。之前有一篇公众号文章由于疏忽，写错位了一段内容，我决定修改这部分内容让逻辑通顺。但是公众号文章最多只能修改 20 个字，且只支持增、删、替换操

作（跟编辑距离问题一模一样），于是我就用算法求出了一个最优方案，只用了 16 步就完成了修改。

再比如高大上一点的应用，DNA 序列是由 A,G,C,T 组成的序列，可以类比成字符串。编辑距离可以衡量两个 DNA 序列的相似度，编辑距离越小，说明这两段 DNA 越相似，说不定这俩 DNA 的主人是远古近亲啥的。

下面言归正传，详细讲解一下编辑距离该怎么算，相信本文会让你有收获。

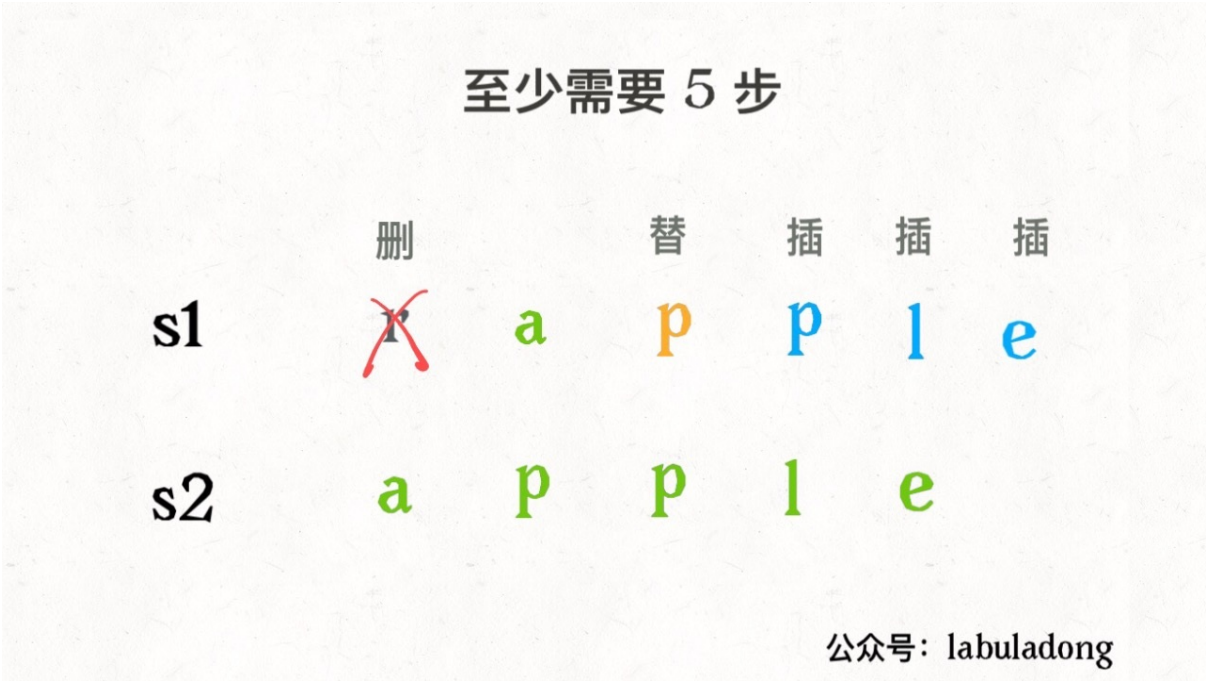
一、思路

编辑距离问题就是给我们两个字符串 `s1` 和 `s2`，只能用三种操作，让我们把 `s1` 变成 `s2`，求最少的操作数。需要明确的是，不管是把 `s1` 变成 `s2` 还是反过来，结果都是一样的，所以后文就以 `s1` 变成 `s2` 举例。

前文「最长公共子序列」说过，**解决两个字符串的动态规划问题，一般都是用两个指针 `i,j` 分别指向两个字符串的最后，然后一步步往前走，缩小问题的规模**。

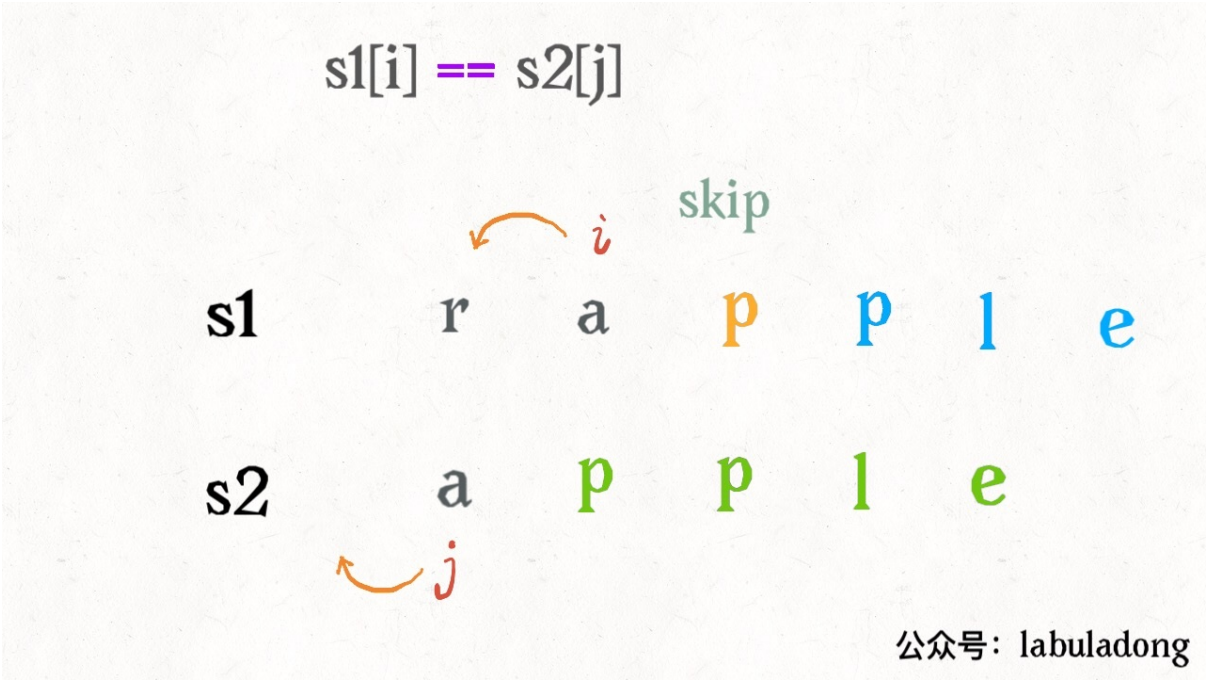
设两个字符串分别为 "rad" 和 "apple"，为了把 `s1` 变成 `s2`，算法会这样进行：

【pdf/mobi格式不支持GIF:editDistance/edit.gif】 请查看【关于本小抄及作者】章节的解决方案



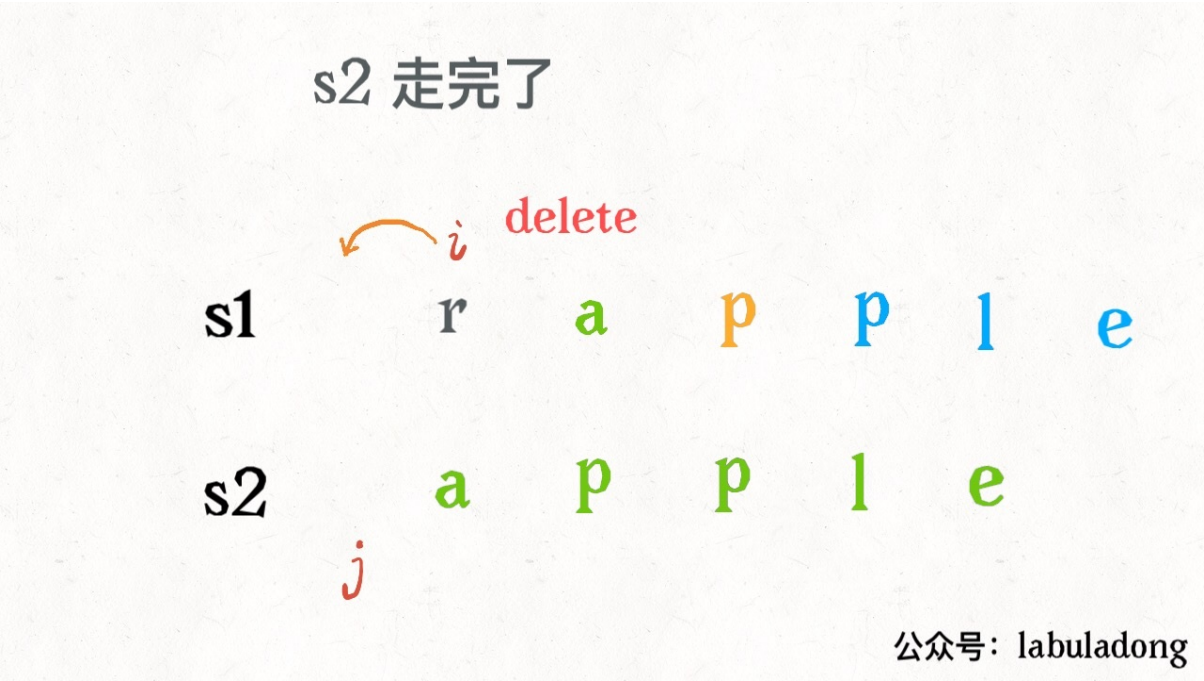
请记住这个 GIF 过程，这样就能算出编辑距离。关键在于如何做出正确的操作，稍后会讲。

根据上面的 GIF，可以发现操作不只有三个，其实还有第四个操作，就是什么都不要做（skip）。比如这个情况：



因为这两个字符本来就相同，为了使编辑距离最小，显然不应该对它们有任何操作，直接往前移动 `i,j` 即可。

还有一个很容易处理的情况，就是 `j` 走完 `s2` 时，如果 `i` 还没走完 `s1`，那么只能用删除操作把 `s1` 缩短为 `s2`。比如这个情况：



类似的，如果 `i` 走完 `s1` 时 `j` 还没走完了 `s2`，那就只能用插入操作把 `s2` 剩下的字符全部插入 `s1`。等会会看到，这两种情况就是算法的 **base case**。

下面详解一下如何将思路转换成代码，坐稳，要发车了。

二、代码详解

先梳理一下之前的思路：

base case 是 `i` 走完 `s1` 或 `j` 走完 `s2`，可以直接返回另一个字符串剩下的长度。

对于每对儿字符 `s1[i]` 和 `s2[j]`，可以有四种操作：

```
if s1[i] == s2[j]:
    啥都别做（skip）
```

```
    i, j 同时向前移动
else:
    三选一：
        插入（insert）
        删除（delete）
        替换（replace）
```

有这个框架，问题就已经解决了。读者也许会问，这个「三选一」到底该怎么选择呢？很简单，全试一遍，哪个操作最后得到的编辑距离最小，就选谁。这里需要递归技巧，理解需要点技巧，先看下代码：

```python
def minDistance(s1, s2) -> int:

    def dp(i, j):
        # base case
        if i == -1: return j + 1
        if j == -1: return i + 1

        if s1[i] == s2[j]:
            return dp(i - 1, j - 1)  # 啥都不做
        else:
            return min(
                dp(i, j - 1) + 1,     # 插入
                dp(i - 1, j) + 1,     # 删除
                dp(i - 1, j - 1) + 1 # 替换
            )

    # i, j 初始化指向最后一个索引
    return dp(len(s1) - 1, len(s2) - 1)
```

下面来详细解释一下这段递归代码，base case 应该不用解释了，主要解释一下递归部分。

都说递归代码的可解释性很好，这是有道理的，只要理解函数的定义，就能很清楚地理解算法的逻辑。我们这里 dp(i, j) 函数的定义是这样的：

```python
def dp(i, j) -> int
# 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离
```

记住这个定义之后，先来看这段代码：

```
if s1[i] == s2[j]:
    return dp(i - 1, j - 1)  # 啥都不做
# 解释：
# 本来就相等，不需要任何操作
# s1[0..i] 和 s2[0..j] 的最小编辑距离等于
# s1[0..i-1] 和 s2[0..j-1] 的最小编辑距离
# 也就是说 dp(i, j) 等于 dp(i-1, j-1)
```

如果 `s1[i]!=s2[j]`，就要对三个操作递归了，稍微需要点思考：

```
dp(i, j - 1) + 1,    # 插入
# 解释：
# 我直接在 s1[i] 插入一个和 s2[j] 一样的字符
# 那么 s2[j] 就被匹配了，前移 j，继续跟 i 对比
# 别忘了操作数加一
```

【pdf/mobi格式不支持GIF:editDistance/insert.gif】 请查看【关于本小抄及作者】章节的解决方案

```
dp(i - 1, j) + 1,    # 删除
# 解释：
# 我直接把 s[i] 这个字符删掉
# 前移 i，继续跟 j 对比
# 操作数加一
```

【pdf/mobi格式不支持GIF:editDistance/delete.gif】 请查看【关于本小抄及作者】章节的解决方案

```
dp(i - 1, j - 1) + 1 # 替换
# 解释：
# 我直接把 s1[i] 替换成 s2[j]，这样它俩就匹配了
# 同时前移 i，j 继续对比
# 操作数加一
```

【pdf/mobi格式不支持GIF:editDistance/replace.gif】 请查看【关于本小抄及作者】章节的解决方案

现在，你应该完全理解这段短小精悍的代码了。还有点小问题就是，这个解法是暴力解法，存在重叠子问题，需要用动态规划技巧来优化。

怎么能一眼看出存在重叠子问题呢？前文「动态规划之正则表达式」有提过，这里再简单提一下，需要抽象出本文算法的递归框架：

```
def dp(i, j):
    dp(i - 1, j - 1) #1
    dp(i, j - 1)     #2
    dp(i - 1, j)     #3
```

对于子问题 `dp(i-1, j-1)`，如何通过原问题 `dp(i, j)` 得到呢？有不止一条路径，比如 `dp(i, j) -> #1` 和 `dp(i, j) -> #2 -> #3`。一旦发现一条重复路径，就说明存在巨量重复路径，也就是重叠子问题。

三、动态规划优化

对于重叠子问题呢，前文「动态规划详解」详细介绍过，优化方法无非是备忘录或者 DP table。

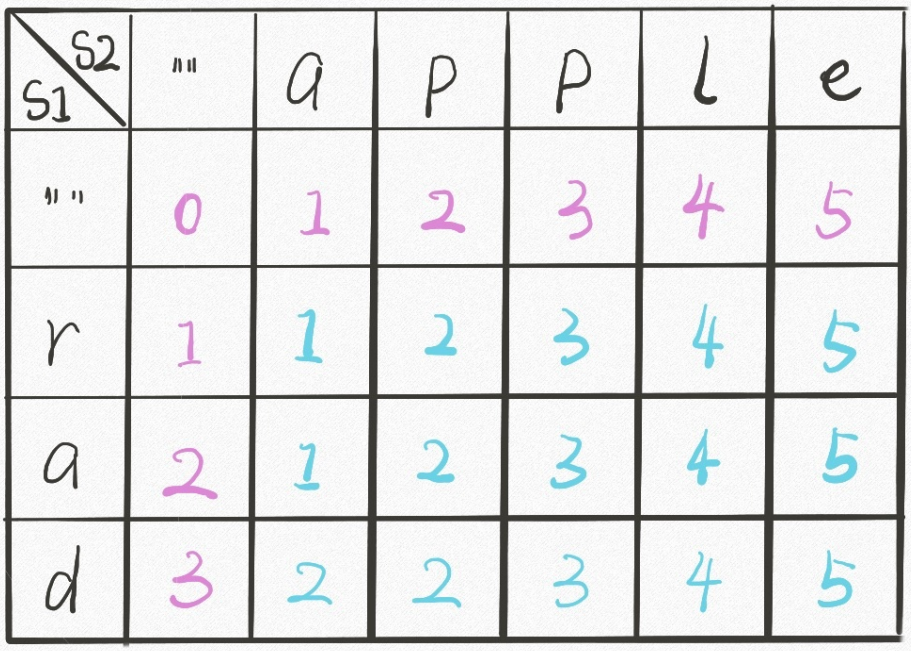
备忘录很好加，原来的代码稍加修改即可：

```
def minDistance(s1, s2) -> int:

    memo = dict() # 备忘录
    def dp(i, j):
        if (i, j) in memo:
            return memo[(i, j)]
        ...

        if s1[i] == s2[j]:
            memo[(i, j)] = ...
        else:
            memo[(i, j)] = ...
        return memo[(i, j)]
```

```
    return dp(len(s1) - 1, len(s2) - 1)
```

主要说下 DP table 的解法：

首先明确 dp 数组的含义，dp 数组是一个二维数组，长这样：



有了之前递归解法的铺垫，应该很容易理解。`dp[..][0]` 和 `dp[0][..]` 对应 base case，`dp[i][j]` 的含义和之前的 dp 函数类似：

```
def dp(i, j) -> int
# 返回 s1[0..i] 和 s2[0..j] 的最小编辑距离

dp[i-1][j-1]
# 存储 s1[0..i] 和 s2[0..j] 的最小编辑距离
```

dp 函数的 base case 是 `i,j` 等于 -1，而数组索引至少是 0，所以 dp 数组会偏移一位。

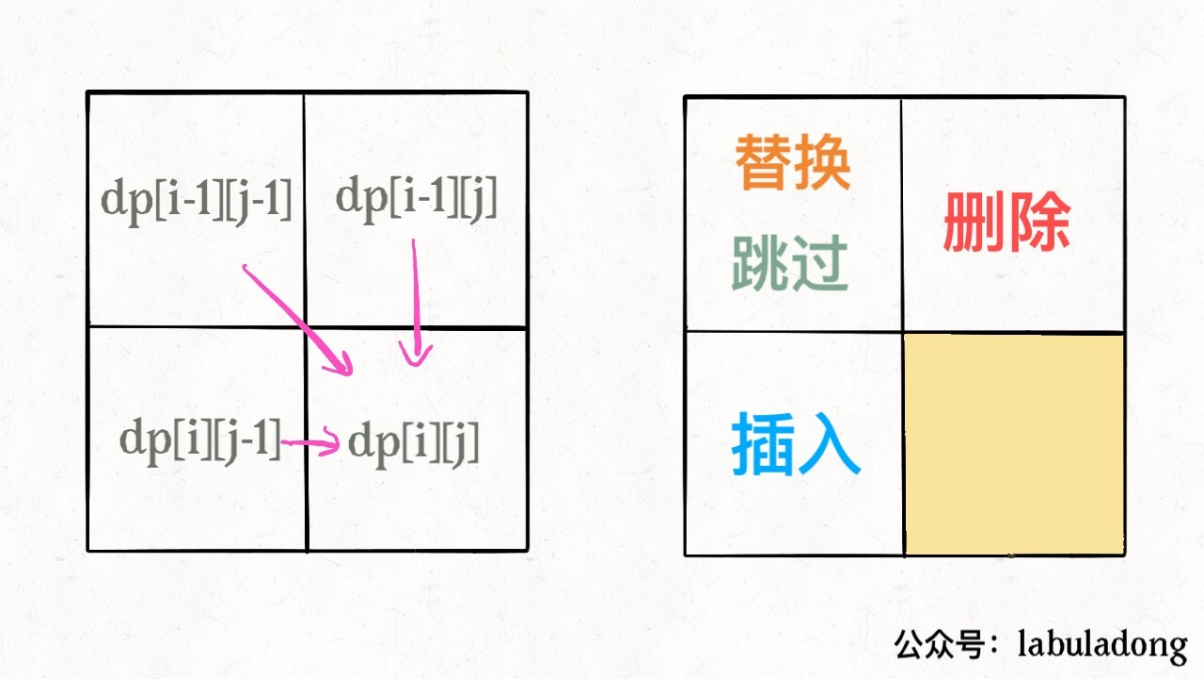
既然 dp 数组和递归 dp 函数含义一样，也就可以直接套用之前的思路写代码，**唯一不同的是，DP table 是自底向上求解，递归解法是自顶向下求解：**

```
int minDistance(String s1, String s2) {
```

```
    int m = s1.length(), n = s2.length();
    int[][] dp = new int[m + 1][n + 1];
    // base case
    for (int i = 1; i <= m; i++)
        dp[i][0] = i;
    for (int j = 1; j <= n; j++)
        dp[0][j] = j;
    // 自底向上求解
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s1.charAt(i-1) == s2.charAt(j-1))
                dp[i][j] = dp[i - 1][j - 1];
            else
                dp[i][j] = min(
                    dp[i - 1][j] + 1,
                    dp[i][j - 1] + 1,
                    dp[i-1][j-1] + 1
                );
    // 储存着整个 s1 和 s2 的最小编辑距离
    return dp[m][n];
}

int min(int a, int b, int c) {
    return Math.min(a, Math.min(b, c));
}
```

三、扩展延伸

一般来说，处理两个字符串的动态规划问题，都是按本文的思路处理，建立 DP table。为什么呢，因为易于找出状态转移的关系，比如编辑距离的 DP table：



还有一个细节，既然每个 `dp[i][j]` 只和它附近的三个状态有关，空间复杂度是可以压缩成 $O(min(M, N))$ 的（M，N 是两个字符串的长度）。不难，但是可解释性大大降低，读者可以自己尝试优化一下。

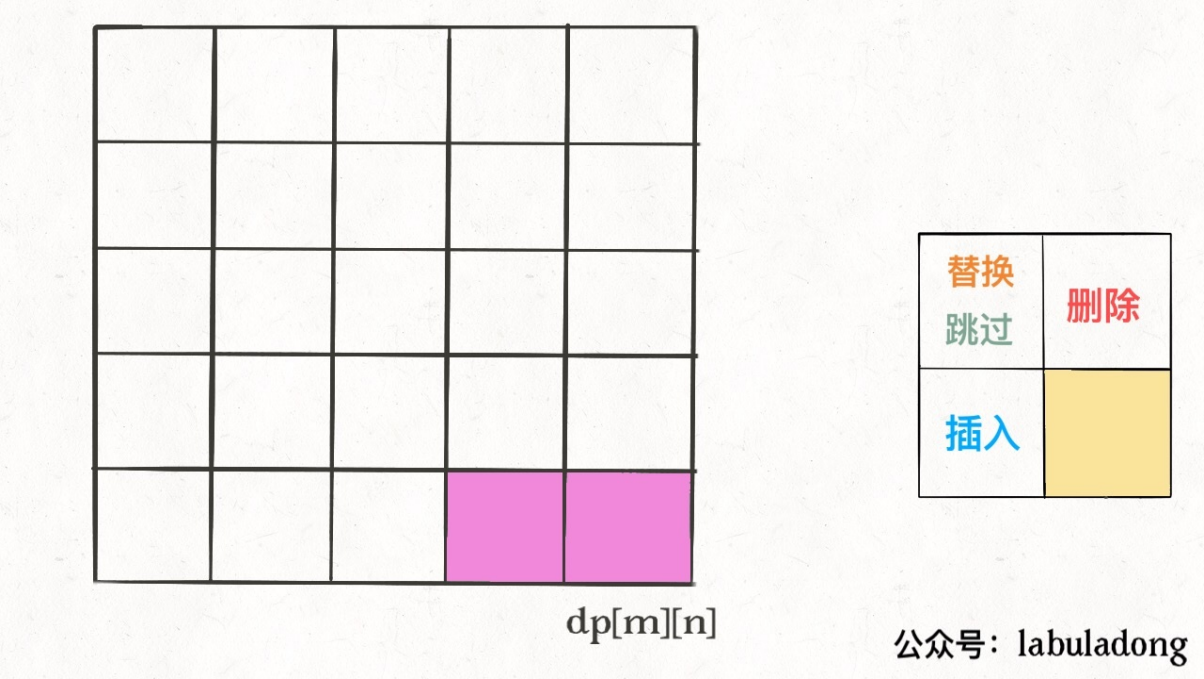
你可能还会问，**这里只求出了最小的编辑距离，那具体的操作是什么**？你之前举的修改公众号文章的例子，只有一个最小编辑距离肯定不够，还得知道具体怎么修改才行。

这个其实很简单，代码稍加修改，给 dp 数组增加额外的信息即可：

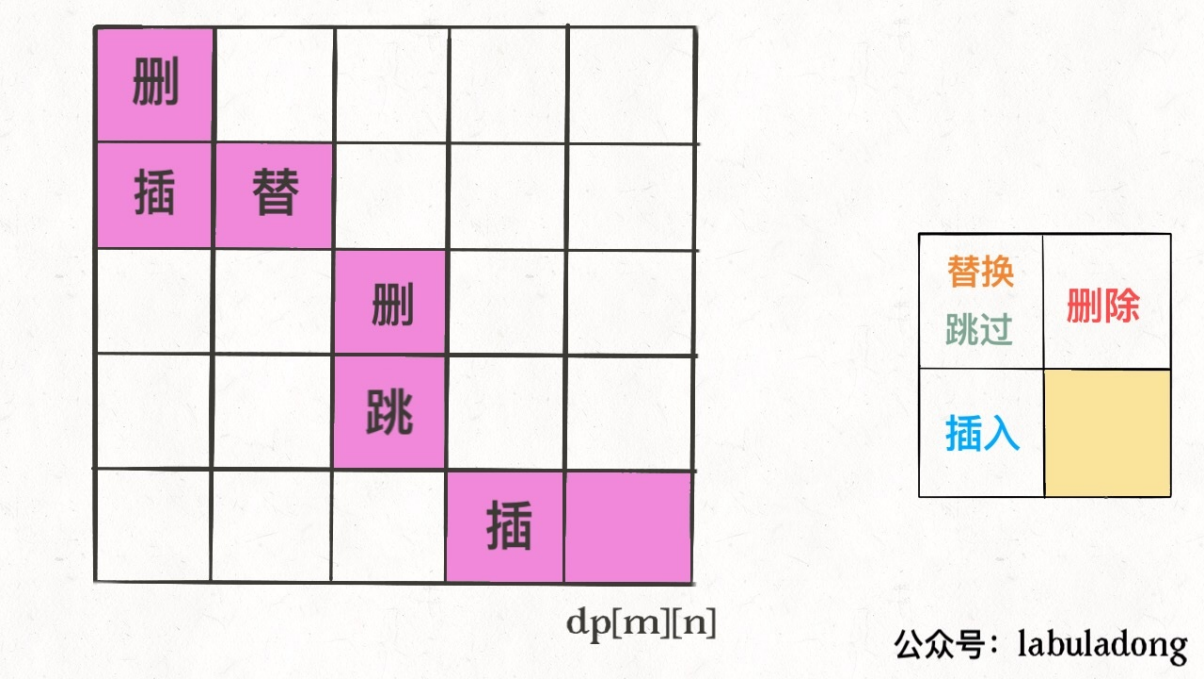
```
// int[][] dp;
Node[][] dp;

class Node {
    int val;
    int choice;
    // 0 代表啥都不做
    // 1 代表插入
    // 2 代表删除
    // 3 代表替换
}
```

`val` 属性就是之前的 dp 数组的数值，`choice` 属性代表操作。在做最优选择时，顺便把操作记录下来，然后就从结果反推具体操作。

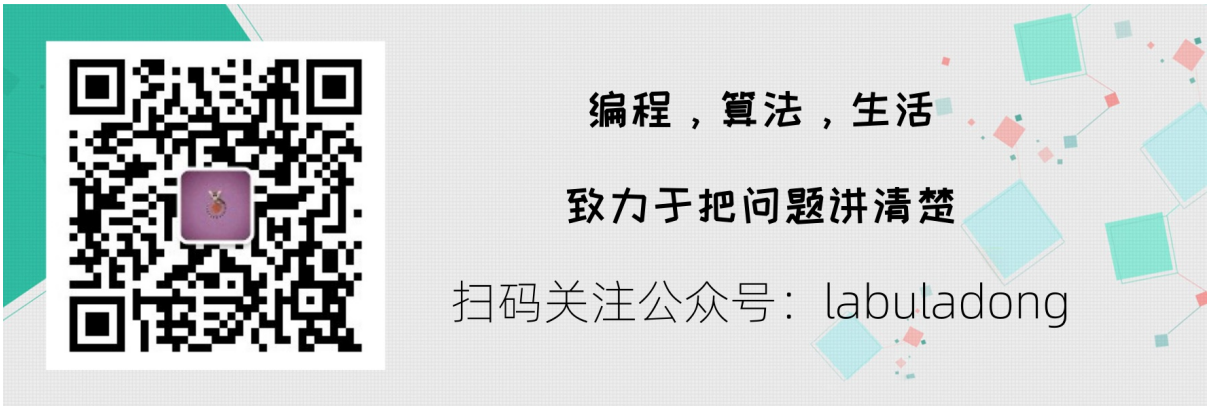
我们的最终结果不是 `dp[m][n]` 吗，这里的 `val` 存着最小编辑距离，`choice` 存着最后一个操作，比如说是插入操作，那么就可以左移一格：



重复此过程，可以一步步回到起点 `dp[0][0]`，形成一条路径，按这条路径上的操作进行编辑，就是最佳方案。



以上就是编辑距离算法的全部内容，如果本文对你有帮助，**欢迎关注我的公众号 labuladong，致力于把算法问题讲清楚～**



经典动态规划问题：高楼扔鸡蛋

今天要聊一个很经典的算法问题，若干层楼，若干个鸡蛋，让你算出最少的尝试次数，找到鸡蛋恰好摔不碎的那层楼。国内大厂以及谷歌脸书面试都经常考察这道题，只不过他们觉得扔鸡蛋太浪费，改成扔杯子，扔破碗什么的。

具体的问题等会再说，但是这道题的解法技巧很多，光动态规划就好几种效率不同的思路，最后还有一种极其高效数学解法。秉承咱们号一贯的作风，拒绝奇技淫巧，拒绝过于诡异的技巧，因为这些技巧无法举一反三，学了也不划算。

下面就来用我们一直强调的动态规划通用思路来研究一下这道题。

一、解析题目

题目是这样：你面前有一栋从 1 到 `N` 共 `N` 层的楼，然后给你 `K` 个鸡蛋（ `K` 至少为 1）。现在确定这栋楼存在楼层 `0 <= F <= N`，在这层楼将鸡蛋扔下去，鸡蛋**恰好没摔碎**（高于 `F` 的楼层都会碎，低于 `F` 的楼层都不会碎）。现在问你，**最坏**情况下，你**至少**要扔几次鸡蛋，才能**确定**这个楼层 `F` 呢？

也就是让你找摔不碎鸡蛋的最高楼层 `F`，但什么叫「最坏情况」下「至少」要扔几次呢？我们分别举个例子就明白了。

比方说**现在先不管鸡蛋个数的限制**，有 7 层楼，你怎么去找鸡蛋恰好摔碎的那层楼？

最原始的方式就是线性扫描：我先在 1 楼扔一下，没碎，我再去 2 楼扔一下，没碎，我再去 3 楼……

以这种策略，**最坏**情况应该就是我试到第 7 层鸡蛋也没碎（ `F = 7` ），也就是我扔了 7 次鸡蛋。

先在你应该理解什么叫做「最坏情况」下了，**鸡蛋破碎一定发生在搜索区间穷尽时**，不会说你在第 1 层摔一下鸡蛋就碎了，这是你运气好，不是最坏情况。

现在再来理解一下什么叫「至少」要扔几次。依然不考虑鸡蛋个数限制，同样是 7 层楼，我们可以优化策略。

最好的策略是使用二分查找思路，我先去第 `(1 + 7) / 2 = 4` 层扔一下：

如果碎了说明 `F` 小于 4，我就去第 `(1 + 3) / 2 = 2` 层试……

如果没碎说明 `F` 大于等于 4，我就去第 `(5 + 7) / 2 = 6` 层试……

以这种策略，**最坏**情况应该是试到第 7 层鸡蛋还没碎（`F = 7`），或者鸡蛋一直碎到第 1 层（`F = 0`）。然而无论那种最坏情况，只需要试 `log7` 向上取整等于 3 次，比刚才尝试 7 次要少，这就是所谓的**至少**要扔几次。

PS：这有点像 Big O 表示法计算算法的复杂度。

实际上，如果不限制鸡蛋个数的话，二分思路显然可以得到最少尝试的次数，但问题是，**现在给你了鸡蛋个数的限制 `K`，直接使用二分思路就不行了**。

比如说只给你 1 个鸡蛋，7 层楼，你敢用二分吗？你直接去第 4 层扔一下，如果鸡蛋没碎还好，但如果碎了你就没有鸡蛋继续测试了，无法确定鸡蛋恰好摔不碎的楼层 `F` 了。这种情况下只能用线性扫描的方法，算法返回结果应该是 7。

有的读者也许会有这种想法：二分查找排除楼层的速度无疑是最快的，那干脆先用二分查找，等到只剩 1 个鸡蛋的时候再执行线性扫描，这样得到的结果是不是就是最少的扔鸡蛋次数呢？

很遗憾，并不是，比如说把楼层变高一些，100 层，给你 2 个鸡蛋，你在 50 层扔一下，碎了，那就只能线性扫描 1～49 层了，最坏情况下要扔 50 次。

如果不要「二分」，变成「五分」「十分」都会大幅减少最坏情况下的尝试次数。比方说第一个鸡蛋每隔十层楼扔，在哪里碎了第二个鸡蛋一个个线性扫描，总共不会超过 20 次。

最优解其实是 14 次。最优策略非常多，而且并没有什么规律可言。

说了这么多废话，就是确保大家理解了题目的意思，而且认识到这个题目确实复杂，就连我们手算都不容易，如何用算法解决呢？

二、思路分析

对动态规划问题，直接套我们以前多次强调的框架即可：这个问题有什么「状态」，有什么「选择」，然后穷举。

「状态」很明显，就是当前拥有的鸡蛋数 `K` 和需要测试的楼层数 `N`。随着测试的进行，鸡蛋个数可能减少，楼层的搜索范围会减小，这就是状态的变化。

「选择」其实就是去选择哪层楼扔鸡蛋。回顾刚才的线性扫描和二分思路，二分查找每次选择到楼层区间的中间去扔鸡蛋，而线性扫描选择一层层向上测试。不同的选择会造成状态的转移。

现在明确了「状态」和「选择」，**动态规划的基本思路就形成了**：肯定是个二维的 `dp` 数组或者带有两个状态参数的 `dp` 函数来表示状态转移；外加一个 for 循环来遍历所有选择，择最优的选择更新状态：

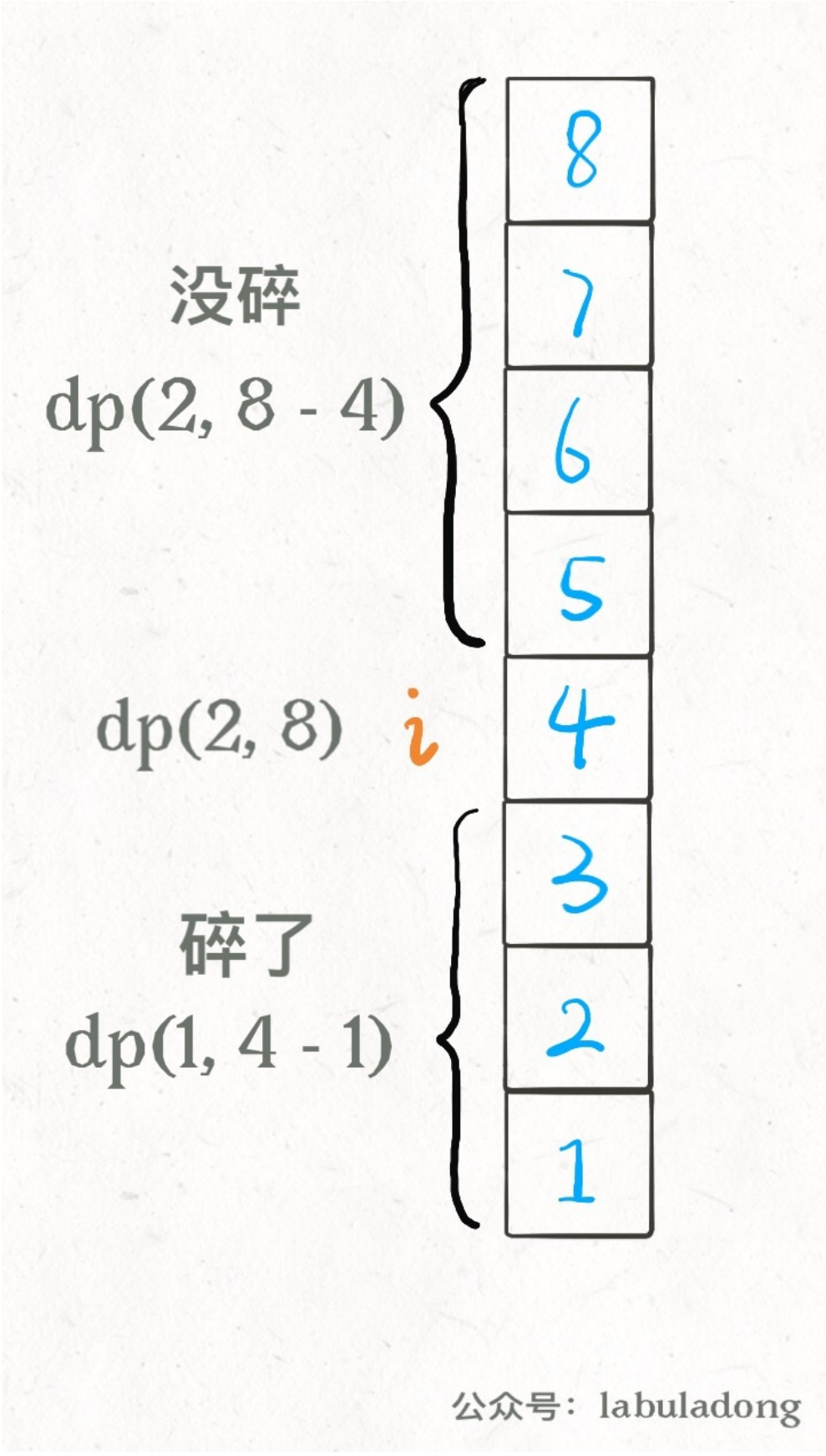
```
# 当前状态为 K 个鸡蛋，面对 N 层楼
# 返回这个状态下的最优结果
def dp(K, N):
    int res
    for 1 <= i <= N:
        res = min(res, 这次在第 i 层楼扔鸡蛋)
    return res
```

这段伪码还没有展示递归和状态转移，不过大致的算法框架已经完成了。

我们选择在第 `i` 层楼扔了鸡蛋之后，可能出现两种情况：鸡蛋碎了，鸡蛋没碎。**注意，这时候状态转移就来了**：

如果鸡蛋碎了，那么鸡蛋的个数 `K` 应该减一，搜索的楼层区间应该从 `[1..N]` 变为 `[1..i-1]` 共 `i-1` 层楼；

如果鸡蛋没碎，那么鸡蛋的个数 `K` 不变，搜索的楼层区间应该从 `[1..N]` 变为 `[i+1..N]` 共 `N-i` 层楼。


8
7
6
5
4
3
2
1
没碎
dp(2, 8 - 4)
dp(2, 8)
i
碎了
dp(1, 4 - 1)
公众号：labuladong

PS：细心的读者可能会问，在第i层楼扔鸡蛋如果没碎，楼层的搜索区间缩小至上面的楼层，是不是应该包含第i层楼呀？不必，因为已经包含了。开头说了 F 是可以等于 0 的，向上递归后，第i层楼其实就相当于第 0 层，可以被取到，所以说并没有错误。

因为我们要求的是**最坏情况**下扔鸡蛋的次数，所以鸡蛋在第 `i` 层楼碎没碎，取决于那种情况的结果**更大**：

```
def dp(K, N):
    for 1 <= i <= N:
        # 最坏情况下的最少扔鸡蛋次数
        res = min(res,
                  max(
                        dp(K - 1, i - 1), # 碎
                        dp(K, N - i)      # 没碎
                    ) + 1 # 在第 i 楼扔了一次
                )
    return res
```

递归的 base case 很容易理解：当楼层数 `N` 等于 0 时，显然不需要扔鸡蛋；当鸡蛋数 `K` 为 1 时，显然只能线性扫描所有楼层：

```
def dp(K, N):
    if K == 1: return N
    if N == 0: return 0
    ...
```

至此，其实这道题就解决了！只要添加一个备忘录消除重叠子问题即可：

```
def superEggDrop(K: int, N: int):

    memo = dict()
    def dp(K, N) -> int:
        # base case
        if K == 1: return N
        if N == 0: return 0
        # 避免重复计算
```

```
        if (K, N) in memo:
            return memo[(K, N)]

        res = float('INF')
        # 穷举所有可能的选择
        for i in range(1, N + 1):
            res = min(res,
                      max(
                            dp(K, N - i),
                            dp(K - 1, i - 1)
                        ) + 1
                    )
        # 记入备忘录
        memo[(K, N)] = res
        return res

    return dp(K, N)
```

这个算法的时间复杂度是多少呢？**动态规划算法的时间复杂度就是子问题个数 × 函数本身的复杂度**。

函数本身的复杂度就是忽略递归部分的复杂度，这里 `dp` 函数中有一个 for 循环，所以函数本身的复杂度是 O(N)。

子问题个数也就是不同状态组合的总数，显然是两个状态的乘积，也就是 O(KN)。

所以算法的总时间复杂度是 O(K*N^2), 空间复杂度 O(KN)。

三、疑难解答

这个问题很复杂，但是算法代码却十分简洁，这就是动态规划的特性，穷举加备忘录/DP table 优化，真的没啥新意。

首先，有读者可能不理解代码中为什么用一个 for 循环遍历楼层 `[1..N]`，也许会把这个逻辑和之前探讨的线性扫描混为一谈。其实不是的，**这只是在做一次「选择」**。

比方说你有 2 个鸡蛋，面对 10 层楼，你**这次**选择去哪一层楼扔呢？不知道，那就把这 10 层楼全试一遍。至于下次怎么选择不用你操心，有正确的状态转移，递归会算出每个选择的代价，我们取最优的那个就是最优解。

另外，这个问题还有更好的解法，比如修改代码中的 for 循环为二分搜索，可以将时间复杂度降为 O(K*N*logN)；再改进动态规划解法可以进一步降为 O(KN)；使用数学方法解决，时间复杂度达到最优 O(K*logN)，空间复杂度达到 O(1)。

二分的解法也有点误导性，你很可能以为它跟我们之前讨论的二分思路扔鸡蛋有关系，实际上没有半毛钱关系。能用二分搜索是因为状态转移方程的函数图像具有单调性，可以快速找到最值。

简单介绍一下二分查找的优化吧，其实只是在优化这段代码：

```
def dp(K, N):
    for 1 <= i <= N:
        # 最坏情况下的最少扔鸡蛋次数
        res = min(res,
                  max(
                        dp(K - 1, i - 1), # 碎
                        dp(K, N - i)      # 没碎
                    ) + 1 # 在第 i 楼扔了一次
                )
    return res
```

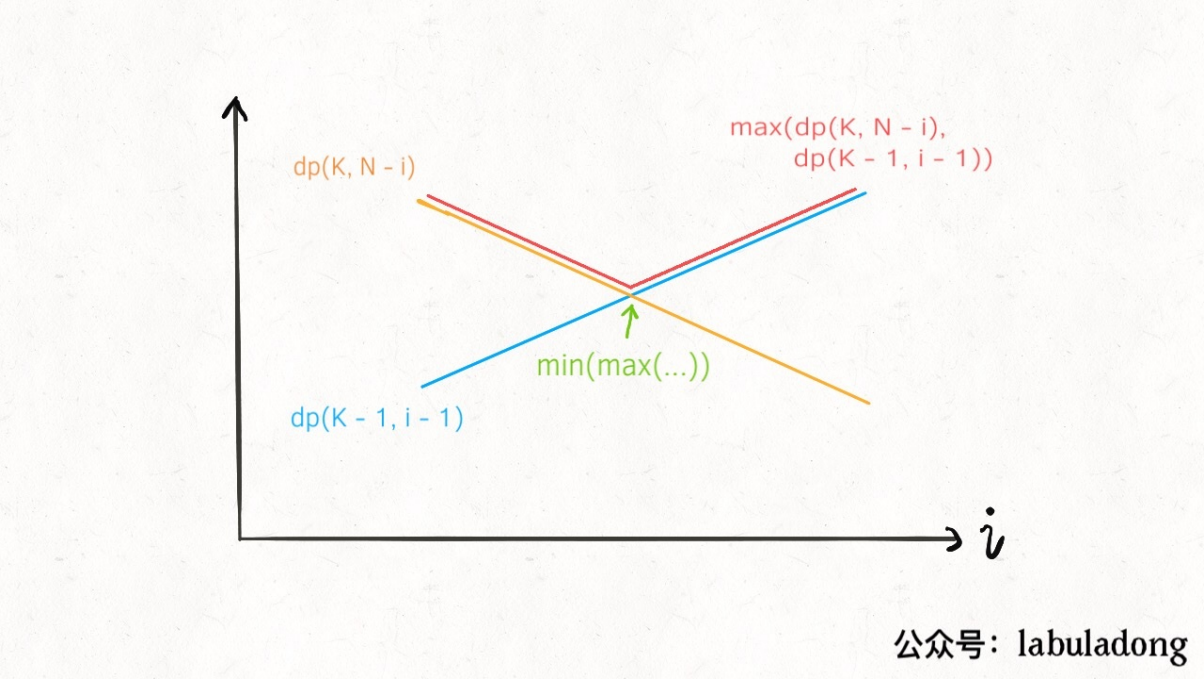
这个 for 循环就是下面这个状态转移方程的具体代码实现：

$$ dp(K, N) = \min_{0 <= i <= N}{\max{dp(K - 1, i - 1), dp(K, N - i)} + 1}$$

首先我们根据 `dp(K, N)` 数组的定义（有 `K` 个鸡蛋面对 `N` 层楼，最少需要扔几次），**很容易知道 `K` 固定时，这个函数一定是单调递增的**，无论你策略多聪明，楼层增加测试次数一定要增加。

那么注意 `dp(K - 1, i - 1)` 和 `dp(K, N - i)` 这两个函数，其中 `i` 是从 1 到 `N` 单增的，如果我们固定 `K` 和 `N`，**把这两个函数看做关于 `i` 的函数，前者随着 `i` 的增加应该也是单调递增的，而后者随着 `i` 的增加应该**

是单调递减的：



这时候求二者的较大值，再求这些最大值之中的最小值，其实就是求这个交点嘛，熟悉二分搜索的同学肯定敏感地想到了，这不就是相当于求 Valley（山谷）值嘛，可以用二分查找来快速寻找这个点的。

直接贴一下代码吧，思路还是完全一样的：

```
def superEggDrop(self, K: int, N: int) -> int:

    memo = dict()
    def dp(K, N):
        if K == 1: return N
        if N == 0: return 0
        if (K, N) in memo:
            return memo[(K, N)]

        # for 1 <= i <= N:
        #     res = min(res,
        #             max(
    #                     dp(K - 1, i - 1),
    #                     dp(K, N - i)
        #                 ) + 1
        #             )
```

```
        res = float('INF')
        # 用二分搜索代替线性搜索
        lo, hi = 1, N
        while lo <= hi:
            mid = (lo + hi) // 2
            broken = dp(K - 1, mid - 1) # 碎
            not_broken = dp(K, N - mid) # 没碎
            # res = min(max(碎, 没碎) + 1)
            if broken > not_broken:
                hi = mid - 1
                res = min(res, broken + 1)
            else:
                lo = mid + 1
                res = min(res, not_broken + 1)

        memo[(K, N)] = res
        return res

    return dp(K, N)
```

这里就不展开其他解法了，留在下一篇文章 高楼扔鸡蛋进阶

我觉得吧，我们这种解法就够了：找状态，做选择，足够清晰易懂，可流程化，可举一反三。掌握这套框架学有余力的话，再去考虑那些奇技淫巧也不迟。

最后预告一下，《动态规划详解（修订版）》和《回溯算法详解（修订版）》已经动笔了，教大家用模板的力量来对抗变化无穷的算法题，敬请期待。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

经典动态规划问题：高楼扔鸡蛋（进阶）

上篇文章聊了高楼扔鸡蛋问题，讲了一种效率不是很高，但是较为容易理解的动态规划解法。后台很多读者问如何更高效地解决这个问题，今天就谈两种思路，来优化一下这个问题，分别是二分查找优化和重新定义状态转移。

如果还不知道高楼扔鸡蛋问题的读者可以看下「经典动态规划：高楼扔鸡蛋」，那篇文章详解了题目的含义和基本的动态规划解题思路，请确保理解前文，因为今天的优化都是基于这个基本解法的。

二分搜索的优化思路也许是我们可以尽力尝试写出的，而修改状态转移的解法可能是不容易想到的，可以借此见识一下动态规划算法设计的玄妙，当做思维拓展。

二分搜索优化

之前提到过这个解法，核心是因为状态转移方程的单调性，这里可以具体展开看看。

首先简述一下原始动态规划的思路：

1、暴力穷举尝试在所有楼层 `1 <= i <= N` 扔鸡蛋，每次选择尝试次数**最少**的那一层；

2、每次扔鸡蛋有两种可能，要么碎，要么没碎；

3、如果鸡蛋碎了，`F` 应该在第 `i` 层下面，否则，`F` 应该在第 `i` 层上面；

4、鸡蛋是碎了还是没碎，取决于哪种情况下尝试次数**更多**，因为我们想求的是最坏情况下的结果。

核心的状态转移代码是这段：

```
# 当前状态为 K 个鸡蛋，面对 N 层楼
# 返回这个状态下的最优结果
def dp(K, N):
    for 1 <= i <= N:
        # 最坏情况下的最少扔鸡蛋次数
        res = min(res,
                  max(
                        dp(K - 1, i - 1), # 碎
                        dp(K, N - i)      # 没碎
                    ) + 1 # 在第 i 楼扔了一次
                )
    return res
```

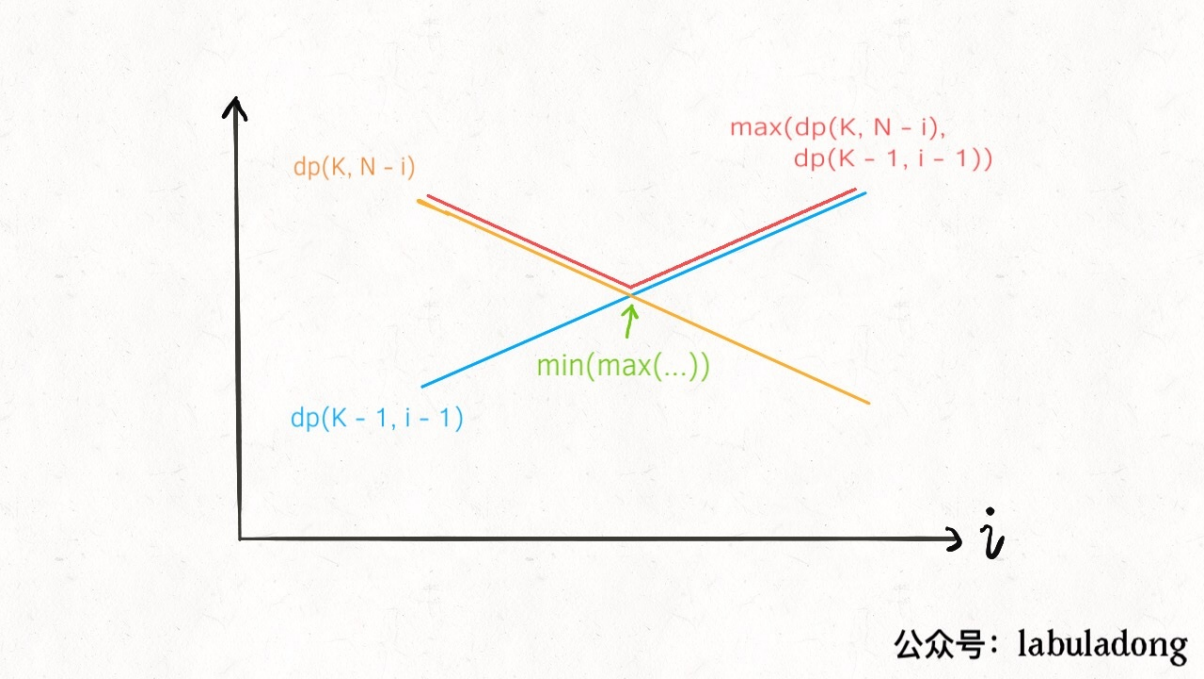
这个 for 循环就是下面这个状态转移方程的具体代码实现：

$$dp(K,N) = \min_{0<=i<=N}\{\max\{dp(K-1,i-1), dp(K,N-i)\}+1\}$$

如果能够理解这个状态转移方程，那么就很容易理解二分查找的优化思路。

首先我们根据 `dp(K, N)` 数组的定义（有 `K` 个鸡蛋面对 `N` 层楼，最少需要扔几次），**很容易知道 `K` 固定时，这个函数随着 `N` 的增加一定是单调递增的**，无论你策略多聪明，楼层增加测试次数一定要增加。

那么注意 `dp(K - 1, i - 1)` 和 `dp(K, N - i)` 这两个函数，其中 `i` 是从 1 到 `N` 单增的，如果我们固定 `K` 和 `N`，**把这两个函数看做关于 `i` 的函数，前者随着 `i` 的增加应该也是单调递增的，而后者随着 `i` 的增加应该是单调递减的**：



这时候求二者的较大值，再求这些最大值之中的最小值，其实就是求这两条直线交点，也就是红色折线的最低点嘛。

我们前文「二分查找只能用来查找元素吗」讲过，二分查找的运用很广泛，形如下面这种形式的 for 循环代码：

```
for (int i = 0; i < n; i++) {
    if (isOK(i))
        return i;
}
```

都很有可能可以运用二分查找来优化线性搜索的复杂度，回顾这两个 `dp` 函数的曲线，我们要找的最低点其实就是这种情况：

```
for (int i = 1; i <= N; i++) {
    if (dp(K - 1, i - 1) == dp(K, N - i))
        return dp(K, N - i);
}
```

熟悉二分搜索的同学肯定敏感地想到了，这不就是相当于求 Valley（山谷）值嘛，可以用二分查找来快速寻找这个点的，直接看代码吧，整体的思路还是一样，只是加快了搜索速度：

```python
def superEggDrop(self, K: int, N: int) -> int:

    memo = dict()
    def dp(K, N):
        if K == 1: return N
        if N == 0: return 0
        if (K, N) in memo:
            return memo[(K, N)]

        # for 1 <= i <= N:
        #     res = min(res,
        #             max(
        #                 dp(K - 1, i - 1),
        #                 dp(K, N - i)
        #                 ) + 1
        #             )

        res = float('INF')
        # 用二分搜索代替线性搜索
        lo, hi = 1, N
        while lo <= hi:
            mid = (lo + hi) // 2
            broken = dp(K - 1, mid - 1) # 碎
            not_broken = dp(K, N - mid) # 没碎
            # res = min(max(碎, 没碎) + 1)
            if broken > not_broken:
                hi = mid - 1
                res = min(res, broken + 1)
            else:
                lo = mid + 1
                res = min(res, not_broken + 1)

        memo[(K, N)] = res
        return res

    return dp(K, N)
```

这个算法的时间复杂度是多少呢？**动态规划算法的时间复杂度就是子问题个数 × 函数本身的复杂度**。

函数本身的复杂度就是忽略递归部分的复杂度，这里 `dp` 函数中用了一个二分搜索，所以函数本身的复杂度是 O(logN)。

子问题个数也就是不同状态组合的总数，显然是两个状态的乘积，也就是 O(KN)。

所以算法的总时间复杂度是 O(K*N*logN), 空间复杂度 O(KN)。效率上比之前的算法 O(KN^2) 要高效一些。

重新定义状态转移

前文「不同定义有不同解法」就提过，找动态规划的状态转移本就是见仁见智，比较玄学的事情，不同的状态定义可以衍生出不同的解法，其解法和复杂程度都可能有巨大差异。这里就是一个很好的例子。

再回顾一下我们之前定义的 `dp` 数组含义：

```
def dp(k, n) -> int
# 当前状态为 k 个鸡蛋，面对 n 层楼
# 返回这个状态下最少的扔鸡蛋次数
```

用 dp 数组表示的话也是一样的：

```
dp[k][n] = m
# 当前状态为 k 个鸡蛋，面对 n 层楼
# 这个状态下最少的扔鸡蛋次数为 m
```

按照这个定义，就是**确定当前的鸡蛋个数和面对的楼层数，就知道最小扔鸡蛋次数**。最终我们想要的答案就是 `dp(K, N)` 的结果。

这种思路下，肯定要穷举所有可能的扔法的，用二分搜索优化也只是做了「剪枝」，减小了搜索空间，但本质思路没有变，还是穷举。

现在，我们稍微修改 `dp` 数组的定义，**确定当前的鸡蛋个数和最多允许的扔鸡蛋次数，就知道能够确定 `F` 的最高楼层数**。具体来说是这个意思：

```
dp[k][m] = n
# 当前有 k 个鸡蛋，可以尝试扔 m 次鸡蛋
# 这个状态下，最坏情况下最多能确切测试一栋 n 层的楼

# 比如说 dp[1][7] = 7 表示：
# 现在有 1 个鸡蛋，允许你扔 7 次；
# 这个状态下最多给你 7 层楼，
# 使得你可以确定楼层 F 使得鸡蛋恰好摔不碎
# （一层一层线性探查嘛）
```

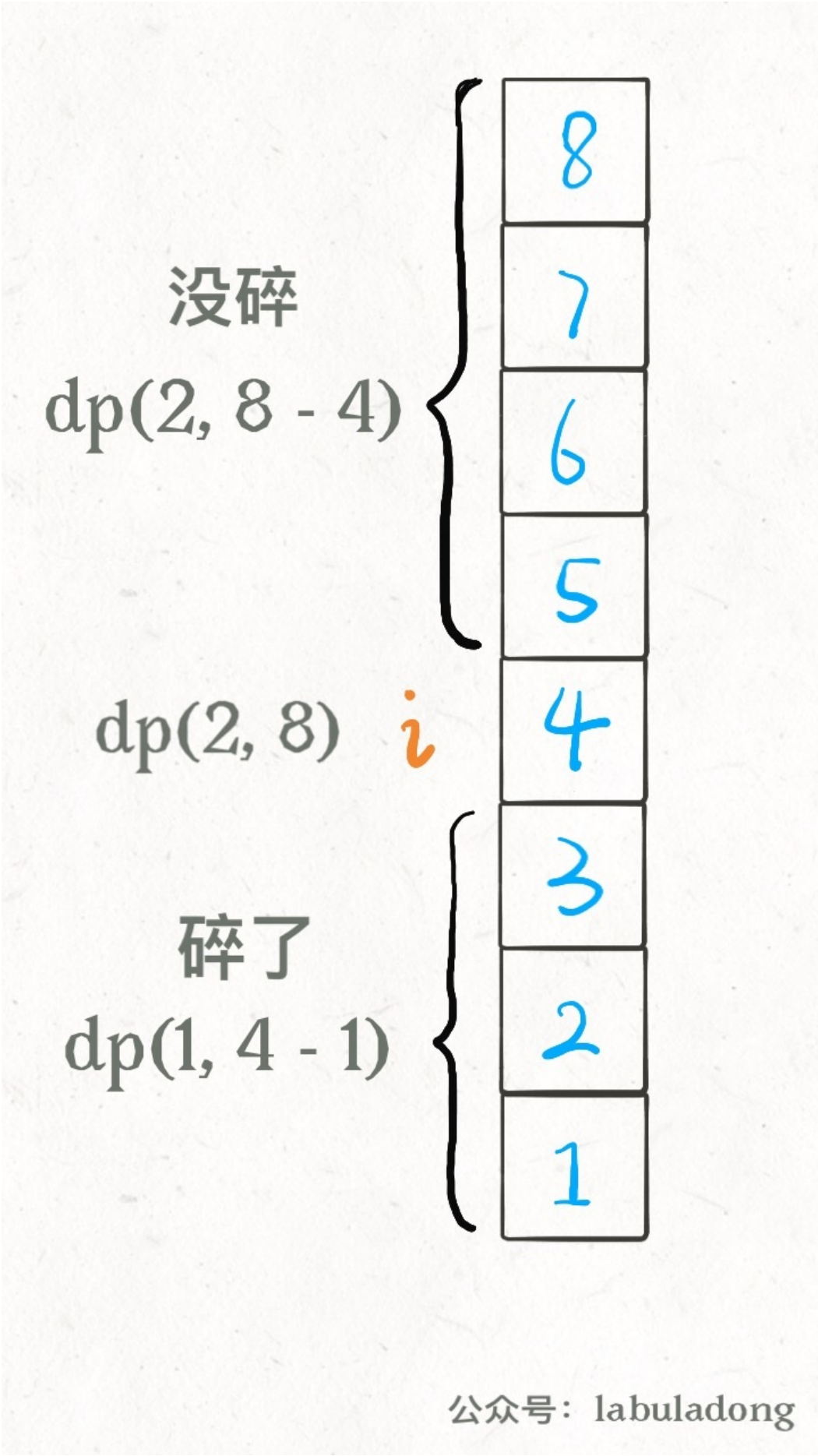
这其实就是我们原始思路的一个「反向」版本，我们先不管这种思路的状态转移怎么写，先来思考一下这种定义之下，最终想求的答案是什么？

我们最终要求的其实是扔鸡蛋次数 `m`，但是这时候 `m` 在状态之中而不是 `dp` 数组的结果，可以这样处理：

```
int superEggDrop(int K, int N) {

    int m = 0;
    while (dp[K][m] < N) {
        m++;
        // 状态转移...
    }
    return m;
}
```

题目不是**给你 `K` 鸡蛋，`N` 层楼，让你求最坏情况下最少的测试次数 `m`** 吗？`while` 循环结束的条件是 `dp[K][m] == N`，也就是**给你 `K` 个鸡蛋，测试 `m` 次，最坏情况下最多能测试 `N` 层楼**。

注意看这两段描述，是完全一样的！所以说这样组织代码是正确的，关键就是状态转移方程怎么找呢？还得从我们原始的思路开始讲。之前的解法配了这样图帮助大家理解状态转移思路：


没碎
dp(2, 8 - 4)
8
7
6
5
dp(2, 8)
i
4
碎了
dp(1, 4 - 1)
3
2
1
公众号：labuladong

这个图描述的仅仅是某一个楼层 `i`，原始解法还得线性或者二分扫描所有楼层，要求最大值、最小值。但是现在这种 `dp` 定义根本不需要这些了，基于下面两个事实：

1、无论你在哪层楼扔鸡蛋，鸡蛋只可能摔碎或者没摔碎，碎了的话就测楼下，没碎的话就测楼上。

2、无论你上楼还是下楼，总的楼层数 = 楼上的楼层数 + 楼下的楼层数 + 1（当前这层楼）。

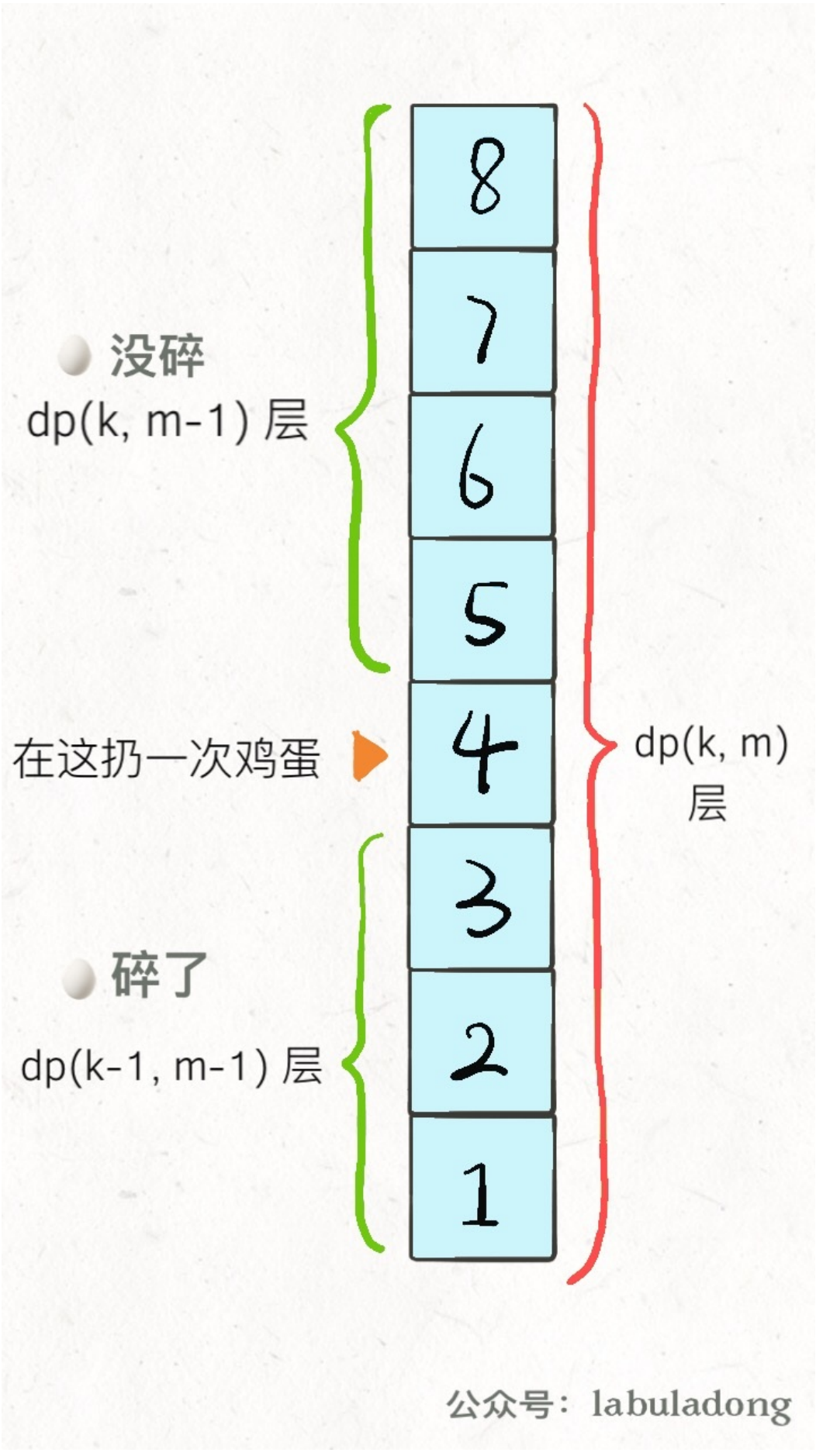
根据这个特点，可以写出下面的状态转移方程：

```
dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1
```

`dp[k][m - 1]` 就是楼上的楼层数，因为鸡蛋个数 `k` 不变，也就是鸡蛋没碎，扔鸡蛋次数 `m` 减一；

`dp[k - 1][m - 1]` 就是楼下的楼层数，因为鸡蛋个数 `k` 减一，也就是鸡蛋碎了，同时扔鸡蛋次数 `m` 减一。

PS：这个 `m` 为什么要减一而不是加一？之前定义得很清楚，这个 `m` 是一个允许的次数上界，而不是扔了几次。


8
7
6
5
4
3
2
1
没碎
dp(k, m-1) 层
在这扔一次鸡蛋
碎了
dp(k-1, m-1) 层
dp(k, m)
层
公众号：labuladong

至此，整个思路就完成了，只要把状态转移方程填进框架即可：

```
int superEggDrop(int K, int N) {
    // m 最多不会超过 N 次（线性扫描）
    int[][] dp = new int[K + 1][N + 1];
    // base case:
    // dp[0][..] = 0
    // dp[..][0] = 0
    // Java 默认初始化数组都为 0
    int m = 0;
    while (dp[K][m] < N) {
        m++;
        for (int k = 1; k <= K; k++)
            dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1;
    }
    return m;
}
```

如果你还觉得这段代码有点难以理解，其实它就等同于这样写：

```
for (int m = 1; dp[K][m] < N; m++)
    for (int k = 1; k <= K; k++)
        dp[k][m] = dp[k][m - 1] + dp[k - 1][m - 1] + 1;
```

看到这种代码形式就熟悉多了吧，因为我们要求的不是 `dp` 数组里的值，而是某个符合条件的索引 `m`，所以用 `while` 循环来找到这个 `m` 而已。

这个算法的时间复杂度是多少？很明显就是两个嵌套循环的复杂度 O(KN)。

另外注意到 `dp[m][k]` 转移只和左边和左上的两个状态有关，所以很容易优化成一维 `dp` 数组，这里就不写了。

还可以再优化

再往下就要用一些数学方法了，不具体展开，就简单提一下思路吧。

在刚才的思路之上，**注意函数 `dp(m, k)` 是随着 `m` 单增的，因为鸡蛋个数 `k` 不变时，允许的测试次数越多，可测试的楼层就越高**。

这里又可以借助二分搜索算法快速逼近 `dp[K][m] == N` 这个终止条件，时间复杂度进一步下降为 O(KlogN)，我们可以设 `g(k, m) =` ……

算了算了，打住吧。我觉得我们能够写出 O(K*N*logN) 的二分优化算法就行了，后面的这些解法呢，听个响鼓个掌就行了，把欲望限制在能力的范围之内才能拥有快乐！

不过可以肯定的是，根据二分搜索代替线性扫描 `m` 的取值，代码的大致框架肯定是修改穷举 `m` 的 for 循环：

```
// 把线性搜索改成二分搜索
// for (int m = 1; dp[K][m] < N; m++)
int lo = 1, hi = N;
while (lo < hi) {
    int mid = (lo + hi) / 2;
    if (... < N) {
        lo = ...
    } else {
        hi = ...
    }

    for (int k = 1; k <= K; k++)
        // 状态转移方程
}
```

简单总结一下吧，第一个二分优化是利用了 `dp` 函数的单调性，用二分查找技巧快速搜索答案；第二种优化是巧妙地修改了状态转移方程，简化了求解了流程，但相应的，解题逻辑比较难以想到；后续还可以用一些数学方法和二分搜索进一步优化第二种解法，不过看了看镜子中的发量，算了。

本文终，希望对你有一点启发。

动态规划之子序列问题解题模板

子序列问题是常见的算法问题，而且并不好解决。

首先，子序列问题本身就相对子串、子数组更困难一些，因为前者是不连续的序列，而后两者是连续的，就算穷举你都不一定会，更别说求解相关的算法问题了。

而且，子序列问题很可能涉及到两个字符串，比如前文「最长公共子序列」，如果没有一定的处理经验，真的不容易想出来。所以本文就来扒一扒子序列问题的套路，其实就有两种模板，相关问题只要往这两种思路上想，十拿九稳。

一般来说，这类问题都是让你求一个**最长子序列**，因为最短子序列就是一个字符嘛，没啥可问的。一旦涉及到子序列和最值，那几乎可以肯定，**考察的是动态规划技巧，时间复杂度一般都是 O(n^2)**。

原因很简单，你想想一个字符串，它的子序列有多少种可能？起码是指数级的吧，这种情况下，不用动态规划技巧，还想怎么着？

既然要用动态规划，那就要定义 dp 数组，找状态转移关系。我们说的两种思路模板，就是 dp 数组的定义思路。不同的问题可能需要不同的 dp 数组定义来解决。

一、两种思路

1、第一种思路模板是一个一维的 dp 数组：

```
int n = array.length;
int[] dp = new int[n];

for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
        dp[i] = 最值(dp[i], dp[j] + ...)
```

```
    }
}
```

举个我们写过的例子「最长递增子序列」，在这个思路中 dp 数组的定义是：

在子数组 `array[0..i]` 中，我们要求的子序列（最长递增子序列）的长度是 `dp[i]`。

为啥最长递增子序列需要这种思路呢？前文说得很清楚了，因为这样符合归纳法，可以找到状态转移的关系，这里就不具体展开了。

2、第二种思路模板是一个二维的 dp 数组：

```
int n = arr.length;
int[][] dp = new dp[n][n];

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (arr[i] == arr[j])
            dp[i][j] = dp[i][j] + ...
        else
            dp[i][j] = 最值(...)
    }
}
```

这种思路运用相对更多一些，尤其是涉及两个字符串/数组的子序列，比如前文讲的「最长公共子序列」。本思路中 dp 数组含义又分为「只涉及一个字符串」和「涉及两个字符串」两种情况。

2.1 涉及两个字符串/数组时（比如最长公共子序列），dp 数组的含义如下：

在子数组 `arr1[0..i]` 和子数组 `arr2[0..j]` 中，我们要求的子序列（最长公共子序列）长度为 `dp[i][j]`。

2.2 只涉及一个字符串/数组时（比如本文要讲的最长回文子序列），dp 数组的含义如下：

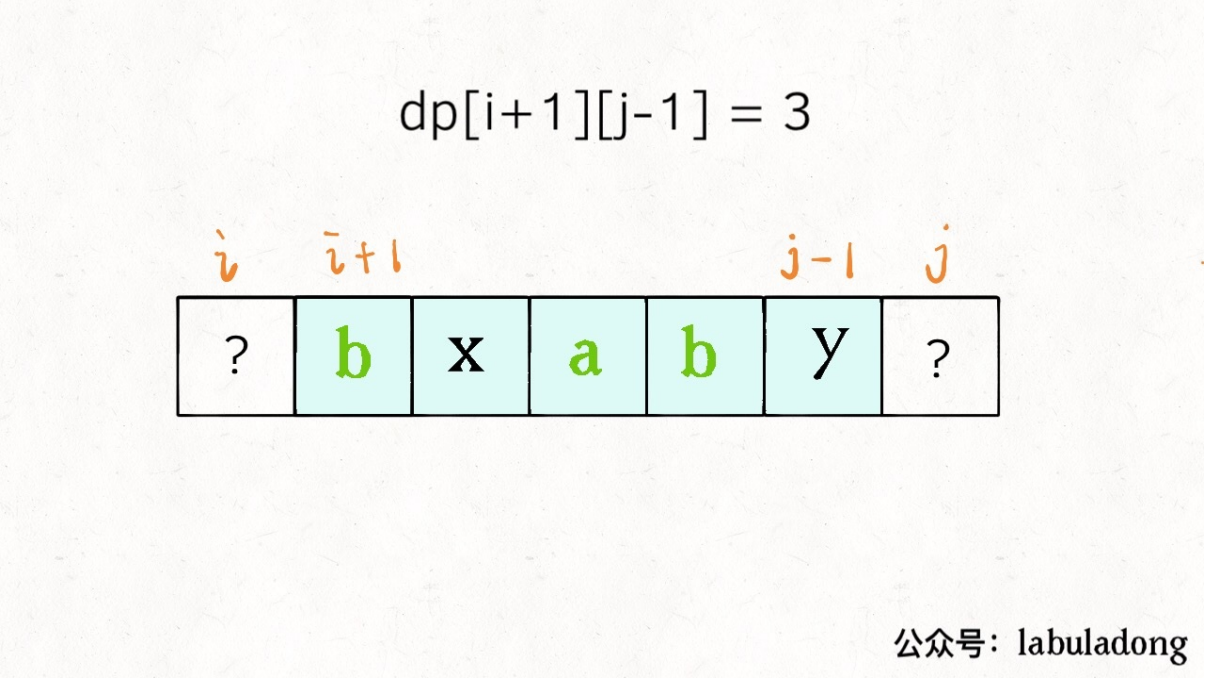
在子数组 `array[i..j]` 中，我们要求的子序列（最长回文子序列）的长度为 `dp[i][j]` 。

第一种情况可以参考这两篇旧文：「编辑距离」「公共子序列」

下面就借最长回文子序列这个问题，详解一下第二种情况下如何使用动态规划。

二、最长回文子序列

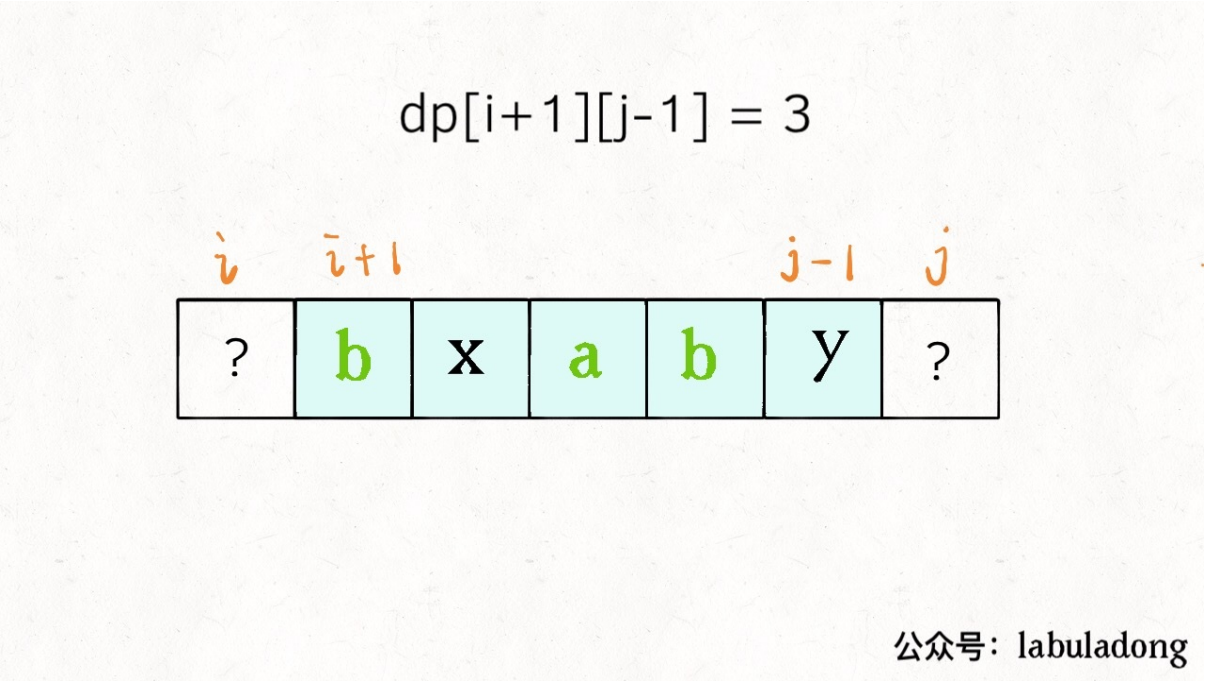
之前解决了「最长回文子串」的问题，这次提升难度，求最长回文子序列的长度：



我们说这个问题对 dp 数组的定义是：**在子串 `s[i..j]` 中，最长回文子序列的长度为 `dp[i][j]`** 。一定要记住这个定义才能理解算法。

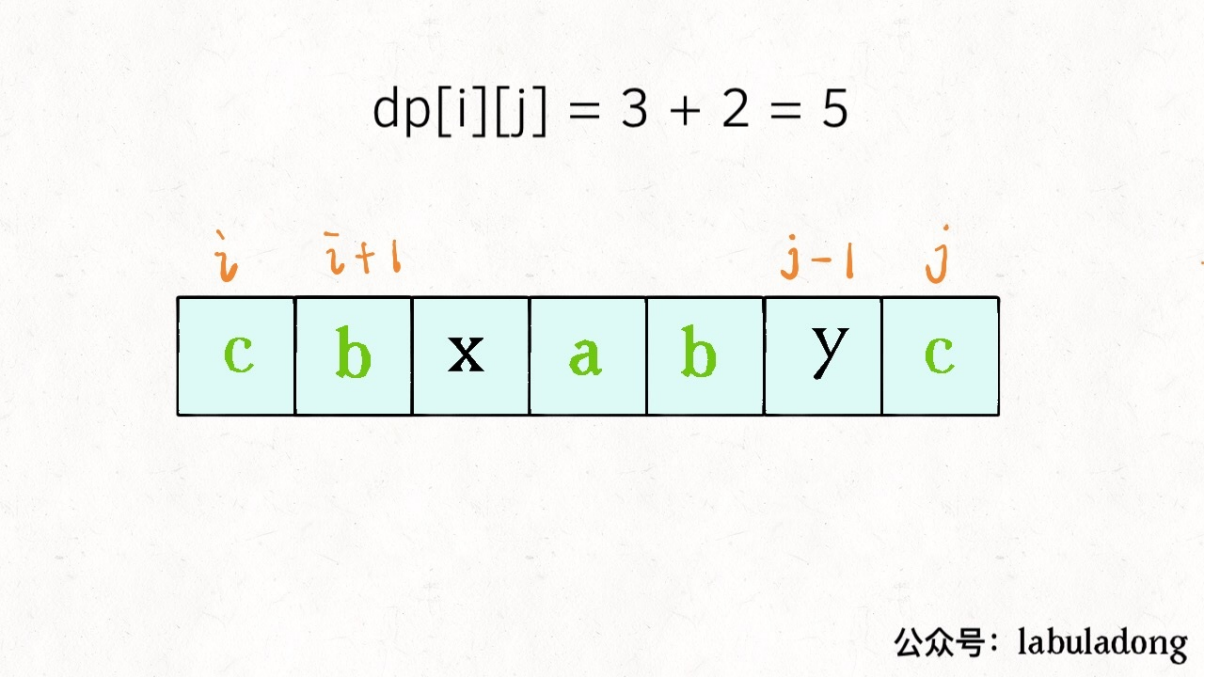
为啥这个问题要这样定义二维的 dp 数组呢？我们前文多次提到，**找状态转移需要归纳思维，说白了就是如何从已知的结果推出未知的部分**，这样定义容易归纳，容易发现状态转移关系。

具体来说，如果我们想求 `dp[i][j]`，假设你知道了子问题 `dp[i+1][j-1]` 的结果（`s[i+1..j-1]` 中最长回文子序列的长度），你是否能想办法算出 `dp[i][j]` 的值（`s[i..j]` 中，最长回文子序列的长度）呢？

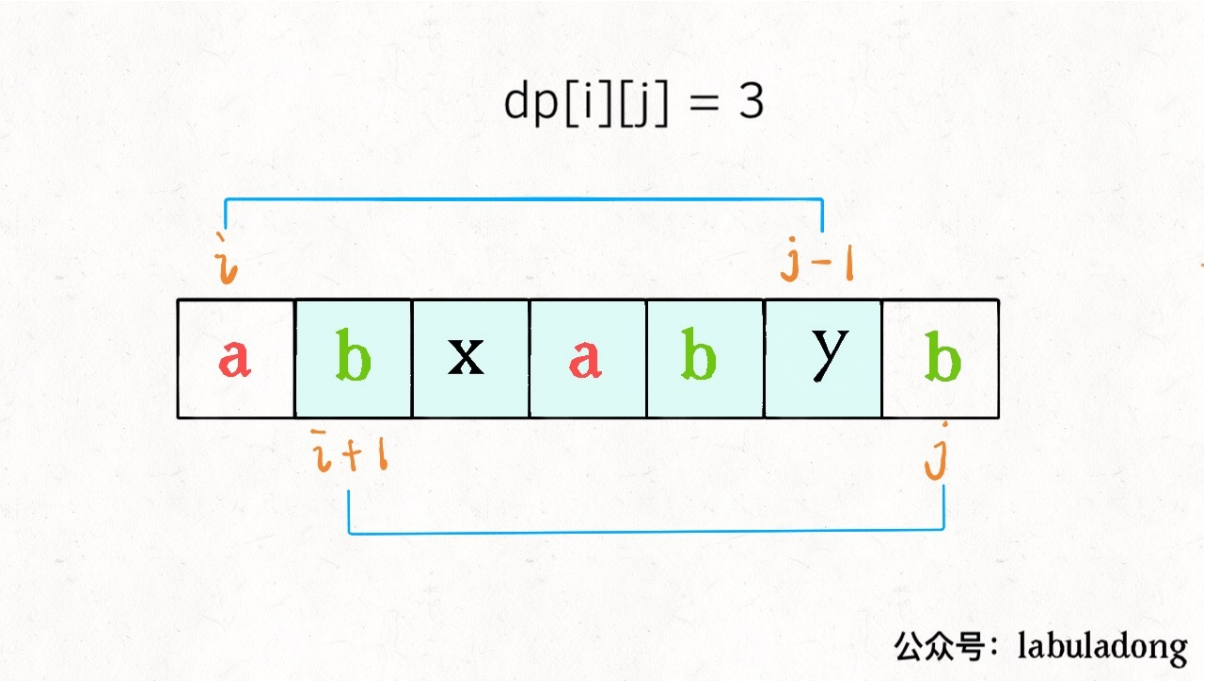


可以！这取决于 `s[i]` 和 `s[j]` 的字符：

如果它俩相等，那么它俩加上 `s[i+1..j-1]` 中的最长回文子序列就是 `s[i..j]` 的最长回文子序列：



如果它俩不相等，说明它俩**不可能同时**出现在 `s[i..j]` 的最长回文子序列中，那么把它俩**分别**加入 `s[i+1..j-1]` 中，看看哪个子串产生的回文子序列更长即可：



以上两种情况写成代码就是这样：

```
if (s[i] == s[j])
    // 它俩一定在最长回文子序列中
    dp[i][j] = dp[i + 1][j - 1] + 2;
else
    // s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长？
    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
```

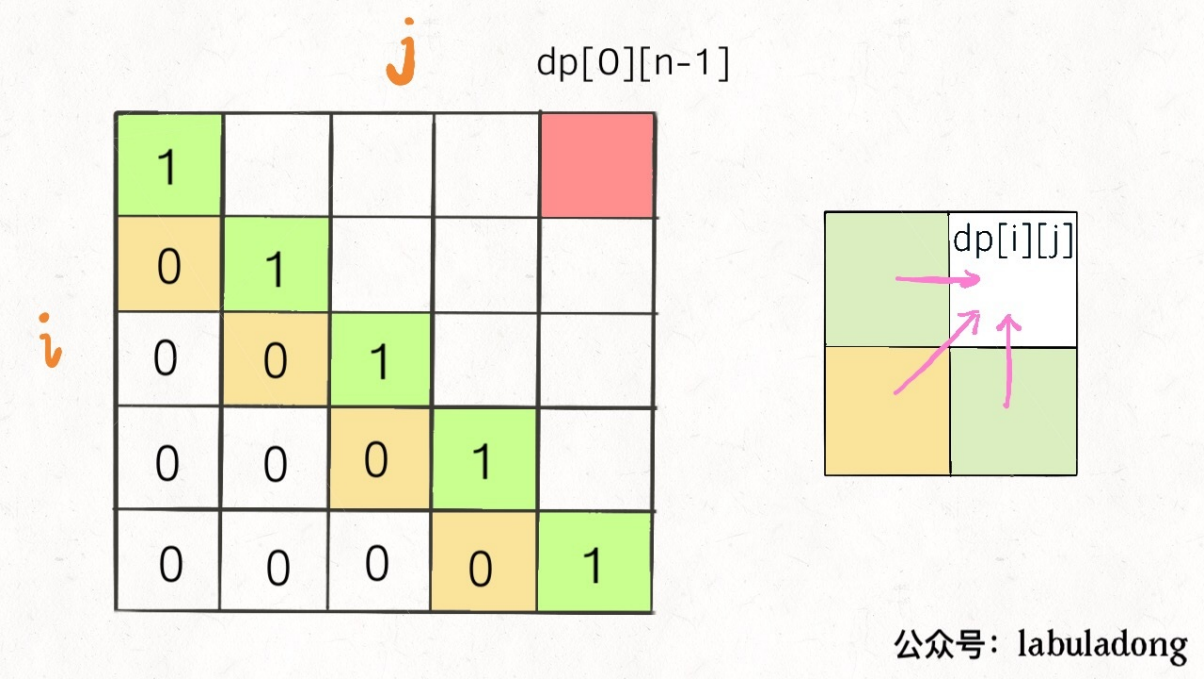
至此，状态转移方程就写出来了，根据 dp 数组的定义，我们要求的就是 `dp[0][n - 1]`，也就是整个 `s` 的最长回文子序列的长度。

三、代码实现

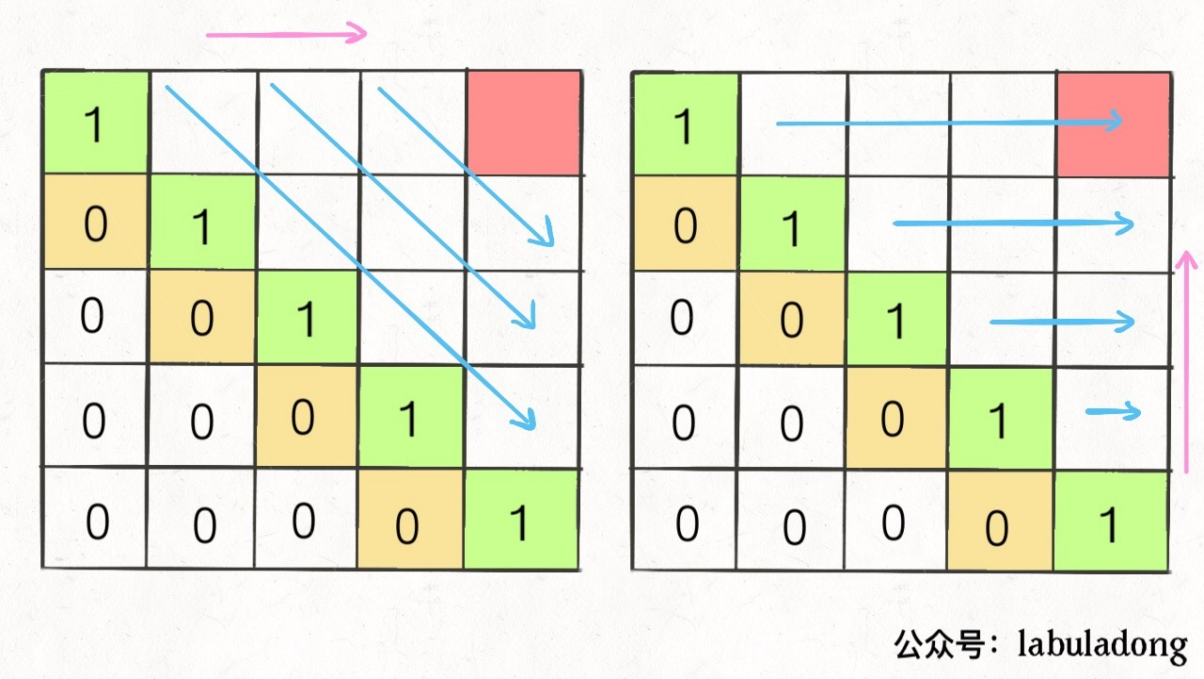
首先明确一下 base case，如果只有一个字符，显然最长回文子序列长度是 1，也就是 `dp[i][j] = 1 (i == j)`。

因为 `i` 肯定小于等于 `j`，所以对于那些 `i > j` 的位置，根本不存在什么子序列，应该初始化为 0。

另外，看看刚才写的状态转移方程，想求 `dp[i][j]` 需要知道 `dp[i+1][j-1]`，`dp[i+1][j]`，`dp[i][j-1]` 这三个位置；再看看我们确定的 base case，填入 dp 数组之后是这样：



为了保证每次计算 `dp[i][j]`，左下右方向的位置已经被计算出来，只能斜着遍历或者反着遍历：



我选择反着遍历，代码如下：

```
int longestPalindromeSubseq(string s) {
    int n = s.size();
    // dp 数组全部初始化为 0
    vector<vector<int>> dp(n, vector<int>(n, 0));
    // base case
    for (int i = 0; i < n; i++)
        dp[i][i] = 1;
    // 反着遍历保证正确的状态转移
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            // 状态转移方程
            if (s[i] == s[j])
                dp[i][j] = dp[i + 1][j - 1] + 2;
            else
                dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
        }
    }
    // 整个 s 的最长回文子串长度
    return dp[0][n - 1];
}
```

至此，最长回文子序列的问题就解决了。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

动态规划之博弈问题

上一篇文章 几道智力题 中讨论到一个有趣的「石头游戏」，通过题目的限制条件，这个游戏是先手必胜的。但是智力题终究是智力题，真正的算法问题肯定不会是投机取巧能搞定的。所以，本文就借石头游戏来讲讲「假设两个人都足够聪明，最后谁会获胜」这一类问题该如何用动态规划算法解决。

博弈类问题的套路都差不多，下文举例讲解，其核心思路是在二维 dp 的基础上使用元组分别存储两个人的博弈结果。掌握了这个技巧以后，别人再问你什么俩海盗分宝石，俩人拿硬币的问题，你就告诉别人：我懒得想，直接给你写个算法算一下得了。

我们「石头游戏」改的更具有一般性：

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

石头的堆数可以是任意正整数，石头的总数也可以是任意正整数，这样就能打破先手必胜的局面了。比如有三堆石头 `piles = [1, 100, 3]`，先手不管拿 1 还是 3，能够决定胜负的 100 都会被后手拿走，后手会获胜。

假设两人都很聪明，请你设计一个算法，返回先手和后手的最后得分（石头总数）之差。比如上面那个例子，先手能获得 4 分，后手会获得 100 分，你的算法应该返回 -96。

这样推广之后，这个问题算是一道 Hard 的动态规划问题了。**博弈问题的难点在于，两个人要轮流进行选择，而且都贼精明，应该如何编程表示这个过程呢？**

还是强调多次的套路，首先明确 dp 数组的含义，然后和股票买卖系列问题类似，只要找到「状态」和「选择」，一切就水到渠成了。

一、定义 dp 数组的含义

定义 dp 数组的含义是很有技术含量的，同一问题可能有多种定义方法，不同的定义会引出不同的状态转移方程，不过只要逻辑没有问题，最终都能得到相同的答案。

我建议不要迷恋那些看起来很牛逼，代码很短小的奇技淫巧，最好是稳一点，采取可解释性最好，最容易推广的设计思路。本文就给出一种博弈问题的通用设计框架。

介绍 dp 数组的含义之前，我们先看一下 dp 数组最终的样子：

piles = [3, 9, 1, 2]

start \ end	0	1	2	3
0	(3, 0)	(9, 3)	(4, 9)	(11, 4)
1		(9, 0)	(9, 1)	(10, 2)
2			(1, 0)	(2, 1)
3				(2, 0)

下文讲解时，认为元组是包含 first 和 second 属性的一个类，而且为了节省篇幅，将这两个属性简写为 fir 和 sec。比如按上图的数据，我们说 `dp[1][3].fir = 10`，`dp[0][1].sec = 3`。

先回答几个读者可能提出的问题：

这个二维 dp table 中存储的是元组，怎么编程表示呢？这个 dp table 有一半根本没用上，怎么优化？很简单，都不要管，先把解题的思路想明白了再谈也不迟。

以下是对 dp 数组含义的解释：

```
dp[i][j].fir 表示，对于 piles[i...j] 这部分石头堆，先手能获得的最高分数。
dp[i][j].sec 表示，对于 piles[i...j] 这部分石头堆，后手能获得的最高分数。

举例理解一下，假设 piles = [3, 9, 1, 2]，索引从 0 开始
dp[0][1].fir = 9 意味着：面对石头堆 [3, 9]，先手最终能够获得 9 分。
dp[1][3].sec = 2 意味着：面对石头堆 [9, 1, 2]，后手最终能够获得 2 分。
```

我们想求的答案是先手和后手最终分数之差，按照这个定义也就是 $dp[0][n-1].fir - dp[0][n-1].sec$，即面对整个 piles，先手的最优得分和后手的最优得分之差。

二、状态转移方程

写状态转移方程很简单，首先要找到所有「状态」和每个状态可以做的「选择」，然后择优。

根据前面对 dp 数组的定义，**状态显然有三个：开始的索引 i，结束的索引 j，当前轮到的人**。

```
dp[i][j][fir or sec]
其中：
0 <= i < piles.length
i <= j < piles.length
```

对于这个问题的每个状态，可以做的**选择有两个：选择最左边的那堆石头，或者选择最右边的那堆石头**。我们可以这样穷举所有状态：

```
n = piles.length
for 0 <= i < n:
    for j <= i < n:
```

```
            for who in {fir, sec}:
                dp[i][j][who] = max(left, right)
```

上面的伪码是动态规划的一个大致的框架，股票系列问题中也有类似的伪码。这道题的难点在于，两人是交替进行选择的，也就是说先手的选择会对后手有影响，这怎么表达出来呢？

根据我们对 dp 数组的定义，很容易解决这个难点，**写出状态转移方程：**

```
dp[i][j].fir = max(piles[i] + dp[i+1][j].sec, piles[j] + dp[i][j-1].s
ec)
dp[i][j].fir = max(     选择最左边的石头堆      ,      选择最右边的石头堆      )
# 解释：我作为先手，面对 piles[i...j] 时，有两种选择：
# 要么我选择最左边的那一堆石头，然后面对 piles[i+1...j]
# 但是此时轮到对方，相当于我变成了后手；
# 要么我选择最右边的那一堆石头，然后面对 piles[i...j-1]
# 但是此时轮到对方，相当于我变成了后手。

if 先手选择左边:
    dp[i][j].sec = dp[i+1][j].fir
if 先手选择右边:
    dp[i][j].sec = dp[i][j-1].fir
# 解释：我作为后手，要等先手先选择，有两种情况：
# 如果先手选择了最左边那堆，给我剩下了 piles[i+1...j]
# 此时轮到我，我变成了先手；
# 如果先手选择了最右边那堆，给我剩下了 piles[i...j-1]
# 此时轮到我，我变成了先手。
```

根据 dp 数组的定义，我们也可以找出 **base case**，也就是最简单的情况：

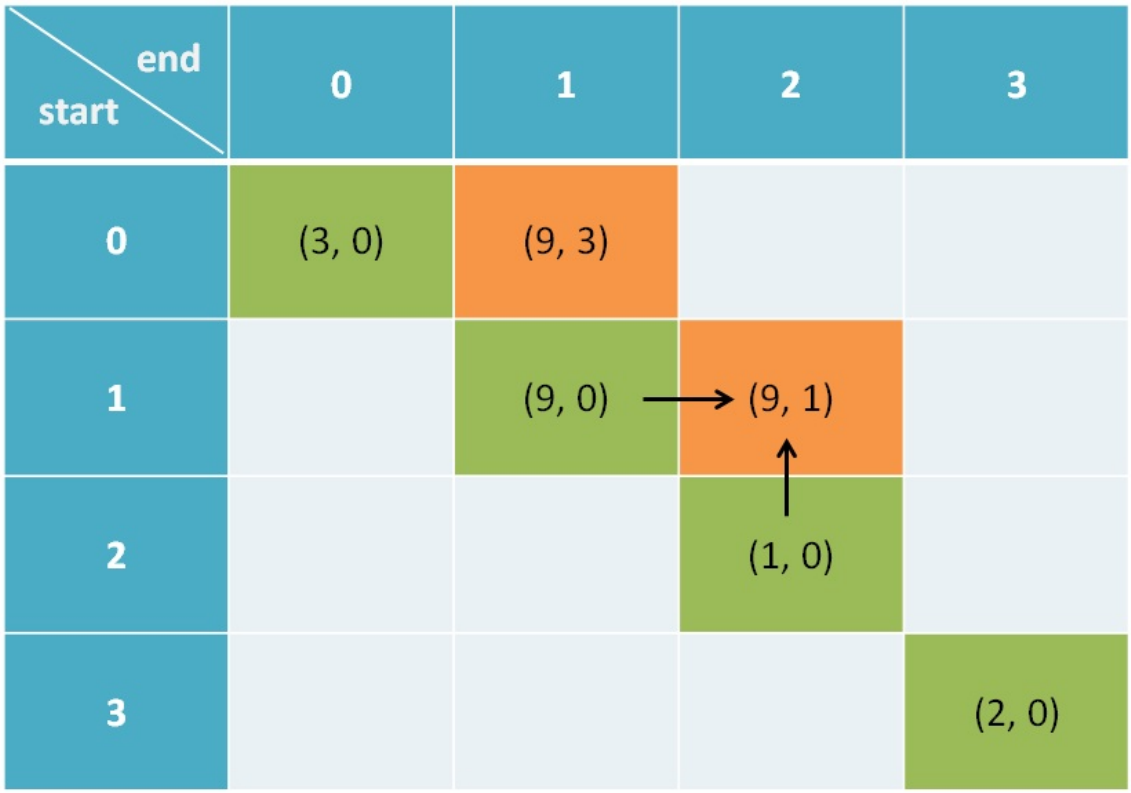
```
dp[i][j].fir = piles[i]
dp[i][j].sec = 0
其中 0 <= i == j < n
# 解释：i 和 j 相等就是说面前只有一堆石头 piles[i]
# 那么显然先手的得分为 piles[i]
# 后手没有石头拿了，得分为 0
```

piles = [3, 9, 1, 2]

start \ end	0	1	2	3
0	(3, 0)			
1		(9, 0)		
2			(1, 0)	
3				(2, 0)

这里需要注意一点，我们发现 base case 是斜着的，而且我们推算 dp[i][j] 时需要用到 dp[i+1][j] 和 dp[i][j-1]：

piles = [3, 9, 1, 2]



所以说算法不能简单的一行一行遍历 dp 数组，**而要斜着遍历数组：**

piles = [3, 9, 1, 2]

start \ end	0	1	2	3
0	(3, 0)	(9, 3)	(4, 9)	(11, 4)
1		(9, 0)	(9, 1)	(10, 2)
2			(1, 0)	(2, 1)
3				(2, 0)

说实话，斜着遍历二维数组说起来容易，你还真不一定能想出来怎么实现，不信你思考一下？这么巧妙的状态转移方程都列出来了，要是不会写代码实现，那真的很尴尬了。

三、代码实现

如何实现这个 fir 和 sec 元组呢，你可以用 python，自带元组类型；或者使用 C++ 的 pair 容器；或者用一个三维数组 `dp[n][n][2]`，最后一个维度就相当于元组；或者我们自己写一个 Pair 类：

```
class Pair {
    int fir, sec;
    Pair(int fir, int sec) {
        this.fir = fir;
        this.sec = sec;
    }
}
```

然后直接把我们的状态转移方程翻译成代码即可，可以注意一下斜着遍历数组的技巧：

```
/* 返回游戏最后先手和后手的得分之差 */
int stoneGame(int[] piles) {
    int n = piles.length;
    // 初始化 dp 数组
    Pair[][] dp = new Pair[n][n];
    for (int i = 0; i < n; i++)
        for (int j = i; j < n; j++)
            dp[i][j] = new Pair(0, 0);
    // 填入 base case
    for (int i = 0; i < n; i++) {
        dp[i][i].fir = piles[i];
        dp[i][i].sec = 0;
    }
    // 斜着遍历数组
    for (int l = 2; l <= n; l++) {
        for (int i = 0; i <= n - l; i++) {
            int j = l + i - 1;
            // 先手选择最左边或最右边的分数
            int left = piles[i] + dp[i+1][j].sec;
            int right = piles[j] + dp[i][j-1].sec;
            // 套用状态转移方程
            if (left > right) {
                dp[i][j].fir = left;
                dp[i][j].sec = dp[i+1][j].fir;
            } else {
                dp[i][j].fir = right;
                dp[i][j].sec = dp[i][j-1].fir;
            }
        }
    }
    Pair res = dp[0][n-1];
    return res.fir - res.sec;
}
```

动态规划解法，如果没有状态转移方程指导，绝对是一头雾水，但是根据前面的详细解释，读者应该可以清晰理解这一大段代码的含义。

而且，注意到计算 `dp[i][j]` 只依赖其左边和下边的元素，所以说肯定有优化空间，转换成一维 dp，想象一下把二维平面压扁，也就是投影到一维。但是，一维 dp 比较复杂，可解释性很差，大家就不必浪费这个时间去理解了。

四、最后总结

本文给出了解决博弈问题的动态规划解法。博弈问题的前提一般都是在两个聪明人之间进行，编程描述这种游戏的一般方法是二维 dp 数组，数组中通过元组分别表示两人的最优决策。

之所以这样设计，是因为先手在做出选择之后，就成了后手，后手在对方做完选择后，就变成了先手。这种角色转换使得我们可以重用之前的结果，典型的动态规划标志。

读到这里的朋友应该能理解算法解决博弈问题的套路了。学习算法，一定要注重算法的模板框架，而不是一些看起来牛逼的思路，也不要奢求上来就写一个最优的解法。不要舍不得多用空间，不要过早尝试优化，不要惧怕多维数组。dp 数组就是存储信息避免重复计算的，随便用，直到咱满意为止。

希望本文对你有帮助。

贪心算法之区间调度问题

什么是贪心算法呢？贪心算法可以认为是动态规划算法的一个特例，相比动态规划，使用贪心算法需要满足更多的条件（贪心选择性质），但是效率比动态规划要高。

比如说一个算法问题使用暴力解法需要指数级时间，如果能使用动态规划消除重叠子问题，就可以降到多项式级别的时间，如果满足贪心选择性质，那么可以进一步降低时间复杂度，达到线性级别的。

什么是贪心选择性质呢，简单说就是：每一步都做出一个局部最优的选择，最终的结果就是全局最优。注意哦，这是一种特殊性质，其实只有一部分问题拥有这个性质。

比如你面前放着 100 张人民币，你只能拿十张，怎么才能拿最多的面额？显然每次选择剩下钞票中面值最大的一张，最后你的选择一定是最优的。

然而，大部分问题明显不具有贪心选择性质。比如打斗地主，对手出对儿三，按照贪心策略，你应该出尽可能小的牌刚好压制住对方，但现实情况我们甚至可能会出王炸。这种情况就不能用贪心算法，而得使用动态规划解决，参见前文「动态规划解决博弈问题」。

一、问题概述

言归正传，本文解决一个很经典的贪心算法问题 Interval Scheduling（区间调度问题）。给你很多形如 `[start, end]` 的闭区间，请你设计一个算法，**算出这些区间中最多有几个互不相交的区间**。

```
int intervalSchedule(int[][] intvs) {}
```

举个例子，`intvs = [[1,3], [2,4], [3,6]]`，这些区间最多有 2 个区间互不相交，即 `[[1,3], [3,6]]`，你的算法应该返回 2。注意边界相同并不算相交。

这个问题在生活中的应用广泛，比如你今天有好几个活动，每个活动都可以用区间 `[start, end]` 表示开始和结束的时间，请问你今天**最多能参加几个活动呢？**显然你一个人不能同时参加两个活动，所以说这个问题就是求这些时间区间的最大不相交子集。

二、贪心解法

这个问题有许多看起来不错的贪心思路，却都不能得到正确答案。比如说：

也许我们可以每次选择可选区间中开始最早的那个？但是可能存在某些区间开始很早，但是很长，使得我们错误地错过了一些短的区间。或者我们每次选择可选区间中最短的那个？或者选择出现冲突最少的那个区间？这些方案都能很容易举出反例，不是正确的方案。

正确的思路其实很简单，可以分为以下三步：

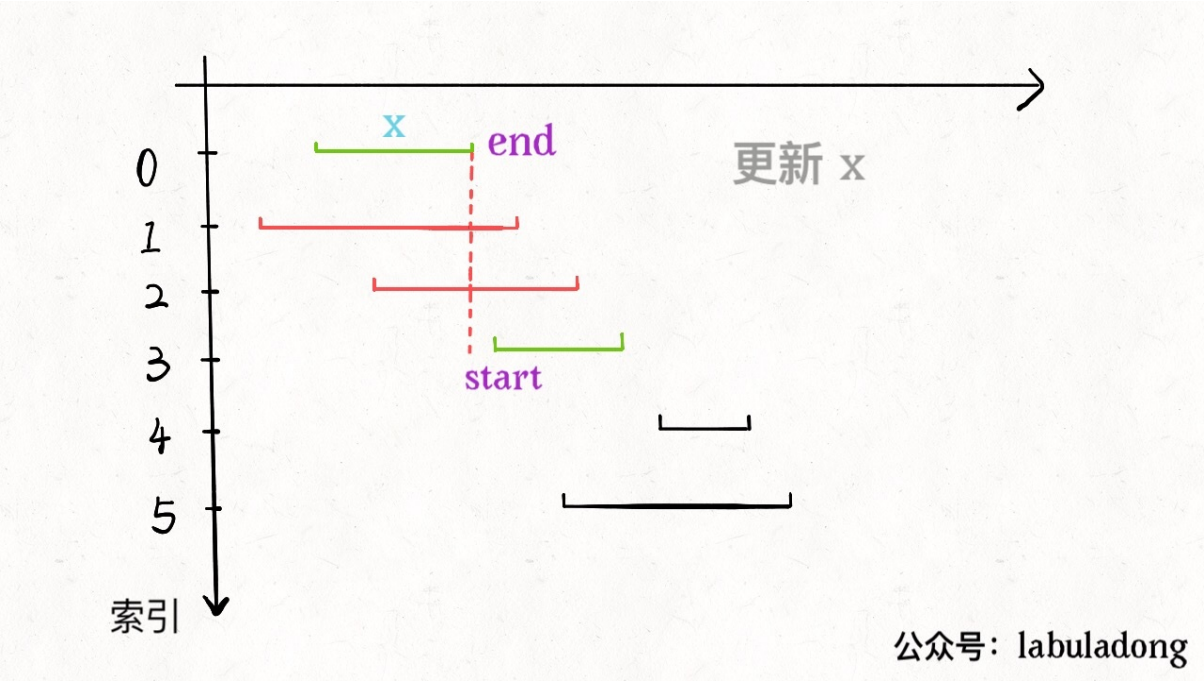
1. 从区间集合 intvs 中选择一个区间 x，这个 x 是在当前所有区间中**结束最早的**（end 最小）。
2. 把所有与 x 区间相交的区间从区间集合 intvs 中删除。
3. 重复步骤 1 和 2，直到 intvs 为空为止。之前选出的那些 x 就是最大不相交子集。

把这个思路实现成算法的话，可以按每个区间的 `end` 数值升序排序，因为这样处理之后实现步骤 1 和步骤 2 都方便很多:

【pdf/mobi格式不支持GIF:interval/1.gif】 请查看【关于本小抄及作者】章节的解决方案

现在来实现算法，对于步骤 1，由于我们预先按照 `end` 排了序，所以选择 x 是很容易的。关键在于，如何去除与 x 相交的区间，选择下一轮循环的 x 呢？

由于我们事先排了序，不难发现所有与 x 相交的区间必然会与 x 的 `end` 相交；如果一个区间不想与 x 的 `end` 相交，它的 `start` 必须要大于（或等于）x 的 `end`：



看下代码：

```
public int intervalSchedule(int[][] intvs) {
    if (intvs.length == 0) return 0;
    // 按 end 升序排序
    Arrays.sort(intvs, new Comparator<int[]>() {
        public int compare(int[] a, int[] b) {
            return a[1] - b[1];
        }
    });
    // 至少有一个区间不相交
    int count = 1;
    // 排序后，第一个区间就是 x
    int x_end = intvs[0][1];
    for (int[] interval : intvs) {
        int start = interval[0];
        if (start >= x_end) {
            // 找到下一个选择的区间了
            count++;
            x_end = interval[1];
        }
```

```
        }
        return count;
    }
```

三、应用举例

下面举例几道 LeetCode 题目应用一下区间调度算法。

第 435 题，无重叠区间：

给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

注意:

1. 可以认为区间的终点总是大于它的起点。
2. 区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。

示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。

我们已经会求最多有几个区间不会重叠了，那么剩下的不就是至少需要去除的区间吗？

```
int eraseOverlapIntervals(int[][] intervals) {
    int n = intervals.length;
    return n - intervalSchedule(intervals);
}
```

第 452 题，用最少的箭头射爆气球：

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在10^4个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 x_{start}，x_{end}，且满足 $x_{start} \le x \le x_{end}$，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

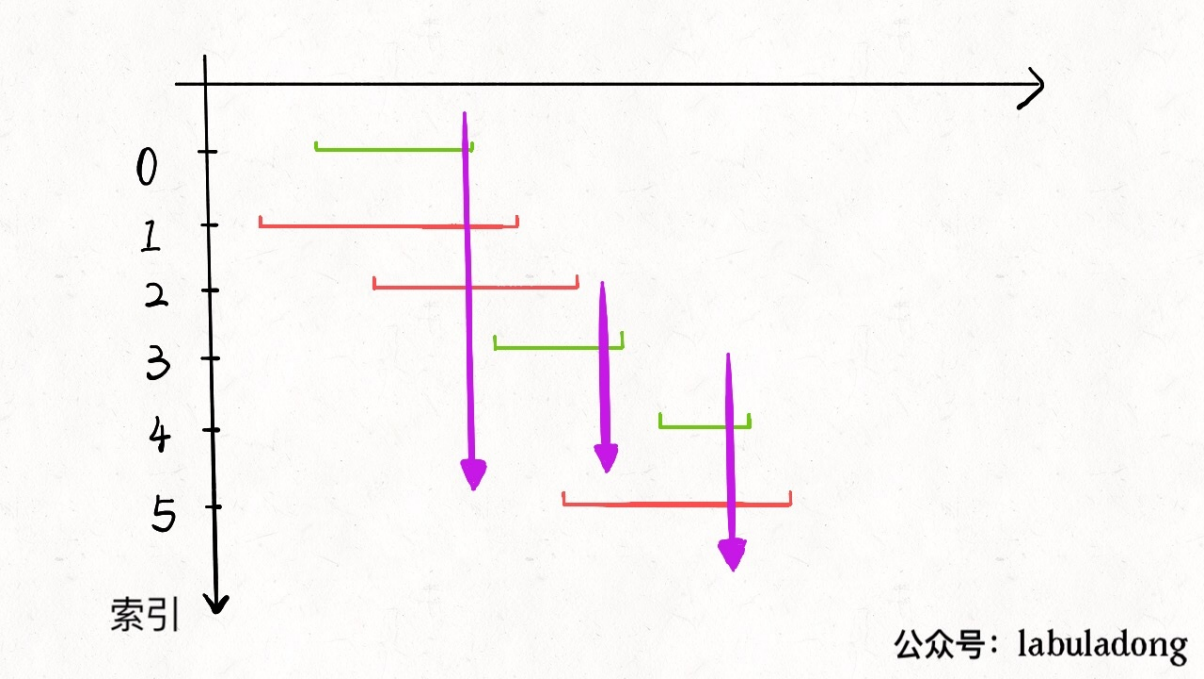
Example:

输入:
`[[10,16], [2,8], [1,6], [7,12]]`

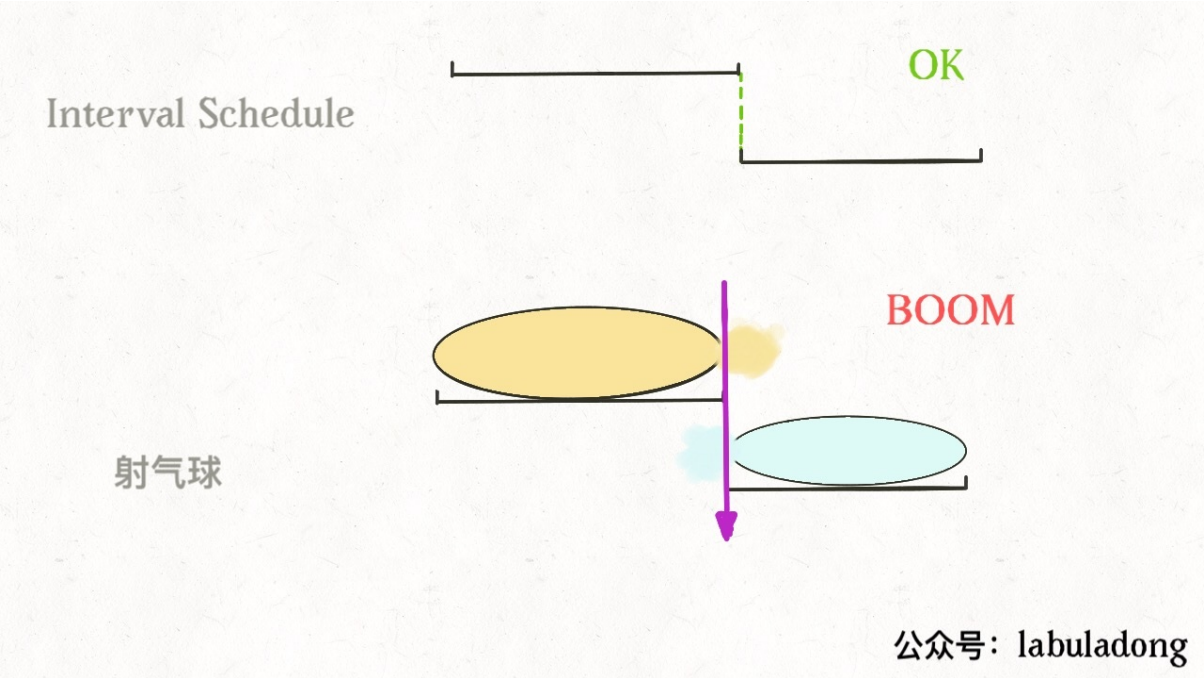
输出:
`2`

解释:
对于该样例，我们可以在`x = 6`（射爆`[2,8],[1,6]`两个气球）和 `x = 11`（射爆另外两个气球）。

其实稍微思考一下，这个问题和区间调度算法一模一样！如果最多有 `n` 个不重叠的区间，那么就至少需要 `n` 个箭头穿透所有区间：



只是有一点不一样，在 `intervalSchedule` 算法中，如果两个区间的边界触碰，不算重叠；而按照这道题目的描述，箭头如果碰到气球的边界气球也会爆炸，所以说相当于区间的边界触碰也算重叠：



所以只要将之前的算法稍作修改，就是这道题目的答案：

```
int findMinArrowShots(int[][] intvs) {
    // ...
```

```
        for (int[] interval : intvs) {
            int start = interval[0];
            // 把 >= 改成 > 就行了
            if (start > x_end) {
                count++;
                x_end = interval[1];
            }
        }
        return count;
    }
```

这么做的原因也不难理解，因为现在边界接触也算重叠，所以 `start == x_end` 时不能更新 x。

如果本文对你有帮助，欢迎关注我的公众号 labuladong，致力于把算法问题讲清楚～

动态规划之KMP字符匹配算法

KMP 算法（Knuth-Morris-Pratt 算法）是一个著名的字符串匹配算法，效率很高，但是确实有点复杂。

很多读者抱怨 KMP 算法无法理解，这很正常，想到大学教材上关于 KMP 算法的讲解，也不知道有多少未来的 Knuth、Morris、Pratt 被提前劝退了。有一些优秀的同学通过手推 KMP 算法的过程来辅助理解该算法，这是一种办法，不过本文要从逻辑层面帮助读者理解算法的原理。十行代码之间，KMP 灰飞烟灭。

先在开头约定，本文用 `pat` 表示模式串，长度为 `M`，`txt` 表示文本串，长度为 `N`。KMP 算法是在 `txt` 中查找子串 `pat`，如果存在，返回这个子串的起始索引，否则返回 -1。

为什么我认为 KMP 算法就是个动态规划问题呢，等会再解释。对于动态规划，之前多次强调了要明确 `dp` 数组的含义，而且同一个问题可能有不止一种定义 `dp` 数组含义的方法，不同的定义会有不同的解法。

读者见过的 KMP 算法应该是，一波诡异的操作处理 `pat` 后形成一个一维的数组 `next`，然后根据这个数组经过又一波复杂操作去匹配 `txt`。时间复杂度 O(N)，空间复杂度 O(M)。其实它这个 `next` 数组就相当于 `dp` 数组，其中元素的含义跟 `pat` 的前缀和后缀有关，判定规则比较复杂，不好理解。**本文则用一个二维的 `dp` 数组（但空间复杂度还是 O(M)），重新定义其中元素的含义，使得代码长度大大减少，可解释性大大提高**。

PS：本文的代码参考《算法4》，原代码使用的数组名称是 `dfa`（确定有限状态机），因为我们的公众号之前有一系列动态规划的文章，就不说这么高大上的名词了，我对书中代码进行了一点修改，并沿用 `dp` 数组的名称。

一、KMP 算法概述

首先还是简单介绍一下 KMP 算法和暴力匹配算法的不同在哪里，难点在哪里，和动态规划有啥关系。

暴力的字符串匹配算法很容易写，看一下它的运行逻辑：

```
// 暴力匹配（伪码）
int search(String pat, String txt) {
    int M = pat.length;
    int N = txt.length;
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (pat[j] != txt[i+j])
                break;
        }
        // pat 全都匹配了
        if (j == M) return i;
    }
    // txt 中不存在 pat 子串
    return -1;
}
```

对于暴力算法，如果出现不匹配字符，同时回退 `txt` 和 `pat` 的指针，嵌套 for 循环，时间复杂度 $O(MN)$，空间复杂度$O(1)$。最主要的问题是，如果字符串中重复的字符比较多，该算法就显得很蠢。

比如 txt = "aaacaaab" pat = "aaab"：

【pdf/mobi格式不支持GIF:kmp/1.gif】 请查看【关于本小抄及作者】章节的解决方案

很明显，`pat` 中根本没有字符 c，根本没必要回退指针 `i`，暴力解法明显多做了很多不必要的操作。

KMP 算法的不同之处在于，它会花费空间来记录一些信息，在上述情况中就会显得很聪明：

【pdf/mobi格式不支持GIF:kmp/2.gif】 请查看【关于本小抄及作者】章节的解决方案

再比如类似的 txt = "aaaaaaab" pat = "aaab"，暴力解法还会和上面那个例子一样蠢蠢地回退指针 `i`，而 KMP 算法又会耍聪明：

【pdf/mobi格式不支持GIF:kmp/3.gif】 请查看【关于本小抄及作者】章节的解决方案

因为 KMP 算法知道字符 b 之前的字符 a 都是匹配的，所以每次只需要比较字符 b 是否被匹配就行了。

KMP 算法永不回退 `txt` 的指针 `i`，不走回头路（不会重复扫描 `txt`），而是借助 `dp` 数组中储存的信息把 `pat` 移到正确的位置继续匹配，时间复杂度只需 O(N)，用空间换时间，所以我认为它是一种动态规划算法。

KMP 算法的难点在于，如何计算 `dp` 数组中的信息？如何根据这些信息正确地移动 `pat` 的指针？这个就需要**确定有限状态自动机**来辅助了，别怕这种高大上的文学词汇，其实和动态规划的 `dp` 数组如出一辙，等你学会了也可以拿这个词去吓唬别人。

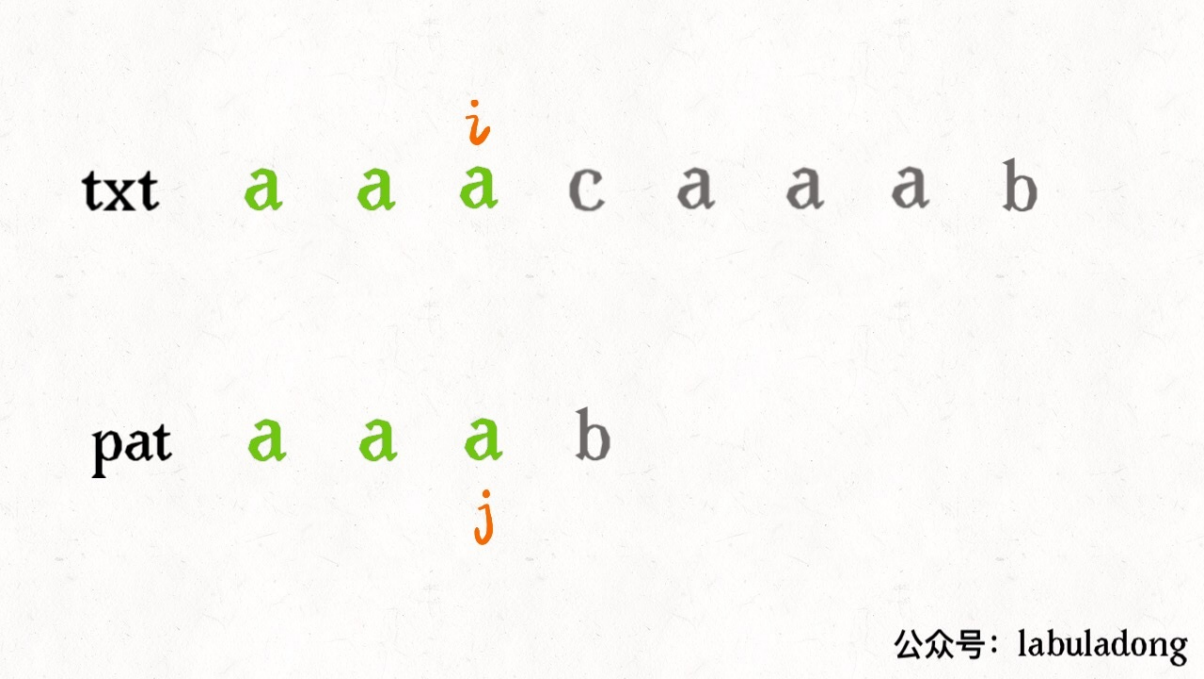
还有一点需要明确的是：**计算这个 `dp` 数组，只和 `pat` 串有关**。意思是说，只要给我个 `pat`，我就能通过这个模式串计算出 `dp` 数组，然后你可以给我不同的 `txt`，我都不怕，利用这个 `dp` 数组我都能在 O(N) 时间完成字符串匹配。

具体来说，比如上文举的两个例子：

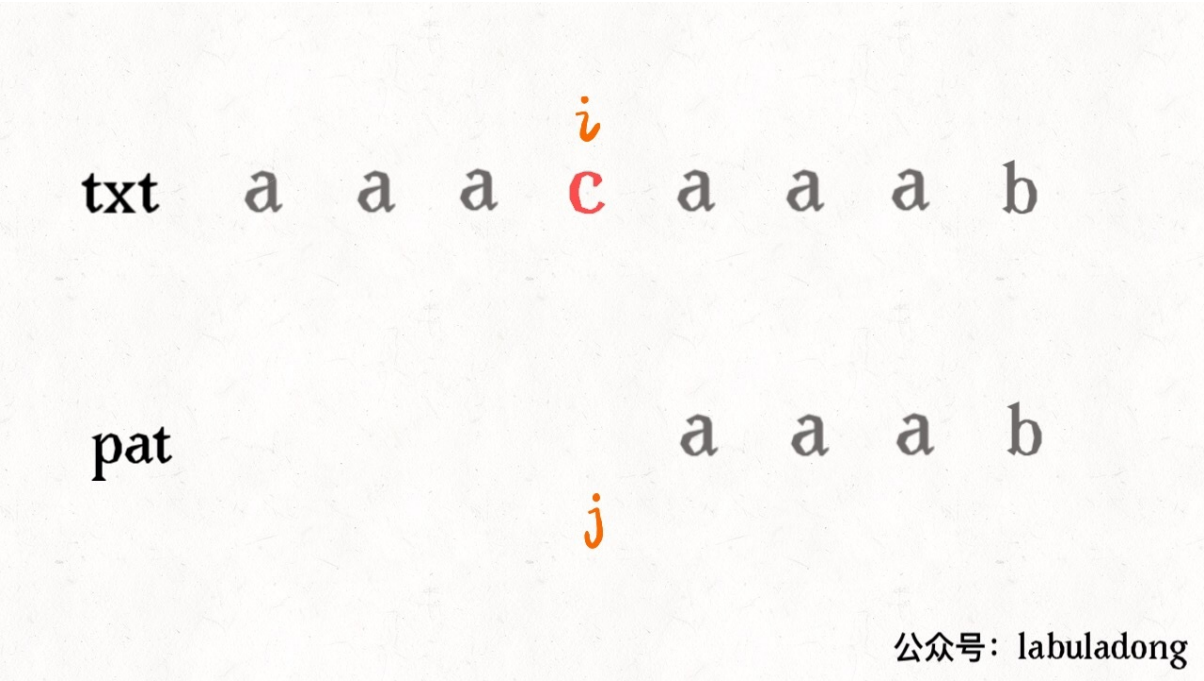
```
txt1 = "aaacaaab"
pat = "aaab"
txt2 = "aaaaaaab"
pat = "aaab"
```

我们的 `txt` 不同，但是 `pat` 是一样的，所以 KMP 算法使用的 `dp` 数组是同一个。

只不过对于 `txt1` 的下面这个即将出现的未匹配情况：

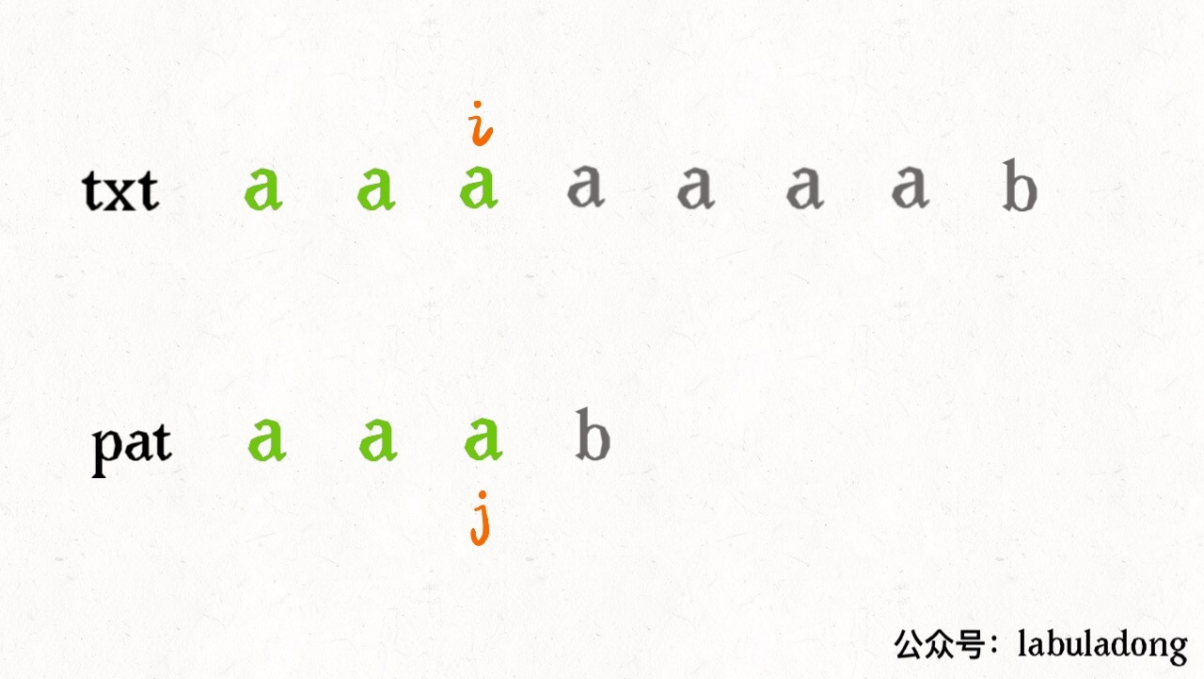


`dp` 数组指示 `pat` 这样移动：

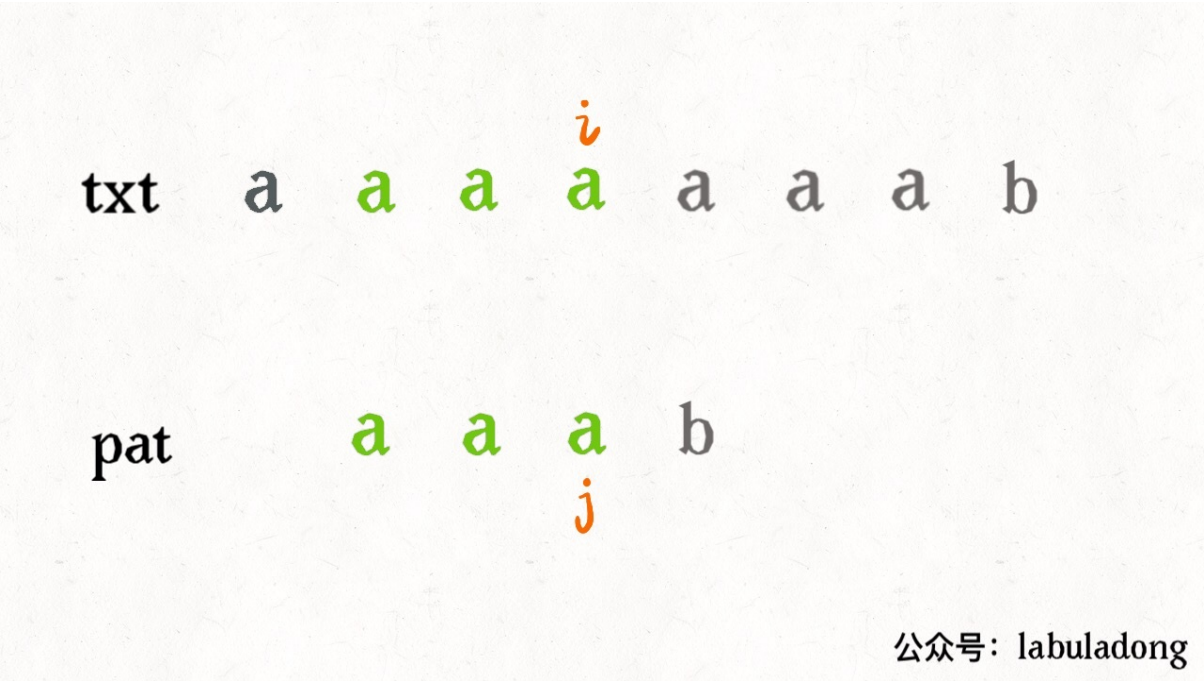


PS：这个 `j` 不要理解为索引，它的含义更准确地说应该是**状态**（state），所以它会出现这个奇怪的位置，后文会详述。

而对于 `txt2` 的下面这个即将出现的未匹配情况：



`dp` 数组指示 `pat` 这样移动：



明白了 `dp` 数组只和 `pat` 有关，那么我们这样设计 KMP 算法就会比较漂亮：

```
public class KMP {
    private int[][] dp;
    private String pat;
```

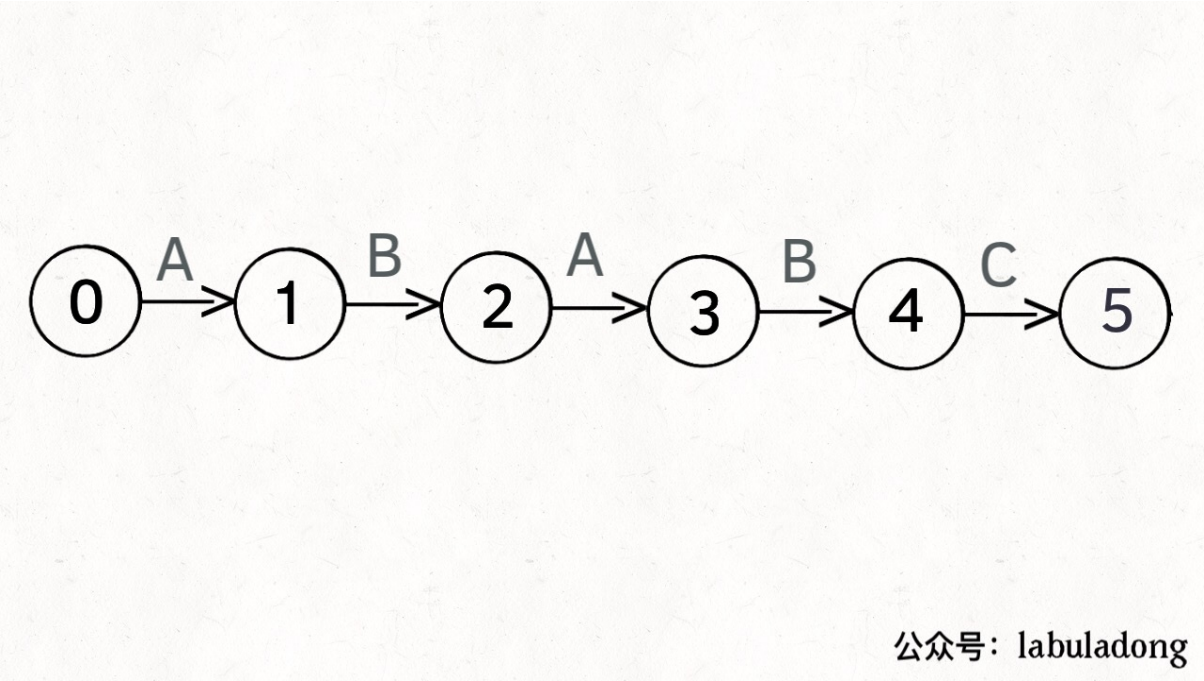
```
    public KMP(String pat) {
        this.pat = pat;
        // 通过 pat 构建 dp 数组
        // 需要 O(M) 时间
    }

    public int search(String txt) {
        // 借助 dp 数组去匹配 txt
        // 需要 O(N) 时间
    }
}
```

这样，当我们需要用同一 `pat` 去匹配不同 `txt` 时，就不需要浪费时间构造 `dp` 数组了：

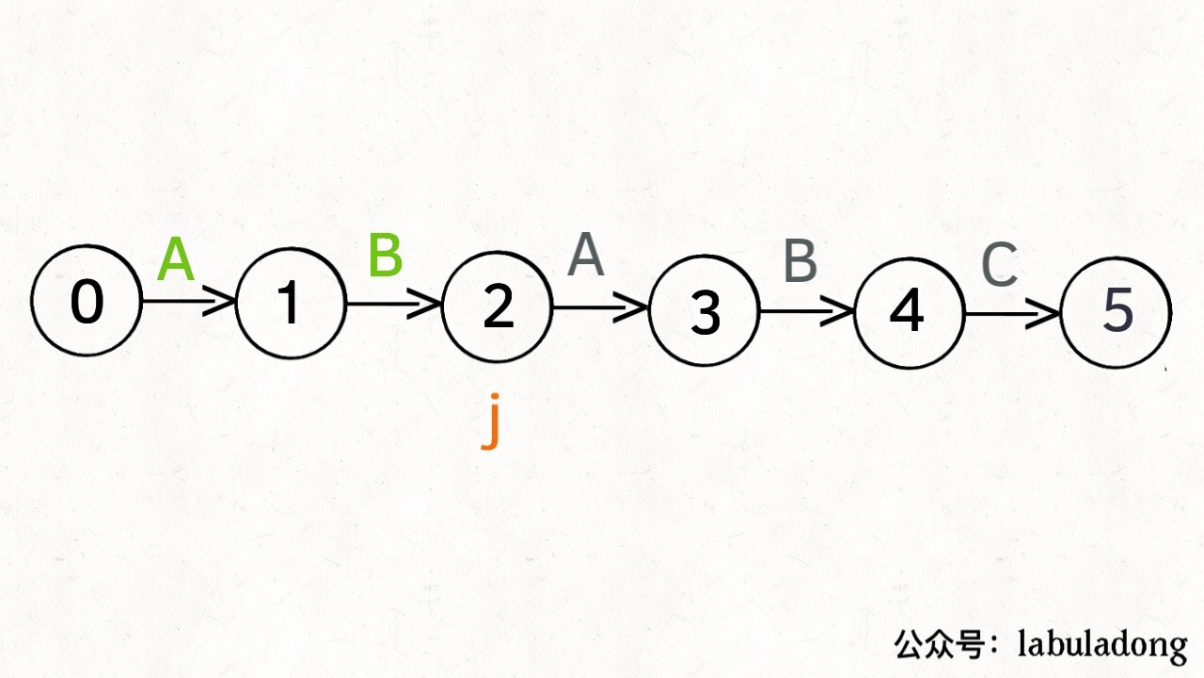
```
KMP kmp = new KMP("aaab");
int pos1 = kmp.search("aaacaaab"); //4
int pos2 = kmp.search("aaaaaaab"); //4
```

二、状态机概述

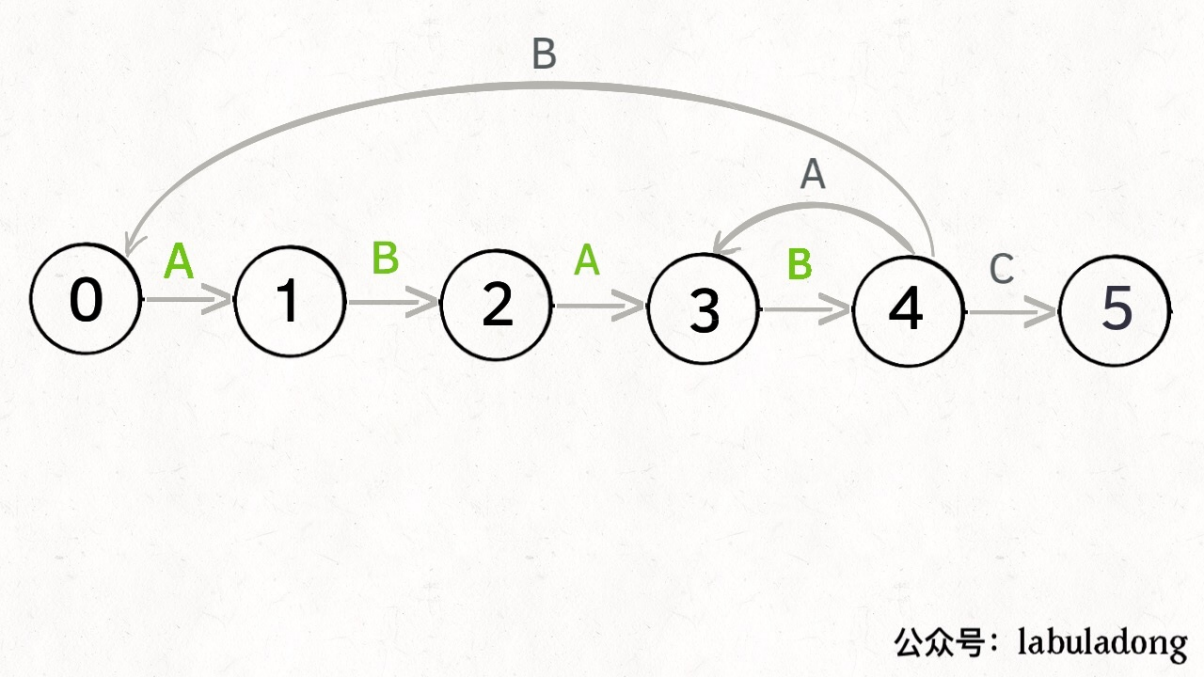
为什么说 KMP 算法和状态机有关呢？是这样的，我们可以认为 `pat` 的匹配就是状态的转移。比如当 pat = "ABABC"：



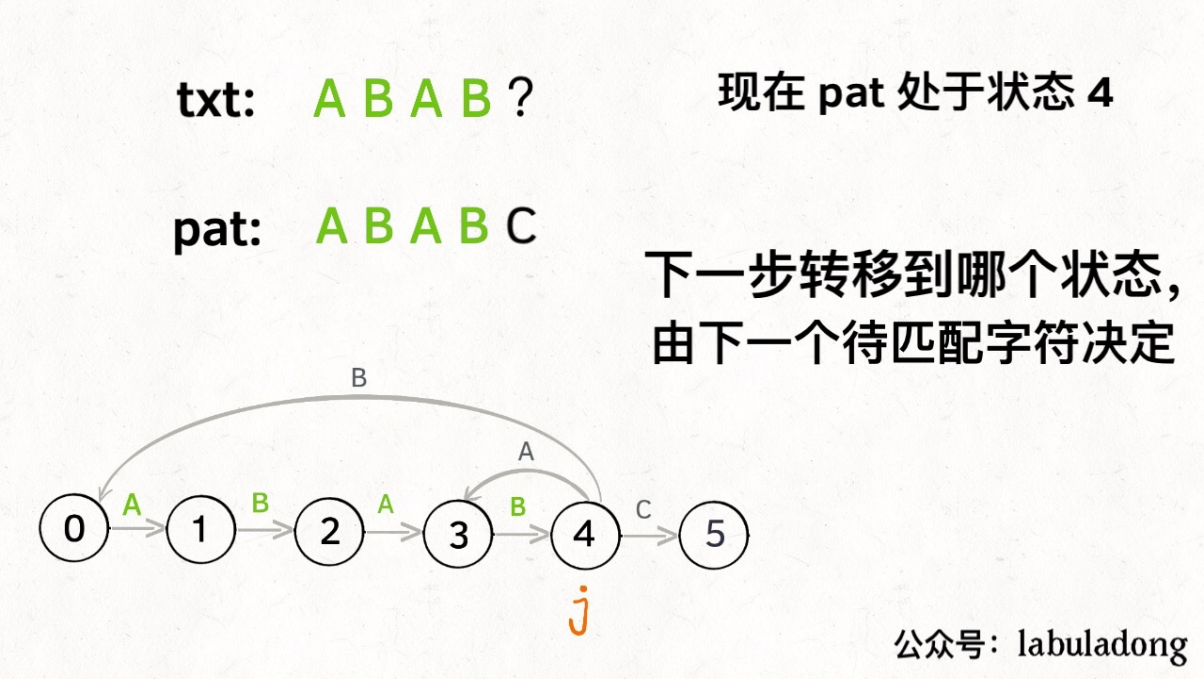
如上图，圆圈内的数字就是状态，状态 0 是起始状态，状态 5（`pat.length`）是终止状态。开始匹配时 `pat` 处于起始状态，一旦转移到终止状态，就说明在 `txt` 中找到了 `pat`。比如说当前处于状态 2，就说明字符 "AB" 被匹配：



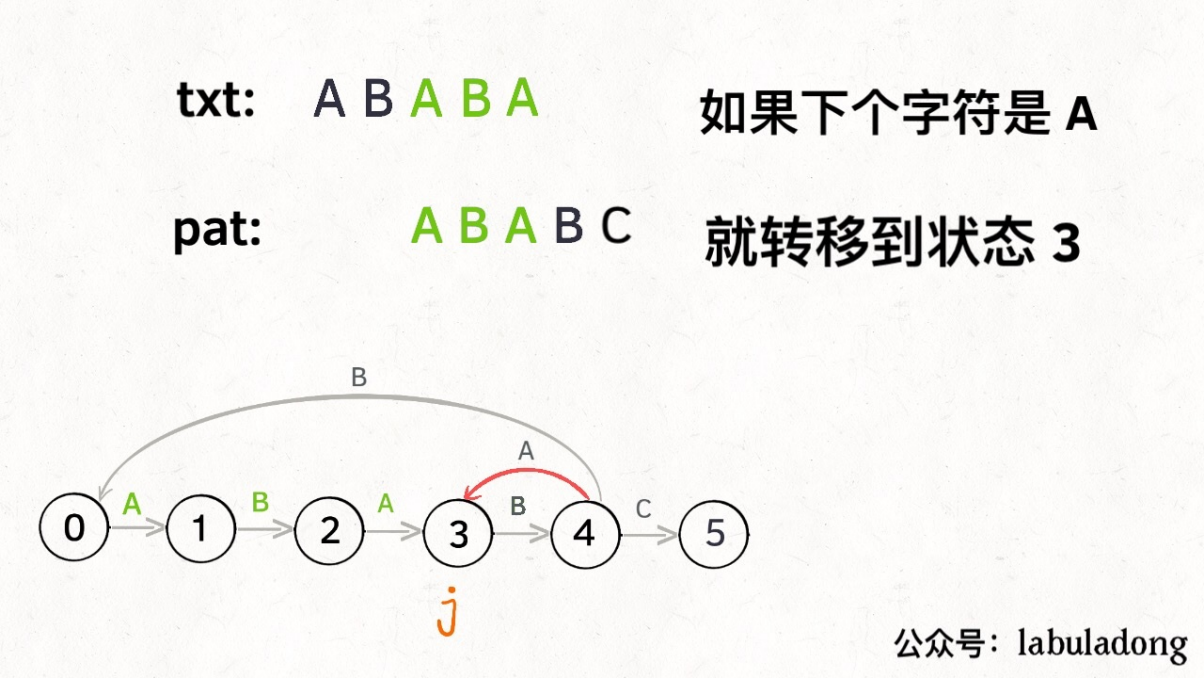
另外，处于不同状态时，`pat` 状态转移的行为也不同。比如说假设现在匹配到了状态 4，如果遇到字符 A 就应该转移到状态 3，遇到字符 C 就应该转移到状态 5，如果遇到字符 B 就应该转移到状态 0：



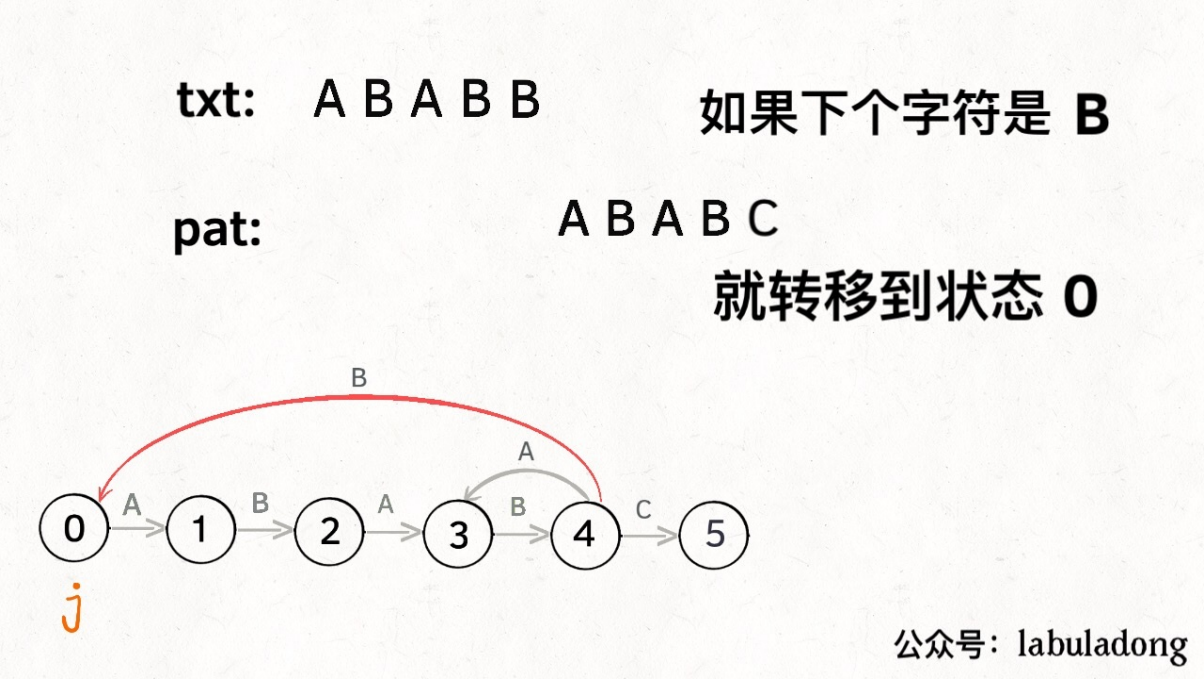
具体什么意思呢，我们来一个个举例看看。用变量 `j` 表示指向当前状态的指针，当前 `pat` 匹配到了状态 4：



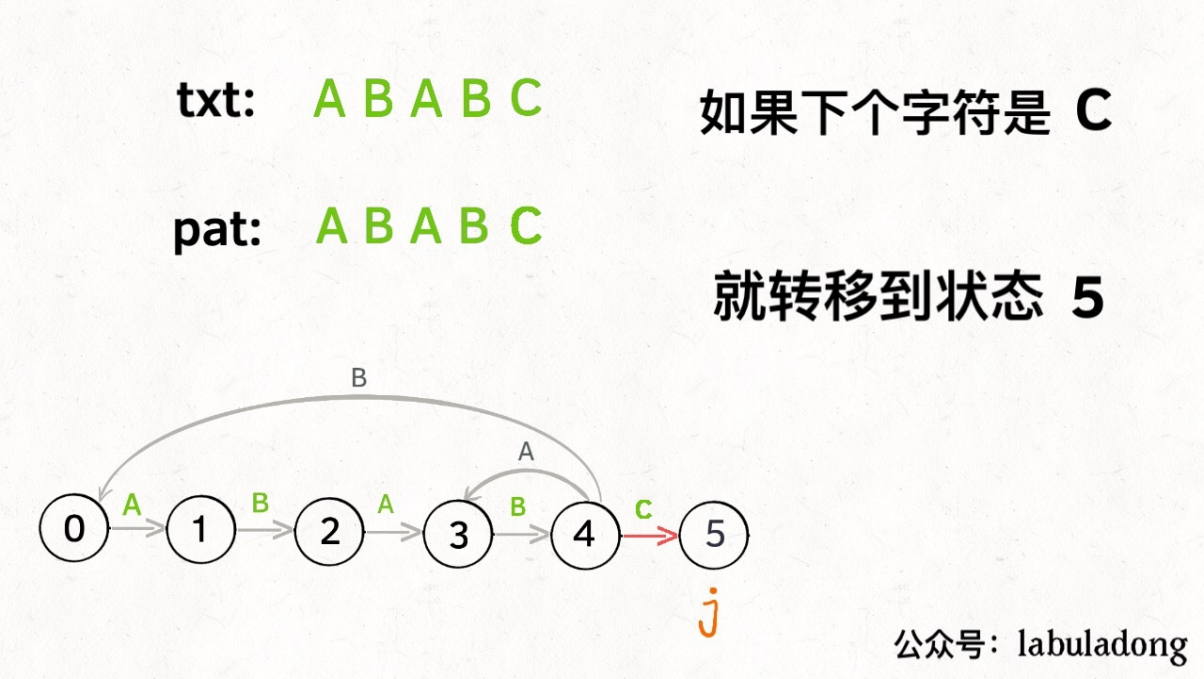
如果遇到了字符 "A"，根据箭头指示，转移到状态 3 是最聪明的：



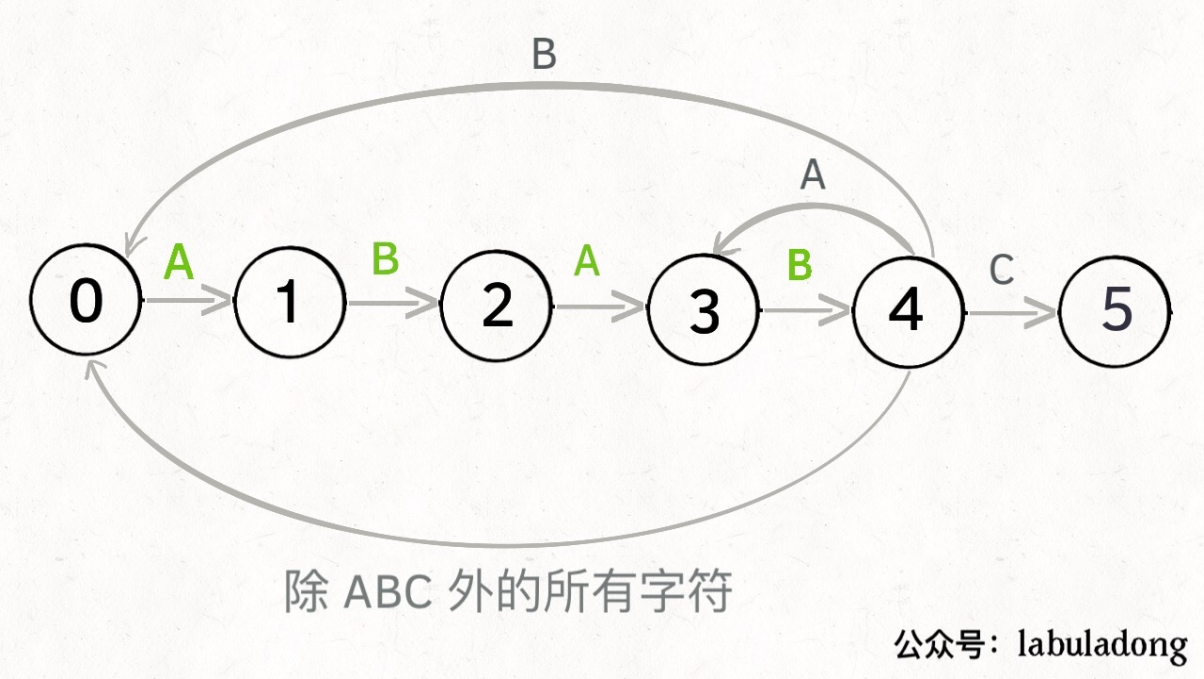
如果遇到了字符 "B"，根据箭头指示，只能转移到状态 0（一夜回到解放前）：



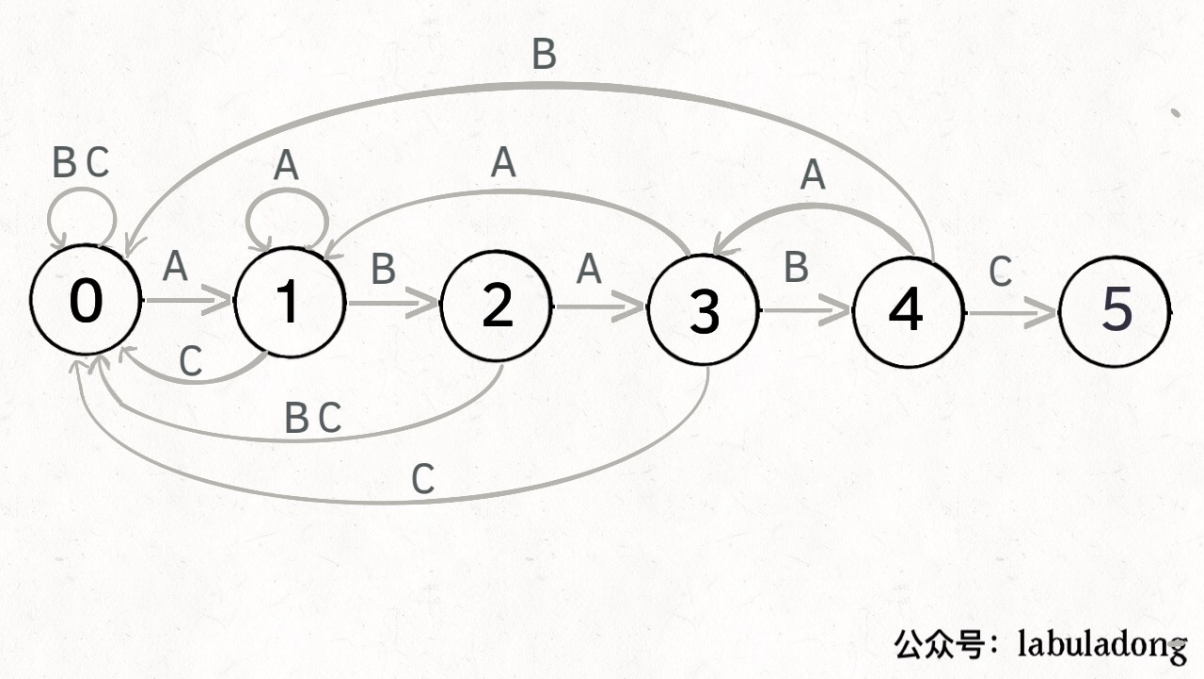
如果遇到了字符 "C"，根据箭头指示，应该转移到终止状态 5，这也就意味着匹配完成：



当然了，还可能遇到其他字符，比如 Z，但是显然应该转移到起始状态 0，因为 `pat` 中根本都没有字符 Z：



这里为了清晰起见，我们画状态图时就把其他字符转移到状态 0 的箭头省略，只画 `pat` 中出现的字符的状态转移：



KMP 算法最关键的步骤就是构造这个状态转移图。**要确定状态转移的行为，得明确两个变量，一个是当前的匹配状态，另一个是遇到的字符**；确定了这两个变量后，就可以知道这个情况下应该转移到哪个状态。

下面看一下 KMP 算法根据这幅状态转移图匹配字符串 `txt` 的过程：

【pdf/mobi格式不支持GIF:kmp/kmp.gif】 请查看【关于本小抄及作者】章节的解决方案

请记住这个 GIF 的匹配过程，这就是 KMP 算法的核心逻辑！

为了描述状态转移图，我们定义一个二维 dp 数组，它的含义如下：

```
dp[j][c] = next
0 <= j < M，代表当前的状态
0 <= c < 256，代表遇到的字符（ASCII 码）
0 <= next <= M，代表下一个状态

dp[4]['A'] = 3 表示：
当前是状态 4，如果遇到字符 A，
pat 应该转移到状态 3

dp[1]['B'] = 2 表示：
当前是状态 1，如果遇到字符 B，
pat 应该转移到状态 2
```

根据我们这个 dp 数组的定义和刚才状态转移的过程，我们可以先写出 KMP 算法的 search 函数代码：

```
public int search(String txt) {
    int M = pat.length();
    int N = txt.length();
    // pat 的初始态为 0
    int j = 0;
    for (int i = 0; i < N; i++) {
        // 当前是状态 j，遇到字符 txt[i]，
        // pat 应该转移到哪个状态？
        j = dp[j][txt.charAt(i)];
        // 如果达到终止态，返回匹配开头的索引
        if (j == M) return i - M + 1;
    }
    // 没到达终止态，匹配失败
    return -1;
}
```

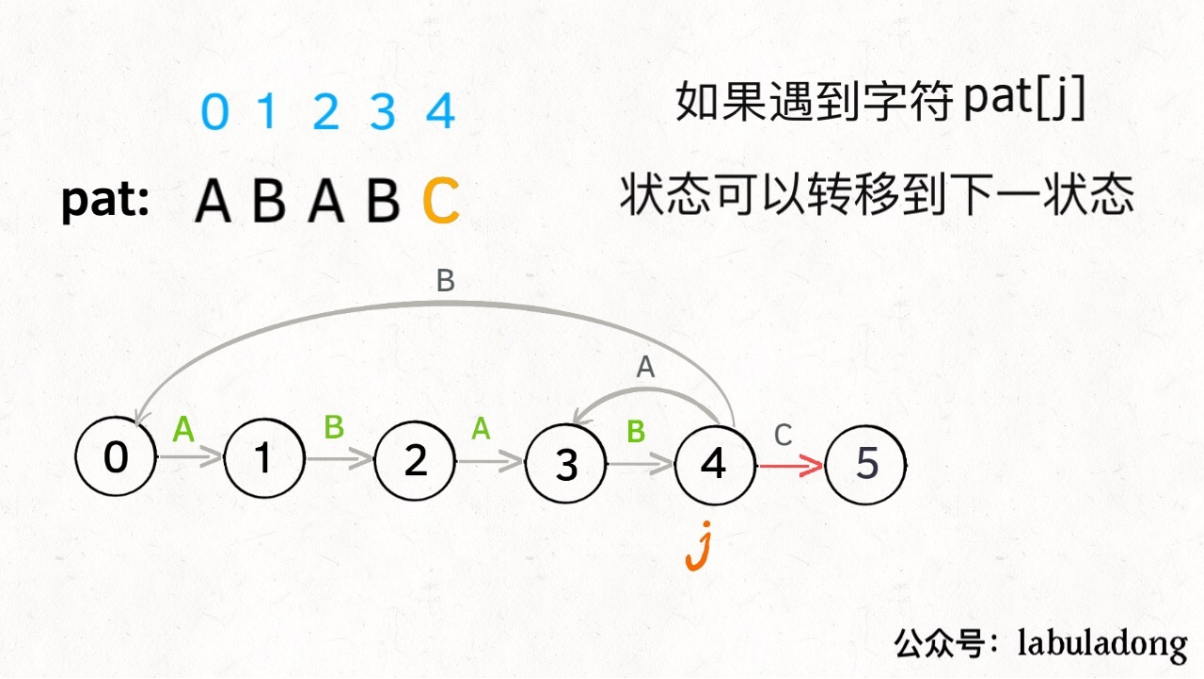
到这里，应该还是很好理解的吧，`dp` 数组就是我们刚才画的那幅状态转移图，如果不清楚的话回去看下 GIF 的算法演进过程。下面讲解：如何通过 `pat` 构建这个 `dp` 数组？

三、构建状态转移图

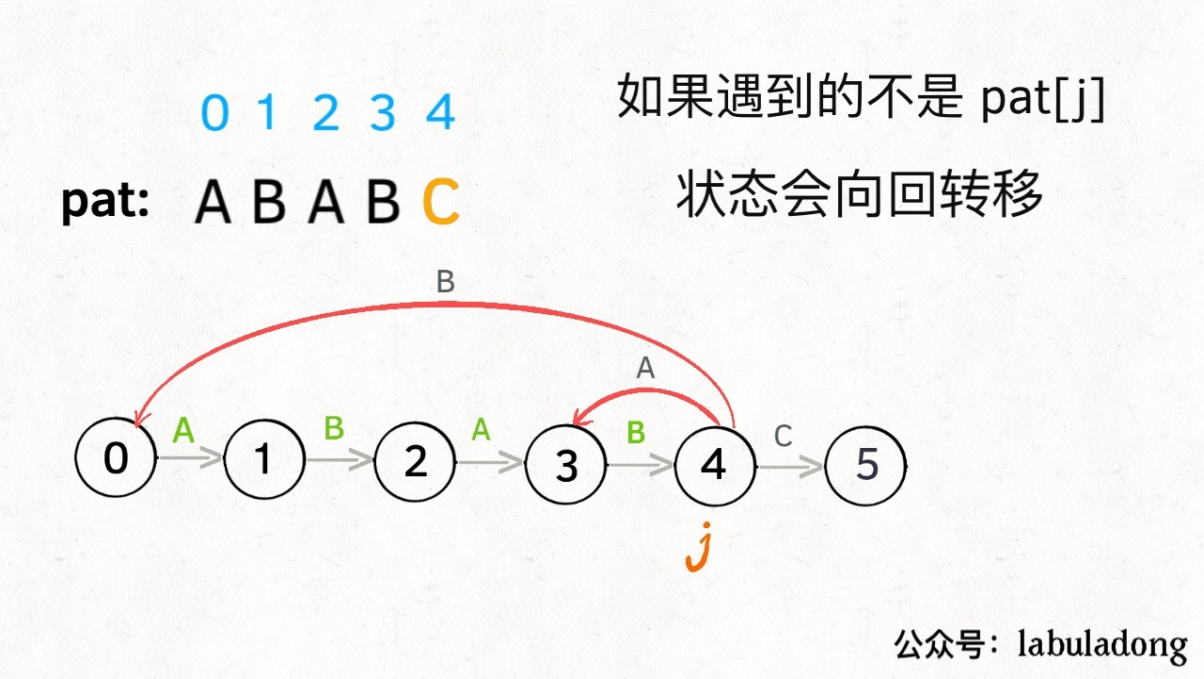
回想刚才说的：**要确定状态转移的行为，必须明确两个变量，一个是当前的匹配状态，另一个是遇到的字符**，而且我们已经根据这个逻辑确定了 `dp` 数组的含义，那么构造 `dp` 数组的框架就是这样：

```
for 0 <= j < M: # 状态
    for 0 <= c < 256: # 字符
        dp[j][c] = next
```

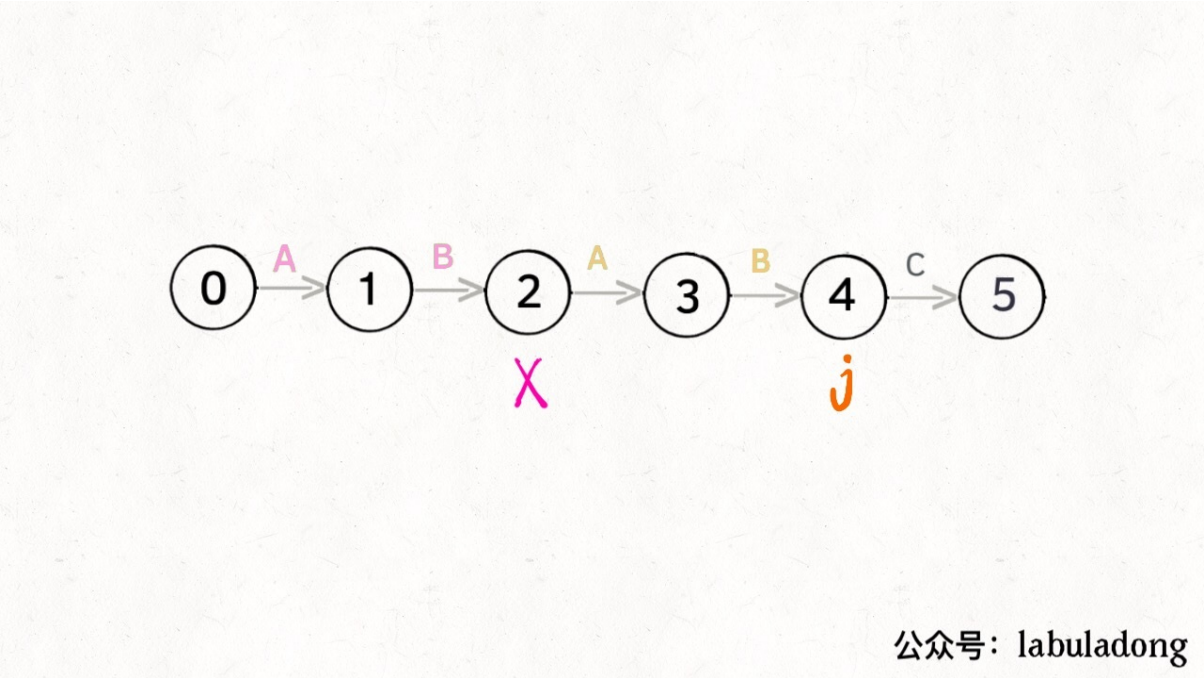
这个 next 状态应该怎么求呢？显然，**如果遇到的字符 `c` 和 `pat[j]` 匹配的话**，状态就应该向前推进一个，也就是说 `next = j + 1`，我们不妨称这种情况为**状态推进**：



如果字符 `c` 和 `pat[j]` 不匹配的话，状态就要回退（或者原地不动），我们不妨称这种情况为**状态重启**：

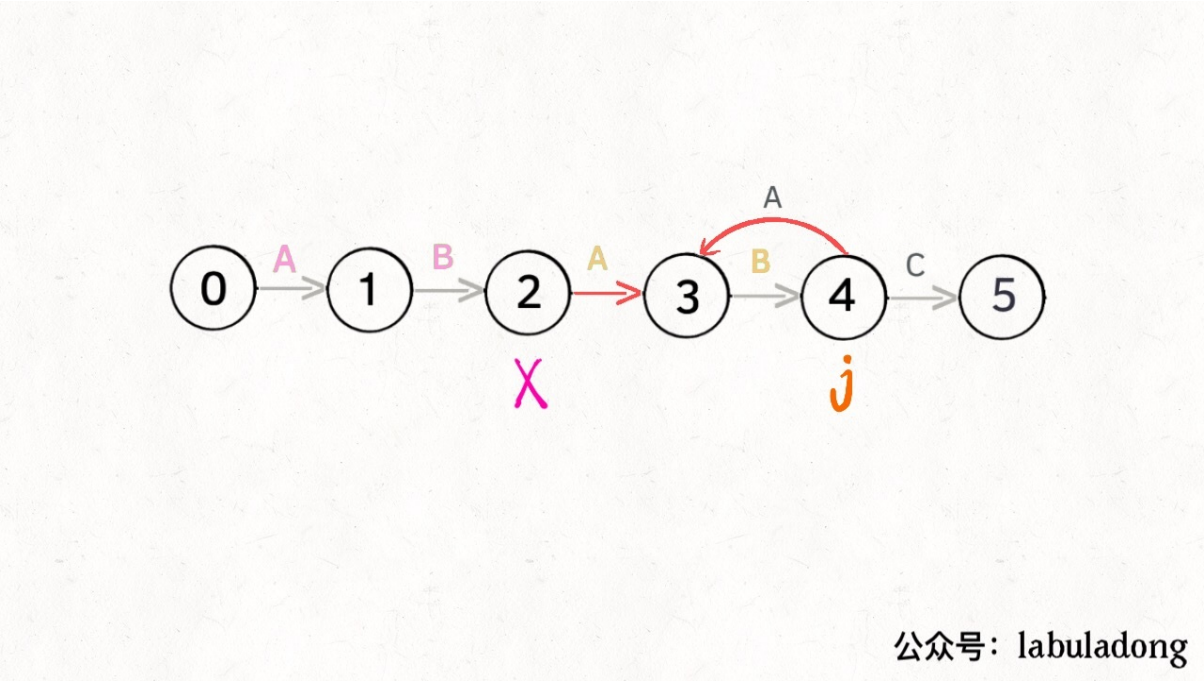


那么，如何得知在哪个状态重启呢？解答这个问题之前，我们再定义一个名字：**影子状态**（我编的名字），用变量 `X` 表示。**所谓影子状态，就是和当前状态具有相同的前缀**。比如下面这种情况：



当前状态 `j = 4`，其影子状态为 `X = 2`，它们都有相同的前缀 "AB"。因为状态 `X` 和状态 `j` 存在相同的前缀，所以当状态 `j` 准备进行状态重启的时候（遇到的字符 `c` 和 `pat[j]` 不匹配），可以通过 `X` 的状态转移图来获得**最近的重启位置**。

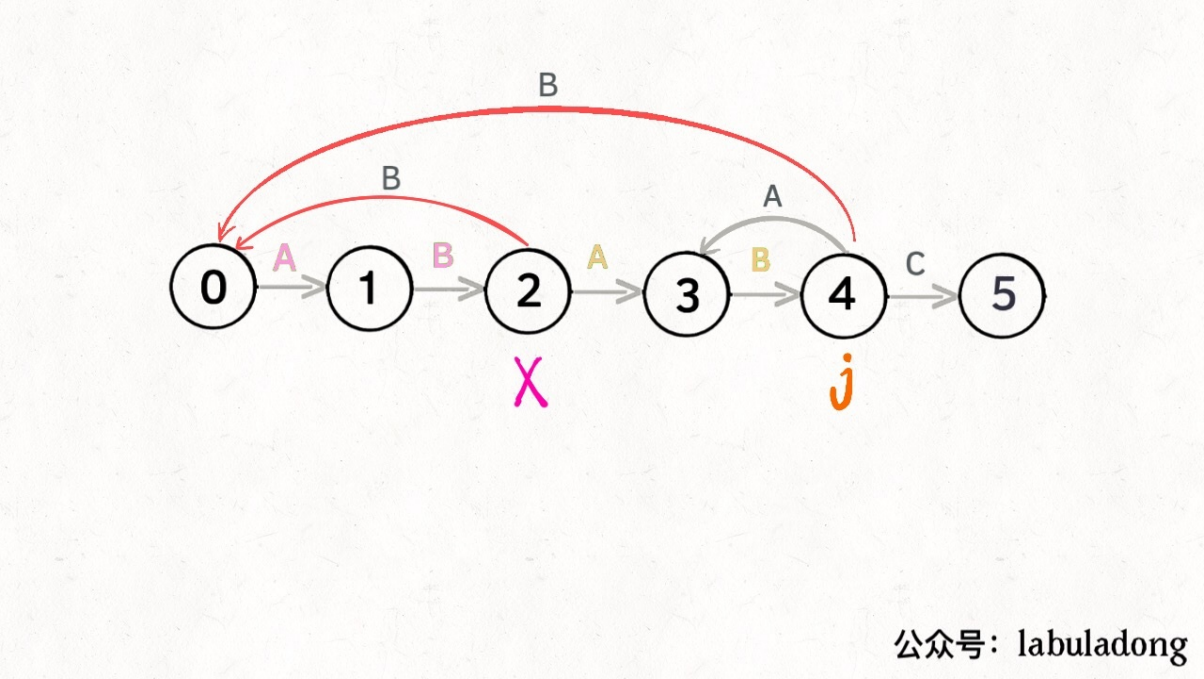
比如说刚才的情况，如果状态 `j` 遇到一个字符 "A"，应该转移到哪里呢？首先只有遇到 "C" 才能推进状态，遇到 "A" 显然只能进行状态重启。**状态 `j` 会把这个字符委托给状态 `X` 处理，也就是 `dp[j]['A'] = dp[X]['A']`**：



为什么这样可以呢？因为：既然 `j` 这边已经确定字符 "A" 无法推进状态，**只能回退**，而且 KMP 就是要**尽可能少的回退**，以免多余的计算。那么 `j` 就可以去问问和自己具有相同前缀的 `X`，如果 `X` 遇见 "A" 可以进行「状态推进」，那就转移过去，因为这样回退最少。

【pdf/mobi格式不支持GIF:kmp/A.gif】 请查看【关于本小抄及作者】章节的解决方案

当然，如果遇到的字符是 "B"，状态 `X` 也不能进行「状态推进」，只能回退，`j` 只要跟着 `X` 指引的方向回退就行了：



你也许会问，这个 `X` 怎么知道遇到字符 "B" 要回退到状态 0 呢？因为 `X` 永远跟在 `j` 的身后，状态 `X` 如何转移，在之前就已经算出来了。动态规划算法不就是利用过去的结果解决现在的问题吗？

这样，我们就细化一下刚才的框架代码：

```
int X # 影子状态
for 0 <= j < M:
    for 0 <= c < 256:
        if c == pat[j]:
            # 状态推进
            dp[j][c] = j + 1
        else:
            # 状态重启
            # 委托 X 计算重启位置
            dp[j][c] = dp[X][c]
```

四、代码实现

如果之前的内容你都能理解，恭喜你，现在就剩下一个问题：影子状态 `X` 是如何得到的呢？下面先直接看完整代码吧。

```
public class KMP {
```

```
    private int[][] dp;
    private String pat;

    public KMP(String pat) {
        this.pat = pat;
        int M = pat.length();
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat.charAt(0)] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 当前状态 j 从 1 开始
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++) {
                if (pat.charAt(j) == c)
                    dp[j][c] = j + 1;
                else
                    dp[j][c] = dp[X][c];
            }
            // 更新影子状态
            X = dp[X][pat.charAt(j)];
        }
    }

    public int search(String txt) {...}
}
```

先解释一下这一行代码：

```
// base case
dp[0][pat.charAt(0)] = 1;
```

这行代码是 base case，只有遇到 pat[0] 这个字符才能使状态从 0 转移到 1，遇到其它字符的话还是停留在状态 0（Java 默认初始化数组全为 0）。

影子状态 `X` 是先初始化为 0，然后随着 `j` 的前进而不断更新的。下面看看到底应该**如何更新影子状态 `X`**：

```
int X = 0;
```

```
for (int j = 1; j < M; j++) {
    ...
    // 更新影子状态
    // 当前是状态 X，遇到字符 pat[j]，
    // pat 应该转移到哪个状态？
    X = dp[X][pat.charAt(j)];
}
```

更新 `X` 其实和 `search` 函数中更新状态 `j` 的过程是非常相似的：

```
int j = 0;
for (int i = 0; i < N; i++) {
    // 当前是状态 j，遇到字符 txt[i]，
    // pat 应该转移到哪个状态？
    j = dp[j][txt.charAt(i)];
    ...
}
```

其中的原理非常微妙，注意代码中 for 循环的变量初始值，可以这样理解：后者是在 `txt` 中匹配 `pat`，前者是在 `pat` 中匹配 `pat[1..end]`，状态 `X` 总是落后状态 `j` 一个状态，与 `j` 具有最长的相同前缀。所以我把 `X` 比喻为影子状态，似乎也有一点贴切。

另外，构建 dp 数组是根据 base case `dp[0][..]` 向后推演。这就是我认为 KMP 算法就是一种动态规划算法的原因。

下面来看一下状态转移图的完整构造过程，你就能理解状态 `X` 作用之精妙了：

【pdf/mobi格式不支持GIF:kmp/dfa.gif】 请查看【关于本小抄及作者】章节的解决方案

至此，KMP 算法的核心终于写完啦啦啦啦！看下 KMP 算法的完整代码吧：

```
public class KMP {
    private int[][] dp;
    private String pat;
```

```
    public KMP(String pat) {
        this.pat = pat;
        int M = pat.length();
        // dp[状态][字符] = 下个状态
        dp = new int[M][256];
        // base case
        dp[0][pat.charAt(0)] = 1;
        // 影子状态 X 初始为 0
        int X = 0;
        // 构建状态转移图（稍改的更紧凑了）
        for (int j = 1; j < M; j++) {
            for (int c = 0; c < 256; c++)
                dp[j][c] = dp[X][c];
            dp[j][pat.charAt(j)] = j + 1;
            // 更新影子状态
            X = dp[X][pat.charAt(j)];
        }
    }

    public int search(String txt) {
        int M = pat.length();
        int N = txt.length();
        // pat 的初始态为 0
        int j = 0;
        for (int i = 0; i < N; i++) {
            // 计算 pat 的下一个状态
            j = dp[j][txt.charAt(i)];
            // 到达终止态，返回结果
            if (j == M) return i - M + 1;
        }
        // 没到达终止态，匹配失败
        return -1;
    }
}
```

经过之前的详细举例讲解，你应该可以理解这段代码的含义了，当然你也可以把 KMP 算法写成一个函数。核心代码也就是两个函数中 for 循环的部分，数一下有超过十行吗？

五、最后总结

传统的 KMP 算法是使用一个一维数组 `next` 记录前缀信息，而本文是使用一个二维数组 `dp` 以状态转移的角度解决字符匹配问题，但是空间复杂度仍然是 O(256M) = O(M)。

在 `pat` 匹配 `txt` 的过程中，只要明确了「当前处在哪个状态」和「遇到的字符是什么」这两个问题，就可以确定应该转移到哪个状态（推进或回退）。

对于一个模式串 `pat`，其总共就有 M 个状态，对于 ASCII 字符，总共不会超过 256 种。所以我们就构造一个数组 `dp[M][256]` 来包含所有情况，并且明确 `dp` 数组的含义：

`dp[j][c] = next` 表示，当前是状态 `j`，遇到了字符 `c`，应该转移到状态 `next`。

明确了其含义，就可以很容易写出 search 函数的代码。

对于如何构建这个 `dp` 数组，需要一个辅助状态 `X`，它永远比当前状态 `j` 落后一个状态，拥有和 `j` 最长的相同前缀，我们给它起了个名字叫「影子状态」。

在构建当前状态 `j` 的转移方向时，只有字符 `pat[j]` 才能使状态推进（`dp[j][pat[j]] = j+1`）；而对于其他字符只能进行状态回退，应该去请教影子状态 `X` 应该回退到哪里（`dp[j][other] = dp[X][other]`，其中 `other` 是除了 `pat[j]` 之外所有字符）。

对于影子状态 `X`，我们把它初始化为 0，并且随着 `j` 的前进进行更新，更新的方式和 search 过程更新 `j` 的过程非常相似（`X = dp[X][pat[j]]`）。

KMP 算法也就是动态规划那点事，我们的公众号文章目录有一系列专门讲动态规划的，而且都是按照一套框架来的，无非就是描述问题逻辑，明确 `dp` 数组含义，定义 base case 这点破事。希望这篇文章能让大家对动态规划有更深的理解。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

团灭 LeetCode 股票买卖问题

很多读者抱怨 LeetCode 的股票系列问题奇技淫巧太多，如果面试真的遇到这类问题，基本不会想到那些巧妙的办法，怎么办？**所以本文拒绝奇技淫巧，而是稳扎稳打，只用一种通用方法解决所用问题，以不变应万变**。

这篇文章用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

先随便抽出一道题，看看别人的解法：

```
int maxProfit(vector<int>& prices) {
    if(prices.empty()) return 0;
    int s1=-prices[0],s2=INT_MIN,s3=INT_MIN,s4=INT_MIN;

    for(int i=1;i<prices.size();++i) {
        s1 = max(s1, -prices[i]);
        s2 = max(s2, s1+prices[i]);
        s3 = max(s3, s2-prices[i]);
        s4 = max(s4, s3+prices[i]);
    }
    return max(0,s4);
}
```

能看懂吧？会做了吗？不可能的，你看不懂，这才正常。就算你勉强看懂了，下一个问题你还是做不出来。为什么别人能写出这么诡异却又高效的解法呢？因为这类问题是有框架的，但是人家不会告诉你的，因为一旦告诉你，你五分钟就学会了，该算法题就不再神秘，变得不堪一击了。

本文就来告诉你这个框架，然后带着你一道一道秒杀。这篇文章用状态机的技巧来解决，可以全部提交通过。不要觉得这个名词高大上，文学词汇而已，实际上就是 DP table，看一眼就明白了。

这 6 道题目是有共性的，我就抽出来第 4 道题目，因为这道题是一个最泛化的形式，其他的问题都是这个形式的简化，看下题目：

给定一个数组，它的第 *i* 个元素是一支给定的股票在第 *i* 天的价格。

设计一个算法来计算你所能获取的最大利润。你最多可以完成 **k** 笔交易。

注意: 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。

示例 1:

```
输入: [2,4,1], k = 2
输出: 2
解释: 在第 1 天 (股票价格 = 2) 的时候买入，在第 2 天 (股票价格 = 4) 的时候卖出，这笔交易所能获得利润 = 4-2 = 2 。
```

示例 2:

```
输入: [3,2,6,5,0,3], k = 2
输出: 7
解释: 在第 2 天 (股票价格 = 2) 的时候买入，在第 3 天 (股票价格 = 6) 的时候卖出, 这笔交易所能获得利润 = 6-2 = 4 。
     随后，在第 5 天 (股票价格 = 0) 的时候买入，在第 6 天 (股票价格 = 3) 的时候卖出, 这笔交易所能获得利润 = 3-0 = 3 。
```

第一题是只进行一次交易，相当于 k = 1；第二题是不限交易次数，相当于 k = +infinity（正无穷）；第三题是只进行 2 次交易，相当于 k = 2；剩下两道也是不限次数，但是加了交易「冷冻期」和「手续费」的额外条件，其实就是第二题的变种，都很容易处理。

如果你还不熟悉题目，可以去 LeetCode 查看这些题目的内容，本文为了节省篇幅，就不列举这些题目的具体内容了。下面言归正传，开始解题。

一、穷举框架

首先，还是一样的思路：如何穷举？这里的穷举思路和上篇文章递归的思想不太一样。

递归其实是符合我们思考的逻辑的，一步步推进，遇到无法解决的就丢给递归，一不小心就做出来了，可读性还很好。缺点就是一旦出错，你也不容易找到错误出现的原因。比如上篇文章的递归解法，肯定还有计算冗余，但确实不容易找到。

而这里，我们不用递归思想进行穷举，而是利用「状态」进行穷举。我们具体到每一天，看看总共有几种可能的「状态」，再找出每个「状态」对应的「选择」。我们要穷举所有「状态」，穷举的目的是根据对应的「选择」更新状态。听起来抽象，你只要记住「状态」和「选择」两个词就行，下面实操一下就很容易明白了。

```
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

比如说这个问题，**每天都有三种「选择」**：买入、卖出、无操作，我们用 buy, sell, rest 表示这三种选择。但问题是，并不是每天都可以任意选择这三种选择的，因为 sell 必须在 buy 之后，buy 必须在 sell 之后。那么 rest 操作还应该分两种状态，一种是 buy 之后的 rest（持有了股票），一种是 sell 之后的 rest（没有持有股票）。而且别忘了，我们还有交易次数 k 的限制，就是说你 buy 还只能在 k > 0 的前提下操作。

很复杂对吧，不要怕，我们现在的目的只是穷举，你有再多的状态，老夫要做的就是一把梭全部列举出来。**这个问题的「状态」有三个**，第一个是天数，第二个是允许交易的最大次数，第三个是当前的持有状态（即之前说的 rest 的状态，我们不妨用 1 表示持有，0 表示没有持有）。然后我们用一个三维数组就可以装下这几种状态的全部组合：

```
dp[i][k][0 or 1]
0 <= i <= n-1, 1 <= k <= K
n 为天数，大 K 为最多交易数
此问题共 n × K × 2 种状态，全部穷举就能搞定。

for 0 <= i < n:
    for 1 <= k <= K:
        for s in {0, 1}:
            dp[i][k][s] = max(buy, sell, rest)
```

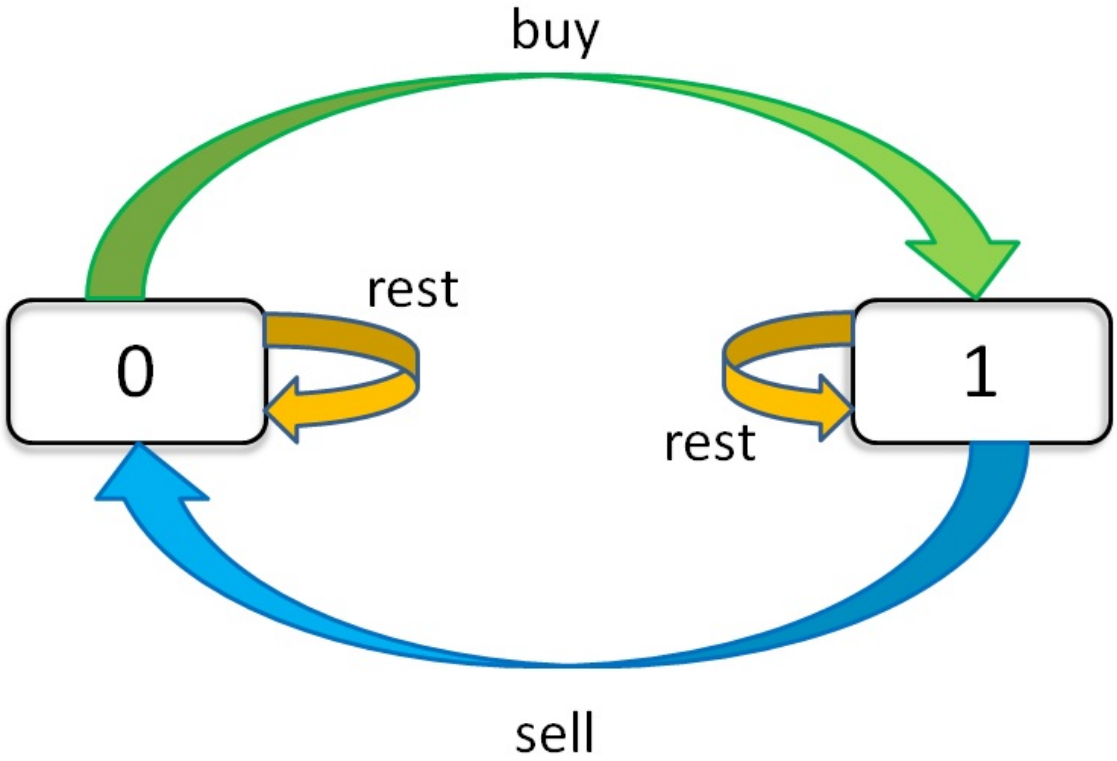
而且我们可以用自然语言描述出每一个状态的含义，比如说 `dp[3][2][1]` 的含义就是：今天是第三天，我现在手上持有着股票，至今最多进行 2 次交易。再比如 `dp[2][3][0]` 的含义：今天是第二天，我现在手上没有持有股票，至今最多进行 3 次交易。很容易理解，对吧？

我们想求的最终答案是 dp[n - 1][K][0]，即最后一天，最多允许 K 次交易，最多获得多少利润。读者可能问为什么不是 dp[n - 1][K][1]？因为 [1] 代表手上还持有股票，[0] 表示手上的股票已经卖出去了，很显然后者得到的利润一定大于前者。

记住如何解释「状态」，一旦你觉得哪里不好理解，把它翻译成自然语言就容易理解了。

二、状态转移框架

现在，我们完成了「状态」的穷举，我们开始思考每种「状态」有哪些「选择」，应该如何更新「状态」。只看「持有状态」，可以画个状态转移图。



通过这个图可以很清楚地看到，每种状态（0 和 1）是如何转移而来的。根据这个图，我们来写一下状态转移方程：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
              max(   选择 rest  ,             选择 sell      )

解释：今天我没有持有股票，有两种可能：
要么是我昨天就没有持有，然后今天选择 rest，所以我今天还是没有持有；
要么是我昨天持有股票，但是今天我 sell 了，所以我今天没有持有股票了。

dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
              max(   选择 rest  ,           选择 buy         )

解释：今天我持有着股票，有两种可能：
要么我昨天就持有着股票，然后今天选择 rest，所以我今天还持有着股票；
要么我昨天本没有持有，但今天我选择 buy，所以今天我就持有股票了。
```

这个解释应该很清楚了，如果 buy，就要从利润中减去 prices[i]，如果 sell，就要给利润增加 prices[i]。今天的最大利润就是这两种可能选择中较大的那个。而且注意 k 的限制，我们在选择 buy 的时候，把 k 减小了 1，很好理解吧，当然你也可以在 sell 的时候减 1，一样的。

现在，我们已经完成了动态规划中最困难的一步：状态转移方程。**如果之前的内容你都可以理解，那么你已经可以秒杀所有问题了，只要套这个框架就行了**。不过还差最后一点点，就是定义 base case，即最简单的情况。

```
dp[-1][k][0] = 0
解释：因为 i 是从 0 开始的，所以 i = -1 意味着还没有开始，这时候的利润当然是 0
。
dp[-1][k][1] = -infinity
解释：还没开始的时候，是不可能持有股票的，用负无穷表示这种不可能。
dp[i][0][0] = 0
解释：因为 k 是从 1 开始的，所以 k = 0 意味着根本不允许交易，这时候利润当然是 0
。
dp[i][0][1] = -infinity
解释：不允许交易的情况下，是不可能持有股票的，用负无穷表示这种不可能。
```

把上面的状态转移方程总结一下：

```
base case：
dp[-1][k][0] = dp[i][0][0] = 0
```

```
dp[-1][k][1] = dp[i][0][1] = -infinity

状态转移方程：
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

读者可能会问，这个数组索引是 -1 怎么编程表示出来呢，负无穷怎么表示呢？这都是细节问题，有很多方法实现。现在完整的框架已经完成，下面开始具体化。

三、秒杀题目

第一题，k = 1

直接套状态转移方程，根据 base case，可以做一些化简：

```
dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
dp[i][1][1] = max(dp[i-1][1][1], dp[i-1][0][0] - prices[i])
            = max(dp[i-1][1][1], -prices[i])
解释：k = 0 的 base case，所以 dp[i-1][0][0] = 0。

现在发现 k 都是 1，不会改变，即 k 对状态转移已经没有影响了。
可以进行进一步化简去掉所有 k：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], -prices[i])
```

直接写出代码：

```
int n = prices.length;
int[][] dp = new int[n][2];
for (int i = 0; i < n; i++) {
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

显然 i = 0 时 dp[i-1] 是不合法的。这是因为我们没有对 i 的 base case 进行处理。可以这样处理：

```
for (int i = 0; i < n; i++) {
    if (i - 1 == -1) {
        dp[i][0] = 0;
        // 解释：
        //   dp[i][0]
        // = max(dp[-1][0], dp[-1][1] + prices[i])
        // = max(0, -infinity + prices[i]) = 0
        dp[i][1] = -prices[i];
        //解释：
        //   dp[i][1]
        // = max(dp[-1][1], dp[-1][0] - prices[i])
        // = max(-infinity, 0 - prices[i])
        // = -prices[i]
        continue;
    }
    dp[i][0] = Math.max(dp[i-1][0], dp[i-1][1] + prices[i]);
    dp[i][1] = Math.max(dp[i-1][1], -prices[i]);
}
return dp[n - 1][0];
```

第一题就解决了，但是这样处理 base case 很麻烦，而且注意一下状态转移方程，新状态只和相邻的一个状态有关，其实不用整个 dp 数组，只需要一个变量储存相邻的那个状态就足够了，这样可以把空间复杂度降到 O(1):

```
// k == 1
int maxProfit_k_1(int[] prices) {
    int n = prices.length;
    // base case: dp[-1][0] = 0, dp[-1][1] = -infinity
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        // dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        // dp[i][1] = max(dp[i-1][1], -prices[i])
        dp_i_1 = Math.max(dp_i_1, -prices[i]);
    }
    return dp_i_0;
}
```

两种方式都是一样的，不过这种编程方法简洁很多。但是如果没有前面状态转移方程的引导，是肯定看不懂的。后续的题目，我主要写这种空间复杂度 O(1) 的解法。

第二题，k = +infinity

如果 k 为正无穷，那么就可以认为 k 和 k - 1 是一样的。可以这样改写框架：

```
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
            = max(dp[i-1][k][1], dp[i-1][k][0] - prices[i])

我们发现数组中的 k 已经不会改变了，也就是说不需要记录 k 这个状态了：
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i])
```

直接翻译成代码：

```
int maxProfit_k_inf(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i]);
    }
    return dp_i_0;
}
```

第三题，k = +infinity with cooldown

每次 sell 之后要等一天才能继续交易。只要把这个特点融入上一题的状态转移方程即可：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-2][0] - prices[i])
解释：第 i 天选择 buy 的时候，要从 i-2 的状态转移，而不是 i-1 。
```

翻译成代码：

```
int maxProfit_with_cool(int[] prices) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    int dp_pre_0 = 0; // 代表 dp[i-2][0]
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, dp_pre_0 - prices[i]);
        dp_pre_0 = temp;
    }
    return dp_i_0;
}
```

第四题，k = +infinity with fee

每次交易要支付手续费，只要把手续费从利润中减去即可。改写方程：

```
dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i])
dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i] - fee)
解释：相当于买入股票的价格升高了。
在第一个式子里减也是一样的，相当于卖出股票的价格减小了。
```

直接翻译成代码：

```
int maxProfit_with_fee(int[] prices, int fee) {
    int n = prices.length;
    int dp_i_0 = 0, dp_i_1 = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        int temp = dp_i_0;
        dp_i_0 = Math.max(dp_i_0, dp_i_1 + prices[i]);
        dp_i_1 = Math.max(dp_i_1, temp - prices[i] - fee);
    }
    return dp_i_0;
}
```

第五题，k = 2

k = 2 和前面题目的情况稍微不同，因为上面的情况都和 k 的关系不太大。要么 k 是正无穷，状态转移和 k 没关系了；要么 k = 1，跟 k = 0 这个 base case 挨得近，最后也没有存在感。

这道题 k = 2 和后面要讲的 k 是任意正整数的情况中，对 k 的处理就凸显出来了。我们直接写代码，边写边分析原因。

```
原始的动态转移方程，没有可化简的地方
dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i])
dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
```

按照之前的代码，我们可能想当然这样写代码（错误的）：

```
int k = 2;
int[][][] dp = new int[n][k + 1][2];
for (int i = 0; i < n; i++)
    if (i - 1 == -1) { /* 处理一下 base case*/ }
    dp[i][k][0] = Math.max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
    dp[i][k][1] = Math.max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i]
);
}
return dp[n - 1][k][0];
```

为什么错误？我这不是照着状态转移方程写的吗？

还记得前面总结的「穷举框架」吗？就是说我们必须穷举所有状态。其实我们之前的解法，都在穷举所有状态，只是之前的题目中 k 都被化简掉了。比如说第一题，k = 1：

「代码截图」

这道题由于没有消掉 k 的影响，所以必须要对 k 进行穷举：

```
int max_k = 2;
int[][][] dp = new int[n][max_k + 1][2];
for (int i = 0; i < n; i++) {
```

```
    for (int k = max_k; k >= 1; k--) {
        if (i - 1 == -1) { /*处理 base case */ }
        dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i]);
        dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices[i])
;
    }
}
// 穷举了 n × max_k × 2 个状态，正确。
return dp[n - 1][max_k][0];
```

如果你不理解，可以返回第一点「穷举框架」重新阅读体会一下。

这里 k 取值范围比较小，所以可以不用 for 循环，直接把 k = 1 和 2 的情况全部列举出来也可以：

```
dp[i][2][0] = max(dp[i-1][2][0], dp[i-1][2][1] + prices[i])
dp[i][2][1] = max(dp[i-1][2][1], dp[i-1][1][0] - prices[i])
dp[i][1][0] = max(dp[i-1][1][0], dp[i-1][1][1] + prices[i])
dp[i][1][1] = max(dp[i-1][1][1], -prices[i])

int maxProfit_k_2(int[] prices) {
    int dp_i10 = 0, dp_i11 = Integer.MIN_VALUE;
    int dp_i20 = 0, dp_i21 = Integer.MIN_VALUE;
    for (int price : prices) {
        dp_i20 = Math.max(dp_i20, dp_i21 + price);
        dp_i21 = Math.max(dp_i21, dp_i10 - price);
        dp_i10 = Math.max(dp_i10, dp_i11 + price);
        dp_i11 = Math.max(dp_i11, -price);
    }
    return dp_i20;
}
```

有状态转移方程和含义明确的变量名指导，相信你很容易看懂。其实我们可以故弄玄虚，把上述四个变量换成 a, b, c, d。这样当别人看到你的代码时就会大惊失色，对你肃然起敬。

第六题，k = any integer

有了上一题 k = 2 的铺垫，这题应该和上一题的第一个解法没啥区别。但是出现了一个超内存的错误，原来是传入的 k 值会非常大，dp 数组太大了。现在想想，交易次数 k 最多有多大呢？

一次交易由买入和卖出构成，至少需要两天。所以说有效的限制 k 应该不超过 n/2，如果超过，就没有约束作用了，相当于 k = +infinity。这种情况是之前解决过的。

直接把之前的代码重用：

```
int maxProfit_k_any(int max_k, int[] prices) {
    int n = prices.length;
    if (max_k > n / 2)
        return maxProfit_k_inf(prices);

    int[][][] dp = new int[n][max_k + 1][2];
    for (int i = 0; i < n; i++)
        for (int k = max_k; k >= 1; k--) {
            if (i - 1 == -1) { /* 处理 base case */ }
            dp[i][k][0] = max(dp[i-1][k][0], dp[i-1][k][1] + prices[i
]);
            dp[i][k][1] = max(dp[i-1][k][1], dp[i-1][k-1][0] - prices
[i]);
        }
    return dp[n - 1][max_k][0];
}
```

至此，6 道题目通过一个状态转移方程全部解决。

四、最后总结

本文给大家讲了如何通过状态转移的方法解决复杂的问题，用一个状态转移方程秒杀了 6 道股票买卖问题，现在想想，其实也不算难对吧？这已经属于动态规划问题中较困难的了。

关键就在于列举出所有可能的「状态」，然后想想怎么穷举更新这些「状态」。一般用一个多维 dp 数组储存这些状态，从 base case 开始向后推进，推进到最后的状态，就是我们想要的答案。想想这个过程，你是不是有点理

解「动态规划」这个名词的意义了呢？

具体到股票买卖问题，我们发现了三个状态，使用了一个三维数组，无非还是穷举 + 更新，不过我们可以说的高大上一点，这叫「三维 DP」，怕不怕？这个大实话一说，立刻显得你高人一等，名利双收有没有，所以给个在看/分享吧，鼓励一下我。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

团灭 LeetCode 打家劫舍问题

有读者私下问我 LeetCode 「打家劫舍」系列问题（英文版叫 House Robber）怎么做，我发现这一系列题目的点赞非常之高，是比较有代表性和技巧性的动态规划题目，今天就来聊聊这道题目。

打家劫舍系列总共有三道，难度设计非常合理，层层递进。第一道是比较标准的动态规划问题，而第二道融入了环形数组的条件，第三道更绝，把动态规划的自底向上和自顶向下解法和二叉树结合起来，我认为很有启发性。如果没做过的朋友，建议学习一下。

下面，我们从第一道开始分析。

House Robber I

你是一个专业的盗贼，计划偷打劫街的房屋。每间房内都藏有一定的现金，影响你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盗贼闯入，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:

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

示例 2:

```
输入：[2,7,9,3,1]
输出：12
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）。
     偷窃到的最高金额 = 2 + 9 + 1 = 12 。
```

```
public int rob(int[] nums);
```

题目很容易理解，而且动态规划的特征很明显。我们前文「动态规划详解」做过总结，**解决动态规划问题就是找「状态」和「选择」，仅此而已**。

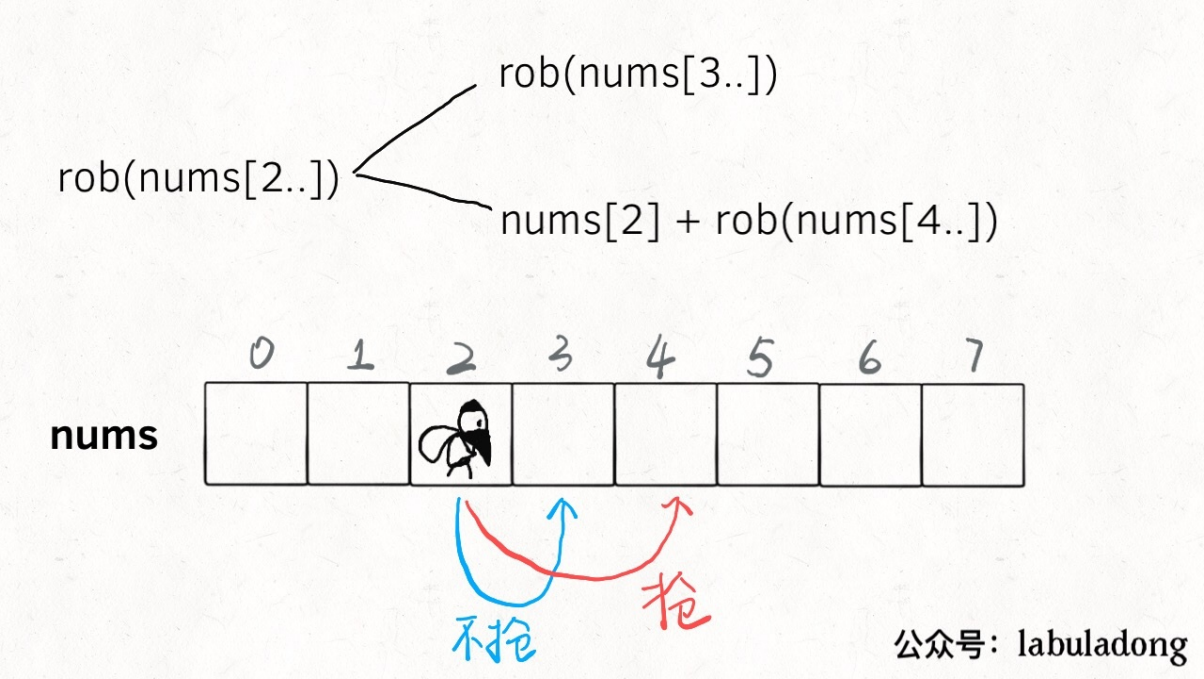
假想你就是这个专业强盗，从左到右走过这一排房子，在每间房子前都有两种**选择**：抢或者不抢。

如果你抢了这间房子，那么你**肯定**不能抢相邻的下一间房子了，只能从下下间房子开始做选择。

如果你不抢这件房子，那么你可以走到下一间房子前，继续做选择。

当你走过了最后一间房子后，你就没得抢了，能抢到的钱显然是 0（**base case**）。

以上的逻辑很简单吧，其实已经明确了「状态」和「选择」：**你面前房子的索引就是状态，抢和不抢就是选择**。

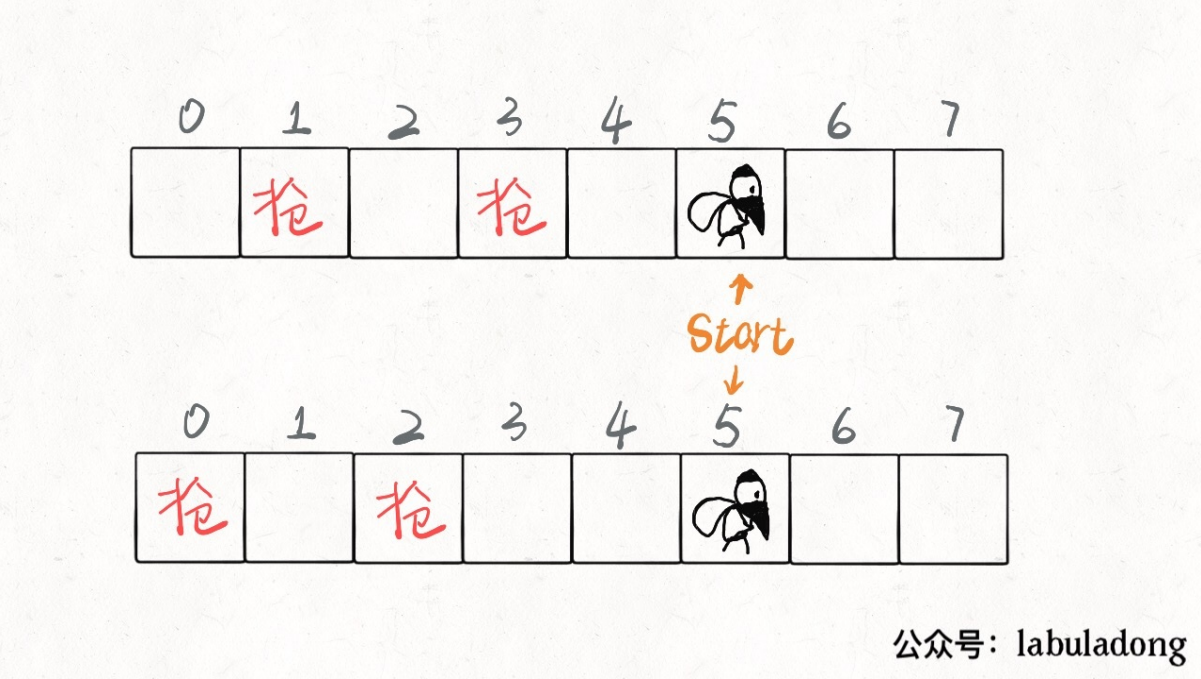


在两个选择中，每次都选更大的结果，最后得到的就是最多能抢到的 money：

```
// 主函数
public int rob(int[] nums) {
    return dp(nums, 0);
}
```

```
// 返回 nums[start..] 能抢到的最大值
private int dp(int[] nums, int start) {
    if (start >= nums.length) {
        return 0;
    }

    int res = Math.max(
            // 不抢，去下家
            dp(nums, start + 1),
            // 抢，去下下家
            nums[start] + dp(nums, start + 2)
        );
    return res;
}
```

明确了状态转移，就可以发现对于同一 `start` 位置，是存在重叠子问题的，比如下图：



盗贼有多种选择可以走到这个位置，如果每次到这都进入递归，岂不是浪费时间？所以说存在重叠子问题，可以用备忘录进行优化：

```
private int[] memo;
// 主函数
public int rob(int[] nums) {
    // 初始化备忘录
```

```
    memo = new int[nums.length];
    Arrays.fill(memo, -1);
    // 强盗从第 0 间房子开始抢劫
    return dp(nums, 0);
}

// 返回 dp[start..] 能抢到的最大值
private int dp(int[] nums, int start) {
    if (start >= nums.length) {
        return 0;
    }
    // 避免重复计算
    if (memo[start] != -1) return memo[start];

    int res = Math.max(dp(nums, start + 1),
                    nums[start] + dp(nums, start + 2));
    // 记入备忘录
    memo[start] = res;
    return res;
}
```

这就是自顶向下的动态规划解法，我们也可以略作修改，写出**自底向上**的解法：

```
int rob(int[] nums) {
    int n = nums.length;
    // dp[i] = x 表示：
    // 从第 i 间房子开始抢劫，最多能抢到的钱为 x
    // base case: dp[n] = 0
    int[] dp = new int[n + 2];
    for (int i = n - 1; i >= 0; i--) {
        dp[i] = Math.max(dp[i + 1], nums[i] + dp[i + 2]);
    }
    return dp[0];
}
```

我们又发现状态转移只和 `dp[i]` 最近的两个状态有关，所以可以进一步优化，将空间复杂度降低到 O(1)。

```
int rob(int[] nums) {
```

```
    int n = nums.length;
    // 记录 dp[i+1] 和 dp[i+2]
    int dp_i_1 = 0, dp_i_2 = 0;
    // 记录 dp[i]
    int dp_i = 0;
    for (int i = n - 1; i >= 0; i--) {
        dp_i = Math.max(dp_i_1, nums[i] + dp_i_2);
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i;
}
```

以上的流程，在我们「动态规划详解」中详细解释过，相信大家都能手到擒来了。我认为很有意思的是这个问题的 follow up，需要基于我们现在的思路做一些巧妙的应变。

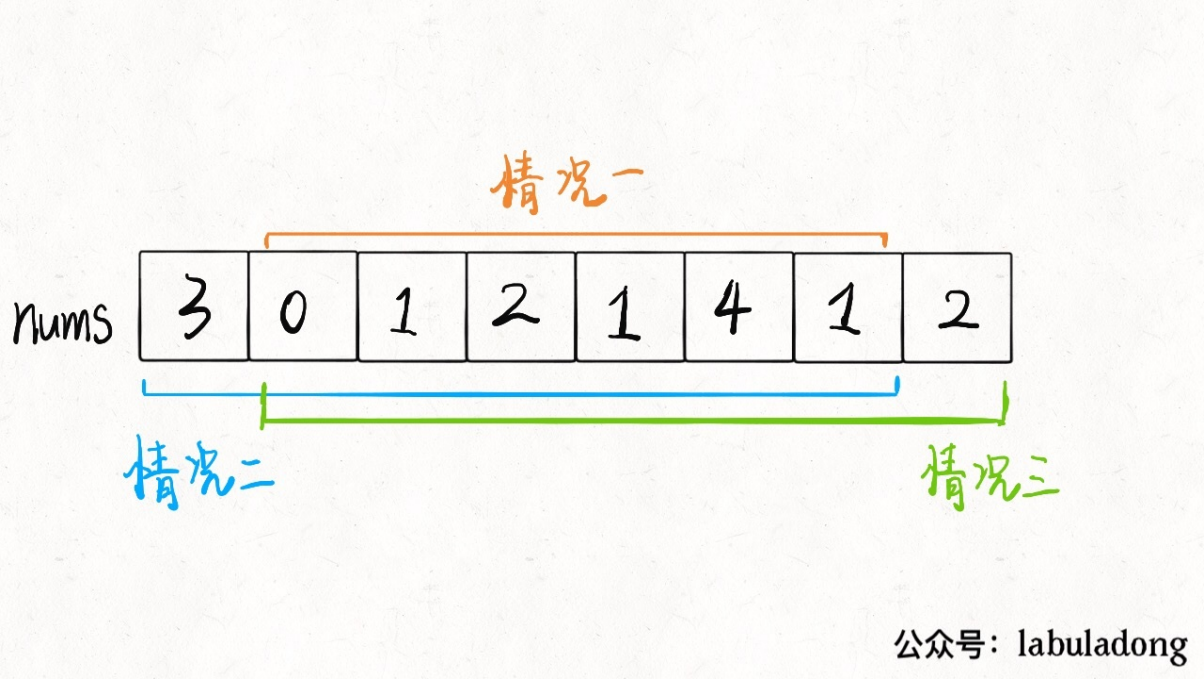
House Robber II

这道题目和第一道描述基本一样，强盗依然不能抢劫相邻的房子，输入依然是一个数组，但是告诉你**这些房子不是一排，而是围成了一个圈**。

也就是说，现在第一间房子和最后一间房子也相当于是相邻的，不能同时抢。比如说输入数组 `nums=[2,3,2]`，算法返回的结果应该是 3 而不是 4，因为开头和结尾不能同时被抢。

这个约束条件看起来应该不难解决，我们前文「单调栈解决 Next Greater Number」说过一种解决环形数组的方案，那么在这个问题上怎么处理呢？

首先，首尾房间不能同时被抢，那么只可能有三种不同情况：要么都不被抢；要么第一间房子被抢最后一间不抢；要么最后一间房子被抢第一间不抢。



那就简单了啊，这三种情况，那种的结果最大，就是最终答案呗！不过，其实我们不需要比较三种情况，只要比较情况二和情况三就行了，**因为这两种情况对于房子的选择余地比情况一大呀，房子里的钱数都是非负数，所以选择余地大，最优决策结果肯定不会小**。

所以只需对之前的解法稍作修改即可：

```
public int rob(int[] nums) {
    int n = nums.length;
    if (n == 1) return nums[0];
    return Math.max(robRange(nums, 0, n - 2),
                    robRange(nums, 1, n - 1));
}

// 仅计算闭区间 [start,end] 的最优结果
int robRange(int[] nums, int start, int end) {
    int n = nums.length;
    int dp_i_1 = 0, dp_i_2 = 0;
    int dp_i = 0;
    for (int i = end; i >= start; i--) {
        dp_i = Math.max(dp_i_1, nums[i] + dp_i_2);
        dp_i_2 = dp_i_1;
        dp_i_1 = dp_i;
    }
    return dp_i;
```

```
}
```

至此，第二问也解决了。

House Robber III

第三题又想法设法地变花样了，此强盗发现现在面对的房子不是一排，不是一圈，而是一棵二叉树！房子在二叉树的节点上，相连的两个房子不能同时被抢劫，果然是传说中的高智商犯罪：

示例 1:

```
输入: [3,2,3,null,3,null,1]

     3
    / \
   2   3
    \   \
     3   1

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

示例 2:

```
输入: [3,4,5,1,3,null,1]

     3
    / \
   4   5
  / \   \
 1   3   1

输出: 9
解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.
```

整体的思路完全没变，还是做抢或者不抢的选择，去收益较大的选择。甚至我们可以直接按这个套路写出代码：

```
Map<TreeNode, Integer> memo = new HashMap<>();
public int rob(TreeNode root) {
    if (root == null) return 0;
    // 利用备忘录消除重叠子问题
    if (memo.containsKey(root))
        return memo.get(root);
    // 抢，然后去下下家
    int do_it = root.val
        + (root.left == null ?
            0 : rob(root.left.left) + rob(root.left.right))
        + (root.right == null ?
            0 : rob(root.right.left) + rob(root.right.right));
    // 不抢，然后去下家
    int not_do = rob(root.left) + rob(root.right);

    int res = Math.max(do_it, not_do);
    memo.put(root, res);
    return res;
}
```

这道题就解决了，时间复杂度 O(N)，`N` 为数的节点数。

但是这道题让我觉得巧妙的点在于，还有更漂亮的解法。比如下面是我在评论区看到的一个解法：

```
int rob(TreeNode root) {
    int[] res = dp(root);
    return Math.max(res[0], res[1]);
}

/* 返回一个大小为 2 的数组 arr
arr[0] 表示不抢 root 的话，得到的最大钱数
arr[1] 表示抢 root 的话，得到的最大钱数 */
int[] dp(TreeNode root) {
    if (root == null)
        return new int[]{0, 0};
    int[] left = dp(root.left);
    int[] right = dp(root.right);
```

```
        // 抢，下家就不能抢了
        int rob = root.val + left[0] + right[0];
        // 不抢，下家可抢可不抢，取决于收益大小
        int not_rob = Math.max(left[0], left[1])
                    + Math.max(right[0], right[1]);

        return new int[]{not_rob, rob};
    }
```

时间复杂度 O(N)，空间复杂度只有递归函数堆栈所需的空间，不需要备忘录的额外空间。

你看他和我们的思路不一样，修改了递归函数的定义，略微修改了思路，使得逻辑自洽，依然得到了正确的答案，而且代码更漂亮。这就是我们前文「不同定义产生不同解法」所说过的动态规划问题的一个特性。

实际上，这个解法比我们的解法运行时间要快得多，虽然算法分析层面时间复杂度是相同的。原因在于此解法没有使用额外的备忘录，减少了数据操作的复杂性，所以实际运行效率会快。

动态规划之四键键盘

四键键盘问题很有意思，而且可以明显感受到：对 dp 数组的不同定义需要完全不同的逻辑，从而产生完全不同的解法。

首先看一下题目：

> 假设你有一个特殊的键盘包含下面的按键：
>
> `Key 1: (A)`：在屏幕上打印一个 'A'。
>
> `Key 2: (Ctrl-A)`：选中整个屏幕。
>
> `Key 3: (Ctrl-C)`：复制选中区域到缓冲区。
>
> `Key 4: (Ctrl-V)`：将缓冲区内容输出到上次输入的结束位置，并显示在屏幕上。
>
> 现在，你只可以按键 **N** 次（使用上述四种按键），请问屏幕上最多可以显示几个 'A'呢？
>
> **样例 1:**
>
> ```
> 输入: N = 3
> 输出: 3
> 解释:
> 我们最多可以在屏幕上显示三个'A'通过如下顺序按键：
> A, A, A
> ```
>
> **样例 2:**
>
> ```
> 输入: N = 7
> 输出: 9
> 解释:
> 我们最多可以在屏幕上显示九个'A'通过如下顺序按键：
> A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V
> ```

如何在 N 次敲击按钮后得到最多的 A？我们穷举呗，每次有对于每次按键，我们可以穷举四种可能，很明显就是一个动态规划问题。

第一种思路

这种思路会很容易理解，但是效率并不高，我们直接走流程：**对于动态规划问题，首先要明白有哪些「状态」，有哪些「选择」**。

具体到这个问题，对于每次敲击按键，有哪些「选择」是很明显的：4 种，就是题目中提到的四个按键，分别是 `A`、`C-A`、`C-C`、`C-V`（`Ctrl` 简写为 `C`）。

接下来，思考一下对于这个问题有哪些「状态」？**或者换句话说，我们需要知道什么信息，才能将原问题分解为规模更小的子问题**？

你看我这样定义三个状态行不行：第一个状态是剩余的按键次数，用 `n` 表示；第二个状态是当前屏幕上字符 A 的数量，用 `a_num` 表示；第三个状态是剪切板中字符 A 的数量，用 `copy` 表示。

如此定义「状态」，就可以知道 base case：当剩余次数 `n` 为 0 时，`a_num` 就是我们想要的答案。

结合刚才说的 4 种「选择」，我们可以把这几种选择通过状态转移表示出来：

```
dp(n - 1, a_num + 1, copy),     # A
解释：按下 A 键，屏幕上加一个字符
同时消耗 1 个操作数

dp(n - 1, a_num + copy, copy), # C-V
解释：按下 C-V 粘贴，剪切板中的字符加入屏幕
同时消耗 1 个操作数

dp(n - 2, a_num, a_num)         # C-A C-C
解释：全选和复制必然是联合使用的，
剪切板中 A 的数量变为屏幕上 A 的数量
同时消耗 2 个操作数
```

这样可以看到问题的规模 `n` 在不断减小，肯定可以到达 `n = 0` 的 base case，所以这个思路是正确的：

```python
def maxA(N: int) -> int:

    # 对于 (n, a_num, copy) 这个状态，
    # 屏幕上能最终最多能有 dp(n, a_num, copy) 个 A
    def dp(n, a_num, copy):
        # base case
        if n <= 0: return a_num;
        # 几种选择全试一遍，选择最大的结果
        return max(
                dp(n - 1, a_num + 1, copy),    # A
                dp(n - 1, a_num + copy, copy), # C-V
                dp(n - 2, a_num, a_num)        # C-A C-C
            )

    # 可以按 N 次按键，屏幕和剪切板里都还没有 A
    return dp(N, 0, 0)
```

这个解法应该很好理解，因为语义明确。下面就继续走流程，用备忘录消除一下重叠子问题：

```python
def maxA(N: int) -> int:
    # 备忘录
    memo = dict()
    def dp(n, a_num, copy):
        if n <= 0: return a_num;
        # 避免计算重叠子问题
        if (n, a_num, copy) in memo:
            return memo[(n, a_num, copy)]

        memo[(n, a_num, copy)] = max(
                # 几种选择还是一样的
            )
        return memo[(n, a_num, copy)]

    return dp(N, 0, 0)
```

这样优化代码之后，子问题虽然没有重复了，但数目仍然很多，在 LeetCode 提交会超时的。

我们尝试分析一下这个算法的时间复杂度，就会发现不容易分析。我们可以把这个 dp 函数写成 dp 数组：

```
dp[n][a_num][copy]
# 状态的总数（时空复杂度）就是这个三维数组的体积
```

我们知道变量 `n` 最多为 `N`，但是 `a_num` 和 `copy` 最多为多少我们很难计算，复杂度起码也有 O(N^3) 把。所以这个算法并不好，复杂度太高，且已经无法优化了。

这也就说明，我们这样定义「状态」是不太优秀的，下面我们换一种定义 dp 的思路。

第二种思路

这种思路稍微有点复杂，但是效率高。继续走流程，「选择」还是那 4 个，但是这次我们只定义一个「状态」，也就是剩余的敲击次数 `n`。

这个算法基于这样一个事实，**最优按键序列一定只有两种情况**：

要么一直按 `A`：A,A,...A（当 N 比较小时）。

要么是这么一个形式：A,A,...C-A,C-C,C-V,C-V,...C-V（当 N 比较大时）。

因为字符数量少（N 比较小）时，`C-A C-C C-V` 这一套操作的代价相对比较高，可能不如一个个按 `A`；而当 N 比较大时，后期 `C-V` 的收获肯定很大。这种情况下整个操作序列大致是：**开头连按几个 `A`，然后 `C-A C-C` 组合再接若干 `C-V`，然后再 `C-A C-C` 接着若干 `C-V`，循环下去**。

换句话说，最后一次按键要么是 `A` 要么是 `C-V`。明确了这一点，可以通过这两种情况来设计算法：

```
int[] dp = new int[N + 1];
// 定义：dp[i] 表示 i 次操作后最多能显示多少个 A
for (int i = 0; i <= N; i++)
    dp[i] = max(
```

```
            这次按 A 键，
            这次按 C-V
        )
```

对于「按 `A` 键」这种情况，就是状态 `i - 1` 的屏幕上新增了一个 A 而已，很容易得到结果：

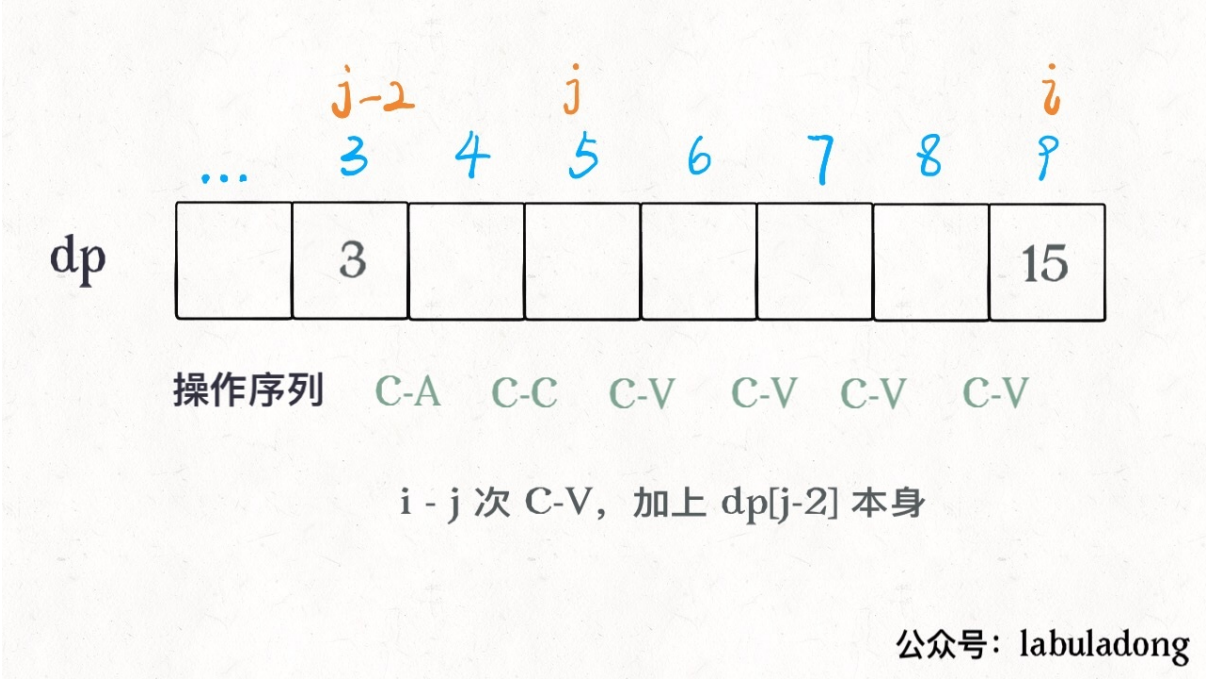
```
// 按 A 键，就比上次多一个 A 而已
dp[i] = dp[i - 1] + 1;
```

但是，如果要按 `C-V`，还要考虑之前是在哪里 `C-A C-C` 的。

刚才说了，最优的操作序列一定是 `C-A C-C` 接着若干 `C-V`，所以我们用一个变量 `j` 作为若干 `C-V` 的起点。那么 `j` 之前的 2 个操作就应该是 `C-A C-C` 了：

```
public int maxA(int N) {
    int[] dp = new int[N + 1];
    dp[0] = 0;
    for (int i = 1; i <= N; i++) {
        // 按 A 键
        dp[i] = dp[i - 1] + 1;
        for (int j = 2; j < i; j++) {
            // 全选 & 复制 dp[j-2]，连续粘贴 i - j 次
            // 屏幕上共 dp[j - 2] * (i - j + 1) 个 A
            dp[i] = Math.max(dp[i], dp[j - 2] * (i - j + 1));
        }
    }
    // N 次按键之后最多有几个 A？
    return dp[N];
}
```

其中 `j` 变量减 2 是给 `C-A C-C` 留下操作数，看个图就明白了：



这样，此算法就完成了，时间复杂度 O(N^2)，空间复杂度 O(N)，这种解法应该是比较高效的了。

最后总结

动态规划难就难在寻找状态转移，不同的定义可以产生不同的状态转移逻辑，虽然最后都能得到正确的结果，但是效率可能有巨大的差异。

回顾第一种解法，重叠子问题已经消除了，但是效率还是低，到底低在哪里呢？抽象出递归框架：

```
def dp(n, a_num, copy):
    dp(n - 1, a_num + 1, copy),    # A
    dp(n - 1, a_num + copy, copy), # C-V
    dp(n - 2, a_num, a_num)        # C-A C-C
```

看这个穷举逻辑，是有可能出现这样的操作序列 `C-A C-C, C-A C-C...` 或者 `C-V,C-V,...`。然这种操作序列的结果不是最优的，但是我们并没有想办法规避这些情况的发生，从而增加了很多没必要的子问题计算。

回顾第二种解法，我们稍加思考就能想到，最优的序列应该是这种形式：`A,A..C-A,C-C,C-V,C-V..C-A,C-C,C-V..`。

根据这个事实，我们重新定义了状态，重新寻找了状态转移，从逻辑上减少了无效的子问题个数，从而提高了算法的效率。

动态规划之正则表达

之前的文章「动态规划详解」收到了普遍的好评，今天写一个动态规划的实际应用：正则表达式。如果有读者对「动态规划」还不了解，建议先看一下上面那篇文章。

正则表达式匹配是一个很精妙的算法，而且难度也不小。本文主要写两个正则符号的算法实现：点号「.」和星号「*」，如果你用过正则表达式，应该明白他们的用法，不明白也没关系，等会会介绍。文章的最后，介绍了一种快速看出重叠子问题的技巧。

本文还有一个重要目的，就是教会读者如何设计算法。我们平时看别人的解法，直接看到一个面面俱到的完整答案，总觉得无法理解，以至觉得问题太难，自己太菜。我力求向读者展示，算法的设计是一个螺旋上升、逐步求精的过程，绝不是一步到位就能写出正确算法。本文会带你解决这个较为复杂的问题，让你明白如何化繁为简，逐个击破，从最简单的框架搭建出最终的答案。

前文无数次强调的框架思维，就是在这种设计过程中逐步培养的。下面进入正题，首先看一下题目：

给定一个字符串 (s) 和一个字符模式 (p)。实现支持 '.' 和 '*' 的正则表达式匹配。

```
'.' 匹配任意单个字符。
'*' 匹配零个或多个前面的元素。
```

匹配应该覆盖**整个**字符串 (s)，而不是部分字符串。

示例 1:

```
输入:
s = "aa"
p = "a*"
输出: true
解释: '*' 代表可匹配零个或多个前面的元素，即可以匹配 'a' 。因此，重复 'a' 一次，字符串可变为 "aa"。
```

示例 2:

```
输入:
s = "aab"
p = "c*a*b"
输出: true
解释: 'c' 可以出现零次，'a' 可以被重复一次。因此可以匹配字符串 "aab"。
```

示例 3:

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

一、热身

第一步，我们暂时不管正则符号，如果是两个普通的字符串进行比较，如何进行匹配？我想这个算法应该谁都会写：

```
bool isMatch(string text, string pattern) {
    if (text.size() != pattern.size())
        return false;
    for (int j = 0; j < pattern.size(); j++) {
        if (pattern[j] != text[j])
            return false;
    }
    return true;
}
```

然后，我稍微改造一下上面的代码，略微复杂了一点，但意思还是一样的，很容易理解吧：

```
bool isMatch(string text, string pattern) {
    int i = 0; // text 的索引位置
    int j = 0; // pattern 的索引位置
    while (j < pattern.size()) {
        if (i >= text.size())
            return false;
        if (pattern[j++] != text[i++])
            return false;
    }
    // 相等则说明完成匹配
    return j == text.size();
}
```

如上改写，是为了将这个算法改造成递归算法（伪码）：

```
def isMatch(text, pattern) -> bool:
    if pattern is empty: return (text is empty?)
    first_match = (text not empty) and pattern[0] == text[0]
    return first_match and isMatch(text[1:], pattern[1:])
```

如果你能够理解这段代码，恭喜你，你的递归思想已经到位，正则表达式算法虽然有点复杂，其实是基于这段递归代码逐步改造而成的。

二、处理点号「.」通配符

点号可以匹配任意一个字符，万金油嘛，其实是最简单的，稍加改造即可：

```
def isMatch(text, pattern) -> bool:
    if not pattern: return not text
    first_match = bool(text) and pattern[0] in {text[0], '.'}
    return first_match and isMatch(text[1:], pattern[1:])
```

三、处理「*」通配符

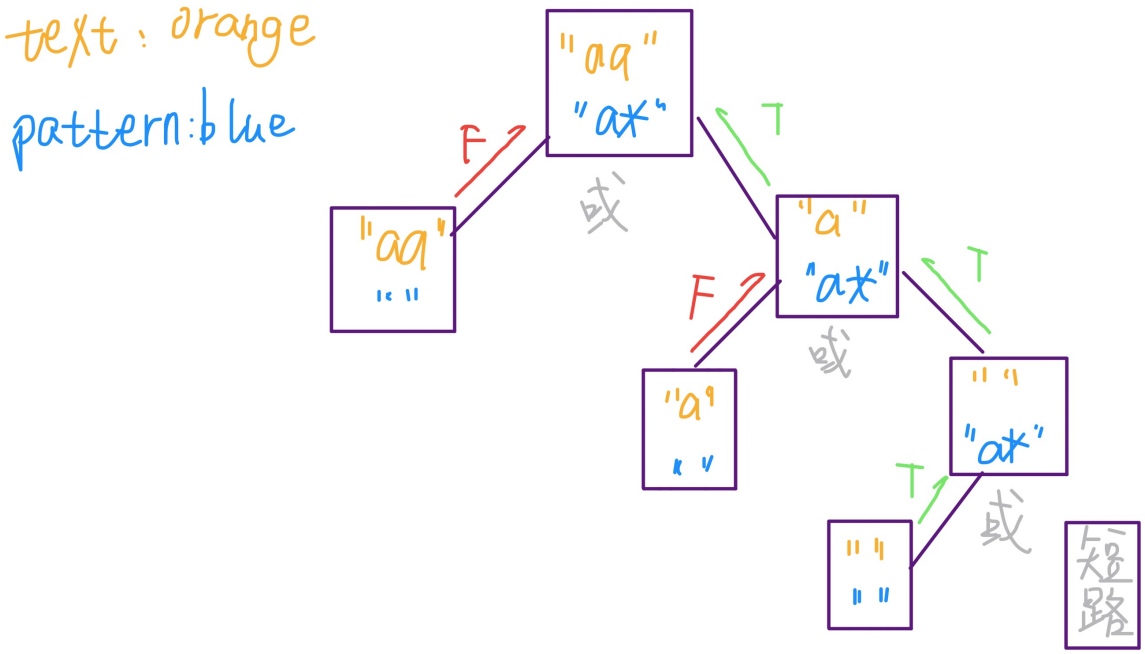
星号通配符可以让前一个字符重复任意次数，包括零次。那到底是重复几次呢？这似乎有点困难，不过不要着急，我们起码可以把框架的搭建再进一步：

```
def isMatch(text, pattern) -> bool:
    if not pattern: return not text
    first_match = bool(text) and pattern[0] in {text[0], '.'}
    if len(pattern) >= 2 and pattern[1] == '*':
        # 发现 '*' 通配符
    else:
        return first_match and isMatch(text[1:], pattern[1:])
```

星号前面的那个字符到底要重复几次呢？这需要计算机暴力穷举来算，假设重复 N 次吧。前文多次强调过，写递归的技巧是管好当下，之后的事抛给递归。具体到这里，不管 N 是多少，当前的选择只有两个：匹配 0 次、匹配 1 次。所以可以这样处理：

```
if len(pattern) >= 2 and pattern[1] == '*':
    return isMatch(text, pattern[2:]) or \
            first_match and isMatch(text[1:], pattern)
# 解释：如果发现有字符和 '*' 结合，
    # 或者匹配该字符 0 次，然后跳过该字符和 '*'
    # 或者当 pattern[0] 和 text[0] 匹配后，移动 text
```

可以看到，我们是通过保留 pattern 中的「*」，同时向后推移 text，来实现「」将字符重复匹配多次的功能。举个简单的例子就能理解这个逻辑了。假设 `*pattern* = *a* ，text = aaa`，画个图看看匹配过程：



至此，正则表达式算法就基本完成了，

四、动态规划

我选择使用「备忘录」递归的方法来降低复杂度。有了暴力解法，优化的过程及其简单，就是使用两个变量 i, j 记录当前匹配到的位置，从而避免使用子字符串切片，并且将 i, j 存入备忘录，避免重复计算即可。

我将暴力解法和优化解法放在一起，方便你对比，你可以发现优化解法无非就是把暴力解法「翻译」了一遍，加了个 memo 作为备忘录，仅此而已。

```
# 带备忘录的递归
def isMatch(text, pattern) -> bool:
    memo = dict() # 备忘录
    def dp(i, j):
        if (i, j) in memo: return memo[(i, j)]
        if j == len(pattern): return i == len(text)

        first = i < len(text) and pattern[j] in {text[i], '.'}

        if j <= len(pattern) - 2 and pattern[j + 1] == '*':
            ans = dp(i, j + 2) or \
                    first and dp(i + 1, j)
        else:
```

```
                ans = first and dp(i + 1, j + 1)

            memo[(i, j)] = ans
            return ans

        return dp(0, 0)

    # 暴力递归
    def isMatch(text, pattern) -> bool:
        if not pattern: return not text

        first = bool(text) and pattern[0] in {text[0], '.'}

        if len(pattern) >= 2 and pattern[1] == '*':
            return isMatch(text, pattern[2:]) or \
                    first and isMatch(text[1:], pattern)
        else:
            return first and isMatch(text[1:], pattern[1:])
```

有的读者也许会问，你怎么知道这个问题是个动态规划问题呢，你怎么知道它就存在「重叠子问题」呢，这似乎不容易看出来呀？

解答这个问题，最直观的应该是随便假设一个输入，然后画递归树，肯定是可以发现相同节点的。这属于定量分析，其实不用这么麻烦，下面我来教你定性分析，一眼就能看出「重叠子问题」性质。

先拿最简单的斐波那契数列举例，我们抽象出递归算法的框架：

```
def fib(n):
    fib(n - 1) #1
    fib(n - 2) #2
```

看着这个框架，请问原问题 f(n) 如何触达子问题 f(n - 2) ？有两种路径，一是 f(n) -> #1 -> #1, 二是 f(n) -> #2。前者经过两次递归，后者进过一次递归而已。两条不同的计算路径都到达了同一个问题，这就是「重叠子问题」，而且可以肯定的是，**只要你发现一条重复路径，这样的重复路径一定存在千万条，意味着巨量子问题重叠**。

同理，对于本问题，我们依然先抽象出算法框架：

```
def dp(i, j):
    dp(i, j + 2)     #1
    dp(i + 1, j)     #2
    dp(i + 1, j + 1) #3
```

提出类似的问题，请问如何从原问题 dp(i, j) 触达子问题 dp(i + 2, j + 2) ？至少有两种路径，一是 dp(i, j) -> #3 -> #3，二是 dp(i, j) -> #1 -> #2 -> #2。因此，本问题一定存在重叠子问题，一定需要动态规划的优化技巧来处理。

五、最后总结

通过本文，你深入理解了正则表达式的两种常用通配符的算法实现。其实点号「.」的实现及其简单，关键是星号「*」的实现需要用到动态规划技巧，稍微复杂些，但是也架不住我们对问题的层层拆解，逐个击破。另外，你掌握了一种快速分析「重叠子问题」性质的技巧，可以快速判断一个问题是否可以使用动态规划套路解决。

回顾整个解题过程，你应该能够体会到算法设计的流程：从简单的类似问题入手，给基本的框架逐渐组装新的逻辑，最终成为一个比较复杂、精巧的算法。所以说，读者不必畏惧一些比较复杂的算法问题，多思考多类比，再高大上的算法在你眼里也不过一个脆皮。

如果本文对你有帮助，欢迎关注我的公众号 labuladong，致力于把算法问题讲清楚～

最长公共子序列

最长公共子序列（Longest Common Subsequence，简称 LCS）是一道非常经典的面试题目，因为它的解法是典型的二维动态规划，大部分比较困难的字符串问题都和这个问题一个套路，比如说编辑距离。而且，这个算法稍加改造就可以用于解决其他问题，所以说 LCS 算法是值得掌握的。

题目就是让我们求两个字符串的 LCS 长度：

```
输入: str1 = "abcde", str2 = "ace"
输出: 3
解释: 最长公共子序列是 "ace"，它的长度是 3
```

肯定有读者会问，为啥这个问题就是动态规划来解决呢？因为子序列类型的问题，穷举出所有可能的结果都不容易，而动态规划算法做的就是穷举 + 剪枝，它俩天生一对儿。所以可以说只要涉及子序列问题，十有八九都需要动态规划来解决，往这方面考虑就对了。

下面就来手把手分析一下，这道题目如何用动态规划技巧解决。

一、动态规划思路

第一步，一定要明确 `dp` 数组的含义。对于两个字符串的动态规划问题，套路是通用的。

比如说对于字符串 `s1` 和 `s2`，一般来说都要构造一个这样的 DP table：

		0	1	2	3	4	5	6
	str1 \ str2	”	b	a	b	c	d	e
0	”	0	0	0	0	0	0	0
1	a	0	0	1	1	1	1	1
2	c	0	0	1	1	2	2	2
3	e	0	0	1	1	2	2	3

为了方便理解此表，我们暂时认为索引是从 1 开始的，待会的代码中只要稍作调整即可。其中，`dp[i][j]` 的含义是：对于 `s1[1..i]` 和 `s2[1..j]`，它们的 LCS 长度是 `dp[i][j]`。

比如上图的例子，d[2][4] 的含义就是：对于 `"ac"` 和 `"babc"`，它们的 LCS 长度是 2。我们最终想得到的答案应该是 `dp[3][6]`。

第二步，定义 base case。

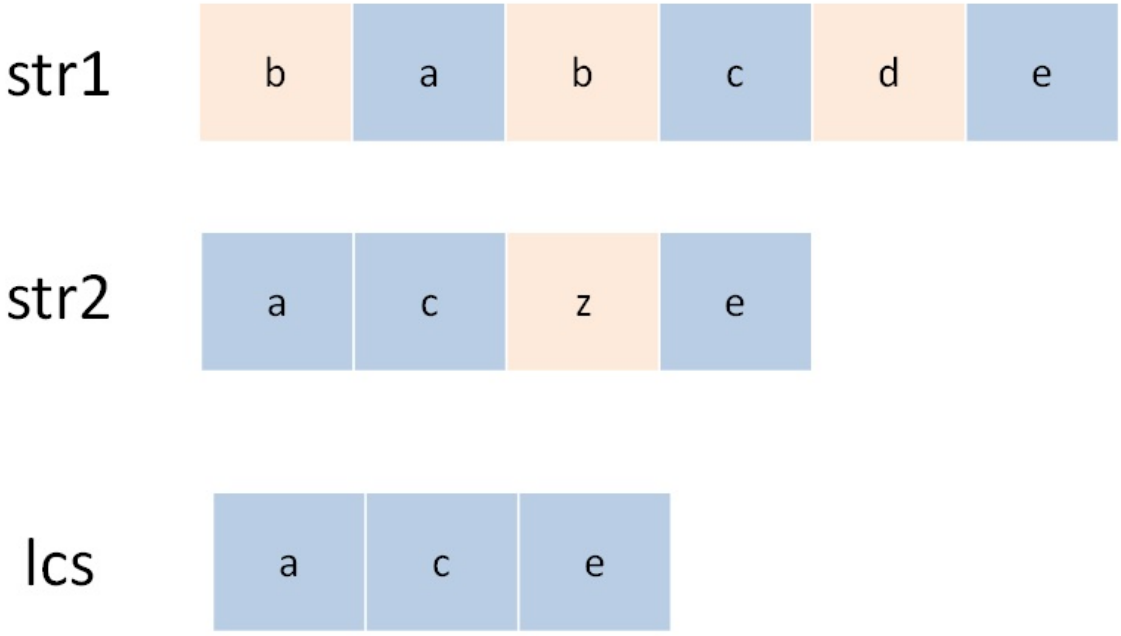
我们专门让索引为 0 的行和列表示空串，`dp[0][..]` 和 `dp[..][0]` 都应该初始化为 0，这就是 base case。

比如说，按照刚才 dp 数组的定义，`dp[0][3]=0` 的含义是：对于字符串 `""` 和 `"bab"`，其 LCS 的长度为 0。因为有一个字符串是空串，它们的最长公共子序列的长度显然应该是 0。

第三步，找状态转移方程。

这是动态规划最难的一步，不过好在这种字符串问题的套路都差不多，权且借这道题来聊聊处理这类问题的思路。

状态转移说简单些就是做选择，比如说这个问题，是求 `s1` 和 `s2` 的最长公共子序列，不妨称这个子序列为 `lcs` 。那么对于 `s1` 和 `s2` 中的每个字符，有什么选择？很简单，两种选择，要么在 `lcs` 中，要么不在。



这个「在」和「不在」就是选择，关键是，应该如何选择呢？这个需要动点脑筋：如果某个字符应该在 `lcs` 中，那么这个字符肯定同时存在于 `s1` 和 `s2` 中，因为 `lcs` 是最长**公共**子序列嘛。所以本题的思路是这样：

用两个指针 `i` 和 `j` 从后往前遍历 `s1` 和 `s2` ，如果 `s1[i]==s2[j]` ，那么这个字符一**定在 `lcs` 中**；否则的话， `s1[i]` 和 `s2[j]` 这两个字符**至少有一个不在 `lcs` 中**，需要丢弃一个。先看一下递归解法，比较容易理解：

```
def longestCommonSubsequence(str1, str2) -> int:
    def dp(i, j):
        # 空串的 base case
        if i == -1 or j == -1:
            return 0
        if str1[i] == str2[j]:
            # 这边找到一个 lcs 的元素，继续往前找
            return dp(i - 1, j - 1) + 1
        else:
            # 谁能让 lcs 最长，就听谁的
            return max(dp(i-1, j), dp(i, j-1))
```

```
    # i 和 j 初始化为最后一个索引
    return dp(len(str1)-1, len(str2)-1)
```

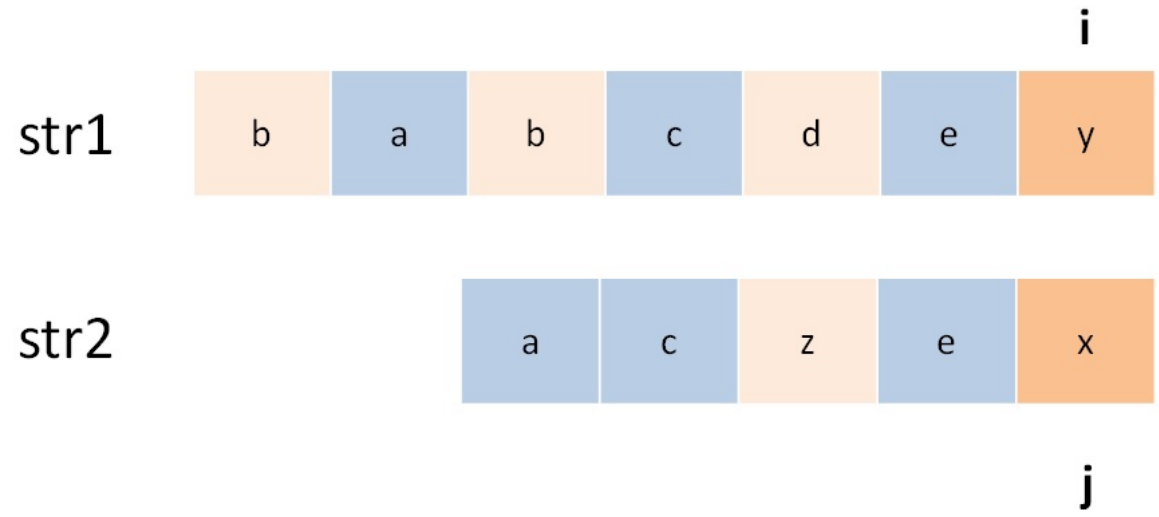
对于第一种情况，找到一个 `lcs` 中的字符，同时将 `i` `j` 向前移动一位，并给 `lcs` 的长度加一；对于后者，则尝试两种情况，取更大的结果。

其实这段代码就是暴力解法，我们可以通过备忘录或者 DP table 来优化时间复杂度，比如通过前文描述的 DP table 来解决：

```
def longestCommonSubsequence(str1, str2) -> int:
    m, n = len(str1), len(str2)
    # 构建 DP table 和 base case
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    # 进行状态转移
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if str1[i - 1] == str2[j - 1]:
                # 找到一个 lcs 中的字符
                dp[i][j] = 1 + dp[i-1][j-1]
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[-1][-1]
```

二、疑难解答

对于 `s1[i]` 和 `s2[j]` 不相等的情况，**至少有一个**字符不在 `lcs` 中，会不会两个字符都不在呢？比如下面这种情况：

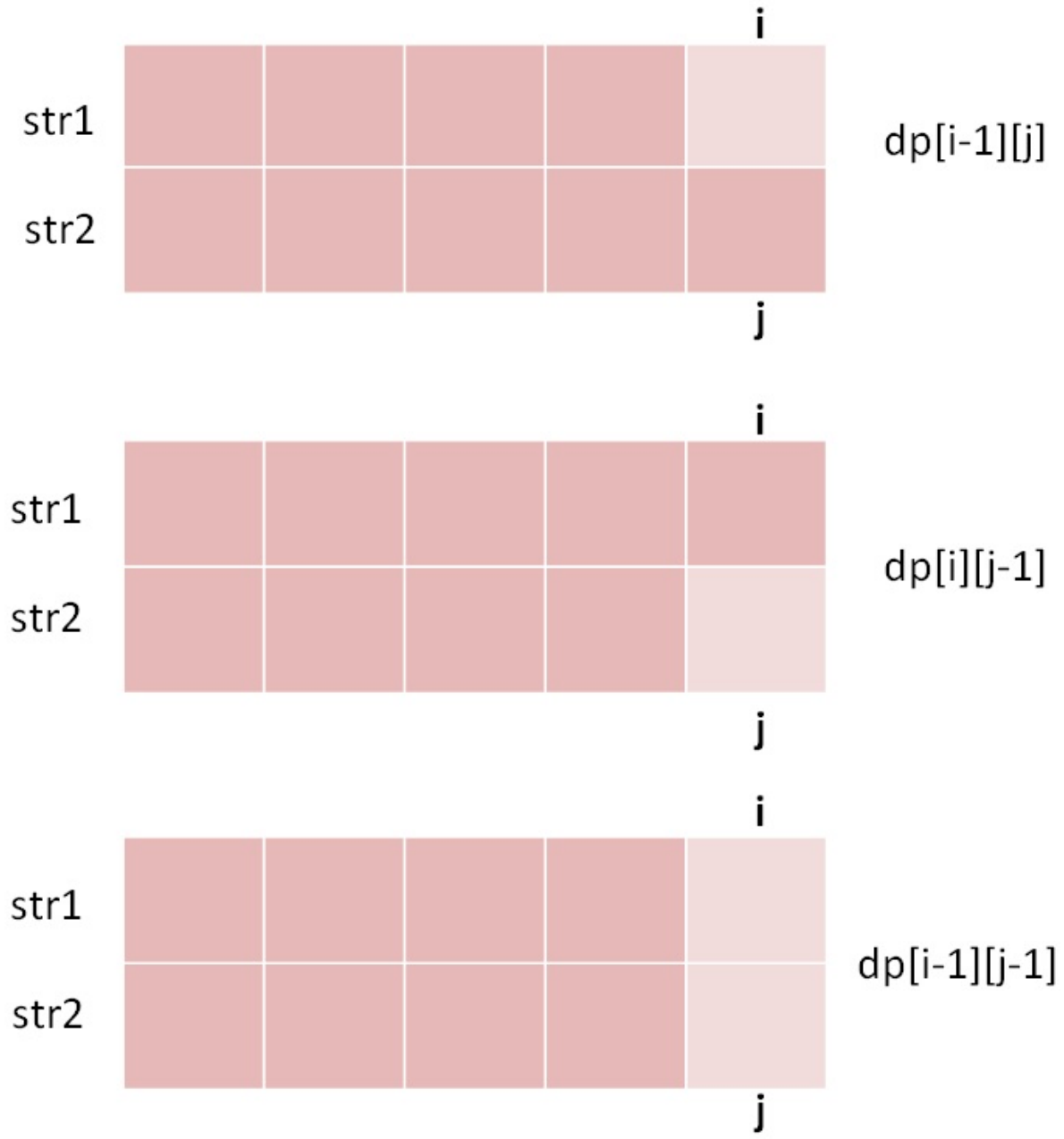


所以代码是不是应该考虑这种情况，改成这样：

```
if str1[i - 1] == str2[j - 1]:
    # ...
else:
    dp[i][j] = max(dp[i-1][j],
                   dp[i][j-1],
                   dp[i-1][j-1])
```

我一开始也有这种怀疑，其实可以这样改，也能得到正确答案，但是多此一举，因为 `dp[i-1][j-1]` 永远是三者中最小的，max 根本不可能取到它。

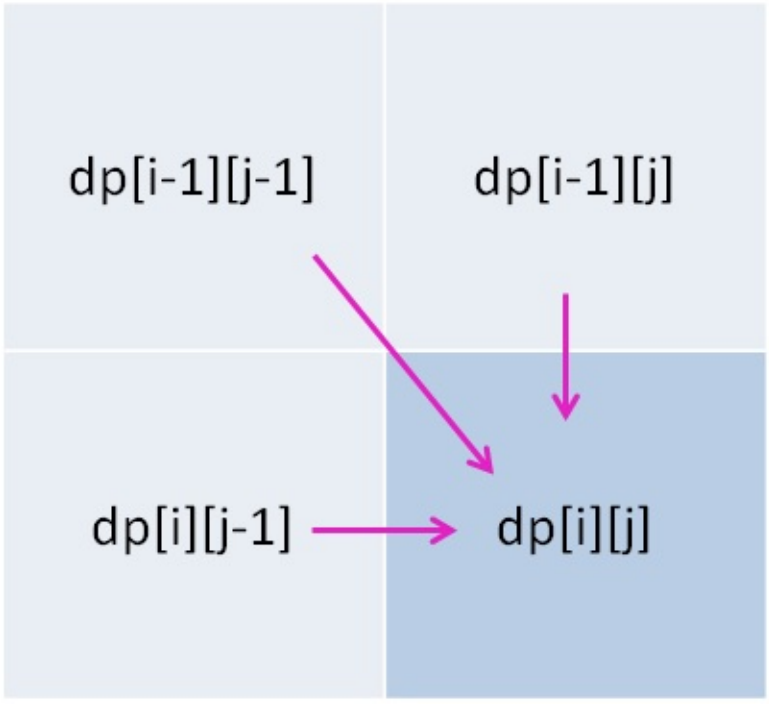
原因在于我们对 dp 数组的定义：对于 `s1[1..i]` 和 `s2[1..j]`，它们的 LCS 长度是 `dp[i][j]`。



这样一看，显然 `dp[i-1][j-1]` 对应的 `lcs` 长度不可能比前两种情况大，所以没有必要参与比较。

三、总结

对于两个字符串的动态规划问题，一般来说都是像本文一样定义 DP table，因为这样定义有一个好处，就是容易写出状态转移方程，`dp[i][j]` 的状态可以通过之前的状态推导出来：



找状态转移方程的方法是，思考每个状态有哪些「选择」，只要我们能用正确的逻辑做出正确的选择，算法就能够正确运行。

数据结构系列

这一章主要是一些特殊的数据结构设计，比如单调栈解决 Next Greater Number，单调队列解决滑动窗口问题；还有常用数据结构的操作，比如链表、树、二叉堆。

欢迎关注我的公众号 labuladong，方便获得最新的优质文章：

学习数据结构和算法的框架思维

这是好久之前的一篇文章「学习数据结构和算法的框架思维」的修订版。之前那篇文章收到广泛好评，没看过也没关系，这篇文章会涵盖之前的所有内容，并且会举很多代码的实例，教你如何使用框架思维。

首先，这里讲的都是普通的数据结构，咱不是搞算法竞赛的，野路子出生，我只会解决常规的问题。另外，以下是我个人的经验的总结，没有哪本算法书会写这些东西，所以请读者试着理解我的角度，别纠结于细节问题，因为这篇文章就是希望对数据结构和算法建立一个框架性的认识。

从整体到细节，自顶向下，从抽象到具体的框架思维是通用的，不只是学习数据结构和算法，学习其他任何知识都是高效的。

一、数据结构的存储方式

数据结构的存储方式只有两种：数组（顺序存储）和链表（链式存储）。

这句话怎么理解，不是还有散列表、栈、队列、堆、树、图等等各种数据结构吗？

我们分析问题，一定要有递归的思想，自顶向下，从抽象到具体。你上来就列出这么多，那些都属于「上层建筑」，而数组和链表才是「结构基础」。因为那些多样化的数据结构，究其源头，都是在链表或者数组上的特殊操作，API 不同而已。

比如说「队列」、「栈」这两种数据结构既可以使用链表也可以使用数组实现。用数组实现，就要处理扩容缩容的问题；用链表实现，没有这个问题，但需要更多的内存空间存储节点指针。

「图」的两种表示方法，邻接表就是链表，邻接矩阵就是二维数组。邻接矩阵判断连通性迅速，并可以进行矩阵运算解决一些问题，但是如果图比较稀疏的话很耗费空间。邻接表比较节省空间，但是很多操作的效率上肯定比不

过邻接矩阵。

「散列表」就是通过散列函数把键映射到一个大数组里。而且对于解决散列冲突的方法，拉链法需要链表特性，操作简单，但需要额外的空间存储指针；线性探查法就需要数组特性，以便连续寻址，不需要指针的存储空间，但操作稍微复杂些。

「树」，用数组实现就是「堆」，因为「堆」是一个完全二叉树，用数组存储不需要节点指针，操作也比较简单；用链表实现就是很常见的那种「树」，因为不一定是完全二叉树，所以不适合用数组存储。为此，在这种链表「树」结构之上，又衍生出各种巧妙的设计，比如二叉搜索树、AVL 树、红黑树、区间树、B 树等等，以应对不同的问题。

了解 Redis 数据库的朋友可能也知道，Redis 提供列表、字符串、集合等等几种常用数据结构，但是对于每种数据结构，底层的存储方式都至少有两种，以便于根据存储数据的实际情况使用合适的存储方式。

综上，数据结构种类很多，甚至你也可以发明自己的数据结构，但是底层存储无非数组或者链表，**二者的优缺点如下**：

数组由于是紧凑连续存储,可以随机访问，通过索引快速找到对应元素，而且相对节约存储空间。但正因为连续存储，内存空间必须一次性分配够，所以说数组如果要扩容，需要重新分配一块更大的空间，再把数据全部复制过去，时间复杂度 O(N)；而且你如果想在数组中间进行插入和删除，每次必须搬移后面的所有数据以保持连续，时间复杂度 O(N)。

链表因为元素不连续，而是靠指针指向下一个元素的位置，所以不存在数组的扩容问题；如果知道某一元素的前驱和后驱，操作指针即可删除该元素或者插入新元素，时间复杂度 O(1)。但是正因为存储空间不连续，你无法根据一个索引算出对应元素的地址，所以不能随机访问；而且由于每个元素必须存储指向前后元素位置的指针，会消耗相对更多的储存空间。

二、数据结构的基本操作

对于任何数据结构，其基本操作无非遍历 + 访问，再具体一点就是：增删查改。

数据结构种类很多，但它们存在的目的都是在不同的应用场景，尽可能高效地增删查改。话说这不就是数据结构的使命么？

如何遍历 + 访问？我们仍然从最高层来看，各种数据结构的遍历 + 访问无非两种形式：线性的和非线性的。

线性就是 for/while 迭代为代表，非线性就是递归为代表。再具体一步，无非以下几种框架：

数组遍历框架，典型的线性迭代结构：

```
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        // 迭代访问 arr[i]
    }
}
```

链表遍历框架，兼具迭代和递归结构：

```
/* 基本的单链表节点 */
class ListNode {
    int val;
    ListNode next;
}

void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {
        // 迭代访问 p.val
    }
}

void traverse(ListNode head) {
    // 递归访问 head.val
    traverse(head.next)
}
```

二叉树遍历框架，典型的非线性递归遍历结构：

```
/* 基本的二叉树节点 */
class TreeNode {
    int val;
    TreeNode left, right;
}

void traverse(TreeNode root) {
    traverse(root.left)
    traverse(root.right)
}
```

你看二叉树的递归遍历方式和链表的递归遍历方式，相似不？再看看二叉树结构和单链表结构，相似不？如果再多几条叉，N 叉树你会不会遍历？

二叉树框架可以扩展为 N 叉树的遍历框架：

```
/* 基本的 N 叉树节点 */
class TreeNode {
    int val;
    TreeNode[] children;
}

void traverse(TreeNode root) {
    for (TreeNode child : root.children)
        traverse(child)
}
```

N 叉树的遍历又可以扩展为图的遍历，因为图就是好几 N 叉棵树的结合体。你说图是可能出现环的？这个很好办，用个布尔数组 visited 做标记就行了，这里就不写代码了。

所谓框架，就是套路。不管增删查改，这些代码都是永远无法脱离的结构，你可以把这个结构作为大纲，根据具体问题在框架上添加代码就行了，下面会具体举例。

三、算法刷题指南

首先要明确的是，**数据结构是工具，算法是通过合适的工具解决特定问题的方法**。也就是说，学习算法之前，最起码得了解那些常用的数据结构，了解它们的特性和缺陷。

那么该如何在 LeetCode 刷题呢？之前的文章算法学习之路写过一些，什么按标签刷，坚持下去云云。现在距那篇文章已经过去将近一年了，我不说那些不痛不痒的话，直接说具体的建议：

先刷二叉树，先刷二叉树，先刷二叉树！

这是我这刷题一年的亲身体会，下图是去年十月份的提交截图：

公众号文章的阅读数据显示，大部分人对数据结构相关的算法文章不感兴趣，而是更关心动规回溯分治等等技巧。为什么要先刷二叉树呢，**因为二叉树是最容易培养框架思维的，而且大部分算法技巧，本质上都是树的遍历问题**。

刷二叉树看到题目没思路？根据很多读者的问题，其实大家不是没思路，只是没有理解我们说的「框架」是什么。**不要小看这几行破代码，几乎所有二叉树的题目都是一套这个框架就出来了**。

```
void traverse(TreeNode root) {
    // 前序遍历
    traverse(root.left)
    // 中序遍历
    traverse(root.right)
    // 后序遍历
}
```

比如说我随便拿几道题的解法出来，不用管具体的代码逻辑，只要看看框架在其中是如何发挥作用的就行。

LeetCode 124 题，难度 Hard，让你求二叉树中最大路径和，主要代码如下：

```
int ans = INT_MIN;
int oneSideMax(TreeNode* root) {
    if (root == nullptr) return 0;
    int left = max(0, oneSideMax(root->left));
    int right = max(0, oneSideMax(root->right));
    ans = max(ans, left + right + root->val);
    return max(left, right) + root->val;
}
```

你看，这就是个后序遍历嘛。

LeetCode 105 题，难度 Medium，让你根据前序遍历和中序遍历的结果还原一棵二叉树，很经典的问题吧，主要代码如下：

```
TreeNode buildTree(int[] preorder, int preStart, int preEnd,
    int[] inorder, int inStart, int inEnd, Map<Integer, Integer> inMa
p) {

    if(preStart > preEnd || inStart > inEnd) return null;

    TreeNode root = new TreeNode(preorder[preStart]);
    int inRoot = inMap.get(root.val);
    int numsLeft = inRoot - inStart;

    root.left = buildTree(preorder, preStart + 1, preStart + numsLeft
```

```
,
                            inorder, inStart, inRoot - 1, inMap);
    root.right = buildTree(preorder, preStart + numsLeft + 1, preEnd,

                            inorder, inRoot + 1, inEnd, inMap);
    return root;
}
```

不要看这个函数的参数很多，只是为了控制数组索引而已，本质上该算法也就是一个前序遍历。

LeetCode 99 题，难度 Hard，恢复一棵 BST，主要代码如下：

```
void traverse(TreeNode* node) {
    if (!node) return;
    traverse(node->left);
    if (node->val < prev->val) {
        s = (s == NULL) ? prev : s;
        t = node;
    }
    prev = node;
    traverse(node->right);
}
```

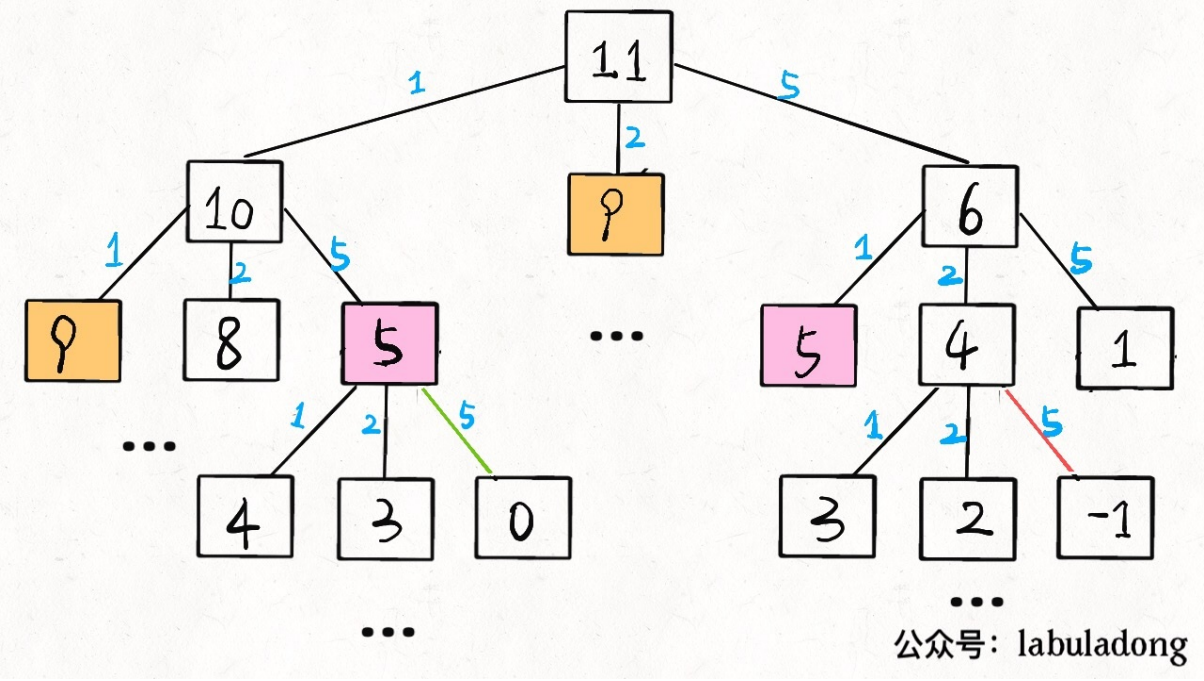
这不就是个中序遍历嘛，对于一棵 BST 中序遍历意味着什么，应该不需要解释了吧。

你看，Hard 难度的题目不过如此，而且还这么有规律可循，只要把框架写出来，然后往相应的位置加东西就行了，这不就是思路吗。

对于一个理解二叉树的人来说，刷一道二叉树的题目花不了多长时间。那么如果你对刷题无从下手或者有畏惧心理，不妨从二叉树下手，前 10 道也许有点难受；结合框架再做 20 道，也许你就有点自己的理解了；刷完整个专题，再去做什么回溯动规分治专题，**你就会发现只要涉及递归的问题，都是树的问题**。

再举例吧，说几道我们之前文章写过的问题。

动态规划详解说过凑零钱问题，暴力解法就是遍历一棵 N 叉树：




```
def coinChange(coins: List[int], amount: int):

    def dp(n):
        if n == 0: return 0
        if n < 0: return -1

        res = float('INF')
        for coin in coins:
            subproblem = dp(n - coin)
            # 子问题无解，跳过
            if subproblem == -1: continue
            res = min(res, 1 + subproblem)
        return res if res != float('INF') else -1

    return dp(amount)
```

这么多代码看不懂咋办？直接提取出框架，就能看出核心思路了：

```
# 不过是一个 N 叉树的遍历问题而已
def dp(n):
    for coin in coins:
        dp(n - coin)
```

其实很多动态规划问题就是在遍历一棵树，你如果对树的遍历操作烂熟于心，起码知道怎么把思路转化成代码，也知道如何提取别人解法的核心思路。

再看看回溯算法，前文回溯算法详解干脆直接说了，回溯算法就是个 N 叉树的前后序遍历问题，没有例外。

比如 N 皇后问题吧，主要代码如下：

```
void backtrack(int[] nums, LinkedList<Integer> track) {
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        if (track.contains(nums[i]))
            continue;
        track.add(nums[i]);
        // 进入下一层决策树
        backtrack(nums, track);
        track.removeLast();
    }

/* 提取出 N 叉树遍历框架 */
void backtrack(int[] nums, LinkedList<Integer> track) {
    for (int i = 0; i < nums.length; i++) {
        backtrack(nums, track);
}
```

N 叉树的遍历框架，找出来了把～你说，树这种结构重不重要？

综上，对于畏惧算法的朋友来说，可以先刷树的相关题目，试着从框架上看问题，而不要纠结于细节问题。

纠结细节问题，就比如纠结 i 到底应该加到 n 还是加到 n - 1，这个数组的大小到底应该开 n 还是 n + 1 ？

从框架上看问题，就是像我们这样基于框架进行抽取和扩展，既可以在看别人解法时快速理解核心逻辑，也有助于找到我们自己写解法时的思路方向。

当然，如果细节出错，你得不到正确的答案，但是只要有框架，你再错也错不到哪去，因为你的方向是对的。

但是，你要是心中没有框架，那么你根本无法解题，给了你答案，你也不会发现这就是个树的遍历问题。

这种思维是很重要的，动态规划详解中总结的找状态转移方程的几步流程，有时候按照流程写出解法，说实话我自己都不知道为啥是对的，反正它就是对了。。。

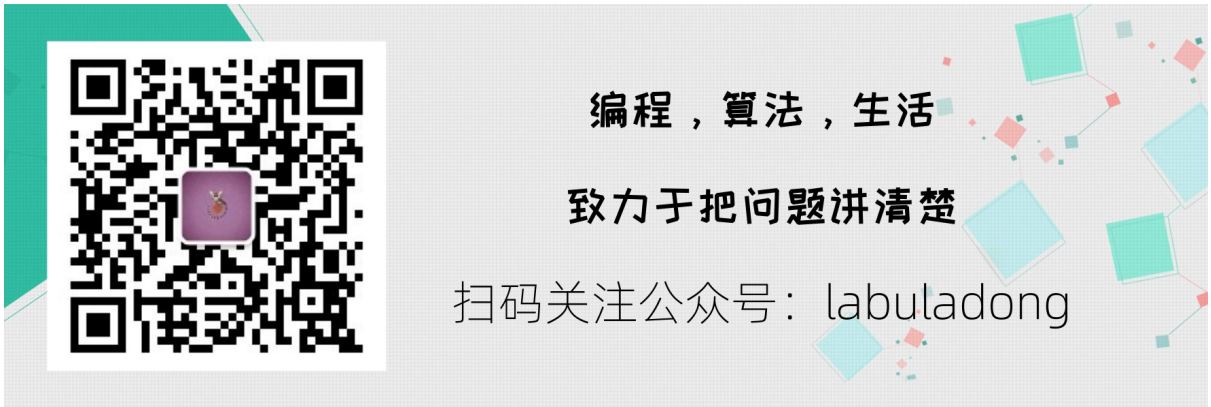
这就是框架的力量，能够保证你在快睡着的时候，依然能写出正确的程序；就算你啥都不会，都能比别人高一个级别。

四、总结几句

数据结构的基本存储方式就是链式和顺序两种，基本操作就是增删查改，遍历方式无非迭代和递归。

刷算法题建议从「树」分类开始刷，结合框架思维，把这几十道题刷完，对于树结构的理解应该就到位了。这时候去看回溯、动规、分治等算法专题，对思路的理解可能会更加深刻一些。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：



为什么我推荐《算法4》

咱们的公众号有很多硬核的算法文章，今天就聊点轻松的，就具体聊聊我非常“鼓吹”的《算法4》。这本书我在之前的文章多次推荐过，但是没有具体的介绍，今天就来正式介绍一下。。

我的推荐不会直接甩一大堆书目，而是会联系实际生活，讲一些书中有趣有用的知识，无论你最后会不会去看这本书，本文都会给你带来一些收获。

首先这本书是适合初学者的。总是有很多读者问，我只会 C 语言，能不能看《算法4》？学算法最好用什么语言？诸如此类的问题。

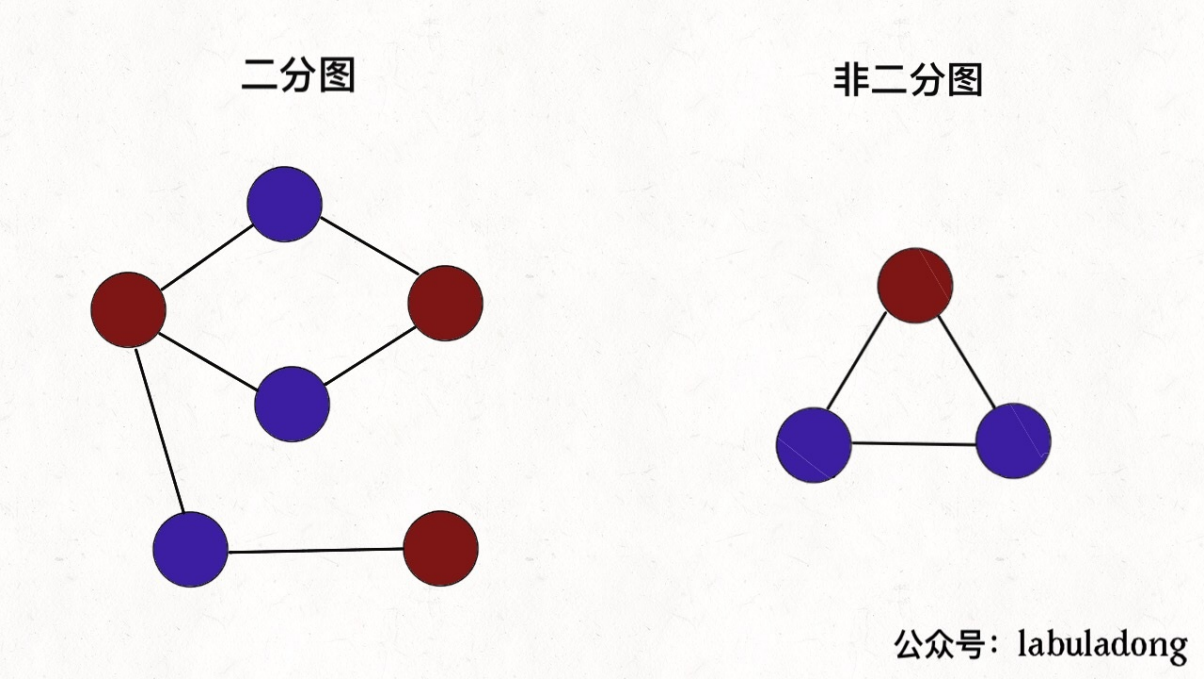
经常看咱们公众号的读者应该体会到了，算法其实是一种思维模式，和你用什么语言没啥关系。我们的文章也不会固定用某一种语言，而是什么语言写出来容易理解就用什么语言。再退一步说，到底适不适合你，网上找个 PDF 亲自看一下不就知道了？

《算法4》看起来挺厚的，但是前面几十页是教你 Java 的；每章后面还有习题，占了不少页数；每章还有一些数学证明，这些都可以忽略。这样算下来，剩下的就是基础知识和疑难解答之类的内容，含金量很高，把这些基础知识动手实践一遍，真的就可以达到不错的水平了。

我觉得这本书之所以能有这么高的评分，一个是因为讲解详细，还有大量配图，另一个原因就是书中把一些算法和现实生活中的使用场景联系起来，你不仅知道某个算法怎么实现，也知道它大概能运用到什么场景，下面我就来介绍两个图算法的简单应用。

一、二分图的应用

我想举的第一个例子是二**分图**。简单来说，二分图就是一幅拥有特殊性质的图：能够用两种颜色为所有顶点着色，使得任何一条边的两个顶点颜色不同。



明白了二分图是什么，能解决什么实际问题呢？**算法方面，常见的操作是如何判定一幅图是不是二分图**。比如说下面这道 LeetCode 题目：

给定一组 `N` 人（编号为 `1, 2, ..., N`），我们想把每个人分进**任意**大小的两组。

每个人都可能不喜欢其他人，那么他们不应该属于同一组。

形式上，如果 `dislikes[i] = [a, b]`，表示不允许将编号为 `a` 和 `b` 的人归入同一组。

当可以用这种方法将每个人分进两组时，返回 `true`；否则返回 `false`。

示例 1：

```
输入：N = 4, dislikes = [[1,2],[1,3],[2,4]]
输出：true
解释：group1 [1,4], group2 [2,3]
```

示例 2：

```
输入：N = 3, dislikes = [[1,2],[1,3],[2,3]]
输出：false
```

你想想，如果我们把每个人视为一个顶点，边代表讨厌；相互讨厌的两个人之间连接一条边，就可以形成一幅图。那么根据刚才二分图的定义，如果这幅图是一幅二分图，就说明这些人可以被分为两组，否则的话就不行。

这是判定二分图算法的一个应用，**其实二分图在数据结构方面也有一些不错的特性**。

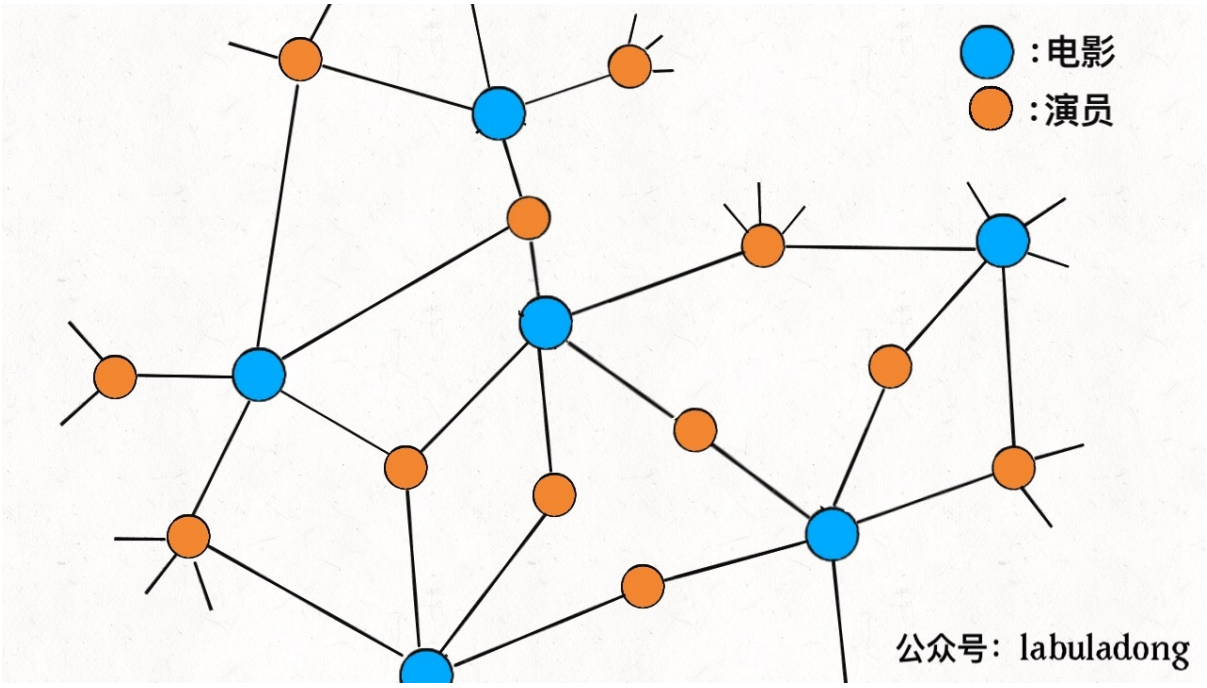
比如说我们需要一种数据结构来储存电影和演员之间的关系：某一部电影肯定是由多位演员出演的，且某一位演员可能会出演多部电影。你使用什么数据结构来存储这种关系呢？

既然是存储映射关系，最简单的不就是使用哈希表嘛，我们可以使用一个 `HashMap<String, List<String>>` 来存储电影到演员列表的映射，如果给一部电影的名字，就能快速得到出演该电影的演员。

但是如果给出一个演员的名字，我们想快速得到该演员演出的所有电影，怎么办呢？这就需要「反向索引」，对之前的哈希表进行一些操作，新建另一个哈希表，把演员作为键，把电影列表作为值。

对于上面这个例子，可以使用二分图来取代哈希表。电影和演员是具有二分图性质的：如果把电影和演员视为图中的顶点，出演关系作为边，那么与电影顶点相连的一定是演员，与演员相邻的一定是电影，不存在演员和演员相连，电影和电影相连的情况。

回顾二分图的定义，如果对演员和电影顶点着色，肯定就是一幅二分图：



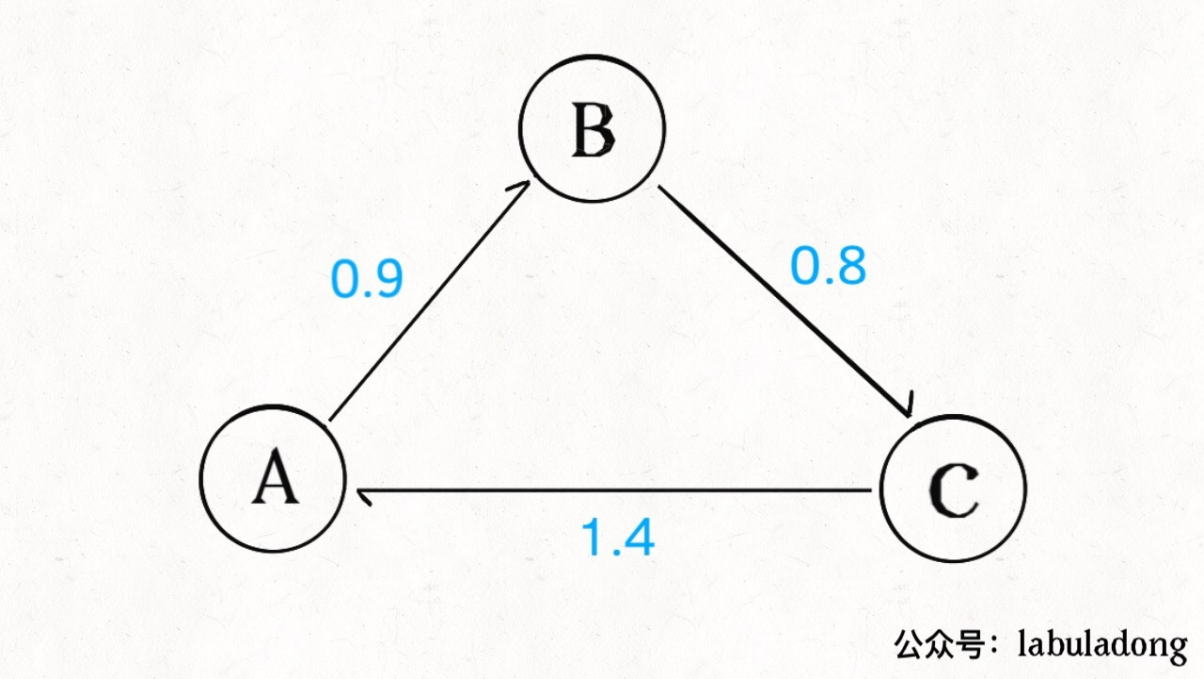
如果这幅图构建完成，就不需要反向索引，对于演员顶点，其直接连接的顶点就是他出演的电影，对于电影顶点，其直接连接的顶点就是出演演员。

当然，对于这个问题，书中还提到了一些其他有趣的玩法，比如说社交网络中「间隔度数」的计算（六度空间理论应该听说过）等等，其实就是一个 BFS 广度优先搜索寻找最短路径的问题，具体代码实现这里就不展开了。

二、套汇的算法

如果我们说货币 A 到货币 B 的汇率是 10，意思就是 1 单位的货币 A 可以换 10 单位货币 B。如果我们把每种货币视为一幅图的顶点，货币之间的汇率视为加权有向边，那么整个汇率市场就是一幅「完全加权有向图」。

一旦把现实生活中的情景抽象成图，就有可能运用算法解决一些问题。比如说图中可能存在下面的情况：



图中的加权有向边代表汇率，我们可以发现如果把 100 单位的货币 A 换成 B，再换成 C，最后换回 A，就可以得到 100×0.9×0.8×1.4 = 100.8 单位的 A！如果交易的金额大一些的话，赚的钱是很可观的，这种空手套白狼的操作就是套汇。

现实中交易会有种种限制，而且市场瞬息万变，但是套汇的利润还是很高的，关键就在于如何**快速**找到这种套汇机会呢？

借助图的抽象，我们发现套汇机会其实就是一个环，且这个环上的权重之积大于 1，只要在顺着这个环交易一圈就能空手套白狼。

图论中有一个经典算法叫做 **Bellman-Ford 算法，可以用于寻找负权重环**。对于我们说的套汇问题，可以先把所有边的权重 w 替换成 -ln(w)，这样「寻找权重乘积大于 1 的环」就转化成了「寻找权重和小于 0 的环」，就可以使用 Bellman-Ford 算法在 O(EV) 的时间内寻找负权重环，也就是寻找套汇机会。

《算法4》就介绍到这里，关于上面两个例子的具体内容，可以自己去看书，公众号后台回复关键词「算法4」就有 PDF。

三、最后说几句

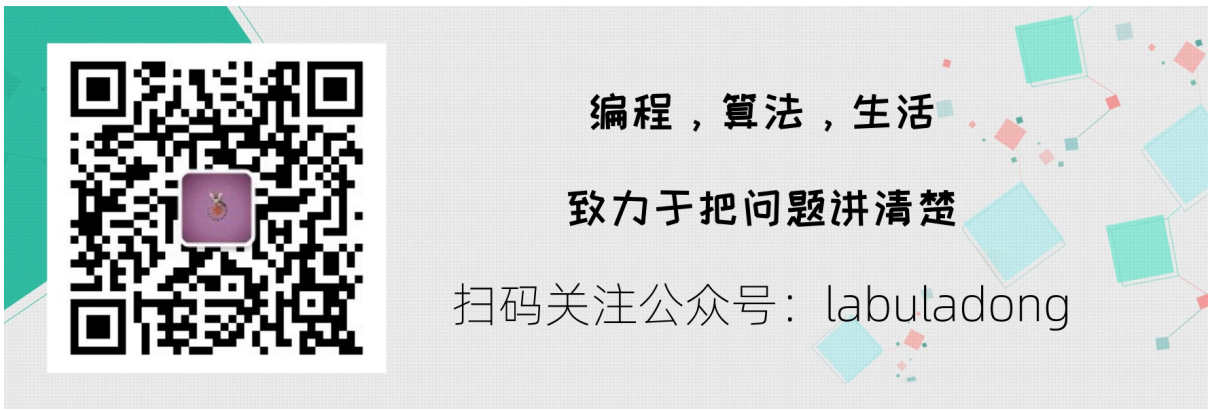
首先，前文说对于数学证明、章后习题可以忽略，可能有人要抬杠了：难道习题和数学证明不重要吗？

那我想说，就是不重要，起码对大多数人来说不重要。我觉得吧，学习就要带着目的性去学，大部分人学算法不就是巩固计算机知识，对付面试题目吗？**如果是这个目的**，那就学些基本的数据结构和经典算法，明白它们的时间复杂度，然后去刷题就好了，何必和习题、证明过不去？

这也是我从来不推荐《算法导论》这本书的原因。如果有人给你推荐这本书，只可能有两个原因，要么他是真大佬，要么他在装大佬。《算法导论》中充斥大量数学证明，而且很多数据结构是很少用到的，顶多当个字典用。你说你学了那些有啥用呢，饶过自己呗。

另外，读书在精不在多。你花时间《算法4》过个大半（最后小半部分有点困难），同时刷点题，看看咱们的公众号文章，算法这块真就够了，别对细节问题太较真。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章，公众号后台回复关键词「算法4」可以获得 PDF 下载：



二叉堆详解实现优先级队列

二叉堆（Binary Heap）没什么神秘，性质比二叉搜索树 BST 还简单。其主要操作就两个，`sink`（下沉）和`swim`（上浮），用以维护二叉堆的性质。其主要应用有两个，首先是一种排序方法「堆排序」，第二是一种很有用的数据结构「优先级队列」。

本文就以实现优先级队列（Priority Queue）为例，通过图片和人类的语言来描述一下二叉堆怎么运作的。

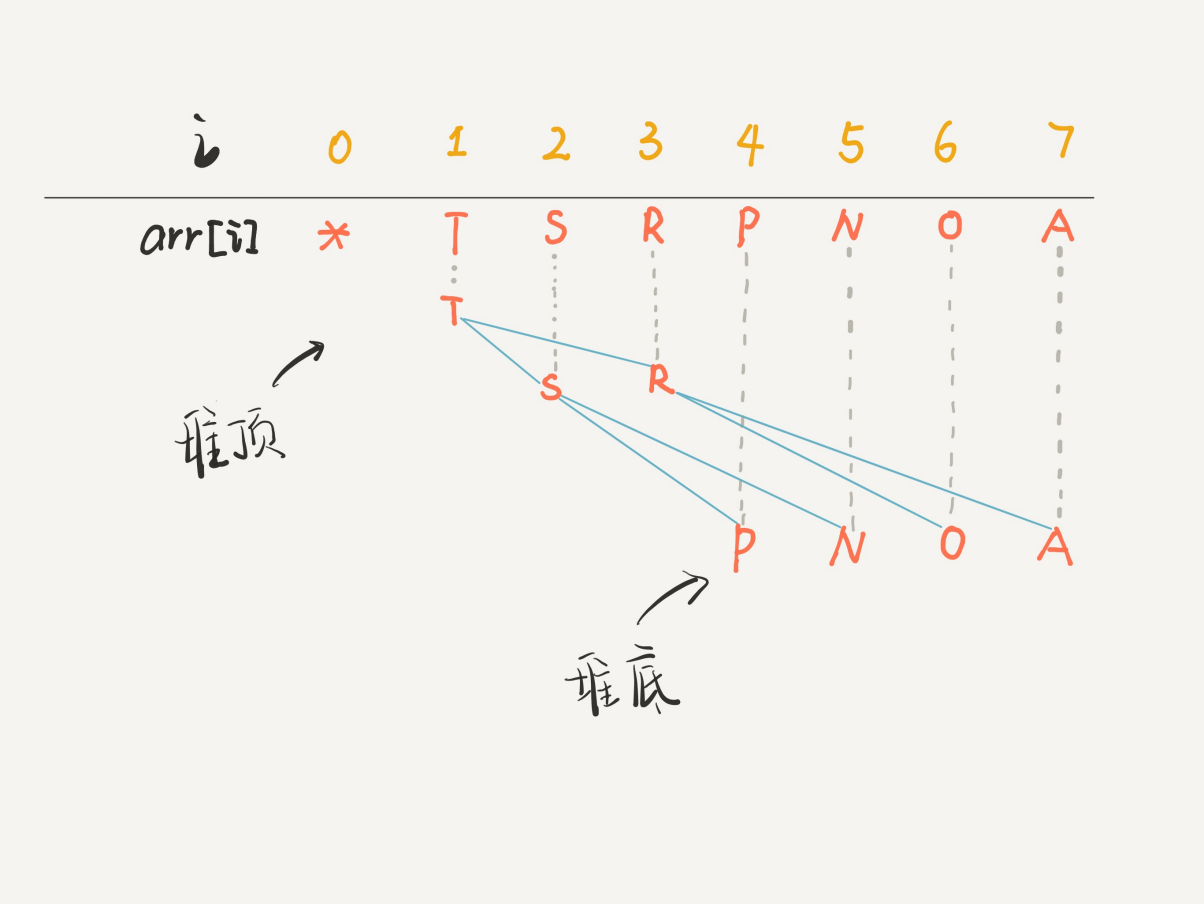
一、二叉堆概览

首先，二叉堆和二叉树有啥关系呢，为什么人们总数把二叉堆画成一棵二叉树？

因为，二叉堆其实就是一种特殊的二叉树（完全二叉树），只不过存储在数组里。一般的链表二叉树，我们操作节点的指针，而在数组里，我们把数组索引作为指针：

```
// 父节点的索引
int parent(int root) {
    return root / 2;
}
// 左孩子的索引
int left(int root) {
    return root * 2;
}
// 右孩子的索引
int right(int root) {
    return root * 2 + 1;
}
```

画个图你立即就能理解了，注意数组的第一个索引 0 空着不用，



PS：因为数组索引是数组，为了方便区分，将字符作为数组元素。

你看到了，把 arr[1] 作为整棵树的根的话，每个节点的父节点和左右孩子的索引都可以通过简单的运算得到，这就是二叉堆设计的一个巧妙之处。为了方便讲解，下面都会画的图都是二叉树结构，相信你能把树和数组对应起来。

二叉堆还分为最大堆和最小堆。**最大堆的性质是：每个节点都大于等于它的两个子节点**。类似的，最小堆的性质是：每个节点都小于等于它的子节点。

两种堆核心思路都是一样的，本文以最大堆为例讲解。

对于一个最大堆，根据其性质，显然堆顶，也就是 arr[1] 一定是所有元素中最大的元素。

二、优先级队列概览

优先级队列这种数据结构有一个很有用的功能，你插入或者删除元素的时候，元素会自动排序，这底层的原理就是二叉堆的操作。

数据结构的功能无非增删查该，优先级队列有两个主要 API，分别是 `insert` 插入一个元素和 `delMax` 删除最大元素（如果底层用最小堆，那么就是 `delMin`）。

下面我们实现一个简化的优先级队列，先看下代码框架：

PS：为了清晰起见，这里用到 Java 的泛型，`Key` 可以是任何一种可比较大小的数据类型，你可以认为它是 int、char 等。

```
public class MaxPQ
    <Key extends Comparable<Key>> {
    // 存储元素的数组
    private Key[] pq;
    // 当前 Priority Queue 中的元素个数
    private int N = 0;

    public MaxPQ(int cap) {
        // 索引 0 不用，所以多分配一个空间
        pq = (Key[]) new Comparable[cap + 1];
    }

    /* 返回当前队列中最大元素 */
    public Key max() {
        return pq[1];
    }

    /* 插入元素 e */
    public void insert(Key e) {...}

    /* 删除并返回当前队列中最大元素 */
    public Key delMax() {...}

    /* 上浮第 k 个元素，以维护最大堆性质 */
    private void swim(int k) {...}

    /* 下沉第 k 个元素，以维护最大堆性质 */
    private void sink(int k) {...}
```

```
    /* 交换数组的两个元素 */
    private void exch(int i, int j) {
        Key temp = pq[i];
        pq[i] = pq[j];
        pq[j] = temp;
    }

    /* pq[i] 是否比 pq[j] 小？ */
    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    /* 还有 left, right, parent 三个方法 */
}
```

空出来的四个方法是二叉堆和优先级队列的奥妙所在，下面用图文来逐个理解。

三、实现 swim 和 sink

为什么要有上浮 swim 和下沉 sink 的操作呢？为了维护堆结构。

我们要讲的是最大堆，每个节点都比它的两个子节点大，但是在插入元素和删除元素时，难免破坏堆的性质，这就需要通过这两个操作来恢复堆的性质了。

对于最大堆，会破坏堆性质的有有两种情况：

1. 如果某个节点 A 比它的子节点（中的一个）小，那么 A 就不配做父节点，应该下去，下面那个更大的节点上来做父节点，这就是对 A 进行**下沉**。

2. 如果某个节点 A 比它的父节点大，那么 A 不应该做子节点，应该把父节点换下来，自己去做父节点，这就是对 A 的**上浮**。

当然，错位的节点 A 可能要上浮（或下沉）很多次，才能到达正确的位置，恢复堆的性质。所以代码中肯定有一个 `while` 循环。

细心的读者也许会问，这两个操作不是互逆吗，所以上浮的操作一定能用下沉来完成，为什么我还要费劲写两个方法？

是的，操作是互逆等价的，但是最终我们的操作只会在堆底和堆顶进行（等会讲原因），显然堆底的「错位」元素需要上浮，堆顶的「错位」元素需要下沉。

上浮的代码实现：

```
private void swim(int k) {
    // 如果浮到堆顶，就不能再上浮了
    while (k > 1 && less(parent(k), k)) {
        // 如果第 k 个元素比上层大
        // 将 k 换上去
        exch(parent(k), k);
        k = parent(k);
    }
}
```

画个 GIF 看一眼就明白了：

【pdf/mobi格式不支持GIF:heap/swim.gif】 请查看【关于本小抄及作者】章节的解决方案

下沉的代码实现：

下沉比上浮略微复杂一点，因为上浮某个节点 A，只需要 A 和其父节点比较大小即可；但是下沉某个节点 A，需要 A 和其**两个子节点**比较大小，如果 A 不是最大的就需要调整位置，要把较大的那个子节点和 A 交换。

```
private void sink(int k) {
    // 如果沉到堆底，就沉不下去了
    while (left(k) <= N) {
        // 先假设左边节点较大
        int older = left(k);
        // 如果右边节点存在，比一下大小
        if (right(k) <= N && less(older, right(k)))
            older = right(k);
        // 结点 k 比俩孩子都大，就不必下沉了
```

```
            if (less(older, k)) break;
            // 否则，不符合最大堆的结构，下沉 k 结点
            exch(k, older);
            k = older;
        }
    }
```

画个 GIF 看下就明白了：

【pdf/mobi格式不支持GIF:heap/sink.gif】 请查看【关于本小抄及作者】章节的解决方案

至此，二叉堆的主要操作就讲完了，一点都不难吧，代码加起来也就十行。明白了 `sink` 和 `swim` 的行为，下面就可以实现优先级队列了。

四、实现 delMax 和 insert

这两个方法就是建立在 `swim` 和 `sink` 上的。

`insert` 方法先把要插入的元素添加到堆底的最后，然后让其上浮到正确位置。

【pdf/mobi格式不支持GIF:heap/insert.gif】 请查看【关于本小抄及作者】章节的解决方案

```
public void insert(Key e) {
    N++;
    // 先把新元素加到最后
    pq[N] = e;
    // 然后让它上浮到正确的位置
    swim(N);
}
```

`delMax` 方法先把堆顶元素 A 和堆底最后的元素 B 对调，然后删除 A，最后让 B 下沉到正确位置。

```
public Key delMax() {
```

```
        // 最大堆的堆顶就是最大元素
        Key max = pq[1];
        // 把这个最大元素换到最后，删除之
        exch(1, N);
        pq[N] = null;
        N--;
        // 让 pq[1] 下沉到正确位置
        sink(1);
        return max;
    }
```

【pdf/mobi格式不支持GIF:heap/delete.gif】 请查看【关于本小抄及作者】章节的解决方案

至此，一个优先级队列就实现了，插入和删除元素的时间复杂度为 $O(logK)$，K 为当前二叉堆（优先级队列）中的元素总数。因为我们时间复杂度主要花费在 `sink` 或者 `swim` 上，而不管上浮还是下沉，最多也就树（堆）的高度，也就是 log 级别。

五、最后总结

二叉堆就是一种完全二叉树，所以适合存储在数组中，而且二叉堆拥有一些特殊性质。

二叉堆的操作很简单，主要就是上浮和下沉，来维护堆的性质（堆有序），核心代码也就十行。

优先级队列是基于二叉堆实现的，主要操作是插入和删除。插入是先插到最后，然后上浮到正确位置；删除是调换位置后再删除，然后下沉到正确位置。核心代码也就十行。

也许这就是数据结构的威力，简单的操作就能实现巧妙的功能，真心佩服发明二叉堆算法的人！

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

LRU算法详解

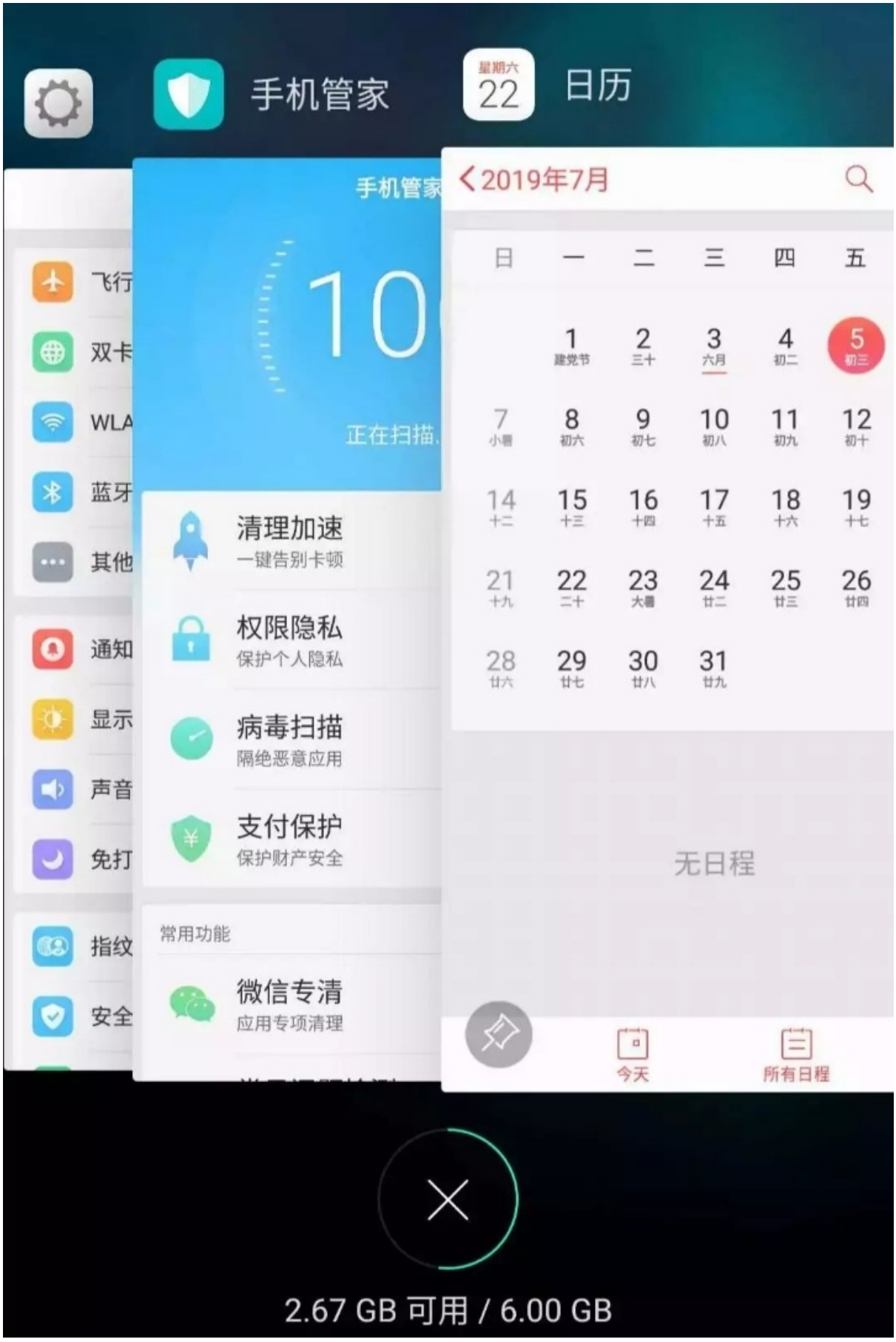
一、什么是 LRU 算法

就是一种缓存淘汰策略。

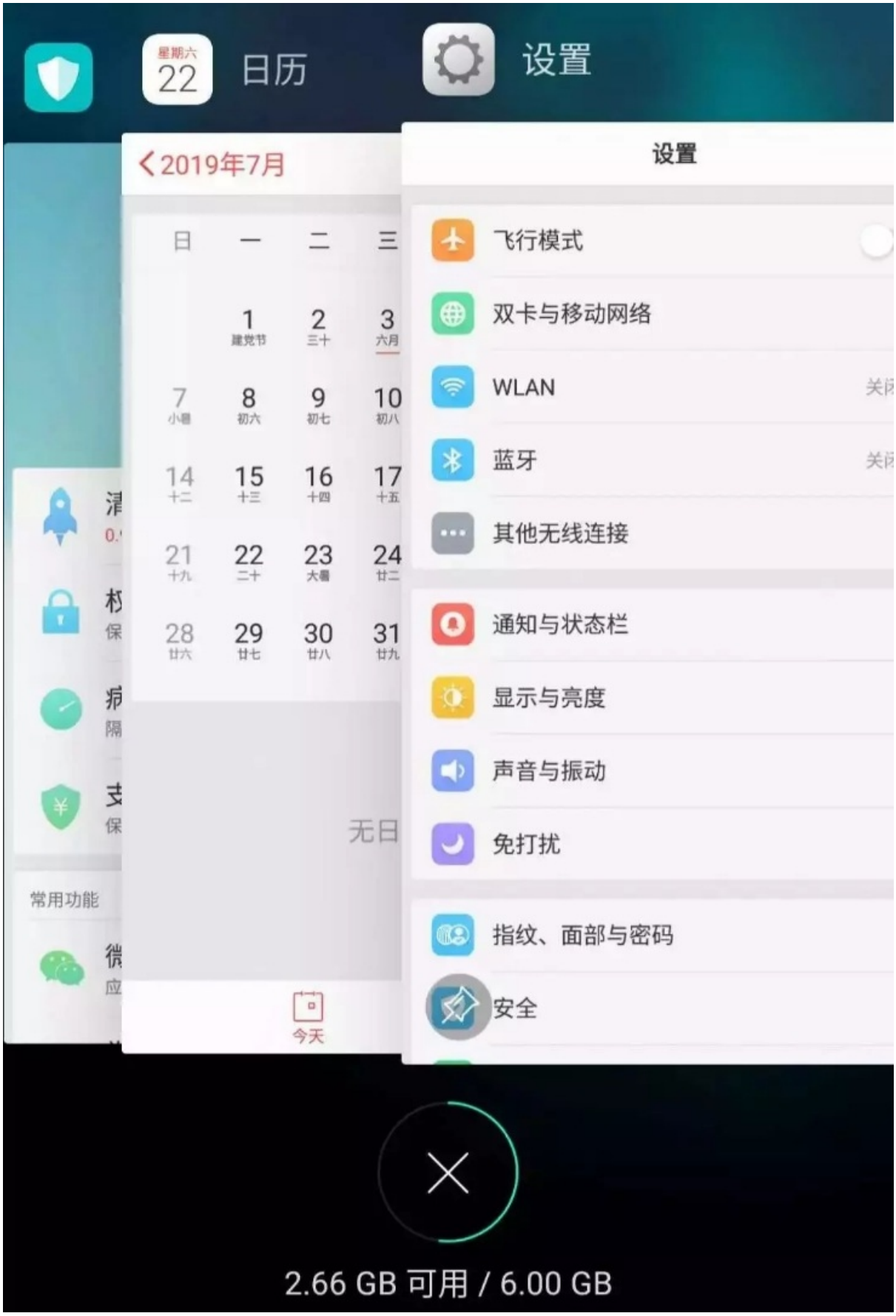
计算机的缓存容量有限，如果缓存满了就要删除一些内容，给新内容腾位置。但问题是，删除哪些内容呢？我们肯定希望删掉哪些没什么用的缓存，而把有用的数据继续留在缓存里，方便之后继续使用。那么，什么样的数据，我们判定为「有用的」的数据呢？

LRU 缓存淘汰算法就是一种常用策略。LRU 的全称是 Least Recently Used，也就是说我们认为最近使用过的数据应该是是「有用的」，很久都没用过的数据应该是无用的，内存满了就优先删那些很久没用过的数据。

举个简单的例子，安卓手机都可以把软件放到后台运行，比如我先后打开了「设置」「手机管家」「日历」，那么现在他们在后台排列的顺序是这样的：

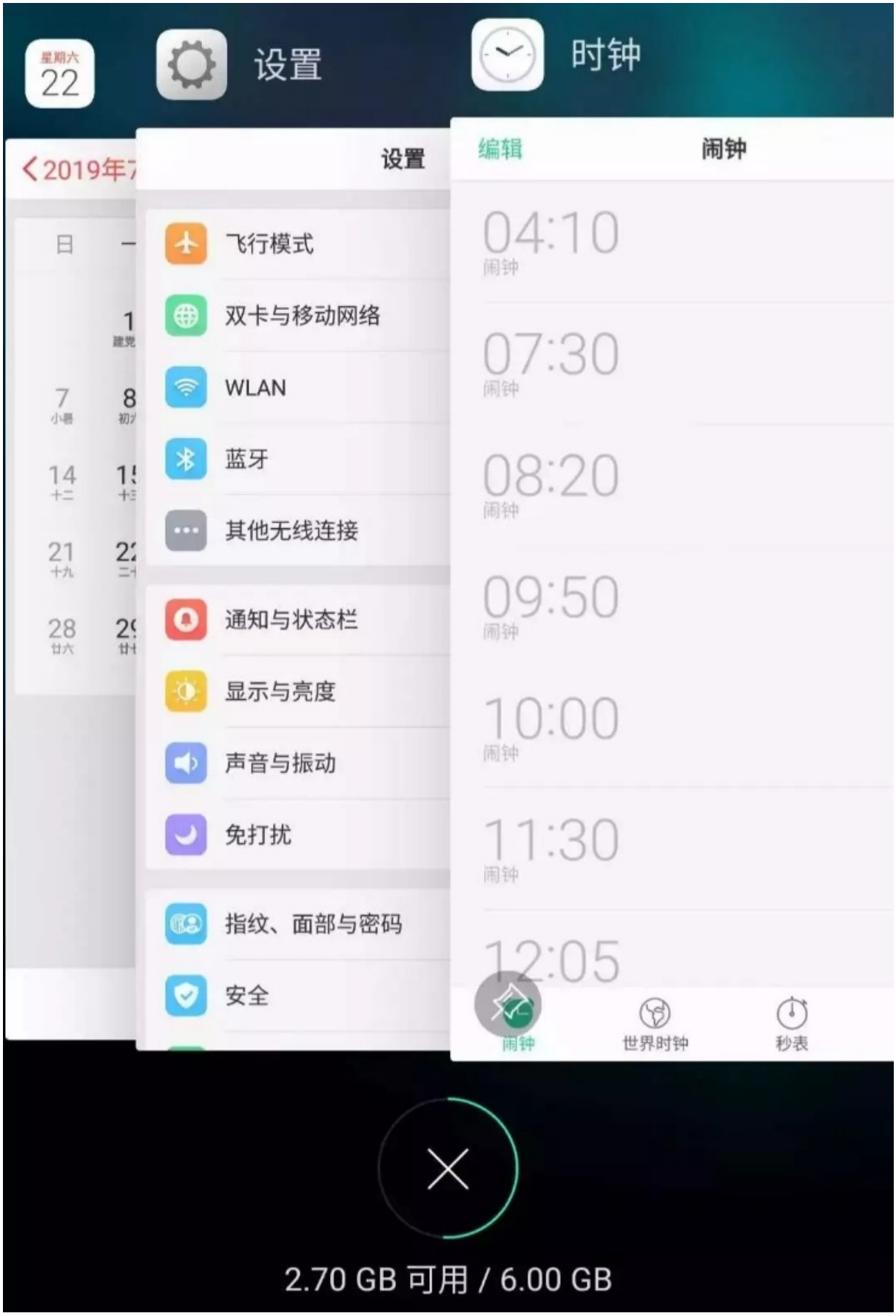

手机管家
星期六
22
日历
手机管家
2019年7月
日 一 二 三 四 五
1 建党节
2 三十
3 六月
4 初二
5 初三
7 小暑
8 初六
9 初七
10 初八
11 初九
12 初十
14 十二
15 十三
16 十四
17 十五
18 十六
19 十七
21 十九
22 二十
23 大暑
24 廿二
25 廿三
26 廿四
28 廿六
29 廿七
30 廿八
31 廿九
无日程
今天
所有日程
飞行
双卡
WLA
蓝牙
其他
通知
显示
声音
免打
指纹
安全
正在扫描.
清理加速
一键告别卡顿
权限隐私
保护个人隐私
病毒扫描
隔绝恶意应用
支付保护
保护财产安全
常用功能
微信专清
应用专项清理
2.67 GB 可用 / 6.00 GB

但是这时候如果我访问了一下「设置」界面，那么「设置」就会被提前到第一个，变成这样：


星期六
22
日历
设置
2019年7月
日 一 二 三
1 建党节
2 三十
3 六月
7 小暑
8 初六
9 初七
10 初八
14 十二
15 十三
16 十四
17 十五
21 十九
22 二十
23 大暑
24 廿二
28 廿六
29 廿七
30 廿八
31 廿九
无日
今天
常用功能
设置
飞行模式
双卡与移动网络
WLAN
关闭
蓝牙
关闭
其他无线连接
通知与状态栏
显示与亮度
声音与振动
免打扰
指纹、面部与密码
安全
2.66 GB 可用 / 6.00 GB

假设我的手机只允许我同时开 3 个应用程序，现在已经满了。那么如果我新开了一个应用「时钟」，就必须关闭一个应用为「时钟」腾出一个位置，关那个呢？

按照 LRU 的策略，就关最底下的「手机管家」，因为那是最久未使用的，然后把新开的应用放到最上面：


星期六
22
设置
时钟
设置
编辑
闹钟
飞行模式
双卡与移动网络
WLAN
蓝牙
其他无线连接
通知与状态栏
显示与亮度
声音与振动
免打扰
指纹、面部与密码
安全
04:10
闹钟
07:30
闹钟
08:20
闹钟
09:50
闹钟
10:00
闹钟
11:30
闹钟
12:05
闹钟
世界时钟
秒表
2.70 GB 可用 / 6.00 GB

现在你应该理解 LRU（Least Recently Used）策略了。当然还有其他缓存淘汰策略，比如不要按访问的时序来淘汰，而是按访问频率（LFU 策略）来淘汰等等，各有应用场景。本文讲解 LRU 算法策略。

二、LRU 算法描述

LRU 算法实际上是让你设计数据结构：首先要接收一个 capacity 参数作为缓存的最大容量，然后实现两个 API，一个是 put(key, val) 方法存入键值对，另一个是 get(key) 方法获取 key 对应的 val，如果 key 不存在则返回 -1。

注意哦，get 和 put 方法必须都是 $O(1)$ 的时间复杂度，我们举个具体例子来看看 LRU 算法怎么工作。

```
/* 缓存容量为 2 */
LRUCache cache = new LRUCache(2);
// 你可以把 cache 理解成一个队列
// 假设左边是队头，右边是队尾
// 最近使用的排在队头，久未使用的排在队尾
// 圆括号表示键值对 (key, val)

cache.put(1, 1);
// cache = [(1, 1)]
cache.put(2, 2);
// cache = [(2, 2), (1, 1)]
cache.get(1);       // 返回 1
// cache = [(1, 1), (2, 2)]
// 解释：因为最近访问了键 1，所以提前至队头
// 返回键 1 对应的值 1
cache.put(3, 3);
// cache = [(3, 3), (1, 1)]
// 解释：缓存容量已满，需要删除内容空出位置
// 优先删除久未使用的数据，也就是队尾的数据
// 然后把新的数据插入队头
cache.get(2);       // 返回 -1 (未找到)
// cache = [(3, 3), (1, 1)]
// 解释：cache 中不存在键为 2 的数据
cache.put(1, 4);
// cache = [(1, 4), (3, 3)]
```

```
// 解释：键 1 已存在，把原始值 1 覆盖为 4
// 不要忘了也要将键值对提前到队头
```

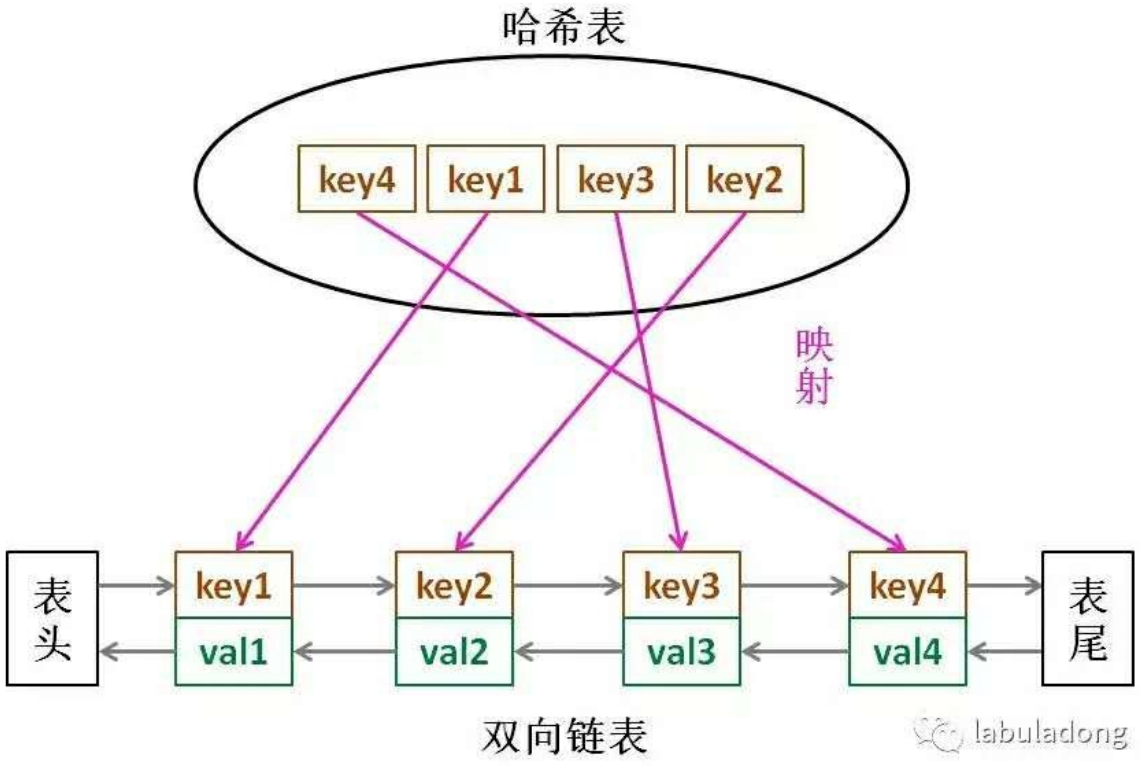
三、LRU 算法设计

分析上面的操作过程，要让 put 和 get 方法的时间复杂度为 O(1)，我们可以总结出 cache 这个数据结构必要的条件：查找快，插入快，删除快，有顺序之分。

因为显然 cache 必须有顺序之分，以区分最近使用的和久未使用的数据；而且我们要在 cache 中查找键是否已存在；如果容量满了要删除最后一个数据；每次访问还要把数据插入到队头。

那么，什么数据结构同时符合上述条件呢？哈希表查找快，但是数据无固定顺序；链表有顺序之分，插入删除快，但是查找慢。所以结合一下，形成一种新的数据结构：哈希链表。

LRU 缓存算法的核心数据结构就是哈希链表，双向链表和哈希表的结合体。这个数据结构长这样：



思想很简单，就是借助哈希表赋予了链表快速查找的特性嘛：可以快速查找某个 key 是否存在缓存（链表）中，同时可以快速删除、添加节点。回想刚才的例子，这种数据结构是不是完美解决了 LRU 缓存的需求？

也许读者会问，为什么要是双向链表，单链表行不行？另外，既然哈希表中已经存了 key，为什么链表中还要存键值对呢，只存值不就行了？

想的时候都是问题，只有做的时候才有答案。这样设计的原因，必须等我们亲自实现 LRU 算法之后才能理解，所以我们开始看代码吧～

四、代码实现

很多编程语言都有内置的哈希链表或者类似 LRU 功能的库函数，但是为了帮大家理解算法的细节，我们用 Java 自己造轮子实现一遍 LRU 算法。

首先，我们把双链表的节点类写出来，为了简化，key 和 val 都认为是 int 类型：

```
class Node {
    public int key, val;
    public Node next, prev;
    public Node(int k, int v) {
        this.key = k;
        this.val = v;
    }
}
```

然后依靠我们的 Node 类型构建一个双链表，实现几个需要的 API（这些操作的时间复杂度均为 $O(1)$）：

```
class DoubleList {
    // 在链表头部添加节点 x，时间 O(1)
    public void addFirst(Node x);

    // 删除链表中的 x 节点（x 一定存在）
    // 由于是双链表且给的是目标 Node 节点，时间 O(1)
    public void remove(Node x);
```

```
    // 删除链表中最后一个节点，并返回该节点，时间 O(1)
    public Node removeLast();

    // 返回链表长度，时间 O(1)
    public int size();
}
```

PS：这就是普通双向链表的实现，为了让读者集中精力理解 LRU 算法的逻辑，就省略链表的具体代码。

到这里就能回答刚才“为什么必须要用双向链表”的问题了，因为我们需要删除操作。删除一个节点不光要得到该节点本身的指针，也需要操作其前驱节点的指针，而双向链表才能支持直接查找前驱，保证操作的时间复杂度 $O(1)$。

有了双向链表的实现，我们只需要在 LRU 算法中把它和哈希表结合起来即可。我们先把逻辑理清楚：

```
// key 映射到 Node(key, val)
HashMap<Integer, Node> map;
// Node(k1, v1) <-> Node(k2, v2)...
DoubleList cache;

int get(int key) {
    if (key 不存在) {
        return -1;
    } else {
        将数据 (key, val) 提到开头；
        return val;
    }
}

void put(int key, int val) {
    Node x = new Node(key, val);
    if (key 已存在) {
        把旧的数据删除；
        将新节点 x 插入到开头；
    } else {
        if (cache 已满) {
```

```
                删除链表的最后一个数据腾位置；
                删除 map 中映射到该数据的键；
            }
            将新节点 x 插入到开头；
            map 中新建 key 对新节点 x 的映射；
        }
    }
```

如果能够看懂上述逻辑，翻译成代码就很容易理解了：

```
class LRUCache {
    // key -> Node(key, val)
    private HashMap<Integer, Node> map;
    // Node(k1, v1) <-> Node(k2, v2)...
    private DoubleList cache;
    // 最大容量
    private int cap;

    public LRUCache(int capacity) {
        this.cap = capacity;
        map = new HashMap<>();
        cache = new DoubleList();
    }

    public int get(int key) {
        if (!map.containsKey(key))
            return -1;
        int val = map.get(key).val;
        // 利用 put 方法把该数据提前
        put(key, val);
        return val;
    }

    public void put(int key, int val) {
        // 先把新节点 x 做出来
        Node x = new Node(key, val);

        if (map.containsKey(key)) {
            // 删除旧的节点，新的插到头部
            cache.remove(map.get(key));
            cache.addFirst(x);
            // 更新 map 中对应的数据
```

```
            map.put(key, x);
        } else {
            if (cap == cache.size()) {
                // 删除链表最后一个数据
                Node last = cache.removeLast();
                map.remove(last.key);
            }
            // 直接添加到头部
            cache.addFirst(x);
            map.put(key, x);
        }
    }
}
```

这里就能回答之前的问答题“为什么要在链表中同时存储 key 和 val，而不是只存储 val”，注意这段代码：

```
if (cap == cache.size()) {
    // 删除链表最后一个数据
    Node last = cache.removeLast();
    map.remove(last.key);
}
```

当缓存容量已满，我们不仅仅要删除最后一个 Node 节点，还要把 map 中映射到该节点的 key 同时删除，而这个 key 只能由 Node 得到。如果 Node 结构中只存储 val，那么我们就无法得知 key 是什么，就无法删除 map 中的键，造成错误。

至此，你应该已经掌握 LRU 算法的思想和实现了，很容易犯错的一点是：处理链表节点的同时不要忘了更新哈希表中对节点的映射。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

二叉搜索树操作集锦

通过之前的文章框架思维，二叉树的遍历框架应该已经印到你的脑子里了，这篇文章就来实操一下，看看框架思维是怎么灵活运用，秒杀一切二叉树问题的。

二叉树算法的设计的总路线：明确一个节点要做的事情，然后剩下的事抛给框架。

```
void traverse(TreeNode root) {
    // root 需要做什么？在这做。
    // 其他的不用 root 操心，抛给框架
    traverse(root.left);
    traverse(root.right);
}
```

举两个简单的例子体会一下这个思路，热热身。

1. 如何把二叉树所有的节点中的值加一？

```
void plusOne(TreeNode root) {
    if (root == null) return;
    root.val += 1;

    plusOne(root.left);
    plusOne(root.right);
}
```

2. 如何判断两棵二叉树是否完全相同？

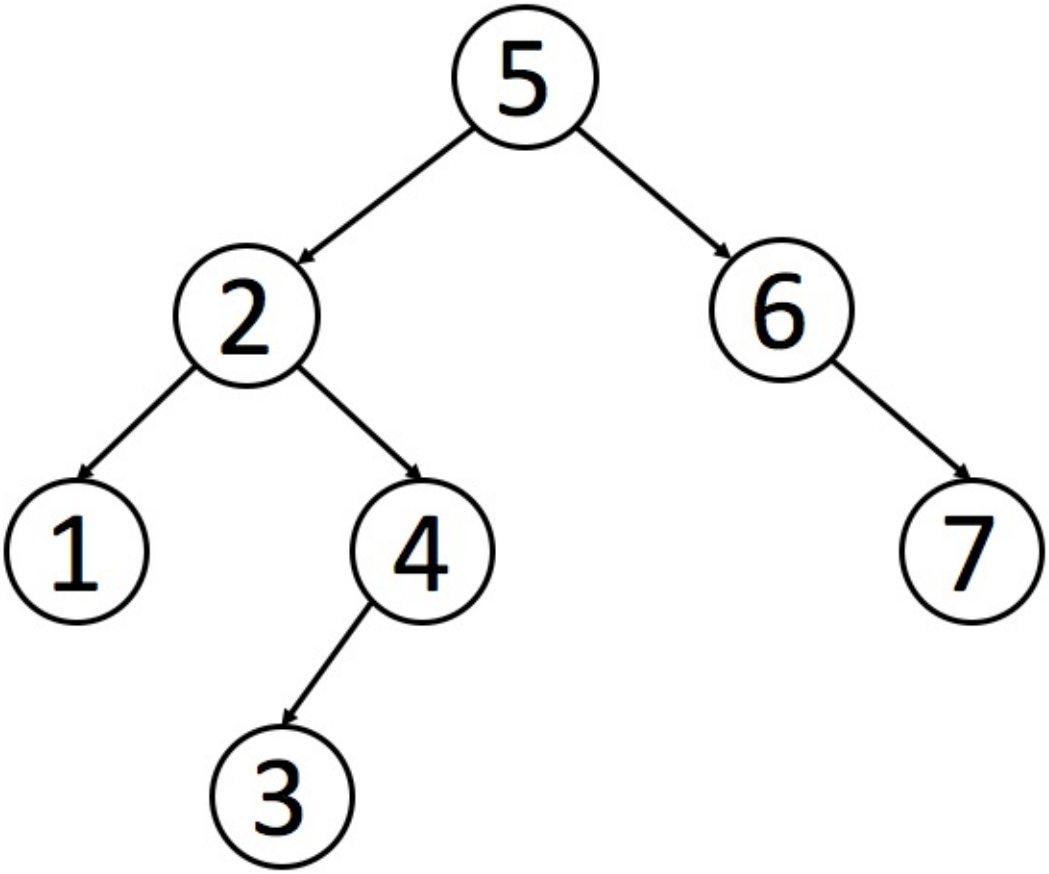
```
boolean isSameTree(TreeNode root1, TreeNode root2) {
    // 都为空的话，显然相同
    if (root1 == null && root2 == null) return true;
    // 一个为空，一个非空，显然不同
    if (root1 == null || root2 == null) return false;
```

```
        // 两个都非空，但 val 不一样也不行
        if (root1.val != root2.val) return false;

        // root1 和 root2 该比的都比完了
        return isSameTree(root1.left, root2.left)
            && isSameTree(root1.right, root2.right);
    }
```

借助框架，上面这两个例子不难理解吧？如果可以理解，那么所有二叉树算法你都能解决。

二叉搜索树（Binary Search Tree，简称 BST）是一种很常用的的二叉树。它的定义是：一个二叉树中，任意节点的值要大于等于左子树所有节点的值，且要小于等于右边子树的所有节点的值。

如下就是一个符合定义的 BST：



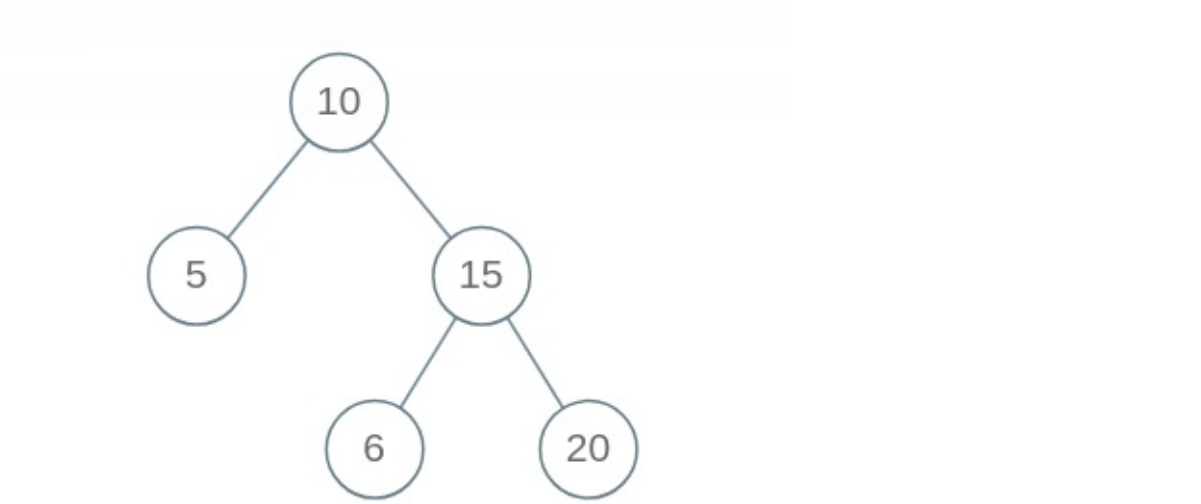
下面实现 BST 的基础操作：判断 BST 的合法性、增、删、查。其中“删”和“判断合法性”略微复杂。

零、判断 BST 的合法性

这里是有坑的哦，我们按照刚才的思路，每个节点自己要做的事不就是比较自己和左右孩子吗？看起来应该这样写代码：

```
boolean isValidBST(TreeNode root) {
    if (root == null) return true;
    if (root.left != null && root.val <= root.left.val) return false;
    if (root.right != null && root.val >= root.right.val) return fals
e;

    return isValidBST(root.left)
        && isValidBST(root.right);
}
```

但是这个算法出现了错误，BST 的每个节点应该要小于右边子树的所有节点，下面这个二叉树显然不是 BST，但是我们的算法会把它判定为 BST。



出现错误，不要慌张，框架没有错，一定是某个细节问题没注意到。我们重新看一下 BST 的定义，root 需要做的不只是和左右子节点比较，而是要整个左子树和右子树所有节点比较。怎么办，鞭长莫及啊！

这种情况，我们可以使用辅助函数，增加函数参数列表，在参数中携带额外信息，请看正确的代码：

```
boolean isValidBST(TreeNode root) {
    return isValidBST(root, null, null);
}

boolean isValidBST(TreeNode root, TreeNode min, TreeNode max) {
    if (root == null) return true;
    if (min != null && root.val <= min.val) return false;
    if (max != null && root.val >= max.val) return false;
    return isValidBST(root.left, min, root)
        && isValidBST(root.right, root, max);
}
```

一、在 BST 中查找一个数是否存在

根据我们的指导思想，可以这样写代码：

```
boolean isInBST(TreeNode root, int target) {
    if (root == null) return false;
    if (root.val == target) return true;

    return isInBST(root.left, target)
        || isInBST(root.right, target);
}
```

这样写完全正确，充分证明了你的框架性思维已经养成。现在你可以考虑一点细节问题了：如何充分利用信息，把 BST 这个“左小右大”的特性用上？

很简单，其实不需要递归地搜索两边，类似二分查找思想，根据 target 和 root.val 的大小比较，就能排除一边。我们把上面的思路稍稍改动：

```
boolean isInBST(TreeNode root, int target) {
    if (root == null) return false;
    if (root.val == target)
        return true;
    if (root.val < target)
        return isInBST(root.right, target);
```

```
    if (root.val > target)
        return isInBST(root.left, target);
    // root 该做的事做完了，顺带把框架也完成了，妙
}
```

于是，我们对原始框架进行改造，抽象出一套**针对 BST 的遍历框架**：

```
void BST(TreeNode root, int target) {
    if (root.val == target)
        // 找到目标，做点什么
    if (root.val < target)
        BST(root.right, target);
    if (root.val > target)
        BST(root.left, target);
}
```

二、在 BST 中插入一个数

对数据结构的操作无非遍历 + 访问，遍历就是“找”，访问就是“改”。具体到这个问题，插入一个数，就是先找到插入位置，然后进行插入操作。

上一个问题，我们总结了 BST 中的遍历框架，就是“找”的问题。直接套框架，加上“改”的操作即可。一旦涉及“改”，函数就要返回 TreeNode 类型，并且对递归调用的返回值进行接收。

```
TreeNode insertIntoBST(TreeNode root, int val) {
    // 找到空位置插入新节点
    if (root == null) return new TreeNode(val);
    // if (root.val == val)
    //     BST 中一般不会插入已存在元素
    if (root.val < val)
        root.right = insertIntoBST(root.right, val);
    if (root.val > val)
        root.left = insertIntoBST(root.left, val);
    return root;
}
```

三、在 BST 中删除一个数

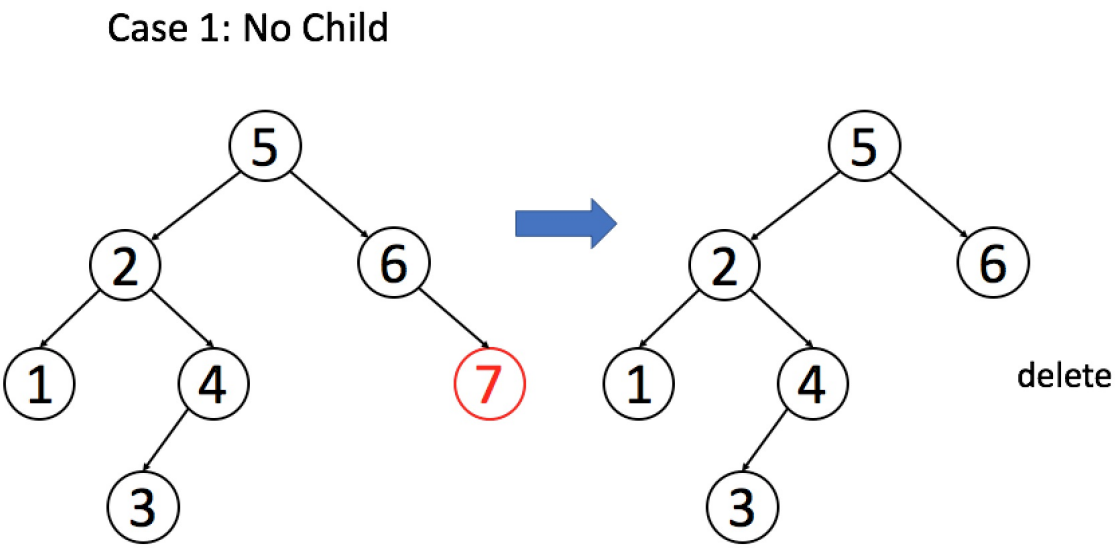
这个问题稍微复杂，不过你有框架指导，难不住你。跟插入操作类似，先“找”再“改”，先把框架写出来再说：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root.val == key) {
        // 找到啦，进行删除
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        root.right = deleteNode(root.right, key);
    }
    return root;
}
```

找到目标节点了，比方说是节点 A，如何删除这个节点，这是难点。因为删除节点的同时不能破坏 BST 的性质。有三种情况，用图片来说明。

情况 1：A 恰好是末端节点，两个子节点都为空，那么它可以当场去世了。

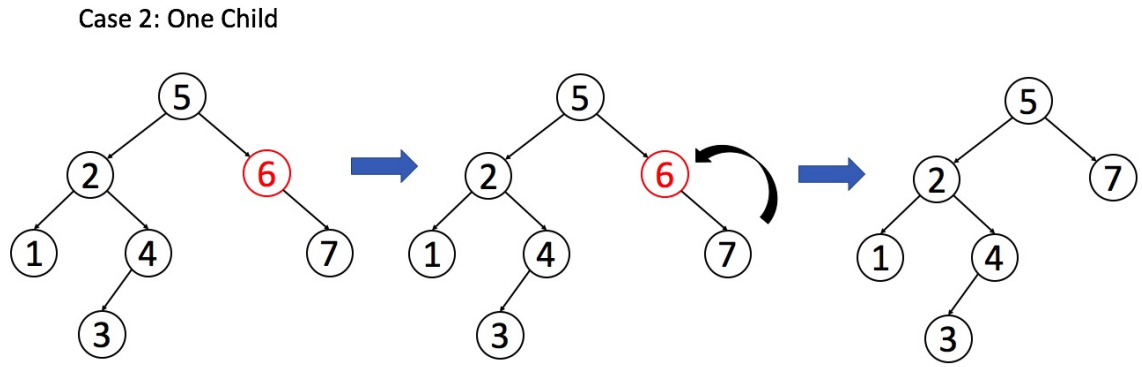
图片来自 LeetCode




```
if (root.left == null && root.right == null)
    return null;
```

情况 2：A 只有一个非空子节点，那么它要让这个孩子接替自己的位置。

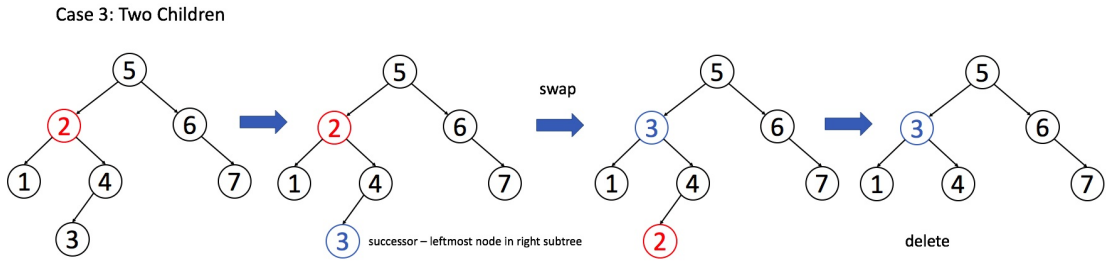
图片来自 LeetCode




```
// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;
```

情况 3：A 有两个子节点，麻烦了，为了不破坏 BST 的性质，A 必须找到左子树中最大的那个节点，或者右子树中最小的那个节点来接替自己。我们以第二种方式讲解。

图片来自 LeetCode




```
if (root.left != null && root.right != null) {
    // 找到右子树的最小节点
    TreeNode minNode = getMin(root.right);
    // 把 root 改成 minNode
    root.val = minNode.val;
    // 转而去删除 minNode
    root.right = deleteNode(root.right, minNode.val);
}
```

三种情况分析完毕，填入框架，简化一下代码：

```
TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) return null;
    if (root.val == key) {
        // 这两个 if 把情况 1 和 2 都正确处理了
        if (root.left == null) return root.right;
        if (root.right == null) return root.left;
        // 处理情况 3
        TreeNode minNode = getMin(root.right);
        root.val = minNode.val;
        root.right = deleteNode(root.right, minNode.val);
    } else if (root.val > key) {
        root.left = deleteNode(root.left, key);
    } else if (root.val < key) {
        root.right = deleteNode(root.right, key);
    }
    return root;
}

TreeNode getMin(TreeNode node) {
    // BST 最左边的就是最小的
    while (node.left != null) node = node.left;
    return node;
}
```

删除操作就完成了。注意一下，这个删除操作并不完美，因为我们一般不会通过 root.val = minNode.val 修改节点内部的值来交换节点，而是通过一系列略微复杂的链表操作交换 root 和 minNode 两个节点。因为具体应用中，val 域可能会很大，修改起来很耗时，而链表操作无非改一改指针，而不会去碰内部数据。

但这里忽略这个细节，旨在突出 BST 基本操作的共性，以及借助框架逐层细化问题的思维方式。

四、最后总结

通过这篇文章，你学会了如下几个技巧：

1. 二叉树算法设计的总路线：把当前节点要做的事做好，其他的交给递归框架，不用当前节点操心。

2. 如果当前节点会对下面的子节点有整体影响，可以通过辅助函数增长参数列表，借助参数传递信息。

3. 在二叉树框架之上，扩展出一套 BST 遍历框架：

```
void BST(TreeNode root, int target) {
 if (root.val == target)
     // 找到目标，做点什么
 if (root.val < target)
     BST(root.right, target);
 if (root.val > target)
     BST(root.left, target);
}
```

4. 掌握了 BST 的基本操作。

如何使用单调栈解题

栈（stack）是很简单的一种数据结构，先进后出的逻辑顺序，符合某些问题的特点，比如说函数调用栈。

单调栈实际上就是栈，只是利用了一些巧妙的逻辑，使得每次新元素入栈后，栈内的元素都保持有序（单调递增或单调递减）。

听起来有点像堆（heap）？不是的，单调栈用途不太广泛，只处理一种典型的问题，叫做 Next Greater Element。本文用讲解单调队列的算法模版解决这类问题，并且探讨处理「循环数组」的策略。

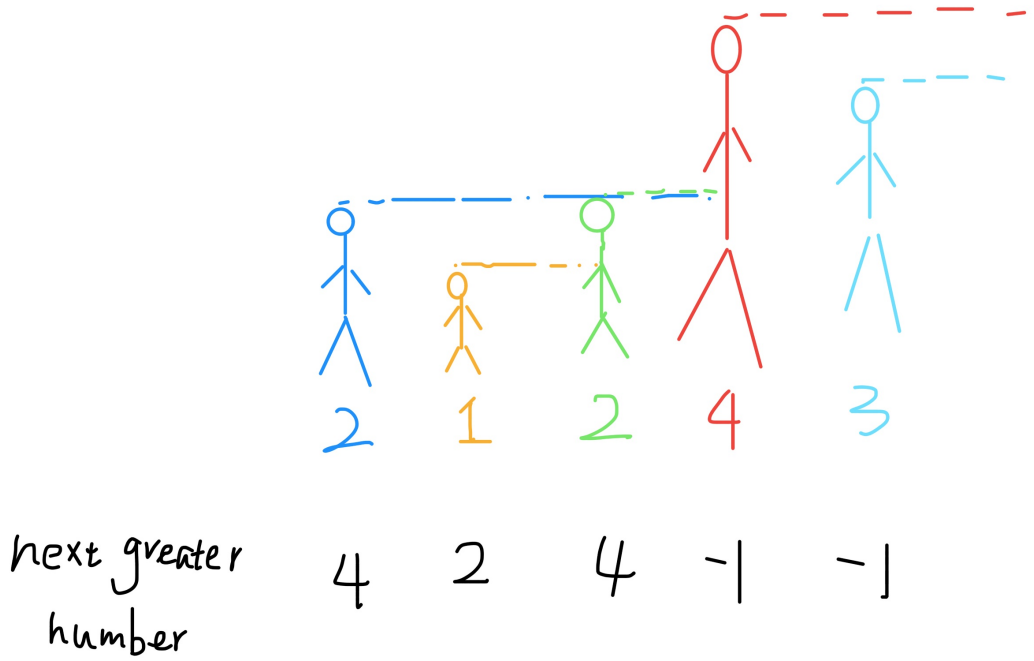
首先，讲解 Next Greater Number 的原始问题：给你一个数组，返回一个等长的数组，对应索引存储着下一个更大元素，如果没有更大的元素，就存 -1。不好用语言解释清楚，直接上一个例子：

给你一个数组 [2,1,2,4,3]，你返回数组 [4,2,4,-1,-1]。

解释：第一个 2 后面比 2 大的数是 4; 1 后面比 1 大的数是 2；第二个 2 后面比 2 大的数是 4; 4 后面没有比 4 大的数，填 -1；3 后面没有比 3 大的数，填 -1。

这道题的暴力解法很好想到，就是对每个元素后面都进行扫描，找到第一个更大的元素就行了。但是暴力解法的时间复杂度是 O(n^2)。

这个问题可以这样抽象思考：把数组的元素想象成并列站立的人，元素大小想象成人的身高。这些人面对你站成一列，如何求元素「2」的 Next Greater Number 呢？很简单，如果能够看到元素「2」，那么他后面可见的第一个人就是「2」的 Next Greater Number，因为比「2」小的元素身高不够，都被「2」挡住了，第一个露出来的就是答案。



这个情景很好理解吧？带着这个抽象的情景，先来看下代码。

```
vector<int> nextGreaterElement(vector<int>& nums) {
    vector<int> ans(nums.size()); // 存放答案的数组
    stack<int> s;
    for (int i = nums.size() - 1; i >= 0; i--) { // 倒着往栈里放
        while (!s.empty() && s.top() <= nums[i]) { // 判定个子高矮
            s.pop(); // 矮个起开，反正也被挡着了。。。
        }
        ans[i] = s.empty() ? -1 : s.top(); // 这个元素身后的第一个高个
        s.push(nums[i]); // 进队，接受之后的身高判定吧！
    }
    return ans;
}
```

这就是单调队列解决问题的模板。for 循环要从后往前扫描元素，因为我们借助的是栈的结构，倒着入栈，其实是正着出栈。while 循环是把两个“高个”元素之间的元素排除，因为他们的存在没有意义，前面挡着个“更高”的元素，所以他们不可能被作为后续进来的元素的 Next Great Number 了。

这个算法的时间复杂度不是那么直观，如果你看到 for 循环嵌套 while 循环，可能认为这个算法的复杂度也是 O(n^2)，但是实际上这个算法的复杂度只有 O(n)。

分析它的时间复杂度，要从整体来看：总共有 n 个元素，每个元素都被 push 入栈了一次，而最多会被 pop 一次，没有任何冗余操作。所以总的计算规模是和元素规模 n 成正比的，也就是 O(n) 的复杂度。

现在，你已经掌握了单调栈的使用技巧，来一个简单的变形来加深一下理解。

给你一个数组 T = [73, 74, 75, 71, 69, 72, 76, 73]，这个数组存放的是近几天的天气气温（这气温是铁板烧？不是的，这里用的华氏度）。你返回一个数组，计算：对于每一天，你还要至少等多少天才能等到一个更暖和的气温；如果等不到那一天，填 0 。

举例：给你 T = [73, 74, 75, 71, 69, 72, 76, 73]，你返回 [1, 1, 4, 2, 1, 1, 0, 0]。

解释：第一天 73 华氏度，第二天 74 华氏度，比 73 大，所以对于第一天，只要等一天就能等到一个更暖和的气温。后面的同理。

你已经对 Next Greater Number 类型问题有些敏感了，这个问题本质上也是找 Next Greater Number，只不过现在不是问你 Next Greater Number 是多少，而是问你当前距离 Next Greater Number 的距离而已。

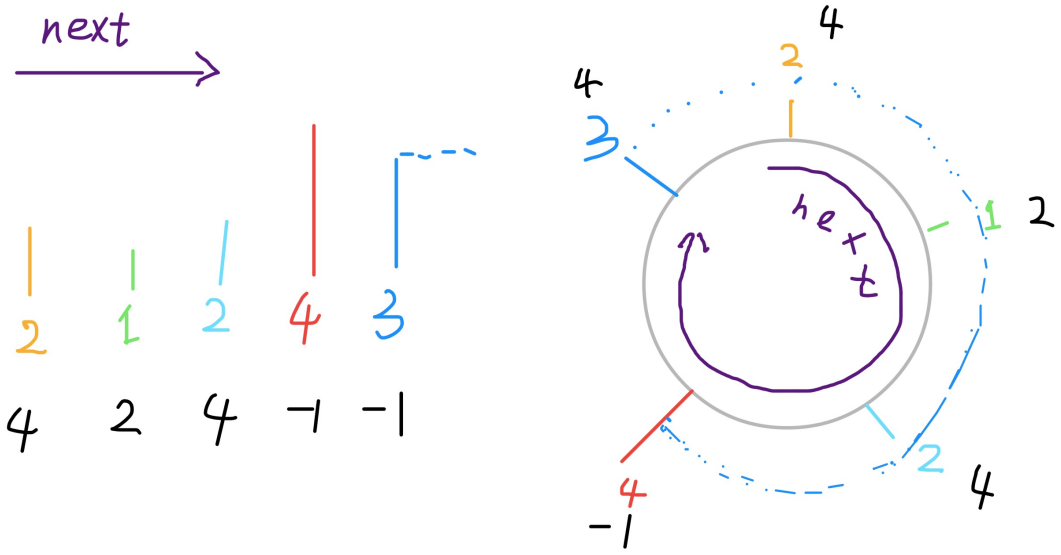
相同类型的问题，相同的思路，直接调用单调栈的算法模板，稍作改动就可以啦，直接上代码把。

```
vector<int> dailyTemperatures(vector<int>& T) {
    vector<int> ans(T.size());
    stack<int> s; // 这里放元素索引，而不是元素
    for (int i = T.size() - 1; i >= 0; i--) {
        while (!s.empty() && T[s.top()] <= T[i]) {
            s.pop();
        }
        ans[i] = s.empty() ? 0 : (s.top() - i); // 得到索引间距
        s.push(i); // 加入索引，而不是元素
    }
    return ans;
}
```

单调栈讲解完毕。下面开始另一个重点：如何处理「循环数组」。

同样是 Next Greater Number，现在假设给你的数组是个环形的，如何处理？

给你一个数组 [2,1,2,4,3]，你返回数组 [4,2,4,-1,4]。拥有了环形属性，最后一个元素 3 绕了一圈后找到了比自己大的元素 4 。

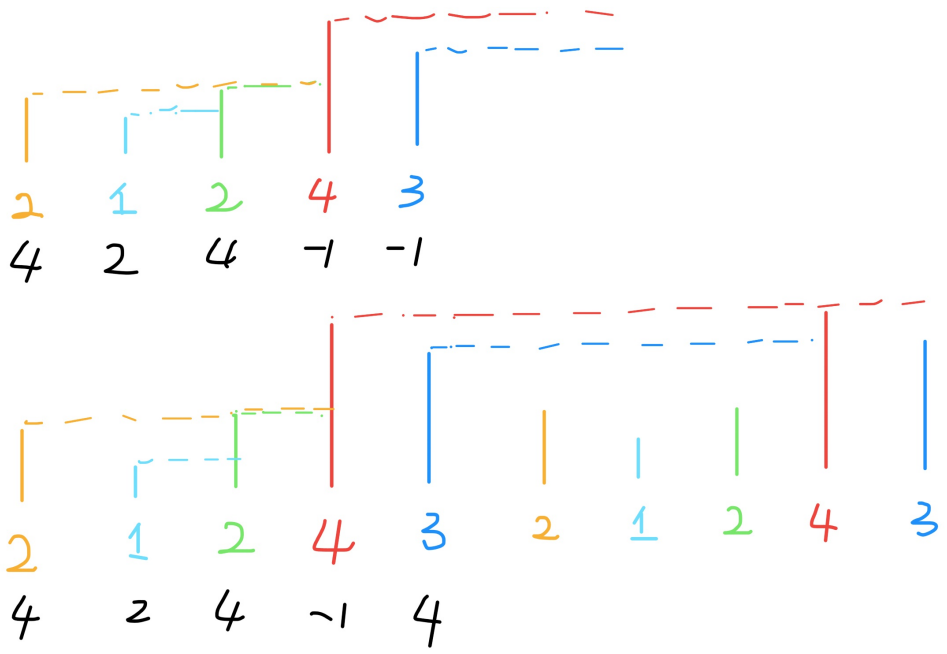


首先，计算机的内存都是线性的，没有真正意义上的环形数组，但是我们可以模拟出环形数组的效果，一般是通过 % 运算符求模（余数），获得环形特效：

```
int[] arr = {1,2,3,4,5};
int n = arr.length, index = 0;
while (true) {
    print(arr[index % n]);
    index++;
}
```

回到 Next Greater Number 的问题，增加了环形属性后，问题的难点在于：这个 Next 的意义不仅仅是当前元素的右边了，有可能出现在当前元素的左边（如上例）。

明确问题，问题就已经解决了一半了。我们可以考虑这样的思路：将原始数组“翻倍”，就是在后面再接一个原始数组，这样的话，按照之前“比身高”的流程，每个元素不仅可以比较自己右边的元素，而且也可以和左边的元素比较了。



怎么实现呢？你当然可以把这个双倍长度的数组构造出来，然后套用算法模板。但是，我们可以不用构造新数组，而是利用循环数组的技巧来模拟。直接看代码吧：

```
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> res(n); // 存放结果
    stack<int> s;
    // 假装这个数组长度翻倍了
    for (int i = 2 * n - 1; i >= 0; i--) {
        while (!s.empty() && s.top() <= nums[i % n])
            s.pop();
        res[i % n] = s.empty() ? -1 : s.top();
        s.push(nums[i % n]);
    }
    return res;
}
```

至此，你已经掌握了单调栈的设计方法及代码模板，学会了解决 Next Greater Number，并能够处理循环数组了。

你的在看，是对我的鼓励。关注公众号：labuladong

特殊数据结构：单调队列

前文讲了一种特殊的数据结构「单调栈」monotonic stack，解决了一类问题「Next Greater Number」，本文写一个类似的数据结构「单调队列」。

也许这种数据结构的名字你没听过，其实没啥难的，就是一个「队列」，只是使用了一点巧妙的方法，使得队列中的元素单调递增（或递减）。这个数据结构有什么用？可以解决滑动窗口的一系列问题。

看一道 LeetCode 题目，难度 hard：

> 给定一个数组 *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 *k* 内的数字。滑动窗口每次只向右移动一位。
>
> 返回滑动窗口最大值。
>
> **示例:**
>
> **输入:** *nums* = [1,3,-1,-3,5,3,6,7], 和 *k* = 3
> **输出:** [3,3,5,5,6,7]
> **解释:**
>
> ```
>   滑动窗口的位置                最大值
> ---------------               -----
> [1  3  -1] -3  5  3  6  7       3
>  1 [3  -1  -3] 5  3  6  7       3
>  1  3 [-1  -3  5] 3  6  7       5
>  1  3  -1 [-3  5  3] 6  7       5
>  1  3  -1  -3 [5  3  6] 7       6
>  1  3  -1  -3  5 [3  6  7]      7
> ```

一、搭建解题框架

这道题不复杂，难点在于如何在 O(1) 时间算出每个「窗口」中的最大值，使得整个算法在线性时间完成。在之前我们探讨过类似的场景，得到一个结论：

在一堆数字中，已知最值，如果给这堆数添加一个数，那么比较一下就可以很快算出最值；但如果减少一个数，就不一定能很快得到最值了，而要遍历所有数重新找最值。

回到这道题的场景，每个窗口前进的时候，要添加一个数同时减少一个数，所以想在 O(1) 的时间得出新的最值，就需要「单调队列」这种特殊的数据结构来辅助了。

一个普通的队列一定有这两个操作：

```
class Queue {
    void push(int n);
    // 或 enqueue，在队尾加入元素 n
    void pop();
    // 或 dequeue，删除队头元素
}
```

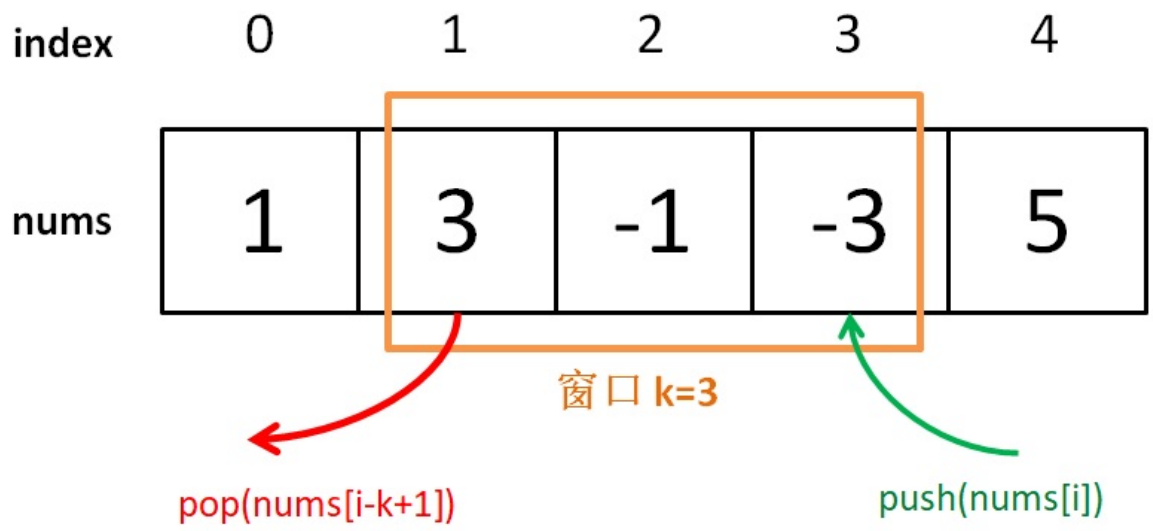
一个「单调队列」的操作也差不多：

```
class MonotonicQueue {
    // 在队尾添加元素 n
    void push(int n);
    // 返回当前队列中的最大值
    int max();
    // 队头元素如果是 n，删除它
    void pop(int n);
}
```

当然，这几个 API 的实现方法肯定跟一般的 Queue 不一样，不过我们暂且不管，而且认为这几个操作的时间复杂度都是 O(1)，先把这道「滑动窗口」问题的解答框架搭出来：

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
```

```
    MonotonicQueue window;
    vector<int> res;
    for (int i = 0; i < nums.size(); i++) {
        if (i < k - 1) { //先把窗口的前 k - 1 填满
            window.push(nums[i]);
        } else { // 窗口开始向前滑动
            window.push(nums[i]);
            res.push_back(window.max());
            window.pop(nums[i - k + 1]);
            // nums[i - k + 1] 就是窗口最后的元素
        }
    }
    return res;
}
```



这个思路很简单，能理解吧？下面我们开始重头戏，单调队列的实现。

二、实现单调队列数据结构

首先我们要认识另一种数据结构：deque，即双端队列。很简单：

```
class deque {
    // 在队头插入元素 n
    void push_front(int n);
    // 在队尾插入元素 n
```

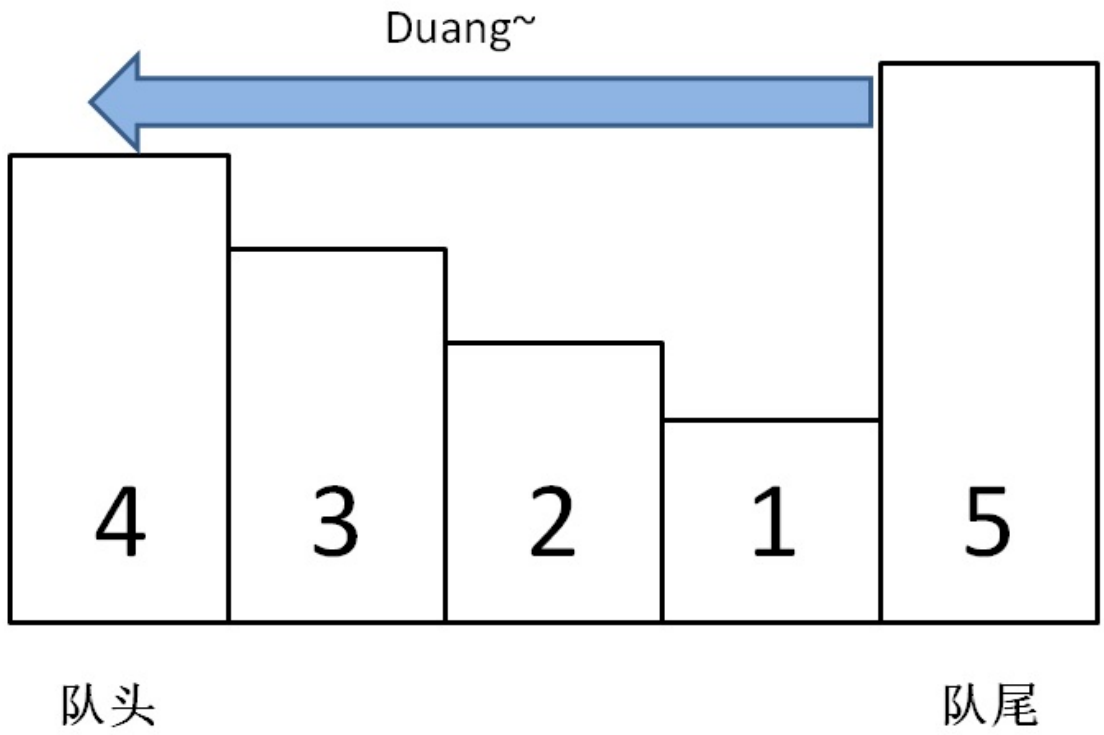
```
    void push_back(int n);
    // 在队头删除元素
    void pop_front();
    // 在队尾删除元素
    void pop_back();
    // 返回队头元素
    int front();
    // 返回队尾元素
    int back();
}
```

而且，这些操作的复杂度都是 O(1)。这其实不是啥稀奇的数据结构，用链表作为底层结构的话，很容易实现这些功能。

「单调队列」的核心思路和「单调栈」类似。单调队列的 push 方法依然在队尾添加元素，但是要把前面比新元素小的元素都删掉：

```
class MonotonicQueue {
private:
    deque<int> data;
public:
    void push(int n) {
        while (!data.empty() && data.back() < n)
            data.pop_back();
        data.push_back(n);
    }
};
```

你可以想象，加入数字的大小代表人的体重，把前面体重不足的都压扁了，直到遇到更大的量级才停住。



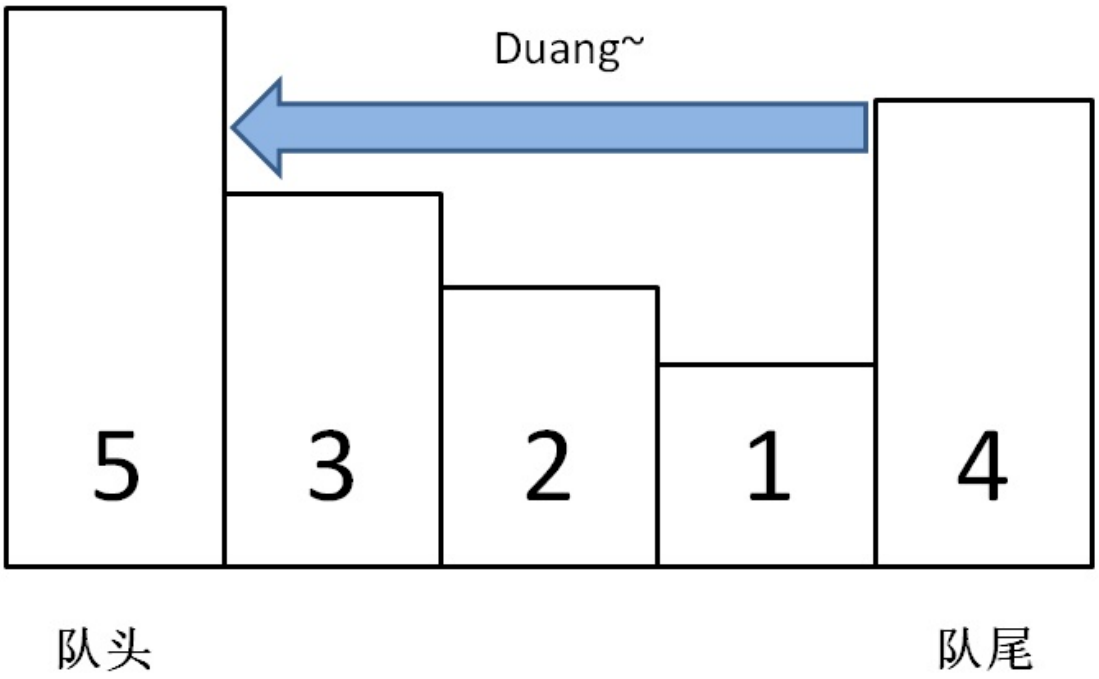
如果每个元素被加入时都这样操作，最终单调队列中的元素大小就会保持一个单调递减的顺序，因此我们的 max() API 可以可以这样写：

```
int max() {
    return data.front();
}
```

pop() API 在队头删除元素 n，也很好写：

```
void pop(int n) {
    if (!data.empty() && data.front() == n)
        data.pop_front();
}
```

之所以要判断 `data.front() == n`，是因为我们想删除的队头元素 n 可能已经被「压扁」了，这时候就不用删除了：



至此，单调队列设计完毕，看下完整的解题代码：

```
class MonotonicQueue {
private:
    deque<int> data;
public:
    void push(int n) {
        while (!data.empty() && data.back() < n)
            data.pop_back();
        data.push_back(n);
    }

    int max() { return data.front(); }

    void pop(int n) {
        if (!data.empty() && data.front() == n)
            data.pop_front();
    }
};

vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    MonotonicQueue window;
    vector<int> res;
    for (int i = 0; i < nums.size(); i++) {
```

```
            if (i < k - 1) { //先填满窗口的前 k - 1
                window.push(nums[i]);
            } else { // 窗口向前滑动
                window.push(nums[i]);
                res.push_back(window.max());
                window.pop(nums[i - k + 1]);
            }
        }
        return res;
    }
```

三、算法复杂度分析

读者可能疑惑，push 操作中含有 while 循环，时间复杂度不是 O(1) 呀，那么本算法的时间复杂度应该不是线性时间吧？

单独看 push 操作的复杂度确实不是 O(1)，但是算法整体的复杂度依然是 O(N) 线性时间。要这样想，nums 中的每个元素最多被 push_back 和 pop_back 一次，没有任何多余操作，所以整体的复杂度还是 O(N)。

空间复杂度就很简单了，就是窗口的大小 O(k)。

四、最后总结

有的读者可能觉得「单调队列」和「优先级队列」比较像，实际上差别很大的。

单调队列在添加元素的时候靠删除元素保持队列的单调性，相当于抽取出某个函数中单调递增（或递减）的部分；而优先级队列（二叉堆）相当于自动排序，差别大了去了。

赶紧去拿下 LeetCode 第 239 道题吧～

设计Twitter

「design Twitter」是 LeetCode 上第 335 道题目，不仅题目本身很有意思，而且把合并多个有序链表的算法和面向对象设计（OO design）结合起来了，很有实际意义，本文就带大家来看看这道题。

至于 Twitter 的什么功能跟算法有关系，等我们描述一下题目要求就知道了。

一、题目及应用场景简介

Twitter 和微博功能差不多，我们主要要实现这样几个 API：

```
class Twitter {

    /** user 发表一条 tweet 动态 */
    public void postTweet(int userId, int tweetId) {}

    /** 返回该 user 关注的人（包括他自己）最近的动态 id，
    最多 10 条，而且这些动态必须按从新到旧的时间线顺序排列。*/
    public List<Integer> getNewsFeed(int userId) {}

    /** follower 关注 followee，如果 Id 不存在则新建 */
    public void follow(int followerId, int followeeId) {}

    /** follower 取关 followee，如果 Id 不存在则什么都不做 */
    public void unfollow(int followerId, int followeeId) {}
}
```

举个具体的例子，方便大家理解 API 的具体用法：

```
Twitter twitter = new Twitter();

twitter.postTweet(1, 5);
// 用户 1 发送了一条新推文 5
```

```
twitter.getNewsFeed(1);
// return [5]，因为自己是关注自己的

twitter.follow(1, 2);
// 用户 1 关注了用户 2

twitter.postTweet(2, 6);
// 用户2发送了一个新推文 (id = 6)

twitter.getNewsFeed(1);
// return [6, 5]
// 解释：用户 1 关注了自己和用户 2，所以返回他们的最近推文
// 而且 6 必须在 5 之前，因为 6 是最近发送的

twitter.unfollow(1, 2);
// 用户 1 取消关注了用户 2

twitter.getNewsFeed(1);
// return [5]
```

这个场景在我们的现实生活中非常常见。拿朋友圈举例，比如我刚加到女神的微信，然后我去刷新一下我的朋友圈动态，那么女神的动态就会出现在我的动态列表，而且会和其他动态按时间排好序。只不过 Twitter 是单向关注，微信好友相当于双向关注。除非，被屏蔽...

这几个 API 中大部分都很好实现，最核心的功能难点应该是 `getNewsFeed`，因为返回的结果必须在时间上有序，但问题是用户的关注是动态变化的，怎么办？

这里就涉及到算法了：如果我们把每个用户各自的推文存储在链表里，每个链表节点存储文章 id 和一个时间戳 time（记录发帖时间以便比较），而且这个链表是按 time 有序的，那么如果某个用户关注了 k 个用户，我们就可以用合并 k 个有序链表的算法合并出有序的推文列表，正确地 `getNewsFeed` 了！

具体的算法等会讲解。不过，就算我们掌握了算法，应该如何编程表示用户 user 和推文动态 tweet 才能把算法流畅地用出来呢？**这就涉及简单的面向对象设计了**，下面我们来由浅入深，一步一步进行设计。

二、面向对象设计

根据刚才的分析，我们需要一个 User 类，储存 user 信息，还需要一个 Tweet 类，储存推文信息，并且要作为链表的节点。所以我们先搭建一下整体的框架：

```
class Twitter {
    private static int timestamp = 0;
    private static class Tweet {}
    private static class User {}

    /* 还有那几个 API 方法 */
    public void postTweet(int userId, int tweetId) {}
    public List<Integer> getNewsFeed(int userId) {}
    public void follow(int followerId, int followeeId) {}
    public void unfollow(int followerId, int followeeId) {}
}
```

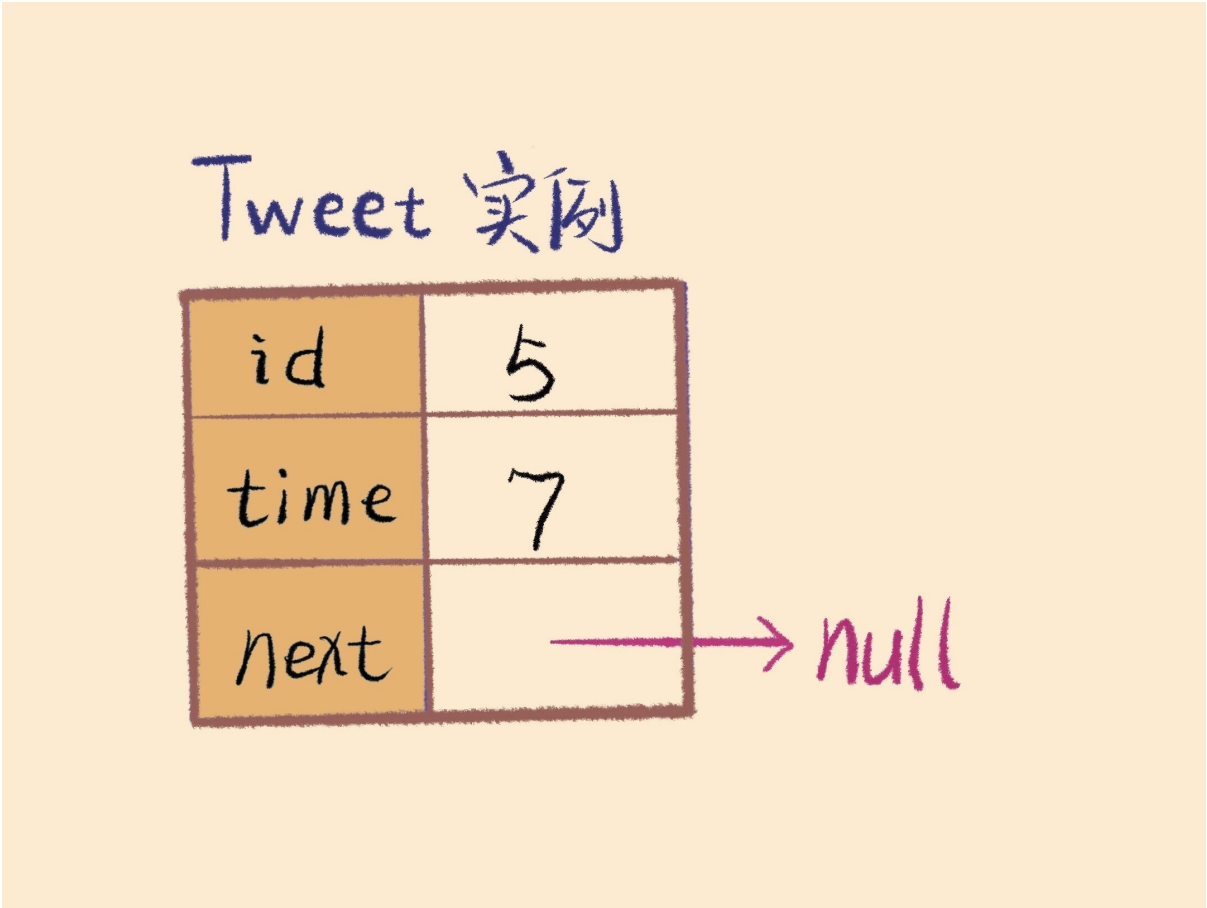
之所以要把 Tweet 和 User 类放到 Twitter 类里面，是因为 Tweet 类必须要用到一个全局时间戳 timestamp，而 User 类又需要用到 Tweet 类记录用户发送的推文，所以它们都作为内部类。不过为了清晰和简洁，下文会把每个内部类和 API 方法单独拿出来实现。

1、Tweet 类的实现

根据前面的分析，Tweet 类很容易实现：每个 Tweet 实例需要记录自己的 tweetId 和发表时间 time，而且作为链表节点，要有一个指向下一个节点的 next 指针。

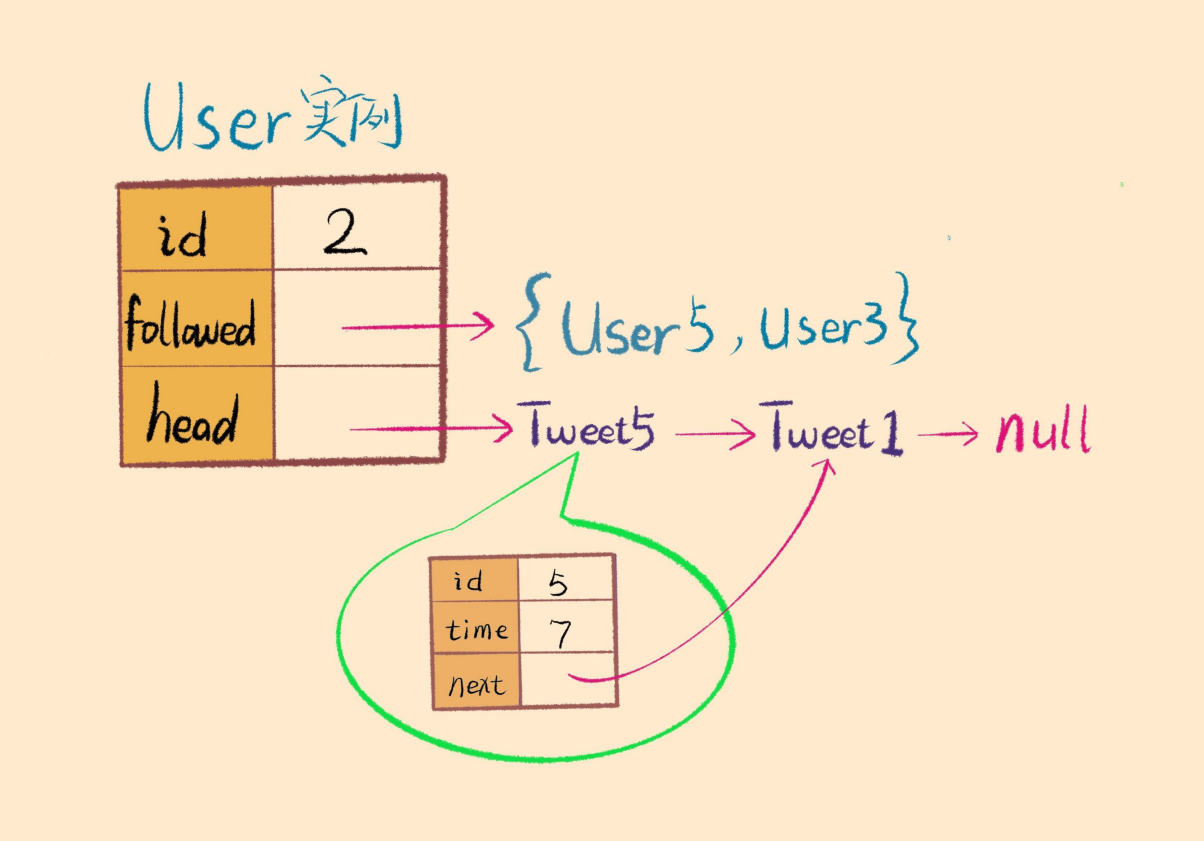
```
class Tweet {
    private int id;
    private int time;
    private Tweet next;
```

```
    // 需要传入推文内容（id）和发文时间
    public Tweet(int id, int time) {
        this.id = id;
        this.time = time;
        this.next = null;
    }
}
```



2、User 类的实现

我们根据实际场景想一想，一个用户需要存储的信息有 userId，关注列表，以及该用户发过的推文列表。其中关注列表应该用集合（Hash Set）这种数据结构来存，因为不能重复，而且需要快速查找；推文列表应该由链表这种数据结构储存，以便于进行有序合并的操作。画个图理解一下：



除此之外，根据面向对象的设计原则，「关注」「取关」和「发文」应该是 User 的行为，况且关注列表和推文列表也存储在 User 类中，所以我们也应该给 User 添加 follow，unfollow 和 post 这几个方法：

```
// static int timestamp = 0
class User {
    private int id;
    public Set<Integer> followed;
    // 用户发表的推文链表头结点
    public Tweet head;

    public User(int userId) {
        followed = new HashSet<>();
        this.id = userId;
        this.head = null;
        // 关注一下自己
        follow(id);
    }

    public void follow(int userId) {
        followed.add(userId);
    }
```

```
    public void unfollow(int userId) {
        // 不可以取关自己
        if (userId != this.id)
            followed.remove(userId);
    }

    public void post(int tweetId) {
        Tweet twt = new Tweet(tweetId, timestamp);
        timestamp++;
        // 将新建的推文插入链表头
        // 越靠前的推文 time 值越大
        twt.next = head;
        head = twt;
    }
}
```

3、几个 API 方法的实现

```
class Twitter {
    private static int timestamp = 0;
    private static class Tweet {...}
    private static class User {...}

    // 我们需要一个映射将 userId 和 User 对象对应起来
    private HashMap<Integer, User> userMap = new HashMap<>();

    /** user 发表一条 tweet 动态 */
    public void postTweet(int userId, int tweetId) {
        // 若 userId 不存在，则新建
        if (!userMap.containsKey(userId))
            userMap.put(userId, new User(userId));
        User u = userMap.get(userId);
        u.post(tweetId);
    }

    /** follower 关注 followee */
    public void follow(int followerId, int followeeId) {
        // 若 follower 不存在，则新建
        if(!userMap.containsKey(followerId)){
            User u = new User(followerId);
            userMap.put(followerId, u);
```

```
            }
            // 若 followee 不存在，则新建
            if(!userMap.containsKey(followeeId)){
                User u = new User(followeeId);
                userMap.put(followeeId, u);
            }
            userMap.get(followerId).follow(followeeId);
        }

        /** follower 取关 followee，如果 Id 不存在则什么都不做 */
        public void unfollow(int followerId, int followeeId) {
            if (userMap.containsKey(followerId)) {
                User flwer = userMap.get(followerId);
                flwer.unfollow(followeeId);
            }
        }

        /** 返回该 user 关注的人（包括他自己）最近的动态 id，
        最多 10 条，而且这些动态必须按从新到旧的时间线顺序排列。*/
        public List<Integer> getNewsFeed(int userId) {
            // 需要理解算法，见下文
        }
    }
```

三、算法设计

实现合并 k 个有序链表的算法需要用到优先级队列（Priority Queue），这种数据结构是「二叉堆」最重要的应用，你可以理解为它可以对插入的元素自动排序。乱序的元素插入其中就被放到了正确的位置，可以按照从小到大（或从大到小）有序地取出元素。

```
PriorityQueue pq
# 乱序插入
for i in {2,4,1,9,6}:
    pq.add(i)
while pq not empty:
    # 每次取出第一个（最小）元素
    print(pq.pop())

# 输出有序：1,2,4,6,9
```

借助这种牛逼的数据结构支持，我们就很容易实现这个核心功能了。注意我们把优先级队列设为按 time 属性**从大到小降序排列**，因为 time 越大意味着时间越近，应该排在前面：

```
public List<Integer> getNewsFeed(int userId) {
    List<Integer> res = new ArrayList<>();
    if (!userMap.containsKey(userId)) return res;
    // 关注列表的用户 Id
    Set<Integer> users = userMap.get(userId).followed;
    // 自动通过 time 属性从大到小排序，容量为 users 的大小
    PriorityQueue<Tweet> pq =
        new PriorityQueue<>(users.size(), (a, b)->(b.time - a.time));

    // 先将所有链表头节点插入优先级队列
    for (int id : users) {
        Tweet twt = userMap.get(id).head;
        if (twt == null) continue;
        pq.add(twt);
    }

    while (!pq.isEmpty()) {
        // 最多返回 10 条就够了
        if (res.size() == 10) break;
        // 弹出 time 值最大的（最近发表的）
        Tweet twt = pq.poll();
        res.add(twt.id);
        // 将下一篇 Tweet 插入进行排序
        if (twt.next != null)
            pq.add(twt.next);
    }
    return res;
}
```

这个过程是这样的，下面是我制作的一个 GIF 图描述合并链表的过程。假设有三个 Tweet 链表按 time 属性降序排列，我们把他们降序合并添加到 res 中。注意图中链表节点中的数字是 time 属性，不是 id 属性：

【pdf/mobi格式不支持GIF:设计Twitter/merge.gif】 请查看【关于本小抄及作者】章节的解决方案

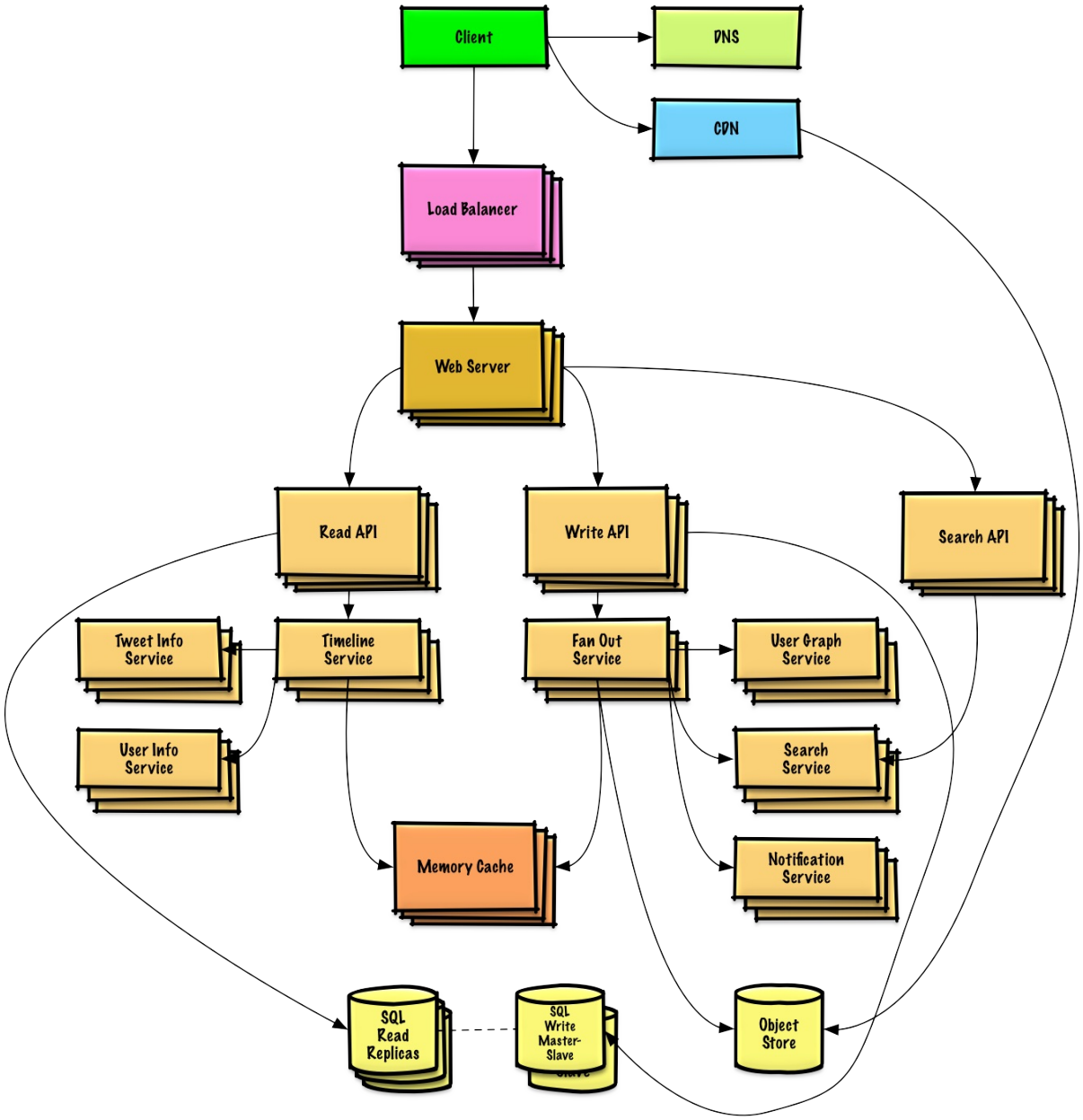
至此，这道一个极其简化的 Twitter 时间线功能就设计完毕了。

四、最后总结

本文运用简单的面向对象技巧和合并 k 个有序链表的算法设计了一套简化的时间线功能，这个功能其实广泛地运用在许多社交应用中。

我们先合理地设计出 User 和 Tweet 两个类，然后基于这个设计之上运用算法解决了最重要的一个功能。可见实际应用中的算法并不是孤立存在的，需要和其他知识混合运用，才能发挥实际价值。

当然，实际应用中的社交 App 数据量是巨大的，考虑到数据库的读写性能，我们的设计可能承受不住流量压力，还是有些太简化了。而且实际的应用都是一个极其庞大的工程，比如下图，是 Twitter 这样的社交网站大致的系统结构：



我们解决的问题应该只能算 Timeline Service 模块的一小部分，功能越多，系统的复杂性可能是指数级增长的。所以说合理的顶层设计十分重要，其作用是远超某一个算法的。

最后，Github 上有一个优秀的开源项目，专门收集了很多大型系统设计的案例和解析，而且有中文版本，上面这个图也出自该项目。对系统设计感兴趣的读者可以点击「阅读原文」查看。

PS：本文前两张图片和 GIF 是我第一次尝试用平板的绘图软件制作的，花了很多时间，尤其是 GIF 图，需要一帧一帧制作。如果本文内容对你有帮助，点个赞分个享，鼓励一下我呗！

递归反转链表的一部分

反转单链表的迭代实现不是一个困难的事情，但是递归实现就有点难度了，如果再加一点难度，让你仅仅反转单链表中的一部分，你是否能**够递归实现**呢？

本文就来由浅入深，step by step 地解决这个问题。如果你还不会递归地反转单链表也没关系，**本文会从递归反转整个单链表开始拓展**，只要你明白单链表的结构，相信你能够有所收获。

```
// 单链表节点的结构
public class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

什么叫反转单链表的一部分呢，就是给你一个索引区间，让你把单链表中这部分元素反转，其他部分不变：

反转从位置 *m* 到 *n* 的链表。请使用一趟扫描完成反转。

说明:
1 ≤ *m* ≤ *n* ≤ 链表长度。

示例:

输入: 1->2->3->4->5->NULL, *m* = 2, *n* = 4
输出: 1->4->3->2->5->NULL

注意这里的索引是从 1 开始的。迭代的思路大概是：先用一个 for 循环找到第 `m` 个位置，然后再用一个 for 循环将 `m` 和 `n` 之间的元素反转。但是我们的递归解法不用一个 for 循环，纯递归实现反转。

迭代实现思路看起来虽然简单，但是细节问题很多的，反而不容易写对。相反，递归实现就很简洁优美，下面就由浅入深，先从反转整个单链表说起。

一、递归反转整个链表

这个算法可能很多读者都听说过，这里详细介绍一下，先直接看实现代码：

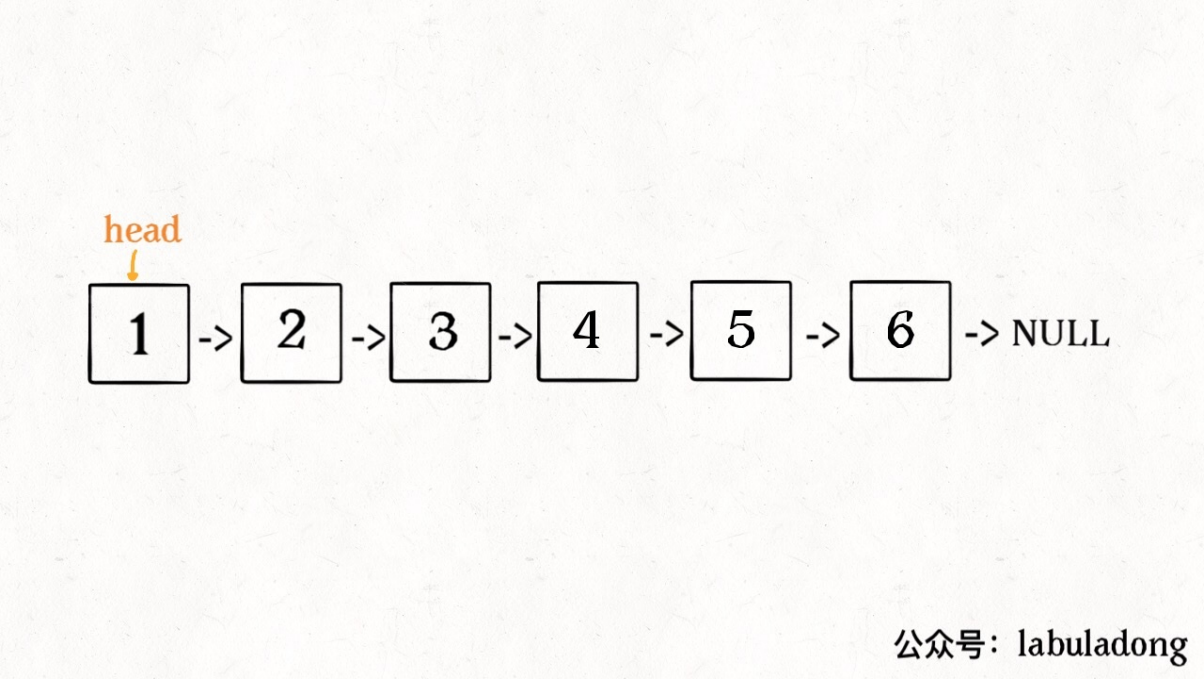
```
ListNode reverse(ListNode head) {
    if (head.next == null) return head;
    ListNode last = reverse(head.next);
    head.next.next = head;
    head.next = null;
    return last;
}
```

看起来是不是感觉不知所云，完全不能理解这样为什么能够反转链表？这就对了，这个算法常常拿来显示递归的巧妙和优美，我们下面来详细解释一下这段代码。

对于递归算法，最重要的就是明确递归函数的定义。具体来说，我们的 `reverse` 函数定义是这样的：

输入一个节点 `head`，将「以 `head` 为起点」的链表反转，并返回反转之后的头结点。

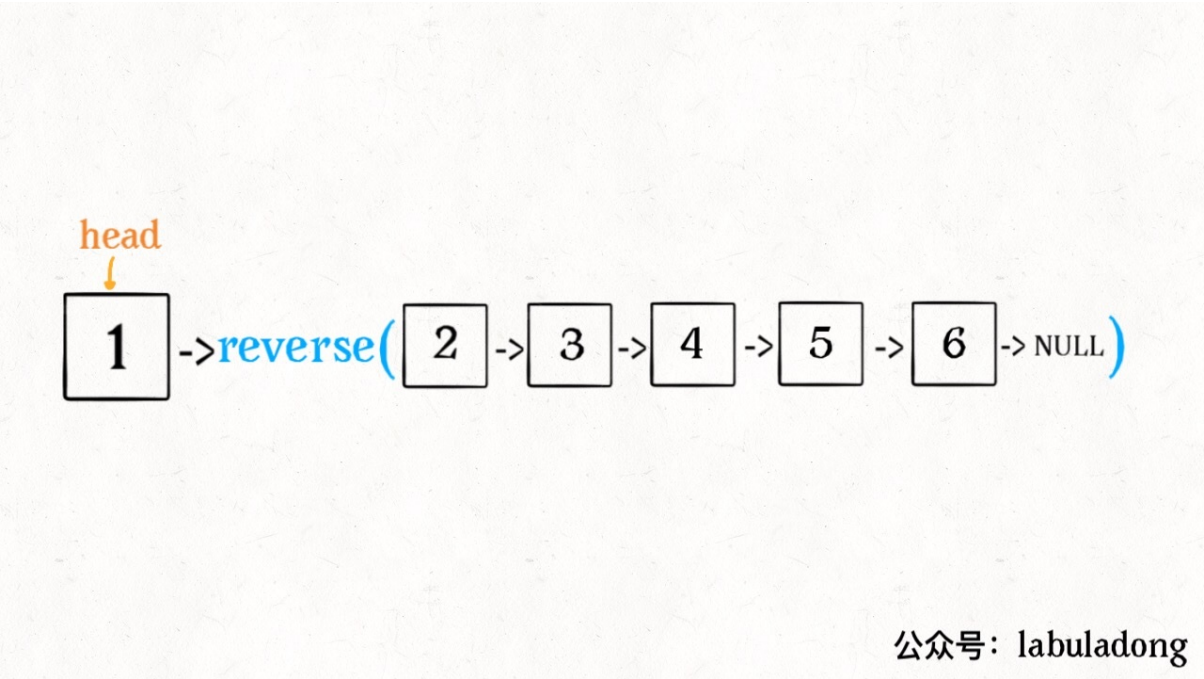
明白了函数的定义，在来看这个问题。比如说我们想反转这个链表：



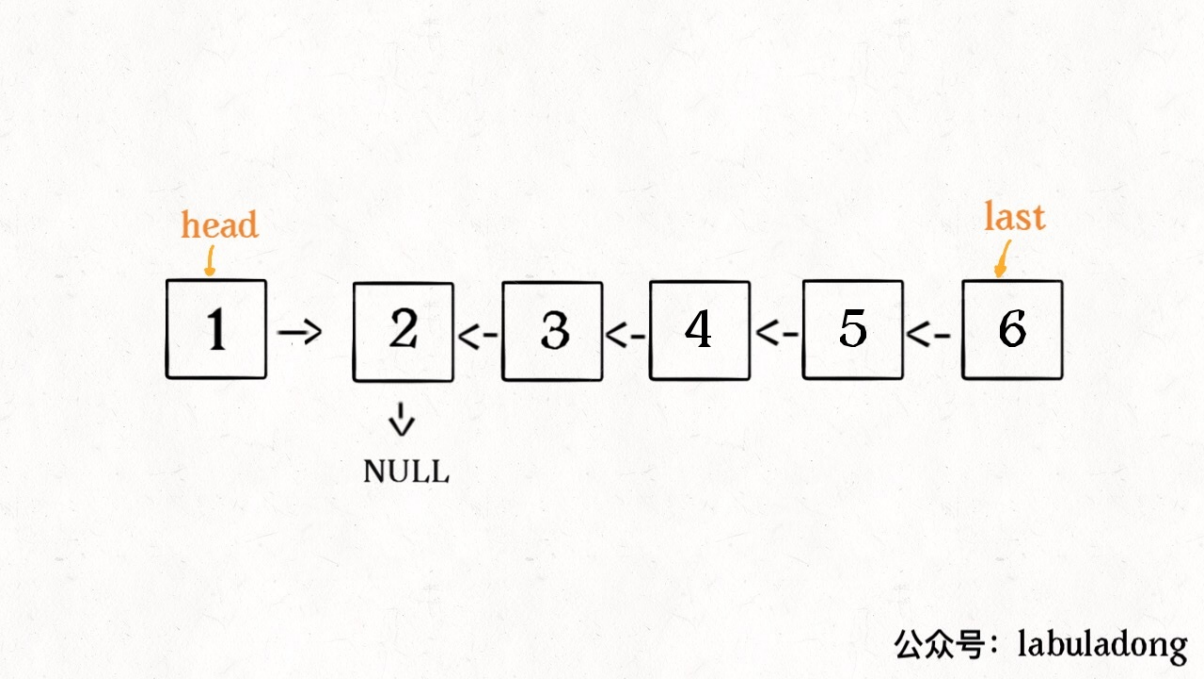
那么输入 `reverse(head)` 后，会在这里进行递归：

```
ListNode last = reverse(head.next);
```

不要跳进递归（你的脑袋能压几个栈呀？），而是要根据刚才的函数定义，来弄清楚这段代码会产生什么结果：



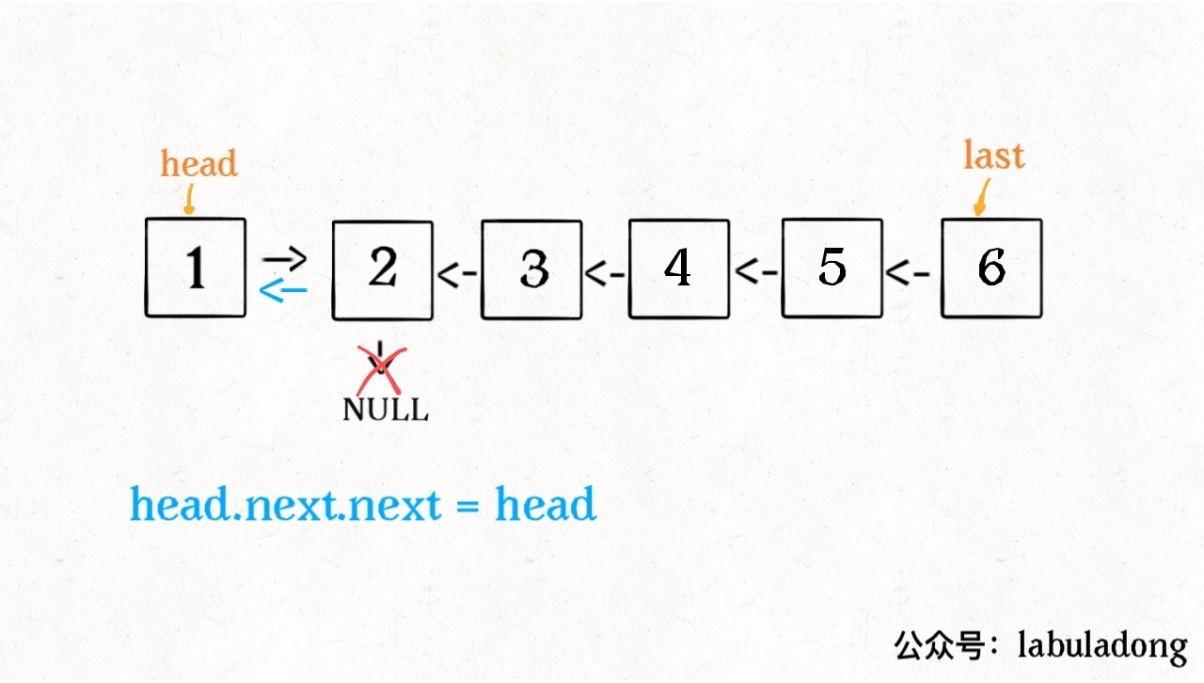
这个 `reverse(head.next)` 执行完成后，整个链表就成了这样：



并且根据函数定义，`reverse` 函数会返回反转之后的头结点，我们用变量 `last` 接收了。

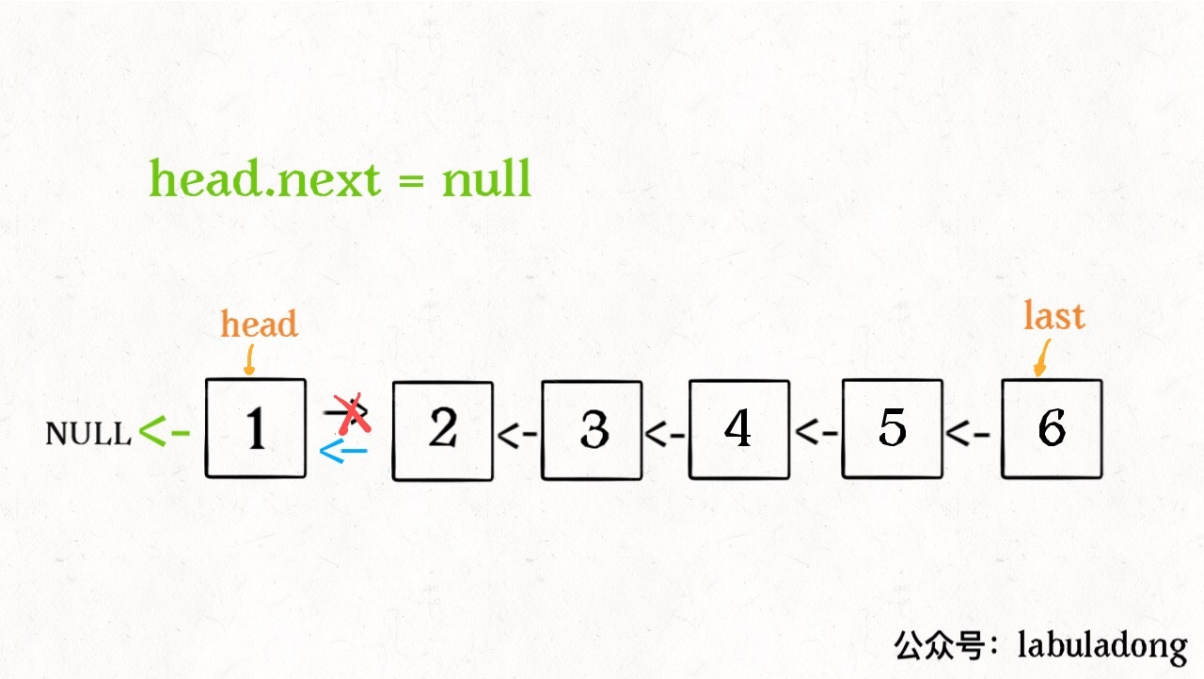
现在再来看下面的代码：

```
head.next.next = head;
```



接下来：

```
head.next = null;
return last;
```



神不神奇，这样整个链表就反转过来了！递归代码就是这么简洁优雅，不过其中有两个地方需要注意：

1、递归函数要有 base case，也就是这句：

```
if (head.next == null) return head;
```

意思是如果链表只有一个节点的时候反转也是它自己，直接返回即可。

2、当链表递归反转之后，新的头结点是 `last`，而之前的 `head` 变成了最后一个节点，别忘了链表的末尾要指向 null：

```
head.next = null;
```

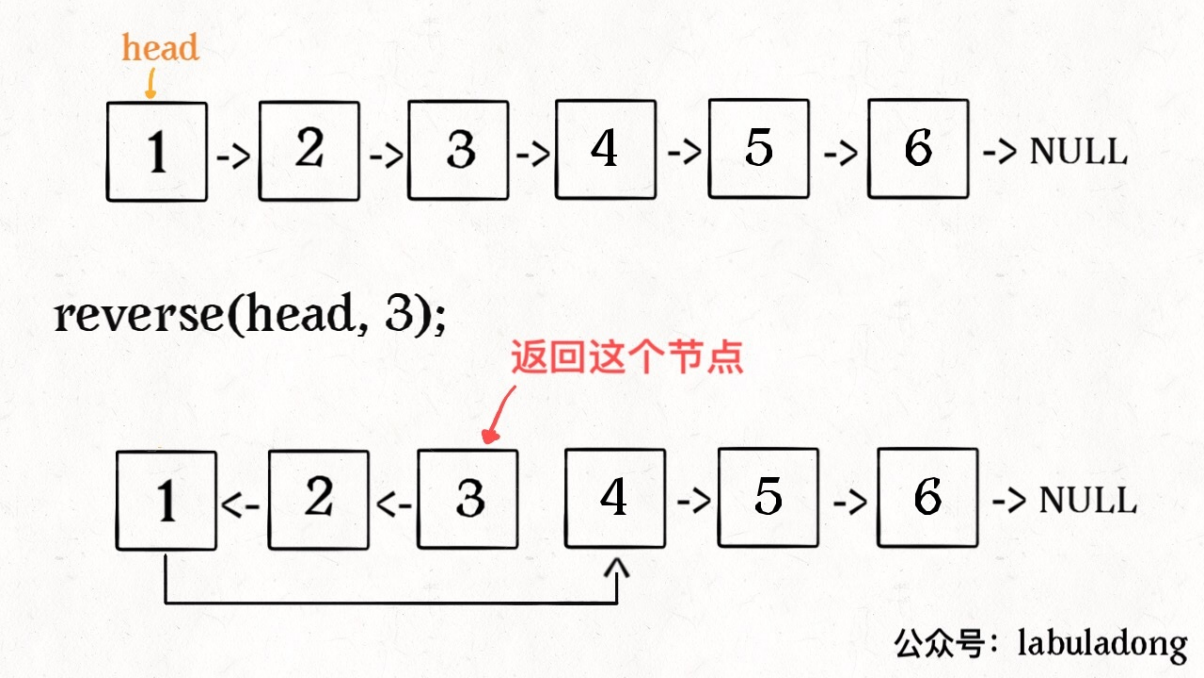
理解了这两点后，我们就可以进一步深入了，接下来的问题其实都是在这个算法上的扩展。

二、反转链表前 N 个节点

这次我们实现一个这样的函数：

```
// 将链表的前 n 个节点反转（n <= 链表长度）
ListNode reverseN(ListNode head, int n)
```

比如说对于下图链表，执行 `reverseN(head, 3)`：



解决思路和反转整个链表差不多，只要稍加修改即可：

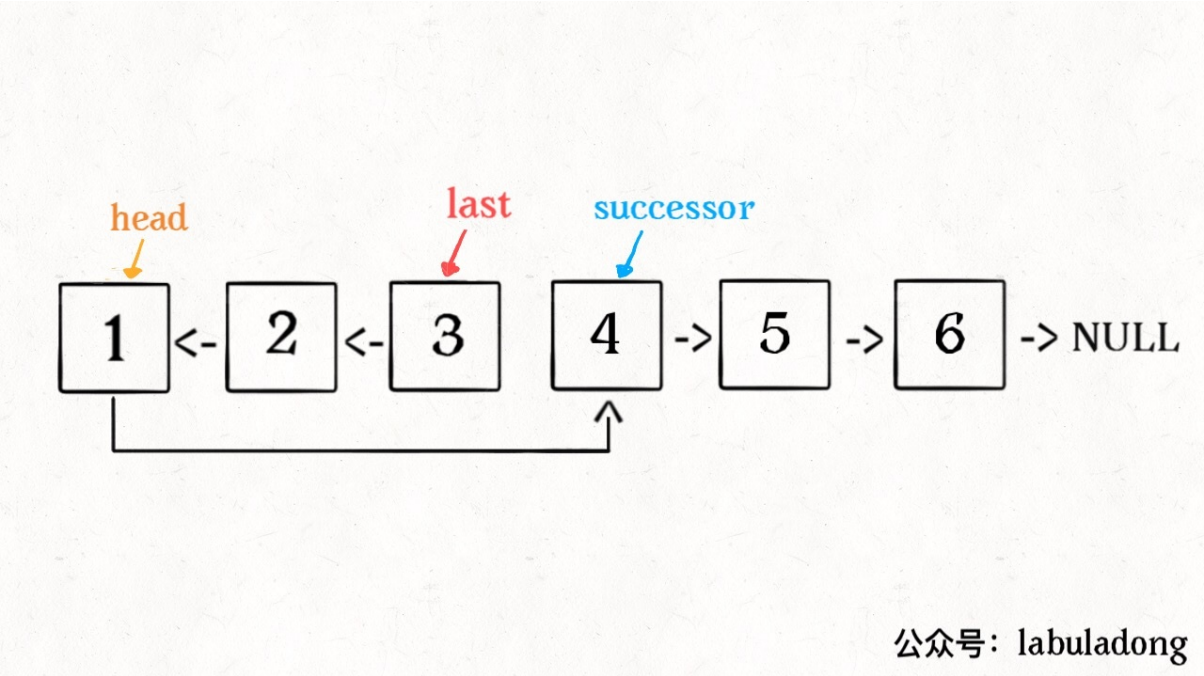
```
ListNode successor = null; // 后驱节点

// 反转以 head 为起点的 n 个节点，返回新的头结点
ListNode reverseN(ListNode head, int n) {
    if (n == 1) {
        // 记录第 n + 1 个节点
        successor = head.next;
        return head;
    }
    // 以 head.next 为起点，需要反转前 n - 1 个节点
    ListNode last = reverseN(head.next, n - 1);

    head.next.next = head;
```

```
        // 让反转之后的 head 节点和后面的节点连起来
        head.next = successor;
        return last;
    }
```

具体的区别：

1、base case 变为 `n == 1`，反转一个元素，就是它本身，同时**要记录后驱节点**。

2、刚才我们直接把 `head.next` 设置为 null，因为整个链表反转后原来的 `head` 变成了整个链表的最后一个节点。但现在 `head` 节点在递归反转之后不一定是最后一个节点了，所以要记录后驱 `successor`（第 n + 1 个节点），反转之后将 `head` 连接上。



OK，如果这个函数你也能看懂，就离实现「反转一部分链表」不远了。

三、反转链表的一部分

现在解决我们最开始提出的问题，给一个索引区间 `[m,n]`（索引从 1 开始），仅仅反转区间中的链表元素。

```
ListNode reverseBetween(ListNode head, int m, int n)
```

首先，如果 `m == 1`，就相当于反转链表开头的 `n` 个元素嘛，也就是我们刚才实现的功能：

```
ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        // 相当于反转前 n 个元素
        return reverseN(head, n);
    }
    // ...
}
```

如果 `m != 1` 怎么办？如果我们把 `head` 的索引视为 1，那么我们是想从第 `m` 个元素开始反转对吧；如果把 `head.next` 的索引视为 1 呢？那么相对于 `head.next`，反转的区间应该是从第 `m - 1` 个元素开始的；那么对于 `head.next.next` 呢.....

区别于迭代思想，这就是递归思想，所以我们可以完成代码：

```
ListNode reverseBetween(ListNode head, int m, int n) {
    // base case
    if (m == 1) {
        return reverseN(head, n);
    }
    // 前进到反转的起点触发 base case
    head.next = reverseBetween(head.next, m - 1, n - 1);
    return head;
}
```

至此，我们的最终大 BOSS 就被解决了。

四、最后总结

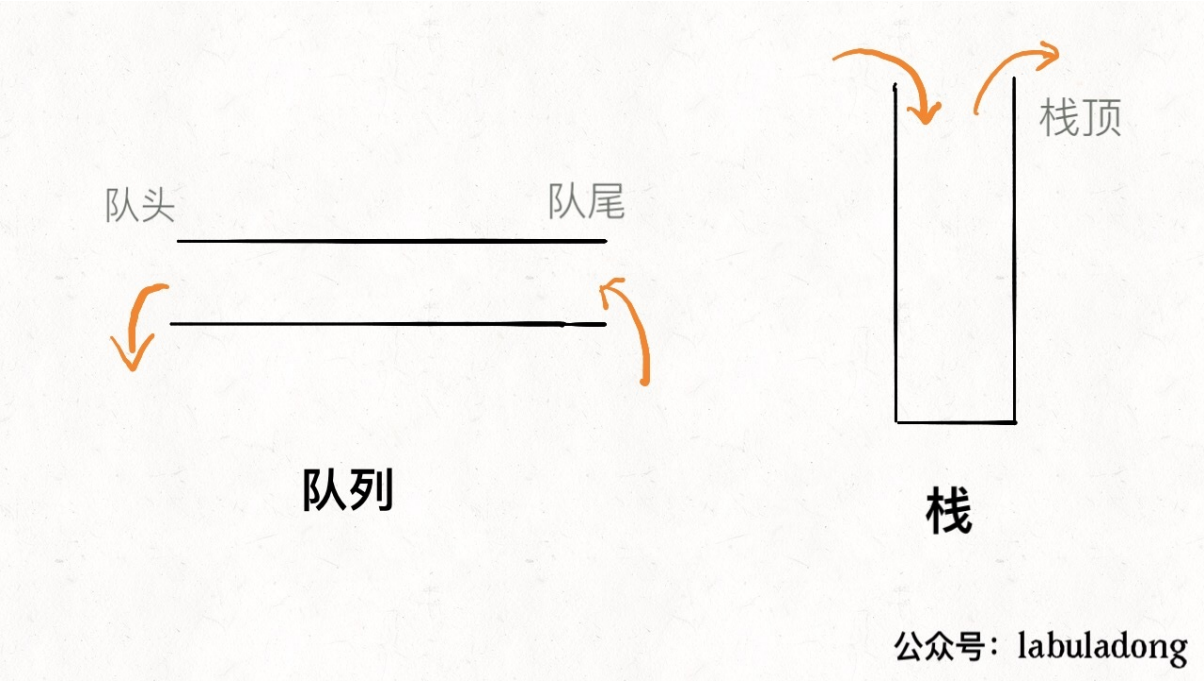
递归的思想相对迭代思想，稍微有点难以理解，处理的技巧是：不要跳进递归，而是利用明确的定义来实现算法逻辑。

处理看起来比较困难的问题，可以尝试化整为零，把一些简单的解法进行修改，解决困难的问题。

值得一提的是，递归操作链表并不高效。和迭代解法相比，虽然时间复杂度都是 O(N)，但是迭代解法的空间复杂度是 O(1)，而递归解法需要堆栈，空间复杂度是 O(N)。所以递归操作链表可以作为对递归算法的练习或者拿去和小伙伴装逼，但是考虑效率的话还是使用迭代算法更好。

队列实现栈|栈实现队列

队列是一种先进先出的数据结构，栈是一种先进后出的数据结构，形象一点就是这样：



这两种数据结构底层其实都是数组或者链表实现的，只是 API 限定了它们的特性，那么今天就来看看如何使用「栈」的特性来实现一个「队列」，如何用「队列」实现一个「栈」。

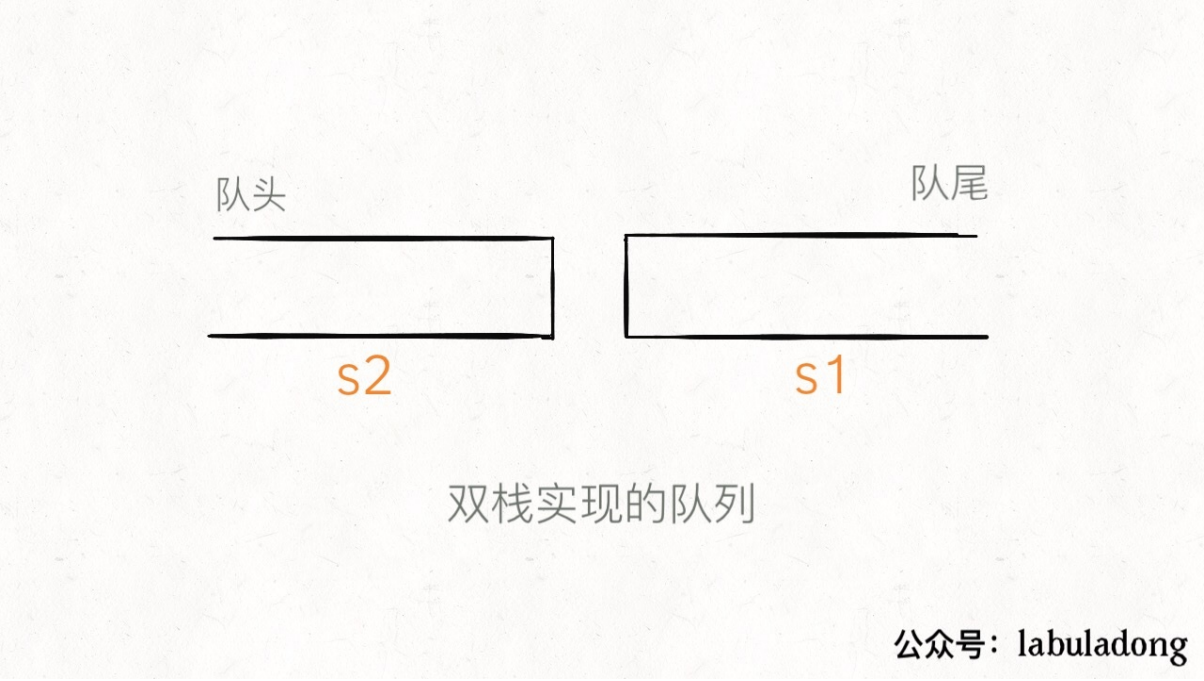
一、用栈实现队列

首先，队列的 API 如下：

```
class MyQueue {

    /** 添加元素到队尾 */
    public void push(int x);

    /** 删除队头的元素并返回 */
    public int pop();
```

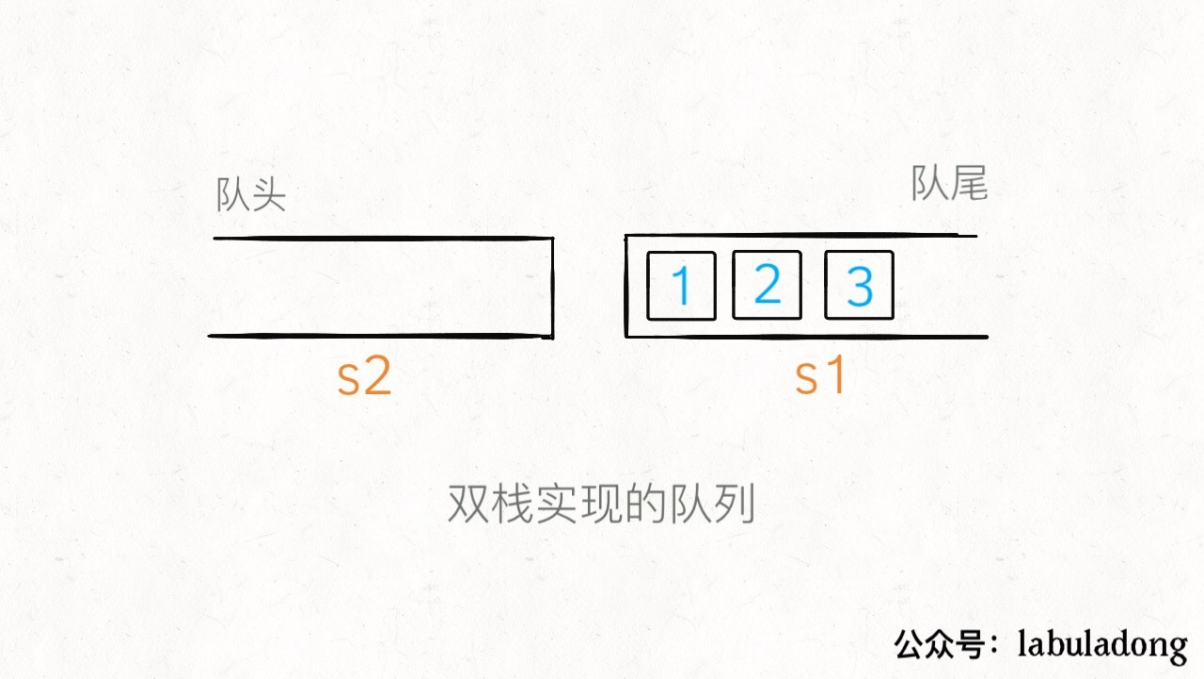
```
    /** 返回队头元素 */
    public int peek();

    /** 判断队列是否为空 */
    public boolean empty();
}
```

我们使用两个栈 `s1, s2` 就能实现一个队列的功能（这样放置栈可能更容易理解）：



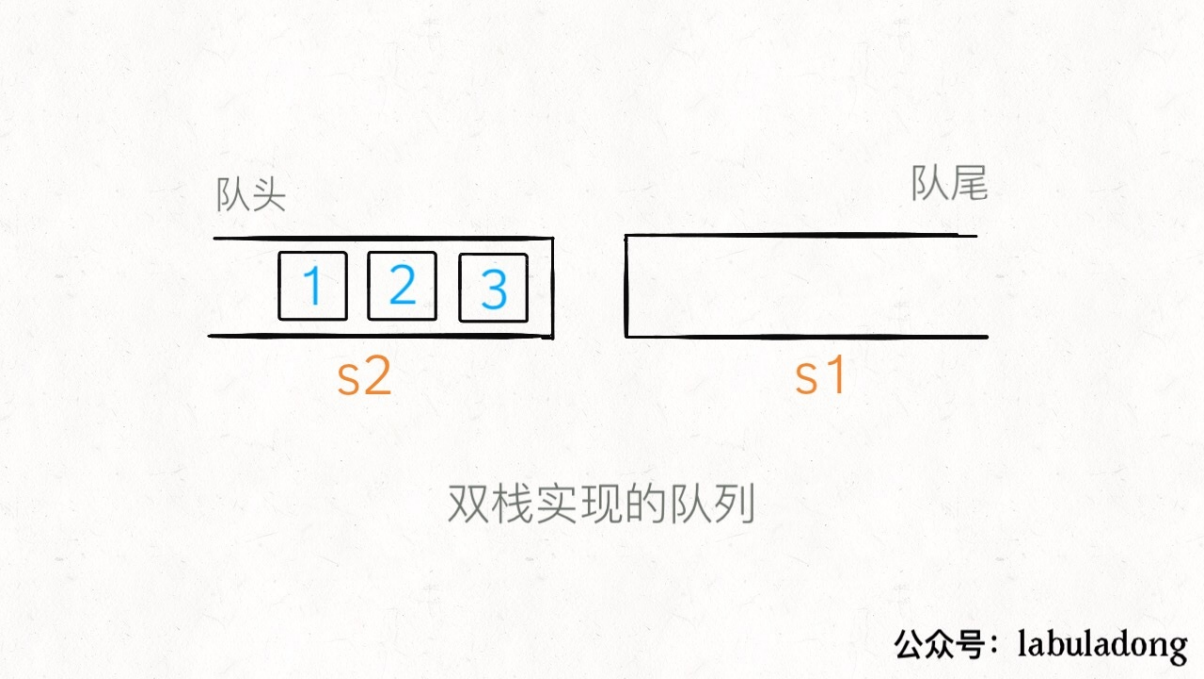

```
class MyQueue {
    private Stack<Integer> s1, s2;

    public MyQueue() {
        s1 = new Stack<>();
        s2 = new Stack<>();
    }
    // ...
}
```

当调用 `push` 让元素入队时，只要把元素压入 `s1` 即可，比如说 `push` 进 3 个元素分别是 1,2,3，那么底层结构就是这样：




```
/** 添加元素到队尾 */
public void push(int x) {
    s1.push(x);
}
```

那么如果这时候使用 `peek` 查看队头的元素怎么办呢？按道理队头元素应该是 1，但是在 `s1` 中 1 被压在栈底，现在就要轮到 `s2` 起到一个中转的作用了：当 `s2` 为空时，可以把 `s1` 的所有元素取出再添加进 `s2`，**这时候 `s2` 中元素就是先进先出顺序了**。




```
/** 返回队头元素 */
public int peek() {
    if (s2.isEmpty())
        // 把 s1 元素压入 s2
        while (!s1.isEmpty())
            s2.push(s1.pop());
    return s2.peek();
}
```

同理，对于 `pop` 操作，只要操作 `s2` 就可以了。

```
/** 删除队头的元素并返回 */
public int pop() {
    // 先调用 peek 保证 s2 非空
    peek();
    return s2.pop();
}
```

最后，如何判断队列是否为空呢？如果两个栈都为空的话，就说明队列为空：

```
/** 判断队列是否为空 */
public boolean empty() {
```

```
        return s1.isEmpty() && s2.isEmpty();
    }
```

至此，就用栈结构实现了一个队列，核心思想是利用两个栈互相配合。

值得一提的是，这几个操作的时间复杂度是多少呢？有点意思的是 `peek` 操作，调用它时可能触发 `while` 循环，这样的话时间复杂度是 O(N)，但是大部分情况下 `while` 循环不会被触发，时间复杂度是 O(1)。由于 `pop` 操作调用了 `peek`，它的时间复杂度和 `peek` 相同。

像这种情况，可以说它们的**最坏时间复杂度**是 O(N)，因为包含 `while` 循环，**可能**需要从 `s1` 往 `s2` 搬移元素。

但是它们的**均摊时间复杂度**是 O(1)，这个要这么理解：对于一个元素，最多只可能被搬运一次，也就是说 `peek` 操作平均到每个元素的时间复杂度是 O(1)。

二、用队列实现栈

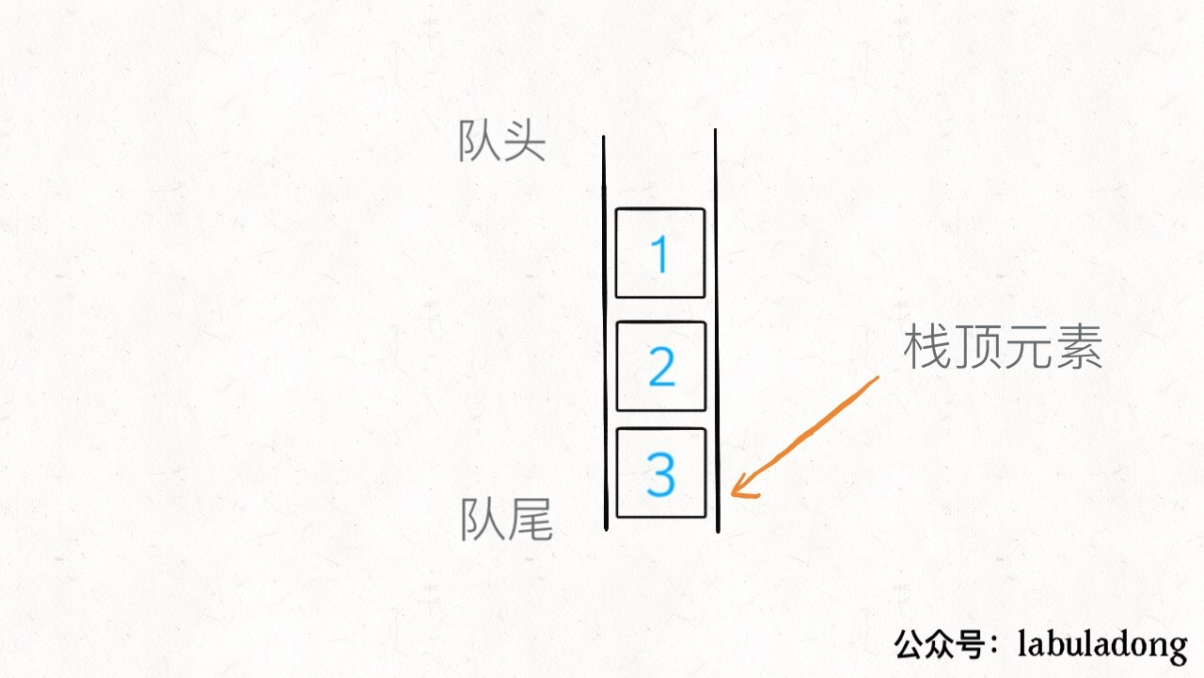
如果说双栈实现队列比较巧妙，那么用队列实现栈就比较简单粗暴了，只需要一个队列作为底层数据结构。首先看下栈的 API：

```
class MyStack {

    /** 添加元素到栈顶 */
    public void push(int x);

    /** 删除栈顶的元素并返回 */
    public int pop();

    /** 返回栈顶元素 */
    public int top();

    /** 判断栈是否为空 */
    public boolean empty();
}
```

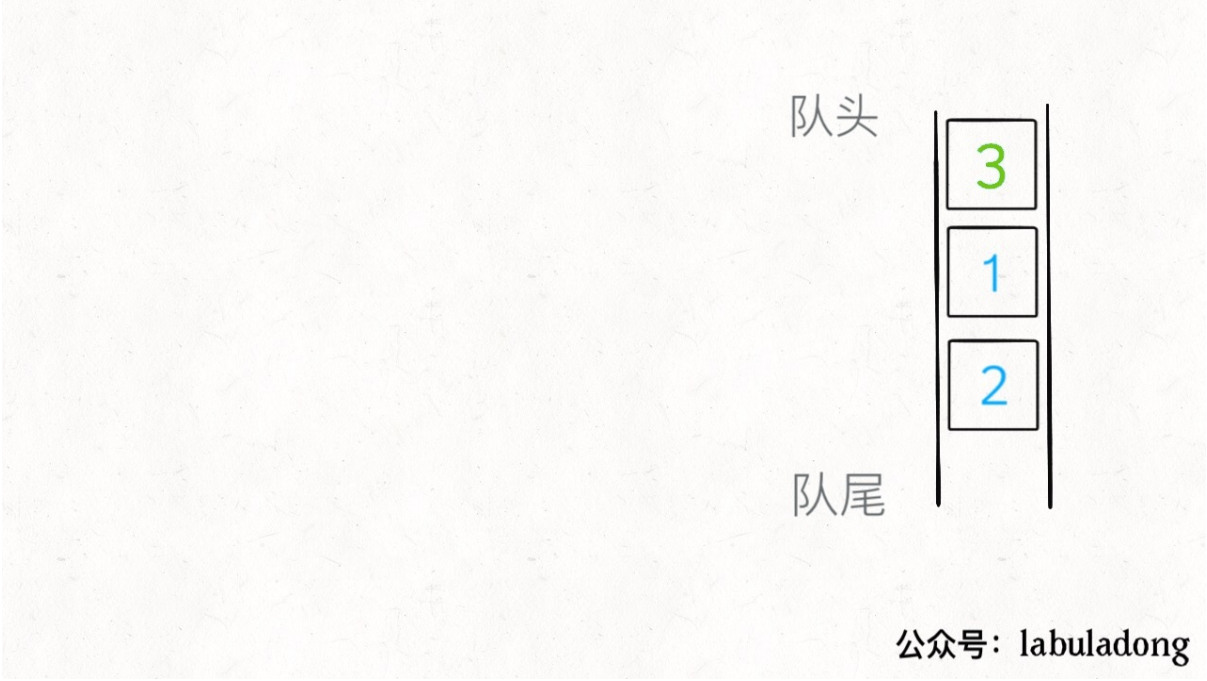
先说 `push` API，直接将元素加入队列，同时记录队尾元素，因为队尾元素相当于栈顶元素，如果要 `top` 查看栈顶元素的话可以直接返回：

```
class MyStack {
    Queue<Integer> q = new LinkedList<>();
    int top_elem = 0;

    /** 添加元素到栈顶 */
    public void push(int x) {
        // x 是队列的队尾，是栈的栈顶
        q.offer(x);
        top_elem = x;
    }

    /** 返回栈顶元素 */
    public int top() {
        return top_elem;
    }
}
```

我们的底层数据结构是先进先出的队列，每次 `pop` 只能从队头取元素；但是栈是后进先出，也就是说 `pop` API 要从队尾取元素。



解决方法简单粗暴，把队列前面的都取出来再加入队尾，让之前的队尾元素排到队头，这样就可以取出了：




```
/** 删除栈顶的元素并返回 */
public int pop() {
    int size = q.size();
    while (size > 1) {
        q.offer(q.poll());
        size--;
    }
    // 之前的队尾元素已经到了队头
    return q.poll();
}
```

这样实现还有一点小问题就是，原来的队尾元素被提到队头并删除了，但是 `top_elem` 变量没有更新，我们还需要一点小修改：

```
/** 删除栈顶的元素并返回 */
public int pop() {
    int size = q.size();
    // 留下队尾 2 个元素
    while (size > 2) {
        q.offer(q.poll());
        size--;
```

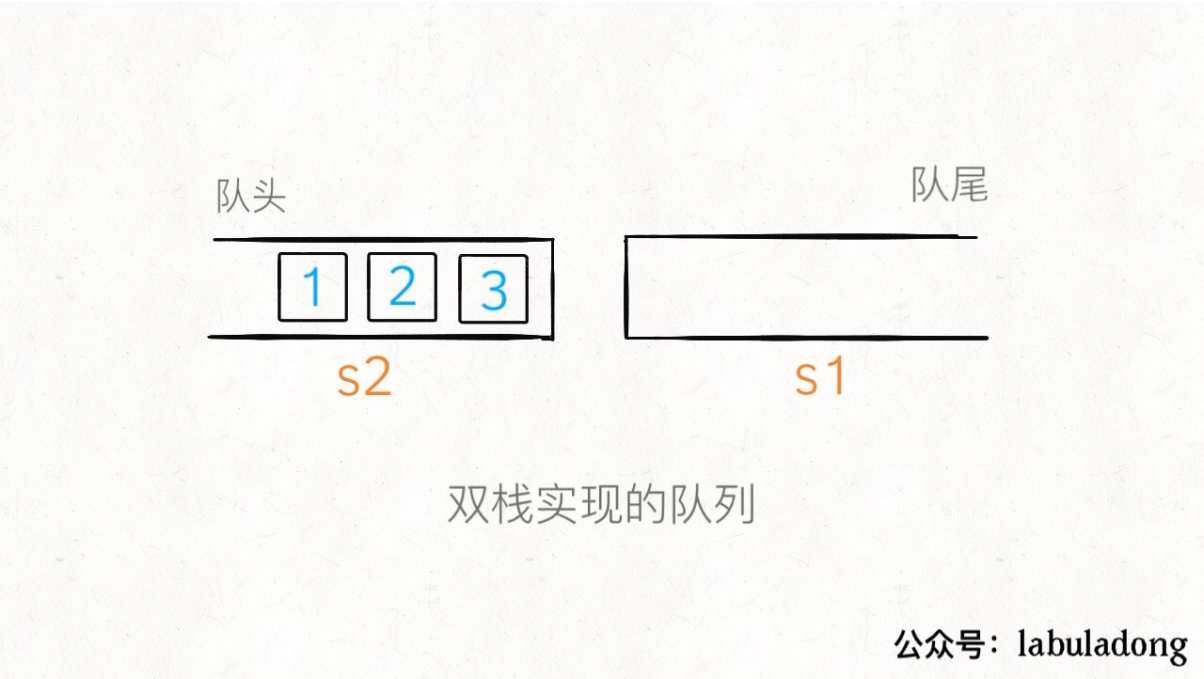
```
        }
        // 记录新的队尾元素
        top_elem = q.peek();
        q.offer(q.poll());
        // 删除之前的队尾元素
        return q.poll();
    }
```

最后，API `empty` 就很容易实现了，只要看底层的队列是否为空即可：

```
/** 判断栈是否为空 */
public boolean empty() {
    return q.isEmpty();
}
```

很明显，用队列实现栈的话，`pop` 操作时间复杂度是 O(N)，其他操作都是 O(1)。

个人认为，用队列实现栈是没啥亮点的问题，但是**用双栈实现队列是值得学习的**。



从栈 `s1` 搬运元素到 `s2` 之后，元素在 `s2` 中就变成了队列的先进先出顺序，这个特性有点类似「负负得正」，确实不太容易想到。

希望本文对你有帮助。

算法思维系列

本章包含一些常用的算法技巧，比如前缀和、回溯思想、位操作、双指针、如何正确书写二分查找等等。

欢迎关注我的公众号 labuladong，方便获得最新的优质文章：

算法学习之路

之前发的那篇关于框架性思维的文章，我也发到了不少其他圈子，受到了大家的普遍好评，这一点我真的没想到，首先感谢大家的认可，我会更加努力，写出通俗易懂的算法文章。

有很多朋友问我数据结构和算法到底该怎么学，尤其是很多朋友说自己是「小白」，感觉这些东西好难啊，就算看了之前的「框架思维」，也感觉自己刷题乏力，希望我能聊聊我从一个非科班小白一路是怎么学过来的。

首先要给怀有这样疑问的朋友鼓掌，因为你现在已经「知道自己不知道」，而且开始尝试学习、刷题、寻求帮助，能做到这一点本身就是及其困难的。

关于「框架性思维」，对于一个小白来说，可能暂时无法完全理解（如果你能理解，说明你水平已经不错啦，不是小白啦）。就像软件工程，对于我这种没带过项目的人来说，感觉其内容枯燥乏味，全是废话，但是对于一个带过团队的人，他就会觉得软件工程里的每一句话都是精华。暂时不太理解没关系，留个印象，功夫到了很快就明白了。

下面写一写我一路过来的一些经验。如果你已经看过很多「如何高效刷题」「如何学习算法」的文章，却还是没有开始行动并坚持下去，本文的第五点就是写给你的。

我觉得之所以有时候认为自己是「小白」，是由于知识某些方面的空白造成的。具体到数据结构的学习，无非就是两个问题搞得不太清楚：**这是啥？有啥用？**

举个例子，比如说你看到了「栈」这个名词，老师可能会讲这些关键词：先进后出、函数堆栈等等。但是，对于初学者，这些描述属于文学词汇，没有实际价值，没有解决最基本的两个问题。如何回答这两个基本问题呢？回答「这是啥」需要看教科书，回答「有啥用」需要刷算法题。

一、这是啥？

这个问题最容易解决，就像一层窗户纸，你只要随便找本书看两天，自己动手实现一个「队列」「栈」之类的数据结构，就能捅破这层窗户纸。

这时候你就能理解「框架思维」文章中的前半部分了：数据结构无非就是数组、链表为骨架的一些特定操作而已；每个数据结构实现的功能无非增删查改罢了。

比如说「列队」这个数据结构，无非就是基于数组或者链表，实现 enqueue 和 dequeue 两个方法。这两个方法就是增和删呀，连查和改的方法都不需要。

二、有啥用？

解决这个问题，就涉及算法的设计了，是个持久战，需要经常进行抽象思考，刷算法题，培养「计算机思维」。

之前的文章讲了，算法就是对数据结构准确而巧妙的运用。常用算法问题也就那几大类，算法题无非就是不断变换场景，给那几个算法框架套上不同的皮。刷题，就是在锻炼你的眼力，看你能不能看穿问题表象揪出相应的解法框架。

比如说，让你求解一个迷宫，你要把这个问题层层抽象：迷宫 -> 图的遍历 -> N 叉树的遍历 -> 二叉树的遍历。然后让框架指导你写具体的解法。

抽象问题，直击本质，是刷题中你需要刻意培养的能力。

三、如何看书

直接推荐一本公认的好书，《算法第 4 版》，我一般简写成《算法4》。不要蜻蜓点水，这本书你能选择性的看上 50%，基本上就达到平均水平了。别怕这本书厚，因为起码有三分之一不用看，下面讲讲怎么看这本书。

看书仍然遵循递归的思想：自顶向下，逐步求精。

这本书知识结构合理，讲解也清楚，所以可以按顺序学习。**书中正文的算法代码一定要亲自敲一遍**，因为这些真的是扎实的基础，要认真理解。不要以为自己看一遍就看懂了，不动手的话理解不了的。但是，开头部分的基础可

以酌情跳过；书中的数学证明，如不影响对算法本身的理解，完全可以跳过；章节最后的练习题，也可以全部跳过。这样一来，这本书就薄了很多。

相信读者现在已经认可了「框架性思维」的重要性，这种看书方式也是一种框架性策略，抓大放小，着重理解整体的知识架构，而忽略证明、练习题这种细节问题，即**保持自己对新知识的好奇心，避免陷入无限的细节被劝退**。

当然，《算法4》到后面的内容也比较难了，比如那几个著名的串算法，以及正则表达式算法。这些属于「经典算法」，看个人接受能力吧，单说刷 LeetCode 的话，基本用不上，量力而行即可。

四、如何刷题

首先声明一下，**算法和数学水平没关系，和编程语言也没关系**，你爱用什么语言用什么。算法，主要是培养一种新的思维方式。所谓「计算机思维」，就跟你考驾照一样，你以前骑自行车，有一套自行车的规则和技巧，现在你开汽车，就需要适应并练习开汽车的规则和技巧。

LeetCode 上的算法题和前面说的「经典算法」不一样，我们权且称为「解闷算法」吧，因为很多题目都比较有趣，有种在做奥数题或者脑筋急转弯的感觉。比如说，让你用队列实现一个栈，或者用栈实现一个队列，以及不用加号做加法，开脑洞吧？

当然，这些问题虽然看起来无厘头，实际生活中也用不到，但是想解决这些问题依然要靠数据结构以及对基础知识的理解，也许这就是很多公司面试都喜欢出这种「智力题」的原因。下面说几点技巧吧。

尽量刷英文版的 LeetCode，中文版的“力扣”是阉割版，不仅很多题目没有答案，而且连个讨论区都没有。英文版的是真的很良心了，很多问题都有官方解答，详细易懂。而且讨论区（Discuss）也沉淀了大量优质内容，甚至好过官方解答。真正能打开你思路的，很可能是讨论区各路大神的思路荟萃。

PS：**如果有的英文题目实在看不懂，有个小技巧**，你在题目页面的 url 里加一个 -cn，即 https://leetcode.com/xxx 改成 https://leetcode-cn.com/xxx，这样就能切换到相应的中文版页面查看。

对于初学者，**强烈建议从 Explore 菜单里最下面的 Learn 开始刷**，这个专题就是专门教你学习数据结构和基本算法的，教学篇和相应的练习题结合，不要太良心。

最近 Learn 专题里新增了一些内容，我们挑数据结构相关的内容刷就行了，像 Ruby，Machine Learning 就没必要刷了。刷完 Learn 专题的基础内容，基本就有能力去 Explore 菜单的 Interview 专题刷面试题，或者去 Problem 菜单，在真正的题海里遨游了。

无论刷 Explore 还是 Problems 菜单，**最好一个分类一个分类的刷，不要蜻蜓点水**。比如说这几天就刷链表，刷完链表再去连刷几天二叉树。这样做是为了帮助你提取「框架」。一旦总结出针对一类问题的框架，解决同类问题可谓是手到擒来。

五、道理我都懂，还是不能坚持下去

这其实无关算法了，还是老生常谈的执行力的问题。不说什么破鸡汤了，我觉得**解决办法就是「激起欲望」**，注意我说的是欲望，而不是常说的兴趣，拿我自己说说吧。

半年前我开始刷题，目的和大部分人都一样的，就是为毕业找工作做准备。只不过，大部分人是等到临近毕业了才开始刷，而我离毕业还有一阵子。这不是炫耀我多有觉悟，而是我承认自己的极度平凡。

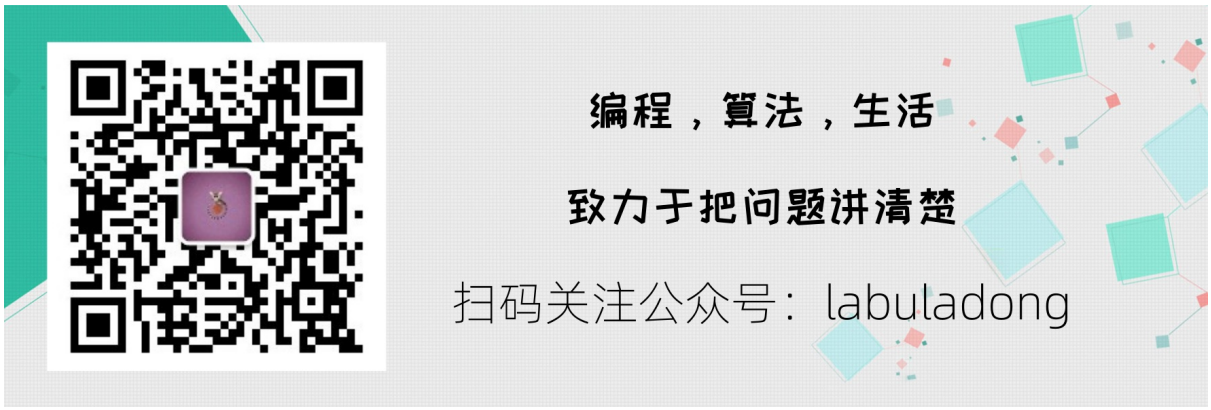
首先，我真的想找到一份不错的工作（谁都想吧？），我想要高薪呀！否则我在朋友面前，女神面前放下的骚话，最终都会反过来啪啪地打我的脸。我也是要恰饭，要面子，要虚荣心的嘛。赚钱，虚荣心，足以激起我的欲望了。

但是，我不擅长 deadline 突击，我理解东西真的慢，所以干脆笨鸟先飞了。智商不够，拿时间来补，我没能力两个月突击，干脆拉长战线，打他个两年游击战，我还不信耗不死算法这个强敌。事实证明，你如果认真学习一个月，就能够取得肉眼可见的进步了。

现在，我依然在坚持刷题，而且为了另外一个原因，这个公众号。我没想到自己的文字竟然能够帮助到他人，甚至能得到认可。这也是虚荣心啊，我不能让读者失望啊，我想让更多的人认可（夸）我呀！

以上，不光是坚持刷算法题吧，很多场景都适用。执行力是要靠「欲望」支撑的，我也是一凡人，只有那些看得见摸得着的东西才能使我快乐呀。读者不妨也尝试把刷题学习和自己的切身利益联系起来，这恐怕是坚持下去最简单直白的理由了。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：



回溯算法详解

这篇文章是很久之前的一篇《回溯算法详解》的进阶版，之前那篇不够清楚，就不必看了，看这篇就行。把框架给你讲清楚，你会发现回溯算法问题都是一个套路。

废话不多说，直接上回溯算法框架。**解决一个回溯问题，实际上就是一个决策树的遍历过程**。你只需要思考 3 个问题：

1、路径：也就是已经做出的选择。

2、选择列表：也就是你当前可以做的选择。

3、结束条件：也就是到达决策树底层，无法再做选择的条件。

如果你不理解这三个词语的解释，没关系，我们后面会用「全排列」和「N 皇后问题」这两个经典的回溯算法问题来帮你理解这些词语是什么意思，现在你先留着印象。

代码方面，回溯算法的框架：

```
result = []
def backtrack(路径, 选择列表):
    if 满足结束条件:
        result.add(路径)
        return

    for 选择 in 选择列表:
        做选择
        backtrack(路径, 选择列表)
        撤销选择
```

其核心就是 for 循环里面的递归，在递归调用之前「做选择」，在递归调用之后「撤销选择」，特别简单。

什么叫做选择和撤销选择呢，这个框架的底层原理是什么呢？下面我们就通过「全排列」这个问题来解开之前的疑惑，详细探究一下其中的奥妙！

一、全排列问题

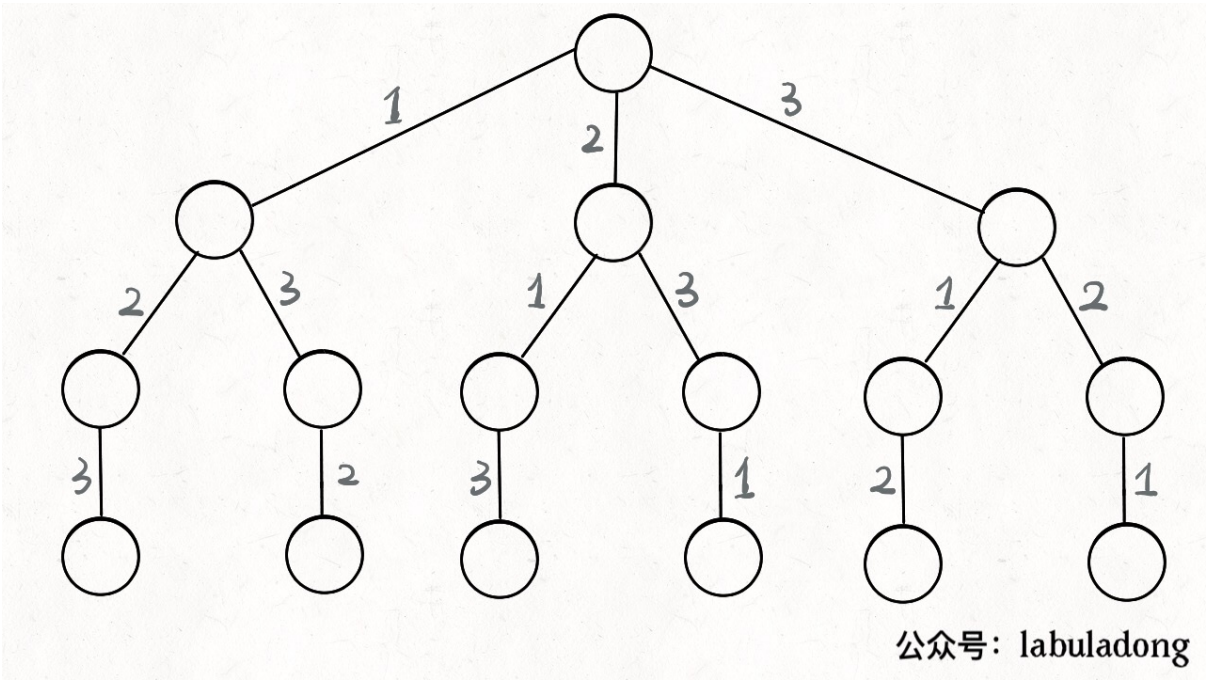
我们在高中的时候就做过排列组合的数学题，我们也知道 `n` 个不重复的数，全排列共有 n! 个。

PS：**为了简单清晰起见，我们这次讨论的全排列问题不包含重复的数字**。

那么我们当时是怎么穷举全排列的呢？比方说给三个数 `[1,2,3]`，你肯定不会无规律地乱穷举，一般是这样：

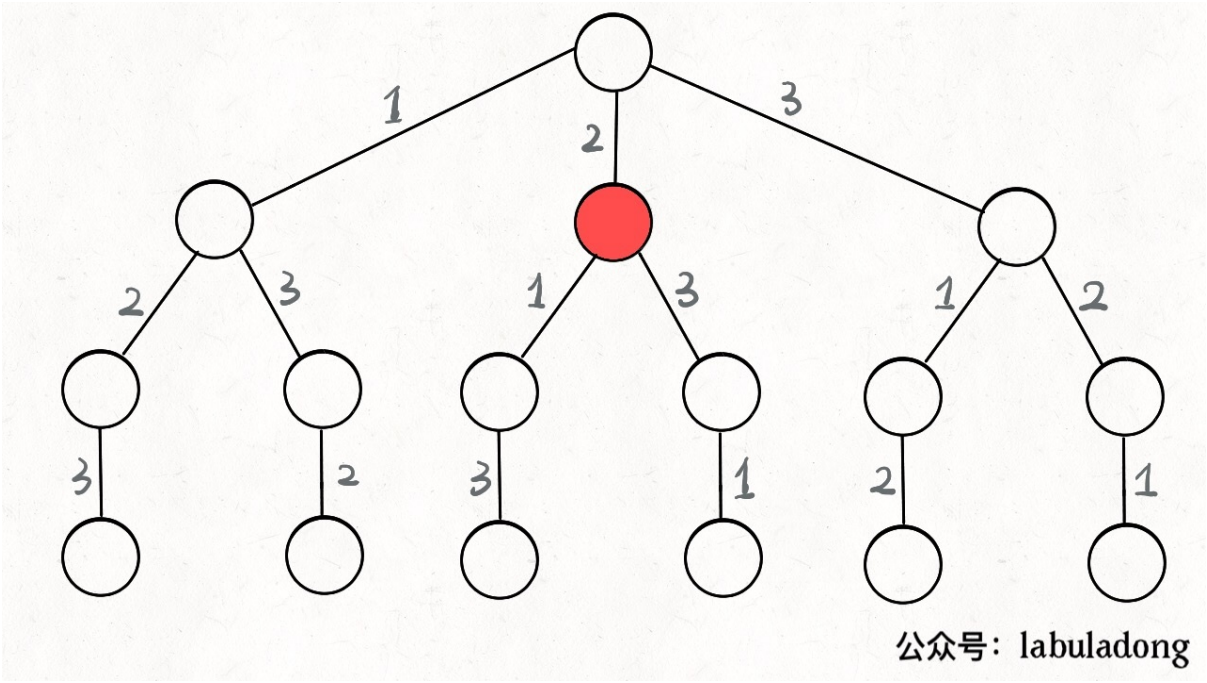
先固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；然后就只能变化第一位，变成 2，然后再穷举后两位……

其实这就是回溯算法，我们高中无师自通就会用，或者有的同学直接画出如下这棵回溯树：



只要从根遍历这棵树，记录路径上的数字，其实就是所有的全排列。**我们不妨把这棵树称为回溯算法的「决策树」**。

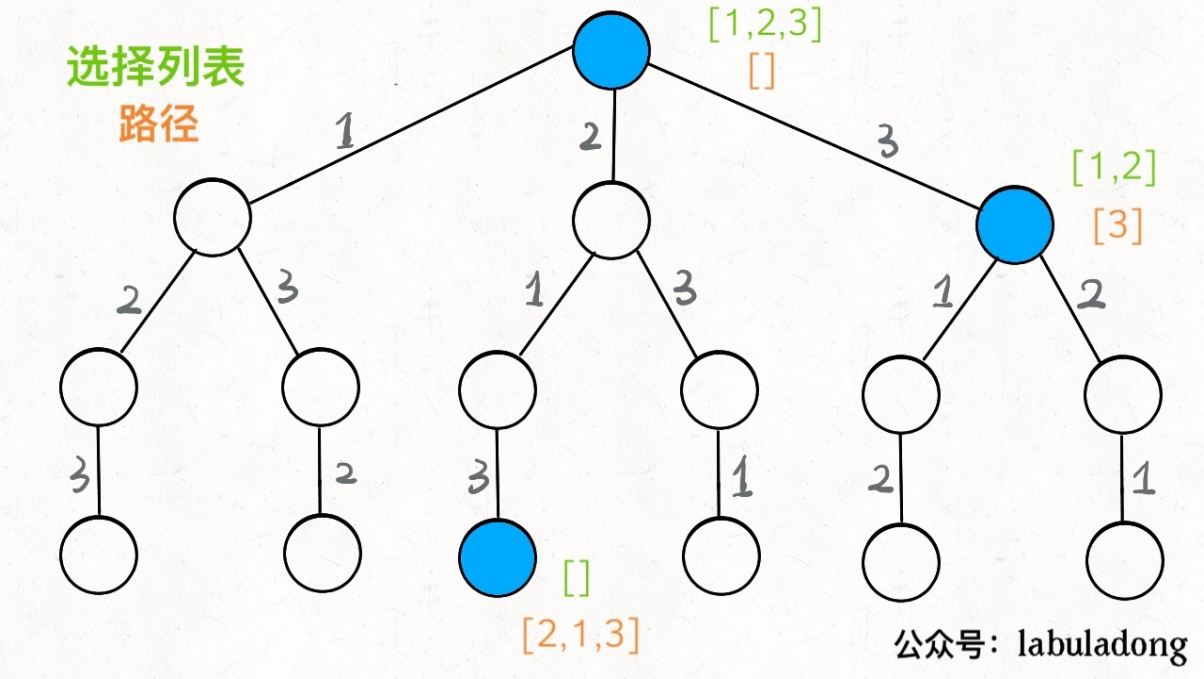
为啥说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



你现在就在做决策，可以选择 1 那条树枝，也可以选择 3 那条树枝。为啥只能在 1 和 3 之中选择呢？因为 2 这个树枝在你身后，这个选择你之前做过了，而全排列是不允许重复使用数字的。

现在可以解答开头的几个名词：`[2]` 就是「路径」，记录你已经做过的选择；`[1,3]` 就是「选择列表」，表示你当前可以做出的选择；「结束条件」就是遍历到树的底层，在这里就是选择列表为空的时候。

如果明白了这几个名词，**可以把「路径」和「选择」列表作为决策树上每个节点的属性**，比如下图列出了几个节点的属性：

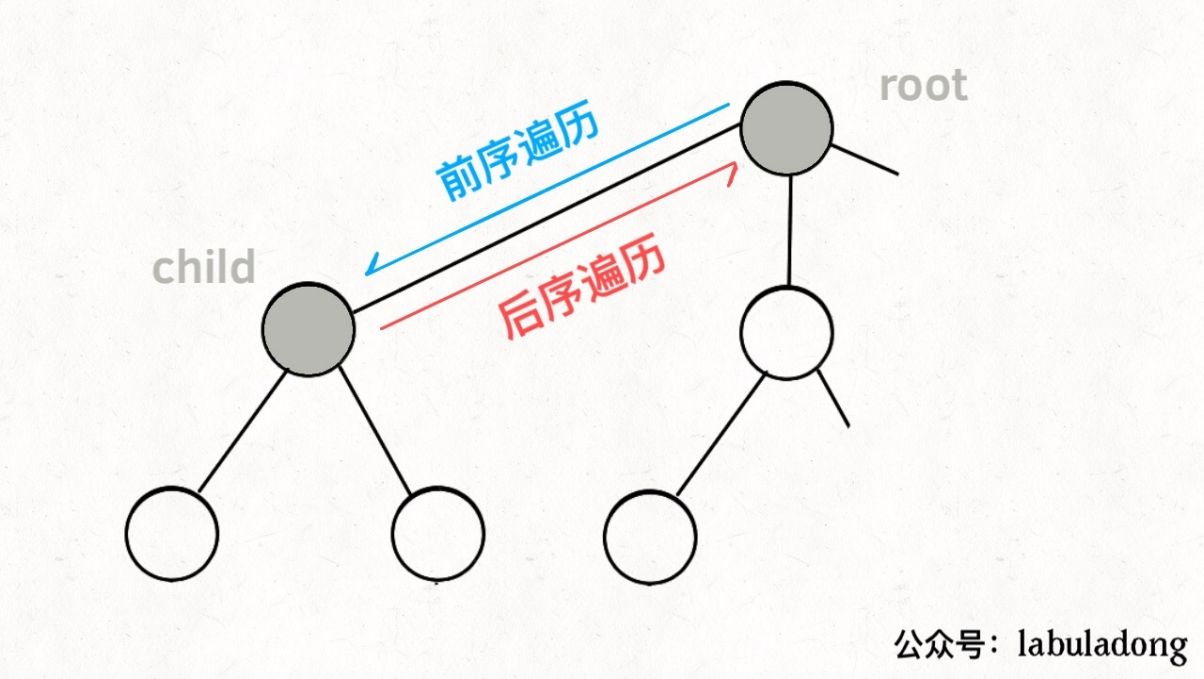


我们定义的 `backtrack` 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层，其「路径」就是一个全排列。

再进一步，如何遍历一棵树？这个应该不难吧。回忆一下之前「学习数据结构的框架思维」写过，各种搜索问题其实都是树的遍历问题，而多叉树的遍历框架就是这样：

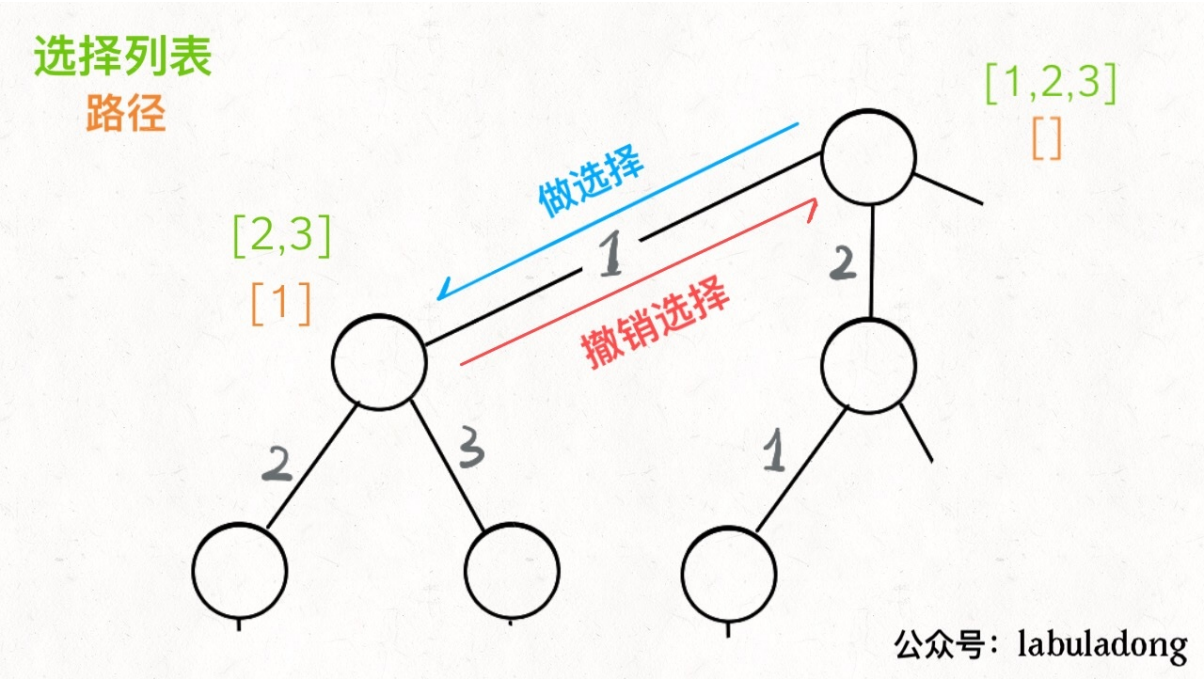
```
void traverse(TreeNode root) {
    for (TreeNode child : root.childern)
        // 前序遍历需要的操作
        traverse(child);
        // 后序遍历需要的操作
}
```

而所谓的前序遍历和后序遍历，他们只是两个很有用的时间点，我给你画张图你就明白了：



前序遍历的代码在进入某一个节点之前的那个时间点执行，后序遍历代码在离开某个节点之后的那个时间点执行。

回想我们刚才说的，「路径」和「选择」是每个节点的属性，函数在树上游走要正确维护节点的属性，那么就要在这两个特殊时间点搞点动作：



现在，你是否理解了回溯算法的这段核心框架？

```
for 选择 in 选择列表:
```

```
    # 做选择
    将该选择从选择列表移除
    路径.add(选择)
    backtrack(路径, 选择列表)
    # 撤销选择
    路径.remove(选择)
    将该选择再加入选择列表
```

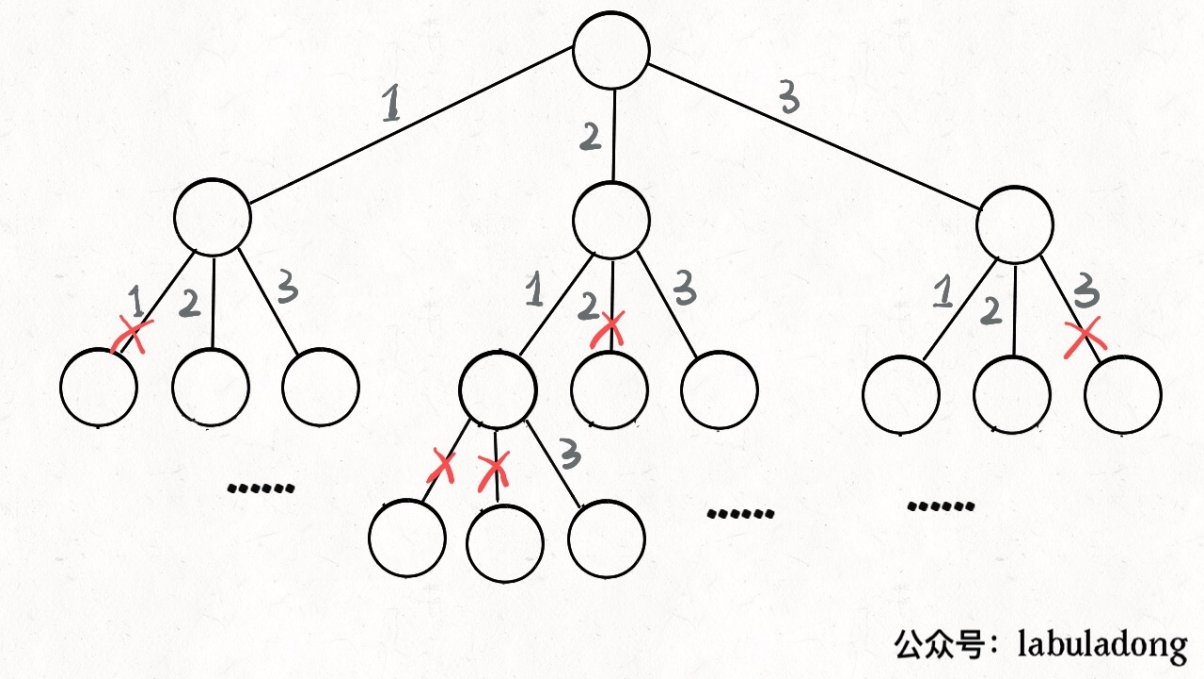
我们只要在递归之前做出选择，在递归之后撤销刚才的选择，就能正确得到每个节点的选择列表和路径。

下面，直接看全排列代码：

```
List<List<Integer>> res = new LinkedList<>();

/* 主函数，输入一组不重复的数字，返回它们的全排列 */
List<List<Integer>> permute(int[] nums) {
    // 记录「路径」
    LinkedList<Integer> track = new LinkedList<>();
    backtrack(nums, track);
    return res;
}

// 路径：记录在 track 中
// 选择列表：nums 中不存在于 track 的那些元素
// 结束条件：nums 中的元素全都在 track 中出现
void backtrack(int[] nums, LinkedList<Integer> track) {
    // 触发结束条件
    if (track.size() == nums.length) {
        res.add(new LinkedList(track));
        return;
    }

    for (int i = 0; i < nums.length; i++) {
        // 排除不合法的选择
        if (track.contains(nums[i]))
            continue;
        // 做选择
        track.add(nums[i]);
        // 进入下一层决策树
        backtrack(nums, track);
        // 取消选择
```

```
            track.removeLast();
        }
    }
```

我们这里稍微做了些变通，没有显式记录「选择列表」，而是通过 `nums` 和 `track` 推导出当前的选择列表：



至此，我们就通过全排列问题详解了回溯算法的底层原理。当然，这个算法解决全排列不是很高效，应为对链表使用 `contains` 方法需要 O(N) 的时间复杂度。有更好的方法通过交换元素达到目的，但是难理解一些，这里就不写了，有兴趣可以自行搜索一下。

但是必须说明的是，不管怎么优化，都符合回溯框架，而且时间复杂度都不可能低于 O(N!)，因为穷举整棵决策树是无法避免的。**这也是回溯算法的一个特点，不像动态规划存在重叠子问题可以优化，回溯算法就是纯暴力穷举，复杂度一般都很高**。

明白了全排列问题，就可以直接套回溯算法框架了，下面简单看看 N 皇后问题。

二、N 皇后问题

这个问题很经典了，简单解释一下：给你一个 N×N 的棋盘，让你放置 N 个皇后，使得它们不能互相攻击。

PS：皇后可以攻击同一行、同一列、左上左下右上右下四个方向的任意单位。

这个问题本质上跟全排列问题差不多，决策树的每一层表示棋盘上的每一行；每个节点可以做出的选择是，在该行的任意一列放置一个皇后。

直接套用框架:

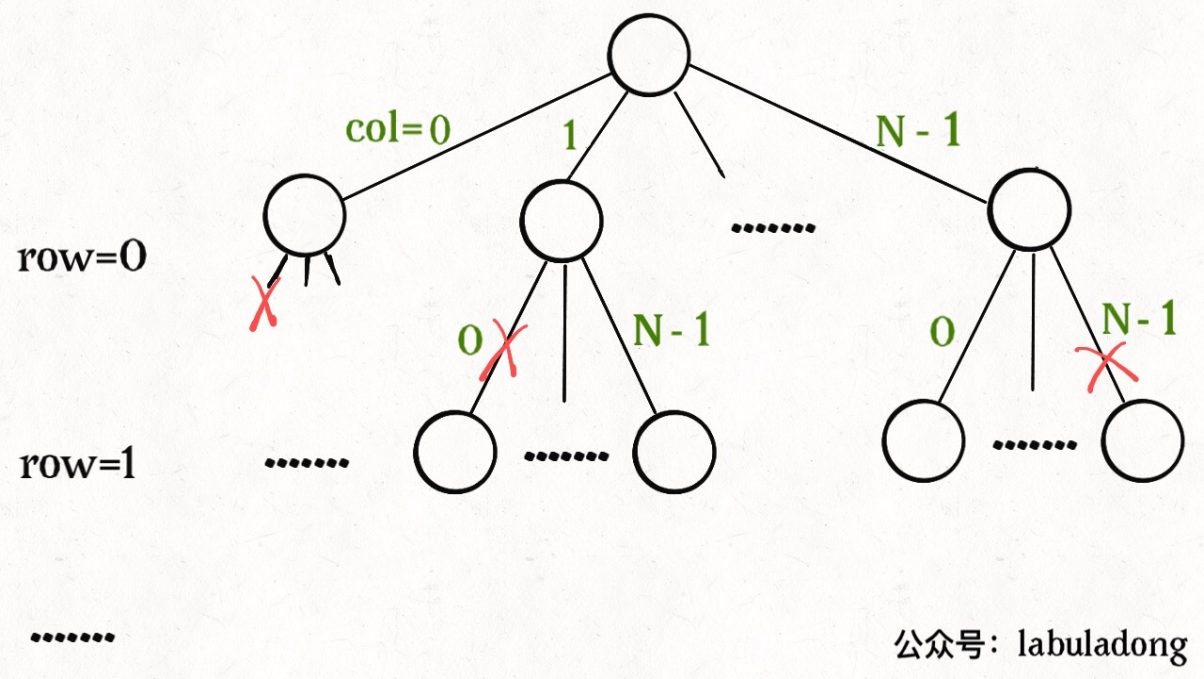
```
vector<vector<string>> res;

/* 输入棋盘边长 n，返回所有合法的放置 */
vector<vector<string>> solveNQueens(int n) {
    // '.' 表示空，'Q' 表示皇后，初始化空棋盘。
    vector<string> board(n, string(n, '.'));
    backtrack(board, 0);
    return res;
}

// 路径：board 中小于 row 的那些行都已经成功放置了皇后
// 选择列表：第 row 行的所有列都是放置皇后的选择
// 结束条件：row 超过 board 的最后一行
void backtrack(vector<string>& board, int row) {
    // 触发结束条件
    if (row == board.size()) {
        res.push_back(board);
        return;
    }

    int n = board[row].size();
    for (int col = 0; col < n; col++) {
        // 排除不合法选择
        if (!isValid(board, row, col))
            continue;
        // 做选择
        board[row][col] = 'Q';
        // 进入下一行决策
        backtrack(board, row + 1);
        // 撤销选择
        board[row][col] = '.';
```

```
    }
}
```

这部分主要代码，其实跟全排列问题差不多，`isValid` 函数的实现也很简单：

```
/* 是否可以在 board[row][col] 放置皇后？ */
bool isValid(vector<string>& board, int row, int col) {
    int n = board.size();
    // 检查列是否有皇后互相冲突
    for (int i = 0; i < n; i++) {
        if (board[i][col] == 'Q')
            return false;
    }
    // 检查右上方是否有皇后互相冲突
    for (int i = row - 1, j = col + 1;
            i >= 0 && j < n; i--, j++) {
        if (board[i][j] == 'Q')
            return false;
    }
    // 检查左上方是否有皇后互相冲突
    for (int i = row - 1, j = col - 1;
            i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 'Q')
            return false;
    }
    return true;
}
```

函数 `backtrack` 依然像个在决策树上游走的指针，通过 `row` 和 `col` 就可以表示函数遍历到的位置，通过 `isValid` 函数可以将不符合条件的情况剪枝：



如果直接给你这么一大段解法代码，可能是懵逼的。但是现在明白了回溯算法的框架套路，还有啥难理解的呢？无非是改改做选择的方式，排除不合法选择的方式而已，只要框架存于心，你面对的只剩下小问题了。

当 `N = 8` 时，就是八皇后问题，数学大佬高斯穷尽一生都没有数清楚八皇后问题到底有几种可能的放置方法，但是我们的算法只需要一秒就可以算出来所有可能的结果。

不过真的不怪高斯。这个问题的复杂度确实非常高，看看我们的决策树，虽然有 `isValid` 函数剪枝，但是最坏时间复杂度仍然是 O(N^(N+1))，而且无法优化。如果 `N = 10` 的时候，计算就已经很耗时了。

有的时候，我们并不想得到所有合法的答案，只想要一个答案，怎么办呢？比如解数独的算法，找所有解法复杂度太高，只要找到一种解法就可以。

其实特别简单，只要稍微修改一下回溯算法的代码即可：

```
// 函数找到一个答案后就返回 true
bool backtrack(vector<string>& board, int row) {
    // 触发结束条件
    if (row == board.size()) {
        res.push_back(board);
        return true;
```

```
    }
    ...
    for (int col = 0; col < n; col++) {
        ...
        board[row][col] = 'Q';

        if (backtrack(board, row + 1))
            return true;

        board[row][col] = '.';
    }

    return false;
}
```

这样修改后，只要找到一个答案，for 循环的后续递归穷举都会被阻断。也许你可以在 N 皇后问题的代码框架上，稍加修改，写一个解数独的算法？

三、最后总结

回溯算法就是个多叉树的遍历问题，关键就是在前序遍历和后序遍历的位置做一些操作，算法框架如下：

```
def backtrack(...):
    for 选择 in 选择列表:
        做选择
        backtrack(...)
        撤销选择
```

写 `backtrack` 函数时，需要维护走过的「路径」和当前可以做的「选择列表」，当触发「结束条件」时，将「路径」记入结果集。

其实想想看，回溯算法和动态规划是不是有点像呢？我们在动态规划系列文章中多次强调，动态规划的三个需要明确的点就是「状态」「选择」和「base case」，是不是就对应着走过的「路径」，当前的「选择列表」和「结束条件」？

某种程度上说，动态规划的暴力求解阶段就是回溯算法。只是有的问题具有重叠子问题性质，可以用 dp table 或者备忘录优化，将递归树大幅剪枝，这就变成了动态规划。而今天的两个问题，都没有重叠子问题，也就是回溯算法问题了，复杂度非常高是不可避免的。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

二分查找详解

先给大家讲个笑话乐呵一下：

有一天阿东到图书馆借了 N 本书，出图书馆的时候，警报响了，于是保安把阿东拦下，要检查一下哪本书没有登记出借。阿东正准备把每一本书在报警器下过一下，以找出引发警报的书，但是保安露出不屑的眼神：你连二分查找都不会吗？于是保安把书分成两堆，让第一堆过一下报警器，报警器响；于是再把这堆书分成两堆……. 最终，检测了 logN 次之后，保安成功的找到了那本引起警报的书，露出了得意和嘲讽的笑容。于是阿东背着剩下的书走了。

从此，图书馆丢了 N - 1 本书。

二分查找真的很简单吗？并不简单。看看 Knuth 大佬（发明 KMP 算法的那位）怎么说的：

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky..、

这句话可以这样理解：**思路很简单，细节是魔鬼**。

本文就来探究几个最常用的二分查找场景：寻找一个数、寻找左侧边界、寻找右侧边界。而且，我们就是要深入细节，比如不等号是否应该带等号，mid 是否应该加一等等。分析这些细节的差异以及出现这些差异的原因，保证你能灵活准确地写出正确的二分查找算法。

零、二分查找框架

```
int binarySearch(int[] nums, int target) {
    int left = 0, right = ...;

    while(...) {
        int mid = (right + left) / 2;
```

```
        if (nums[mid] == target) {
            ...
        } else if (nums[mid] < target) {
            left = ...
        } else if (nums[mid] > target) {
            right = ...
        }
    }
    return ...;
}
```

分析二分查找的一个技巧是：不要出现 else，而是把所有情况用 else if 写清楚，这样可以清楚地展现所有细节。本文都会使用 else if，旨在讲清楚，读者理解后可自行简化。

其中 `...` 标记的部分，就是可能出现细节问题的地方，当你见到一个二分查找的代码时，首先注意这几个地方。后文用实例分析这些地方能有什么样的变化。

另外声明一下，计算 mid 时需要技巧防止溢出，可以「参见前文」，本文暂时忽略这个问题。

一、寻找一个数（基本的二分搜索）

这个场景是最简单的，肯能也是大家最熟悉的，即搜索一个数，如果存在，返回其索引，否则返回 -1。

```
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1; // 注意

    while(left <= right) {
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1; // 注意
        else if (nums[mid] > target)
            right = mid - 1; // 注意
```

```
        }
    return -1;
}
```

1、为什么 while 循环的条件中是 <=，而不是 < ？

答：因为初始化 right 的赋值是 nums.length - 1，即最后一个元素的索引，而不是 nums.length。

这二者可能出现在不同功能的二分查找中，区别是：前者相当于两端都闭区间 [left, right]，后者相当于左闭右开区间 [left, right)，因为索引大小为 nums.length 是越界的。

我们这个算法中使用的是前者 [left, right] 两端都闭的区间。**这个区间其实就是每次进行搜索的区间**。

什么时候应该停止搜索呢？当然，找到了目标值的时候可以终止：

```
    if(nums[mid] == target)
        return mid;
```

但如果没找到，就需要 while 循环终止，然后返回 -1。那 while 循环什么时候应该终止？**搜索区间为空的时候应该终止**，意味着你没得找了，就等于没找到嘛。

`while(left <= right)` 的终止条件是 left == right + 1，写成区间的形式就是 [right + 1, right]，或者带个具体的数字进去 [3, 2]，可见**这时候区间为空**，因为没有数字既大于等于 3 又小于等于 2 的吧。所以这时候 while 循环终止是正确的，直接返回 -1 即可。

`while(left < right)` 的终止条件是 left == right，写成区间的形式就是 [left, right]，或者带个具体的数字进去 [2, 2]，**这时候区间非空**，还有一个数 2，但此时 while 循环终止了。也就是说这区间 [2, 2] 被漏掉了，索引 2 没有被搜索，如果这时候直接返回 -1 就是错误的。

当然，如果你非要用 while(left < right) 也可以，我们已经知道了出错的原因，就打个补丁好了：

```
    //...
    while(left < right) {
        // ...
    }
    return nums[left] == target ? left : -1;
```

2、为什么 left = mid + 1，right = mid - 1？我看有的代码是 right = mid 或者 left = mid，没有这些加加减减，到底怎么回事，怎么判断？

答：这也是二分查找的一个难点，不过只要你能理解前面的内容，就能够很容易判断。

刚才明确了「搜索区间」这个概念，而且本算法的搜索区间是两端都闭的，即 [left, right]。那么当我们发现索引 mid 不是要找的 target 时，如何确定下一步的搜索区间呢？

当然是 [left, mid - 1] 或者 [mid + 1, right] 对不对？因为 mid 已经搜索过，应该从搜索区间中去除。

3、此算法有什么缺陷？

答：至此，你应该已经掌握了该算法的所有细节，以及这样处理的原因。但是，这个算法存在局限性。

比如说给你有序数组 nums = [1,2,2,2,3]，target = 2，此算法返回的索引是 2，没错。但是如果我想得到 target 的左侧边界，即索引 1，或者我想得到 target 的右侧边界，即索引 3，这样的话此算法是无法处理的。

这样的需求很常见。你也许会说，找到一个 target，然后向左或向右线性搜索不行吗？可以，但是不好，因为这样难以保证二分查找对数级的复杂度了。

我们后续的算法就来讨论这两种二分查找的算法。

二、寻找左侧边界的二分搜索

直接看代码，其中的标记是需要注意的细节：

```
int left_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0;
    int right = nums.length; // 注意

    while (left < right) { // 注意
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
            right = mid;
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid; // 注意
        }
    }
    return left;
}
```

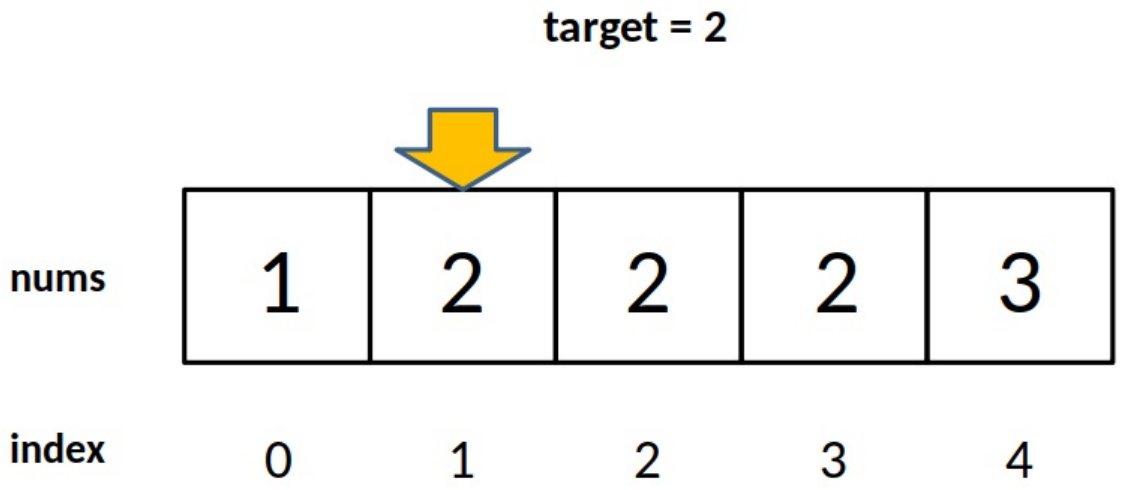
1、为什么 while(left < right) 而不是 <= ？

答：用相同的方法分析，因为 right = nums.length 而不是 nums.length - 1 。因此每次循环的「搜索区间」是 [left, right) 左闭右开。

while(left < right) 终止的条件是 left == right，此时搜索区间 [left, left) 为空，所以可以正确终止。

2、为什么没有返回 -1 的操作？如果 nums 中不存在 target 这个值，怎么办？

答：因为要一步一步来，先理解一下这个「左侧边界」有什么特殊含义：



对于这个数组，算法会返回 1。这个 1 的含义可以这样解读：nums 中小于 2 的元素有 1 个。

比如对于有序数组 nums = [2,3,5,7], target = 1，算法会返回 0，含义是：nums 中小于 1 的元素有 0 个。

再比如说 nums 不变，target = 8，算法会返回 4，含义是：nums 中小于 8 的元素有 4 个。

综上可以看出，函数的返回值（即 left 变量的值）取值区间是闭区间 [0, nums.length]，所以我们简单添加两行代码就能在正确的时候 return -1：

```
while (left < right) {
    //...
}
// target 比所有数都大
if (left == nums.length) return -1;
// 类似之前算法的处理方式
return nums[left] == target ? left : -1;
```

1、为什么 left = mid + 1，right = mid ？和之前的算法不一样？

答：这个很好解释，因为我们的「搜索区间」是 [left, right) 左闭右开，所以当 nums[mid] 被检测之后，下一步的搜索区间应该去掉 mid 分割成两个区间，即 [left, mid) 或 [mid + 1, right)。

4、为什么该算法能够搜索左侧边界？

答：关键在于对于 nums[mid] == target 这种情况的处理：

```
    if (nums[mid] == target)
        right = mid;
```

可见，找到 target 时不要立即返回，而是缩小「搜索区间」的上界 right，在区间 [left, mid) 中继续搜索，即不断向左收缩，达到锁定左侧边界的目的。

5、为什么返回 left 而不是 right？

答：都是一样的，因为 while 终止的条件是 left == right。

三、寻找右侧边界的二分查找

寻找右侧边界和寻找左侧边界的代码差不多，只有两处不同，已标注：

```
int right_bound(int[] nums, int target) {
    if (nums.length == 0) return -1;
    int left = 0, right = nums.length;

    while (left < right) {
        int mid = (left + right) / 2;
        if (nums[mid] == target) {
            left = mid + 1; // 注意
        } else if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid;
        }
    }
    return left - 1; // 注意
}
```

1、为什么这个算法能够找到右侧边界？

答：类似地，关键点还是这里：

```
        if (nums[mid] == target) {
            left = mid + 1;
```

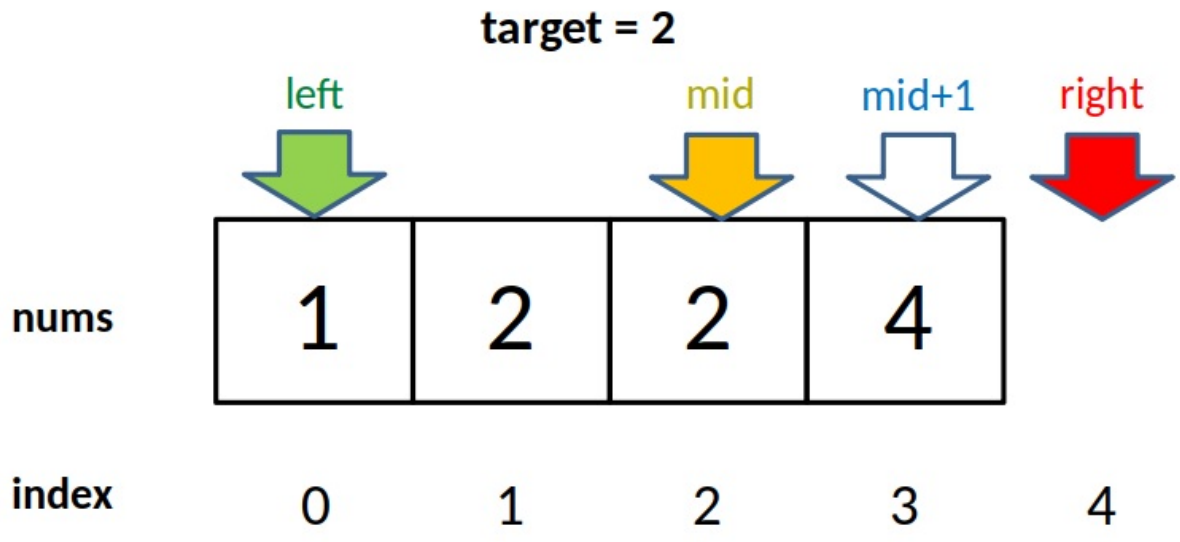
当 nums[mid] == target 时，不要立即返回，而是增大「搜索区间」的下界 left，使得区间不断向右收缩，达到锁定右侧边界的目的。

2、为什么最后返回 left - 1 而不像左侧边界的函数，返回 left？而且我觉得这里既然是搜索右侧边界，应该返回 right 才对。

答：首先，while 循环的终止条件是 left == right，所以 left 和 right 是一样的，你非要体现右侧的特点，返回 right - 1 好了。

至于为什么要减一，这是搜索右侧边界的一个特殊点，关键在这个条件判断：

```
        if (nums[mid] == target) {
            left = mid + 1;
            // 这样想: mid = left - 1
```



因为我们对 left 的更新必须是 left = mid + 1，就是说 while 循环结束时，nums[left] 一定不等于 target 了，而 nums[left-1] 可能是 target。

至于为什么 left 的更新必须是 left = mid + 1，同左侧边界搜索，就不再赘述。

1、为什么没有返回 -1 的操作？如果 nums 中不存在 target 这个值，怎么办？

答：类似之前的左侧边界搜索，因为 while 的终止条件是 left == right，就是说 left 的取值范围是 [0, nums.length]，所以可以添加两行代码，正确地返回 -1：

```
while (left < right) {
    // ...
}
if (left == 0) return -1;
return nums[left-1] == target ? (left-1) : -1;
```

四、最后总结

来梳理一下这些细节差异的因果逻辑：

第一个，最基本的二分查找算法：

```
因为我们初始化 right = nums.length - 1
所以决定了我们的「搜索区间」是 [left, right]
所以决定了 while (left <= right)
同时也决定了 left = mid+1 和 right = mid-1

因为我们只需找到一个 target 的索引即可
所以当 nums[mid] == target 时可以立即返回
```

第二个，寻找左侧边界的二分查找：

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid
```

```
因为我们需找到 target 的最左侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧右侧边界以锁定左侧边界
```

第三个，寻找右侧边界的二分查找：

```
因为我们初始化 right = nums.length
所以决定了我们的「搜索区间」是 [left, right)
所以决定了 while (left < right)
同时也决定了 left = mid + 1 和 right = mid

因为我们需找到 target 的最右侧索引
所以当 nums[mid] == target 时不要立即返回
而要收紧左侧边界以锁定右侧边界

又因为收紧左侧边界时必须 left = mid + 1
所以最后无论返回 left 还是 right，必须减一
```

如果以上内容你都能理解，那么恭喜你，二分查找算法的细节不过如此。

通过本文，你学会了：

1、分析二分查找代码时，不要出现 else，全部展开成 else if 方便理解。

2、注意「搜索区间」和 while 的终止条件，如果存在漏掉的元素，记得在最后检查。

3、如需要搜索左右边界，只要在 nums[mid] == target 时做修改即可。搜索右侧时需要减一。

呵呵，此文对二分查找的问题无敌好吧！**致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章**：


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

双指针技巧总结

我把双指针技巧再分为两类，一类是「快慢指针」，一类是「左右指针」。前者解决主要解决链表中的问题，比如典型的判定链表中是否包含环；后者主要解决数组（或者字符串）中的问题，比如二分查找。

一、快慢指针的常见算法

快慢指针一般都初始化指向链表的头结点 head，前进时快指针 fast 在前，慢指针 slow 在后，巧妙解决一些链表中的问题。

1、判定链表中是否含有环

这应该属于链表最基本的操作了，如果读者已经知道这个技巧，可以跳过。

单链表的特点是每个节点只知道下一个节点，所以一个指针的话无法判断链表中是否含有环的。

如果链表中不含环，那么这个指针最终会遇到空指针 null 表示链表到头了，这还好说，可以判断该链表不含环。

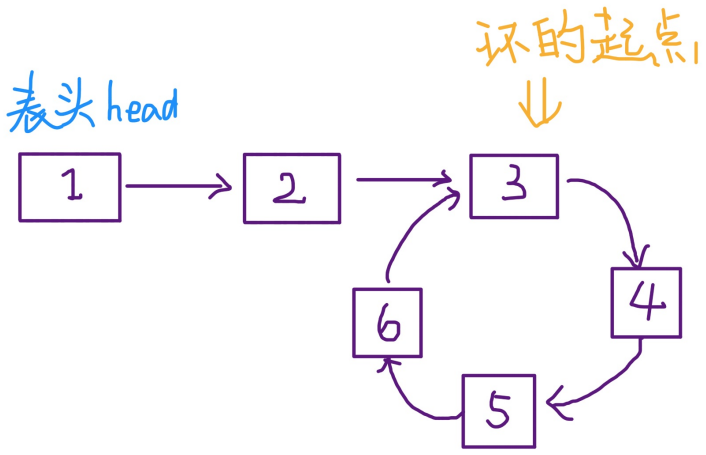
```
boolean hasCycle(ListNode head) {
    while (head != null)
        head = head.next;
    return false;
}
```

但是如果链表中含有环，那么这个指针就会陷入死循环，因为环形数组中没有 null 指针作为尾部节点。

经典解法就是用两个指针，一个跑得快，一个跑得慢。如果不含有环，跑得快的那个指针最终会遇到 null，说明链表不含环；如果含有环，快指针最终会超慢指针一圈，和慢指针相遇，说明链表含有环。

```
boolean hasCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;

        if (fast == slow) return true;
    }
    return false;
}
```

2、已知链表中含有环，返回这个环的起始位置



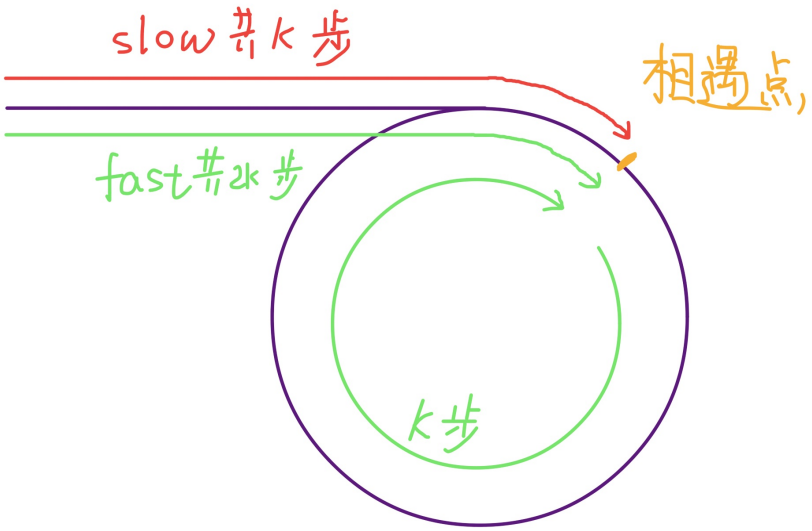
这个问题一点都不困难，有点类似脑筋急转弯，先直接看代码：

```
ListNode detectCycle(ListNode head) {
    ListNode fast, slow;
    fast = slow = head;
    while (fast != null && fast.next != null) {
        fast = fast.next.next;
        slow = slow.next;
        if (fast == slow) break;
    }
    // 上面的代码类似 hasCycle 函数
    slow = head;
    while (slow != fast) {
        fast = fast.next;
        slow = slow.next;
```

```
        }
        return slow;
    }
```

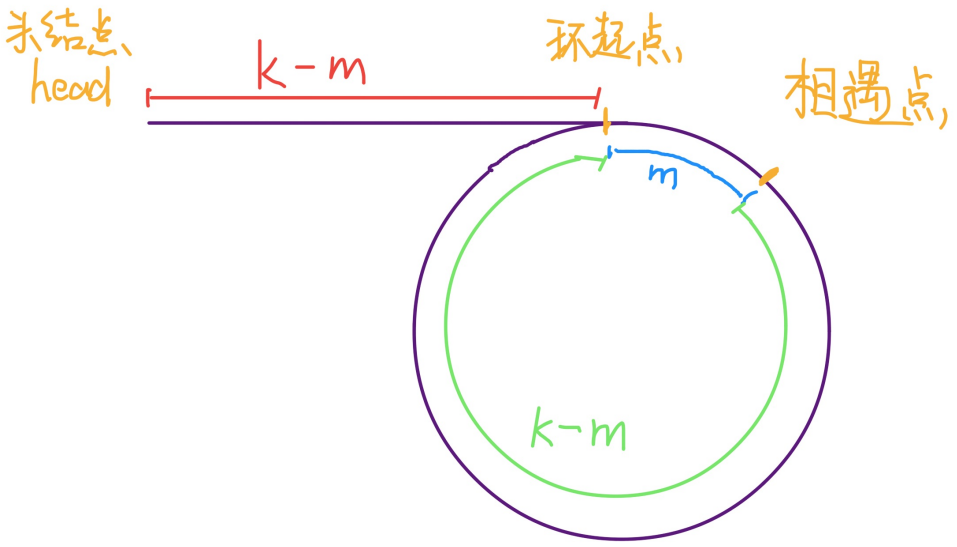
可以看到，当快慢指针相遇时，让其中任一个指针指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置。这是为什么呢？

第一次相遇时，假设慢指针 slow 走了 k 步，那么快指针 fast 一定走了 2k 步，也就是说比 slow 多走了 k 步（也就是环的长度）。



设相遇点距环的起点的距离为 m，那么环的起点距头结点 head 的距离为 k - m，也就是说如果从 head 前进 k - m 步就能到达环起点。

巧的是，如果从相遇点继续前进 k - m 步，也恰好到达环起点。



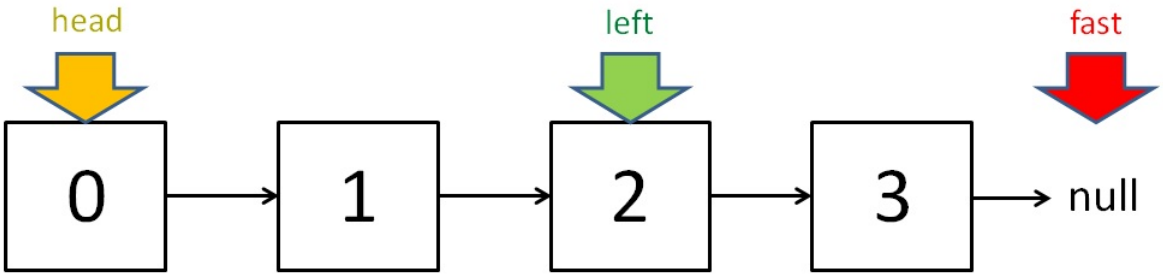
所以，只要我们把快慢指针中的任一个重新指向 head，然后两个指针同速前进，k - m 步后就会相遇，相遇之处就是环的起点了。

3、寻找链表的中点

类似上面的思路，我们还可以让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置。

```
while (fast != null && fast.next != null) {
    fast = fast.next.next;
    slow = slow.next;
}
// slow 就在中间位置
return slow;
```

当链表的长度是奇数时，slow 恰巧停在中点位置；如果长度是偶数，slow 最终的位置是中间偏右：



寻找链表中点的一个重要作用是对链表进行归并排序。

回想数组的归并排序：求中点索引递归地把数组二分，最后合并两个有序数组。对于链表，合并两个有序链表是很简单的，难点就在于二分。

但是现在你学会了找到链表的中点，就能实现链表的二分了。关于归并排序的具体内容本文就不具体展开了。

4、寻找链表的倒数第 k 个元素

我们的思路还是使用快慢指针，让快指针先走 k 步，然后快慢指针开始同速前进。这样当快指针走到链表末尾 null 时，慢指针所在的位置就是倒数第 k 个链表节点（为了简化，假设 k 不会超过链表长度）：

```
ListNode slow, fast;
slow = fast = head;
while (k-- > 0)
    fast = fast.next;

while (fast != null) {
    slow = slow.next;
    fast = fast.next;
}
return slow;
```

二、左右指针的常用算法

左右指针在数组中实际是指两个索引值，一般初始化为 left = 0, right = nums.length - 1 。

1、二分查找

前文「二分查找」有详细讲解，这里只写最简单的二分算法，旨在突出它的双指针特性：

```
int binarySearch(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
```

```
    while(left <= right) {
        int mid = (right + left) / 2;
        if(nums[mid] == target)
            return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid - 1;
    }
    return -1;
}
```

2、两数之和

直接看一道 LeetCode 题目吧：

给定一个已按照***升序排列***的有序数组，找到两个数使得它们相加之和等于目标数。

函数应该返回这两个下标值 index1 和 index2，其中 index1 必须小于 index2。

说明:

- 返回的下标值（index1 和 index2）不是从零开始的。
- 你可以假设每个输入只对应唯一的答案，而且你不可以重复使用相同的元素。

示例:

```
输入: numbers = [2, 7, 11, 15], target = 9
输出: [1,2]
解释: 2 与 7 之和等于目标数 9 。因此 index1 = 1, index2 = 2
```

只要数组有序，就应该想到双指针技巧。这道题的解法有点类似二分查找，通过调节 left 和 right 可以调整 sum 的大小：

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            // 题目要求的索引是从 1 开始的
```

```
            return new int[]{left + 1, right + 1};
        } else if (sum < target) {
            left++; // 让 sum 大一点
        } else if (sum > target) {
            right--; // 让 sum 小一点
        }
    }
    return new int[]{-1, -1};
}
```

3、反转数组

```
void reverse(int[] nums) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        // swap(nums[left], nums[right])
        int temp = nums[left];
        nums[left] = nums[right];
        nums[right] = temp;
        left++; right--;
    }
}
```

4、滑动窗口算法

这也许是双指针技巧的最高境界了，如果掌握了此算法，可以解决一大类子字符串匹配的问题，不过「滑动窗口」稍微比上述的这些算法复杂些。

幸运的是，这类算法是有框架模板的，而且这篇文章就讲解了「滑动窗口」算法模板，帮大家秒杀几道 LeetCode 子串匹配的问题。

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

滑动窗口技巧

本文详解「滑动窗口」这种高级双指针技巧的算法框架，带你秒杀几道高难度的子字符串匹配问题。

LeetCode 上至少有 9 道题目可以用此方法高效解决。但是有几道是 VIP 题目，有几道题目虽不难但太复杂，所以本文只选择点赞最高，较为经典的，最能够讲明白的三道题来讲解。第一题为了让读者掌握算法模板，篇幅相对长，后两题就基本秒杀了。

本文代码为 C++ 实现，不会用到什么编程方面的奇技淫巧，但是还是简单介绍一下一些用到的数据结构，以免有的读者因为语言的细节问题阻碍对算法思想的理解：

`unordered_map` 就是哈希表（字典），它的一个方法 count(key) 相当于 containsKey(key) 可以判断键 key 是否存在。

可以使用方括号访问键对应的值 map[key]。需要注意的是，如果该 key 不存在，C++ 会自动创建这个 key，并把 map[key] 赋值为 0。

所以代码中多次出现的 `map[key]++` 相当于 Java 的 `map.put(key, map.getOrDefault(key, 0) + 1)`。

本文大部分代码都是图片形式，可以点开放大，更重要的是可以左右滑动方便对比代码。下面进入正题。

一、最小覆盖子串

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。

示例：

```
输入：S = "ADOBECODEBANC", T = "ABC"
输出："BANC"
```

说明：

- 如果 S 中不存这样的子串，则返回空字符串 `""`。
- 如果 S 中存在这样的子串，我们保证它是唯一的答案。

题目不难理解，就是说要在 S(source) 中找到包含 T(target) 中全部字母的一个子串，顺序无所谓，但这个子串一定是所有可能子串中最短的。

如果我们使用暴力解法，代码大概是这样的：

```
for (int i = 0; i < s.size(); i++)
    for (int j = i + 1; j < s.size(); j++)
        if s[i:j] 包含 t 的所有字母:
            更新答案
```

思路很直接吧，但是显然，这个算法的复杂度肯定大于 O(N^2) 了，不好。

滑动窗口算法的思路是这样：

1、我们在字符串 S 中使用双指针中的左右指针技巧，初始化 left = right = 0，把索引闭区间 [left, right] 称为一个「窗口」。

2、我们先不断地增加 right 指针扩大窗口 [left, right]，直到窗口中的字符串符合要求（包含了 T 中的所有字符）。

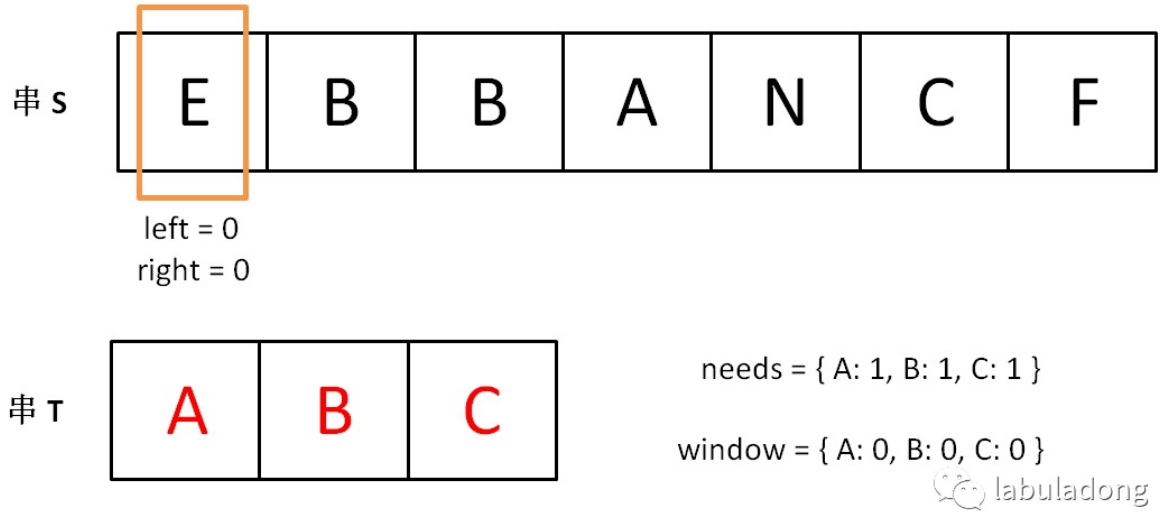
3、此时，我们停止增加 right，转而不断增加 left 指针缩小窗口 [left, right]，直到窗口中的字符串不再符合要求（不包含 T 中的所有字符了）。同时，每次增加 left，我们都要更新一轮结果。

4、重复第 2 和第 3 步，直到 right 到达字符串 S 的尽头。

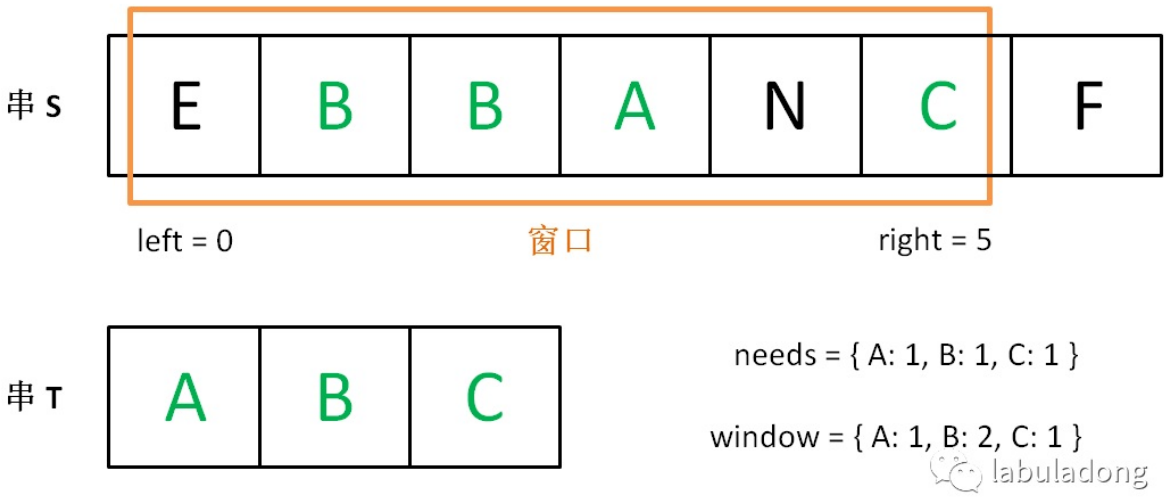
这个思路其实也不难，**第 2 步相当于在寻找一个「可行解」，然后第 3 步在优化这个「可行解」，最终找到最优解**。左右指针轮流前进，窗口大小增增减减，窗口不断向右滑动。

下面画图理解一下，needs 和 window 相当于计数器，分别记录 T 中字符出现次数和窗口中的相应字符的出现次数。

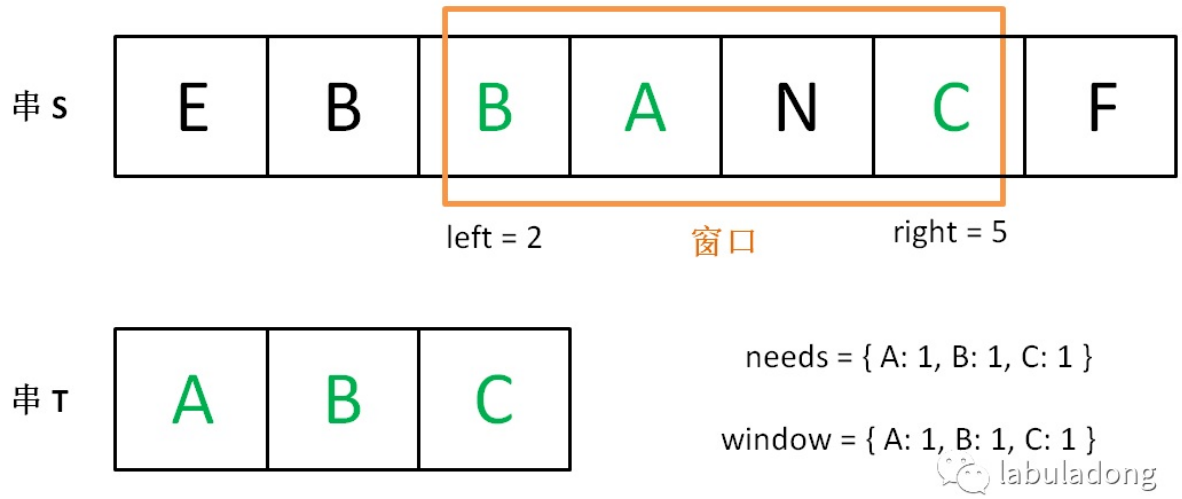
初始状态：



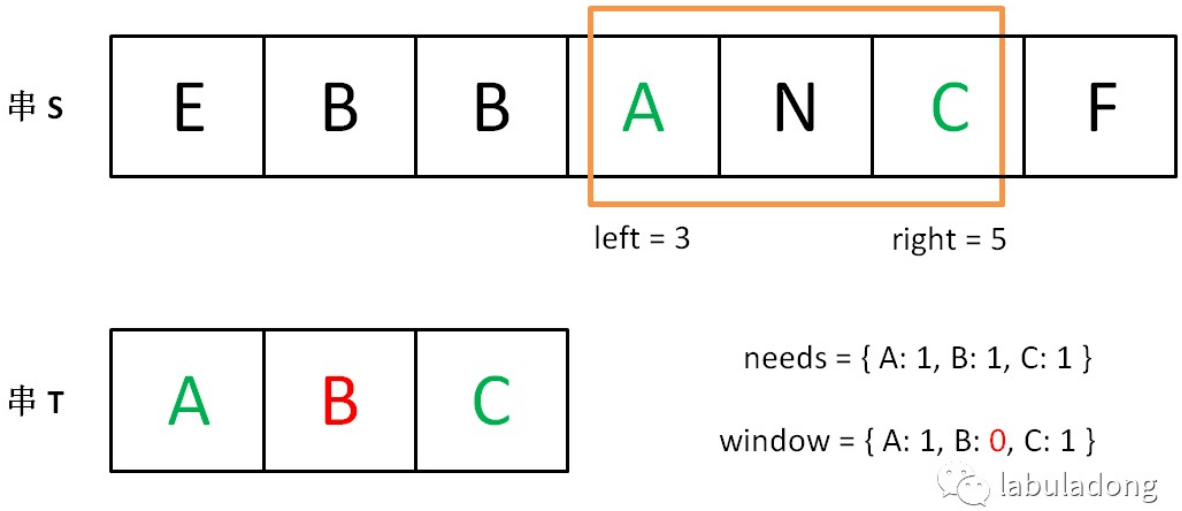
增加 right，直到窗口 [left, right] 包含了 T 中所有字符：



现在开始增加 left，缩小窗口 [left, right]。



直到窗口中的字符串不再符合要求，left 不再继续移动。



之后重复上述过程，先移动 right，再移动 left…… 直到 right 指针到达字符串 S 的末端，算法结束。

如果你能够理解上述过程，恭喜，你已经完全掌握了滑动窗口算法思想。至于如何具体到问题，如何得出此题的答案，都是编程问题，等会提供一套模板，理解一下就会了。

上述过程可以简单地写出如下伪码框架：

```
string s, t;
// 在 s 中寻找 t 的「最小覆盖子串」
int left = 0, right = 0;
```

```
string res = s;

while(right < s.size()) {
    window.add(s[right]);
    right++;
    // 如果符合要求，移动 left 缩小窗口
    while (window 符合要求) {
        // 如果这个窗口的子串更短，则更新 res
        res = minLen(res, window);
        window.remove(s[left]);
        left++;
    }
}
return res;
```

如果上述代码你也能够理解，那么你离解题更近了一步。现在就剩下一个比较棘手的问题：如何判断 window 即子串 s[left...right] 是否符合要求，是否包含 t 的所有字符呢？

可以用两个哈希表当作计数器解决。用一个哈希表 needs 记录字符串 t 中包含的字符及出现次数，用另一个哈希表 window 记录当前「窗口」中包含的字符及出现的次数，如果 window 包含所有 needs 中的键，且这些键对应的值都大于等于 needs 中的值，那么就可以知道当前「窗口」符合要求了，可以开始移动 left 指针了。

现在将上面的框架继续细化：

```
string s, t;
// 在 s 中寻找 t 的「最小覆盖子串」
int left = 0, right = 0;
string res = s;

// 相当于两个计数器
unordered_map<char, int> window;
unordered_map<char, int> needs;
for (char c : t) needs[c]++;

// 记录 window 中已经有多少字符符合要求了
int match = 0;
```

```
while (right < s.size()) {
    char c1 = s[right];
    if (needs.count(c1)) {
        window[c1]++; // 加入 window
        if (window[c1] == needs[c1])
            // 字符 c1 的出现次数符合要求了
            match++;
    }
    right++;

    // window 中的字符串已符合 needs 的要求了
    while (match == needs.size()) {
        // 更新结果 res
        res = minLen(res, window);
        char c2 = s[left];
        if (needs.count(c2)) {
            window[c2]--; // 移出 window
            if (window[c2] < needs[c2])
                // 字符 c2 出现次数不再符合要求
                match--;
        }
        left++;
    }
}
return res;
```

上述代码已经具备完整的逻辑了，只有一处伪码，即更新 res 的地方，不过这个问题太好解决了，直接看解法吧！

```
string minWindow(string s, string t) {
    // 记录最短子串的开始位置和长度
    int start = 0, minLen = INT_MAX;
    int left = 0, right = 0;

    unordered_map<char, int> window;
    unordered_map<char, int> needs;
    for (char c : t) needs[c]++;

    int match = 0;

    while (right < s.size()) {
        char c1 = s[right];
```

```
        if (needs.count(c1)) {
            window[c1]++;
            if (window[c1] == needs[c1])
                match++;
        }
        right++;

        while (match == needs.size()) {
            if (right - left < minLen) {
                // 更新最小子串的位置和长度
                start = left;
                minLen = right - left;
            }
            char c2 = s[left];
            if (needs.count(c2)) {
                window[c2]--;
                if (window[c2] < needs[c2])
                    match--;
            }
            left++;
        }
    }
    return minLen == INT_MAX ?
                "" : s.substr(start, minLen);
}
```

如果直接甩给你这么一大段代码，我想你的心态是爆炸的，但是通过之前的步步跟进，你是否能够理解这个算法的内在逻辑呢？你是否能清晰看出该算法的结构呢？

这个算法的时间复杂度是 O(M + N)，M 和 N 分别是字符串 S 和 T 的长度。因为我们先用 for 循环遍历了字符串 T 来初始化 needs，时间 O(N)，之后的两个 while 循环最多执行 2M 次，时间 O(M)。

读者也许认为嵌套的 while 循环复杂度应该是平方级，但是你这样想，while 执行的次数就是双指针 left 和 right 走的总路程，最多是 2M 嘛。

二、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:

输入:
`s: "cbaebabacd" p: "abc"`

输出:
`[0, 6]`

解释:
起始索引等于 0 的子串是 "cba", 它是 "abc" 的字母异位词。
起始索引等于 6 的子串是 "bac", 它是 "abc" 的字母异位词。

这道题的难度是 Easy，但是评论区点赞最多的一条是这样：

```
How can this problem be marked as easy???
```

实际上，这个 Easy 是属于了解双指针技巧的人的，只要把上一道题的代码改中更新 res 部分的代码稍加修改就成了这道题的解：

```
vector<int> findAnagrams(string s, string t) {
    // 用数组记录答案
    vector<int> res;
    int left = 0, right = 0;
    unordered_map<char, int> needs;
    unordered_map<char, int> window;
    for (char c : t) needs[c]++;
    int match = 0;

    while (right < s.size()) {
        char c1 = s[right];
        if (needs.count(c1)) {
            window[c1]++;
            if (window[c1] == needs[c1])
                match++;
```

```
        }
        right++;

        while (match == needs.size()) {
            // 如果 window 的大小合适
            // 就把起始索引 left 加入结果
            if (right - left == t.size()) {
                res.push_back(left);
            }
            char c2 = s[left];
            if (needs.count(c2)) {
                window[c2]--;
                if (window[c2] < needs[c2])
                    match--;
            }
            left++;
        }
    }
    return res;
}
```

因为这道题和上一道的场景类似，也需要 window 中包含串 t 的所有字符，但上一道题要找长度最短的子串，这道题要找长度相同的子串，也就是「字母异位词」嘛。

三、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：** "bbbbb"
> **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。

遇到子串问题，首先想到的就是滑动窗口技巧。

类似之前的思路，使用 window 作为计数器记录窗口中的字符出现次数，然后先向右移动 right，当 window 中出现重复字符时，开始移动 left 缩小窗口，如此往复：

```
int lengthOfLongestSubstring(string s) {
    int left = 0, right = 0;
    unordered_map<char, int> window;
    int res = 0; // 记录最长长度

    while (right < s.size()) {
        char c1 = s[right];
        window[c1]++;
        right++;
        // 如果 window 中出现重复字符
        // 开始移动 left 缩小窗口
        while (window[c1] > 1) {
            char c2 = s[left];
```

```
            window[c2]--;
            left++;
        }
        res = max(res, right - left);
    }
    return res;
}
```

需要注意的是，因为我们要求的是最长子串，所以需要在每次移动 right 增大窗口时更新 res，而不是像之前的题目在移动 left 缩小窗口时更新 res。

最后总结

通过上面三道题，我们可以总结出滑动窗口算法的抽象思想：

```
int left = 0, right = 0;

while (right < s.size()) {
    window.add(s[right]);
    right++;

    while (valid) {
        window.remove(s[left]);
        left++;
    }
}
```

其中 window 的数据类型可以视具体情况而定，比如上述题目都使用哈希表充当计数器，当然你也可以用一个数组实现同样效果，因为我们只处理英文字母。

稍微麻烦的地方就是这个 valid 条件，为了实现这个条件的实时更新，我们可能会写很多代码。比如前两道题，看起来解法篇幅那么长，实际上思想还是很简单，只是大多数代码都在处理这个问题而已。

如果本文对你有帮助，欢迎关注我的公众号 labuladong，致力于把算法问题讲清楚～


编程，算法，生活
致力于把问题讲清楚
扫码关注公众号：labuladong

twoSum问题的核心思想

Two Sum 系列问题在 LeetCode 上有好几道，这篇文章就挑出有代表性的几道，介绍一下这种问题怎么解决。

TwoSum I

这个问题的**最基本形式**是这样：给你一个数组和一个整数 `target`，可以保证数组中**存在**两个数的和为 `target`，请你返回这两个数的索引。

比如输入 `nums = [3,1,3,6], target = 6`，算法应该返回数组 `[0,2]`，因为 3 + 3 = 6。

这个问题如何解决呢？首先最简单粗暴的办法当然是穷举了：

```
int[] twoSum(int[] nums, int target) {

    for (int i = 0; i < nums.length; i++)
        for (int j = i + 1; j < nums.length; j++)
            if (nums[j] == target - nums[i])
                return new int[] { i, j };

    // 不存在这么两个数
    return new int[] {-1, -1};
}
```

这个解法非常直接，时间复杂度 O(N^2)，空间复杂度 O(1)。

可以通过一个哈希表减少时间复杂度：

```
int[] twoSum(int[] nums, int target) {
    int n = nums.length;
    index<Integer, Integer> index = new HashMap<>();
    // 构造一个哈希表：元素映射到相应的索引
    for (int i = 0; i < n; i++)
```

```
            index.put(nums[i], i);

        for (int i = 0; i < n; i++) {
            int other = target - nums[i];
            // 如果 other 存在且不是 nums[i] 本身
            if (index.containsKey(other) && index.get(other) != i)
                return new int[] {i, index.get(other)};
        }

        return new int[] {-1, -1};
    }
```

这样，由于哈希表的查询时间为 O(1)，算法的时间复杂度降低到 O(N)，但是需要 O(N) 的空间复杂度来存储哈希表。不过综合来看，是要比暴力解法高效的。

我觉得 Two Sum 系列问题就是想教我们如何使用哈希表处理问题。我们接着往后看。

TwoSum II

这里我们稍微修改一下上面的问题。我们设计一个类，拥有两个 API：

```
class TwoSum {
    // 向数据结构中添加一个数 number
    public void add(int number);
    // 寻找当前数据结构中是否存在两个数的和为 value
    public boolean find(int value);
}
```

如何实现这两个 API 呢，我们可以仿照上一道题目，使用一个哈希表辅助 `find` 方法：

```
class TwoSum {
    Map<Integer, Integer> freq = new HashMap<>();

    public void add(int number) {
        // 记录 number 出现的次数
```

```
            freq.put(number, freq.getOrDefault(number, 0) + 1);
        }

        public boolean find(int value) {
            for (Integer key : freq.keySet()) {
                int other = value - key;
                // 情况一
                if (other == key && freq.get(key) > 1)
                    return true;
                // 情况二
                if (other != key && freq.containsKey(other))
                    return true;
            }
            return false;
        }
    }
```

进行 `find` 的时候有两种情况，举个例子：

情况一：`add` 了 `[3,3,2,5]` 之后，执行 `find(6)`，由于 3 出现了两次，3 + 3 = 6，所以返回 true。

情况二：`add` 了 `[3,3,2,5]` 之后，执行 `find(7)`，那么 `key` 为 2，`other` 为 5 时算法可以返回 true。

除了上述两种情况外，`find` 只能返回 false 了。

对于这个解法的时间复杂度呢，`add` 方法是 O(1)，`find` 方法是 O(N)，空间复杂度为 O(N)，和上一道题目比较类似。

但是对于 API 的设计，是需要考虑现实情况的。比如说，我们设计的这个类，使用 `find` 方法非常频繁，那么每次都要 O(N) 的时间，岂不是很浪费费时间吗？对于这种情况，我们是否可以做些优化呢？

是的，对于频繁使用 `find` 方法的场景，我们可以进行优化。我们可以参考上一道题目的暴力解法，借助**哈希集合**来针对性优化 `find` 方法：

```
class TwoSum {
    Set<Integer> sum = new HashSet<>();
    List<Integer> nums = new ArrayList<>();
```

```
    public void add(int number) {
        // 记录所有可能组成的和
        for (int n : nums)
            sum.add(n + number);
        nums.add(number);
    }

    public boolean find(int value) {
        return sum.contains(value);
    }
}
```

这样 `sum` 中就储存了所有加入数字可能组成的和，每次 `find` 只要花费 O(1) 的时间在集合中判断一下是否存在就行了，显然非常适合频繁使用 `find` 的场景。

三、总结

对于 TwoSum 问题，一个难点就是给的数组**无序**。对于一个无序的数组，我们似乎什么技巧也没有，只能暴力穷举所有可能。

一般情况下，我们会首先把数组排序再考虑双指针技巧。TwoSum 启发我们，HashMap 或者 HashSet 也可以帮助我们处理无序数组相关的简单问题。

另外，设计的核心在于权衡，利用不同的数据结构，可以得到一些针对性的加强。

最后，如果 TwoSum I 中给的数组是有序的，应该如何编写算法呢？答案很简单，前文「双指针技巧汇总」写过：

```
int[] twoSum(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return new int[]{left, right};
        } else if (sum < target) {
```

```
            left++; // 让 sum 大一点
        } else if (sum > target) {
            right--; // 让 sum 小一点
        }
    }
    // 不存在这样两个数
    return new int[]{-1, -1};
}
```

常用的位操作

本文分两部分，第一部分列举几个有趣的位操作，第二部分讲解算法中常用的 n & (n - 1) 操作，顺便把用到这个技巧的算法题列出来讲解一下。因为位操作很简单，所以假设读者已经了解与、或、异或这三种基本操作。

位操作（Bit Manipulation）可以玩出很多奇技淫巧，但是这些技巧大部分都过于晦涩，没必要深究，读者只要记住一些有用的操作即可。

一、几个有趣的位操作

1. 利用或操作 `|` 和空格将英文字符转换为小写

```
('a' | ' ') = 'a'
('A' | ' ') = 'a'
```

1. 利用与操作 `&` 和下划线将英文字符转换为大写

```
('b' & '_') = 'B'
('B' & '_') = 'B'
```

1. 利用异或操作 `^` 和空格进行英文字符大小写互换

```
('d' ^ ' ') = 'D'
('D' ^ ' ') = 'd'
```

PS：以上操作能够产生奇特效果的原因在于 ASCII 编码。字符其实就是数字，恰巧这些字符对应的数字通过位运算就能得到正确的结果，有兴趣的读者可以查 ASCII 码表自己算算，本文就不展开讲了。

1. 判断两个数是否异号

```
int x = -1, y = 2;
bool f = ((x ^ y) < 0); // true

int x = 3, y = 2;
bool f = ((x ^ y) < 0); // false
```

PS：这个技巧还是很实用的，利用的是补码编码的符号位。如果不用位运算来判断是否异号，需要使用 if else 分支，还挺麻烦的。读者可能想利用乘积或者商来判断两个数是否异号，但是这种处理方式可能造成溢出，从而出现错误。（关于补码编码和溢出，参见前文）

1. 交换两个数

```
int a = 1, b = 2;
a ^= b;
b ^= a;
a ^= b;
// 现在 a = 2, b = 1
```

1. 加一

```
int n = 1;
n = -~n;
// 现在 n = 2
```

1. 减一

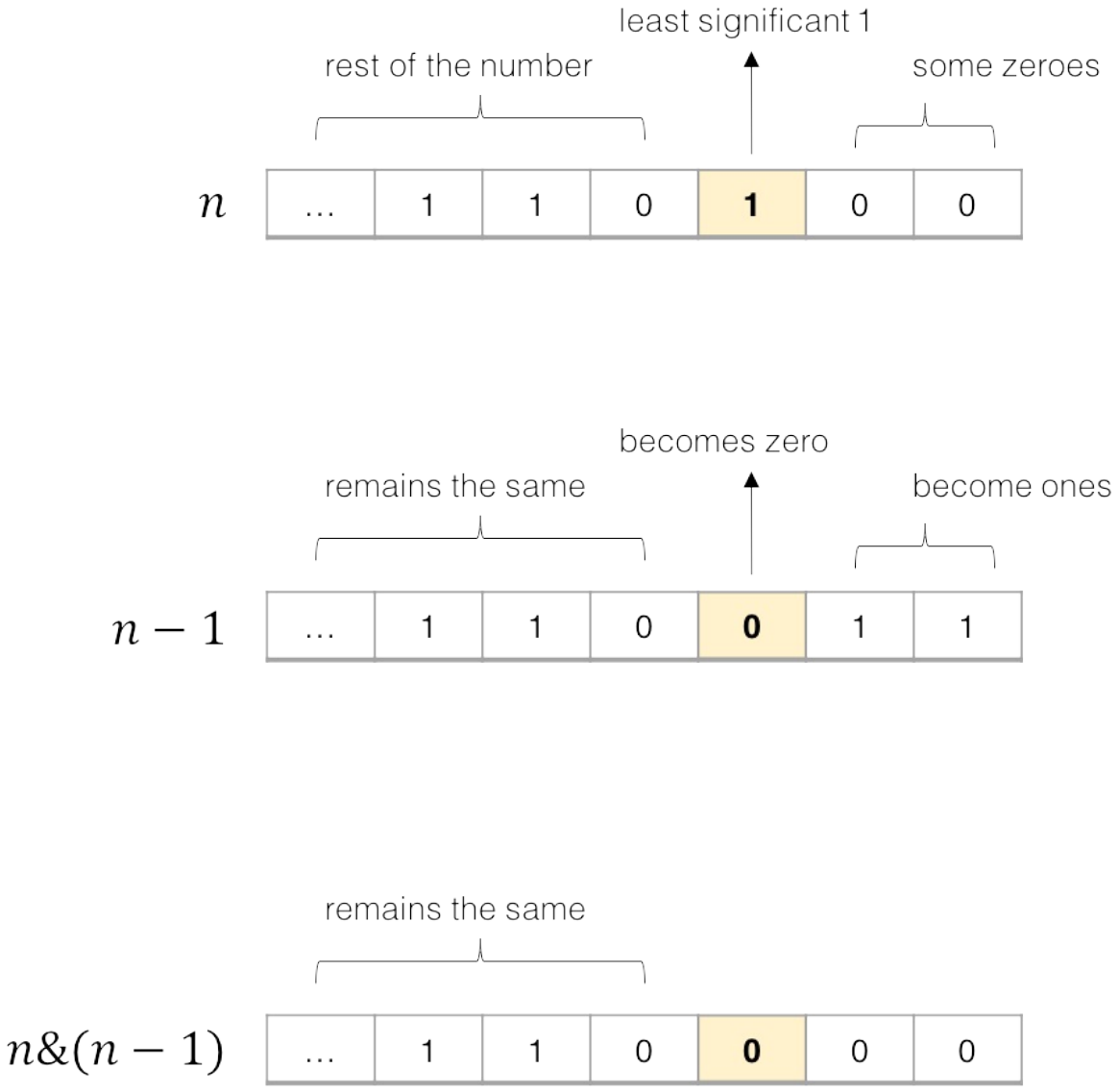
```
int n = 2;
n = ~-n;
// 现在 n = 1
```

PS：上面这三个操作就纯属装逼用的，没啥实际用处，大家了解了解乐呵一下就行。

二、算法常用操作 n&(n-1)

这个操作是算法中常见的，作用是消除数字 n 的二进制表示中的最后一个 1。

看个图就很容易理解了：



1. 计算汉明权重（Hamming Weight）

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：

输入： 00000000000000000000000000001011
输出： 3
解释： 输入的二进制串 **00000000000000000000000000001011** 中，共有三位为 '1'。

示例 2：

输入： 00000000000000000000000010000000
输出： 1
解释： 输入的二进制串 **00000000000000000000000010000000** 中，共有一位为 '1'。

就是让你返回 n 的二进制表示中有几个 1。因为 n & (n - 1) 可以消除最后一个 1，所以可以用一个循环不停地消除 1 同时计数，直到 n 变成 0 为止。

```
int hammingWeight(uint32_t n) {
    int res = 0;
    while (n != 0) {
        n = n & (n - 1);
        res++;
    }
    return res;
}
```

1. 判断一个数是不是 2 的指数

一个数如果是 2 的指数，那么它的二进制表示一定只含有一个 1：

```
2^0 = 1 = 0b0001
2^1 = 2 = 0b0010
2^2 = 4 = 0b0100
```

如果使用位运算技巧就很简单了（注意运算符优先级，括号不可以省略）：

```
bool isPowerOfTwo(int n) {
    if (n <= 0) return false;
    return (n & (n - 1)) == 0;
}
```

以上便是一些有趣/常用的位操作。其实位操作的技巧很多，有一个叫做 Bit Twiddling Hacks 的外国网站收集了几乎所有位操作的黑科技玩法，感兴趣的读者可以点击「阅读原文」按钮查看。

拆解复杂问题：实现计算器

我们最终要实现的计算器功能如下：

1、输入一个字符串，可以包含 `+ - * /` 、数字、括号以及空格，你的算法返回运算结构。

2、要符合运算法则，括号的优先级最高，先乘除后加减。

3、除号是整数除法，无论正负都向 0 取整（5/2=2，-5/2=-2）。

4、可以假定输入的算式一定合法，且计算过程不会出现整型溢出，不会出现除数为 0 的意外情况。

比如输入如下字符串，算法会返回 9：

```
3 * (2-6 /(3 -7))
```

可以看到，这就已经非常接近我们实际生活中使用的计算器了，虽然我们以前肯定都用过计算器，但是如果简单思考一下其算法实现，就会大惊失色：

1、按照常理处理括号，要先计算最内层的括号，然后向外慢慢化简。这个过程我们手算都容易出错，何况写成算法呢！

2、要做到先乘除，后加减，这一点教会小朋友还不算难，但教给计算机恐怕有点困难。

3、要处理空格。我们为了美观，习惯性在数字和运算符之间打个空格，但是计算之中得想办法忽略这些空格。

我记得很多大学数据结构的教材上，在讲栈这种数据结构的时候，应该都会用计算器举例，但是有一说一，讲的真的垃圾，不知道多少未来的计算机科学家就被这种简单的数据结构劝退了。

那么本文就来聊聊怎么实现上述一个功能完备的计算器功能，**关键在于层层拆解问题，化整为零，逐个击破**，相信这种思维方式能帮大家解决各种复杂问题。

下面就来拆解，从最简单的一个问题开始。

一、字符串转整数

是的，就是这么一个简单的问题，首先告诉我，怎么把一个字符串形式的**正**整数，转化成 int 型？

```
string s = "458";

int n = 0;
for (int i = 0; i < s.size(); i++) {
    char c = s[i];
    n = 10 * n + (c - '0');
}
// n 现在就等于 458
```

这个还是很简单的吧，老套路了。但是即便这么简单，依然有坑：**`(c - '0')` 的这个括号不能省略，否则可能造成整型溢出**。

因为变量 `c` 是一个 ASCII 码，如果不加括号就会先加后减，想象一下 `s` 如果接近 INT_MAX，就会溢出。所以用括号保证先减后加才行。

二、处理加减法

现在进一步，**如果输入的这个算式只包含加减法，而且不存在空格**，你怎么计算结果？我们拿字符串算式 `1-12+3` 为例，来说一个很简单的思路：

1、先给第一个数字加一个默认符号 `+`，变成 `+1-12+3`。

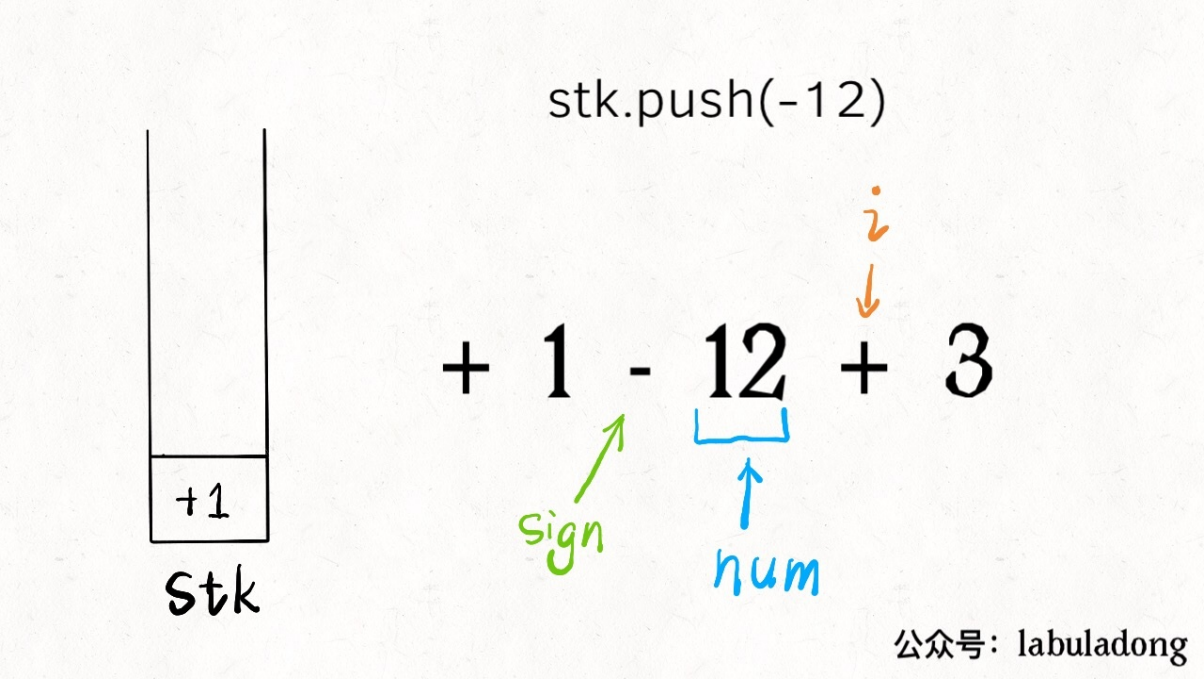
2、把一个运算符和数字组合成一对儿，也就是三对儿 `+1`，`-12`，`+3`，把它们转化成数字，然后放到一个栈中。

3、将栈中所有的数字求和，就是原算式的结果。

我们直接看代码，结合一张图就看明白了：

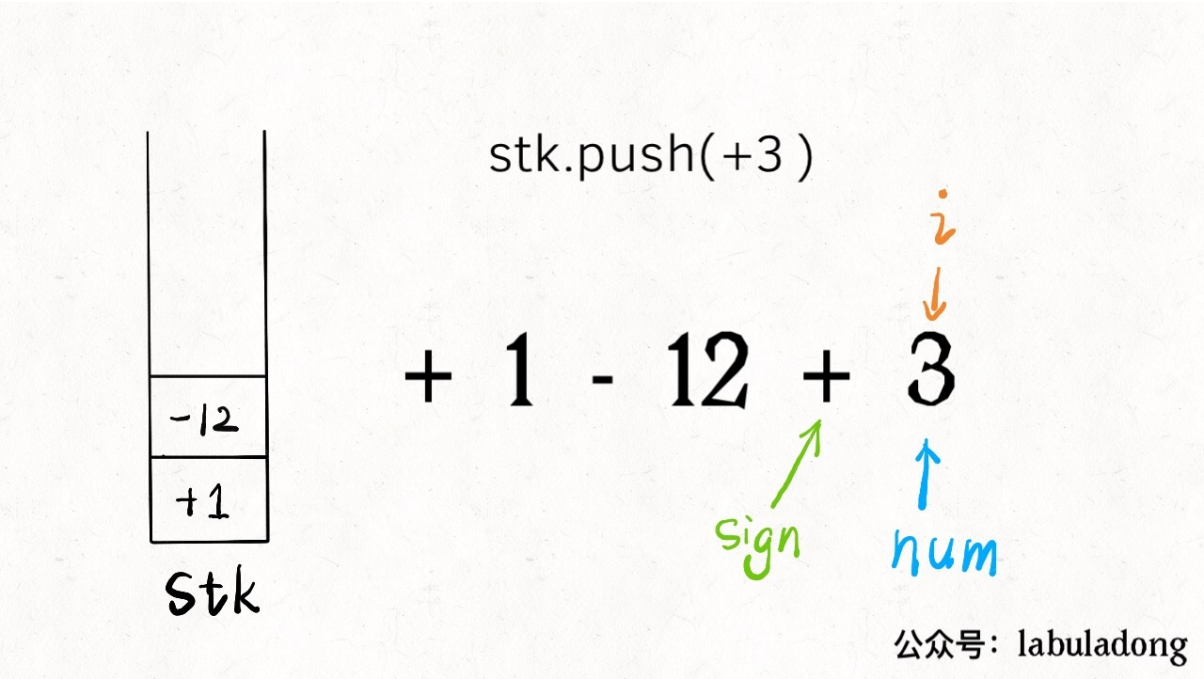
```
int calculate(string s) {
    stack<int> stk;
    // 记录算式中的数字
    int num = 0;
    // 记录 num 前的符号，初始化为 +
    char sign = '+';
    for (int i = 0; i < s.size(); i++) {
        char c = s[i];
        // 如果是数字，连续读取到 num
        if (isdigit(c))
            num = 10 * num + (c - '0');
        // 如果不是数字，就是遇到了下一个符号，
        // 之前的数字和符号就要存进栈中
        if (!isdigit(c) || i == s.size() - 1) {
            switch (sign) {
                case '+':
                    stk.push(num); break;
                case '-':
                    stk.push(-num); break;
            }
            // 更新符号为当前符号，数字清零
            sign = c;
            num = 0;
        }
    }
    // 将栈中所有结果求和就是答案
    int res = 0;
    while (!stk.empty()) {
        res += stk.top();
        stk.pop();
    }
    return res;
}
```

我估计就是中间带 `switch` 语句的部分有点不好理解吧，`i` 就是从左到右扫描，`sign` 和 `num` 跟在它身后。当 `s[i]` 遇到一个运算符时，情况是这样的：



所以说，此时要根据 `sign` 的 case 不同选择 `nums` 的正负号，存入栈中，然后更新 `sign` 并清零 `nums` 记录下一对儿符合和数字的组合。

另外注意，不只是遇到新的符号会触发入栈，当 `i` 走到了算式的尽头（ `i == s.size() - 1` ），也应该将前面的数字入栈，方便后续计算最终结果。



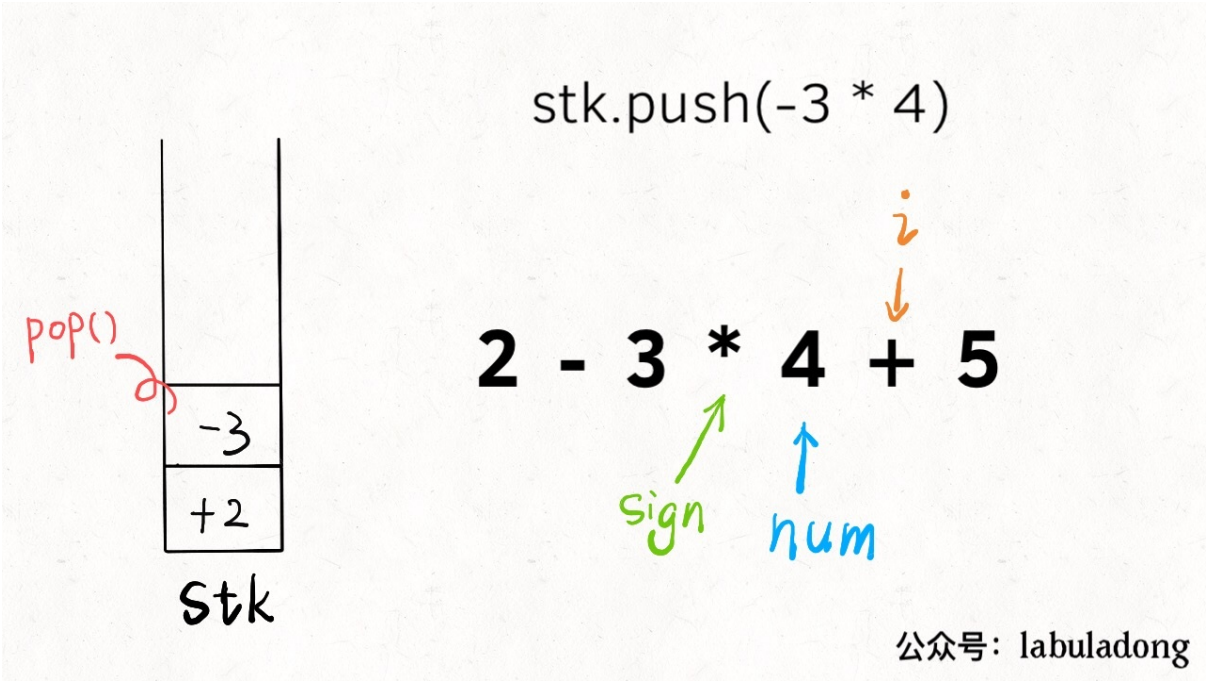
至此，仅处理紧凑加减法字符串的算法就完成了，请确保理解以上内容，后续的内容就基于这个框架修修改改就完事儿了。

三、处理乘除法

其实思路跟仅处理加减法没啥区别，拿字符串 `2-3*4+5` 举例，核心思路依然是把字符串分解成符号和数字的组合。

比如上述例子就可以分解为 `+2` ，`-3` ，`*4` ，`+5` 几对儿，我们刚才不是没有处理乘除号吗，很简单，**其他部分都不用变**，在 `switch` 部分加上对应的 case 就行了：

```
for (int i = 0; i < s.size(); i++) {
    char c = s[i];
    if (isdigit(c))
        num = 10 * num + (c - '0');

    if (!isdigit(c) || i == s.size() - 1) {
        switch (sign) {
            int pre;
            case '+':
                stk.push(num); break;
            case '-':
                stk.push(-num); break;
            // 只要拿出前一个数字做对应运算即可
            case '*':
                pre = stk.top();
                stk.pop();
                stk.push(pre * num);
                break;
            case '/':
                pre = stk.top();
                stk.pop();
                stk.push(pre / num);
                break;
        }
        // 更新符号为当前符号，数字清零
        sign = c;
        num = 0;
    }
}
```



乘除法优先于加减法体现在，乘除法可以和栈顶的数结合，而加减法只能把自己放入栈。

现在我们思考一下**如何处理字符串中可能出现的空格字符**。其实也非常简单，想想空格字符的出现，会影响我们现有代码的哪一部分？

```
// 如果 c 非数字
if (!isdigit(c) || i == s.size() - 1) {
    switch (c) {...}
    sign = c;
    num = 0;
}
```

显然空格会进入这个 if 语句，但是我们并不想让空格的情况进入这个 if，因为这里会更新 `sign` 并清零 `nums`，空格根本就不是运算符，应该被忽略。

那么只要多加一个条件即可：

```
if ((!isdigit(c) && c != ' ') || i == s.size() - 1) {
    ...
}
```

好了，现在我们的算法已经可以按照正确的法则计算加减乘除，并且自动忽略空格符，剩下的就是如何让算法正确识别括号了。

四、处理括号

处理算式中的括号看起来应该是最难的，但真没有看起来那么难。

为了规避编程语言的繁琐细节，我把前面解法的代码翻译成 Python 版本：

```
def calculate(s: str) -> int:

    def helper(s: List) -> int:
        stack = []
        sign = '+'
        num = 0

        while len(s) > 0:
            c = s.pop(0)
            if c.isdigit():
                num = 10 * num + int(c)

            if (not c.isdigit() and c != ' ') or len(s) == 0:
                if sign == '+':
                    stack.append(num)
                elif sign == '-':
                    stack.append(-num)
                elif sign == '*':
                    stack[-1] = stack[-1] * num
                elif sign == '/':
                    # python 除法向 0 取整的写法
                    stack[-1] = int(stack[-1] / float(num))

                num = 0
                sign = c

        return sum(stack)
    # 需要把字符串转成列表方便操作
    return helper(list(s))
```

这段代码跟刚才 C++ 代码完全相同，唯一的区别是，不是从左到右遍历字符串，而是不断从左边 `pop` 出字符，本质还是一样的。

那么，为什么说处理括号没有看起来那么难呢，**因为括号具有递归性质**。我们拿字符串 `3*(4-5/2)-6` 举例：

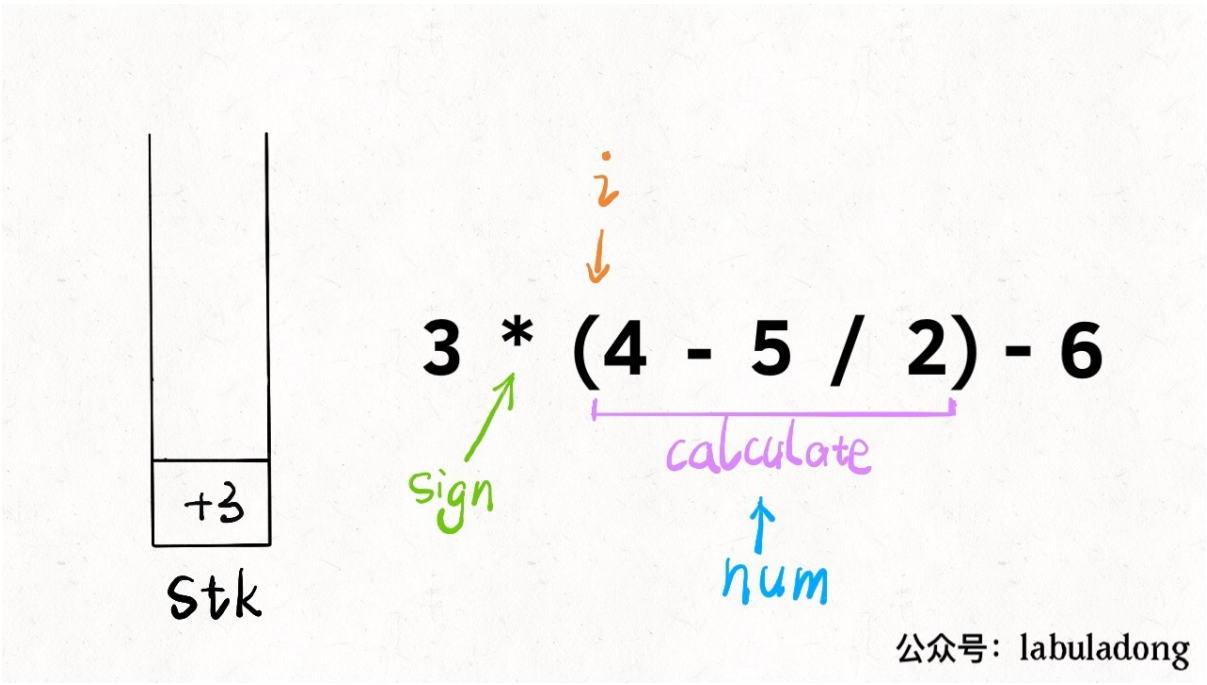
calculate(`3*(4-5/2)-6`) = 3 *calculate(`4-5/2`)* - *6* = *3* 2 - 6 = 0

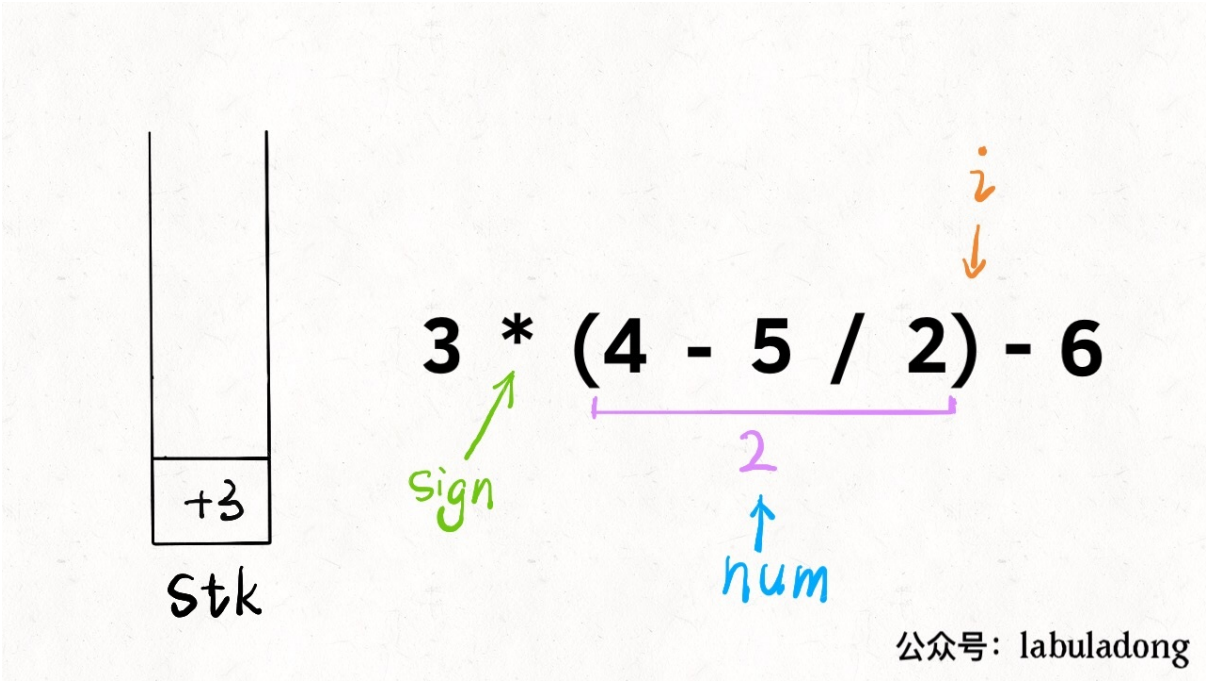
可以脑补一下，无论多少层括号嵌套，通过 calculate 函数递归调用自己，都可以将括号中的算式化简成一个数字。**换句话说，括号包含的算式，我们直接视为一个数字就行了**。

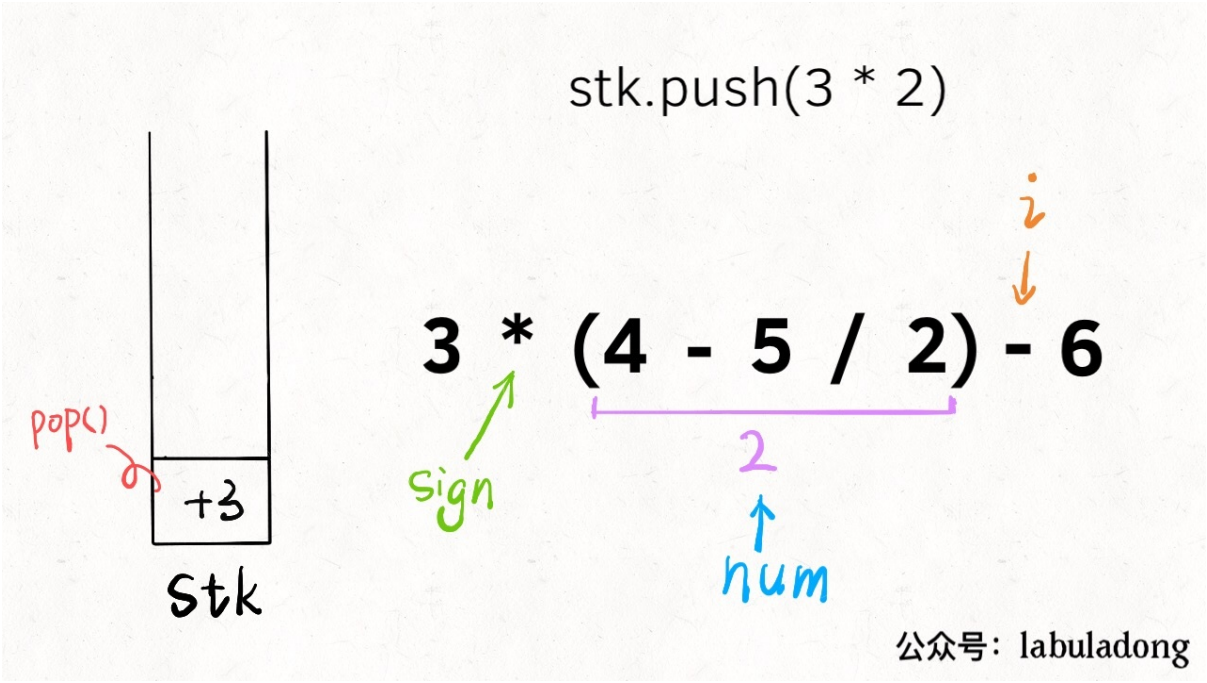
现在的问题是，递归的开始条件和结束条件是什么？**遇到 `(` 开始递归，遇到 `)` 结束递归**：

```
def calculate(s: str) -> int:

    def helper(s: List) -> int:
        stack = []
        sign = '+'
        num = 0

        while len(s) > 0:
            c = s.pop(0)
            if c.isdigit():
                num = 10 * num + int(c)
            # 遇到左括号开始递归计算 num
            if c == '(':
                num = helper(s)

            if (not c.isdigit() and c != ' ') or len(s) == 0:
                if sign == '+': ...
                elif sign == '-': ...
                elif sign == '*': ...
                elif sign == '/': ...
                num = 0
                sign = c
            # 遇到右括号返回递归结果
            if c == ')': break
        return sum(stack)
```

```
        return helper(list(s))
```







你看，加了两三行代码，就可以处理括号了，这就是递归的魅力。至此，计算器的全部功能就实现了，通过对问题的层层拆解化整为零，再回头看，这个问题似乎也没那么复杂嘛。

五、最后总结

本文借实现计算器的问题，主要想表达的是一种处理复杂问题的思路。

我们首先从字符串转数字这个简单问题开始，进而处理只包含加减法的算式，进而处理包含加减乘除四则运算的算式，进而处理空格字符，进而处理包含括号的算式。

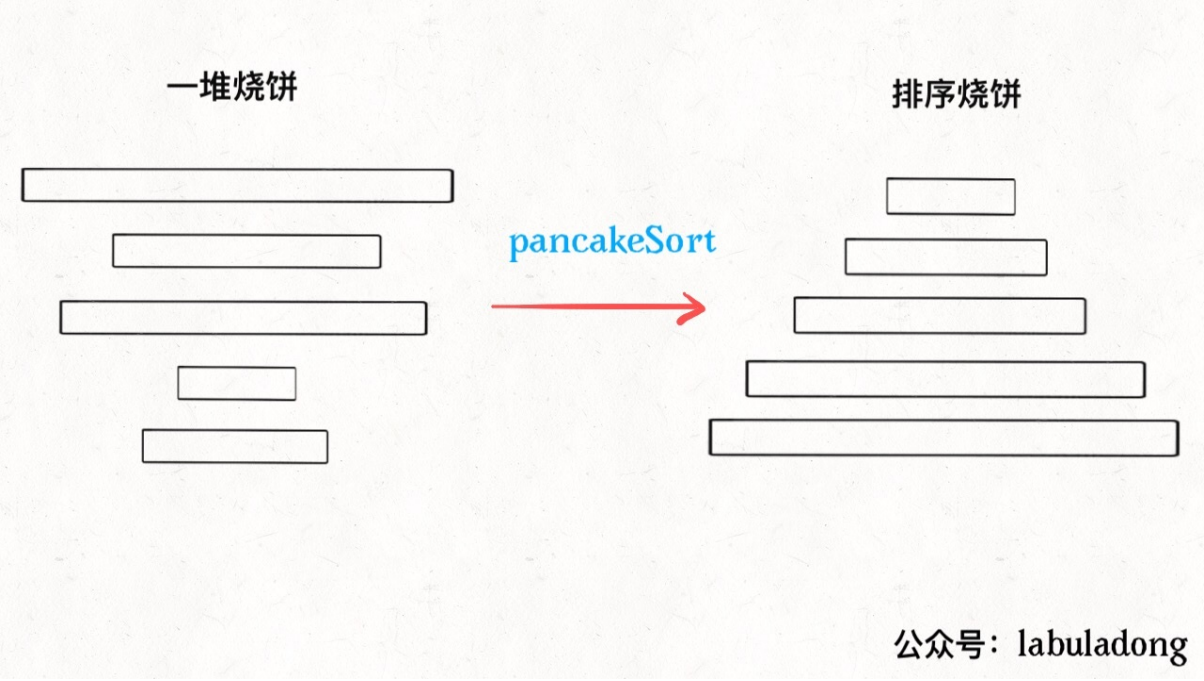
可见，对于一些比较困难的问题，其解法并不是一蹴而就的，而是步步推进，螺旋上升的。如果一开始给你原题，你不会做，甚至看不懂答案，都很正常，关键在于我们自己如何简化问题，如何以退为进。

退而求其次是一种很聪明策略。你想想啊，假设这是一道考试题，你不会实现这个计算器，但是你写了字符串转整数的算法并指出了容易溢出的陷阱，那起码可以得 20 分吧；如果你能够处理加减法，那可以得 40 分吧；如果你能处理加减乘除四则运算，那起码够 70 分了；再加上处理空格字符，80 有了吧。我就是不会处理括号，那就算了，80 已经很 OK 了好不好。

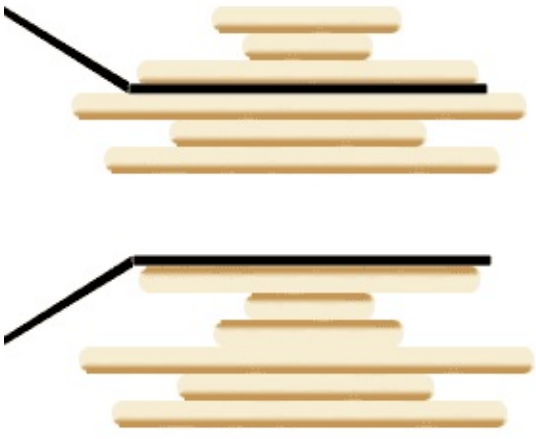
致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

烧饼排序

烧饼排序是个很有意思的实际问题：假设盘子上有 `n` 块**面积大小不**一的烧饼，你如何用一把锅铲进行若干次翻转，让这些烧饼的大小有序（小的在上，大的在下）？



设想一下用锅铲翻转一堆烧饼的情景，其实是有一点限制的，我们每次只能将最上面的若干块饼子翻转：



我们的问题是，**如何使用算法得到一个翻转序列，使得烧饼堆变得有序**？

首先，需要把这个问题抽象，用数组来表示烧饼堆：

给定数组 `A`，我们可以对其进行**煎饼翻转**：我们选择一些正整数 `k <= A.length`，然后反转 `A` 的前 **k** 个元素的顺序。我们要执行若干次煎饼翻转（按顺序一次接一次地进行）以完成对数组 `A` 的排序。

返回能使 `A` 排序的煎饼翻转操作所对应的 k 值序列。任何将数组排序且翻转次数在 `10 * A.length` 范围内的有效答案都将被判断为正确。

示例 1：

```
输入：[3,2,4,1]
输出：[4,2,4,3]
解释：
我们执行 4 次煎饼翻转，k 值分别为 4，2，4，和 3。
初始状态 A = [3, 2, 4, 1]
第一次翻转后 (k=4): A = [1, 4, 2, 3]
第二次翻转后 (k=2): A = [4, 1, 2, 3]
第三次翻转后 (k=4): A = [3, 2, 1, 4]
第四次翻转后 (k=3): A = [1, 2, 3, 4]，此时已完成排序。
```

示例 2：

```
输入：[1,2,3]
输出：[]
解释：
输入已经排序，因此不需要翻转任何内容。
请注意，其他可能的答案，如[3，3]，也将被接受。
```

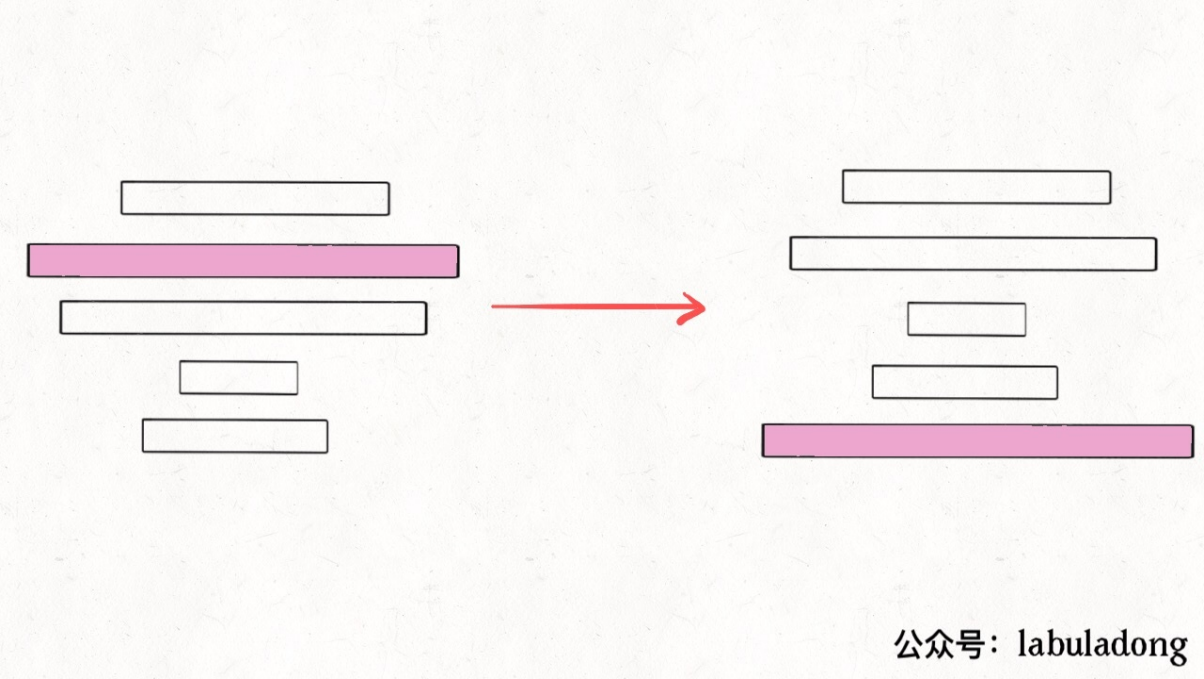
如何解决这个问题呢？其实类似上篇文章 递归反转链表的一部分，这也是需要**递归思想**的。

一、思路分析

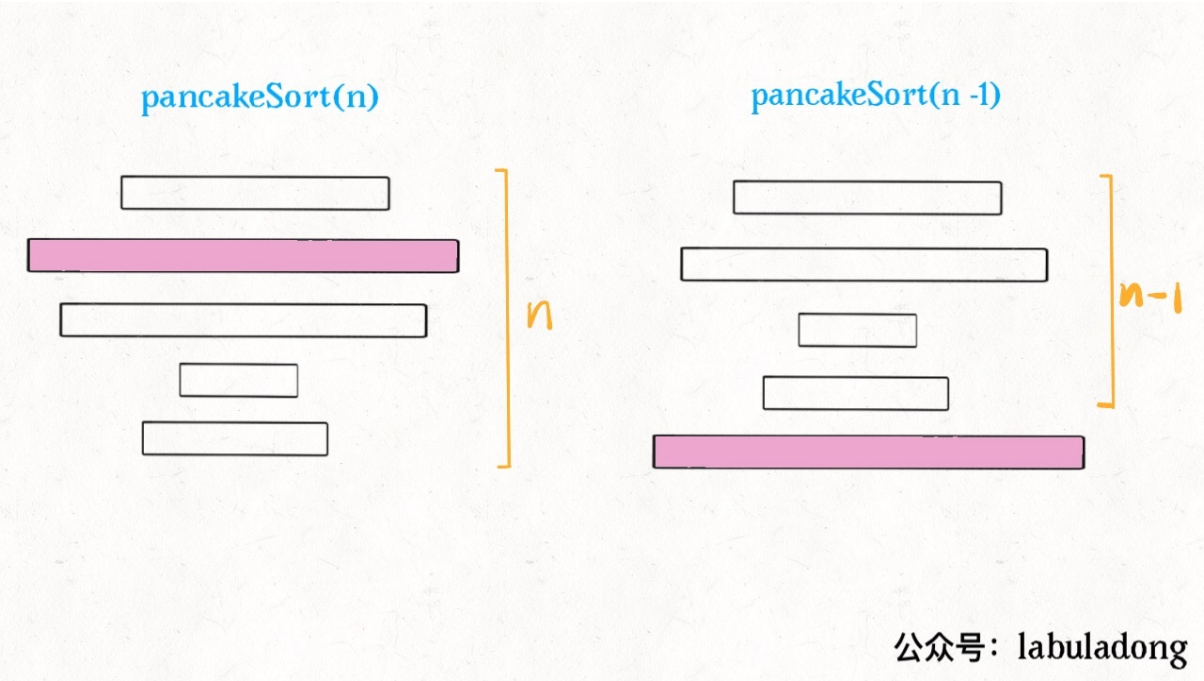
为什么说这个问题有递归性质呢？比如说我们需要实现这样一个函数：

```
// cakes 是一堆烧饼，函数会将前 n 个烧饼排序
void sort(int[] cakes, int n);
```

如果我们找到了前 `n` 个烧饼中最大的那个，然后设法将这个饼子翻转到最底下：



那么，原问题的规模就可以减小，递归调用 `pancakeSort(A, n-1)` 即可：



接下来，对于上面的这 `n - 1` 块饼，如何排序呢？还是先从中找到最大的一块饼，然后把这块饼放到底下，再递归调用 `pancakeSort(A, n-1-1)` .....

你看，这就是递归性质，总结一下思路就是：

1、找到 `n` 个饼中最大的那个。

2、把这个最大的饼移到最底下。

3、递归调用 `pancakeSort(A, n - 1)`。

base case：`n == 1` 时，排序 1 个饼时不需要翻转。

那么，最后剩下个问题，**如何设法将某块烧饼翻到最后呢**？

其实很简单，比如第 3 块饼是最大的，我们想把它换到最后，也就是换到第 `n` 块。可以这样操作：

1、用锅铲将前 3 块饼翻转一下，这样最大的饼就翻到了最上面。

2、用锅铲将前 `n` 块饼全部翻转，这样最大的饼就翻到了第 `n` 块，也就是最后一块。

以上两个流程理解之后，基本就可以写出解法了，不过题目要求我们写出具体的反转操作序列，这也很简单，只要在每次翻转烧饼时记录下来就行了。

二、代码实现

只要把上述的思路用代码实现即可，唯一需要注意的是，数组索引从 0 开始，而我们要返回的结果是从 1 开始算的。

```
// 记录反转操作序列
LinkedList<Integer> res = new LinkedList<>();

List<Integer> pancakeSort(int[] cakes) {
    sort(cakes, cakes.length);
    return res;
}

void sort(int[] cakes, int n) {
    // base case
    if (n == 1) return;

    // 寻找最大饼的索引
    int maxCake = 0;
    int maxCakeIndex = 0;
    for (int i = 0; i < n; i++)
```

```
            if (cakes[i] > maxCake) {
                maxCakeIndex = i;
                maxCake = cakes[i];
            }

        // 第一次翻转，将最大饼翻到最上面
        reverse(cakes, 0, maxCakeIndex);
        res.add(maxCakeIndex + 1);
        // 第二次翻转，将最大饼翻到最下面
        reverse(cakes, 0, n - 1);
        res.add(n);

        // 递归调用
        sort(cakes, n - 1);
    }

    void reverse(int[] arr, int i, int j) {
        while (i < j) {
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
            i++; j--;
        }
    }
```

通过刚才的详细解释，这段代码应该是很清晰了。

算法的时间复杂度很容易计算，因为递归调用的次数是 `n`，每次递归调用都需要一次 for 循环，时间复杂度是 O(n)，所以总的复杂度是 O(n^2)。

最后，我们可以思考一个问题：按照我们这个思路，得出的操作序列长度应该为 `2(n - 1)`，因为每次递归都要进行 2 次翻转并记录操作，总共有 `n` 层递归，但由于 base case 直接返回结果，不进行翻转，所以最终的操作序列长度应该是固定的 `2(n - 1)`。

显然，这个结果不是最优的（最短的），比如说一堆煎饼 `[3,2,4,1]`，我们的算法得到的翻转序列是 `[3,4,2,3,1,2]`，但是最快捷的翻转方法应该是 `[2,3,4]`：

初始状态 ：[3,2,4,1] 翻前 2 个：[2,3,4,1] 翻前 3 个：[4,3,2,1] 翻前 4 个：[1,2,3,4]

如果要求你的算法计算排序烧饼的**最短**操作序列，你该如何计算呢？或者说，解决这种求最优解法的问题，核心思路什么，一定需要使用什么算法技巧呢？

不妨分享一下你的思考。

前缀和技巧

今天来聊一道简单却十分巧妙的算法问题：算出一共有几个和为 k 的子数组。

给定一个整数数组和一个整数 **k**，你需要找到该数组中和为 **k** 的连续的子数组的个数。

示例 1：

```
输入:nums = [1,1,1], k = 2
输出: 2 , [1,1] 与 [1,1] 为两种不同的情况。
```

说明：

1. 数组的长度为 [1, 20,000]。
2. 数组中元素的范围是 [-1000, 1000] ，且整数 **k** 的范围是 [-1e7, 1e7]。

那我把所有子数组都穷举出来，算它们的和，看看谁的和等于 k 不就行了。

关键是，**如何快速得到某个子数组的和呢**，比如说给你一个数组 `nums`，让你实现一个接口 `sum(i, j)`，这个接口要返回 `nums[i..j]` 的和，而且会被多次调用，你怎么实现这个接口呢？

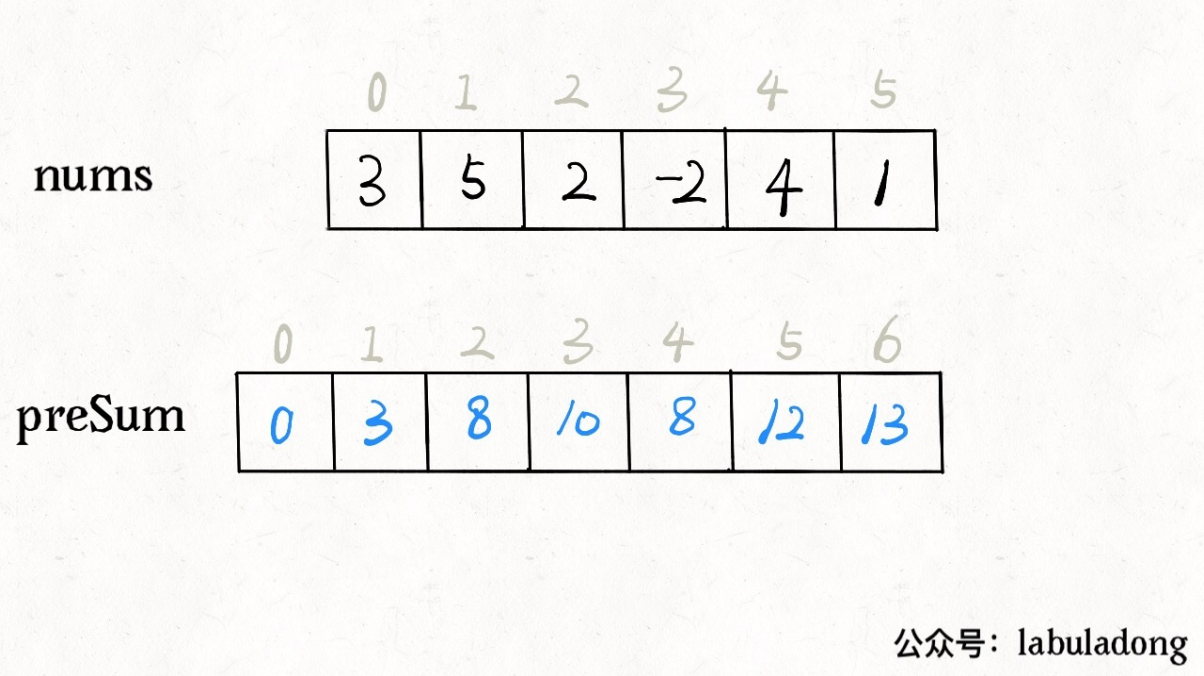
因为接口要被多次调用，显然不能每次都去遍历 `nums[i..j]`，有没有一种快速的方法在 O(1) 时间内算出 `nums[i..j]` 呢？这就需要**前缀和**技巧了。

一、什么是前缀和

前缀和的思路是这样的，对于一个给定的数组 `nums`，我们额外开辟一个前缀和数组进行预处理：

```
int n = nums.length;
// 前缀和数组
int[] preSum = new int[n + 1];
preSum[0] = 0;
```

```
for (int i = 0; i < n; i++)
    preSum[i + 1] = preSum[i] + nums[i];
```



这个前缀和数组 `preSum` 的含义也很好理解，`preSum[i]` 就是 `nums[0..i-1]` 的和。那么如果我们想求 `nums[i..j]` 的和，只需要一步操作 `preSum[j+1]-preSum[i]` 即可，而不需要重新去遍历数组了。

回到这个子数组问题，我们想求有多少个子数组的和为 k，借助前缀和技巧很容易写出一个解法：

```
int subarraySum(int[] nums, int k) {
    int n = nums.length;
    // 构造前缀和
    int[] sum = new int[n + 1];
    sum[0] = 0;
    for (int i = 0; i < n; i++)
        sum[i + 1] = sum[i] + nums[i];

    int ans = 0;
    // 穷举所有子数组
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++)
            // sum of nums[j..i-1]
            if (sum[i] - sum[j] == k)
```

```
                ans++;

        return ans;
    }
```

这个解法的时间复杂度 $O(N^2)$ 空间复杂度 $O(N)$，并不是最优的解法。不过通过这个解法理解了前缀和数组的工作原理之后，可以使用一些巧妙的办法把时间复杂度进一步降低。

二、优化解法

前面的解法有嵌套的 for 循环：

```
for (int i = 1; i <= n; i++)
    for (int j = 0; j < i; j++)
        if (sum[i] - sum[j] == k)
            ans++;
```

第二层 for 循环在干嘛呢？翻译一下就是，**在计算，有几个 `j` 能够使得 `sum[i]` 和 `sum[j]` 的差为 k**。每找到一个这样的 `j`，就把结果加一。

我们可以把 if 语句里的条件判断移项，这样写：

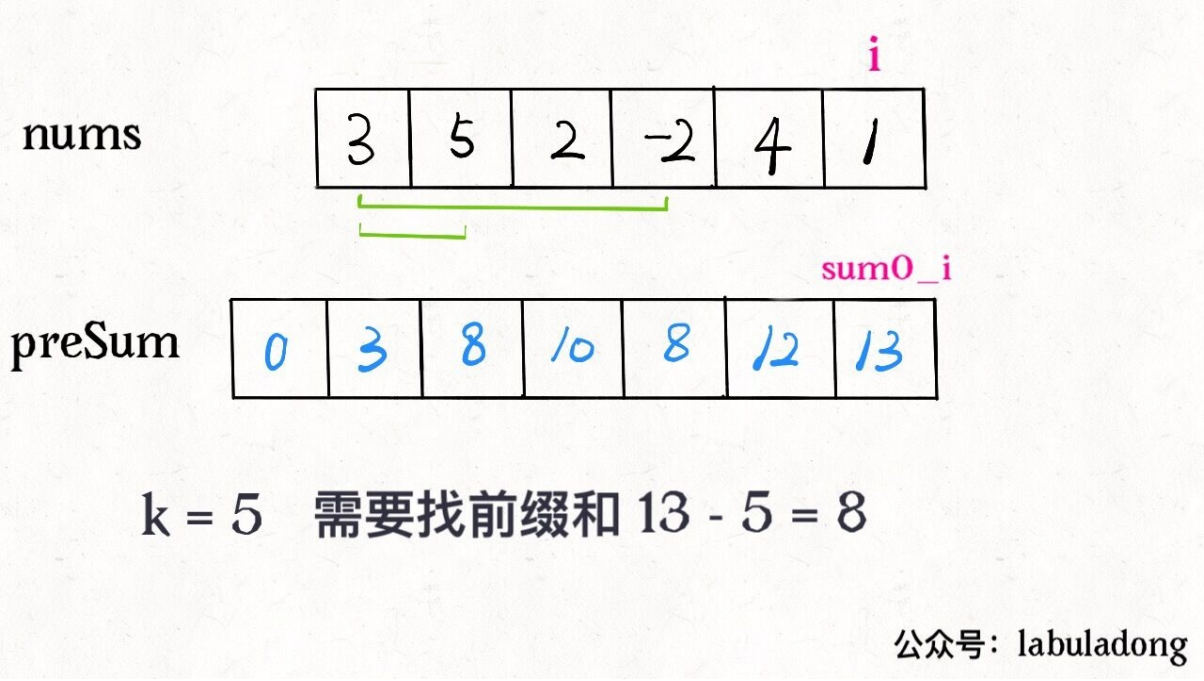
```
if (sum[j] == sum[i] - k)
    ans++;
```

优化的思路是：**我直接记录下有几个 `sum[j]` 和 `sum[i] - k` 相等，直接更新结果，就避免了内层的 for 循环**。我们可以用哈希表，在记录前缀和的同时记录该前缀和出现的次数。

```
int subarraySum(int[] nums, int k) {
    int n = nums.length;
    // map：前缀和 -> 该前缀和出现的次数
    HashMap<Integer, Integer>
        preSum = new HashMap<>();
    // base case
```

```
        preSum.put(0, 1);

        int ans = 0, sum0_i = 0;
        for (int i = 0; i < n; i++) {
            sum0_i += nums[i];
            // 这是我们想找的前缀和 nums[0..j]
            int sum0_j = sum0_i - k;
            // 如果前面有这个前缀和，则直接更新答案
            if (preSum.containsKey(sum0_j))
                ans += preSum.get(sum0_j);
            // 把前缀和 nums[0..i] 加入并记录出现次数
            preSum.put(sum0_i,
                preSum.getOrDefault(sum0_i, 0) + 1);
        }
        return ans;
    }
```

比如说下面这个情况，需要前缀和 8 就能找到和为 k 的子数组了，之前的暴力解法需要遍历数组去数有几个 8，而优化解法借助哈希表可以直接得知有几个前缀和为 8。



这样，就把时间复杂度降到了 $O(N)$，是最优解法了。

三、总结

前缀和不难，却很有用，主要用于处理数组区间的问题。

比如说，让你统计班上同学考试成绩在不同分数段的百分比，也可以利用前缀和技巧：

```
int[] scores; // 存储着所有同学的分数
// 试卷满分 150 分
int[] count = new int[150 + 1]
// 记录每个分数有几个同学
for (int score : scores)
    count[score]++
// 构造前缀和
for (int i = 1; i < count.length; i++)
    count[i] = count[i] + count[i-1];
```

这样，给你任何一个分数段，你都能通过前缀和相减快速计算出这个分数段的人数，百分比也就很容易计算了。

但是，稍微复杂一些的算法问题，不止考察简单的前缀和技巧。比如本文探讨的这道题目，就需要借助前缀和的思路做进一步的优化，借助哈希表去除不必要的嵌套循环。可见对题目的理解和细节的分析能力对于算法的优化是至关重要的。

希望本文对你有帮助。

字符串乘法

对于比较小的数字，做运算可以直接使用编程语言提供的运算符，但是如果相乘的两个因数非常大，语言提供的数据类型可能就会溢出。一种替代方案就是，运算数以字符串的形式输入，然后模仿我们小学学习的乘法算术过程计算出结果，并且也用字符串表示。

给定两个以字符串形式表示的非负整数 `num1` 和 `num2`，返回 `num1` 和 `num2` 的乘积，它们的乘积也表示为字符串形式。

示例 1:

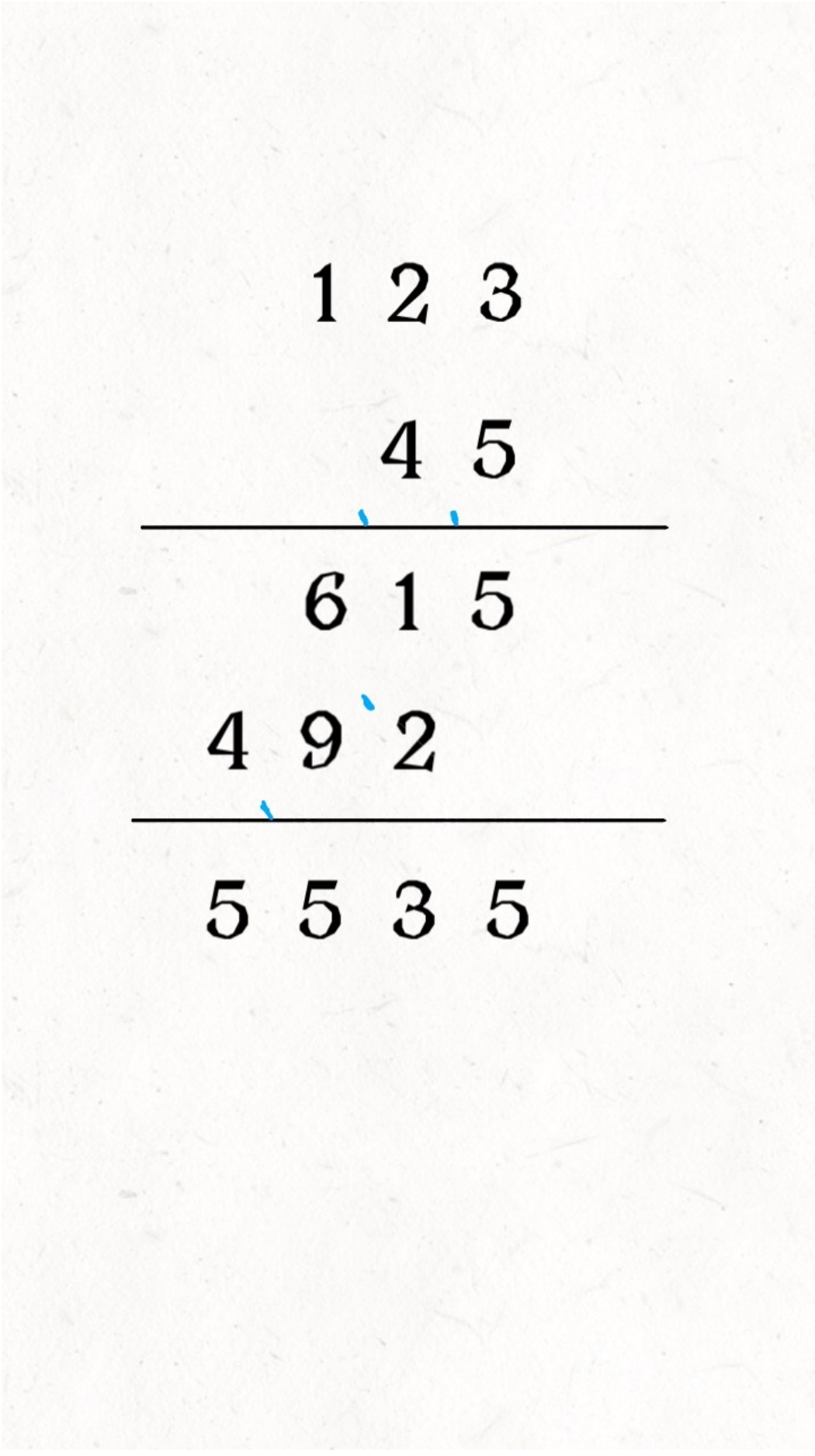
```
输入: num1 = "2", num2 = "3"
输出: "6"
```

示例 2:

```
输入: num1 = "123", num2 = "456"
输出: "56088"
```

需要注意的是，`num1` 和 `num2` 可以非常长，所以不可以把他们直接转成整型然后运算，唯一的思路就是模仿我们手算乘法。

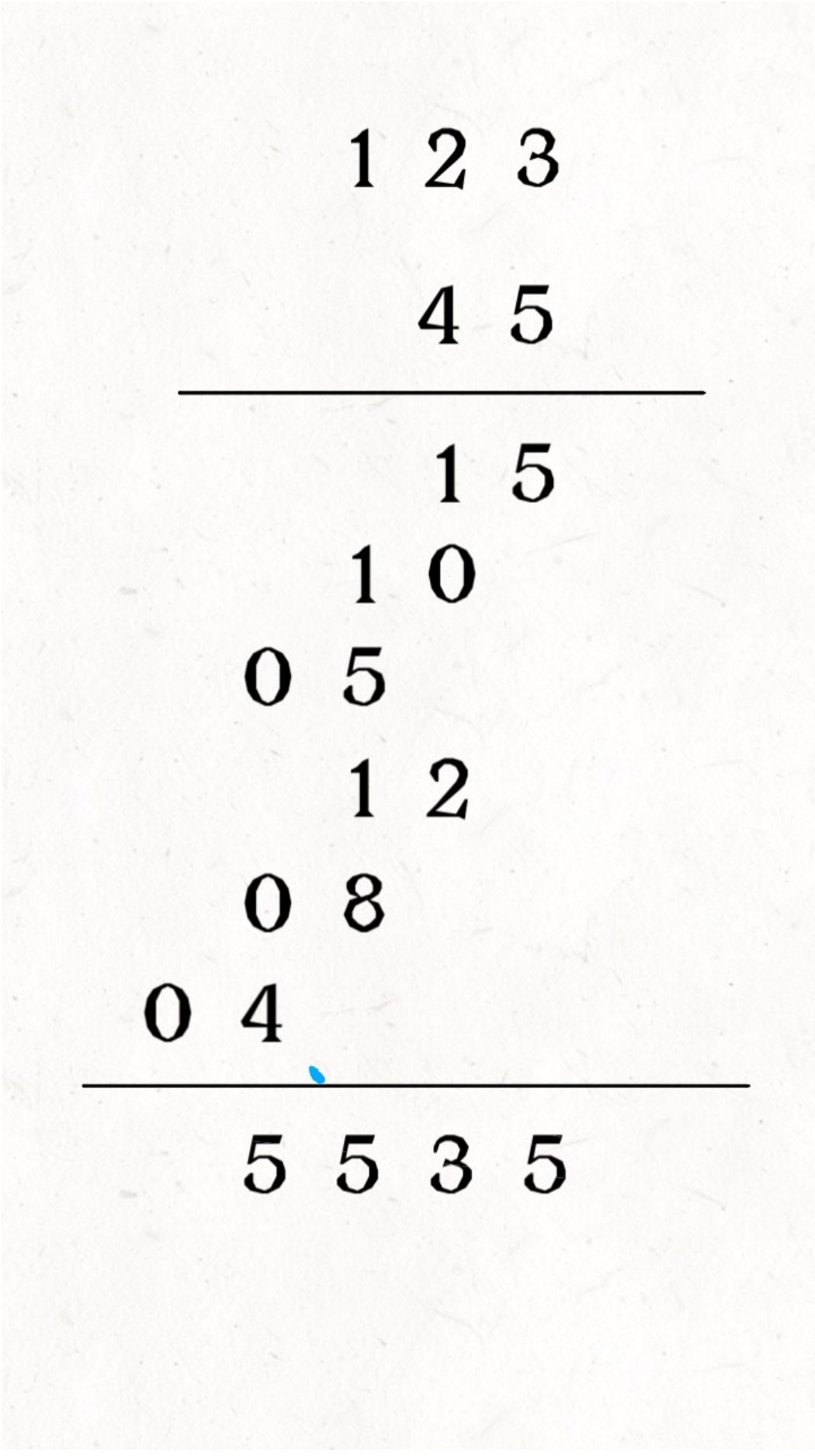
比如说我们手算 `123 × 45`，应该会这样计算：


1 2 3
4 5
6 1 5
4 9 2
5 5 3 5

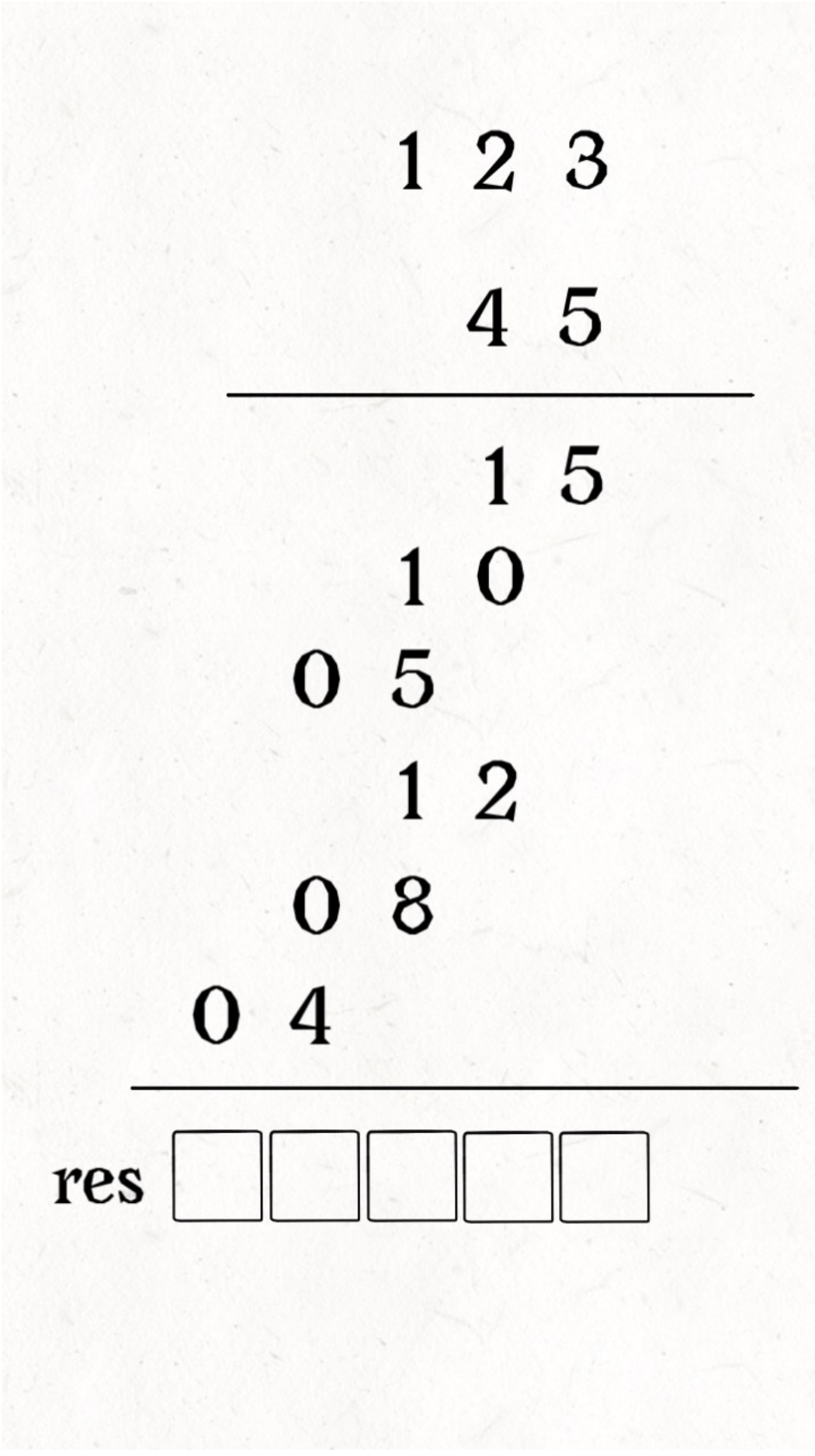
计算 `123 × 5`，再计算 `123 × 4`，最后错一位相加。这个流程恐怕小学生都可以熟练完成，但是你是否能**把这个运算过程进一步机械化**，写成一套算法指令让没有任何智商的计算机来执行呢？

你看这个简单过程，其中涉及乘法进位，涉及错位相加，还涉及加法进位；而且还有一些不易察觉的问题，比如说两位数乘以两位数，结果可能是四位数，也可能是三位数，你怎么想出一个标准化的处理方式？这就是算法的魅力，如果没有计算机思维，简单的问题可能都没办法自动化处理。

首先，我们这种手算方式还是太「高级」了，我们要再「低级」一点，`123 × 5` 和 `123 × 4` 的过程还可以进一步分解，最后再相加：


1 2 3
4 5
1 5
1 0
0 5
1 2
0 8
0 4
5 5 3 5

现在 `123` 并不大，如果是个很大的数字的话，是无法直接计算乘积的。我们可以用一个数组在底下接收相加结果：

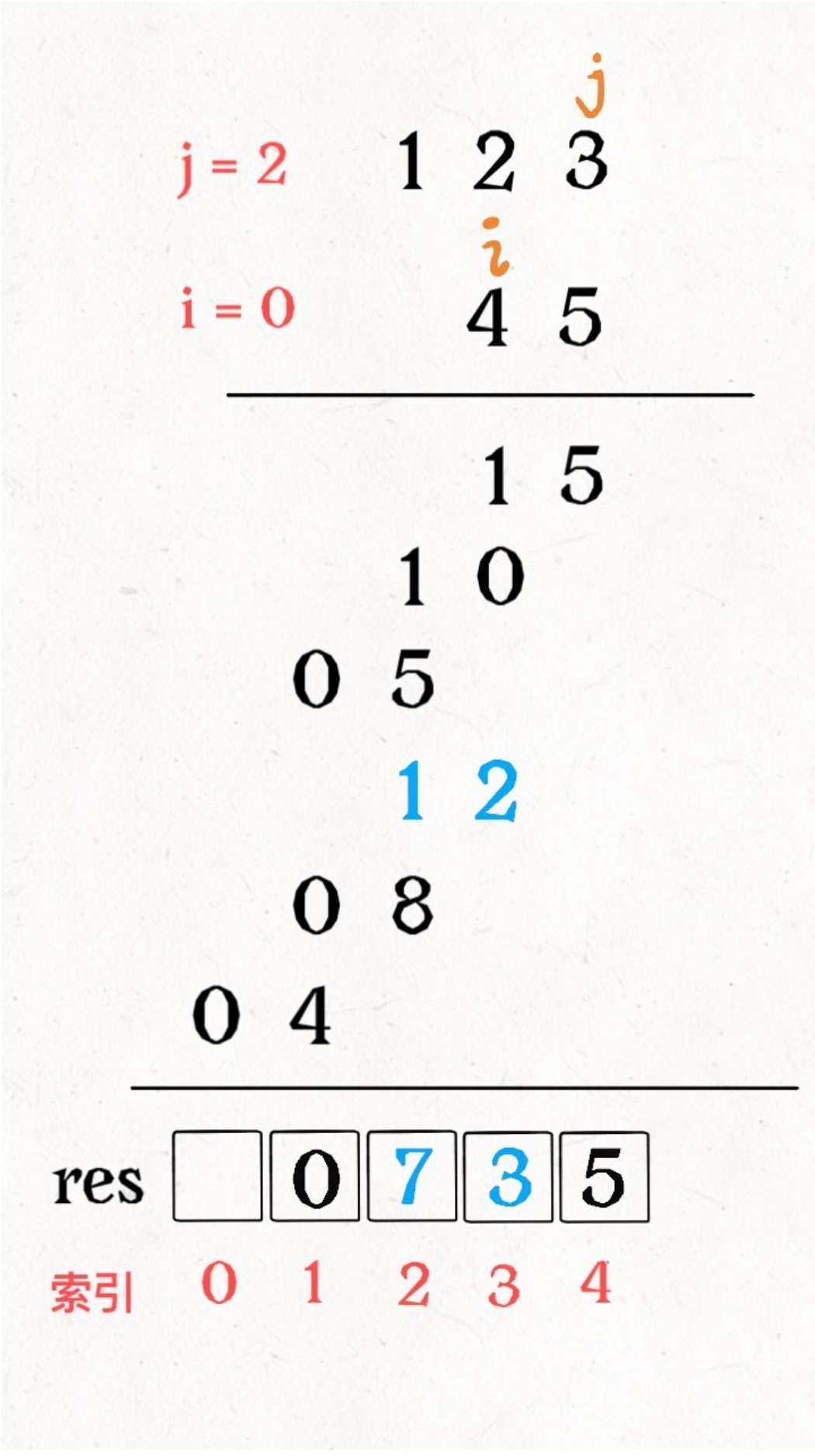

1 2 3
4 5
1 5
1 0
0 5
1 2
0 8
0 4
res

整个计算过程大概是这样，**有两个指针 `i, j` 在 `num1` 和 `num2` 上游走，计算乘积，同时将乘积叠加到 `res` 的正确位置**：

【pdf/mobi格式不支持 GIF:%E5%AD%97%E7%AC%A6%E4%B8%B2%E4%B9%98%E6%B3%95/4.gif】 请查看【关于本小抄及作者】章节的解决方案

现在还有一个关键问题，如何将乘积叠加到 `res` 的正确位置，或者说，如何通过 `i, j` 计算 `res` 的对应索引呢？

其实，细心观察之后就发现，**`num1[i]` 和 `num2[j]` 的乘积对应的就是 `res[i+j]` 和 `res[i+j+1]` 这两个位置**。


j
j = 2
1 2 3
i
i = 0
4 5
1 5
1 0
0 5
1 2
0 8
0 4
res
0 7 3 5
索引
0 1 2 3 4

明白了这一点，就可以用代码模仿出这个计算过程了：

```
string multiply(string num1, string num2) {
    int m = num1.size(), n = num2.size();
    // 结果最多为 m + n 位数
    vector<int> res(m + n, 0);
    // 从个位数开始逐位相乘
    for (int i = m - 1; i >= 0; i--)
        for (int j = n - 1; j >= 0; j--) {
            int mul = (num1[i]-'0') * (num2[j]-'0');
            // 乘积在 res 对应的索引位置
            int p1 = i + j, p2 = i + j + 1;
            // 叠加到 res 上
            int sum = mul + res[p2];
            res[p2] = sum % 10;
            res[p1] += sum / 10;
        }
    // 结果前缀可能存的 0（未使用的位）
    int i = 0;
    while (i < res.size() && res[i] == 0)
        i++;
    // 将计算结果转化成字符串
    string str;
    for (; i < res.size(); i++)
        str.push_back('0' + res[i]);

    return str.size() == 0 ? "0" : str;
}
```

至此，字符串乘法算法就完成了。

总结一下，我们习以为常的一些思维方式，在计算机看来是非常难以做到的。比如说我们习惯的算术流程并不复杂，但是如果让你再进一步，翻译成代码逻辑，并不简单。算法需要将计算流程再简化，通过边算边叠加的方式来得到结果。

俗话教育我们，不要陷入思维定式，不要程序化，要发散思维，要创新。但我觉得程序化并不是坏事，可以大幅提高效率，减小失误率。算法不就是一套程序化的思维吗，只有程序化才能让计算机帮助我们解决复杂问题呀！

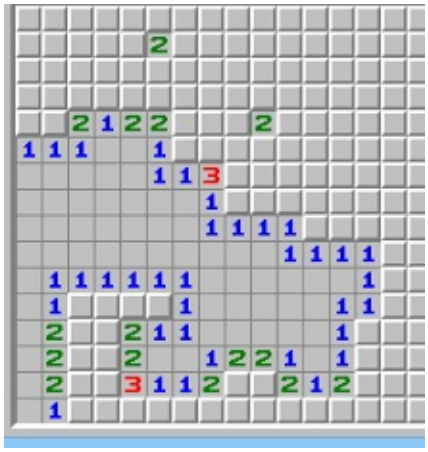
也许算法就是一种**寻找思维定式的思维**吧，希望本文对你有帮助。

FloodFill算法详解及应用

啥是 FloodFill 算法呢，最直接的一个应用就是「颜色填充」，就是 Windows 绘画本中那个小油漆桶的标志，可以把一块被圈起来的区域全部染色。

【pdf/mobi格式不支持GIF:floodfill/floodfill.gif】 请查看【关于本小抄及作者】章节的解决方案

这种算法思想还在许多其他地方有应用。比如说扫雷游戏，有时候你点一个方格，会一下子展开一片区域，这个展开过程，就是 FloodFill 算法实现的。



类似的，像消消乐这类游戏，相同方块积累到一定数量，就全部消除，也是 FloodFill 算法的功劳。

通过以上的几个例子，你应该对 FloodFill 算法有个概念了，现在我们要抽象问题，提取共同点。

一、构建框架

以上几个例子，都可以抽象成一个二维矩阵（图片其实就是像素点矩阵），然后从某个点开始向四周扩展，直到无法再扩展为止。

矩阵，可以抽象为一幅「图」，这就是一个图的遍历问题，也就类似一个 N 叉树遍历的问题。几行代码就能解决，直接上框架吧：

```
// (x, y) 为坐标位置
void fill(int x, int y) {
    fill(x - 1, y); // 上
    fill(x + 1, y); // 下
    fill(x, y - 1); // 左
    fill(x, y + 1); // 右
```

```
}
```

这个框架可以解决所有在二维矩阵中遍历的问题，说得高端一点，这就叫深度优先搜索（Depth First Search，简称 DFS），说得简单一点，这就叫四叉树遍历框架。坐标 (x, y) 就是 root，四个方向就是 root 的四个子节点。

下面看一道 LeetCode 题目，其实就是让我们来实现一个「颜色填充」功能。

有一幅以二维整数数组表示的图画，每一个整数表示该图画的像素值大小，数值在 0 到 65535 之间。

给你一个坐标 `(sr, sc)` 表示图像渲染开始的像素值（行，列）和一个新的颜色值 `newColor`。

让你重新上色这幅图像。最后返回经过上色渲染后的图像。

示例:

```
输入:                                    输出:
image = [[1,1,1],                        [[2,2,2],
         [1,1,0],                         [2,2,0],
         [1,0,1]]                         [2,0,1]]
sr = 1, sc = 1, newColor = 2

解析:
坐标 (sr,sc)=(1,1) 在图像的正中间，与其相连的所有符合条件的像素点的颜色都被更改成2。
注意，右下角的像素没有更改为2，因为它不是在上下左右四个方向上与初始点相连的像素点。
```

根据上篇文章，我们讲了「树」算法设计的一个总路线，今天就可以用到：

```
int[][] floodFill(int[][] image,
        int sr, int sc, int newColor) {

    int origColor = image[sr][sc];
    fill(image, sr, sc, origColor, newColor);
    return image;
}

void fill(int[][] image, int x, int y,
        int origColor, int newColor) {
    // 出界：超出边界索引
    if (!inArea(image, x, y)) return;
    // 碰壁：遇到其他颜色，超出 origColor 区域
    if (image[x][y] != origColor) return;
```

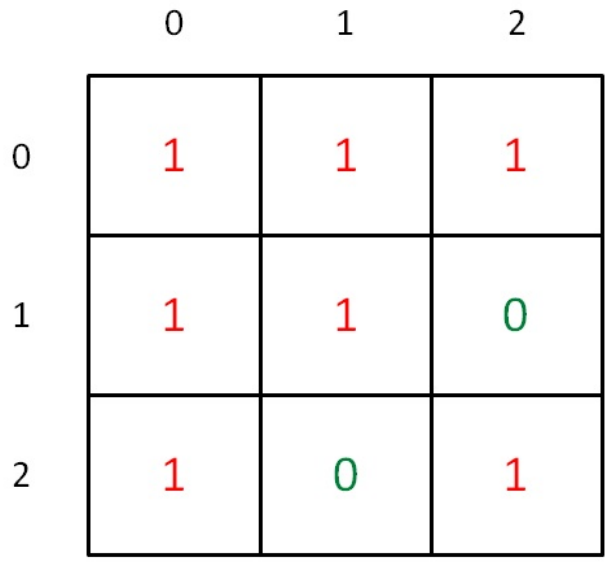
```
        image[x][y] = newColor;

        fill(image, x, y + 1, origColor, newColor);
        fill(image, x, y - 1, origColor, newColor);
        fill(image, x - 1, y, origColor, newColor);
        fill(image, x + 1, y, origColor, newColor);
    }

    boolean inArea(int[][] image, int x, int y) {
        return x >= 0 && x < image.length
            && y >= 0 && y < image[0].length;
    }
```

只要你能够理解这段代码，一定要给你鼓掌，给你 99 分，因为你对「框架思维」的掌控已经炉火纯青，此算法已经 cover 了 99% 的情况，仅有一个细节问题没有解决，就是当 origColor 和 newColor 相同时，会陷入无限递归。

二、研究细节

为什么会陷入无限递归呢，很好理解，因为每个坐标都要搜索上下左右，那么对于一个坐标，一定会被上下左右的坐标搜索。**被重复搜索时，必须保证递归函数能够能正确地退出，否则就会陷入死循环**。

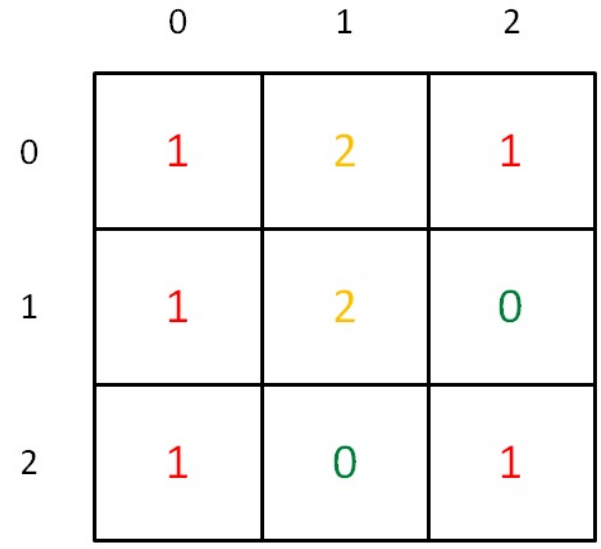
为什么 newColor 和 origColor 不同时可以正常退出呢？把算法流程画个图理解一下：



fill(1, 1):
(1, 1) = newColor
fill(0, 1):
(0, 1) = newColor
fill(-1, 1): *出界, return*
fill(1, 1) *
fill(0, 0)
fill(0, 2)
fill(1, 2)
fill(0, 1)
fill(1, 2)

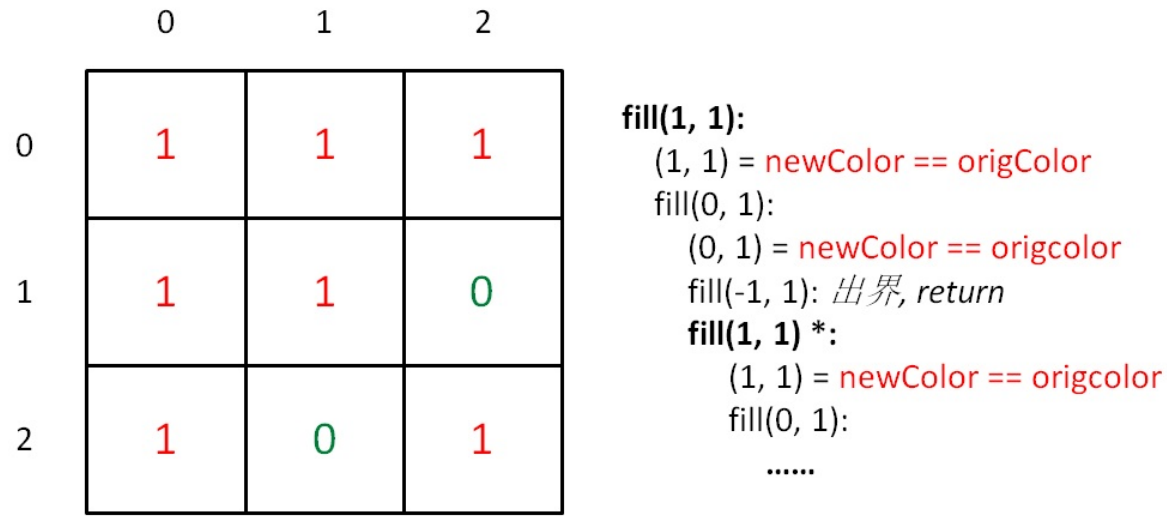
可以看到，fill(1, 1) 被重复搜索了，我们用 fill(1, 1) 表示这次重复搜索。*fill(1, 1)* 执行时，(1, 1) 已经被换成了 newColor，所以 fill(1, 1)* 会在这个 if 语句被怼回去，正确退出了。

```
// 碰壁：遇到其他颜色，超出 origColor 区域
if (image[x][y] != origColor) return;
```



执行到 fill(1, 1)* 时，(1, 1) 已变成 newColor，不再是 origColor，所以会「碰壁」直接返回

但是，如果说 origColor 和 newColor 一样，这个 if 语句就无法让 fill(1, 1)* 正确退出，而是开启了下面的重复递归，形成了死循环。



三、处理细节

如何避免上述问题的发生，最容易想到的就是用一个和 image 一样大小的二维 bool 数组记录走过的地方，一旦发现重复立即 return。

```
 // 出界：超出边界索引
if (!inArea(image, x, y)) return;
// 碰壁：遇到其他颜色，超出 origColor 区域
if (image[x][y] != origColor) return;
// 不走回头路
if (visited[x][y]) return;
visited[x][y] = true;
image[x][y] = newColor;
```

完全 OK，这也是处理「图」的一种常用手段。不过对于此题，不用开数组，我们有一种更好的方法，那就是回溯算法。

前文「回溯算法详解」讲过，这里不再赘述，直接套回溯算法框架：

```
void fill(int[][] image, int x, int y,
        int origColor, int newColor) {
    // 出界：超出数组边界
    if (!inArea(image, x, y)) return;
    // 碰壁：遇到其他颜色，超出 origColor 区域
    if (image[x][y] != origColor) return;
```

```
        // 已探索过的 origColor 区域
        if (image[x][y] == -1) return;

        // choose：打标记，以免重复
        image[x][y] = -1;
        fill(image, x, y + 1, origColor, newColor);
        fill(image, x, y - 1, origColor, newColor);
        fill(image, x - 1, y, origColor, newColor);
        fill(image, x + 1, y, origColor, newColor);
        // unchoose：将标记替换为 newColor
        image[x][y] = newColor;
    }
```

这种解决方法是最常用的，相当于使用一个特殊值 -1 代替 visited 数组的作用，达到不走回头路的效果。为什么是 -1，因为题目中说了颜色取值在 0 - 65535 之间，所以 -1 足够特殊，能和颜色区分开。

四、拓展延伸：自动魔棒工具和扫雷

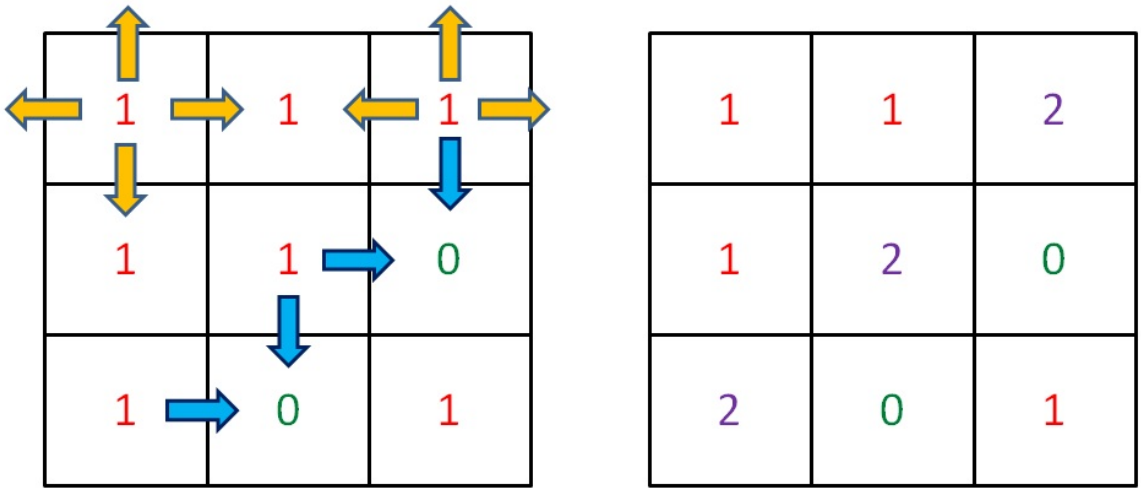
大部分图片编辑软件一定有「自动魔棒工具」这个功能：点击一个地方，帮你自动选中相近颜色的部分。如下图，我想选中老鹰，可以先用自动魔棒选中蓝天背景，然后反向选择，就选中了老鹰。我们来分析一下自动魔棒工具的原理。

显然，这个算法肯定是基于 FloodFill 算法的，但有两点不同：首先，背景色是蓝色，但不能保证都是相同的蓝色，毕竟是像素点，可能存在肉眼无法分辨的深浅差异，而我们希望能够忽略这种细微差异。第二，FloodFill 算法是「区域填充」，这里更像「边界填充」。

对于第一个问题，很好解决，可以设置一个阈值 threshold，在阈值范围内波动的颜色都视为 origColor：

```
if (Math.abs(image[x][y] - origColor) > threshold)
    return;
```

对于第二个问题，我们首先明确问题：不要把区域内所有 origColor 的都染色，而是只给区域最外圈染色。然后，我们分析，如何才能仅给外围染色，即如何才能找到最外围坐标，最外围坐标有什么特点？



可以发现，区域边界上的坐标，至少有一个方向不是 origColor，而区域内部的坐标，四面都是 origColor，这就是解决问题的关键。保持框架不变，使用 visited 数组记录已搜索坐标，主要代码如下：

```
int fill(int[][] image, int x, int y,
    int origColor, int newColor) {
    // 出界：超出数组边界
    if (!inArea(image, x, y)) return 0;
    // 已探索过的 origColor 区域
    if (visited[x][y]) return 1;
    // 碰壁：遇到其他颜色，超出 origColor 区域
    if (image[x][y] != origColor) return 0;

    visited[x][y] = true;

    int surround =
```

```
            fill(image, x - 1, y, origColor, newColor)
          + fill(image, x + 1, y, origColor, newColor)
          + fill(image, x, y - 1, origColor, newColor)
          + fill(image, x, y + 1, origColor, newColor);

    if (surround < 4)
        image[x][y] = newColor;

    return 1;
}
```

这样，区域内部的坐标探索四周后得到的 surround 是 4，而边界的坐标会遇到其他颜色，或超出边界索引，surround 会小于 4。如果你对这句话不理解，我们把逻辑框架抽象出来看：

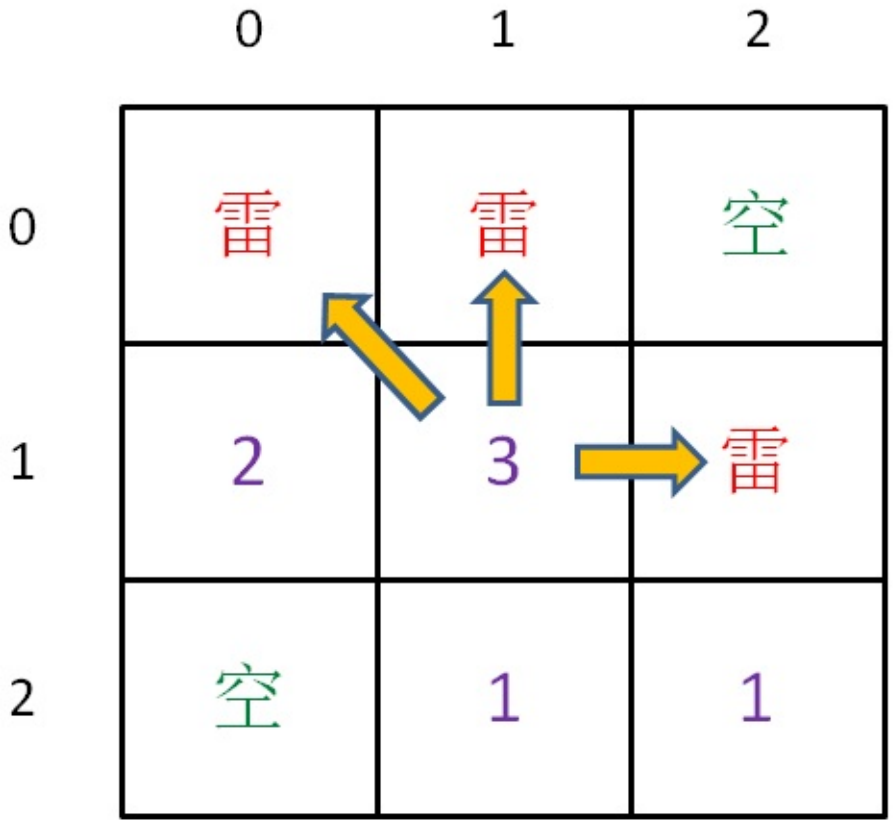
```
int fill(int[][] image, int x, int y,
    int origColor, int newColor) {
    // 出界：超出数组边界
    if (!inArea(image, x, y)) return 0;
    // 已探索过的 origColor 区域
    if (visited[x][y]) return 1;
    // 碰壁：遇到其他颜色，超出 origColor 区域
    if (image[x][y] != origColor) return 0;
    // 未探索且属于 origColor 区域
    if (image[x][y] == origColor) {
        // ...
        return 1;
    }
}
```

这 4 个 if 判断涵盖了 (x, y) 的所有可能情况，surround 的值由四个递归函数相加得到，而每个递归函数的返回值就这四种情况的一种。借助这个逻辑框架，你一定能理解上面那句话了。

这样就实现了仅对 origColor 区域边界坐标染色的目的，等同于完成了魔棒工具选定区域边界的功能。

这个算法有两个细节问题，一是必须借助 visited 来记录已探索的坐标，而无法使用回溯算法；二是开头几个 if 顺序不可打乱。读者可以思考一下原因。

同理，思考扫雷游戏，应用 FloodFill 算法展开空白区域的同时，也需要计算并显示边界上雷的个数，如何实现的？其实也是相同的思路，遇到雷就返回 true，这样 surround 变量存储的就是雷的个数。当然，扫雷的 FloodFill 算法不能只检查上下左右，还得加上四个斜向。



以上详细讲解了 FloodFill 算法的框架设计，**二维矩阵中的搜索问题，都逃不出这个算法框架**。

区间调度问题之区间合并

上篇文章用贪心算法解决了区间调度问题：给你很多区间，让你求其中的最大不重叠子集。

其实对于区间相关的问题，还有很多其他类型，本文就来讲讲区间合并问题（Merge Interval）。

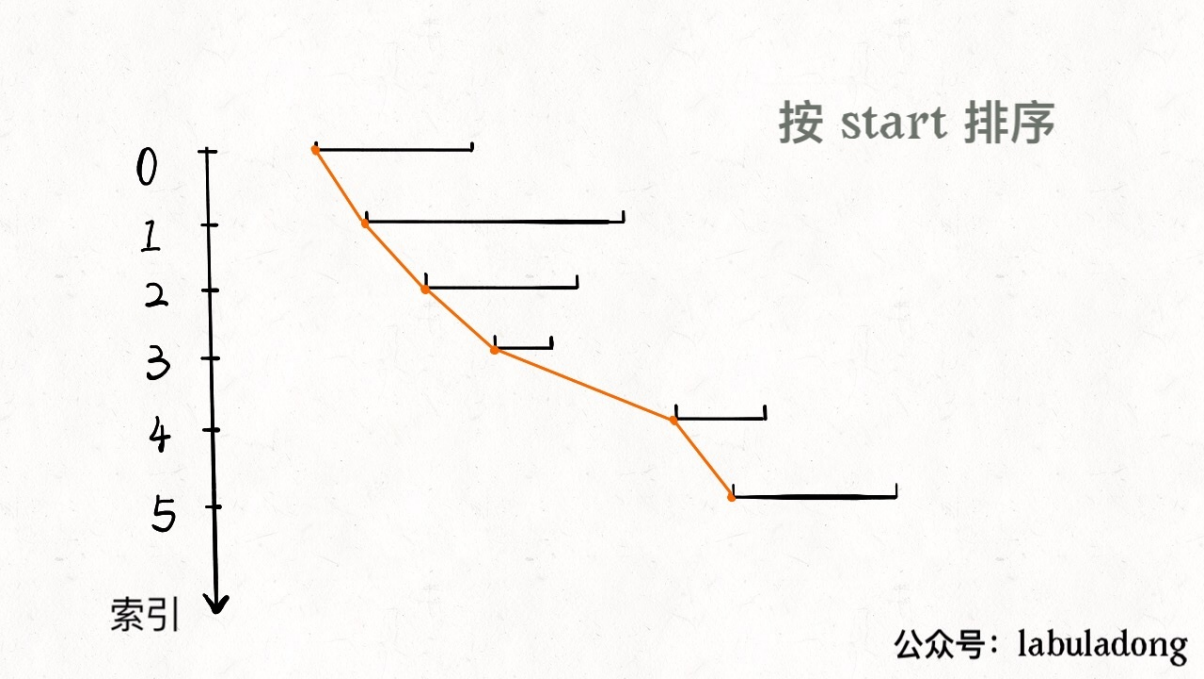
LeetCode 第 56 题就是一道相关问题，题目很好理解：

> 给出一个区间的集合，请合并所有重叠的区间。
>
> **示例 1:**
>
> ```
> 输入: [[1,3],[2,6],[8,10],[15,18]]
> 输出: [[1,6],[8,10],[15,18]]
> 解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
> ```
>
> **示例 2:**
>
> ```
> 输入: [[1,4],[4,5]]
> 输出: [[1,5]]
> 解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
> ```

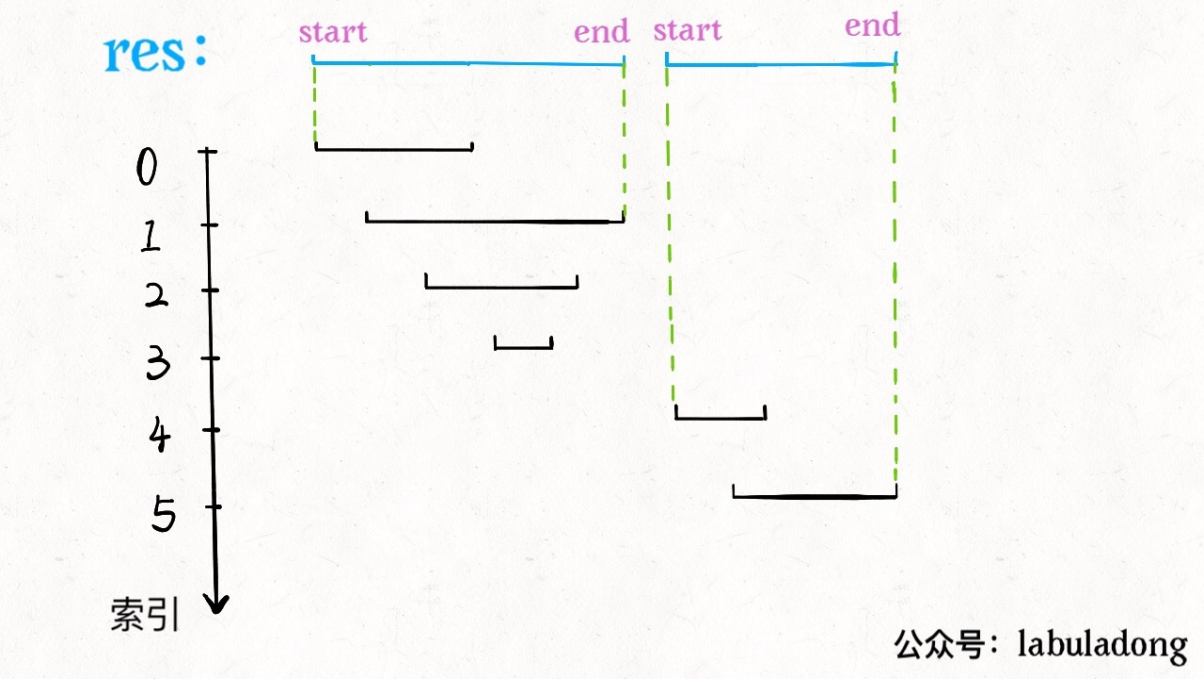
我们解决区间问题的一般思路是先排序，然后观察规律。

一、思路

一个区间可以表示为 `[start, end]`，前文聊的区间调度问题，需要按 `end` 排序，以便满足贪心选择性质。而对于区间合并问题，其实按 `end` 和 `start` 排序都可以，不过为了清晰起见，我们选择按 `start` 排序。



显然，对于几个相交区间合并后的结果区间 `x`，`x.start` 一定是这些相交区间中 `start` 最小的，`x.end` 一定是这些相交区间中 `end` 最大的。



由于已经排了序，`x.start` 很好确定，求 `x.end` 也很容易，可以类比在数组中找最大值的过程：

```
int max_ele = arr[0];
for (int i = 1; i < arr.length; i++)
    max_ele = max(max_ele, arr[i]);
```

```
    return max_ele;
```

二、代码

```
# intervals 形如 [[1,3],[2,6]...]
def merge(intervals):
    if not intervals: return []
    # 按区间的 start 升序排列
    intervals.sort(key=lambda intv: intv[0])
    res = []
    res.append(intervals[0])

    for i in range(1, len(intervals)):
        curr = intervals[i]
        # res 中最后一个元素的引用
        last = res[-1]
        if curr[0] <= last[1]:
            # 找到最大的 end
            last[1] = max(last[1], curr[1])
        else:
            # 处理下一个待合并区间
            res.append(curr)
    return res
```

看下动画就一目了然了：

【pdf/mobi格式不支持GIF:mergeInterval/3.gif】 请查看【关于本小抄及作者】章节的解决方案

至此，区间合并问题就解决了。本文篇幅短小，因为区间合并只是区间问题的一个类型，后续还有一些区间问题。本想把所有问题类型都总结在一篇文章，但有读者反应，长文只会收藏不会看... 所以还是分成小短文吧，读者有什么看法可以在留言板留言交流。

本文终，希望对你有帮助。

区间交集问题

本文是区间系列问题的第三篇，前两篇分别讲了区间的最大不相交子集和重叠区间的合并，今天再写一个算法，可以快速找出两组区间的交集。

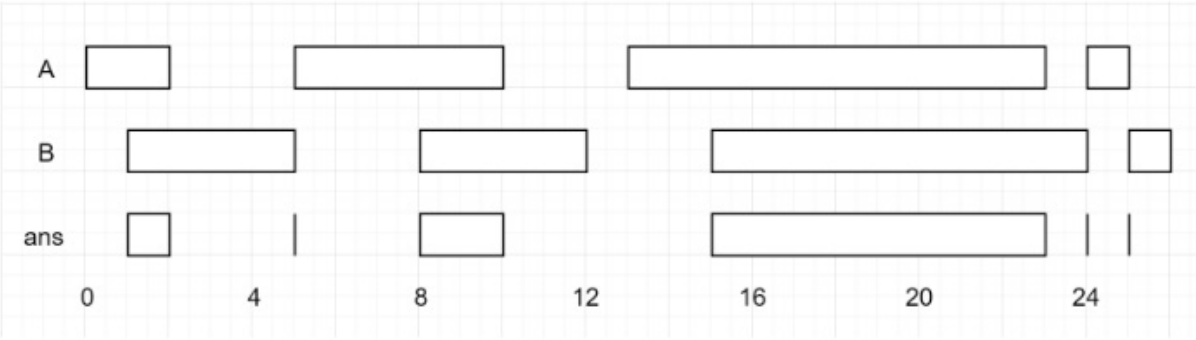
先看下题目，LeetCode 第 986 题就是这个问题：

给定两个由一些**闭区间**组成的列表，每个区间列表都是成对不相交的，并且已经排序。

返回这两个区间列表的交集。

（形式上，闭区间 `[a, b]`（其中 `a <= b`）表示实数 `x` 的集合，而 `a <= x <= b`。两个闭区间的交集是一组实数，要么为空集，要么为闭区间。例如，[1, 3] 和 [2, 4] 的交集为 [2, 3]。）

示例：



输入：A = [[0,2],[5,10],[13,23],[24,25]], B = [[1,5],[8,12],[15,24],[25,26]]
输出：[[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
注意：输入和所需的输出都是区间对象组成的列表，而不是数组或列表。

题目很好理解，就是让你找交集，注意区间都是闭区间。

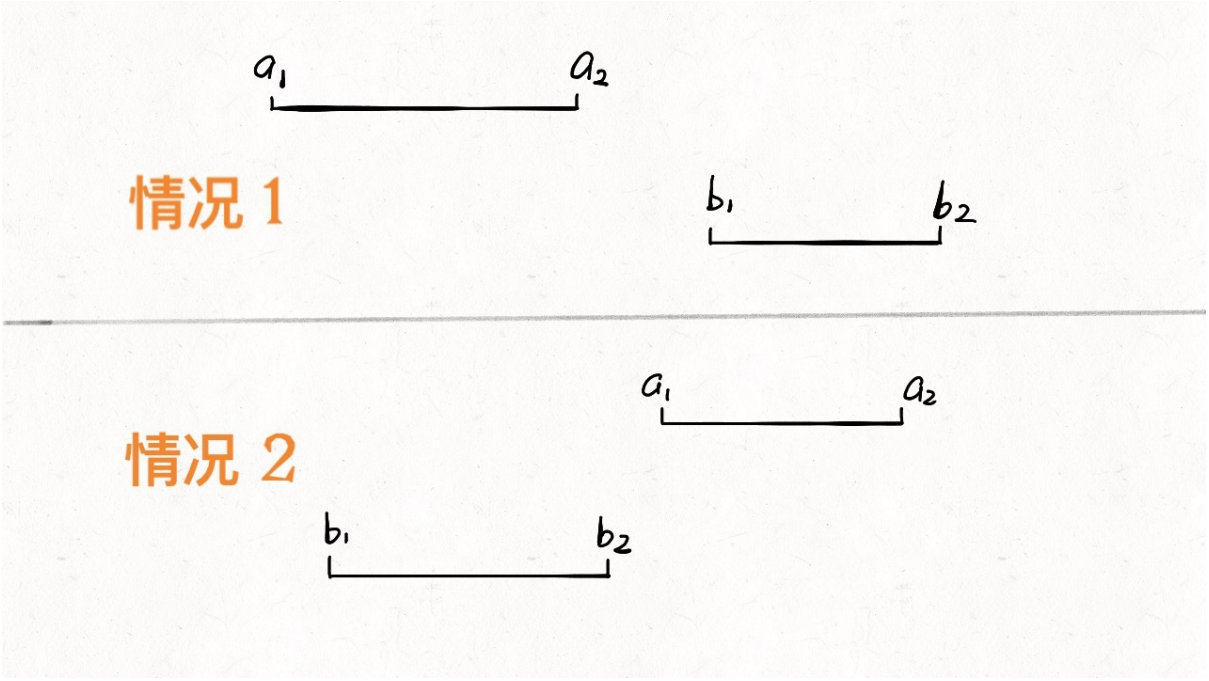
思路

解决区间问题的思路一般是先排序，以便操作，不过题目说已经排好序了，那么可以用两个索引指针在 `A` 和 `B` 中游走，把交集找出来，代码大概是这样的：

```
# A, B 形如 [[0,2],[5,10]...]
def intervalIntersection(A, B):
    i, j = 0, 0
    res = []
    while i < len(A) and j < len(B):
        # ...
        j += 1
        i += 1
    return res
```

不难，我们先老老实实分析一下各种情况。

首先，**对于两个区间**，我们用 `[a1,a2]` 和 `[b1,b2]` 表示在 `A` 和 `B` 中的两个区间，那么什么情况下这两个区间**没有交集**呢：



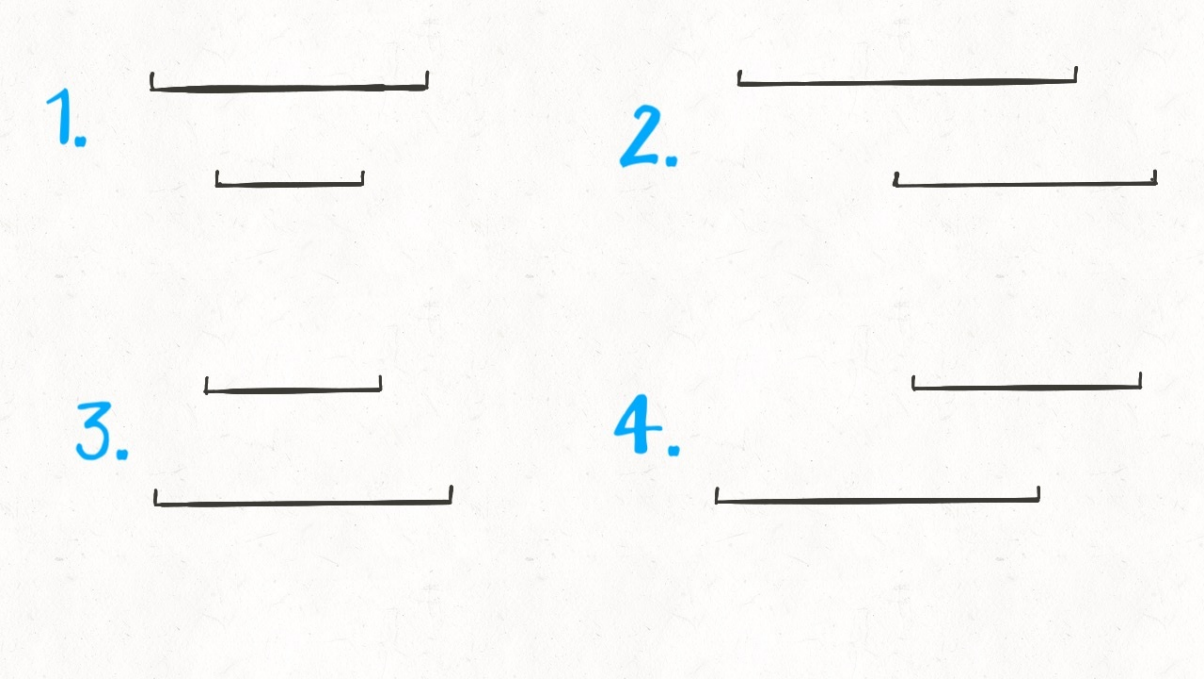
只有这两种情况，写成代码的条件判断就是这样：

```
if b2 < a1 or a2 < b1:
    [a1,a2] 和 [b1,b2] 无交集
```

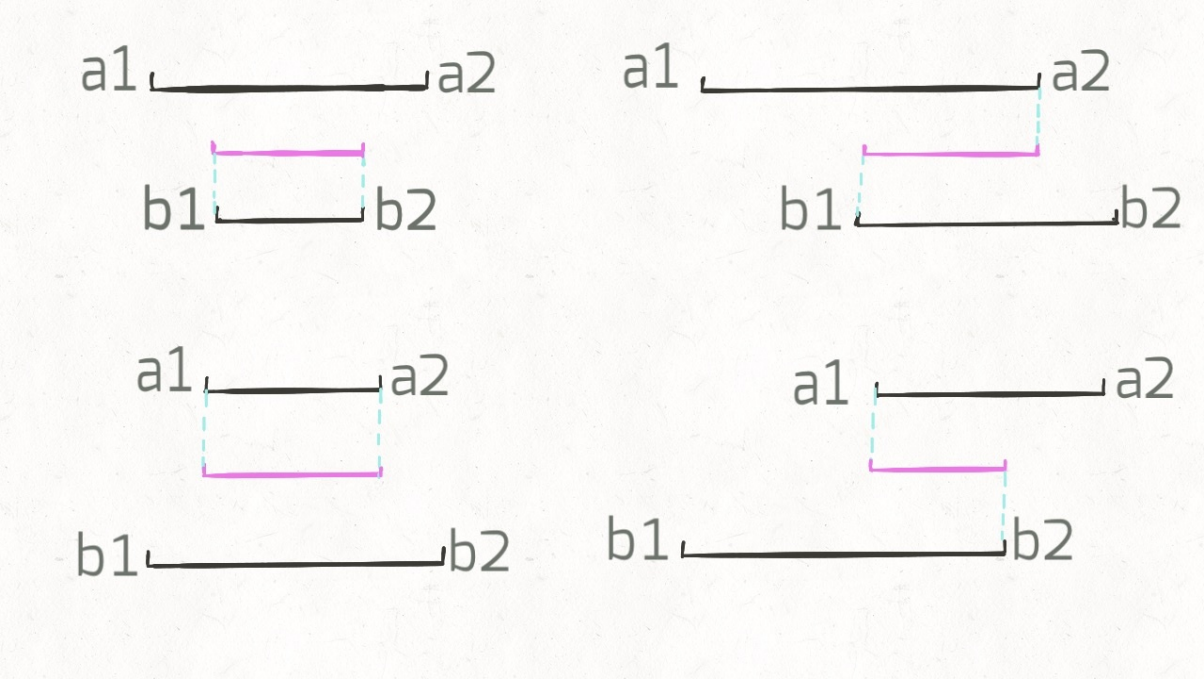
那么，什么情况下，两个区间存在交集呢？根据命题的否定，上面逻辑的否命题就是存在交集的条件：

```
# 不等号取反，or 也要变成 and
if b2 >= a1 and a2 >= b1:
    [a1,a2] 和 [b1,b2] 存在交集
```

接下来，两个区间存在交集的情况有哪些呢？穷举出来：



这很简单吧，就这四种情况而已。那么接下来思考，这几种情况下，交集是否有什么共同点呢？



我们惊奇地发现，交集区间是有规律的！如果交集区间是 `[c1,c2]`，那么 `c1=max(a1,b1)`，`c2=min(a2,b2)`！这一点就是寻找交集的核心，我们把代码更进一步：

```
while i < len(A) and j < len(B):
    a1, a2 = A[i][0], A[i][1]
    b1, b2 = B[j][0], B[j][1]
    if b2 >= a1 and a2 >= b1:
        res.append([max(a1, b1), min(a2, b2)])
    # ...
```

最后一步，我们的指针 `i` 和 `j` 肯定要前进（递增）的，什么时候应该前进呢？

【pdf/mobi格式不支持GIF:intersection/4.gif】 请查看【关于本小抄及作者】章节的解决方案

结合动画示例就很好理解了，是否前进，只取决于 `a2` 和 `b2` 的大小关系：

```
while i < len(A) and j < len(B):
    # ...
    if b2 < a2:
```

```
            j += 1
        else:
            i += 1
```

代码

```
# A, B 形如 [[0,2],[5,10]...]
def intervalIntersection(A, B):
    i, j = 0, 0 # 双指针
    res = []
    while i < len(A) and j < len(B):
        a1, a2 = A[i][0], A[i][1]
        b1, b2 = B[j][0], B[j][1]
        # 两个区间存在交集
        if b2 >= a1 and a2 >= b1:
            # 计算出交集，加入 res
            res.append([max(a1, b1), min(a2, b2)])
        # 指针前进
        if b2 < a2: j += 1
        else:       i += 1
    return res
```

总结一下，区间类问题看起来都比较复杂，情况很多难以处理，但实际上通过观察各种不同情况之间的共性可以发现规律，用简洁的代码就能处理。

另外，区间问题没啥特别厉害的奇技淫巧，其操作也朴实无华，但其应用却十分广泛，接之前的几篇文章：

信封嵌套问题

很多算法问题都需要排序技巧，其难点不在于排序本身，而是需要巧妙地排序进行预处理，将算法问题进行转换，为之后的操作打下基础。

信封嵌套问题就需要先按特定的规则排序，之后就转换为一个 最长递增子序列问题，可以用前文 二分查找详解 的技巧来解决了。

一、题目概述

信封嵌套问题是个很有意思且经常出现在生活中的问题，先看下题目：

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

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

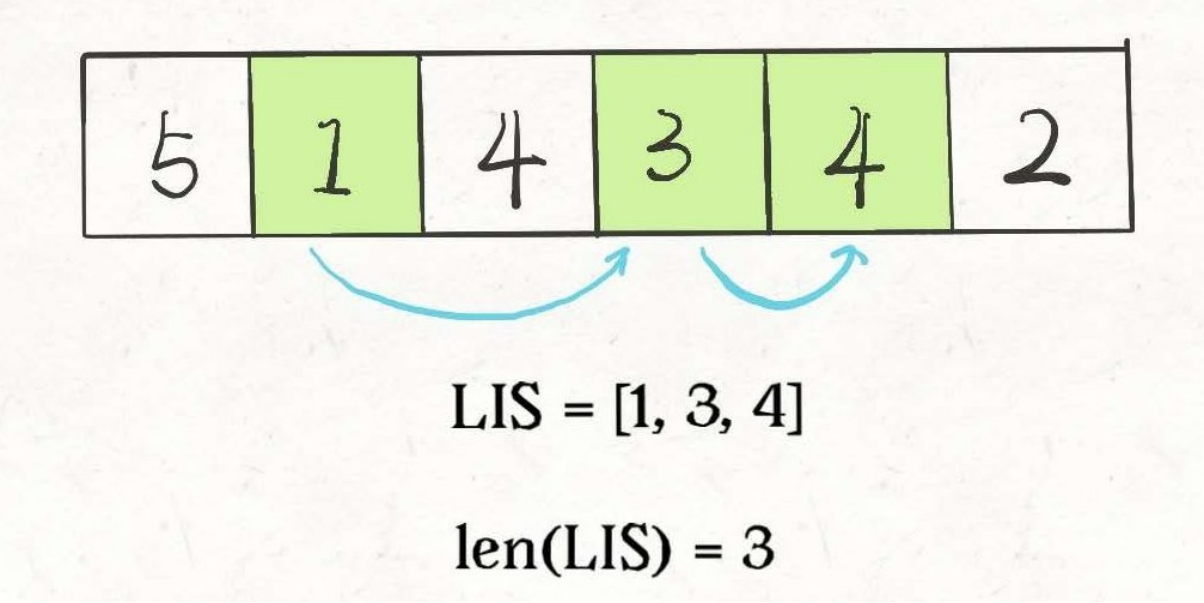
说明:
不允许旋转信封。

示例:

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

这道题目其实是最长递增子序列（Longes Increasing Subsequence，简写为 LIS）的一个变种，因为很显然，每次合法的嵌套是大的套小的，相当于找一个最长递增的子序列，其长度就是最多能嵌套的信封个数。

但是难点在于，标准的 LIS 算法只能在数组中寻找最长子序列，而我们的信封是由 `(w, h)` 这样的二维数对形式表示的，如何把 LIS 算法运用过来呢？



读者也许会想，通过 `w × h` 计算面积，然后对面积进行标准的 LIS 算法。但是稍加思考就会发现这样不行，比如 `1 × 10` 大于 `3 × 3`，但是显然这样的两个信封是无法互相嵌套的。

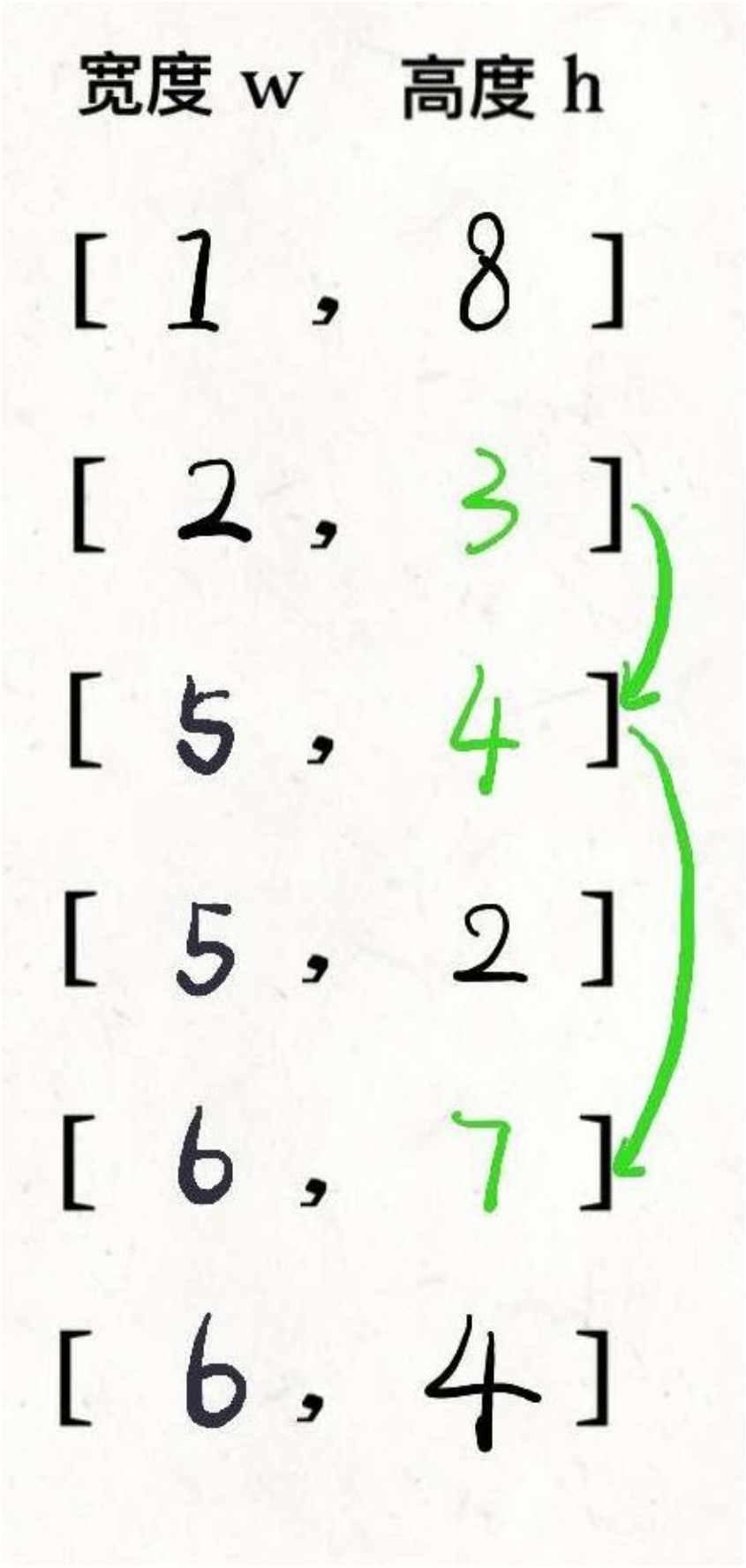
二、解法

这道题的解法是比较巧妙的：

先对宽度 `w` 进行升序排序，如果遇到 `w` 相同的情况，则按照高度 `h` 降序排序。之后把所有的 `h` 作为一个数组，在这个数组上计算 LIS 的长度就是答案。

画个图理解一下，先对这些数对进行排序：

然后在 `h` 上寻找最长递增子序列：



这个子序列就是最优的嵌套方案。

这个解法的关键在于，对于宽度 `w` 相同的数对，要对其高度 `h` 进行降序排序。因为两个宽度相同的信封不能相互包含的，逆序排序保证在 `w` 相同的数对中最多只选取一个。

下面看代码：

```
// envelopes = [[w, h], [w, h]...]
public int maxEnvelopes(int[][] envelopes) {
    int n = envelopes.length;
    // 按宽度升序排列，如果宽度一样，则按高度降序排列
    Arrays.sort(envelopes, new Comparator<int[]>()
    {
        public int compare(int[] a, int[] b) {
            return a[0] == b[0] ?
                b[1] - a[1] : a[0] - b[0];
        }
    });
    // 对高度数组寻找 LIS
    int[] height = new int[n];
    for (int i = 0; i < n; i++)
        height[i] = envelopes[i][1];

    return lengthOfLIS(height);
}
```

关于最长递增子序列的寻找方法，在前文中详细介绍了动态规划解法,并用扑克牌游戏解释了二分查找解法，本文就不展开了，直接套用算法模板：

```
/* 返回 nums 中 LIS 的长度 */
public int lengthOfLIS(int[] nums) {
    int piles = 0, n = nums.length;
    int[] top = new int[n];
    for (int i = 0; i < n; i++) {
        // 要处理的扑克牌
        int poker = nums[i];
        int left = 0, right = piles;
        // 二分查找插入位置
        while (left < right) {
            int mid = (left + right) / 2;
            if (top[mid] >= poker)
```

```
                right = mid;
            else
                left = mid + 1;
        }
        if (left == piles) piles++;
        // 把这张牌放到牌堆顶
        top[left] = poker;
    }
    // 牌堆数就是 LIS 长度
    return piles;
}
```

为了清晰，我将代码分为了两个函数， 你也可以合并，这样可以节省下 `height` 数组的空间。

此算法的时间复杂度为 $O(NlogN)$，因为排序和计算 LIS 各需要 $O(NlogN)$ 的时间。

空间复杂度为 $O(N)$，因为计算 LIS 的函数中需要一个 `top` 数组。

三、总结

这个问题是个 Hard 级别的题目，难就难在排序，正确地排序后此问题就被转化成了一个标准的 LIS 问题，容易解决一些。

其实这种问题还可以拓展到三维，比如说现在不是让你嵌套信封，而是嵌套箱子，每个箱子有长宽高三个维度，请你算算最多能嵌套几个箱子？

我们可能会这样想，先把前两个维度（长和宽）按信封嵌套的思路求一个嵌套序列，最后在这个序列的第三个维度（高度）找一下 LIS，应该能算出答案。

实际上，这个思路是错误的。这类问题叫做「偏序问题」，上升到三维会使难度巨幅提升，需要借助一种高级数据结构「树状数组」，有兴趣的读者可以自行搜索。

有很多算法问题都需要排序后进行处理，阿东正在进行整理总结。希望本文对你有帮助。

几个反直觉的概率问题

上篇文章 洗牌算法详解 讲到了验证概率算法的蒙特卡罗方法，今天聊点轻松的内容：几个和概率相关的有趣问题。

计算概率有下面两个最简单的原则：

原则一、计算概率一定要有一个参照系，称作「样本空间」，即随机事件可能出现的所有结果。事件 A 发生的概率 = A 包含的样本点 / 样本空间的样本总数。

原则二、计算概率一定要明白，概率是一个连续的整体，不可以把连续的概率分割开，也就是所谓的条件概率。

上述两个原则高中就学过，但是我们还是很容易犯错，而且犯错的流程也有异曲同工之妙：

先是忽略了原则二，错误地计算了样本空间，然后通过原则一算出了错误的答案。

下面介绍几个简单却具有迷惑性的问题，分别是男孩女孩问题、生日悖论、三门问题。当然，三门问题可能是大家最耳熟的，所以就多说一些有趣的思考。

一、男孩女孩问题

假设有一个家庭，有两个孩子，现在告诉你其中有一个男孩，请问另一个也是男孩的概率是多少？

很多人，包括我在内，不假思索地回答：1/2 啊，因为另一个孩子要么是男孩，要么是女孩，而且概率相等呀。但是实际上，答案是 1/3。

上述思想为什么错误呢？因为没有正确计算样本空间，导致原则一计算错误。有两个孩子，那么样本空间为 4，即哥哥妹妹，哥哥弟弟，姐姐妹妹，姐姐弟弟这四种情况。已知有一个男孩，那么排除姐姐妹妹这种情况，所以样本空间变成 3。另一个孩子也是男孩只有哥哥弟弟这 1 种情况，所以概率为 1/3。

为什么计算样本空间会出错呢？因为我们忽略了条件概率，即混淆了下面两个问题：

这个家庭只有一个孩子，这个孩子是男孩的概率是多少？

这个家庭有两个孩子，其中一个是男孩，另一个孩子是男孩的概率是多少？

根据原则二，概率问题是连续的，不可以把上述两个问题混淆。第二个问题需要用条件概率，即求一个孩子是男孩的条件下，另一个也是男孩的概率。运用条件概率的公式也很好算，就不多说了。

通过这个问题，读者应该理解两个概率计算原则的关系了，最具有迷惑性的就是条件概率的忽视。为了不要被迷惑，最简单的办法就是把所有可能结果穷举出来。

最后，对于此问题我见过一个很奇葩的质疑：如果这两个孩子是双胞胎，不存在年龄上的差异怎么办？

我竟然觉得有那么一丝道理！但其实，我们只是通过年龄差异来表示两个孩子的独立性，也就是说即便两个孩子同性，也有两种可能。所以不要用双胞胎抬杠了。

二、生日悖论

生日悖论是由这样一个问题引出的：一个屋子里需要有多少人，才能使得存在至少两个人生日是同一天的概率达到 50%？

答案是 23 个人，也就是说房子里如果有 23 个人，那么就有 50% 的概率会存在两个人生日相同。这个结论看起来不可思议，所以被称为悖论。按照直觉，要得到 50% 的概率，起码得有 183 个人吧，因为一年有 365 天呀？其

实不是的，觉得这个结论不可思议主要有两个思维误区：

第一个误区是误解「存在」这个词的含义。

读者可能认为，如果 23 个人中出现相同生日的概率就能达到 50%，是不是意味着：

假设现在屋子里坐着 22 个人，然后我走进去，那么有 50% 的概率我可以找到一个人和我生日相同。但这怎么可能呢？

并不是的，你这种想法是以自我为中心，而题目的概率是在描述整体。也就是说「存在」的含义是指 23 人中的任意两个人，涉及排列组合，大概率和你没啥关系。

如果你非要计算存在和自己生日相同的人的概率是多少，可以这样计算：

1 - P(22 个人都和我的生日不同) = 1 -(364/365)^22 = 0.06

这样计算得到的结果是不是看起来合理多了？生日悖论计算对象的不是某一个人，而是一个整体，其中包含了所有人的排列组合，它们的概率之和当然会大得多。

第二个误区是认为概率是线性变化的。

读者可能认为，如果 23 个人中出现相同生日的概率就能达到 50%，是不是意味着 46 个人的概率就能达到 100%？

不是的，就像中奖率 50% 的游戏，你玩两次的中奖率就是 100% 吗？显然不是，你玩两次的中奖率是 75%：

$P(两次能中奖) = P(第一次就中了) + P(第一次没中但第二次中了) = 1/2 + 1/2*1/2 = 75\%$

那么换到生日悖论也是一个道理，概率不是简单叠加，而要考虑一个连续的过程，所以这个结论并没有什么不合常理之处。

那为什么只要 23 个人出现相同生日的概率就能大于 50% 了呢？我们先计算 23 个人生日都唯一（不重复）的概率。只有 1 个人的时候，生日唯一的概率是 $365/365$，2 个人时，生日唯一的概率是 $365/365 \times 364/365$，以此类推可知 23 人的生日都唯一的概率：

给你一个链表，每 k 个节点一组进行翻转，请你返回翻转后的链表。

k 是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

示例：

给定这个链表：`1->2->3->4->5`

当 $k = 2$ 时，应当返回: `2->1->4->3->5`

当 $k = 3$ 时，应当返回: `3->2->1->4->5`

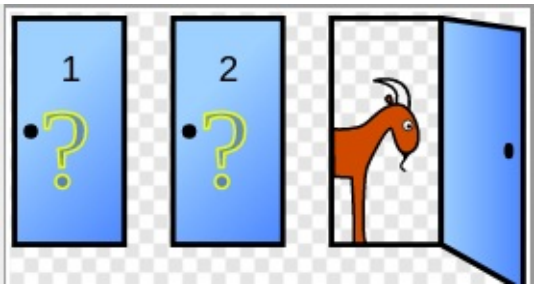
算出来大约是 0.493，所以存在相同生日的概率就是 0.507，差不多就是 50% 了。实际上，按照这个算法，当人数达到 70 时，存在两个人生日相同的概率就上升到了 99.9%，基本可以认为是 100% 了。所以从概率上说，一个几十人的小团体中存在生日相同的人真没啥稀奇的。

三、三门问题

这个游戏很经典了：游戏参与者面对三扇门，其中两扇门后面是山羊，一扇门后面是跑车。参与者只要随便选一扇门，门后面的东西就归他（跑车的价值当然更大）。但是主持人决定帮一下参与者：在他选择之后，先不急着打开这扇门，而是由主持人打开剩下两扇门中的一扇，展示其中的山羊（主持人知道每扇门后面是什么），然后给参与者一次换门的机会，此时参与者应该换门还是不换门呢？

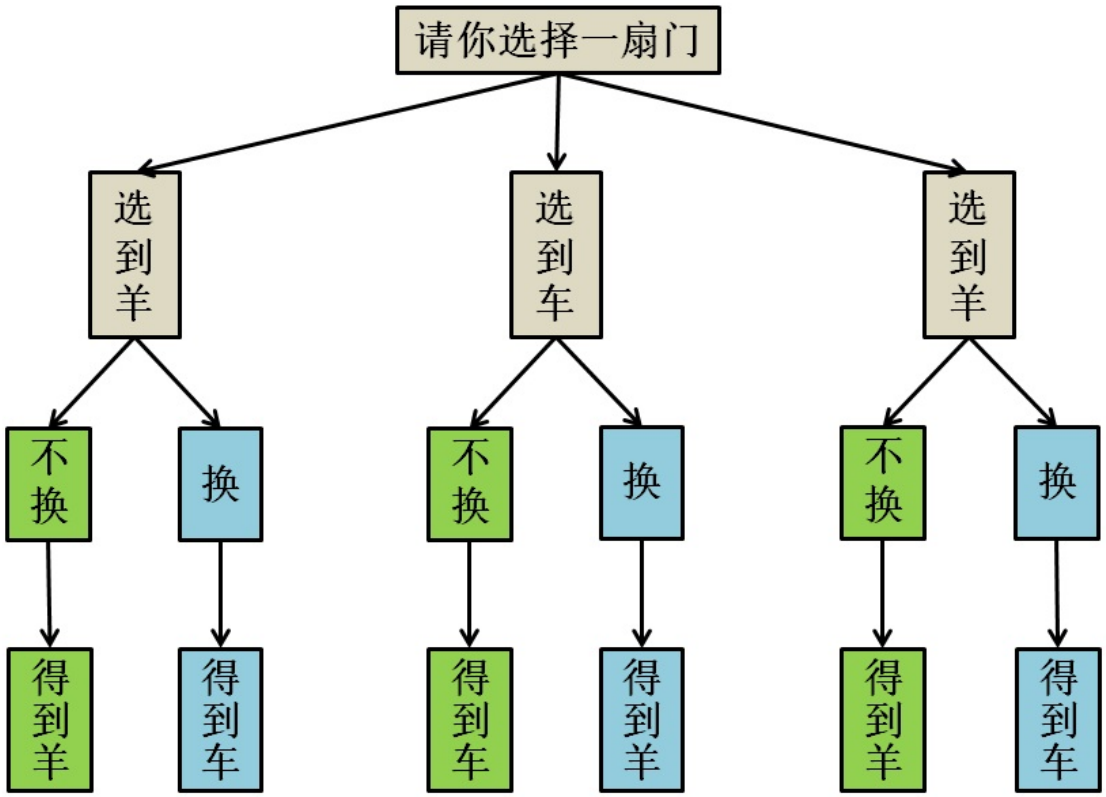
为了防止第一次看到这个问题的读者迷惑，再具体描述一下这个问题：

你是游戏参与者，现在有门 1,2,3，假设你随机选择了门 1，然后主持人打开了门 3 告诉你那后面是山羊。现在，你是坚持你最初的选择门 1，还是选择换成门 2 呢？



答案是应该换门，换门之后抽到跑车的概率是 2/3，不换的话是 1/3。又一次反直觉，感觉换不换的中奖概率应该都一样啊，因为最后肯定就剩两个门，一个是羊，一个是跑车，这是事实，所以不管选哪个的概率不都是 1/2 吗？

类似前面说的男孩女孩问题，最简单稳妥的方法就是把所有可能结果穷举出来：



很容易看到选择换门中奖的概率是 2/3，不换的话是 1/3。

关于这个问题还有更简单的方法：主持人开门实际上在「浓缩」概率。一开始你选择到跑车的概率当然是 1/3，剩下两个门中包含跑车的概率当然是 2/3，这没啥可说的。但是主持人帮你排除了一个含有山羊的门，相当于把那 2/3 的概率浓缩到了剩下的这一扇门上。那么，你说你是抱着原来那扇 1/3 的门，还是换成那扇经过「浓缩」的 2/3 概率的门呢？

再直观一点，假设你三选一，剩下 2 扇门，再给你加入 98 扇装山羊的门，把这 100 扇门随机打乱，问你换不换？肯定不换对吧，这明摆着把概率稀释了，肯定抱着原来的那扇门是最可能中跑车的。再假设，初始有 100 扇门，你选了一扇，然后主持人在剩下 99 扇门中帮你排除 98 个山羊，问你换不换一扇门？肯定换对吧，你手上那扇门是 1%，另一扇门是 99%，或者也可以这样理解，不换只是选择了 1 扇门，换门相当于选择了 99 扇门，这样结果很明显了吧？

以上思想，也许有的读者都思考过，下面我们思考这样一个问题：假设你在决定是否换门的时候，小明破门而入，要求帮你做出选择。他完全不知道之前发生的事，他只知道面前有两扇门，一扇是跑车一扇是山羊，那么他抽中跑车的概率是多大？

当然是 1/2，这也是很多人做错三门问题的根本原因。类似生日悖论，人们总是容易以自我为中心，通过这个小明的视角来计算是否换门，这显然会进入误区。

就好比有两个箱子，一号箱子有 4 个黑球 2 个红球，二号箱子有 2 个黑球 4 个红球，随便选一个箱子，随便摸一个球，问你摸出红球的概率。

对于不知情的小明，他会随机选择一个箱子，随机摸球，摸到红球的概率是：$1/2 \times 2/6 + 1/2 \times 4/6 = 1/2$

对于知情的你，你知道在二号箱子摸球概率大，所以只在二号箱摸，摸到红球的概率是：$0 \times 2/6 + 1 \times 4/6 = 2/3$

三门问题是有指导意义的。比如你蒙选择题，先蒙了 A，后来灵机一动排除了 B 和 C，请问你是否要把 A 换成 D？答案是，换！

也许读者会问，如果只排除了一个答案，比如说 B，那么我是否应该把 A 换成 C 或者 D 呢？答案是，换！

因为按照刚才「浓缩」概率这个思想，只要进行了排除，都是在进行「浓缩」，均摊下来肯定比你一开始蒙的那个答案概率 1/4 高。比如刚才的例子，C 和 D 的正确概率都是 3/8，而你开始蒙的 A 只有 1/4。

当然，运用此策略蒙题的前提是你真的抓瞎，真的随机乱选答案，这样概率才能作为最后的杀手锏。

洗牌算法

我知道大家会各种花式排序算法，但是如果叫你打乱一个数组，你是否能做到胸有成竹？即便你拍脑袋想出一个算法，怎么证明你的算法就是正确的呢？乱序算法不像排序算法，结果唯一可以很容易检验，因为「乱」可以有很多种，你怎么能证明你的算法是「真的乱」呢？

所以我们面临两个问题：

1. 什么叫做「真的乱」？
2. 设计怎样的算法来打乱数组才能做到「真的乱」？

这种算法称为「随机乱置算法」或者「洗牌算法」。

本文分两部分，第一部分详解最常用的洗牌算法。因为该算法的细节容易出错，且存在好几种变体，虽有细微差异但都是正确的，所以本文要介绍一种简单的通用思想保证你写出正确的洗牌算法。第二部分讲解使用「蒙特卡罗方法」来检验我们的打乱结果是不是真的乱。蒙特卡罗方法的思想不难，但是实现方式也各有特点的。

一、洗牌算法

此类算法都是靠随机选取元素交换来获取随机性，直接看代码（伪码），该算法有 4 种形式，都是正确的：

```
// 得到一个在闭区间 [min, max] 内的随机整数
int randInt(int min, int max);

// 第一种写法
void shuffle(int[] arr) {
    int n = arr.length();
    /******** 区别只有这两行 ********/
    for (int i = 0 ; i < n; i++) {
        // 从 i 到最后随机选一个元素
        int rand = randInt(i, n - 1);
```

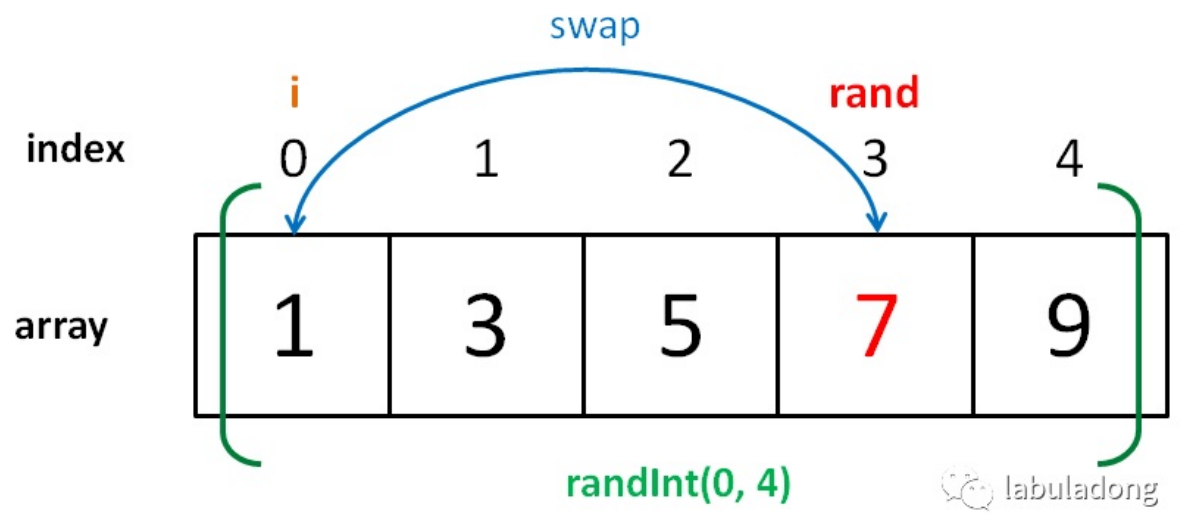
```
        /************************/
        swap(arr[i], arr[rand]);
    }
}

// 第二种写法
    for (int i = 0 ; i < n - 1; i++)
        int rand = randInt(i, n - 1);

// 第三种写法
    for (int i = n - 1 ; i >= 0; i--)
        int rand = randInt(0, i);

// 第四种写法
    for (int i = n - 1 ; i > 0; i--)
        int rand = randInt(0, i);
```

分析洗牌算法正确性的准则：产生的结果必须有 n! 种可能，否则就是错误的。这个很好解释，因为一个长度为 n 的数组的全排列就有 n! 种，也就是说打乱结果总共有 n! 种。算法必须能够反映这个事实，才是正确的。

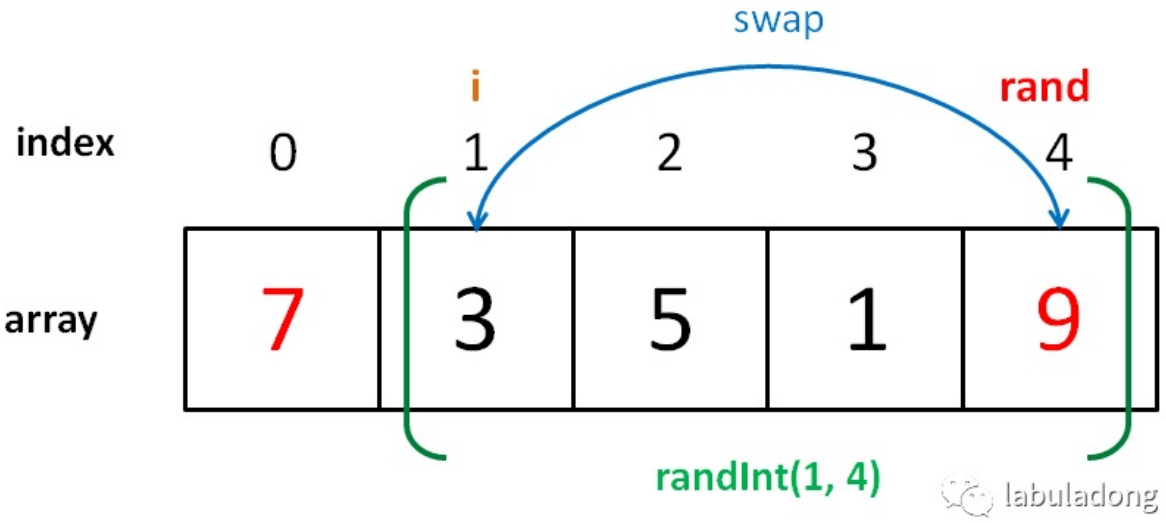
我们先用这个准则分析一下**第一种写法**的正确性：

```
// 假设传入这样一个 arr
int[] arr = {1,3,5,7,9};

void shuffle(int[] arr) {
    int n = arr.length(); // 5
    for (int i = 0 ; i < n; i++) {
        int rand = randInt(i, n - 1);
        swap(arr[i], arr[rand]);
    }
}
```

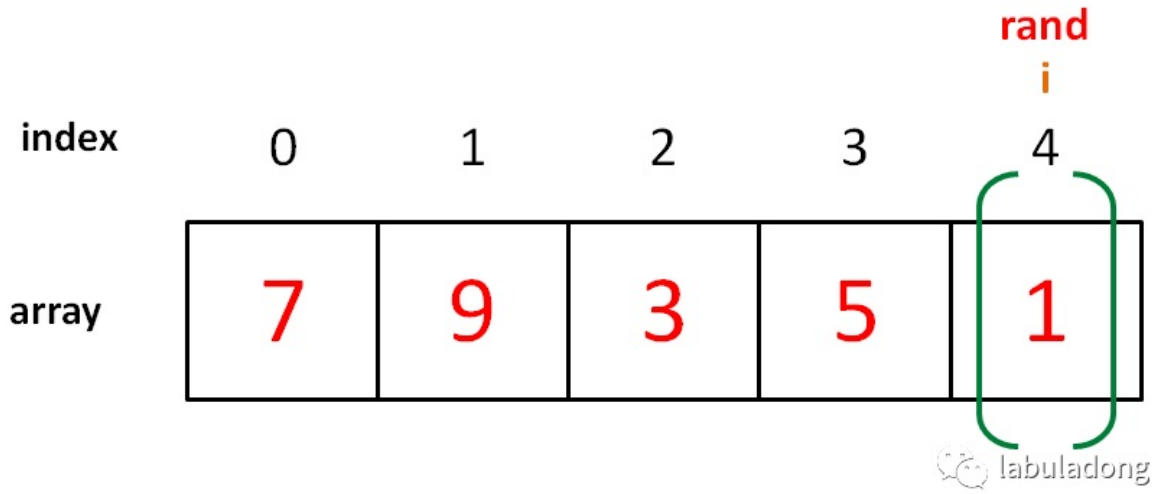
for 循环第一轮迭代时，`i = 0`，`rand` 的取值范围是 `[0, 4]`，有 5 个可能的取值。



for 循环第二轮迭代时，`i = 1`，`rand` 的取值范围是 `[1, 4]`，有 4 个可能的取值。



后面以此类推，直到最后一次迭代，`i = 4`，`rand` 的取值范围是 `[4, 4]`，只有 1 个可能的取值。



可以看到，整个过程产生的所有可能结果有 `n! = 5! = 5*4*3*2*1` 种，所以这个算法是正确的。

分析**第二种写法**，前面的迭代都是一样的，少了一次迭代而已。所以最后一次迭代时 `i = 3`，`rand` 的取值范围是 `[3, 4]`，有 2 个可能的取值。

```
// 第二种写法
// arr = {1,3,5,7,9}, n = 5
    for (int i = 0 ; i < n - 1; i++)
        int rand = randInt(i, n - 1);
```

所以整个过程产生的所有可能结果仍然有 `5*4*3*2 = 5! = n!` 种，因为乘以 1 可有可无嘛。所以这种写法也是正确的。

如果以上内容你都能理解，那么你就能发现**第三种写法**就是第一种写法，只是将数组从后往前迭代而已；**第四种写法**是第二种写法从后往前来。所以它们都是正确的。

如果读者思考过洗牌算法，可能会想出如下的算法，但是**这种写法是错误的**：

```
void shuffle(int[] arr) {
    int n = arr.length();
    for (int i = 0 ; i < n; i++) {
        // 每次都从闭区间 [0, n-1]
```

```
            // 中随机选取元素进行交换
            int rand = randInt(0, n - 1);
            swap(arr[i], arr[rand]);
        }
    }
```

现在你应该明白这种写法为什么会错误了。因为这种写法得到的所有可能结果有 n^n 种，而不是 $n!$ 种，而且 n^n 不可能是 $n!$ 的整数倍。

比如说 `arr = {1,2,3}`，正确的结果应该有 $3!= 6$ 种可能，而这种写法总共有 $3^3 = 27$ 种可能结果。因为 27 不能被 6 整除，所以一定有某些情况被「偏袒」了，也就是说某些情况出现的概率会大一些，所以这种打乱结果不算「真的乱」。

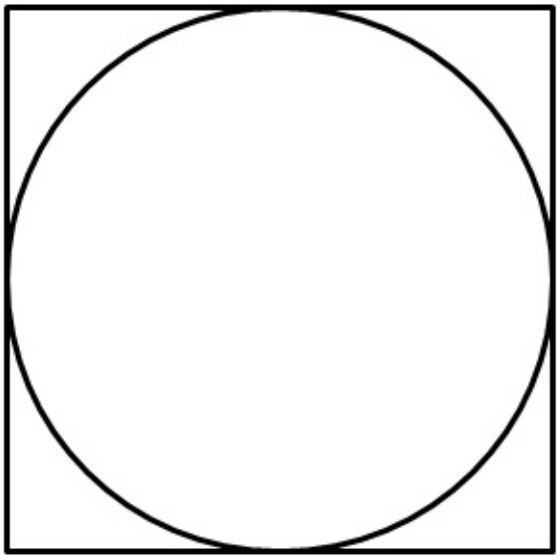
上面我们从直觉上简单解释了洗牌算法正确的准则，没有数学证明，我想大家也懒得证明。对于概率问题我们可以使用「蒙特卡罗方法」进行简单验证。

二、蒙特卡罗方法验证正确性

洗牌算法，或者说随机乱置算法的**正确性衡量标准是：对于每种可能的结果出现的概率必须相等，也就是说要足够随机**。

如果不用数学严格证明概率相等，可以用蒙特卡罗方法近似地估计出概率是否相等，结果是否足够随机。

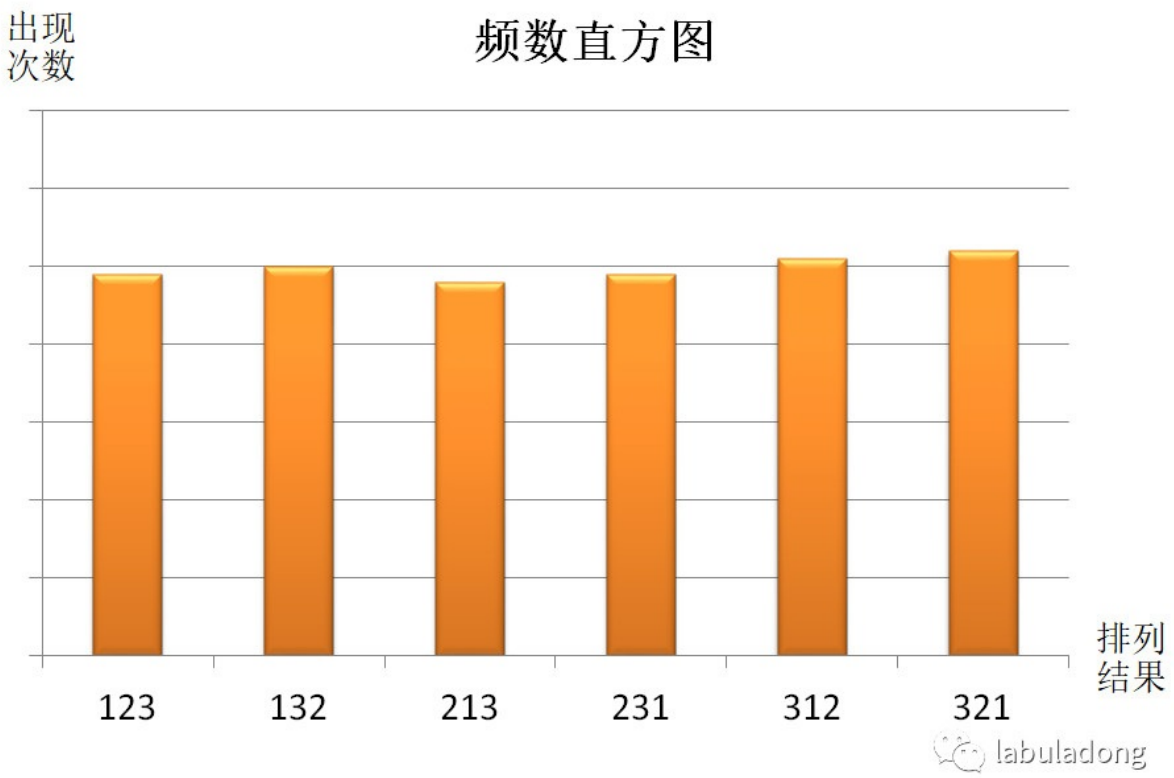
记得高中有道数学题：往一个正方形里面随机打点，这个正方形里紧贴着一个圆，告诉你打点的总数和落在圆里的点的数量，让你计算圆周率。



这其实就是利用了蒙特卡罗方法：当打的点足够多的时候，点的数量就可以近似代表图形的面积。通过面积公式，由正方形和圆的面积比值是可以很容易推出圆周率的。当然打的点越多，算出的圆周率越准确，充分体现了大力出奇迹的真理。

类似的，我们可以对同一个数组进行一百万次洗牌，统计各种结果出现的次数，把频率作为概率，可以很容易看出洗牌算法是否正确。整体思想很简单，不过实现起来也有些技巧的，下面简单分析几种实现思路。

第一种思路，我们把数组 arr 的所有排列组合都列举出来，做成一个直方图（假设 arr = {1,2,3}）：



每次进行洗牌算法后，就把得到的打乱结果对应的频数加一，重复进行 100 万次，如果每种结果出现的总次数差不多，那就说明每种结果出现的概率应该是相等的。写一下这个思路的伪代码：

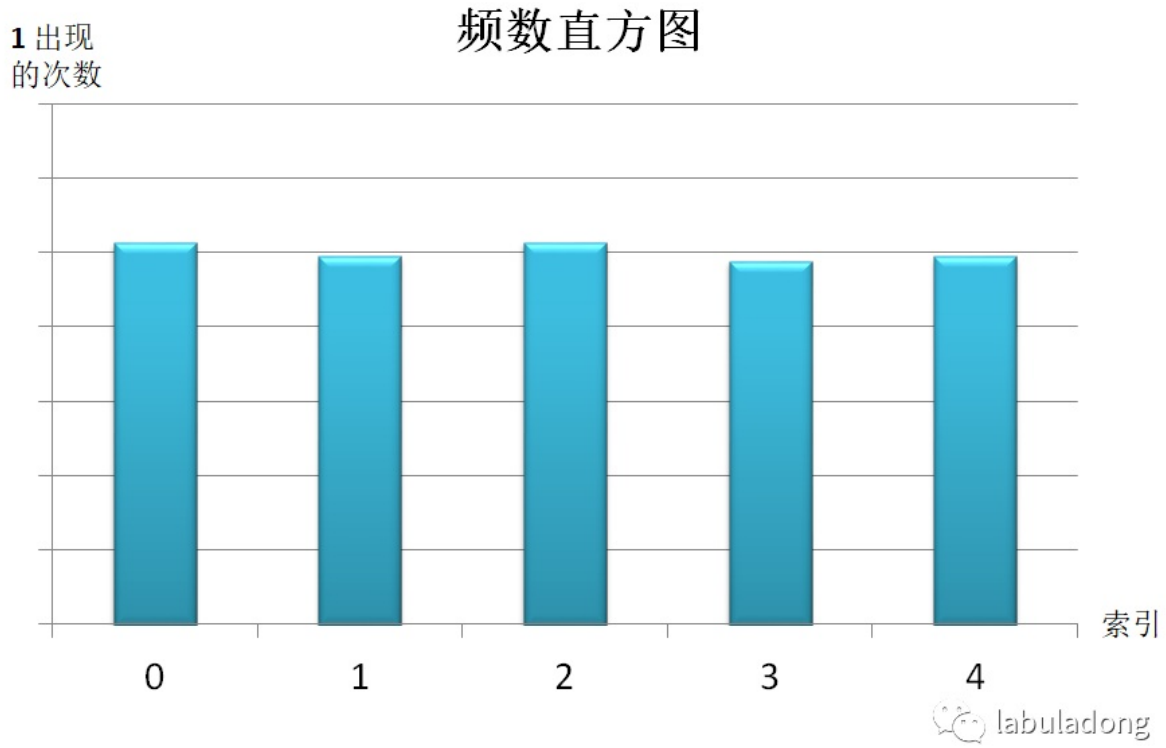
```
void shuffle(int[] arr);

// 蒙特卡罗
int N = 1000000;
HashMap count; // 作为直方图
for (i = 0; i < N; i++) {
    int[] arr = {1,2,3};
    shuffle(arr);
    // 此时 arr 已被打乱
    count[arr] += 1;
}
for (int feq : count.values())
    print(feq / N + " "); // 频率
```

这种检验方案是可行的，不过可能有读者会问，arr 的全部排列有 n! 种（n 为 arr 的长度），如果 n 比较大，那岂不是空间复杂度爆炸了？

是的，不过作为一种验证方法，我们不需要 n 太大，一般用长度为 5 或 6 的 arr 试下就差不多了吧，因为我们只想比较概率验证一下正确性而已。

第二种思路，可以这样想，arr 数组中全都是 0，只有一个 1。我们对 arr 进行 100 万次打乱，记录每个索引位置出现 1 的次数，如果每个索引出现的次数差不多，也可以说明每种打乱结果的概率是相等的。

```
void shuffle(int[] arr);

// 蒙特卡罗方法
int N = 1000000;
int[] arr = {1,0,0,0,0};
int[] count = new int[arr.length];
for (int i = 0; i < N; i++) {
    shuffle(arr); // 打乱 arr
    for (int j = 0; j < arr.length; j++)
        if (arr[j] == 1) {
            count[j]++;
            break;
        }
}
for (int feq : count)
    print(feq / N + " "); // 频率
```



这种思路也是可行的，而且避免了阶乘级的空间复杂度，但是多了嵌套 for 循环，时间复杂度高一点。不过由于我们的测试数据量不会有多大，这些问题都可以忽略。

另外，细心的读者可能发现一个问题，上述两种思路声明 arr 的位置不同，一个在 for 循环里，一个在 for 循环之外。其实效果都是一样的，因为我们的算法总要打乱 arr，所以 arr 的顺序并不重要，只要元素不变就行。

三、最后总结

本文第一部分介绍了洗牌算法（随机乱置算法），通过一个简单的分析技巧证明了该算法的四种正确形式，并且分析了一种常见的错误写法，相信你一定能够写出正确的洗牌算法了。

第二部分写了洗牌算法正确性的衡量标准，即每种随机结果出现的概率必须相等。如果我们不用严格的数学证明，可以通过蒙特卡罗方法大力出奇迹，粗略验证算法的正确性。蒙特卡罗方法也有不同的思路，不过要求不必太严格，因为我们只是寻求一个简单的验证。

递归详解

首先说明一个问题，简单阐述一下递归，分治算法，动态规划，贪心算法这几个东西的区别和联系，心里有个印象就好。

递归是一种编程技巧，一种解决问题的思维方式；分治算法和动态规划很大程度上是递归思想基础上的（虽然动态规划的最终版本大都不是递归了，但解题思想还是离不开递归），解决更具体问题的两类算法思想；贪心算法是动态规划算法的一个子集，可以更高效解决一部分更特殊的问题。

分治算法将在这节讲解，以最经典的归并排序为例，它把待排序数组不断二分为规模更小的子问题处理，这就是 “分而治之” 这个词的由来。显然，排序问题分解出的子问题是不重复的，如果有的问题分解后的子问题有重复的（重叠子问题性质），那么就交给动态规划算法去解决！

递归详解

介绍分治之前，首先要弄清楚递归这个概念。

递归的基本思想是某个函数直接或者间接地调用自身，这样就把原问题的求解转换为许多性质相同但是规模更小的子问题。我们只需要关注如何把原问题划分成符合条件的子问题，而不需要去研究这个子问题是如何被解决的。递归和枚举的区别在于：枚举是横向地把问题划分，然后依次求解子问题，而递归是把问题逐级分解，是纵向的拆分。

以下会举例说明我对递归的一点理解，**如果你不想看下去了，请记住这几个问题怎么回答：**

1. 如何给一堆数字排序？ 答：分成两半，先排左半边再排右半边，最后合并就行了，至于怎么排左边和右边，请重新阅读这句话。
2. 孙悟空身上有多少根毛？ 答：一根毛加剩下的毛。
3. 你今年几岁？ 答：去年的岁数加一岁,1999 年我出生。

递归代码最重要的两个特征：结束条件和自我调用。自我调用是在解决子问题，而结束条件定义了最简子问题的答案。

```
int func(你今年几岁) {
    // 最简子问题，结束条件
    if (你1999年几岁) return 我0岁;
    // 自我调用，缩小规模
    return func(你去年几岁) + 1;
}
```

其实仔细想想，**递归运用最成功的是什么？我认为是数学归纳法**。我们高中都学过数学归纳法，使用场景大概是：我们推不出来某个求和公式，但是我们试了几个比较小的数，似乎发现了一点规律，然后编了一个公式，看起来应该是正确答案。但是数学是很严谨的，你哪怕穷举了一万个数都是正确的，但是第一万零一个数正确吗？这就要数学归纳法发挥神威了，可以假设我们编的这个公式在第 k 个数时成立，如果证明在第 k + 1 时也成立，那么我们编的这个公式就是正确的。

那么数学归纳法和递归有什么联系？我们刚才说了，递归代码必须要有结束条件，如果没有的话就会进入无穷无尽的自我调用，直到内存耗尽。而数学证明的难度在于，你可以尝试有穷种情况，但是难以将你的结论延伸到无穷大。这里就可以看出联系了 —— 无穷。

递归代码的精髓在于调用自己去解决规模更小的子问题，直到到达结束条件；而数学归纳法之所以有用，就在于不断把我们的猜测向上加一，扩大结论的规模，没有结束条件，从而把结论延伸到无穷无尽，也就完成了猜测正确性的证明。

为什么要写递归

首先为了训练逆向思考的能力。递推的思维是正常人的思维，总是看着眼前的问题思考对策，解决问题是将来时；递归的思维，逼迫我们倒着思考，看到问题的尽头，把解决问题的过程看做过去时。

第二，练习分析问题的结构，当问题可以被分解成相同结构的小问题时，你能敏锐发现这个特点，进而高效解决问题。

第三，跳出细节，从整体上看问题。再说说归并排序，其实可以不用递归来划分左右区域的，但是代价就是代码极其难以理解，大概看一下代码（归并排序在后面讲，这里大致看懂意思就行，体会递归的妙处）：

```
void sort(Comparable[] a){
    int N = a.length;
    // 这么复杂，是对排序的不尊重。我拒绝研究这样的代码。
    for (int sz = 1; sz < N; sz = sz + sz)
        for (int lo = 0; lo < N - sz; lo += sz + sz)
            merge(a, lo, lo + sz - 1, Math.min(lo + sz + sz - 1, N -
1));
}

/* 我还是选择递归，简单，漂亮 */
void sort(Comparable[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, lo, mid); // 排序左半边
    sort(a, mid + 1, hi); // 排序右半边
    merge(a, lo, mid, hi); // 合并两边
}
```

看起来简洁漂亮是一方面，关键是**可解释性很强**：把左半边排序，把右半边排序，最后合并两边。而非递归版本看起来不知所云，充斥着各种难以理解的边界计算细节，特别容易出 bug 且难以调试，人生苦短，我更倾向于递归版本。

显然有时候递归处理是高效的，比如归并排序，**有时候是低效的**，比如数孙悟空身上的毛，因为堆栈会消耗额外空间，而简单的递推不会消耗空间。比如这个例子，给一个链表头，计算它的长度：

```
/* 典型的递推遍历框架，需要额外空间 O(1) */
public int size(Node head) {
    int size = 0;
    for (Node p = head; p != null; p = p.next) size++;
```

```
    return size;
}
/* 我偏要递归，万物皆递归，需要额外空间 O(N) */
public int size(Node head) {
    if (head == null) return 0;
    return size(head.next) + 1;
}
```

写递归的技巧

我的一点心得是：**明白一个函数的作用并相信它能完成这个任务，千万不要试图跳进细节**。千万不要跳进这个函数里面企图探究更多细节，否则就会陷入无穷的细节无法自拔，人脑能压几个栈啊。

先举个最简单的例子：遍历二叉树。

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    traverse(root->left);
    traverse(root->right);
}
```

这几行代码就足以扫荡任何一棵二叉树了。我想说的是，对于递归函数 `traverse(root)`，我们只要相信：给它一个根节点 `root`，它就能遍历这棵树，因为写这个函数不就是为了这个目的吗？所以我们只需要把这个节点的左右节点再甩给这个函数就行了，因为我相信它能完成任务的。那么遍历一棵N叉数呢？太简单了好吧，和二叉树一模一样啊。

```
void traverse(TreeNode* root) {
    if (root == nullptr) return;
    for (child : root->children)
        traverse(child);
}
```

至于遍历的什么前、中、后序，那都是显而易见的，对于N叉树，显然没有中序遍历。

以下**详解 LeetCode 的一道题来说明**：给一课二叉树，和一个目标值，节点上的值有正有负，返回树中和等于目标值的路径条数，让你编写 pathSum 函数：

```
/* 来源于 LeetCode PathSum III： https://leetcode.com/problems/path-sum
-iii/ */
root = [10,5,-3,3,2,null,11,3,-2,null,1],
sum = 8

      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1

Return 3. The paths that sum to 8 are:

1.  5 -> 3
2.  5 -> 2 -> 1
3. -3 -> 11
```

```
/* 看不懂没关系，底下有更详细的分析版本，这里突出体现递归的简洁优美 */
int pathSum(TreeNode root, int sum) {
    if (root == null) return 0;
    return count(root, sum) +
        pathSum(root.left, sum) + pathSum(root.right, sum);
}
int count(TreeNode node, int sum) {
    if (node == null) return 0;
    return (node.val == sum) +
        count(node.left, sum - node.val) + count(node.right, sum - no
de.val);
}
```

题目看起来很复杂吧，不过代码却极其简洁，这就是递归的魅力。我来简单总结这个问题的**解决过程**：

首先明确，递归求解树的问题必然是要遍历整棵树的，所以**二叉树的遍历框架**（分别对左右孩子递归调用函数本身）必然要出现在主函数 pathSum 中。那么对于每个节点，他们应该干什么呢？他们应该看看，自己和脚底下的小弟们包含多少条符合条件的路径。好了，这道题就结束了。

按照前面说的技巧，根据刚才的分析来定义清楚每个递归函数应该做的事：

PathSum 函数：给他一个节点和一个目标值，他返回以这个节点为根的树中，和为目标值的路径总数。

count 函数：给他一个节点和一个目标值，他返回以这个节点为根的树中，能凑出几个以该节点为路径开头，和为目标值的路径总数。

```
/* 有了以上铺垫，详细注释一下代码 */
int pathSum(TreeNode root, int sum) {
    if (root == null) return 0;
    int pathImLeading = count(root, sum); // 自己为开头的路径数
    int leftPathSum = pathSum(root.left, sum); // 左边路径总数（相信他能算出来）
    int rightPathSum = pathSum(root.right, sum); // 右边路径总数（相信他能算出来）
    return leftPathSum + rightPathSum + pathImLeading;
}
int count(TreeNode node, int sum) {
    if (node == null) return 0;
    // 我自己能不能独当一面，作为一条单独的路径呢？
    int isMe = (node.val == sum) ? 1 : 0;
    // 左边的小老弟，你那边能凑几个 sum - node.val 呀？
    int leftBrother = count(node.left, sum - node.val);
    // 右边的小老弟，你那边能凑几个 sum - node.val 呀？
    int rightBrother = count(node.right, sum - node.val);
    return  isMe + leftBrother + rightBrother; // 我这能凑这么多个
}
```

还是那句话，明白每个函数能做的事，并相信他们能够完成。

总结下，PathSum 函数提供的二叉树遍历框架，在遍历中对每个节点调用 count 函数，看出先序遍历了吗（这道题什么序都是一样的）；count 函数也是一个二叉树遍历，用于寻找以该节点开头的目标值路径。好好体会吧！

分治算法

归并排序，典型的分治算法；分治，典型的递归结构。

分治算法可以分三步走：分解 -> 解决 -> 合并

1. 分解原问题为结构相同的子问题。
2. 分解到某个容易求解的边界之后，进行第归求解。
3. 将子问题的解合并成原问题的解。

归并排序，我们就叫这个函数 `merge_sort` 吧，按照我们上面说的，要明确该函数的职责，即**对传入的一个数组排序**。OK，那么这个问题能不能分解呢？当然可以！给一个数组排序，不就等于给该数组的两半分别排序，然后合并就完事了。

```
void merge_sort(一个数组) {
    if (可以很容易处理) return;
    merge_sort(左半个数组);
    merge_sort(右半个数组);
    merge(左半个数组, 右半个数组);
}
```

好了，这个算法也就这样了，完全没有任何难度。记住之前说的，相信函数的能力，传给他半个数组，那么这半个数组就已经被排好了。而且你会发现这不就是个二叉树遍历模板吗？为什么是后序遍历？因为我们分治算法的套路是 **分解 -> 解决（触底） -> 合并（回溯）** 啊，先左右分解，再处理合并，回溯就是在退栈，就相当于后序遍历了。至于 `merge` 函数，参考两个有序链表的合并，简直一模一样，下面直接贴代码吧。

下面参考《算法4》的 Java 代码，很漂亮。由此可见，不仅算法思想思想重要，编码技巧也是挺重要的吧！多思考，多模仿。

```
public class Merge {
    // 不要在 merge 函数里构造新数组了，因为 merge 函数会被多次调用，影响性能
    // 直接一次性构造一个足够大的数组，简洁，高效
    private static Comparable[] aux;
```

```
     public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, 0, a.length - 1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (lo >= hi) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, lo, mid);
        sort(a, mid + 1, hi);
        merge(a, lo, mid, hi);
    }

    private static void merge(Comparable[] a, int lo, int mid, int hi
) {
        int i = lo, j = mid + 1;
        for (int k = lo; k <= hi; k++)
            aux[k] = a[k];
        for (int k = lo; k <= hi; k++) {
            if      (i > mid)              { a[k] = aux[j++]; }
            else if (j > hi)               { a[k] = aux[i++]; }
            else if (less(aux[j], aux[i])) { a[k] = aux[j++]; }
            else                           { a[k] = aux[i++]; }
        }
    }

    private static boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }
}
```

LeetCode 上有分治算法的专项练习，可复制到浏览器去做题：

https://leetcode.com/tag/divide-and-conquer/

高频面试系列

8 说了，本章都是高频面试题，配合前面的动态规划系列，祝各位马到成功！

欢迎关注我的公众号 labuladong，方便获得最新的优质文章：

如何高效寻找素数

素数的定义看起来很简单，如果一个数如果只能被 1 和它本身整除，那么这个数就是素数。

不要觉得素数的定义简单，恐怕没多少人真的能把素数相关的算法写得高效。比如让你写这样一个函数：

```
// 返回区间 [2, n) 中有几个素数
int countPrimes(int n)

// 比如 countPrimes(10) 返回 4
// 因为 2,3,5,7 是素数
```

你会如何写这个函数？我想大家应该会这样写：

```
int countPrimes(int n) {
    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrim(i)) count++;
    return count;
}

// 判断整数 n 是否是素数
boolean isPrime(int n) {
    for (int i = 2; i < n; i++)
        if (n % i == 0)
            // 有其他整除因子
            return false;
    return true;
}
```

这样写的话时间复杂度 O(n^2)，问题很大。**首先你用 isPrime 函数来辅助的思路就不够高效；而且就算你要用 isPrime 函数，这样写算法也是存在计算冗余的**。

先来简单说下**如果你要判断一个数是不是素数，应该如何写算法**。只需稍微修改一下上面的 isPrim 代码中的 for 循环条件：

```
boolean isPrime(int n) {
    for (int i = 2; i * i <= n; i++)
        ...
}
```

换句话说，`i` 不需要遍历到 `n`，而只需要到 `sqrt(n)` 即可。为什么呢，我们举个例子，假设 `n = 12`。

```
12 = 2 × 6
12 = 3 × 4
12 = sqrt(12) × sqrt(12)
12 = 4 × 3
12 = 6 × 2
```

可以看到，后两个乘积就是前面两个反过来，反转临界点就在 `sqrt(n)`。

换句话说，如果在 `[2,sqrt(n)]` 这个区间之内没有发现可整除因子，就可以直接断定 `n` 是素数了，因为在区间 `[sqrt(n),n]` 也一定不会发现可整除因子。

现在，`isPrime` 函数的时间复杂度降为 O(sqrt(N))，**但是我们实现 `countPrimes` 函数其实并不需要这个函数**，以上只是希望读者明白 `sqrt(n)` 的含义，因为等会还会用到。

高效实现 `countPrimes`

高效解决这个问题的核心思路是和上面的常规思路反着来：

首先从 2 开始，我们知道 2 是一个素数，那么 2 × 2 = 4, 3 × 2 = 6, 4 × 2 = 8... 都不可能是素数了。

然后我们发现 3 也是素数，那么 3 × 2 = 6, 3 × 3 = 9, 3 × 4 = 12... 也都不可能是素数了。

看到这里，你是否有点明白这个排除法的逻辑了呢？先看我们的第一版代码：

```
int countPrimes(int n) {
    boolean[] isPrim = new boolean[n];
    // 将数组都初始化为 true
    Arrays.fill(isPrim, true);

    for (int i = 2; i < n; i++)
        if (isPrim[i])
            // i 的倍数不可能是素数了
            for (int j = 2 * i; j < n; j += i)
                    isPrim[j] = false;

    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrim[i]) count++;

    return count;
}
```

如果上面这段代码你能够理解，那么你已经掌握了整体思路，但是还有两个细微的地方可以优化。

首先，回想刚才判断一个数是否是素数的 `isPrime` 函数，由于因子的对称性，其中的 for 循环只需要遍历 `[2,sqrt(n)]` 就够了。这里也是类似的，我们外层的 for 循环也只需要遍历到 `sqrt(n)` ：

```
for (int i = 2; i * i < n; i++)
    if (isPrim[i])
        ...
```

除此之外，很难注意到内层的 for 循环也可以优化。我们之前的做法是：

```
for (int j = 2 * i; j < n; j += i)
    isPrim[j] = false;
```

这样可以把 `i` 的整数倍都标记为 `false`，但是仍然存在计算冗余。

比如 `n = 25`，`i = 4` 时算法会标记 4 × 2 = 8，4 × 3 = 12 等等数字，但是这两个数字已经被 `i = 2` 和 `i = 3` 的 2 × 4 和 3 × 4 标记了。

我们可以稍微优化一下，让 `j` 从 `i` 的平方开始遍历，而不是从 `2 * i` 开始：

```
for (int j = i * i; j < n; j += i)
    isPrim[j] = false;
```

这样，素数计数的算法就高效实现了，其实这个算法有一个名字，叫做 Sieve of Eratosthenes。看下完整的最终代码：

```
int countPrimes(int n) {
    boolean[] isPrim = new boolean[n];
    Arrays.fill(isPrim, true);
    for (int i = 2; i * i < n; i++)
        if (isPrim[i])
            for (int j = i * i; j < n; j += i)
                isPrim[j] = false;

    int count = 0;
    for (int i = 2; i < n; i++)
        if (isPrim[i]) count++;

    return count;
}
```

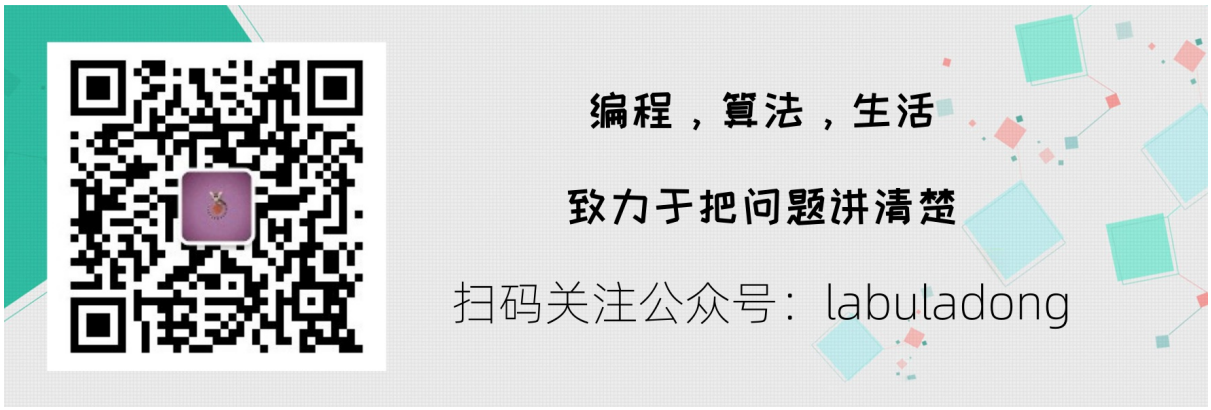
该算法的时间复杂度比较难算，显然时间跟这两个嵌套的 for 循环有关，其操作数应该是：

n/2 + n/3 + n/5 + n/7 + ... = n × (1/2 + 1/3 + 1/5 + 1/7...)

括号中是素数的倒数。其最终结果是 O(N * loglogN)，有兴趣的读者可以查一下该算法的时间复杂度证明。

以上就是素数算法相关的全部内容。怎么样，是不是看似简单的问题却有不少细节可以打磨呀？

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：



如何运用二分查找算法

二分查找到底有能运用在哪里？

最常见的就是教科书上的例子，在**有序数组**中搜索给定的某个目标值的索引。再推广一点，如果目标值存在重复，修改版的二分查找可以返回目标值的左侧边界索引或者右侧边界索引。

PS：以上提到的三种二分查找算法形式在前文「二分查找详解」有代码详解，如果没看过强烈建议看看。

抛开有序数组这个枯燥的数据结构，二分查找如何运用到实际的算法问题中呢？当搜索空间有序的时候，就可以通过二分搜索「剪枝」，大幅提升效率。

说起来玄乎得很，本文先用一个具体的「Koko 吃香蕉」的问题来举个例子。

一、问题分析

珂珂喜欢吃香蕉。这里有 `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。

珂珂可以决定她吃香蕉的速度 `K` （单位：根/小时）。每个小时，她将会选择一堆香蕉，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：

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

示例 2：

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

也就是说，Koko 每小时最多吃一堆香蕉，如果吃不下的话留到下一小时再吃；如果吃完了这一堆还有胃口，也只会等到下一小时才会吃下一堆。在这个条件下，让我们确定 Koko 吃香蕉的**最小速度**（根/小时）。

如果直接给你这个情景，你能想到哪里能用到二分查找算法吗？如果没有见过类似的问题，恐怕是很难把这个问题和二分查找联系起来的。

那么我们先抛开二分查找技巧，想想如何暴力解决这个问题呢？

首先，算法要求的是「`H` 小时内吃完香蕉的最小速度」，我们不妨称为 `speed`，请问 `speed` 最大可能为多少，最少可能为多少呢？

显然最少为 1，最大为 `max(piles)`，因为一小时最多只能吃一堆香蕉。那么暴力解法就很简单了，只要从 1 开始穷举到 `max(piles)`，一旦发现发现某个值可以在 `H` 小时内吃完所有香蕉，这个值就是最小速度：

```
int minEatingSpeed(int[] piles, int H) {
    // piles 数组的最大值
    int max = getMax(piles);
    for (int speed = 1; speed < max; speed++) {
        // 以 speed 是否能在 H 小时内吃完香蕉
        if (canFinish(piles, speed, H))
            return speed;
    }
    return max;
}
```

注意这个 for 循环，就是在**连续的空间线性搜索，这就是二分查找可以发挥作用的标志**。由于我们要求的是最小速度，所以可以用一个**搜索左侧边界的二分查找**来代替线性搜索，提升效率：

```
int minEatingSpeed(int[] piles, int H) {
    // 套用搜索左侧边界的算法框架
    int left = 1, right = getMax(piles) + 1;
    while (left < right) {
        // 防止溢出
        int mid = left + (right - left) / 2;
        if (canFinish(piles, mid, H)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

PS：如果对于这个二分查找算法的细节问题有疑问，建议看下前文「二分查找详解」搜索左侧边界的算法模板，这里不展开了。

剩下的辅助函数也很简单，可以一步步拆解实现：

```
// 时间复杂度 O(N)
boolean canFinish(int[] piles, int speed, int H) {
    int time = 0;
    for (int n : piles) {
```

```
            time += timeOf(n, speed);
        }
        return time <= H;
    }

    int timeOf(int n, int speed) {
        return (n / speed) + ((n % speed > 0) ? 1 : 0);
    }

    int getMax(int[] piles) {
        int max = 0;
        for (int n : piles)
            max = Math.max(n, max);
        return max;
    }
```

至此，借助二分查找技巧，算法的时间复杂度为 O(NlogN)。

二、扩展延伸

类似的，再看一道运输问题：

传送带上的第 `i` 个包裹的重量为 `weights[i]`。每一天，我们都会按给出重量的顺序往传送带上装载包裹。我们装载的重量不会超过船的最大运载重量。

返回能在 `D` 天内将传送带上的所有包裹送达的船的最低运载能力。

示例 1：

输入：weights = [1,2,3,4,5,6,7,8,9,10], D = 5
输出：15
解释：
船舶最低载重 15 就能够在 5 天内送达所有包裹，如下所示：
第 1 天：1, 2, 3, 4, 5
第 2 天：6, 7
第 3 天：8
第 4 天：9
第 5 天：10

请注意，货物必须按照给定的顺序装运，因此使用载重能力为 14 的船舶并将包装分成 (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) 是不允许的。

要在 `D` 天内运输完所有货物，货物不可分割，如何确定运输的最小载重呢（下文称为 `cap`）？

其实本质上和 Koko 吃香蕉的问题一样的，首先确定 `cap` 的最小值和最大值分别为 `max(weights)` 和 `sum(weights)`。

我们要求**最小载重**，所以可以用搜索左侧边界的二分查找算法优化线性搜索：

```
// 寻找左侧边界的二分查找
int shipWithinDays(int[] weights, int D) {
    // 载重可能的最小值
    int left = getMax(weights);
    // 载重可能的最大值 + 1
    int right = getSum(weights) + 1;
    while (left < right) {
```

```
        int mid = left + (right - left) / 2;
        if (canFinish(weights, D, mid)) {
            right = mid;
        } else {
            left = mid + 1;
        }
    }
    return left;
}

// 如果载重为 cap，是否能在 D 天内运完货物？
boolean canFinish(int[] w, int D, int cap) {
    int i = 0;
    for (int day = 0; day < D; day++) {
        int maxCap = cap;
        while ((maxCap -= w[i]) >= 0) {
            i++;
            if (i == w.length)
                return true;
        }
    }
    return false;
}
```

通过这两个例子，你是否明白了二分查找在实际问题中的应用？

```
for (int i = 0; i < n; i++)
    if (isOK(i))
        return ans;
```

接雨水问题详解

接雨水这道题目挺有意思，在面试题中出现频率还挺高的，本文就来步步优化，讲解一下这道题。

先看一下题目：

给定 *n* 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 **感谢 Marcos** 贡献此图。

示例:

```
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

就是用一个数组表示一个条形图，问你这个条形图最多能接多少水。

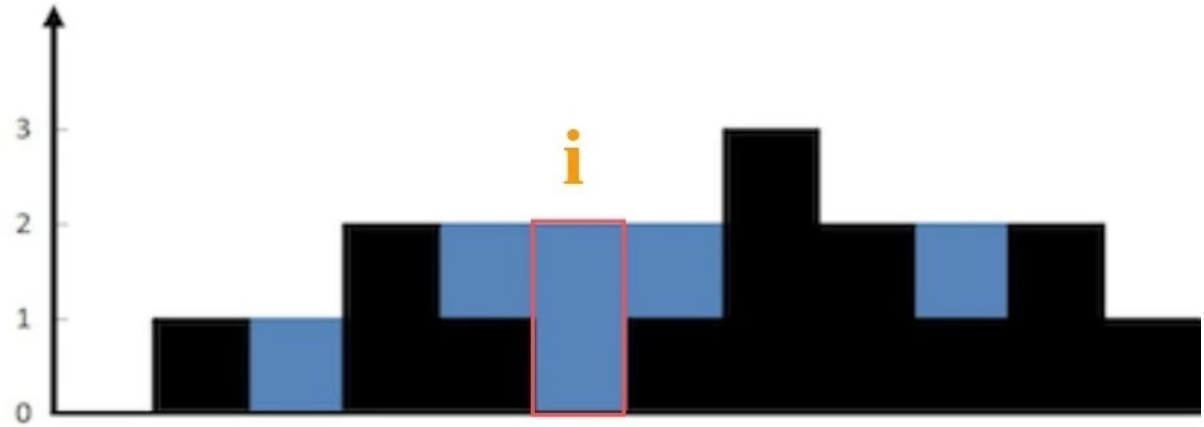
```
int trap(int[] height);
```

下面就来由浅入深介绍暴力解法 -> 备忘录解法 -> 双指针解法，在 O(N) 时间 O(1) 空间内解决这个问题。

一、核心思路

我第一次看到这个问题，无计可施，完全没有思路，相信很多朋友跟我一样。所以对于这种问题，我们不要想整体，而应该去想局部；就像之前的文章处理字符串问题，不要考虑如何处理整个字符串，而是去思考应该如何处理每一个字符。

这么一想，可以发现这道题的思路其实很简单。具体来说，仅仅对于位置 i，能装下多少水呢？

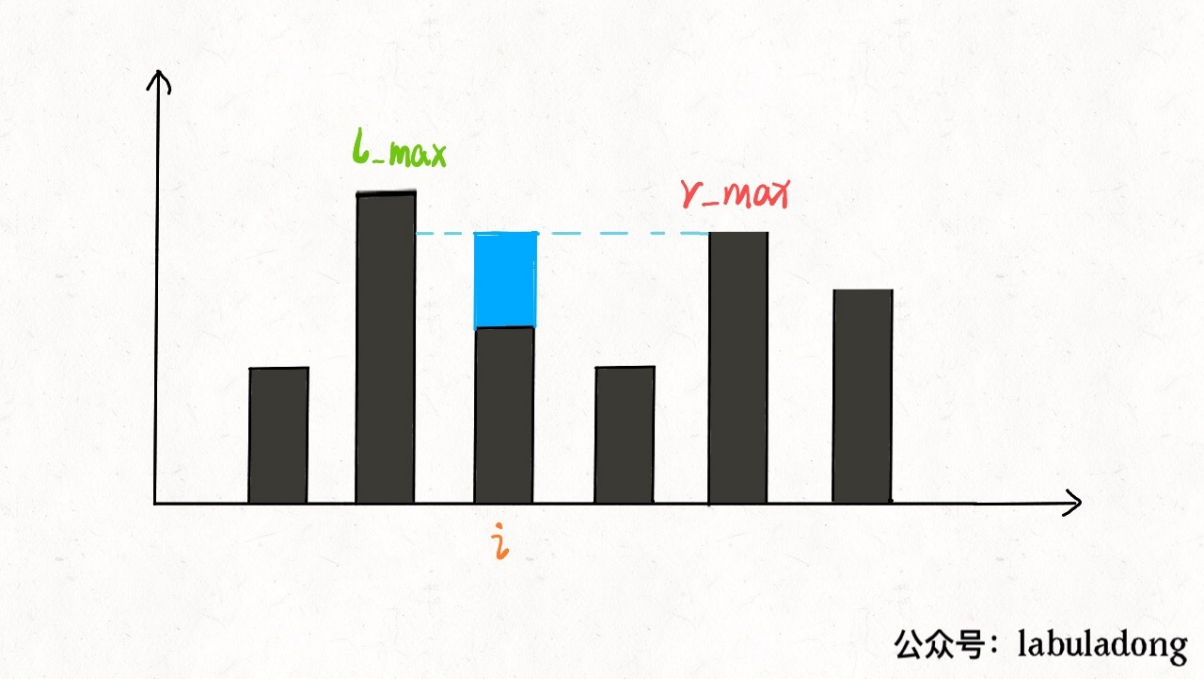


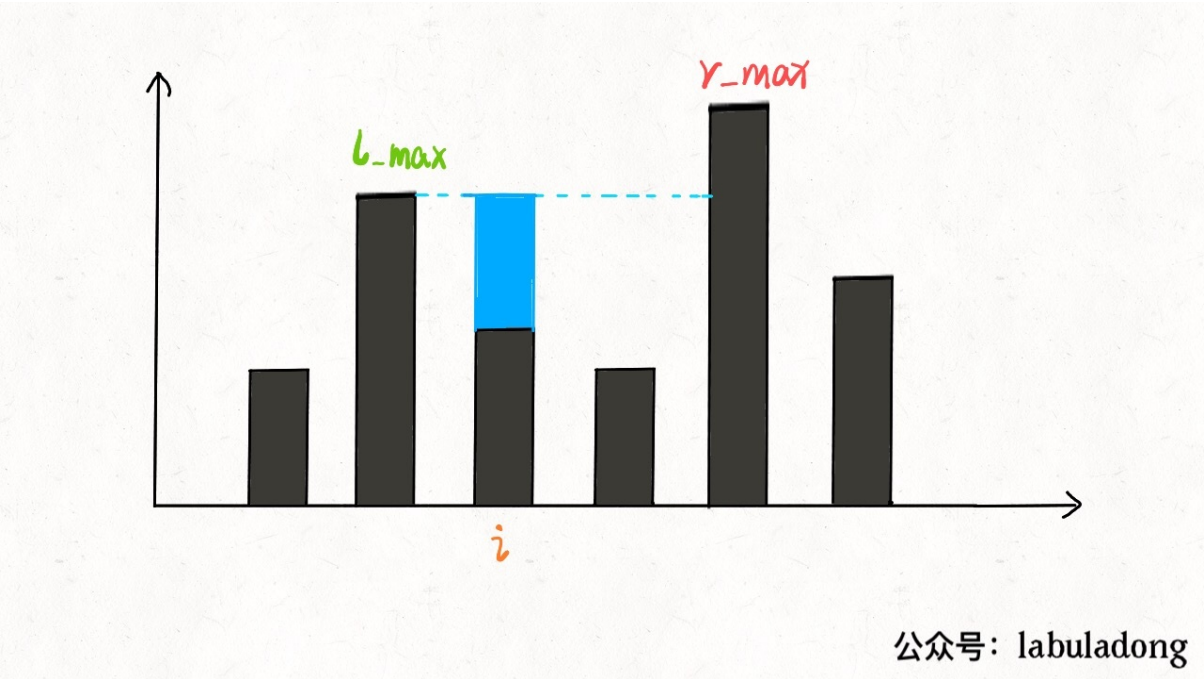
能装 2 格水。为什么恰好是两格水呢？因为 height[i] 的高度为 0，而这里最多能盛 2 格水，2-0=2。

为什么位置 i 最多能盛 2 格水呢？因为，位置 i 能达到的水柱高度和其左边的最高柱子、右边的最高柱子有关，我们分别称这两个柱子高度为 `l_max` 和 `r_max`；**位置 i 最大的水柱高度就是 `min(l_max, r_max)`**。

更进一步，对于位置 i，能够装的水为：

```
water[i] = min(
                # 左边最高的柱子
                max(height[0..i]),
                # 右边最高的柱子
                max(height[i..end])
            ) - height[i]
```





这就是本问题的核心思路，我们可以简单写一个暴力算法：

```
int trap(vector<int>& height) {
    int n = height.size();
    int ans = 0;
    for (int i = 1; i < n - 1; i++) {
        int l_max = 0, r_max = 0;
        // 找右边最高的柱子
        for (int j = i; j < n; j++)
```

```
            r_max = max(r_max, height[j]);
        // 找左边最高的柱子
        for (int j = i; j >= 0; j--)
            l_max = max(l_max, height[j]);
        // 如果自己就是最高的话，
        // l_max == r_max == height[i]
        ans += min(l_max, r_max) - height[i];
    }
    return ans;
}
```

有之前的思路，这个解法应该是很直接粗暴的，时间复杂度 O(N^2)，空间复杂度 O(1)。但是很明显这种计算 `r_max` 和 `l_max` 的方式非常笨拙，一般的优化方法就是备忘录。

二、备忘录优化

之前的暴力解法，不是在每个位置 i 都要计算 `r_max` 和 `l_max` 吗？我们直接把结果都缓存下来，别傻不拉几的每次都遍历，这时间复杂度不就降下来了嘛。

我们开两个**数组** `r_max` 和 `l_max` 充当备忘录，`l_max[i]` 表示位置 i 左边最高的柱子高度，`r_max[i]` 表示位置 i 右边最高的柱子高度。预先把这两个数组计算好，避免重复计算：

```
int trap(vector<int>& height) {
    if (height.empty()) return 0;
    int n = height.size();
    int ans = 0;
    // 数组充当备忘录
    vector<int> l_max(n), r_max(n);
    // 初始化 base case
    l_max[0] = height[0];
    r_max[n - 1] = height[n - 1];
    // 从左向右计算 l_max
    for (int i = 1; i < n; i++)
        l_max[i] = max(height[i], l_max[i - 1]);
    // 从右向左计算 r_max
    for (int i = n - 2; i >= 0; i--)
```

```
        r_max[i] = max(height[i], r_max[i + 1]);
    // 计算答案
    for (int i = 1; i < n - 1; i++)
        ans += min(l_max[i], r_max[i]) - height[i];
    return ans;
}
```

这个优化其实和暴力解法差不多，就是避免了重复计算，把时间复杂度降低为 O(N)，已经是最优了，但是空间复杂度是 O(N)。下面来看一个精妙一些的解法，能够把空间复杂度降低到 O(1)。

三、双指针解法

这种解法的思路是完全相同的，但在实现手法上非常巧妙，我们这次也不要用备忘录提前计算了，而是用双指针**边走边算**，节省下空间复杂度。

首先，看一部分代码：

```
int trap(vector<int>& height) {
    int n = height.size();
    int left = 0, right = n - 1;

    int l_max = height[0];
    int r_max = height[n - 1];

    while (left <= right) {
        l_max = max(l_max, height[left]);
        r_max = max(r_max, height[right]);
        left++; right--;
    }
}
```

对于这部分代码，请问 `l_max` 和 `r_max` 分别表示什么意义呢？

很容易理解，**`l_max` 是 `height[0..left]` 中最高柱子的高度，`r_max` 是 `height[right..end]` 的最高柱子的高度**。

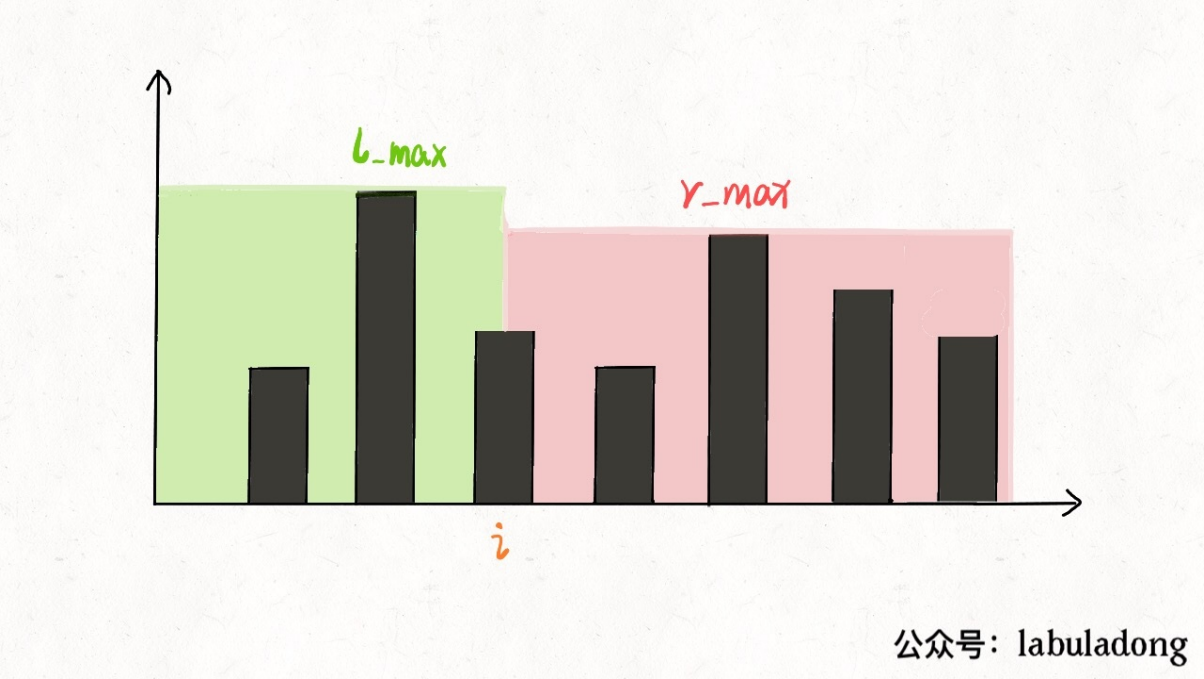
明白了这一点，直接看解法：

```
int trap(vector<int>& height) {
    if (height.empty()) return 0;
    int n = height.size();
    int left = 0, right = n - 1;
    int ans = 0;

    int l_max = height[0];
    int r_max = height[n - 1];

    while (left <= right) {
        l_max = max(l_max, height[left]);
        r_max = max(r_max, height[right]);

        // ans += min(l_max, r_max) - height[i]
        if (l_max < r_max) {
            ans += l_max - height[left];
            left++;
        } else {
            ans += r_max - height[right];
            right--;
        }
    }
    return ans;
}
```

你看，其中的核心思想和之前一模一样，换汤不换药。但是细心的读者可能会发现次解法还是有点细节差异：

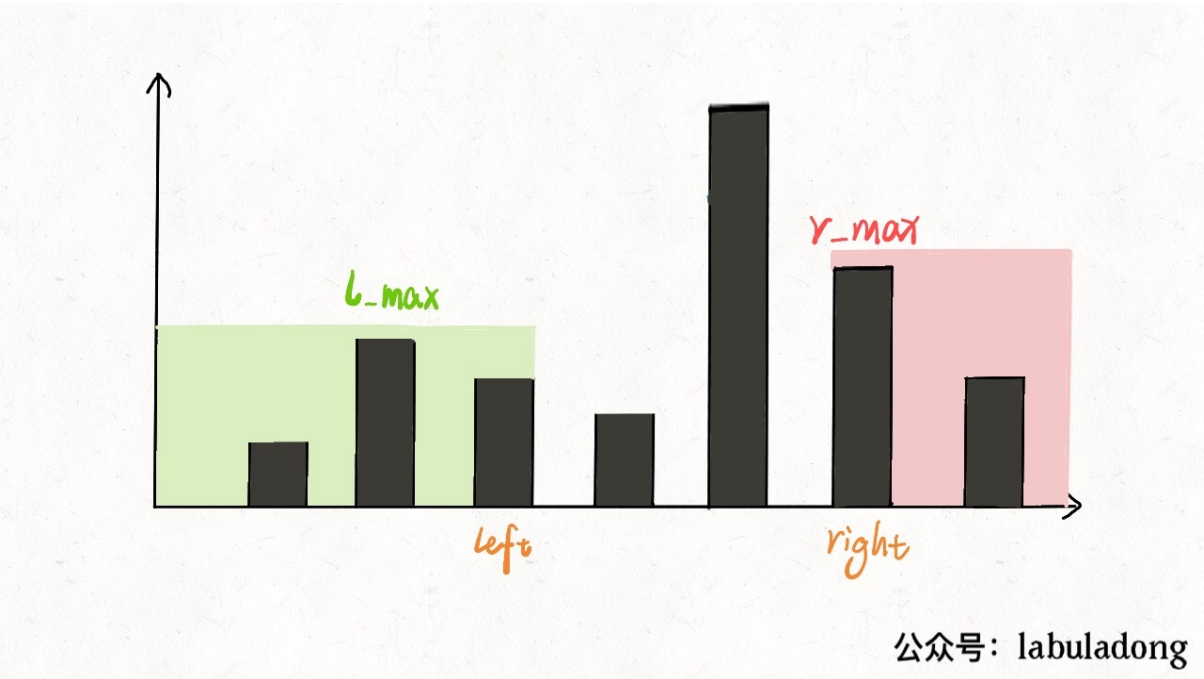
之前的备忘录解法，`l_max[i]` 和 `r_max[i]` 代表的是 `height[0..i]` 和 `height[i..end]` 的最高柱子高度。

```
ans += min(l_max[i], r_max[i]) - height[i];
```



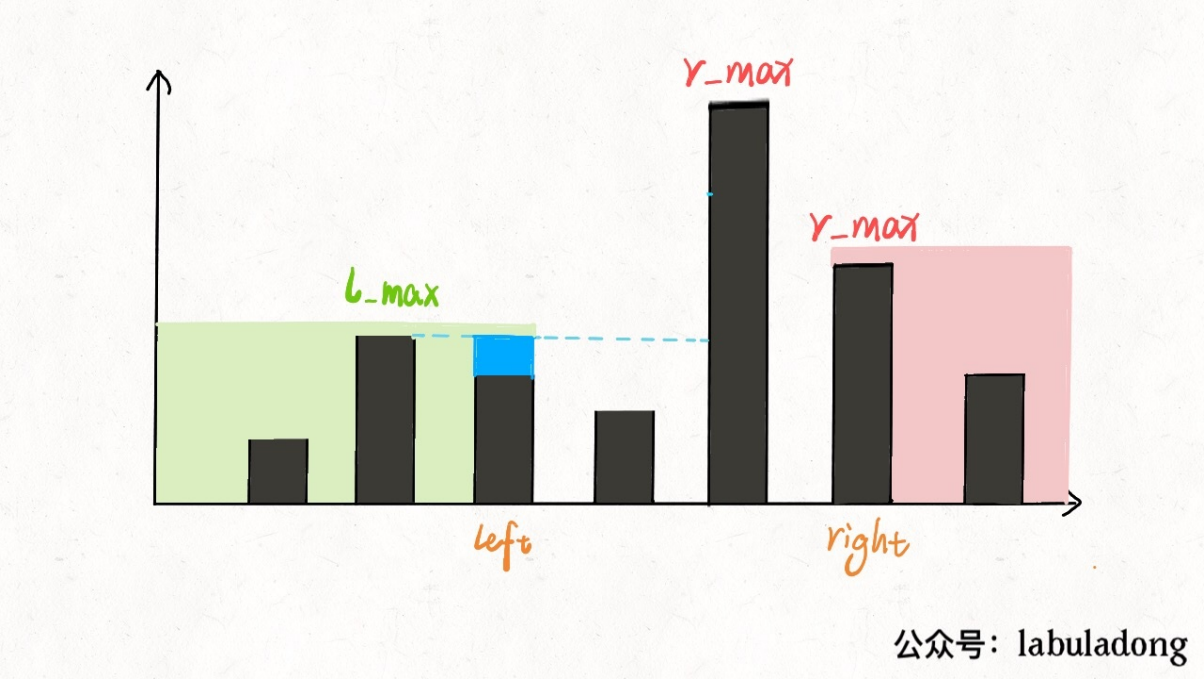
但是双指针解法中，`l_max` 和 `r_max` 代表的是 `height[0..left]` 和 `height[right..end]` 的最高柱子高度。比如这段代码：

```
if (l_max < r_max) {
    ans += l_max - height[left];
    left++;
}
```



此时的 `l_max` 是 `left` 指针左边的最高柱子，但是 `r_max` 并不一定是 `left` 指针右边最高的柱子，这真的可以得到正确答案吗？

其实这个问题要这么思考，我们只在乎 `min(l_max, r_max)`。对于上图的情况，我们已经知道 `l_max < r_max` 了，至于这个 `r_max` 是不是右边最大的，不重要，重要的是 `height[i]` 能够装的水只和 `l_max` 有关。



如何去除有序数组的重复元素

我们知道对于数组来说，在尾部插入、删除元素是比较高效的，时间复杂度是 O(1)，但是如果在中间或者开头插入、删除元素，就会涉及数据的搬移，时间复杂度为 O(N)，效率较低。

所以对于一般处理数组的算法问题，我们要尽可能只对数组尾部的元素进行操作，以避免额外的时间复杂度。

这篇文章讲讲如何对一个有序数组去重，先看下题目：

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:

```
给定数组 nums = [1,1,2],

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。

你不需要考虑数组中超出新长度后面的元素。
```

示例 2:

```
给定 nums = [0,0,1,1,1,2,2,3,3,4],

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。

你不需要考虑数组中超出新长度后面的元素。
```

显然，由于数组已经排序，所以重复的元素一定连在一起，找出它们并不难，但如果每找到一个重复元素就立即删除它，就是在数组中间进行删除操作，整个时间复杂度是会达到 O(N^2)。而且题目要求我们原地修改，也就是说不能用辅助数组，空间复杂度得是 O(1)。

其实，**对于数组相关的算法问题，有一个通用的技巧：要尽量避免在中间删除元素，那我就想先办法把这个元素换到最后去**。这样的话，最终待删除的元素都拖在数组尾部，一个一个 pop 掉就行了，每次操作的时间复杂度也就降低到 O(1) 了。

按照这个思路呢，又可以衍生出解决类似需求的通用方式：双指针技巧。具体一点说，应该是快慢指针。

我们让慢指针 `slow` 走左后面，快指针 `fast` 走在前面探路，找到一个不重复的元素就告诉 `slow` 并让 `slow` 前进一步。这样当 `fast` 指针遍历完整个数组 `nums` 后，**`nums[0..slow]` 就是不重复元素，之后的所有元素都是重复元素**。

```
int removeDuplicates(int[] nums) {
    int n = nums.length;
    if (n == 0) return 0;
    int slow = 0, fast = 1;
    while (fast < n) {
        if (nums[fast] != nums[slow]) {
            slow++;
            // 维护 nums[0..slow] 无重复
            nums[slow] = nums[fast];
        }
        fast++;
    }
    // 长度为索引 + 1
    return slow + 1;
}
```

看下算法执行的过程：

【pdf/mobi格式不支持 GIF:%E6%9C%89%E5%BA%8F%E6%95%B0%E7%BB%84%E5%8E%BB%E9%87%8D/1.gif】 请查看【关于本小抄及作者】章节的解决方案

再简单扩展一下，如果给你一个有序链表，如何去重呢？其实和数组是一模一样的，唯一的区别是把数组赋值操作变成操作指针而已：

```
ListNode deleteDuplicates(ListNode head) {
    if (head == null) return null;
    ListNode slow = head, fast = head.next;
    while (fast != null) {
        if (fast.val != slow.val) {
            // nums[slow] = nums[fast];
            slow.next = fast;
            // slow++;
            slow = slow.next;
        }
        // fast++
        fast = fast.next;
    }
    // 断开与后面重复元素的连接
    slow.next = null;
    return head;
}
```

【pdf/mobi格式不支持GIF:%E6%9C%89%E5%BA%8F%E6%95%B0%E7%BB%84%E5%8E%BB%E9%87%8D/2.gif】 请查看【关于本小抄及作者】章节的解决方案

致力于把算法讲清楚！欢迎关注我的微信公众号 labuladong，查看更多通俗易懂的文章：

如何寻找最长回文子串

回文串是面试常常遇到的问题（虽然问题本身没啥意义），本文就告诉你回文串问题的核心思想是什么。

首先，明确一下什：**回文串就是正着读和反着读都一样的字符串**。

比如说字符串 `aba` 和 `abba` 都是回文串，因为它们对称，反过来还是和本身一样。反之，字符串 `abac` 就不是回文串。

可以看到回文串的的长度可能是奇数，也可能是偶数，这就添加了回文串问题的难度，解决该类问题的核心是**双指针**。下面就通过一道最长回文子串的问题来具体理解一下回文串问题：

给定一个字符串 `s`，找到 `s` 中最长的回文子串。你可以假设 `s` 的最大长度为 1000。

示例 1：

```
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。
```

示例 2：

```
输入: "cbbd"
输出: "bb"
```

```
string longestPalindrome(string s) {}
```

一、思考

对于这个问题，我们首先应该思考的是，给一个字符串 `s`，如何在 `s` 中找到一个回文子串？

有一个很有趣的思路：既然回文串是一个正着反着读都一样的字符串，那么如果我们把 `s` 反转，称为 `s'`，然后在 `s` 和 `s'` 中寻找**最长公共子串**，这样应该就能找到最长回文子串。

比如说字符串 `abacd`，反过来是 `dcaba`，它的最长公共子串是 `aba`，也就是最长回文子串。

但是这个思路是错误的，比如说字符串 `aacxycaa`，反转之后是 `aacyxcaa`，最长公共子串是 `aac`，但是最长回文子串应该是 `aa`。

虽然这个思路不正确，但是**这种把问题转化为其他形式的思考方式是非常值得提倡的**。

下面，就来说一下正确的思路，如何使用双指针。

寻找回文串的问题核心思想是：从中间开始向两边扩散来判断回文串。对于最长回文子串，就是这个意思：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串
    更新答案
```

但是呢，我们刚才也说了，回文串的长度可能是奇数也可能是偶数，如果是 `abba` 这种情况，没有一个中心字符，上面的算法就没辙了。所以我们可以修改一下：

```
for 0 <= i < len(s):
    找到以 s[i] 为中心的回文串
    找到以 s[i] 和 s[i+1] 为中心的回文串
    更新答案
```

PS：读者可能发现这里的索引会越界，等会会处理。

二、代码实现

按照上面的思路，先要实现一个函数来寻找最长回文串，这个函数是有点技巧的：

```
string palindrome(string& s, int l, int r) {
    // 防止索引越界
    while (l >= 0 && r < s.size()
            && s[l] == s[r]) {
        // 向两边展开
        l--; r++;
    }
    // 返回以 s[l] 和 s[r] 为中心的最长回文串
    return s.substr(l + 1, r - l - 1);
}
```

为什么要传入两个指针 `l` 和 `r` 呢？**因为这样实现可以同时处理回文串长度为奇数和偶数的情况**：

```
for 0 <= i < len(s):
    # 找到以 s[i] 为中心的回文串
    palindrome(s, i, i)
    # 找到以 s[i] 和 s[i+1] 为中心的回文串
    palindrome(s, i, i + 1)
    更新答案
```

下面看下 `longestPalindrome` 的完整代码：

```
string longestPalindrome(string s) {
    string res;
    for (int i = 0; i < s.size(); i++) {
        // 以 s[i] 为中心的最长回文子串
        string s1 = palindrome(s, i, i);
        // 以 s[i] 和 s[i+1] 为中心的最长回文子串
        string s2 = palindrome(s, i, i + 1);
        // res = longest(res, s1, s2)
        res = res.size() > s1.size() ? res : s1;
        res = res.size() > s2.size() ? res : s2;
    }
    return res;
}
```

至此，这道最长回文子串的问题就解决了，时间复杂度 O(N^2)，空间复杂度 O(1)。

值得一提的是，这个问题可以用动态规划方法解决，时间复杂度一样，但是空间复杂度至少要 O(N^2) 来存储 DP table。这道题是少有的动态规划非最优解法的问题。

另外，这个问题还有一个巧妙的解法，时间复杂度只需要 O(N)，不过该解法比较复杂，我个人认为没必要掌握。该算法的名字叫 Manacher's Algorithm（马拉车算法），有兴趣的读者可以自行搜索一下。

如何k个一组反转链表

之前的文章「递归反转链表的一部分」讲了如何递归地反转一部分链表，有读者就问如何迭代地反转链表，这篇文章解决的问题也需要反转链表的函数，我们不妨就用迭代方式来解决。

本文要解决「K 个一组反转链表」，不难理解：

> 给你一个链表，每 *k* 个节点一组进行翻转，请你返回翻转后的链表。
>
> *k* 是一个正整数，它的值小于或等于链表的长度。
>
> 如果节点总数不是 *k* 的整数倍，那么请将最后剩余的节点保持原有顺序。
>
> **示例：**
>
> 给定这个链表：`1->2->3->4->5`
>
> 当 *k* = 2 时，应当返回: `2->1->4->3->5`
>
> 当 *k* = 3 时，应当返回: `3->2->1->4->5`

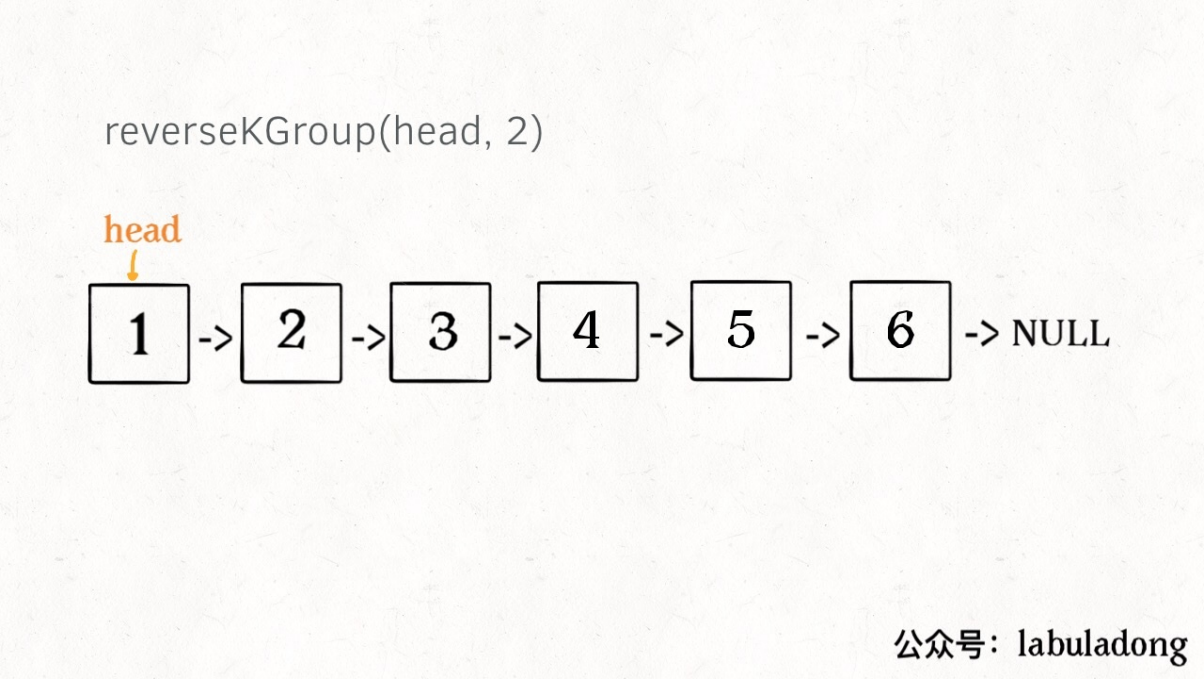
这个问题经常在面经中看到，而且 LeetCode 上难度是 Hard，它真的有那么难吗？

对于基本数据结构的算法问题其实都不难，只要结合特点一点点拆解分析，一般都没啥难点。下面我们就来拆解一下这个问题。

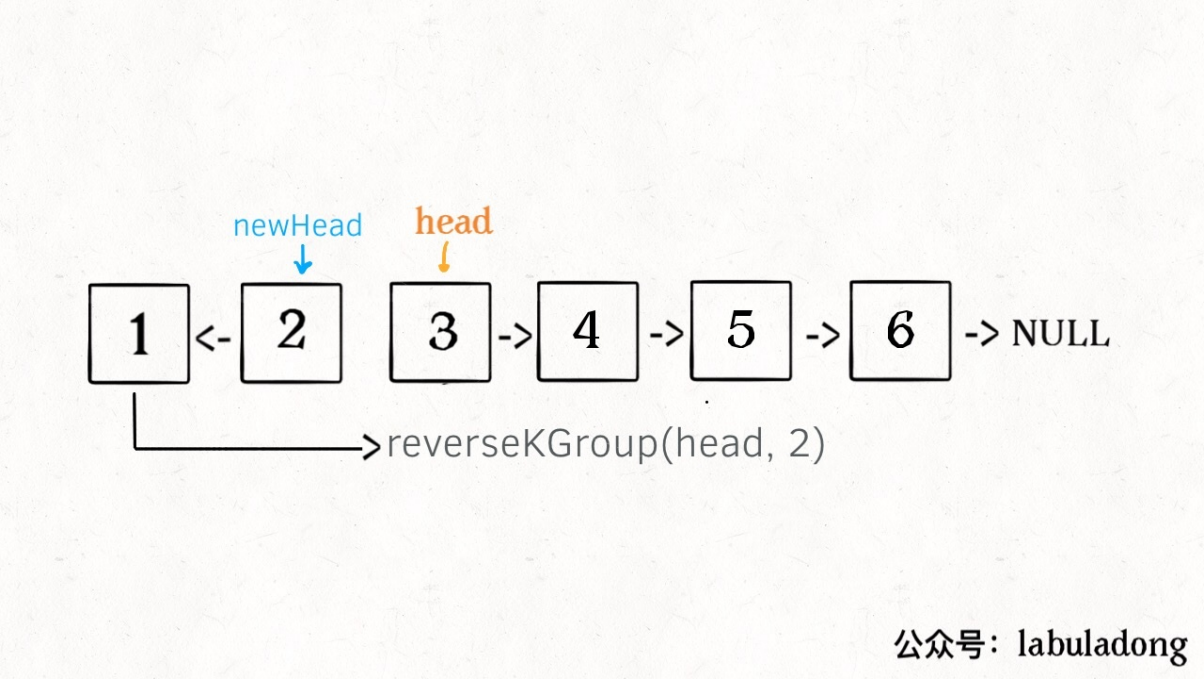
一、分析问题

首先，前文学习数据结构的框架思维提到过，链表是一种兼具递归和迭代性质的数据结构，认真思考一下可以发现**这个问题具有递归性质**。

什么叫递归性质？直接上图理解，比如说我们对这个链表调用 `reverseKGroup(head, 2)`，即以 2 个节点为一组反转链表：



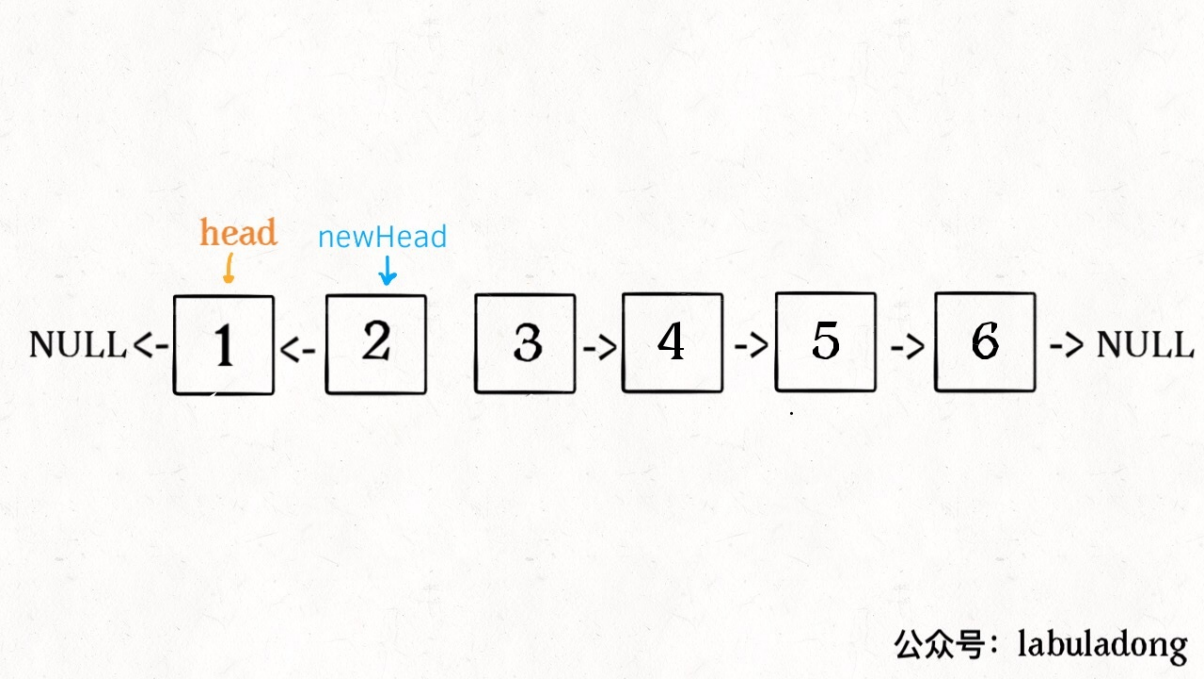
如果我设法把前 2 个节点反转，那么后面的那些节点怎么处理？后面的这些节点也是一条链表，而且规模（长度）比原来这条链表小，这就叫**子问题**。



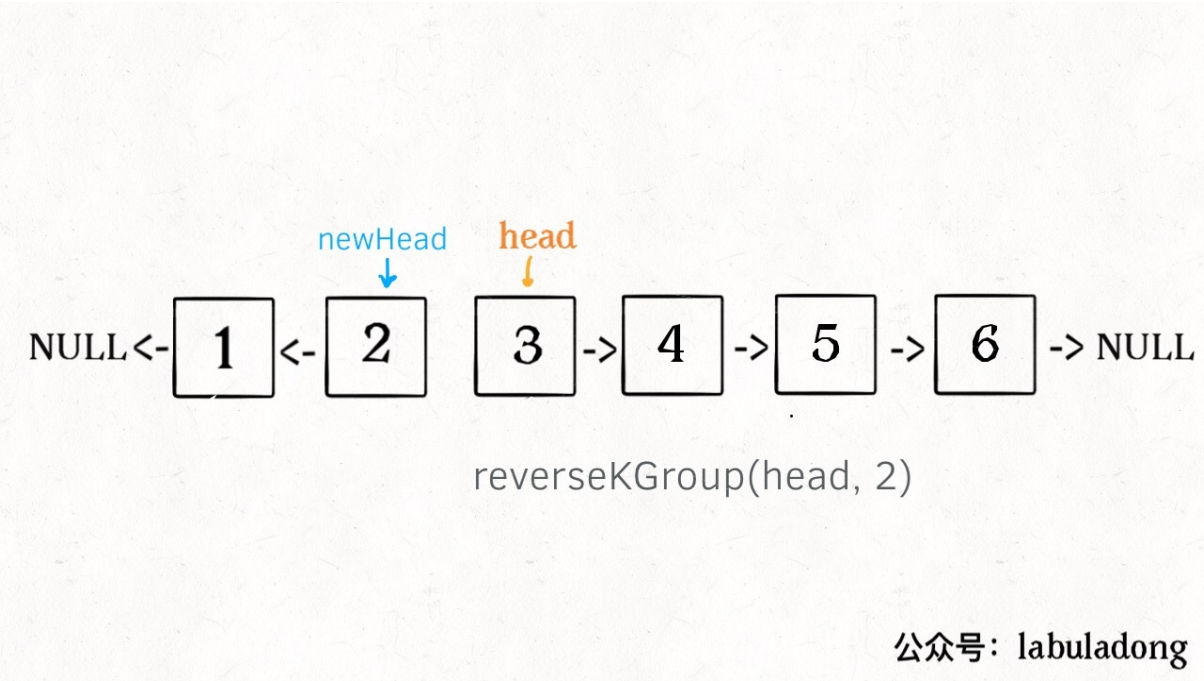
我们可以直接递归调用 `reverseKGroup(cur, 2)`，因为子问题和原问题的结构完全相同，这就是所谓的递归性质。

发现了递归性质，就可以得到大致的算法流程：

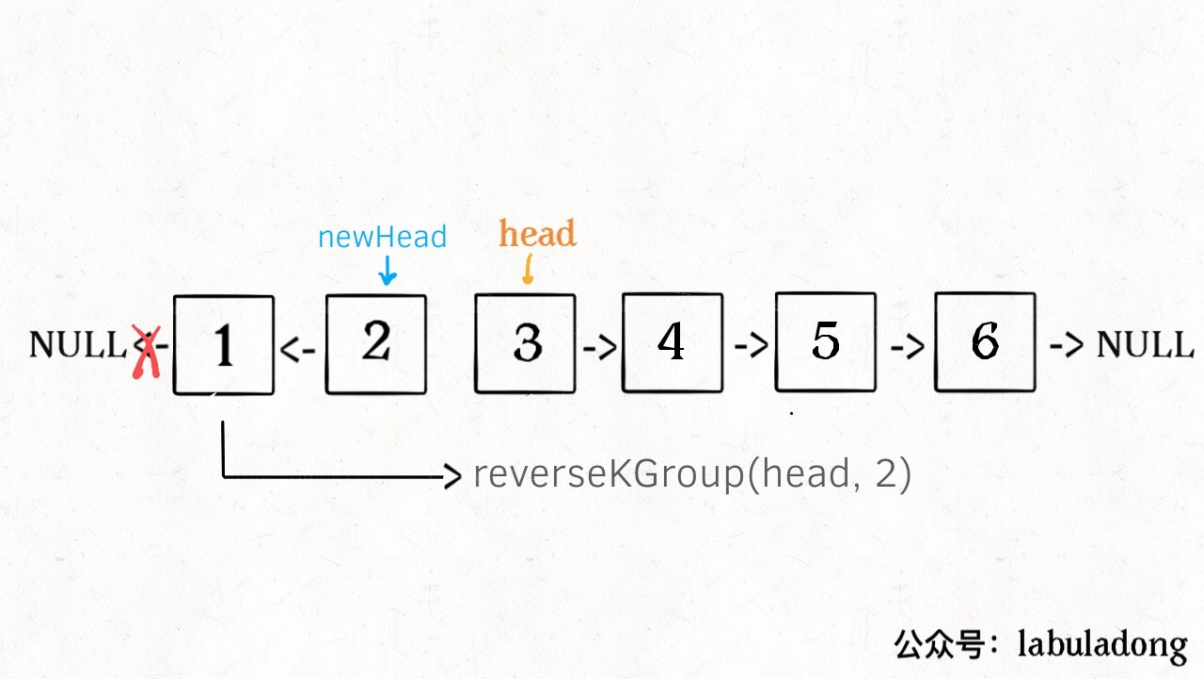
1、先反转以 `head` 开头的 `k` 个元素。



2、将第 `k + 1` 个元素作为 `head` 递归调用 `reverseKGroup` 函数。



3、将上述两个过程的结果连接起来。



整体思路就是这样了，最后一点值得注意的是，递归函数都有个 base case，对于这个问题是什么呢？

题目说了，如果最后的元素不足 `k` 个，就保持不变。这就是 base case，待会会在代码里体现。

二、代码实现

首先，我们要实现一个 `reverse` 函数反转一个区间之内的元素。在此之前我们再简化一下，给定链表头结点，如何反转整个链表？

```
// 反转以 a 为头结点的链表
ListNode reverse(ListNode a) {
    ListNode pre, cur, nxt;
    pre = null; cur = a; nxt = a;
    while (cur != null) {
        nxt = cur.next;
        // 逐个结点反转
        cur.next = pre;
        // 更新指针位置
        pre = cur;
        cur = nxt;
    }
    // 返回反转后的头结点
```

```
    return pre;
}
```

【pdf/mobi格式不支持GIF:kgroup/8.gif】 请查看【关于本小抄及作者】章节的解决方案

这次使用迭代思路来实现的，借助动画理解应该很容易。

「反转以 `a` 为头结点的链表」其实就是「反转 `a` 到 null 之间的结点」，那么如果让你「反转 `a` 到 `b` 之间的结点」，你会不会？

只要更改函数签名，并把上面的代码中 `null` 改成 `b` 即可：

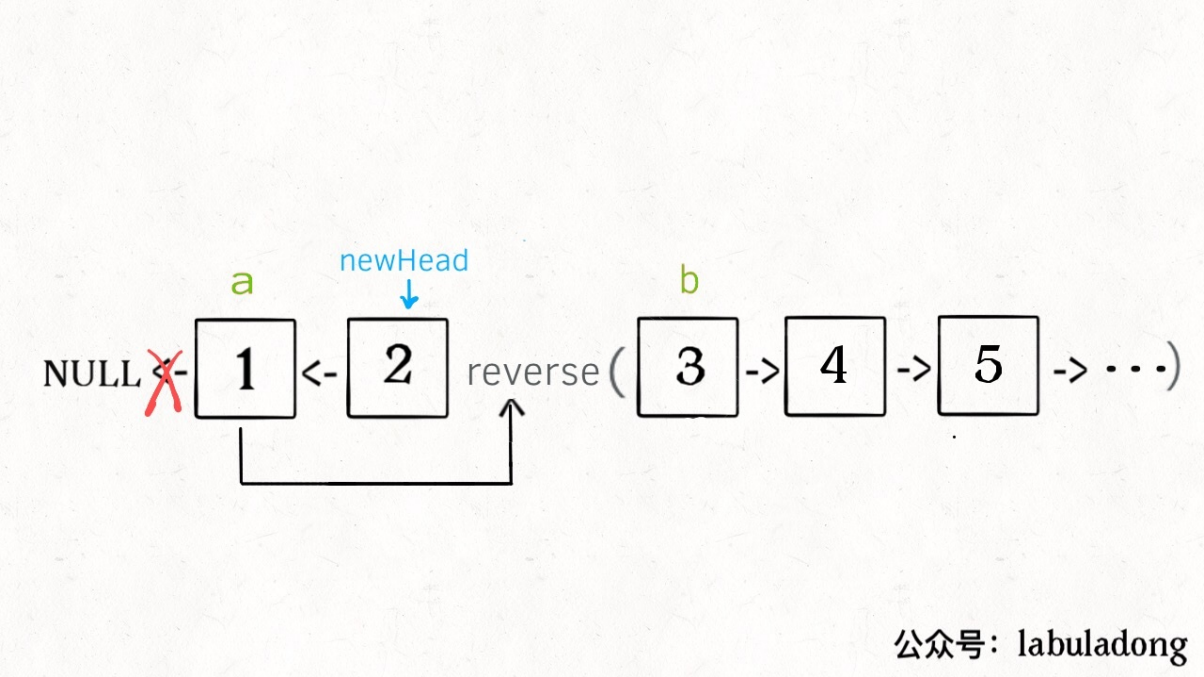
```
/** 反转区间 [a, b) 的元素，注意是左闭右开 */
ListNode reverse(ListNode a, ListNode b) {
    ListNode pre, cur, nxt;
    pre = null; cur = a; nxt = a;
    // while 终止的条件改一下就行了
    while (cur != b) {
        nxt = cur.next;
        cur.next = pre;
        pre = cur;
        cur = nxt;
    }
    // 返回反转后的头结点
    return pre;
}
```

现在我们迭代实现了反转部分链表的功能，接下来就按照之前的逻辑编写 `reverseKGroup` 函数即可：

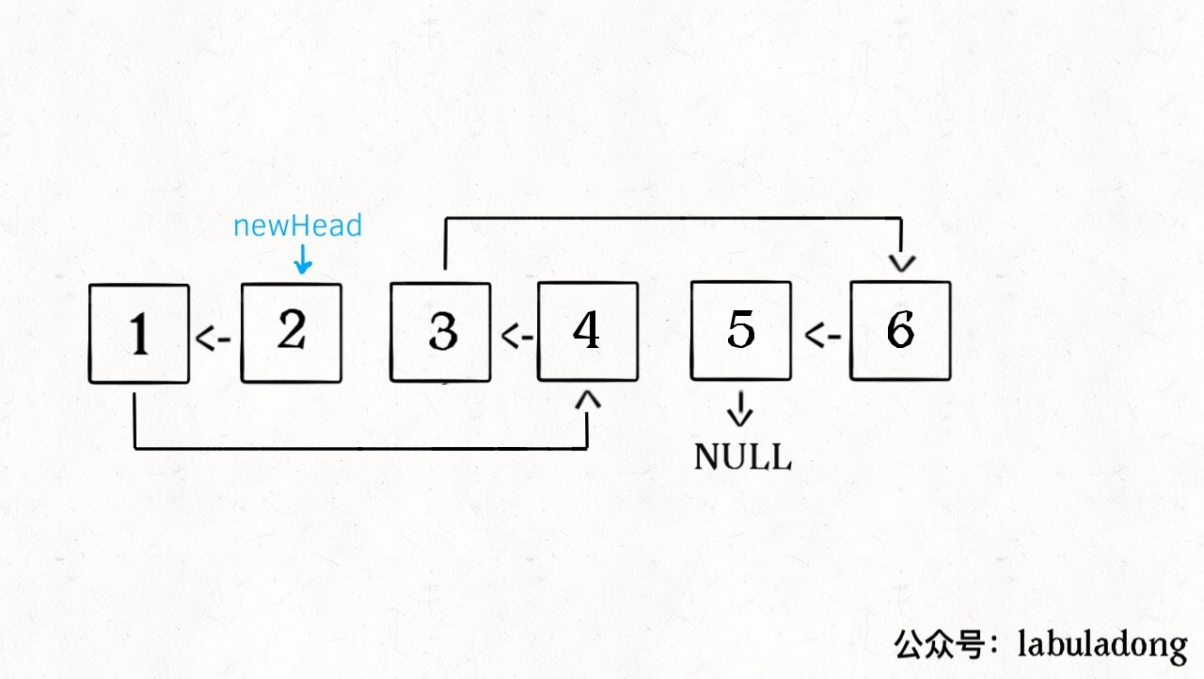
```
ListNode reverseKGroup(ListNode head, int k) {
    if (head == null) return null;
    // 区间 [a, b) 包含 k 个待反转元素
    ListNode a, b;
    a = b = head;
    for (int i = 0; i < k; i++) {
        // 不足 k 个，不需要反转，base case
        if (b == null) return head;
```

```
            b = b.next;
        }
        // 反转前 k 个元素
        ListNode newHead = reverse(a, b);
        // 递归反转后续链表并连接起来
        a.next = reverseKGroup(b, k);
        return newHead;
    }
```

解释一下 `for` 循环之后的几句代码，注意 `reverse` 函数是反转区间 `[a, b)`，所以情形是这样的：



递归部分就不展开了，整个函数递归完成之后就是这个结果，完全符合题意：



三、最后说两句

从阅读量上看，基本数据结构相关的算法文章看的人都不多，我想说这是要吃亏的。

大家喜欢看动态规划相关的问题，可能因为面试很常见，但就我个人理解，很多算法思想都是源于数据结构的。我们公众号的成名之作之一，「学习数据结构的框架思维」就提过，什么动规、回溯、分治算法，其实都是树的遍历，树这种结构它不就是个多叉链表吗？你能处理基本数据结构的问题，解决一般的算法问题应该也不会太费事。

那么如何分解问题、发现递归性质呢？这个只能多练习，也许后续可以专门写一篇文章来探讨一下，本文就到此为止吧，希望对大家有帮助！

如何判定括号合法性

对括号的合法性判断是一个很常见且实用的问题，比如说我们写的代码，编辑器和编译器都会检查括号是否正确闭合。而且我们的代码可能会包含三种括号 `[](){}`，判断起来有一点难度。

本文就来聊一道关于括号合法性判断的算法题，相信能加深你对**栈**这种数据结构的理解。

题目很简单，输入一个字符串，其中包含 `[](){}` 六种括号，请你判断这个字符串组成的括号是否合法。

```
Input: "()[]{}"
Output: true

Input: "([)]"
Output: false

Input: "{[]}"
Output: true
```

解决这个问题之前，我们先降低难度，思考一下，**如果只有一种括号 `()`**，应该如何判断字符串组成的括号是否合法呢？

一、处理一种括号

字符串中只有圆括号，如果想让括号字符串合法，那么必须做到：

每个右括号 `)` 的左边必须有一个左括号 `(` 和它匹配。

比如说字符串 `())(()` 中，中间的两个右括号**左边**就没有左括号匹配，所以这个括号组合是不合法的。

那么根据这个思路，我们可以写出算法：

```
bool isValid(string str) {
    // 待匹配的左括号数量
    int left = 0;
    for (char c : str) {
        if (c == '(')
            left++;
        else // 遇到右括号
            left--;

        if (left < 0)
            return false;
    }
    return left == 0;
}
```

如果只有圆括号，这样就能正确判断合法性。对于三种括号的情况，我一开始想模仿这个思路，定义三个变量 `left1`，`left2`，`left3` 分别处理每种括号，虽然要多写不少 if else 分支，但是似乎可以解决问题。

但实际上直接照搬这种思路是不行的，比如说只有一个括号的情况下 `(())` 是合法的，但是多种括号的情况下，`[(])` 显然是不合法的。

仅仅记录每种左括号出现的次数已经不能做出正确判断了，我们要加大存储的信息量，可以利用栈来模仿类似的思路。

二、处理多种括号

栈是一种先进后出的数据结构，处理括号问题的时候尤其有用。

我们这道题就用一个名为 `left` 的栈代替之前思路中的 `left` 变量，**遇到左括号就入栈，遇到右括号就去栈中寻找最近的左括号，看是否匹配**。

```
bool isValid(string str) {
    stack<char> left;
    for (char c : str) {
        if (c == '(' || c == '{' || c == '[')
            left.push(c);
        else // 字符 c 是右括号
```

```
            if (!left.empty() && leftOf(c) == left.top())
                left.pop();
            else
                // 和最近的左括号不匹配
                return false;
        }
        // 是否所有的左括号都被匹配了
        return left.empty();
    }

    char leftOf(char c) {
        if (c == '}') return '{';
        if (c == ')') return '(';
        return '[';
    }
```

如何寻找消失的元素

之前也有文章写过几个有趣的智力题，今天再聊一道巧妙的题目。

题目非常简单：

> 给定一个包含 `0, 1, 2, ..., n` 中 *n* 个数的序列，找出 0 .. *n* 中没有出现在序列中的那个数。
>
> **示例 1:**
>
> ```
> 输入: [3,0,1]
> 输出: 2
> ```
>
> **示例 2:**
>
> ```
> 输入: [9,6,4,2,3,5,7,0,1]
> 输出: 8
> ```

给一个长度为 n 的数组，其索引应该在 `[0,n)`，但是现在你要装进去 n + 1 个元素 `[0,n]`，那么肯定有一个元素装不下嘛，请你找出这个缺失的元素。

这道题不难的，我们应该很容易想到，把这个数组排个序，然后遍历一遍，不就很容易找到缺失的那个元素了吗？

或者说，借助数据结构的特性，用一个 HashSet 把数组里出现的数字都储存下来，再遍历 `[0,n]` 之间的数字，去 HashSet 中查询，也可以很容易查出那个缺失的元素。

排序解法的时间复杂度是 O(NlogN)，HashSet 的解法时间复杂度是 O(N)，但是还需要 O(N) 的空间复杂度存储 HashSet。

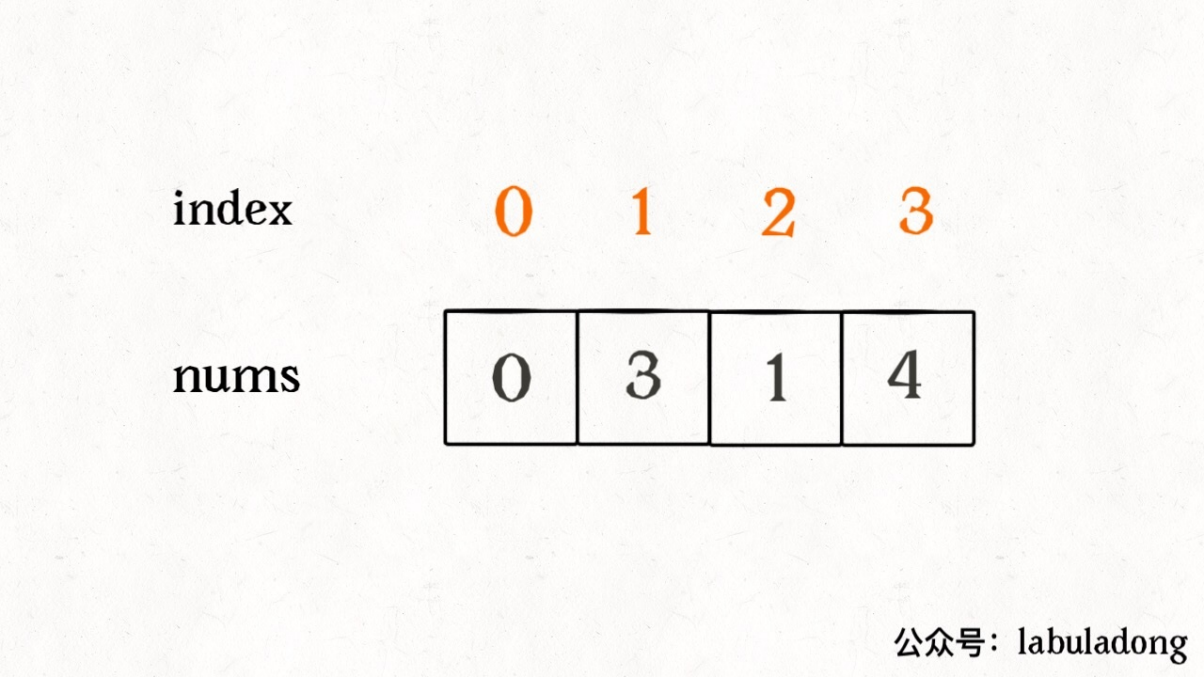
第三种方法是位运算。

对于异或运算（`^`），我们知道它有一个特殊性质：一个数和它本身做异或运算结果为 0，一个数和 0 做异或运算还是它本身。

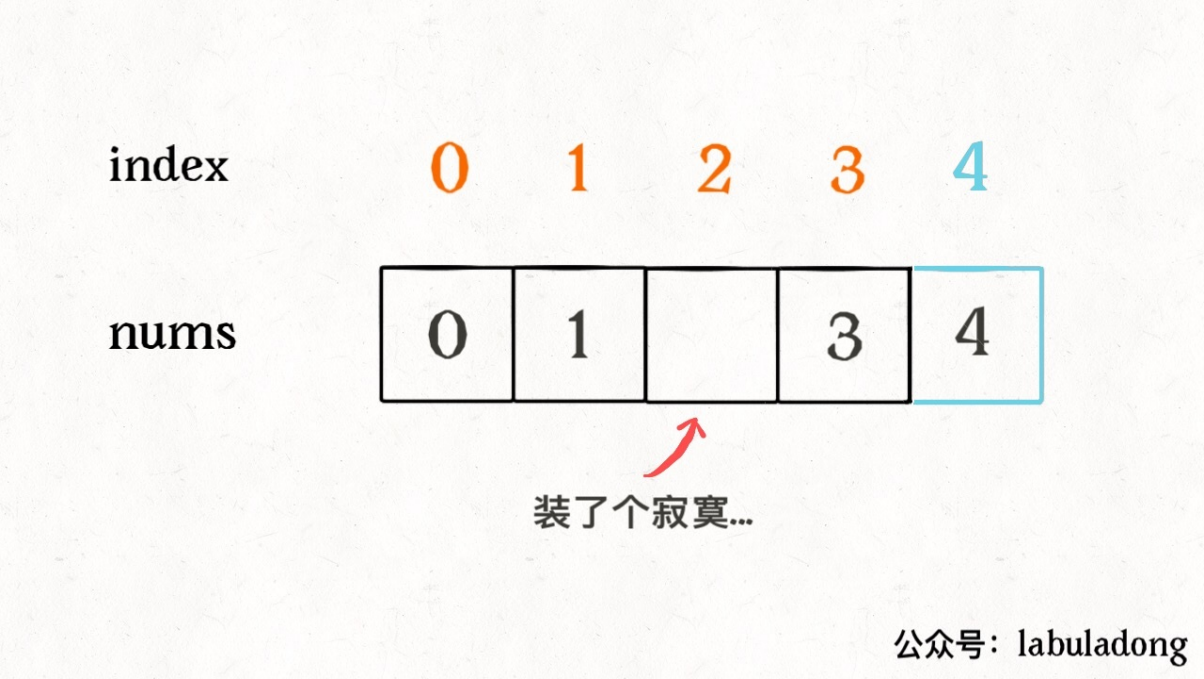
而且异或运算满足交换律和结合律，也就是说：

2 ^ 3 ^ 2 = 3 ^ (2 ^ 2) = 3 ^ 0 = 3

而这道题索就可以通过这些性质巧妙算出缺失的那个元素。比如说 `nums = [0,3,1,4]`：



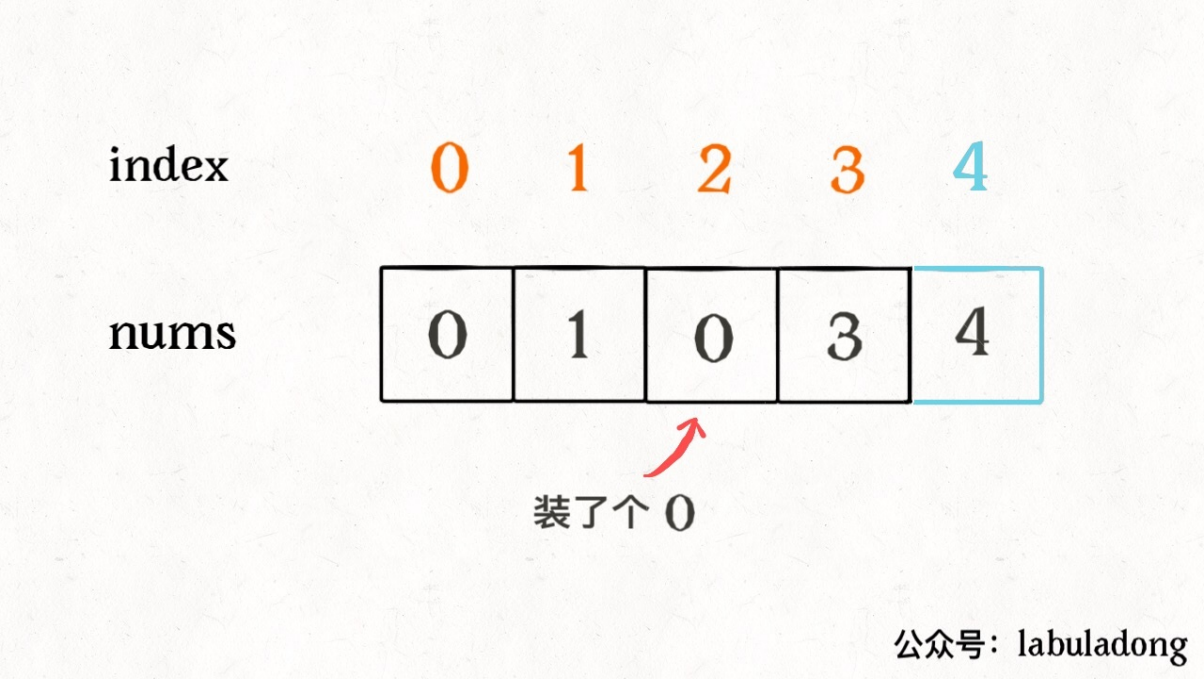
为了容易理解，我们假设先把索引补一位，然后让每个元素和自己相等的索引相对应：



这样做了之后，就可以发现除了缺失元素之外，所有的索引和元素都组成一对儿了，现在如果把这个落单的索引 2 找出来，也就找到了缺失的那个元素。

如何找这个落单的数字呢，**只要把所有的元素和索引做异或运算，成对儿的数字都会消为 0，只有这个落单的元素会剩下**，也就达到了我们的目的。

```
int missingNumber(int[] nums) {
    int n = nums.length;
    int res = 0;
    // 先和新补的索引异或一下
    res ^= n;
    // 和其他的元素、索引做异或
    for (int i = 0; i < n; i++)
        res ^= i ^ nums[i];
    return res;
}
```



由于异或运算满足交换律和结合律，所以总是能把成对儿的数字消去，留下缺失的那个元素的。

至此，时间复杂度 O(N)，空间复杂度 O(1)，已经达到了最优，我们是否就应该打道回府了呢？

如果这样想，说明我们受算法的毒害太深，随着我们学习的知识越来越多，反而容易陷入思维定式，这个问题其实还有一个特别简单的解法：**等差数列求和公式**。

题目的意思可以这样理解：现在有个等差数列 0, 1, 2,..., n，其中少了某一个数字，请你把它找出来。那这个数字不就是 `sum(0,1,..n) - sum(nums)` 嘛？

```
int missingNumber(int[] nums) {
    int n = nums.length;
    // 公式：(首项 + 末项) * 项数 / 2
    int expect = (0 + n) * (n + 1) / 2;

    int sum = 0;
    for (int x : nums)
        sum += x;
    return expect - sum;
```

你看，这种解法应该是最简单的，但说实话，我自己也没想到这个解法，而且我去问了几个大佬，他们也没想到这个最简单的思路。相反，如果去问一个初中生，他也许很快就能想到。

做到这一步了，我们是否就应该打道回府了呢？

如果这样想，说明我们对细节的把控还差点火候。在用求和公式计算 `expect` 时，你考虑过**整型溢出**吗？如果相乘的结果太大导致溢出，那么结果肯定是错误的。

刚才我们的思路是把两个和都加出来然后相减，为了避免溢出，干脆一边求和一边减算了。很类似刚才位运算解法的思路，仍然假设 `nums = [0,3,1,4]`，先补一位索引再让元素跟索引配对：

Index	0	1	2	3
Value	0	3	1	4

$$
\begin{aligned}
missing &= 4 \wedge (0 \wedge 0) \wedge (1 \wedge 3) \wedge (2 \wedge 1) \wedge (3 \wedge 4) \\
&= (4 \wedge 4) \wedge (0 \wedge 0) \wedge (1 \wedge 1) \wedge (3 \wedge 3) \wedge 2 \\
&= 0 \wedge 0 \wedge 0 \wedge 0 \wedge 2 \\
&= 2
\end{aligned}
$$

我们让每个索引减去其对应的元素，再把相减的结果加起来，不就是那个缺失的元素吗？

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    int res = 0;
    // 新补的索引
    res += n - 0;
    // 剩下索引和元素的差加起来
    for (int i = 0; i < n; i++)
        res += i - nums[i];
    return res;
}
```

由于加减法满足交换律和结合律，所以总是能把成对儿的数字消去，留下缺失的那个元素的。

至此这道算法题目经历九曲十八弯，终于再也没有什么坑了。

我们之前有两篇文章写了回文串和回文序列相关的问题。

寻找回文串的核心思想是从中心向两端扩展：

```
string palindrome(string& s, int l, int r) {
    // 防止索引越界
    while (l >= 0 && r < s.size()
            && s[l] == s[r]) {
        // 向两边展开
        l--; r++;
    }
    // 返回以 s[l] 和 s[r] 为中心的最长回文串
    return s.substr(l + 1, r - l - 1);
}
```

因为回文串长度可能为奇数也可能是偶数，长度为奇数时只存在一个中心点，而长度为偶数时存在两个中心点，所以上面这个函数需要传入 `l` 和 `r` 。

而**判断**一个字符串是不是回文串就简单很多，不需要考虑奇偶情况，只需要「双指针技巧」，从两端向中间逼近即可：

```
bool isPalindrome(string s) {
    int left = 0, right = s.length - 1;
    while (left < right) {
        if (s[left] != s[right])
            return false;
        left++; right--;
    }
    return true;
}
```

以上代码很好理解吧，**因为回文串是对称的，所以正着读和倒着读应该是一样的，这一特点是解决回文串问题的关键**。

下面扩展这一最简单的情况，来解决：如何判断一个「单链表」是不是回文。

一、判断回文单链表

输入一个单链表的头结点，判断这个链表中的数字是不是回文：

```
/**
 * 单链表节点的定义：
 * public class ListNode {
 *     int val;
 *     ListNode next;
 * }
 */

boolean isPalindrome(ListNode head);

输入：1->2->null
输出：false

输入：1->2->2->1->null
输出：true
```

这道题的关键在于，单链表无法倒着遍历，无法使用双指针技巧。那么最简单的办法就是，把原始链表反转存入一条新的链表，然后比较这两条链表是否相同。关于如何反转链表，可以参见前文「递归操作链表」。

其实，**借助二叉树后序遍历的思路，不需要显式反转原始链表也可以倒序遍历链表**，下面来具体聊聊。

对于二叉树的几种遍历方式，我们再熟悉不过了：

```
void traverse(TreeNode root) {
    // 前序遍历代码
    traverse(root.left);
    // 中序遍历代码
    traverse(root.right);
    // 后序遍历代码
}
```

在「学习数据结构的框架思维」中说过，链表兼具递归结构，树结构不过是链表的衍生。那么，**链表其实也可以有前序遍历和后序遍历**：

```
void traverse(ListNode head) {
    // 前序遍历代码
    traverse(head.next);
    // 后序遍历代码
}
```

这个框架有什么指导意义呢？如果我想正序打印链表中的 `val` 值，可以在前序遍历位置写代码；反之，如果想倒序遍历链表，就可以在后序遍历位置操作：

```
/* 倒序打印单链表中的元素值 */
void traverse(ListNode head) {
    if (head == null) return;
    traverse(head.next);
    // 后序遍历代码
    print(head.val);
}
```

说到这了，其实可以稍作修改，模仿双指针实现回文判断的功能：

```
// 左侧指针
ListNode left;

boolean isPalindrome(ListNode head) {
    left = head;
    return traverse(head);
}

boolean traverse(ListNode right) {
    if (right == null) return true;
    boolean res = traverse(right.next);
    // 后序遍历代码
    res = res && (right.val == left.val);
    left = left.next;
    return res;
}
```

这么做的核心逻辑是什么呢？**实际上就是把链表节点放入一个栈，然后再拿出来，这时候元素顺序就是反的**，只不过我们利用的是递归函数的堆栈而已。

【pdf/mobi格式不支持GIF:回文链表/1.gif】 请查看【关于本小抄及作者】章节的解决方案

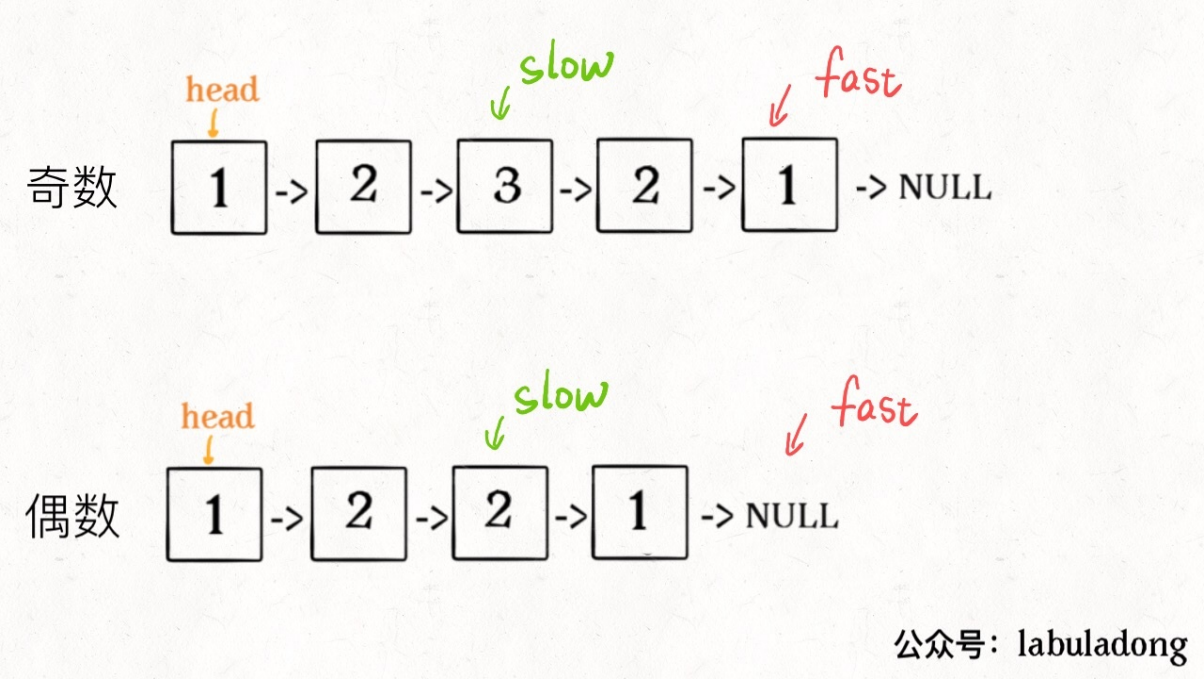
当然，无论造一条反转链表还是利用后续遍历，算法的时间和空间复杂度都是 O(N)。下面我们想想，能不能不用额外的空间，解决这个问题呢？

二、优化空间复杂度

更好的思路是这样的：

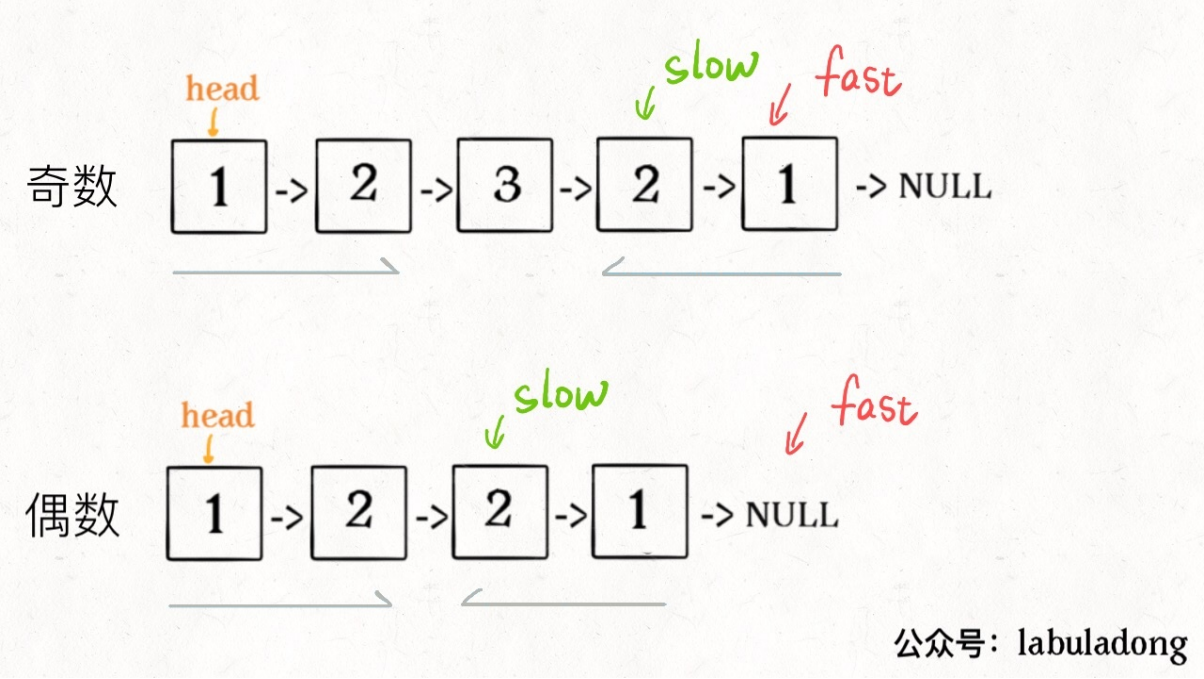
1、先通过「双指针技巧」中的快慢指针来找到链表的中点：

```
ListNode slow, fast;
slow = fast = head;
while (fast != null && fast.next != null) {
    slow = slow.next;
    fast = fast.next.next;
}
// slow 指针现在指向链表中点
```



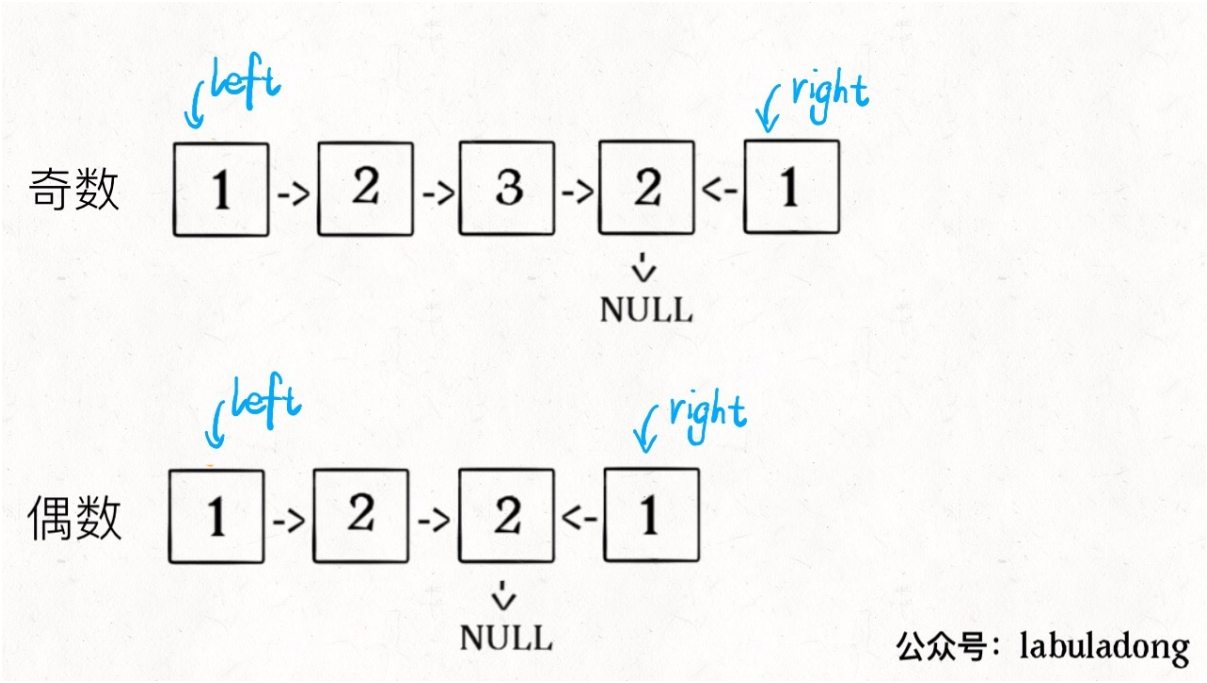
2、如果 `fast` 指针没有指向 `null`，说明链表长度为奇数，`slow` 还要再前进一步：

```
if (fast != null)
    slow = slow.next;
```



3、从 `slow` 开始反转后面的链表，现在就可以开始比较回文串了：

```
ListNode left = head;
ListNode right = reverse(slow);

while (right != null) {
    if (left.val != right.val)
        return false;
    left = left.next;
    right = right.next;
}
return true;
```



至此，把上面 3 段代码合在一起就高效地解决这个问题了，其中 `reverse` 函数很容易实现：

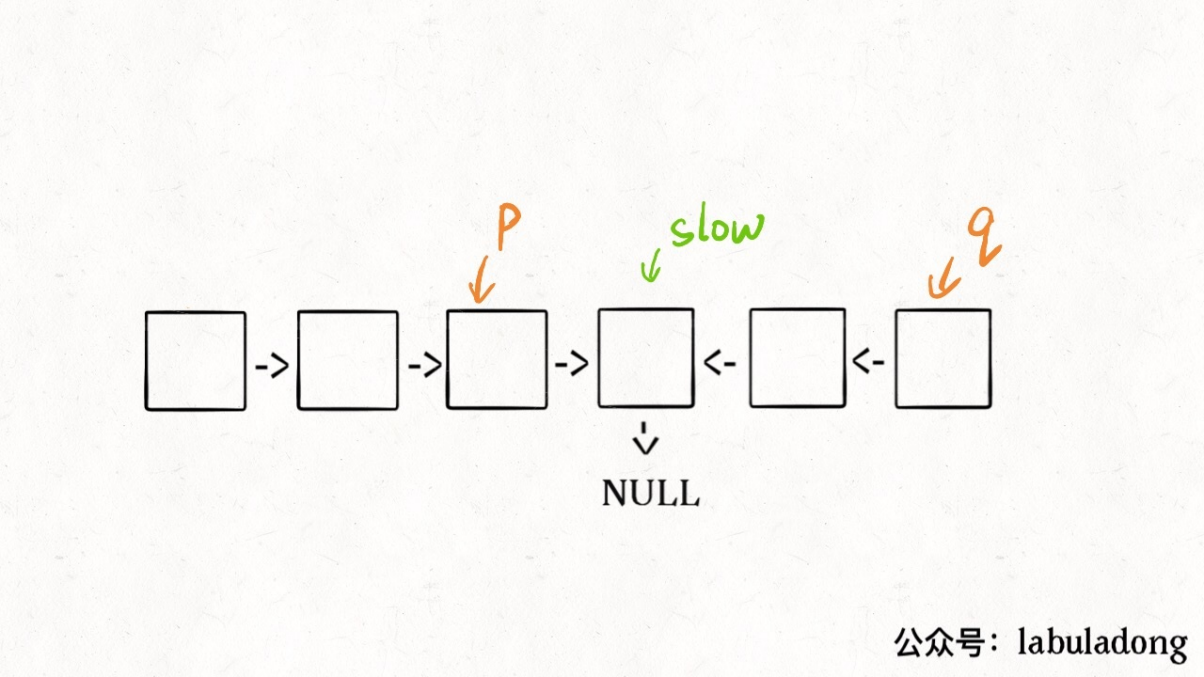
```
ListNode reverse(ListNode head) {
    ListNode pre = null, cur = head;
    while (cur != null) {
        ListNode next = cur.next;
        cur.next = pre;
        pre = cur;
        cur = next;
    }
    return pre;
}
```

【pdf/mobi格式不支持GIF:kgroup/8.gif】 请查看【关于本小抄及作者】章节的解决方案

算法总体的时间复杂度 O(N)，空间复杂度 O(1)，已经是最优的了。

我知道肯定有读者会问：这种解法虽然高效，但破坏了输入链表的原始结构，能不能避免这个瑕疵呢？

其实这个问题很好解决，关键在于得到 `p, q` 这两个指针位置：



这样，只要在函数 return 之前加一段代码即可恢复原先链表顺序：

```
p.next = reverse(q);
```

篇幅所限，我就不写了，读者可以自己尝试一下。

三、最后总结

首先，寻找回文串是从中间向两端扩展，判断回文串是从两端向中间收缩。对于单链表，无法直接倒序遍历，可以造一条新的反转链表，可以利用链表的后序遍历，也可以用栈结构倒序处理单链表。

具体到回文链表的判断问题，由于回文的特殊性，可以不完全反转链表，而是仅仅反转部分链表，将空间复杂度降到 O(1)。

如何调度考生的座位

这是 LeetCode 第 885 题，有趣且具有一定技巧性。这种题目并不像动态规划这类算法拼智商，而是看你对常用数据结构的理解和写代码的水平，个人认为值得重视和学习。

另外说句题外话，很多读者都问，算法框架是如何总结出来的，其实框架反而是慢慢从细节里抠出来的。希望大家看了我们的文章之后，最好能抽时间把相关的问题亲自做一做，纸上得来终觉浅，绝知此事要躬行嘛。

先来描述一下题目：假设有一个考场，考场有一排共 `N` 个座位，索引分别是 `[0..N-1]`，考生会**陆续**进入考场考试，并且可能在**任何时候**离开考场。

你作为考官，要安排考生们的座位，满足：**每当一个学生进入时，你需要最大化他和最近其他人的距离；如果有多个这样的座位，安排到他到索引最小的那个座位**。这很符合实际情况对吧，

也就是请你实现下面这样一个类：

```
class ExamRoom {
    // 构造函数，传入座位总数 N
    public ExamRoom(int N);
    // 来了一名考生，返回你给他分配的座位
    public int seat();
    // 坐在 p 位置的考生离开了
    // 可以认为 p 位置一定坐有考生
    public void leave(int p);
}
```

比方说考场有 5 个座位，分别是 `[0..4]`：

第一名考生进入时（调用 `seat()`），坐在任何位置都行，但是要给他安排索引最小的位置，也就是返回位置 0。

第二名学生进入时（再调用 `seat()`），要和旁边的人距离最远，也就是返回位置 4。

第三名学生进入时，要和旁边的人距离最远，应该做到中间，也就是座位 2。

如果再进一名学生，他可以坐在座位 1 或者 3，取较小的索引 1。

以此类推。

刚才所说的情况，没有调用 `leave` 函数，不过读者肯定能够发现规律：

如果将每两个相邻的考生看做线段的两端点，新安排考生就是找最长的线段，然后让该考生在中间把这个线段「二分」，中点就是给他分配的座位。`leave(p)` 其实就是去除端点 `p`，使得相邻两个线段合并为一个。

核心思路很简单对吧，所以这个问题实际上实在考察你对数据结构的理解。对于上述这个逻辑，你用什么数据结构来实现呢？

一、思路分析

根据上述思路，首先需要把坐在教室的学生抽象成线段，我们可以简单的用一个大小为 2 的数组表示。

另外，思路需要我们找到「最长」的线段，还需要去除线段，增加线段。

但凡遇到在动态过程中取最值的要求，肯定要使用有序数据结构，我们常用的数据结构就是二叉堆和平衡二叉搜索树了。二叉堆实现的优先级队列取最值的时间复杂度是 O(logN)，但是只能删除最大值。平衡二叉树也可以取最值，也可以修改、删除任意一个值，而且时间复杂度都是 O(logN)。

综上，二叉堆不能满足 `leave` 操作，应该使用平衡二叉树。所以这里我们会用到 Java 的一种数据结构 `TreeSet`，这是一种有序数据结构，底层由红黑树维护有序性。

这里顺便提一下，一说到集合（Set）或者映射（Map），有的读者可能就想当然的认为是哈希集合（HashSet）或者哈希表（HashMap），这样理解是有点问题的。

因为哈希集合/映射底层是由哈希函数和数组实现的，特性是遍历无固定顺序，但是操作效率高，时间复杂度为 O(1)。

而集合/映射还可以依赖其他底层数据结构，常见的就是红黑树（一种平衡二叉搜索树），特性是自动维护其中元素的顺序，操作效率是 O(logN)。这种一般称为「有序集合/映射」。

我们使用的 `TreeSet` 就是一个有序集合，目的就是为了保持线段长度的有序性，快速查找最大线段，快速删除和插入。

二、简化问题

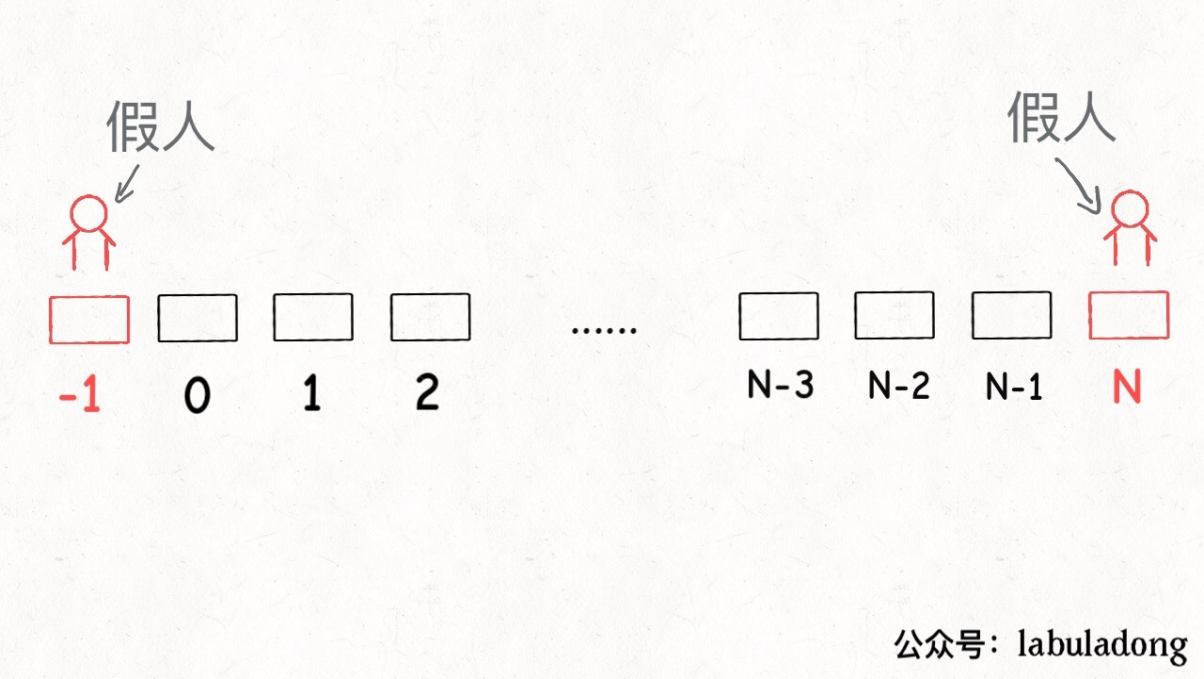
首先，如果有多个可选座位，需要选择索引最小的座位对吧？**我们先简化一下问题，暂时不管这个要求**，实现上述思路。

这个问题还用到一个常用的编程技巧，就是使用一个「虚拟线段」让算法正确启动，这就和链表相关的算法需要「虚拟头结点」一个道理。

```
// 将端点 p 映射到以 p 为左端点的线段
private Map<Integer, int[]> startMap;
// 将端点 p 映射到以 p 为右端点的线段
private Map<Integer, int[]> endMap;
// 根据线段长度从小到大存放所有线段
private TreeSet<int[]> pq;
private int N;

public ExamRoom(int N) {
    this.N = N;
    startMap = new HashMap<>();
    endMap = new HashMap<>();
    pq = new TreeSet<>((a, b) -> {
        // 算出两个线段的长度
        int distA = distance(a);
        int distB = distance(b);
```

```
            // 长度更长的更大，排后面
            return distA - distB;
        });
        // 在有序集合中先放一个虚拟线段
        addInterval(new int[] {-1, N});
    }

    /* 去除一个线段 */
    private void removeInterval(int[] intv) {
        pq.remove(intv);
        startMap.remove(intv[0]);
        endMap.remove(intv[1]);
    }

    /* 增加一个线段 */
    private void addInterval(int[] intv) {
        pq.add(intv);
        startMap.put(intv[0], intv);
        endMap.put(intv[1], intv);
    }

    /* 计算一个线段的长度 */
    private int distance(int[] intv) {
        return intv[1] - intv[0] - 1;
    }
```

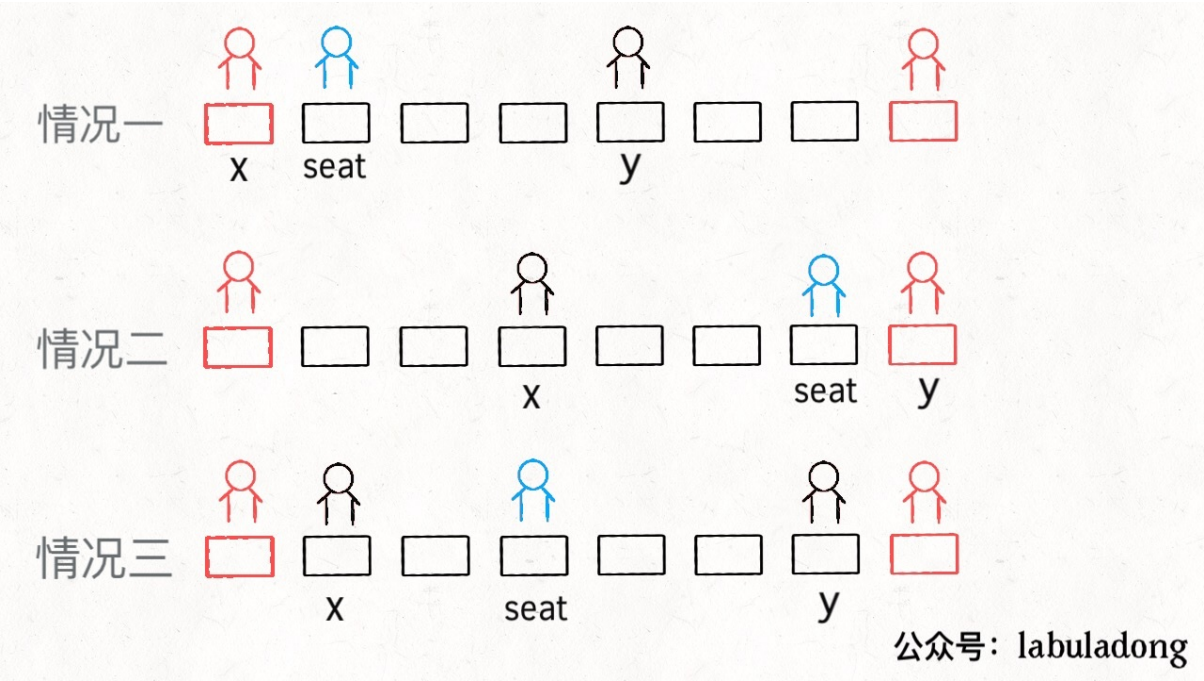
「虚拟线段」其实就是为了将所有座位表示为一个线段：



有了上述铺垫，主要 API `seat` 和 `leave` 就可以写了：

```
public int seat() {
    // 从有序集合拿出最长的线段
    int[] longest = pq.last();
    int x = longest[0];
    int y = longest[1];
    int seat;
    if (x == -1) { // 情况一
        seat = 0;
    } else if (y == N) { // 情况二
        seat = N - 1;
    } else { // 情况三
        seat = (y - x) / 2 + x;
    }
    // 将最长的线段分成两段
    int[] left = new int[] {x, seat};
    int[] right = new int[] {seat, y};
    removeInterval(longest);
    addInterval(left);
    addInterval(right);
    return seat;
}

public void leave(int p) {
    // 将 p 左右的线段找出来
```

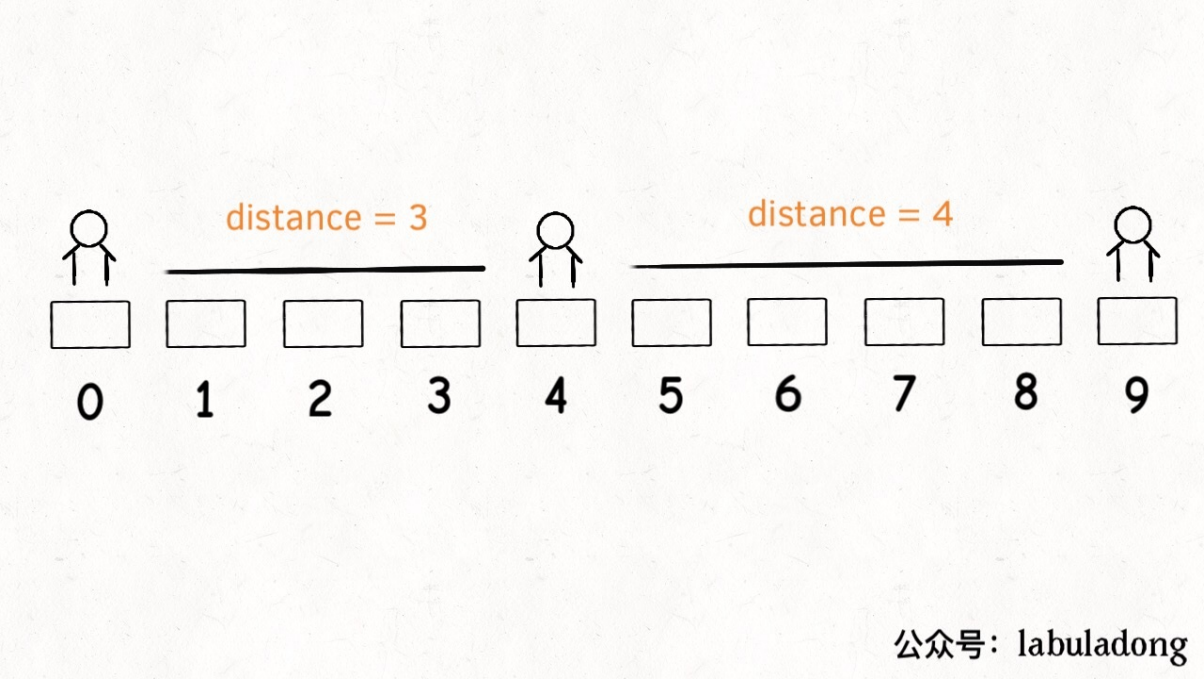
```
        int[] right = startMap.get(p);
        int[] left = endMap.get(p);
        // 合并两个线段成为一个线段
        int[] merged = new int[] {left[0], right[1]};
        removeInterval(left);
        removeInterval(right);
        addInterval(merged);
    }
```



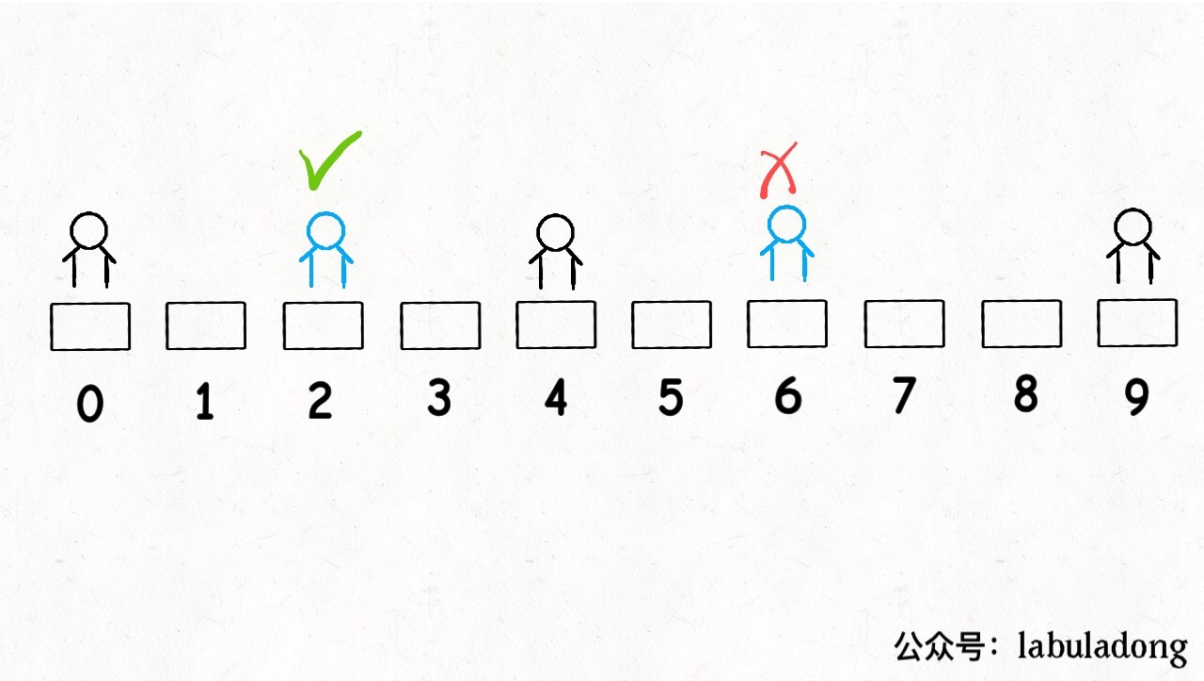
至此，算法就基本实现了，代码虽多，但思路很简单：找最长的线段，从中间分隔成两段，中点就是 `seat()` 的返回值；找 `p` 的左右线段，合并成一个线段，这就是 `leave(p)` 的逻辑。

三、进阶问题

但是，题目要求多个选择时选择索引最小的那个座位，我们刚才忽略了这个问题。比如下面这种情况会出错：



现在有序集合里有线段 `[0,4]` 和 `[4,9]`，那么最长线段 `longest` 就是后者，按照 `seat` 的逻辑，就会分割 `[4,9]`，也就是返回座位 6。但正确答案应该是座位 2，因为 2 和 6 都满足最大化相邻考生距离的条件，二者应该取较小的。

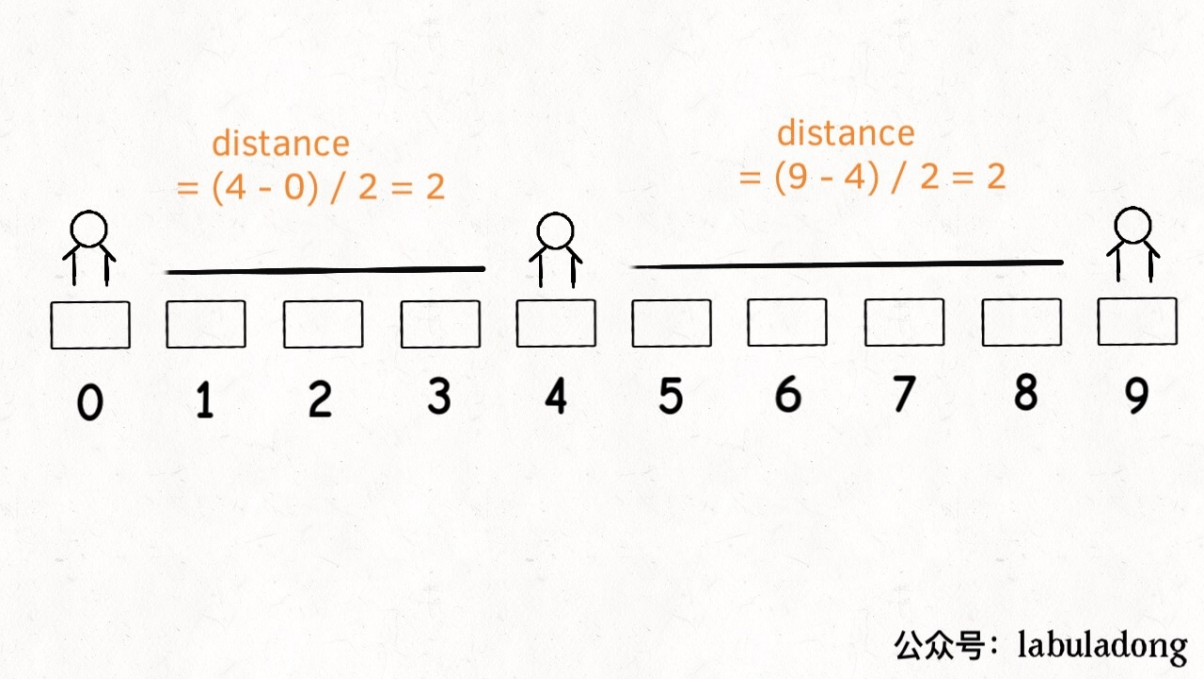


遇到题目的这种要求，解决方式就是修改有序数据结构的排序方式。具体到这个问题，就是修改 `TreeMap` 的比较函数逻辑：

```
pq = new TreeSet<>((a, b) -> {
    int distA = distance(a);
    int distB = distance(b);
    // 如果长度相同，就比较索引
    if (distA == distB)
        return b[0] - a[0];
    return distA - distB;
});
```

除此之外，还要改变 `distance` 函数，**不能简单地让它计算一个线段两个端点间的长度，而是让它计算该线段中点和端点之间的长度**。

```
private int distance(int[] intv) {
    int x = intv[0];
    int y = intv[1];
    if (x == -1) return y;
    if (y == N) return N - 1 - x;
    // 中点和端点之间的长度
    return (y - x) / 2;
}
```



这样，`[0,4]` 和 `[4,9]` 的 `distance` 值就相等了，算法会比较二者的索引，取较小的线段进行分割。到这里，这道算法题目算是完全解决了。

四、最后总结

本文聊的这个问题其实并不算难，虽然看起来代码很多。核心问题就是考察有序数据结构的理解和使用，来梳理一下。

处理动态问题一般都会用到有序数据结构，比如平衡二叉搜索树和二叉堆，二者的时间复杂度差不多，但前者支持的操作更多。

既然平衡二叉搜索树这么好用，还用二叉堆干嘛呢？因为二叉堆底层就是数组，实现简单啊，详见旧文「二叉堆详解」。你实现个红黑树试试？操作复杂，而且消耗的空间相对来说会多一些。具体问题，还是要选择恰当的数据结构来解决。

希望本文对大家有帮助。

Union-Find算法详解

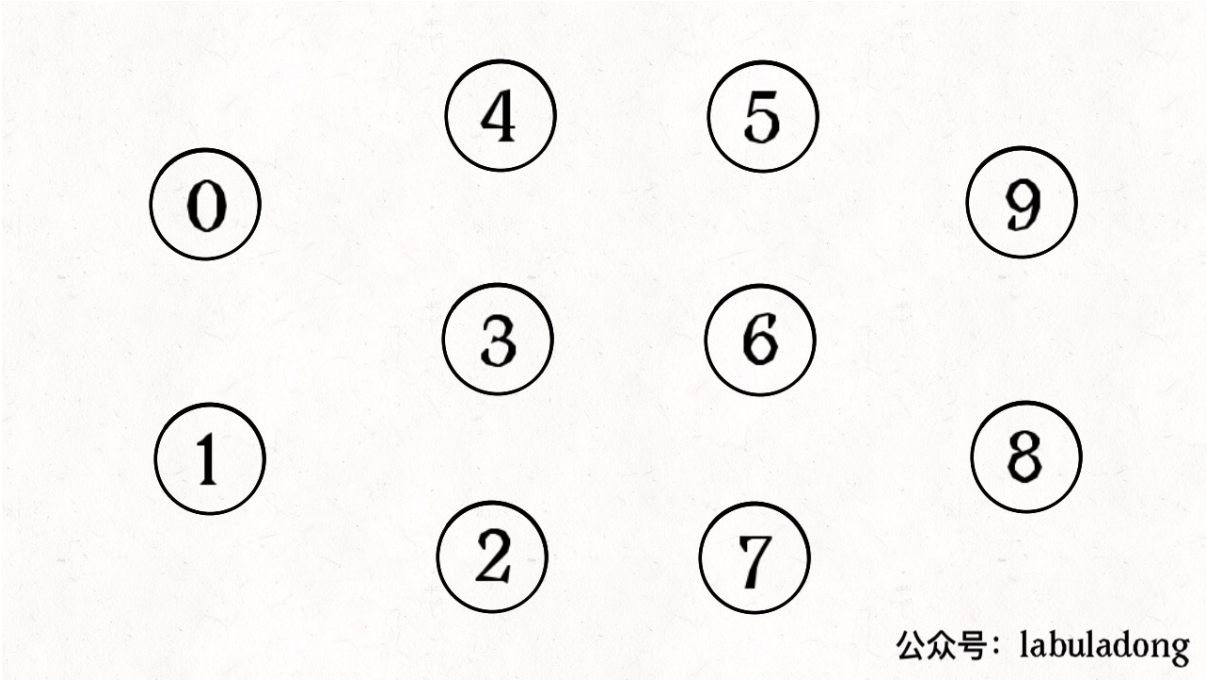
今天讲讲 Union-Find 算法，也就是常说的并查集算法，主要是解决图论中「动态连通性」问题的。名词很高端，其实特别好理解，等会解释，另外这个算法的应用都非常有趣。

说起这个 Union-Find，应该算是我的「启蒙算法」了，因为《算法4》的开头就介绍了这款算法，可是把我秀翻了，感觉好精妙啊！后来刷了 LeetCode，并查集相关的算法题目都非常有意思，而且《算法4》给的解法竟然还可以进一步优化，只要加一个微小的修改就可以把时间复杂度降到 O(1)。

废话不多说，直接上干货，先解释一下什么叫动态连通性吧。

一、问题介绍

简单说，动态连通性其实可以抽象成给一幅图连线。比如下面这幅图，总共有 10 个节点，他们互不相连，分别用 0~9 标记：



现在我们的 Union-Find 算法主要需要实现这两个 API：

```
class UF {
    /* 将 p 和 q 连接 */
    public void union(int p, int q);
    /* 判断 p 和 q 是否连通 */
    public boolean connected(int p, int q);
    /* 返回图中有多少个连通分量 */
    public int count();
}
```

这里所说的「连通」是一种等价关系，也就是说具有如下三个性质：

1、自反性：节点 `p` 和 `p` 是连通的。

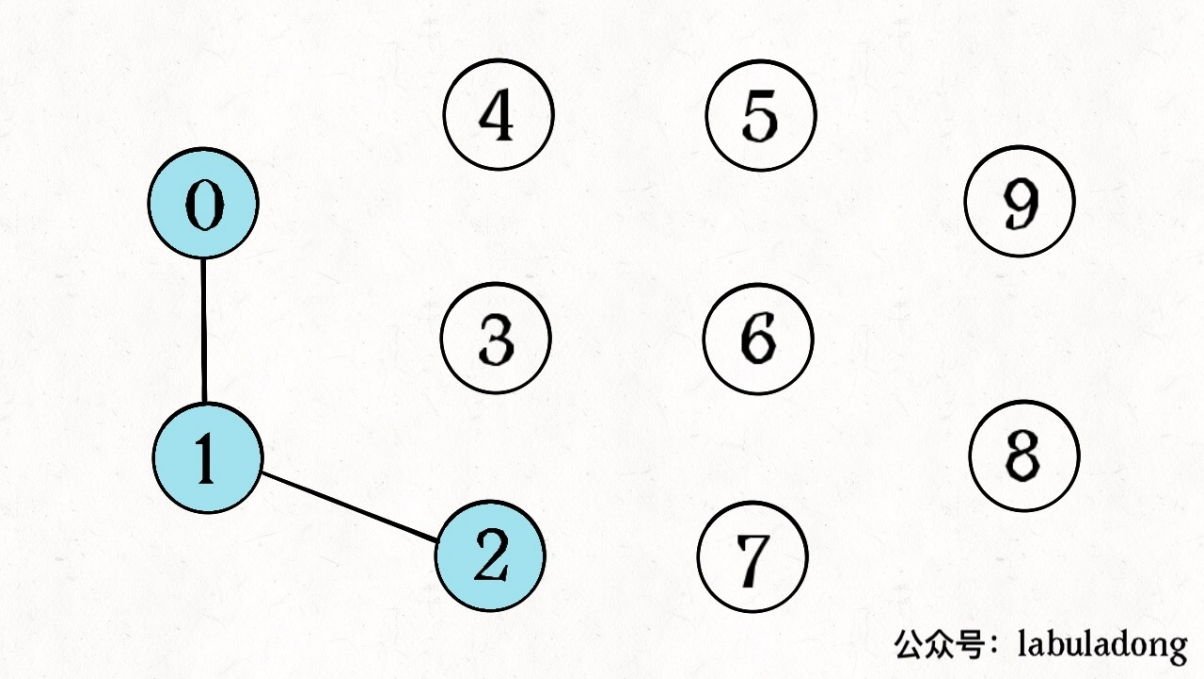
2、对称性：如果节点 `p` 和 `q` 连通，那么 `q` 和 `p` 也连通。

3、传递性：如果节点 `p` 和 `q` 连通，`q` 和 `r` 连通，那么 `p` 和 `r` 也连通。

比如说之前那幅图，0～9 任意两个**不同**的点都不连通，调用 `connected` 都会返回 false，连通分量为 10 个。

如果现在调用 `union(0, 1)`，那么 0 和 1 被连通，连通分量降为 9 个。

再调用 `union(1, 2)`，这时 0,1,2 都被连通，调用 `connected(0, 2)` 也会返回 true，连通分量变为 8 个。



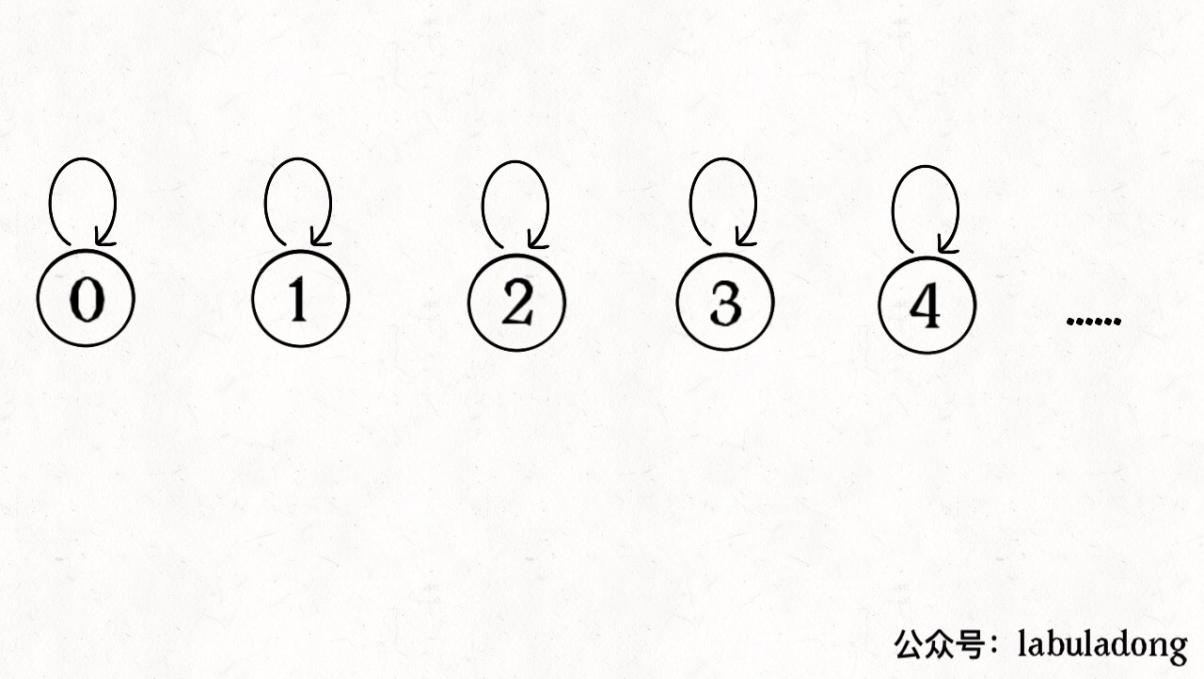
判断这种「等价关系」非常实用，比如说编译器判断同一个变量的不同引用，比如社交网络中的朋友圈计算等等。

这样，你应该大概明白什么是动态连通性了，Union-Find 算法的关键就在于 `union` 和 `connected` 函数的效率。那么用什么模型来表示这幅图的连通状态呢？用什么数据结构来实现代码呢？

二、基本思路

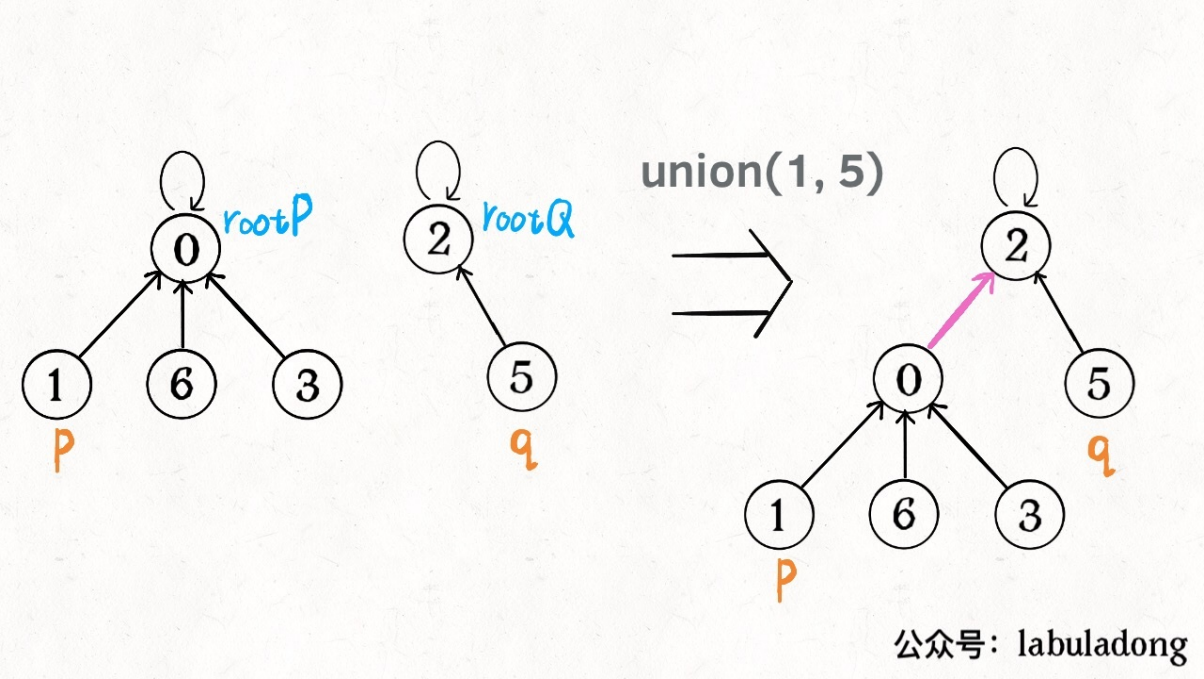
注意我刚才把「模型」和具体的「数据结构」分开说，这么做是有原因的。因为我们使用森林（若干棵树）来表示图的动态连通性，用数组来具体实现这个森林。

怎么用森林来表示连通性呢？我们设定树的每个节点有一个指针指向其父节点，如果是根节点的话，这个指针指向自己。比如说刚才那幅 10 个节点的图，一开始的时候没有相互连通，就是这样：



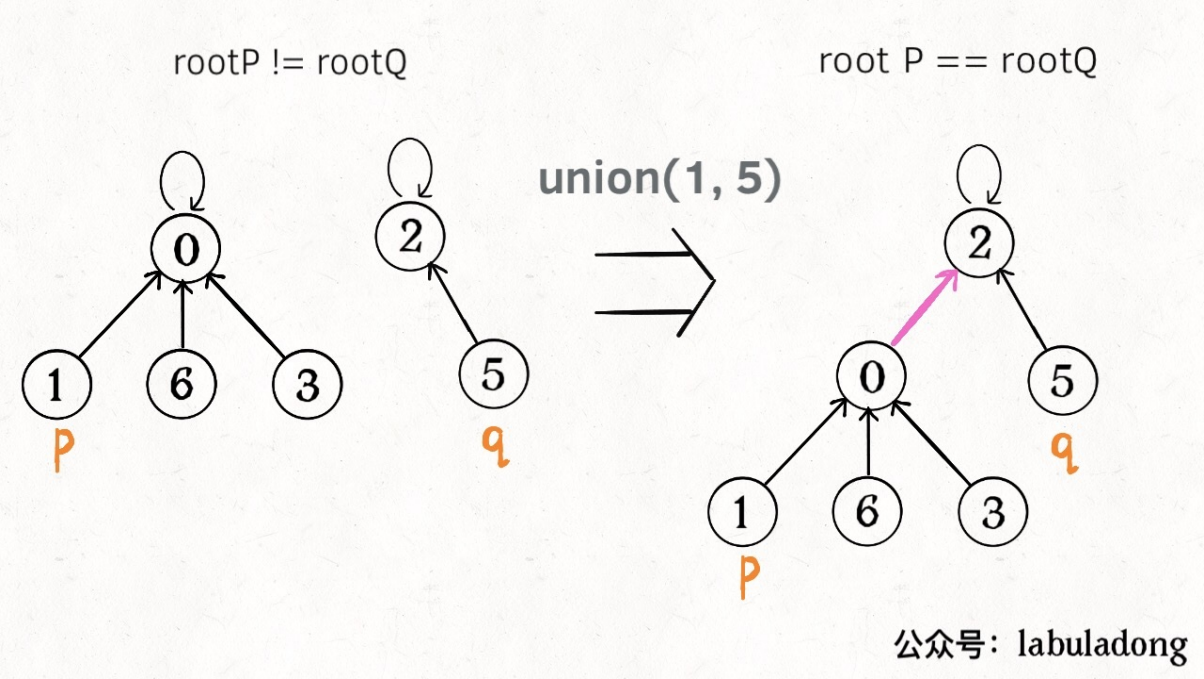

```
class UF {
    // 记录连通分量
    private int count;
    // 节点 x 的节点是 parent[x]
    private int[] parent;

    /* 构造函数，n 为图的节点总数 */
    public UF(int n) {
        // 一开始互不连通
        this.count = n;
        // 父节点指针初始指向自己
        parent = new int[n];
        for (int i = 0; i < n; i++)
            parent[i] = i;
    }

    /* 其他函数 */
}
```

如果某两个节点被连通，则让其中的（任意）一个节点的根节点接到另一个节点的根节点上：




```
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;
    // 将两棵树合并为一棵
    parent[rootP] = rootQ;
    // parent[rootQ] = rootP 也一样
    count--; // 两个分量合二为一
}

/* 返回某个节点 x 的根节点 */
private int find(int x) {
    // 根节点的 parent[x] == x
    while (parent[x] != x)
        x = parent[x];
    return x;
}

/* 返回当前的连通分量个数 */
public int count() {
    return count;
}
```

这样，如果节点 `p` 和 `q` 连通的话，它们一定拥有相同的根节点：

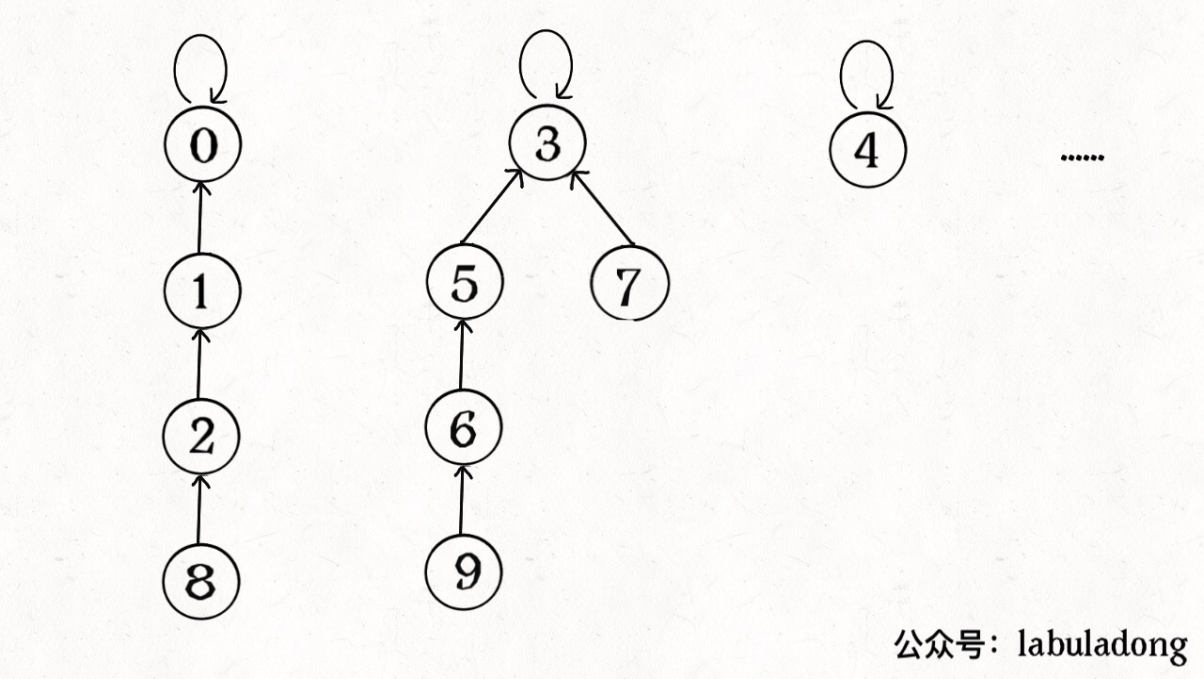



```
public boolean connected(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    return rootP == rootQ;
}
```

至此，Union-Find 算法就基本完成了。是不是很神奇？竟然可以这样使用数组来模拟出一个森林，如此巧妙的解决这个比较复杂的问题！

那么这个算法的复杂度是多少呢？我们发现，主要 API `connected` 和 `union` 中的复杂度都是 `find` 函数造成的，所以说它们的复杂度和 `find` 一样。

`find` 主要功能就是从某个节点向上遍历到树根，其时间复杂度就是树的高度。我们可能习惯性地认为树的高度就是 `logN`，但这并不一定。`logN` 的高度只存在于平衡二叉树，对于一般的树可能出现极端不平衡的情况，使得「树」几乎退化成「链表」，树的高度最坏情况下可能变成 `N`。



所以说上面这种解法，`find`，`union`，`connected`的时间复杂度都是 O(N)。这个复杂度很不理想的，你想图论解决的都是诸如社交网络这样数据规模巨大的问题，对于`union`和`connected`的调用非常频繁，每次调用需要线性时间完全不可忍受。

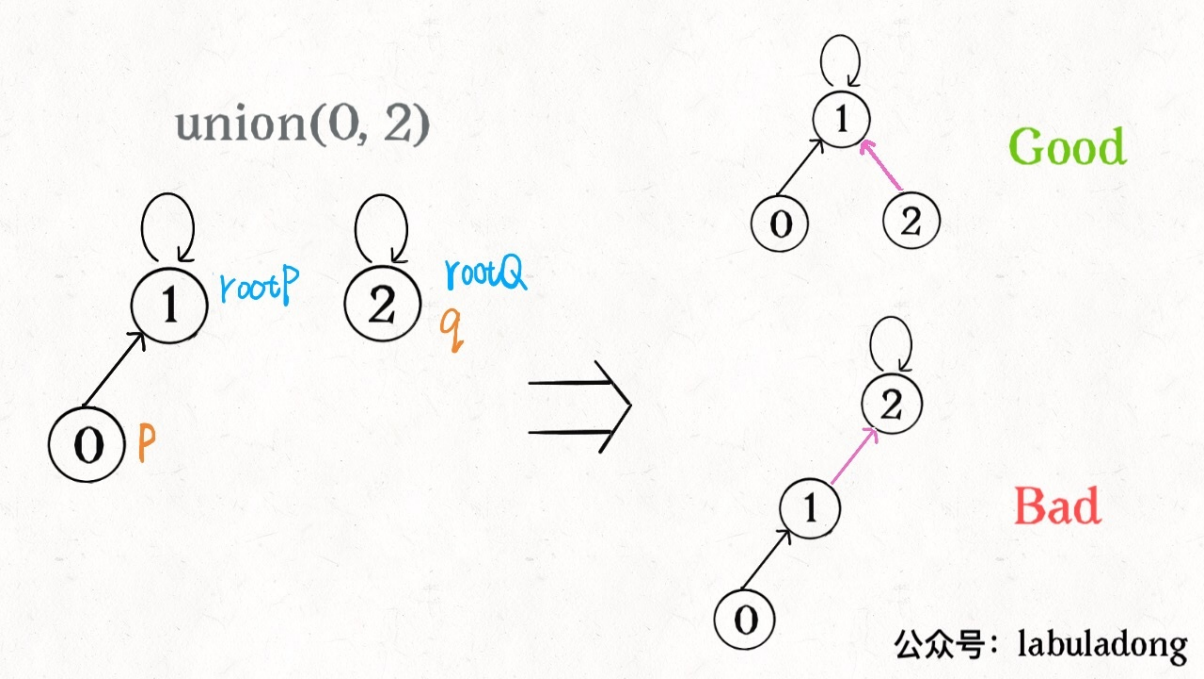
问题的关键在于，如何想办法避免树的不平衡呢？只需要略施小计即可。

三、平衡性优化

我们要知道哪种情况下可能出现不平衡现象，关键在于`union`过程：

```
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;
    // 将两棵树合并为一棵
    parent[rootP] = rootQ;
    // parent[rootQ] = rootP 也可以
    count--;
```

我们一开始就是简单粗暴的把 `p` 所在的树接到 `q` 所在的树的根节点下面，那么这里就可能出现「头重脚轻」的不平衡状况，比如下面这种局面：



长此以往，树可能生长得很不平衡。**我们其实是希望，小一些的树接到大一些的树下面，这样就能避免头重脚轻，更平衡一些**。解决方法是额外使用一个 `size` 数组，记录每棵树包含的节点数，我们不妨称为「重量」：

```
class UF {
    private int count;
    private int[] parent;
    // 新增一个数组记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        // 最初每棵树只有一个节点
        // 重量应该初始化 1
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
    /* 其他函数 */
```

```
}
```

比如说 `size[3] = 5` 表示，以节点 `3` 为根的那棵树，总共有 `5` 个节点。这样我们可以修改一下 `union` 方法：

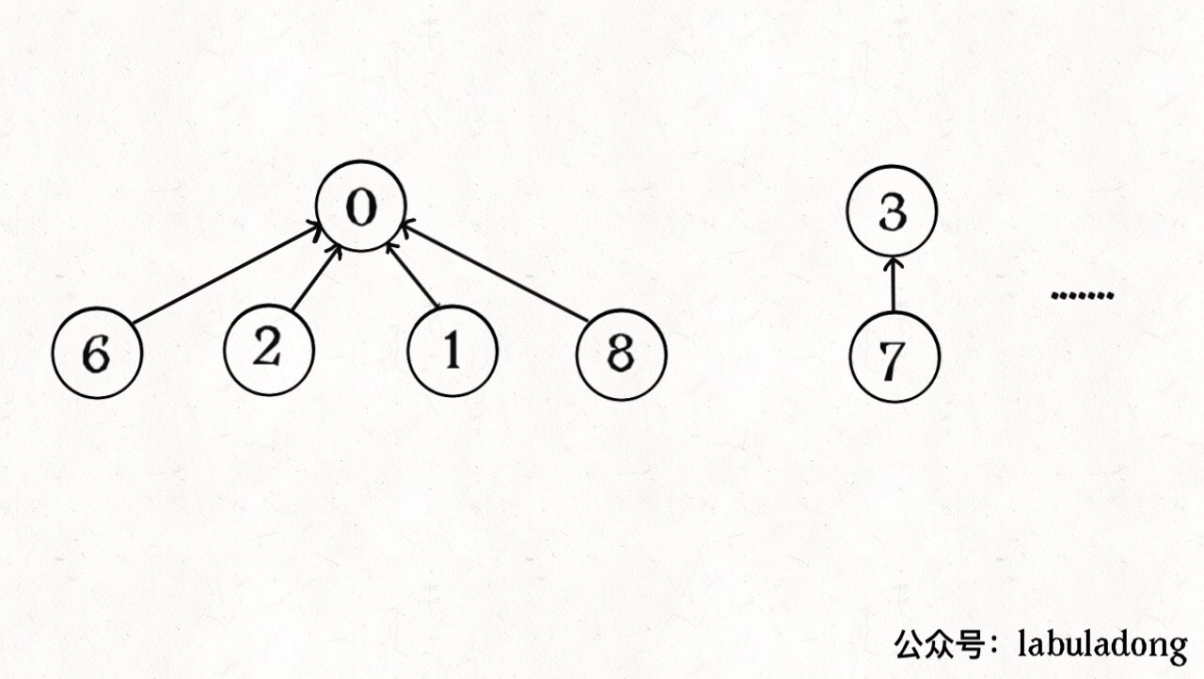
```
public void union(int p, int q) {
    int rootP = find(p);
    int rootQ = find(q);
    if (rootP == rootQ)
        return;

    // 小树接到大树下面，较平衡
    if (size[rootP] > size[rootQ]) {
        parent[rootQ] = rootP;
        size[rootP] += size[rootQ];
    } else {
        parent[rootP] = rootQ;
        size[rootQ] += size[rootP];
    }
    count--;
}
```

这样，通过比较树的重量，就可以保证树的生长相对平衡，树的高度大致在 `logN` 这个数量级，极大提升执行效率。

此时，`find`，`union`，`connected` 的时间复杂度都下降为 O(logN)，即便数据规模上亿，所需时间也非常少。

四、路径压缩

这步优化特别简单，所以非常巧妙。我们能不能进一步压缩每棵树的高度，使树高始终保持为常数？



这样 `find` 就能以 O(1) 的时间找到某一节点的根节点，相应的，`connected` 和 `union` 复杂度都下降为 O(1)。

要做到这一点，非常简单，只需要在 `find` 中加一行代码：

```
private int find(int x) {
    while (parent[x] != x) {
        // 进行路径压缩
        parent[x] = parent[parent[x]];
        x = parent[x];
    }
    return x;
}
```

这个操作有点匪夷所思，看个 GIF 就明白它的作用了（为清晰起见，这棵树比较极端）：

【pdf/mobi格式不支持GIF:unionfind/9.gif】 请查看【关于本小抄及作者】章节的解决方案

可见，调用 `find` 函数每次向树根遍历的同时，顺手将树高缩短了，最终所有树高都不会超过 3（`union` 的时候树高可能达到 3）。

PS：读者可能会问，这个 GIF 图的find过程完成之后，树高恰好等于 3 了，但是如果更高的树，压缩后高度依然会大于 3 呀？不能这么想。这个 GIF 的情景是我编出来方便大家理解路径压缩的，但是实际中，每次find都会进行路径压缩，所以树本来就不可能增长到这么高，你的这种担心应该是多余的。

五、最后总结

我们先来看一下完整代码：

```
class UF {
    // 连通分量个数
    private int count;
    // 存储一棵树
    private int[] parent;
    // 记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        } else {
            parent[rootP] = rootQ;
```

```
            size[rootQ] += size[rootP];
        }
        count--;
    }

    public boolean connected(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        return rootP == rootQ;
    }

    private int find(int x) {
        while (parent[x] != x) {
            // 进行路径压缩
            parent[x] = parent[parent[x]];
            x = parent[x];
        }
        return x;
    }

    public int count() {
        return count;
    }
}
```

Union-Find 算法的复杂度可以这样分析：构造函数初始化数据结构需要 O(N) 的时间和空间复杂度；连通两个节点 `union`、判断两个节点的连通性 `connected`、计算连通分量 `count` 所需的时间复杂度均为 O(1)。

Union-Find算法应用

上篇文章很多读者对于 Union-Find 算法的应用表示很感兴趣，这篇文章就拿几道 LeetCode 题目来讲讲这个算法的巧妙用法。

首先，复习一下，Union-Find 算法解决的是图的动态连通性问题，这个算法本身不难，能不能应用出来主要是看你抽象问题的能力，是否能够把原始问题抽象成一个有关图论的问题。

先复习一下上篇文章写的算法代码，回答读者提出的几个问题：

```
class UF {
    // 记录连通分量个数
    private int count;
    // 存储若干棵树
    private int[] parent;
    // 记录树的“重量”
    private int[] size;

    public UF(int n) {
        this.count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

    /* 将 p 和 q 连通 */
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ)
            return;

        // 小树接到大树下面，较平衡
        if (size[rootP] > size[rootQ]) {
```

```
                parent[rootQ] = rootP;
                size[rootP] += size[rootQ];
            } else {
                parent[rootP] = rootQ;
                size[rootQ] += size[rootP];
            }
            count--;
        }

        /* 判断 p 和 q 是否互相连通 */
        public boolean connected(int p, int q) {
            int rootP = find(p);
            int rootQ = find(q);
            // 处于同一棵树上的节点，相互连通
            return rootP == rootQ;
        }

        /* 返回节点 x 的根节点 */
        private int find(int x) {
            while (parent[x] != x) {
                // 进行路径压缩
                parent[x] = parent[parent[x]];
                x = parent[x];
            }
            return x;
        }

        public int count() {
            return count;
        }
    }
```

算法的关键点有 3 个：

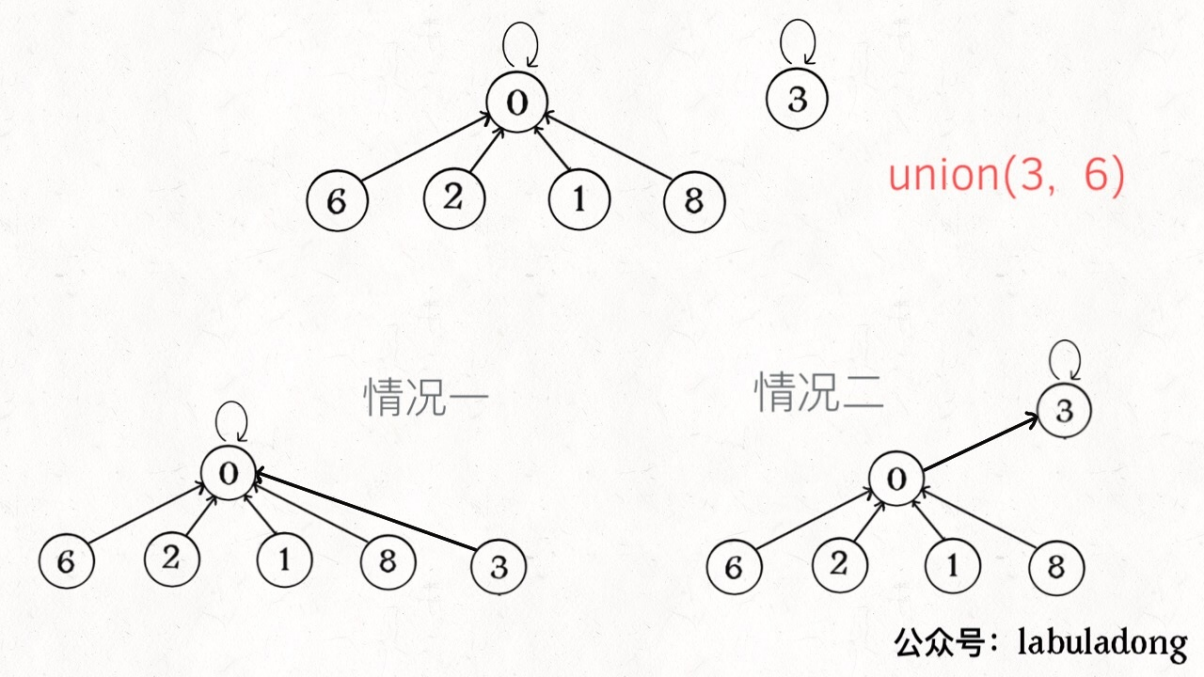
1、用 `parent` 数组记录每个节点的父节点，相当于指向父节点的指针，所以 `parent` 数组内实际存储着一个森林（若干棵多叉树）。

2、用 `size` 数组记录着每棵树的重量，目的是让 `union` 后树依然拥有平衡性，而不会退化成链表，影响操作效率。

3、在 `find` 函数中进行路径压缩，保证任意树的高度保持在常数，使得 `union` 和 `connected` API 时间复杂度为 O(1)。

有的读者问，**既然有了路径压缩，`size` 数组的重量平衡还需要吗**？这个问题很有意思，因为路径压缩保证了树高为常数（不超过 3），那么树就算不平衡，高度也是常数，基本没什么影响。

我认为，论时间复杂度的话，确实，不需要重量平衡也是 O(1)。但是如果加上 `size` 数组辅助，效率还是略微高一些，比如下面这种情况：



如果带有重量平衡优化，一定会得到情况一，而不带重量优化，可能出现情况二。高度为 3 时才会触发路径压缩那个 `while` 循环，所以情况一根本不会触发路径压缩，而情况二会多执行很多次路径压缩，将第三层节点压缩到第二层。

也就是说，去掉重量平衡，虽然对于单个的 `find` 函数调用，时间复杂度依然是 O(1)，但是对于 API 调用的整个过程，效率会有一定的下降。当然，好处就是减少了一些空间，不过对于 Big O 表示法来说，时空复杂度都没变。

下面言归正传，来看看这个算法有什么实际应用。

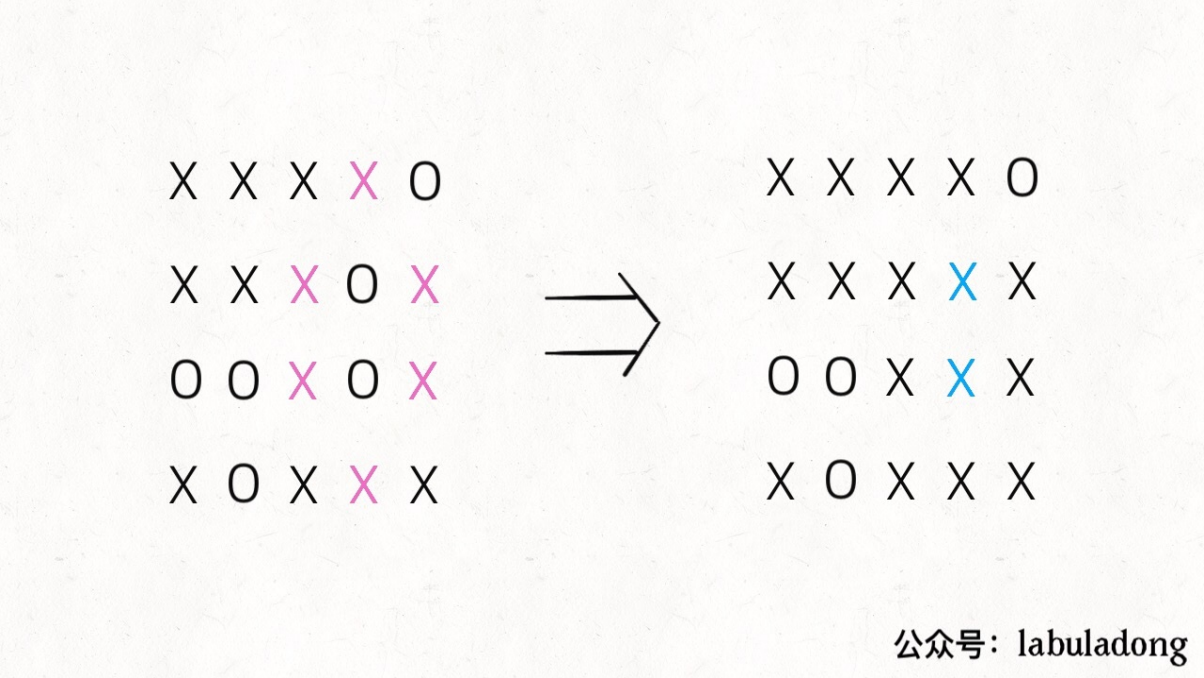
一、DFS 的替代方案

很多使用 DFS 深度优先算法解决的问题，也可以用 Union-Find 算法解决。

比如第 130 题，被围绕的区域：给你一个 M×N 的二维矩阵，其中包含字符 `X` 和 `O`，让你找到矩阵中**四面**被 `X` 围住的 `O`，并且把它们替换成 `X`。

```
void solve(char[][] board);
```

注意哦，必须是四面被围的 `O` 才能被换成 `X`，也就是说边角上的 `O` 一定不会被围，进一步，与边角上的 `O` 相连的 `O` 也不会被 `X` 围四面，也不会被替换。



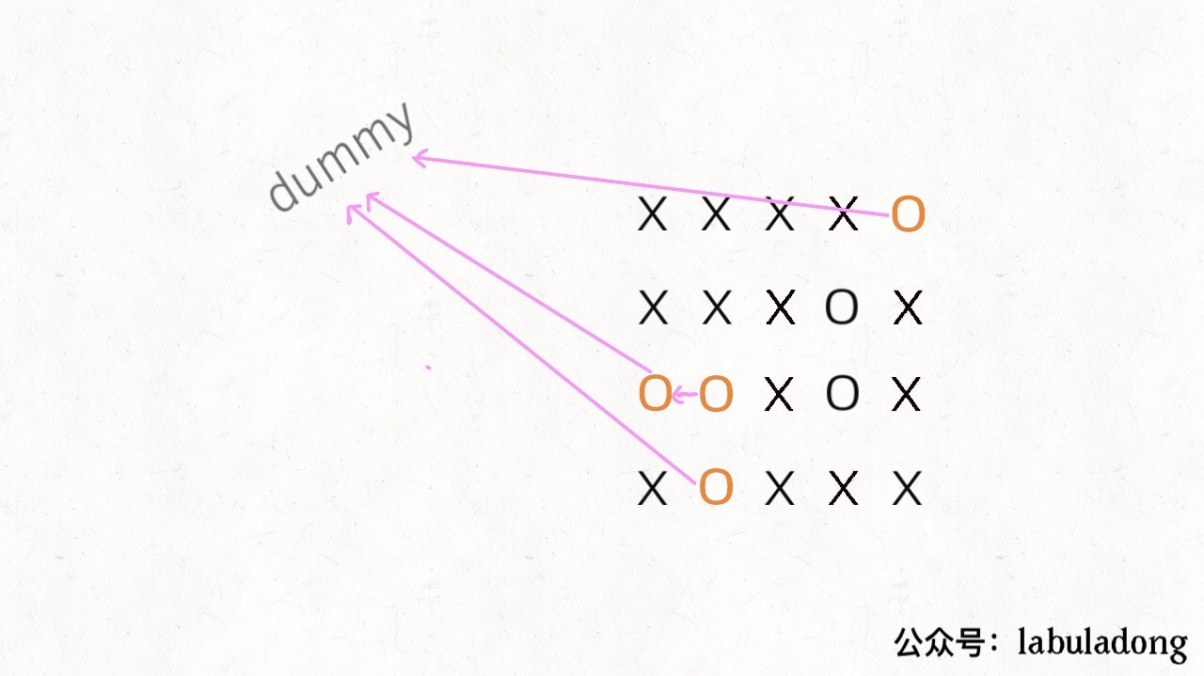
PS：这让我想起小时候玩的棋类游戏「黑白棋」，只要你用两个棋子把对方的棋子夹在中间，对方的子就被替换成你的子。可见，占据四角的棋子是无敌的，与其相连的边棋子也是无敌的（无法被夹掉）。

解决这个问题的传统方法也不困难，先用 for 循环遍历棋盘的**四边**，用 DFS 算法把那些与边界相连的 `O` 换成一个特殊字符，比如 `#`；然后再遍历整个棋盘，把剩下的 `O` 换成 `X`，把 `#` 恢复成 `O`。这样就能完成题目的要

求，时间复杂度 O(MN)。

这个问题也可以用 Union-Find 算法解决，虽然实现复杂一些，甚至效率也略低，但这是使用 Union-Find 算法的通用思想，值得一学。

你可以把那些不需要被替换的 `O` 看成一个拥有独门绝技的门派，它们有一个共同祖师爷叫 `dummy`，这些 `O` 和 `dummy` 互相连通，而那些需要被替换的 `O` 与 `dummy` 不连通。



这就是 Union-Find 的核心思路，明白这个图，就很容易看懂代码了。

首先要解决的是，根据我们的实现，Union-Find 底层用的是一维数组，构造函数需要传入这个数组的大小，而题目给的是一个二维棋盘。

这个很简单，二维坐标 `(x,y)` 可以转换成 `x * n + y` 这个数（`m` 是棋盘的行数，`n` 是棋盘的列数）。敲黑板，**这是将二维坐标映射到一维的常用技巧**。

其次，我们之前描述的「祖师爷」是虚构的，需要给他老人家留个位置。索引 `[0.. m*n-1]` 都是棋盘内坐标的一维映射，那就让这个虚拟的 `dummy` 节点占据索引 `m * n` 好了。

```
void solve(char[][] board) {
```

```
        if (board.length == 0) return;

        int m = board.length;
        int n = board[0].length;
        // 给 dummy 留一个额外位置
        UF uf = new UF(m * n + 1);
        int dummy = m * n;
        // 将首列和末列的 O 与 dummy 连通
        for (int i = 0; i < m; i++) {
            if (board[i][0] == 'O')
                uf.union(i * n, dummy);
            if (board[i][n - 1] == 'O')
                uf.union(i * n + n - 1, dummy);
        }
        // 将首行和末行的 O 与 dummy 连通
        for (int j = 0; j < n; j++) {
            if (board[0][j] == 'O')
                uf.union(j, dummy);
            if (board[m - 1][j] == 'O')
                uf.union(n * (m - 1) + j, dummy);
        }
        // 方向数组 d 是上下左右搜索的常用手法
        int[][] d = new int[][]{{1,0}, {0,1}, {0,-1}, {-1,0}};
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
                if (board[i][j] == 'O')
                    // 将此 O 与上下左右的 O 连通
                    for (int k = 0; k < 4; k++) {
                        int x = i + d[k][0];
                        int y = j + d[k][1];
                        if (board[x][y] == 'O')
                            uf.union(x * n + y, i * n + j);
                    }
        // 所有不和 dummy 连通的 O，都要被替换
        for (int i = 1; i < m - 1; i++)
            for (int j = 1; j < n - 1; j++)
                if (!uf.connected(dummy, i * n + j))
                    board[i][j] = 'X';
    }
```

这段代码很长，其实就是刚才的思路实现，只有和边界 `O` 相连的 `O` 才具有和 `dummy` 的连通性，他们不会被替换。

说实话，Union-Find 算法解决这个简单的问题有点杀鸡用牛刀，它可以解决更复杂，更具有技巧性的问题，**主要思路是适时增加虚拟节点，想办法让元素「分门别类」，建立动态连通关系**。

二、判定合法等式

这个问题用 Union-Find 算法就显得十分优美了。题目是这样：

给你一个数组 `equations`，装着若干字符串表示的算式。每个算式 `equations[i]` 长度都是 4，而且只有这两种情况：`a==b` 或者 `a!=b`，其中 `a,b` 可以是任意小写字母。你写一个算法，如果 `equations` 中所有算式都不会互相冲突，返回 true，否则返回 false。

比如说，输入 `["a==b","b!=c","c==a"]`，算法返回 false，因为这三个算式不可能同时正确。

再比如，输入 `["c==c","b==d","x!=z"]`，算法返回 true，因为这三个算式并不会造成逻辑冲突。

我们前文说过，动态连通性其实就是一种等价关系，具有「自反性」「传递性」和「对称性」，其实 `==` 关系也是一种等价关系，具有这些性质。所以这个问题用 Union-Find 算法就很自然。

核心思想是，**将 `equations` 中的算式根据 `==` 和 `!=` 分成两部分，先处理 `==` 算式，使得他们通过相等关系各自勾结成门派；然后处理 `!=` 算式，检查不等关系是否破坏了相等关系的连通性**。

```
boolean equationsPossible(String[] equations) {
    // 26 个英文字母
    UF uf = new UF(26);
    // 先让相等的字母形成连通分量
    for (String eq : equations) {
        if (eq.charAt(1) == '=') {
            char x = eq.charAt(0);
            char y = eq.charAt(3);
            uf.union(x - 'a', y - 'a');
        }
```

```
        }
        // 检查不等关系是否打破相等关系的连通性
        for (String eq : equations) {
            if (eq.charAt(1) == '!') {
                char x = eq.charAt(0);
                char y = eq.charAt(3);
                // 如果相等关系成立，就是逻辑冲突
                if (uf.connected(x - 'a', y - 'a'))
                    return false;
            }
        }
        return true;
    }
```

至此，这道判断算式合法性的问题就解决了，借助 Union-Find 算法，是不是很简单呢？

三、简单总结

使用 Union-Find 算法，主要是如何把原问题转化成图的动态连通性问题。对于算式合法性问题，可以直接利用等价关系，对于棋盘包围问题，则是利用一个虚拟节点，营造出动态连通特性。

另外，将二维数组映射到一维数组，利用方向数组 `d` 来简化代码量，都是在写算法时常用的一些小技巧，如果没见过可以注意一下。

很多更复杂的 DFS 算法问题，都可以利用 Union-Find 算法更漂亮的解决。LeetCode 上 Union-Find 相关的问题也就二十多道，有兴趣的读者可以去做一做。

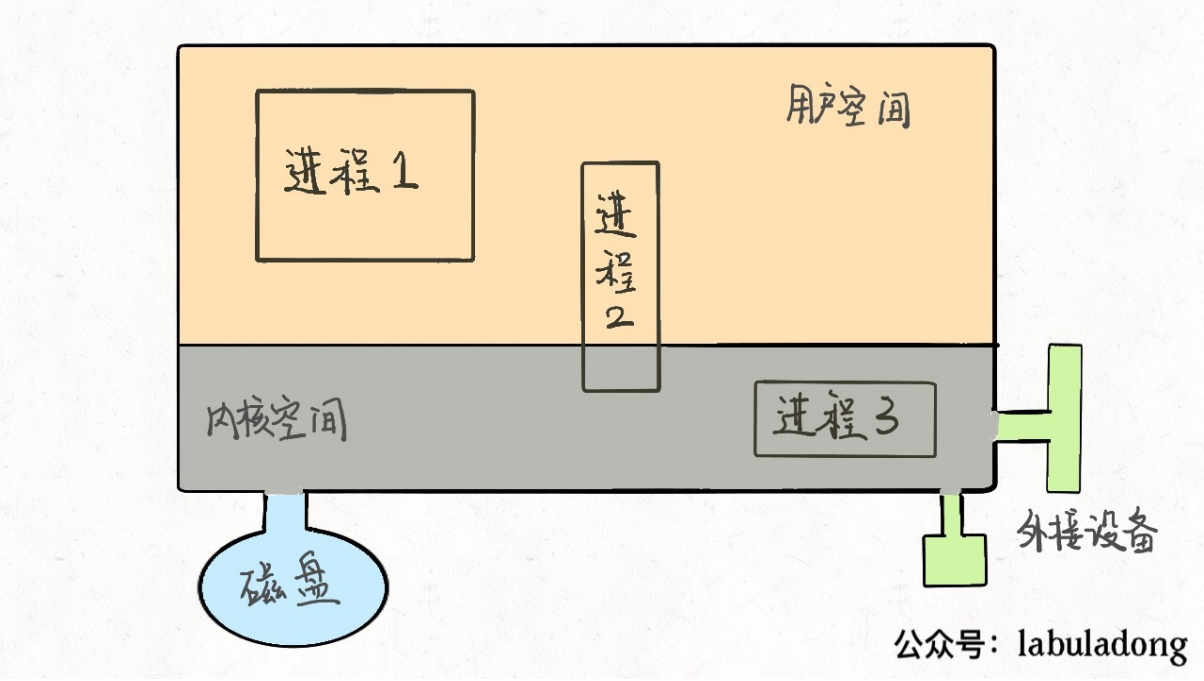
Linux的进程、线程、文件描述符是什么

说到进程，恐怕面试中最常见的问题就是线程和进程的关系了，那么先说一下答案：**在 Linux 系统中，进程和线程几乎没有区别**。

Linux 中的进程就是一个数据结构，看明白就可以理解文件描述符、重定向、管道命令的底层工作原理，最后我们从操作系统的角度看看为什么说线程和进程基本没有区别。

一、进程是什么

首先，抽象地来说，我们的计算机就是这个东西：



这个大的矩形表示计算机的**内存空间**，其中的小矩形代表**进程**，左下角的圆形表示**磁盘**，右下角的图形表示一些**输入输出设备**，比如鼠标键盘显示器等等。另外，注意到内存空间被划分为了两块，上半部分表示**用户空间**，下半部分表示**内核空间**。

用户空间装着用户进程需要使用的资源，比如你在程序代码里开一个数组，这个数组肯定存在用户空间；内核空间存放内核进程需要加载的系统资源，这一些资源一般是不允许用户访问的。但是注意有的用户进程会共享一些内核空间的资源，比如一些动态链接库等等。

我们用 C 语言写一个 hello 程序，编译后得到一个可执行文件，在命令行运行就可以打印出一句 hello world，然后程序退出。在操作系统层面，就是新建了一个进程，这个进程将我们编译出来的可执行文件读入内存空间，然后执行，最后退出。

你编译好的那个可执行程序只是一个文件，不是进程，可执行文件必须要载入内存，包装成一个进程才能真正跑起来。进程是要依靠操作系统创建的，每个进程都有它的固有属性，比如进程号（PID）、进程状态、打开的文件等等，进程创建好之后，读入你的程序，你的程序才被系统执行。

那么，操作系统是如何创建进程的呢？**对于操作系统，进程就是一个数据结构**，我们直接来看 Linux 的源码：

```
struct task_struct {
    // 进程状态
    long              state;
    // 虚拟内存结构体
    struct mm_struct  *mm;
    // 进程号
    pid_t              pid;
    // 指向父进程的指针
    struct task_struct __rcu  *parent;
    // 子进程列表
    struct list_head        children;
    // 存放文件系统信息的指针
    struct fs_struct        *fs;
    // 一个数组，包含该进程打开的文件指针
    struct files_struct        *files;
};
```

`task_struct` 就是 Linux 内核对于一个进程的描述，也可以称为「进程描述符」。源码比较复杂，我这里就截取了一小部分比较常见的。

其中比较有意思的是 `mm` 指针和 `files` 指针。 `mm` 指向的是进程的虚拟内存，也就是载入资源和可执行文件的地方； `files` 指针指向一个数组，这个数组里装着所有该进程打开的文件的指针。

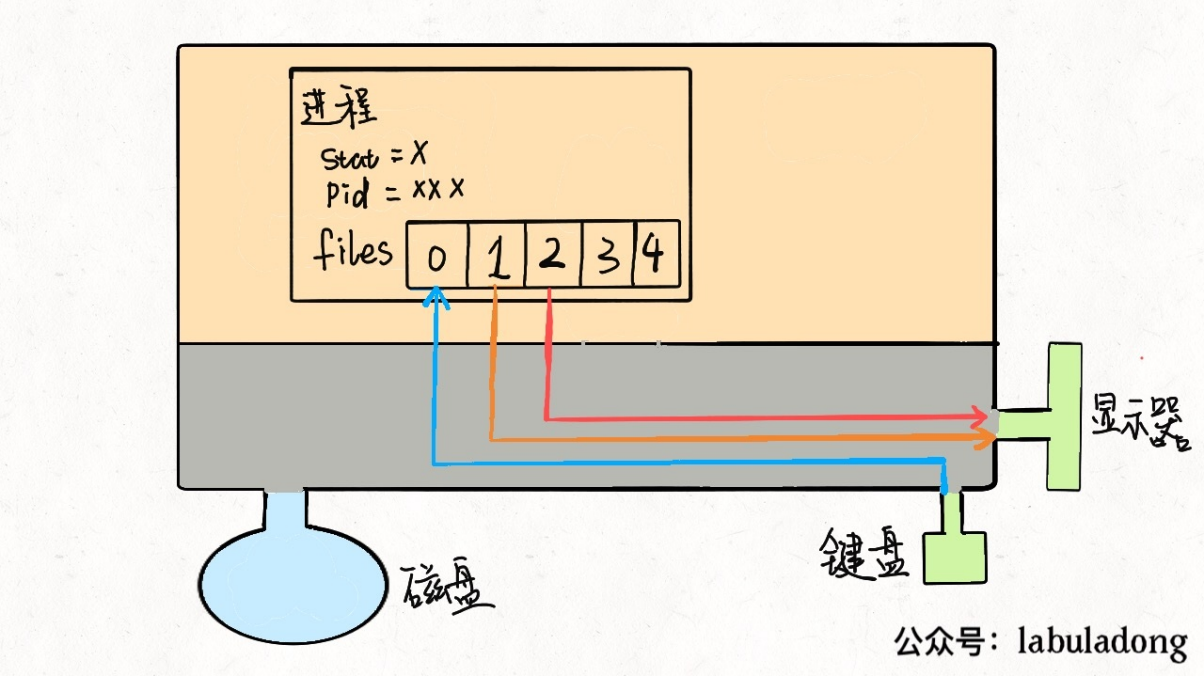
二、文件描述符是什么

先说 `files` ，它是一个文件指针数组。一般来说，一个进程会从 `files[0]` 读取输入，将输出写入 `files[1]` ，将错误信息写入 `files[2]` 。

举个例子，以我们的角度 C 语言的 `printf` 函数是向命令行打印字符，但是从进程的角度来看，就是向 `files[1]` 写入数据；同理， `scanf` 函数就是进程试图从 `files[0]` 这个文件中读取数据。

每个进程被创建时， `files` 的前三位被填入默认值，分别指向标准输入流、标准输出流、标准错误流。我们常说的「文件描述符」就是指这个文件指针数组的索引，所以程序的文件描述符默认情况下 0 是输入，1 是输出，2 是错误。

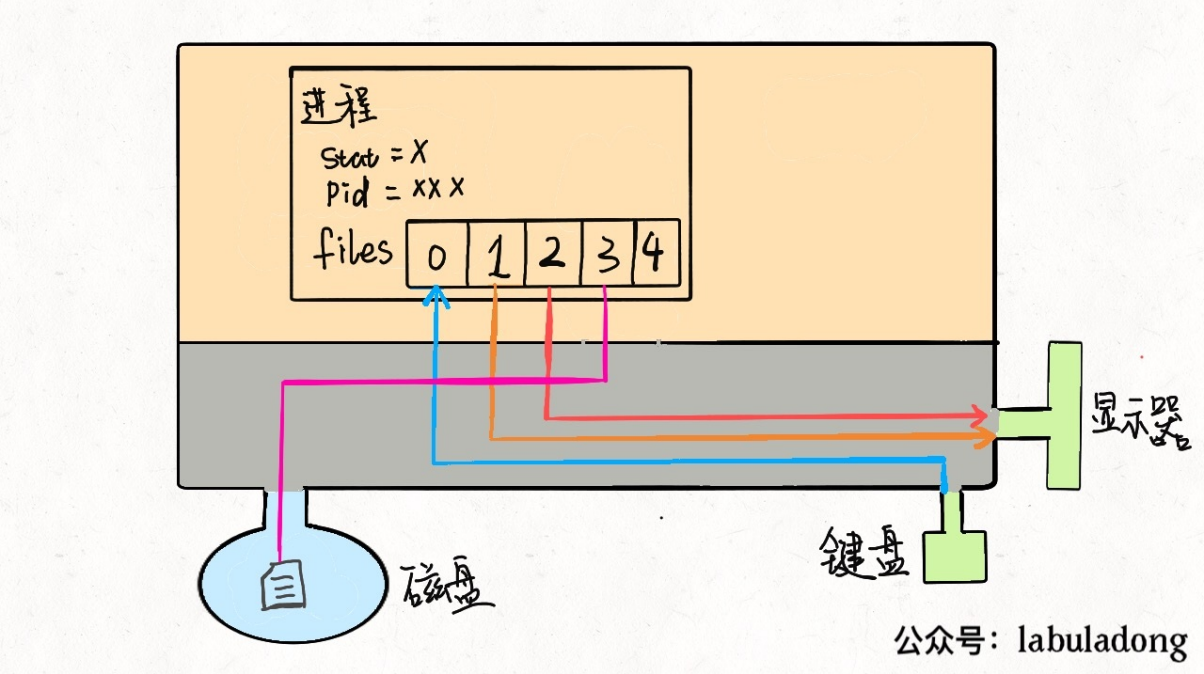
我们可以重新画一幅图：



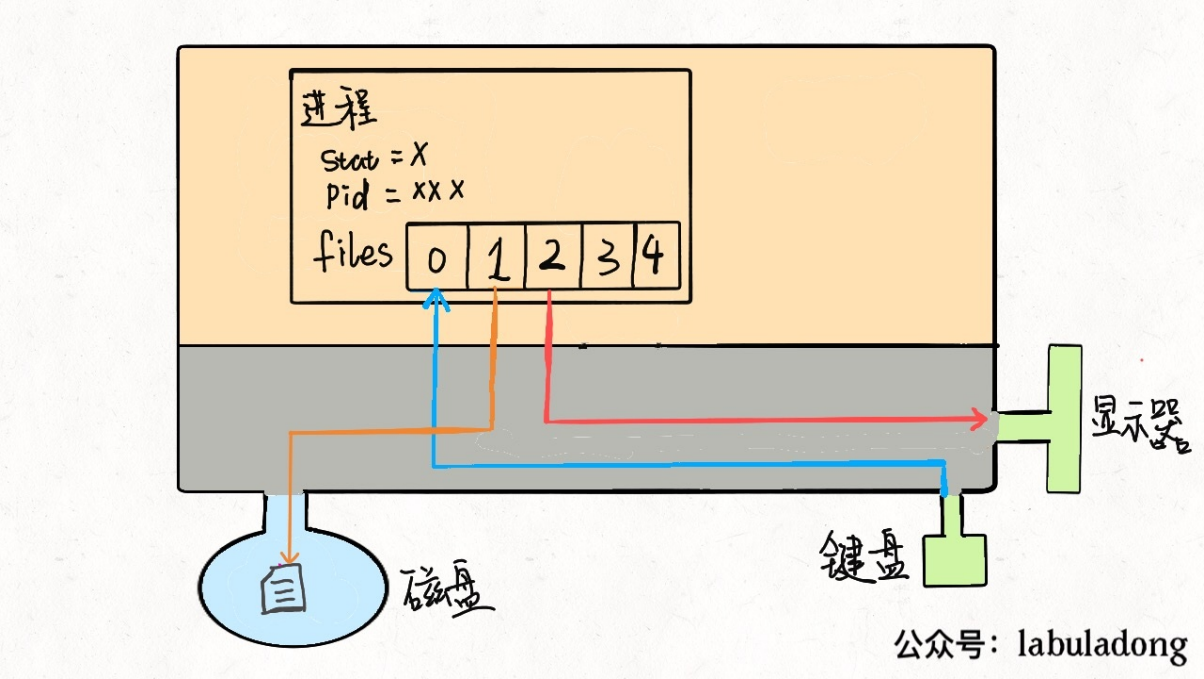
对于一般的计算机，输入流是键盘，输出流是显示器，错误流也是显示器，所以现在这个进程和内核连了三根线。因为硬件都是由内核管理的，我们的进程需要通过「系统调用」让内核进程访问硬件资源。

PS：不要忘了，Linux 中一切都被抽象成文件，设备也是文件，可以进行读和写。

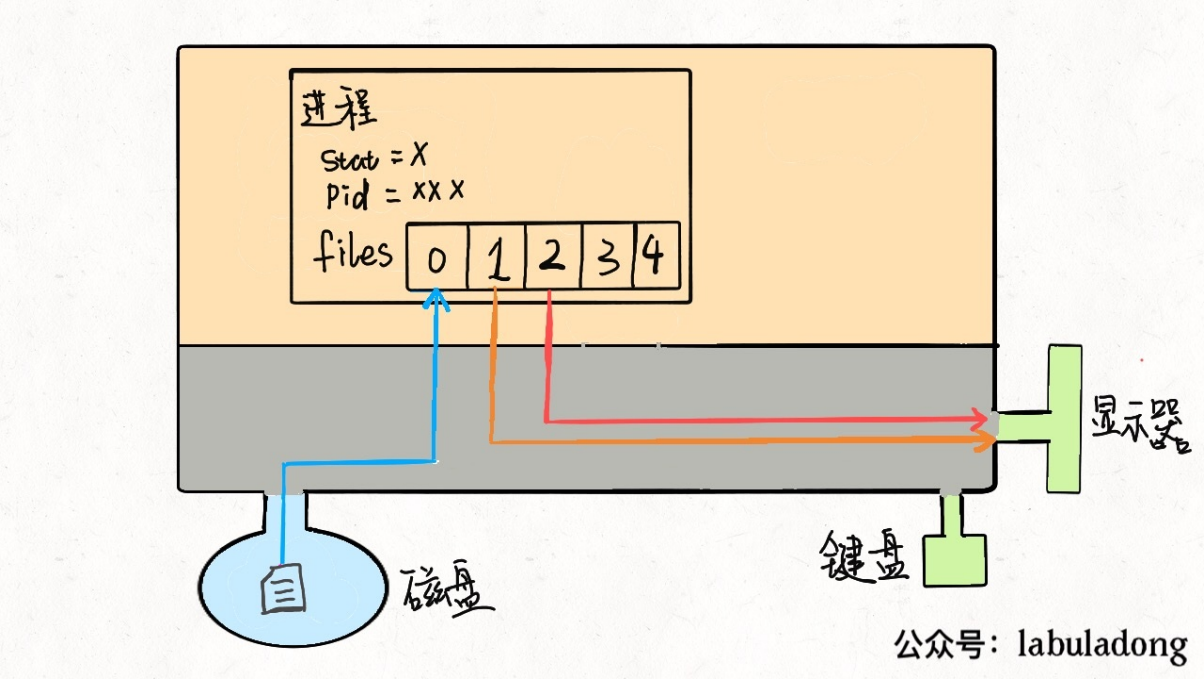
如果我们写的程序需要其他资源，比如打开一个文件进行读写，这也很简单，进行系统调用，让内核把文件打开，这个文件就会被放到 `files` 的第 4 个位置：



明白了这个原理，**输入重定向**就很好理解了，程序想读取数据的时候就会去 `files[0]` 读取，所以我们只要把 `files[0]` 指向一个文件，那么程序就会从这个文件中读取数据，而不是从键盘：

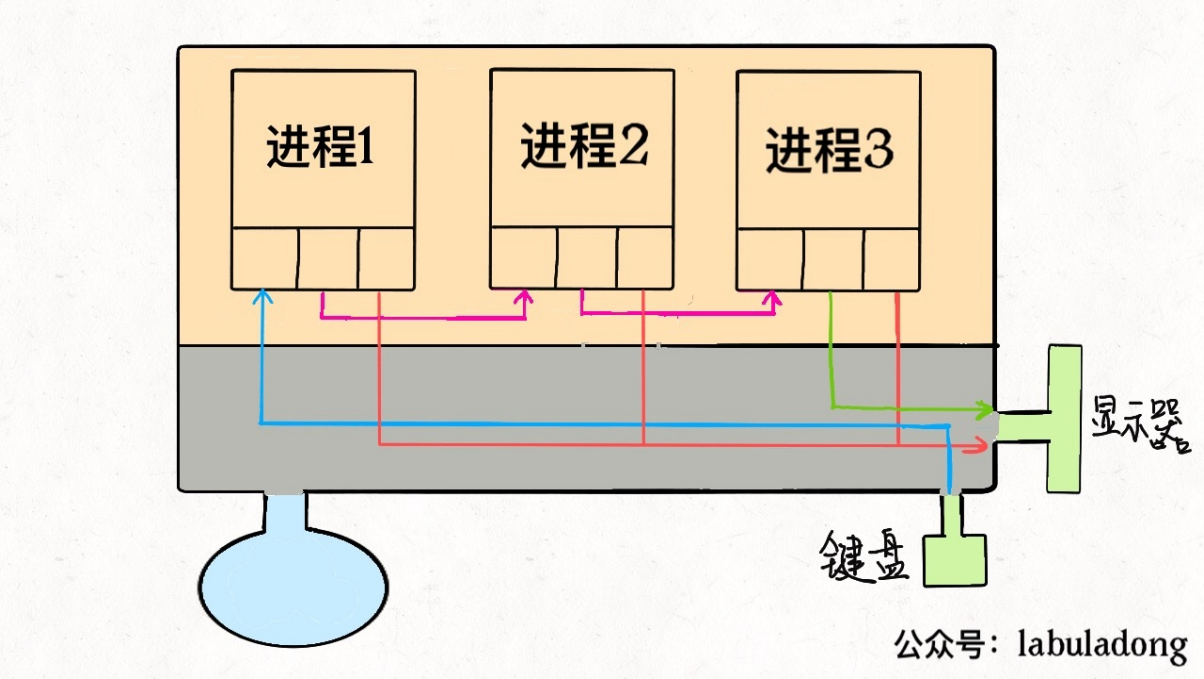


同理，**输出重定向**就是把 `files[1]` 指向一个文件，那么程序的输出就不会写入到显示器，而是写入到这个文件中：



错误重定向也是一样的，就不再赘述。

管道符其实也是异曲同工，把一个进程的输出流和另一个进程的输入流接起一条「管道」，数据就在其中传递，不得不说这种设计思想真的很优美：



到这里，你可能也看出「Linux 中一切皆文件」设计思路的高明了，不管是设备、另一个进程、socket 套接字还是真正的文件，全部都可以读写，统一装进一个简单的 `files` 数组，进程通过简单的文件描述符访问相应资源，具体细节交于操作系统，有效解耦，优美高效。

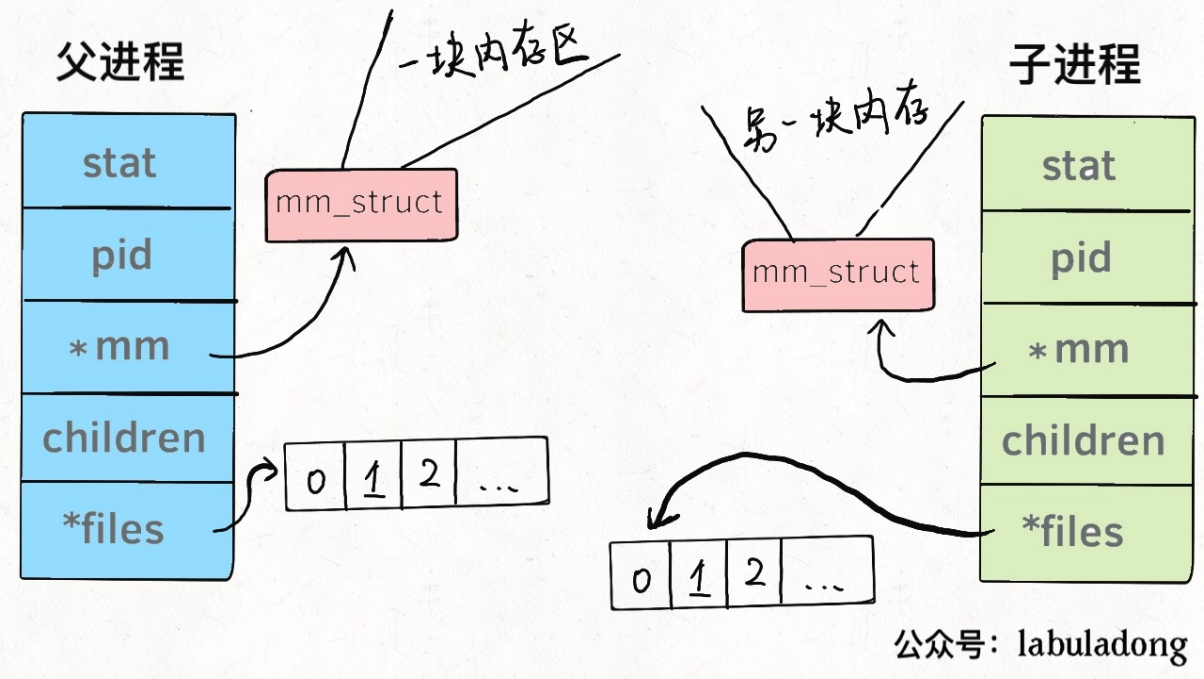
三、线程是什么

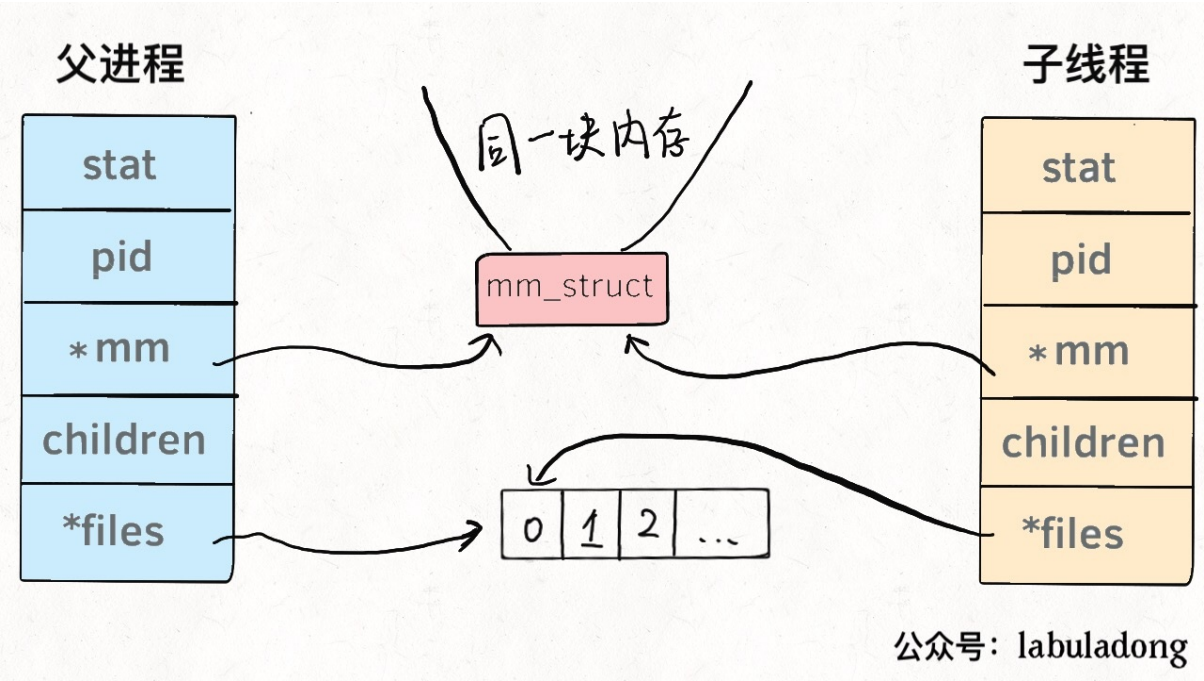
首先要明确的是，多进程和多线程都是并发，都可以提高处理器的利用效率，所以现在的关键是，多线程和多进程有啥区别。

为什么说 Linux 中线程和进程基本没有区别呢，因为从 Linux 内核的角度来看，并没有把线程和进程区别对待。

我们知道系统调用 `fork()` 可以新建一个子进程，函数 `pthread()` 可以新建一个线程。**但无论线程还是进程，都是用 `task_struct` 结构表示的，唯一的区别就是共享的数据区域不同**。

换句话说，线程看起来跟进程没有区别，只是线程的某些数据区域和其父进程是共享的，而子进程是拷贝副本，而不是共享。就比如说，`mm` 结构和 `files` 结构在线程中都是共享的，我画两张图你就明白了：





所以说，我们的多线程程序要利用锁机制，避免多个线程同时往同一区域写入数据，否则可能造成数据错乱。

那么你可能问，**既然进程和线程差不多，而且多进程数据不共享，即不存在数据错乱的问题，为什么多线程的使用比多进程普遍得多呢**？

因为现实中数据共享的并发更普遍呀，比如十个人同时从一个账户取十元，我们希望的是这个共享账户的余额正确减少一百元，而不是希望每人获得一个账户的拷贝，每个拷贝账户减少十元。

当然，必须要说明的是，只有 Linux 系统将线程看做共享数据的进程，不对其做特殊看待，其他的很多操作系统是对线程和进程区别对待的，线程有其特有的数据结构，我个人认为不如 Linux 的这种设计简洁，增加了系统的复杂度。

在 Linux 中新建线程和进程的效率都是很高的，对于新建进程时内存区域拷贝的问题，Linux 采用了 copy-on-write 的策略优化，也就是并不真正复制父进程的内存空间，而是等到需要写操作时才去复制。**所以 Linux 中新建进程和新建线程都是很迅速的**。

一行代码就能解决的算法题

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

一、Nim 游戏

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

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

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

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

我们解决这种问题的思路一般都是反着思考：

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

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

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

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

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

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

二、石头游戏

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

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

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

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

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

`piles=[1, 9]` 轮到你拿，你拿 9。

最后，你的对手只能拿 1 了。

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

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

这也是强调双方都很聪明的原因，算法也是求最优决策过程下你是否能赢。

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

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

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

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

而作为第一个拿石头的人，你可以控制自己拿到所有偶数堆，或者所有的奇数堆。

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

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

三、电灯开关问题

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

第 1 轮操作是把每一盏电灯的开关按一下（全部打开）。

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

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

如此往复，直到第 n 轮，即只按一下第 n 盏灯的开关。

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

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

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

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

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

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

为什么第 1、2、3、6 轮会被按呢？因为 $6=1\times6=2\times3$。一般情况下，因子都是成对出现的，也就是说开关被按的次数一般是偶数次。但是有特殊情况，比如说总共有 16 盏灯，那么第 16 盏灯会被按几次?

$16=1\times16=2\times8=4\times4$

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

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

就假设现在总共有 16 盏灯，我们求 16 的平方根，等于 4，这就说明最后会有 4 盏灯亮着，它们分别是第 $1\times1=1$ 盏、第 $2\times2=4$ 盏、第 $3\times3=9$ 盏和第 $4\times4=16$ 盏。

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

说到密码，我们第一个想到的就是登陆账户的密码，但是从密码学的角度来看，这种根本就不算合格的密码。

为什么呢，因为我们的账户密码，是依靠隐蔽性来达到加密作用：密码藏在我心里，你不知道，所以你登不上我的账户。

然而密码技术认为，「保密」信息总有一天会被扒出来，所以加密算法不应该依靠「保密」来保证机密性，而应该做到：即便知道了加密算法，依然无计可施。说的魔幻一点就是，告诉你我的密码，你依然不知道我的密码。

最玄学的就是 Diffie-Hellman 密钥交换算法，我当初就觉得很惊奇，两个人当着你的面互相报几个数字，他们就可以拥有一个共同的秘密，而你却根本不可能算出来这个秘密。下文会着重介绍一下这个算法。

本文讨论的密码技术要解决的主要是信息传输中的加密和解密问题。要假设数据传输过程是不安全的，所有信息都在被窃听的，所以发送端要把信息加密，接收方收到信息之后，肯定得知道如何解密。有意思的是，如果你能够让接收者知道如何解密，那么窃听者不是也能够知道如何解密了吗？

下面，**我们会介绍对称加密算法、密钥交换算法、非对称加密算法、数字签名、公钥证书**，看看解决安全传输问题的一路坎坷波折。

一、对称性加密

对称性密码，也叫共享密钥密码，顾名思义，这种加密方式用相同的密钥进行加密和解密。

比如我说一种最简单的对称加密的方法。首先我们知道信息都可以表示成 0/1 比特序列，也知道相同的两个比特序列做异或运算的结果为 0。

那么我们就可以生成一个长度和原始信息一样的随机比特序列作为密钥，然后用它对原始信息做异或运算，就生成了密文。反之，再用该密钥对密文做一次异或运算，就可以恢复原始信息。

这是一个简单例子，不过有些过于简单，有很多问题。比如密钥的长度和原始信息完全一致，如果原始信息很大，密钥也会一样大，而且生成大量真随机比特序列的计算开销也比较大。

当然，有很多更复杂优秀的对称加密算法解决了这些问题，比如 Rijndael 算法、三重 DES 算法等等。**它们从算法上是无懈可击的，也就是拥有巨大的密钥空间，基本无法暴力破解，而且加密过程相对快速**。

但是，一切对称加密算法的软肋在于密钥的配送。加密和解密用同一个密钥，发送方必须设法把密钥发送给接收方。如果窃听者有能力窃取密文，肯定也可以窃取密钥，那么再无懈可击的算法依然不攻自破。

所以，下面介绍两种解决密钥配送问题最常见的算法，分别是 Diffie-Hellman 密钥交换算法和非对称加密算法。

二、密钥交换算法

我们所说的密钥一般就是一个很大的数字，算法用这个数加密、解密。问题在于，信道是不安全的，所有发出的数据都会被窃取。换句话说，有没有一种办法，能够让两个人在众目睽睽之下，光明正大地交换一个秘密，把对称性密钥安全地送到接收方的手中？

Diffie-Hellman 密钥交换算法可以做到。**准确的说，该算法并不是把一个秘密安全地「送给」对方，而是通过一些共享的数字，双方「心中」各自「生成」了一个相同的秘密，而且双方的这个秘密，是第三方窃听者无法生成的**。

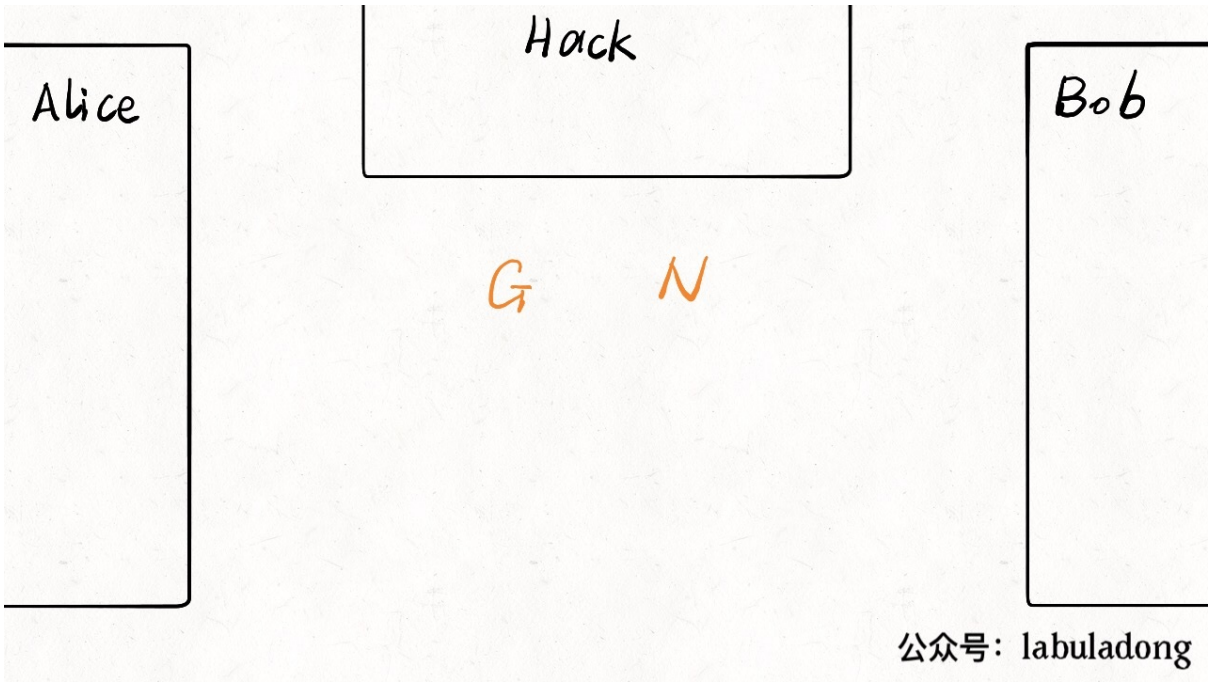
也许这就是传说中的心有灵犀一点通吧。

这个算法规则不算复杂，你甚至都可以找个朋友尝试一下共享秘密，等会我会简单画出它的基本流程。在此之前，需要明确一个问题：**并不是所有运算都有逆运算**。

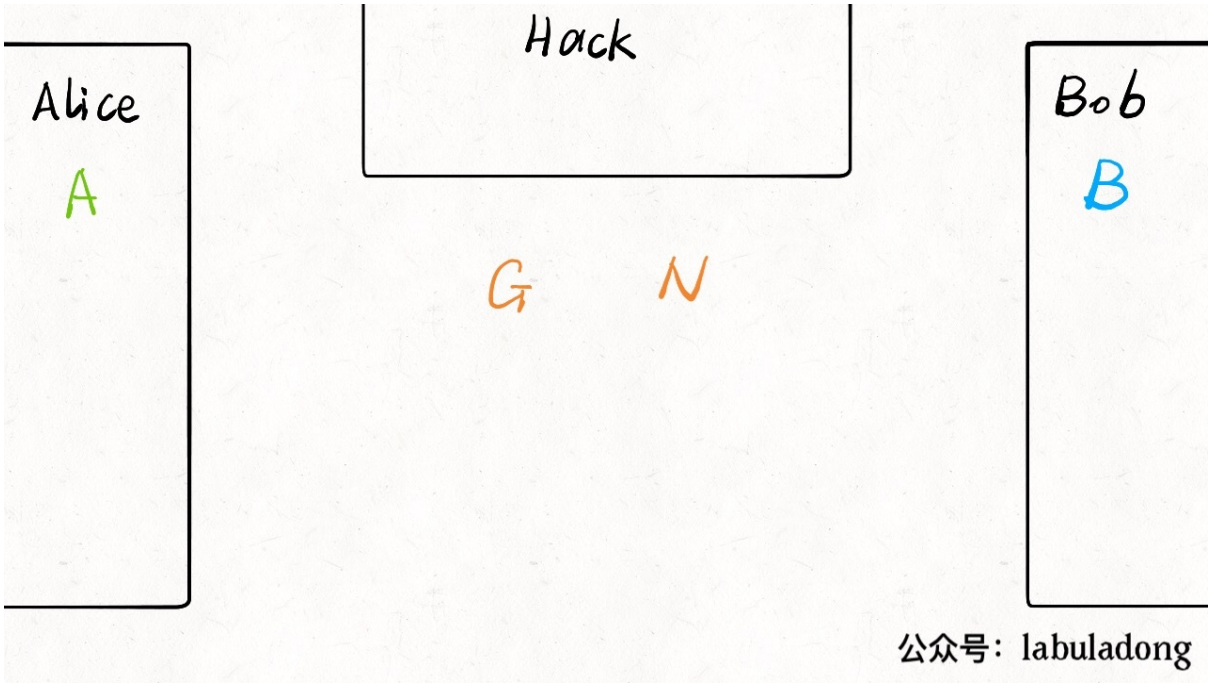
最简单的例子就是我们熟知的单向散列函数，给一个数字 `a` 和一个散列函数 `f`，你可以很快计算出 `f(a)`，但是如果给你 `f(a)` 和 `f`，推出 `a` 是一件基本做不到的事。密钥交换算法之所以看起来如此玄幻，就是利用了这种不可逆的性质。

下面，看下密钥交换算法的流程是什么，按照命名惯例，准备执行密钥交换算法的双方称为 Alice 和 Bob，在网络中企图窃取他俩通信内容的坏人称为 Hack 吧。

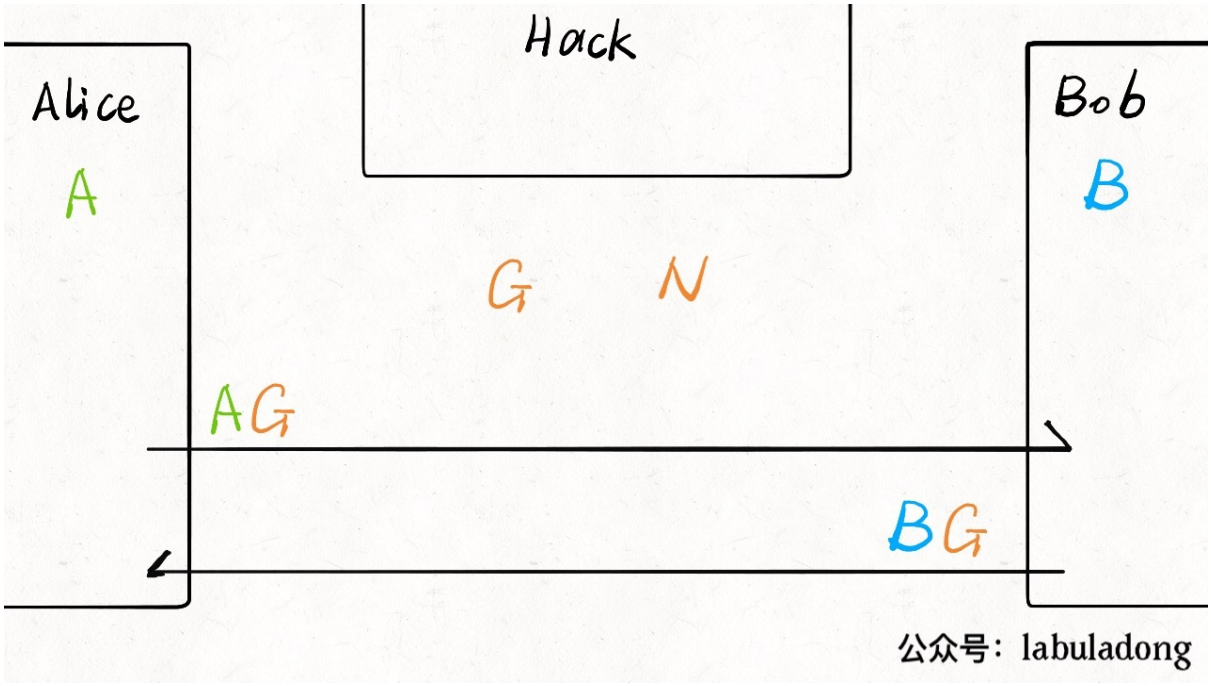
首先，Alice 和 Bob 协商出两个数字 `N` 和 `G` 作为生成元，当然协商过程可以被窃听者 Hack 窃取，所以我把这两个数画到中间，代表三方都知道：



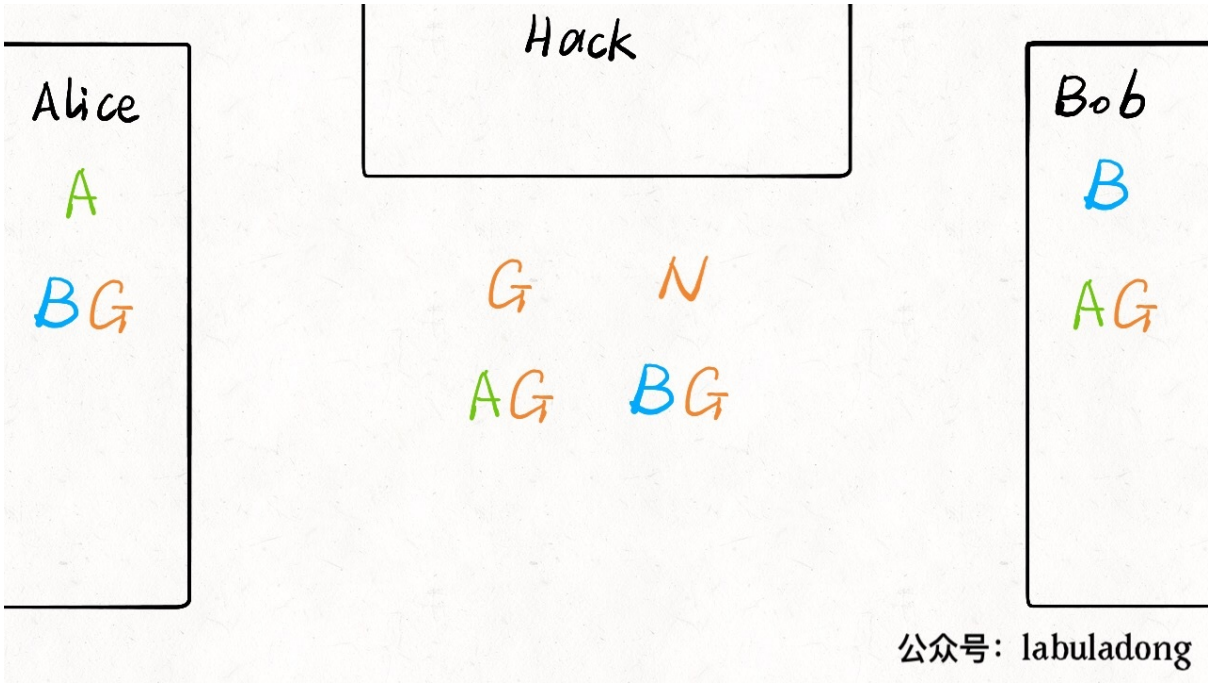
现在 Alice 和 Bob **心中**各自想一个数字出来，分别称为 `A` 和 `B` 吧：



现在 Alice 将自己心里的这个数字 `A` 和 `G` 通过某些运算得出一个数 `AG`，然后发给 Bob；Bob 将自己心里的数 `B` 和 `G` 通过相同的运算得出一个数 `BG`，然后发给 Alice：



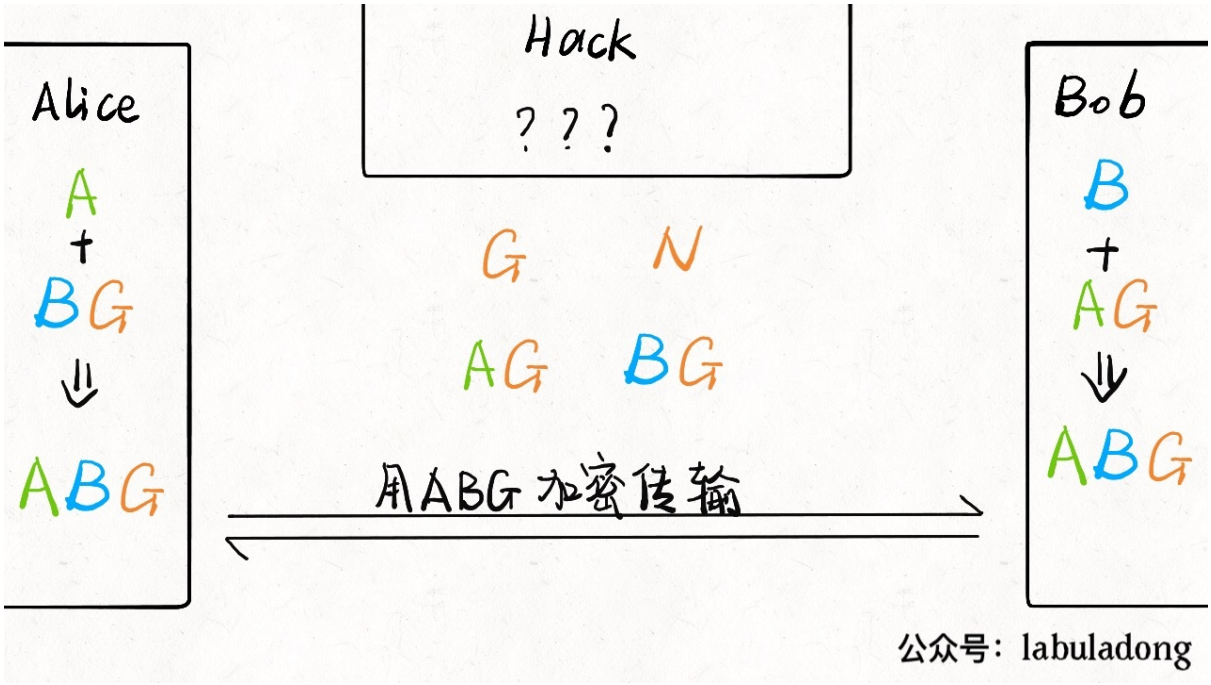
现在的情况变成这样了：



注意，类似刚才举的散列函数的例子，知道 `AG` 和 `G`，并不能反推出 `A` 是多少，`BG` 同理。

那么，Alice 可以通过 `BG` 和自己的 `A` 通过某些运算得到一个数 `ABG`，Bob 也可以通过 `AG` 和自己的 `B` 通过某些运算得到 `ABG`，这个数就是 Alice 和 Bob 共有的秘密。

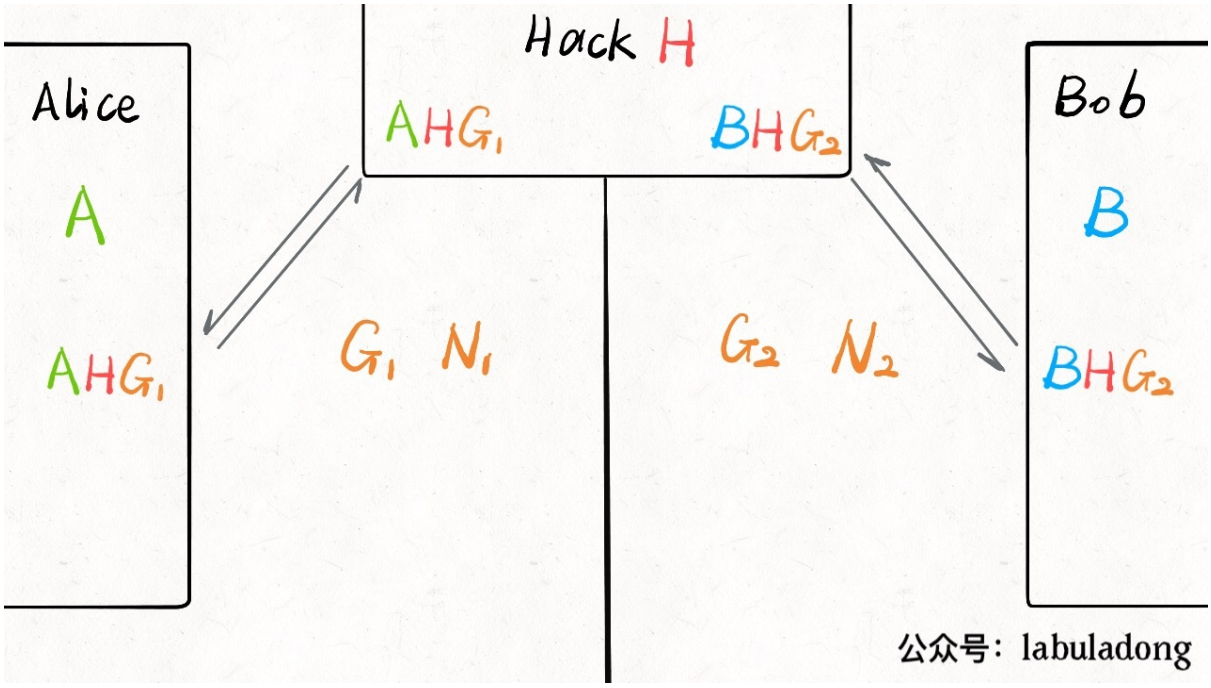
而对于 Hack，可以窃取传输过程中的 `G`，`AG`，`BG`，但是由于计算不可逆，怎么都无法结合出 `ABG` 这个数字。



以上就是基本流程，至于具体的数字取值是有讲究的，运算方法在百度上很容易找到，限于篇幅我就不具体写了。

该算法可以在第三者窃听的前提下，算出一个别人无法算出的秘密作为对称性加密算法的密钥，开始对称加密的通信。

对于该算法，Hack 又想到一种破解方法，不是窃听 Alice 和 Bob 的通信数据，而是直接同时冒充 Alice 和 Bob 的身份，也就是我们说的「**中间人攻击**」：



这样，双方根本无法察觉在和 Hack 共享秘密，后果就是 Hack 可以解密甚至修改数据。

可见，密钥交换算法也不算完全解决了密钥配送问题，缺陷在于无法核实对方身份。所以密钥交换算法之前一般要核实对方身份，比如使用数字签名。

三、非对称加密

非对称加密的思路就是，干脆别偷偷摸摸传输密钥了，我把加密密钥和解密密钥分开，公钥用于加密，私钥用于解密。只把公钥传送给对方，然后对方开始给我发送加密的数据，我用私钥就可以解密。至于窃听者，拿到公钥和加密数据也没用，因为只有我手上的私钥才能解密。

可以这样想，**私钥是钥匙，而公钥是锁，可以把锁公开出去，让别人把数据锁起来发给我；而钥匙一定要留在自己手里，用于解锁**。我们常见的 RSA 算法就是典型的非对称加密算法，具体实现比较复杂，我就不写了，网上很多资料。

在实际应用中，非对称性加密的运算速度要比对称性加密慢很多的，所以传输大量数据时，一般不会用公钥直接加密数据，而是加密对称性加密的密钥，传输给对方，然后双方使用对称性加密算法传输数据。

需要注意的是，类似 Diffie-Hellman 算法，**非对称加密算法也无法确定通信双方的身份，依然会遭到中间人攻击**。比如 Hack 拦截 Bob 发出的公钥，然后冒充 Bob 的身份给 Alice 发送自己的公钥，那么不知情的 Alice 就会把私密数据用 Hack 的公钥加密，Hack 可以通过私钥解密窃取。

那么，Diffie-Hellman 算法和 RSA 非对称加密算法都可以一定程度上解决密钥配送的问题，也具有相同的缺陷，二者的应用场景有什么区别呢？

简单来说，根据两种算法的基本原理就可以看出来：

如果双方有一个对称加密方案，希望加密通信，而且不能让别人得到钥匙，那么可以使用 Diffie-Hellman 算法交换密钥。

如果你希望任何人都可以对信息加密，而只有你能够解密，那么就使用 RSA 非对称加密算法，公布公钥。

下面，我们尝试着解决认证发送方身份的问题。

四、数字签名

刚才说非对称加密，把公钥公开用于他人对数据加密然后发给你，只有用你手上对应的私钥才能将密文解密。其实，**私钥也可用用来加密数据的，对于 RSA 算法，私钥加密的数据只有公钥才能解开**。

数字签名也是利用了非对称性密钥的特性，但是和公钥加密完全颠倒过来：**仍然公布公钥，但是用你的私钥加密数据，然后把加密的数据公布出去，这就是数字签名**。

你可能问，这有什么用，公钥可以解开私钥加密，我还加密发出去，不是多此一举吗？

是的，但是**数字签名的作用本来就不是保证数据的机密性，而是证明你的身份**，证明这些数据确实是由你本人发出的。

你想想，你的私钥加密的数据，只有你的公钥才能解开，那么如果一份加密数据能够被你的公钥解开，不就说明这份数据是你（私钥持有者）本人发布的吗？

当然，加密数据仅仅是一个签名，签名应该和数据一同发出，具体流程应该是：

1、Bob 生成公钥和私钥，然后把公钥公布出去，私钥自己保留。

2、**用私钥加密数据作为签名，然后将数据附带着签名一同发布出去**。

3、Alice 收到数据和签名，需要检查此份数据是否是 Bob 所发出，于是用 Bob 之前发出的公钥尝试解密签名，将收到的数据和签名解密后的结果作对比，如果完全相同，说明数据没被篡改，且确实由 Bob 发出。

为什么 Alice 这么肯定呢，毕竟数据和签名是两部分，都可以被掉包呀？原因如下：

1、如果有人修改了数据，那么 Alice 解密签名之后，对比发现二者不一致，察觉出异常。

2、如果有人替换了签名，那么 Alice 用 Bob 的公钥只能解出一串乱码，显然和数据不一致。

3、也许有人企图修改数据，然后将修改之后的数据制成签名，使得 Alice 的对比无法发现不一致；但是一旦解开签名，就不可能再重新生成 Bob 的签名了，因为没有 Bob 的私钥。

综上，**数字签名可以一定程度上认证数据的来源**。之所以说是一定程度上，是因为这种方式依然可能受到中间人攻击。一旦涉及公钥的发布，接收方就可能收到中间人的假公钥，进行错误的认证，这个问题始终避免不了。

说来可笑，数字签名就是验证对方身份的一种方式，但是前提是对方的身份必须是真的... 这似乎陷入一个先有鸡还是先有蛋的死循环，**要想确定对方的身份，必须有一个信任的源头，否则的话，再多的流程也只是在转移问题，而不是真正解决问题**。

五、公钥证书

证书其实就是公钥 + 签名，由第三方认证机构颁发。引入可信任的第三方，是终结信任循环的一种可行方案。

证书认证的流程大致如下：

1、Bob 去可信任的认证机构证实本人真实身份，并提供自己的公钥。

2、Alice 想跟 Bob 通信，首先向认证机构请求 Bob 的公钥，认证机构会把一张证书（Bob 的公钥以及自己对其公钥的签名）发送给 Alice。

3、Alice 检查签名，确定该公钥确实由这家认证机构发送，中途未被篡改。

4、Alice 通过这个公钥加密数据，开始和 Bob 通信。

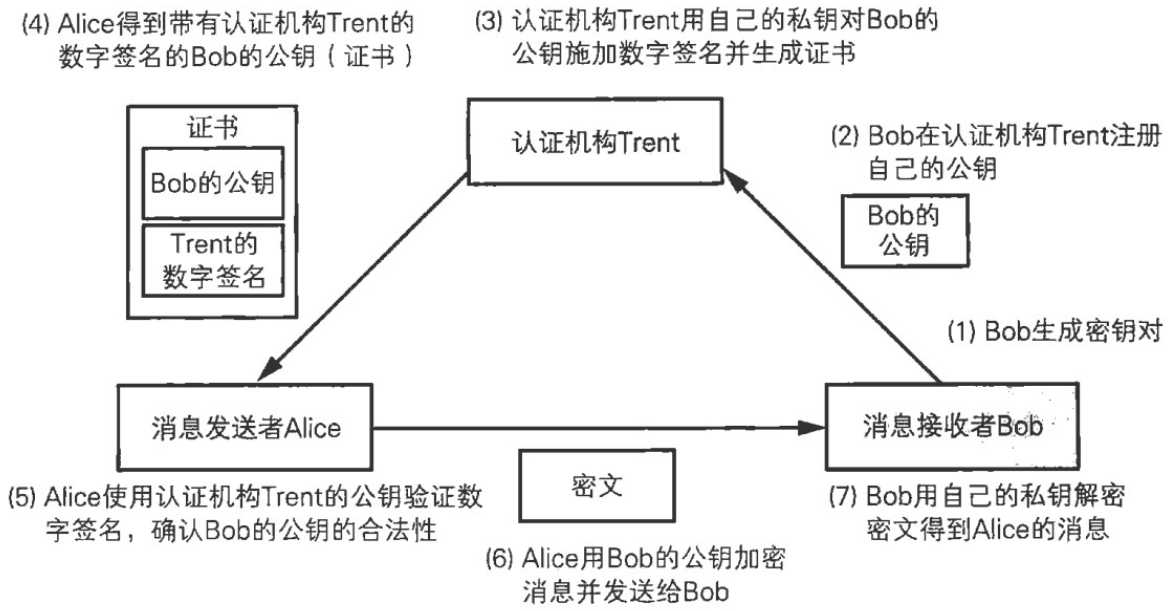


图 10-1　Alice 利用认证机构 Trent 向 Bob 发送密文的示例

PS：以上只是为了说明，证书只需要安装一次，并不需要每次都向认证机构请求；一般是服务器直接给客户端发送证书，而不是认证机构。

也许有人问，Alice 要想通过数字签名确定证书的有效性，前提是要有该机构的（认证）公钥，这不是又回到刚才的死循环了吗？

我们安装的正规浏览器中都预存了正规认证机构的证书（包含其公钥），用于确认机构身份，所以说证书的认证是可信的。

Bob 向机构提供公钥的过程中，需要提供很多个人信息进行身份验证，比较严格，所以说也算是可靠的。

获得了 Bob 的可信公钥，Alice 和 Bob 之间的通信基于加密算法的保护，是完全无懈可击的。

现在的正规网站，大都使用 HTTPS 协议，就是在 HTTP 协议和 TCP 协议之间加了一个 SSL/TLS 安全层。在你的浏览器和网站服务器完成 TCP 握手后，SSL 协议层也会进行 SSL 握手交换安全参数，其中就包含该网站的证书，以便浏览器验证站点身份。SSL 安全层验证完成之后，上层的 HTTP 协议内容都会被加密，保证数据的安全传输。

这样一来，传统的中间人攻击就几乎没有了生存空间，攻击手段只能由技术缺陷转变为坑蒙拐骗。事实上，这种手段的效果反而更高效，比如我就发现**网上不少下载网站发布的浏览器，不仅包含乱七八糟的导航和收藏网址，还包含一些不正规的认证机构证书。任何人都可以申请证书，这些不正规证书很可能造成安全隐患**。

六、最后总结

对称性加密算法使用同一个密钥加密和解密，难以破解，加密速度较快，但是存在密钥配送问题。

Diffie-Hellman 密钥交换算法可以让双方「心有灵犀一点通」，一定程度解决密钥配送问题，但是无法验证通信方的身份，所以可能受到中间人攻击。

非对称性加密算法生成一对儿密钥，把加密和解密的工作分开了。

RSA 算法作为经典的非对称加密算法，有两种用途：如果用于加密，可以把公钥发布出去用于加密，只有自己的私钥可以解密，保证了数据的机密性；如果用于数字签名，把公钥发布出去后，用私钥加密数据作为签名，以

证明该数据由私钥持有者所发送。但是无论那种用法，涉及公钥的发布，都无法避免中间人攻击。

公钥证书就是公钥 + 签名，由可信任的第三方认证机构颁发。由于正规浏览器都预装了可信的认证机构的公钥，所以可以有效防止中间人攻击。

HTTPS 协议中的 SSL/TLS 安全层会组合使用以上几种加密方式，**所以说不要安装非正规的浏览器，不要乱安装未知来源的证书**。

密码技术只是安全的一小部分，即便是通过正规机构认证的 HTTPS 站点，也不意味着可信任，只能说明其数据传输是安全的。技术永远不可能真正保护你，最重要的还是得提高个人的安全防范意识，多留心眼儿，谨慎处理敏感数据。

二分查找高效判定子序列

二分查找本身不难理解，难在巧妙地运用二分查找技巧。对于一个问题，你可能都很难想到它跟二分查找有关，比如前文 最长递增子序列 就借助一个纸牌游戏衍生出二分查找解法。

今天再讲一道巧用二分查找的算法问题：如何判定字符串 `s` 是否是字符串 `t` 的子序列（可以假定 `s` 长度比较小，且 `t` 的长度非常大）。举两个例子：

s = "abc", t = "**a**h**b**gd**c**", return true.

s = "axc", t = "ahbgdc", return false.

题目很容易理解，而且看起来很简单，但很难想到这个问题跟二分查找有关吧？

一、问题分析

首先，一个很简单的解法是这样的：

```
bool isSubsequence(string s, string t) {
    int i = 0, j = 0;
    while (i < s.size() && j < t.size()) {
        if (s[i] == t[j]) i++;
        j++;
    }
    return i == s.size();
}
```

其思路也非常简单，利用双指针 `i, j` 分别指向 `s, t`，一边前进一边匹配子序列：

【pdf/mobi格式不支持GIF:%E5%AD%90%E5%BA%8F%E5%88%97/1.gif】请查看【关于本小抄及作者】章节的解决方案

读者也许会问，这不就是最优解法了吗，时间复杂度只需 O(N)，N 为 `t` 的长度。

是的，如果仅仅是这个问题，这个解法就够好了，**不过这个问题还有 follow up**：

如果给你一系列字符串 `s1,s2,...` 和字符串 `t`，你需要判定每个串 `s` 是否是 `t` 的子序列（可以假定 `s` 较短，`t` 很长）。

```
boolean[] isSubsequence(String[] sn, String t);
```

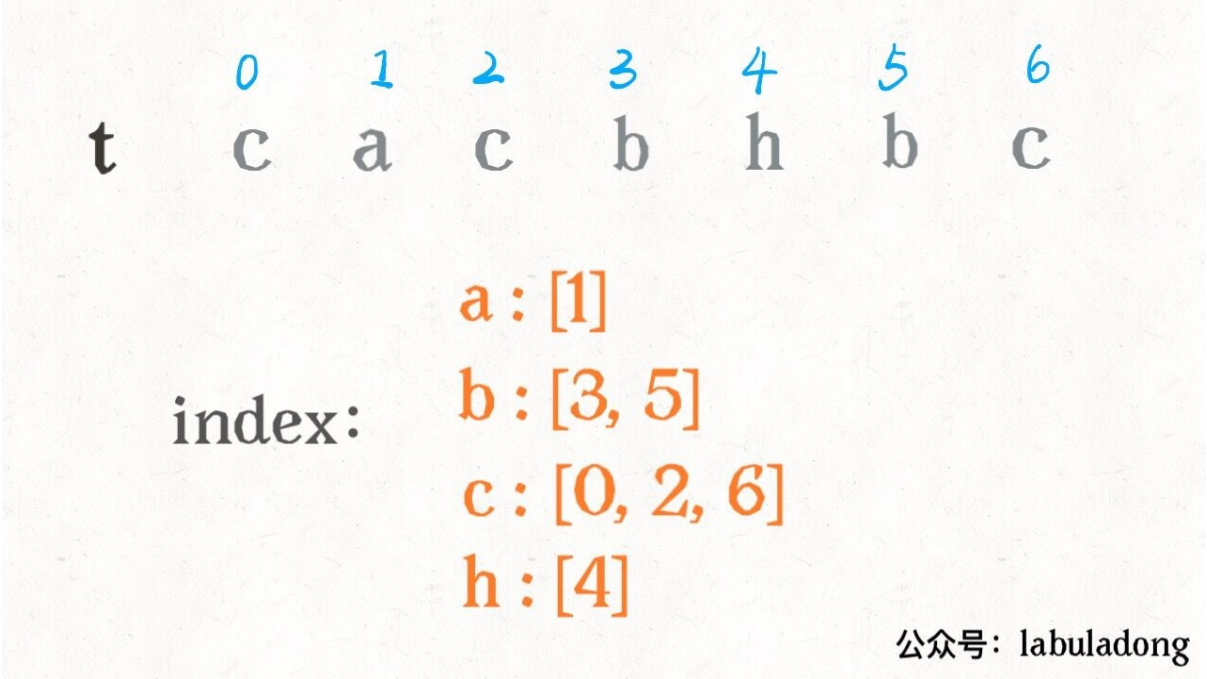
你也许会问，这不是很简单吗，还是刚才的逻辑，加个 for 循环不就行了？

可以，但是此解法处理每个 `s` 时间复杂度仍然是 O(N)，而如果巧妙运用二分查找，可以将时间复杂度降低，大约是 O(MlogN)。由于 N 相对 M 大很多，所以后者效率会更高。

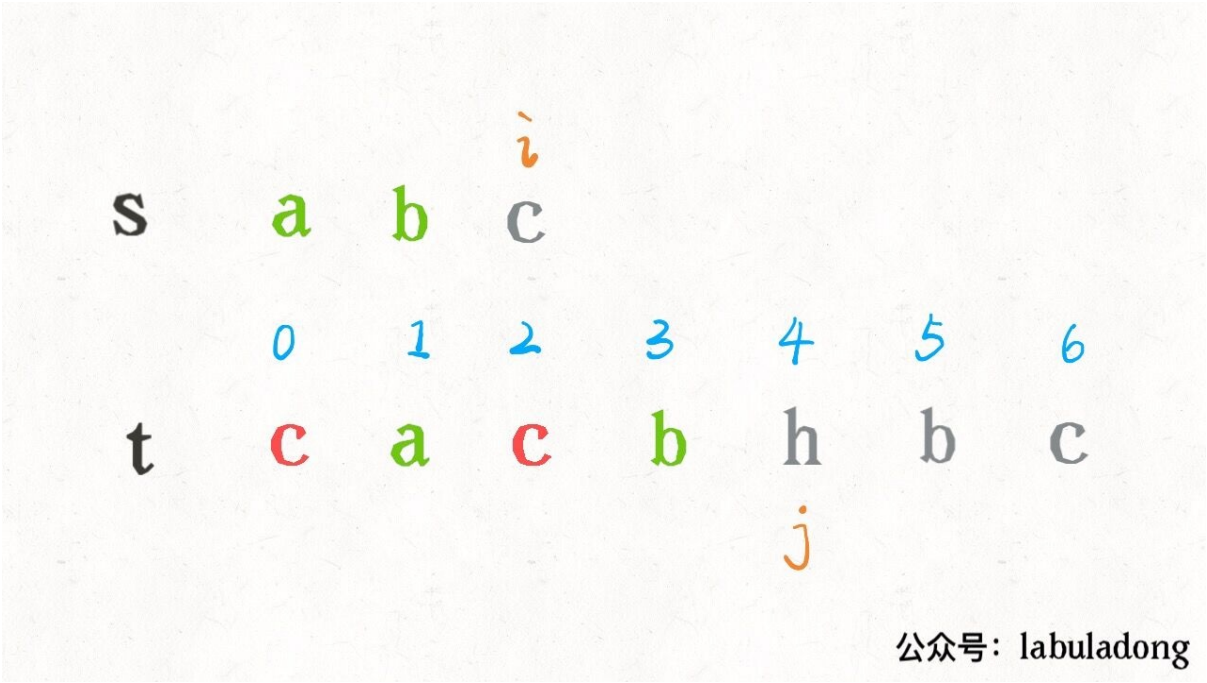
二、二分思路

二分思路主要是对 `t` 进行预处理，用一个字典 `index` 将每个字符出现的索引位置按顺序存储下来：

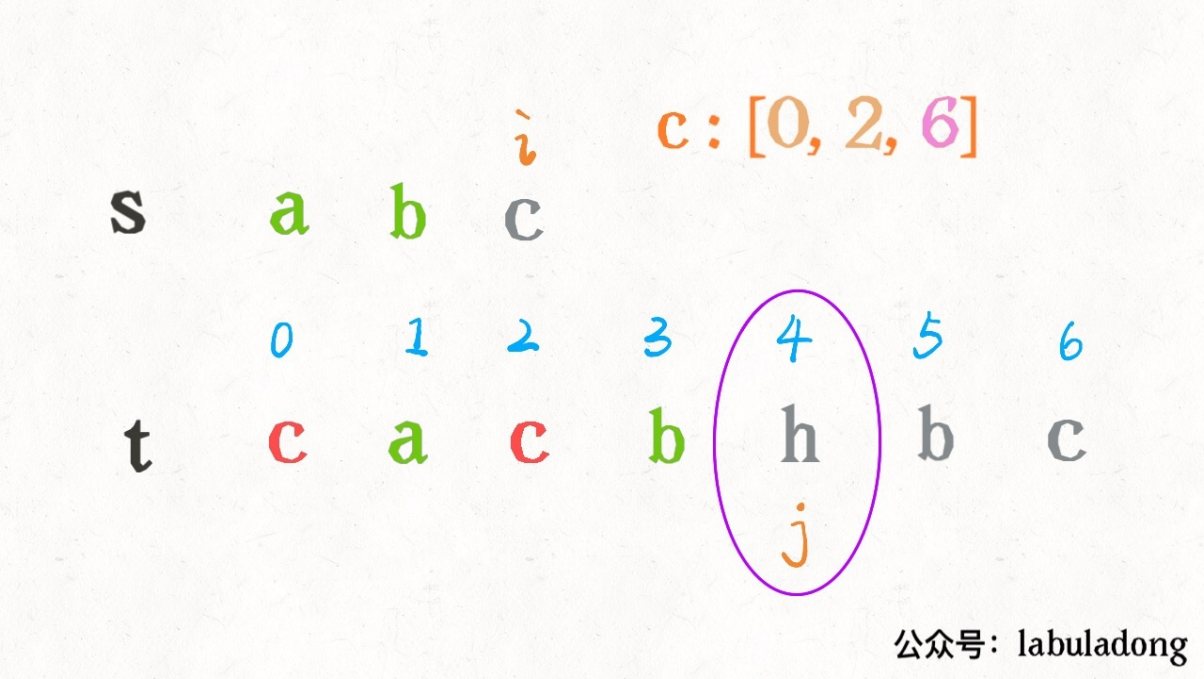
```
int m = s.length(), n = t.length();
ArrayList<Integer>[] index = new ArrayList[256];
// 先记下 t 中每个字符出现的位置
for (int i = 0; i < n; i++) {
    char c = t.charAt(i);
    if (index[c] == null)
        index[c] = new ArrayList<>();
    index[c].add(i);
}
```



比如对于这个情况，匹配了 "ab"，应该匹配 "c" 了：



按照之前的解法，我们需要 `j` 线性前进扫描字符 "c"，但借助 `index` 中记录的信息，**可以二分搜索 `index[c]` 中比 j 大的那个索引**，在上图的例子中，就是在 `[0,2,6]` 中搜索比 4 大的那个索引：



这样就可以直接得到下一个 "c" 的索引。现在的问题就是，如何用二分查找计算那个恰好比 4 大的索引呢？答案是，寻找左侧边界的二分搜索就可以做到。

三、再谈二分查找

在前文 二分查找详解 中，详解了如何正确写出三种二分查找算法的细节。二分查找返回目标值 `val` 的索引，对于搜索**左侧边界**的二分查找，有一个特殊性质：

当 `val` 不存在时，得到的索引恰好是比 `val` 大的最小元素索引。

什么意思呢，就是说如果在数组 `[0,1,3,4]` 中搜索元素 2，算法会返回索引 2，也就是元素 3 的位置，元素 3 是数组中大于 2 的最小元素。所以我们可以利用二分搜索避免线性扫描。

```
// 查找左侧边界的二分查找
int left_bound(ArrayList<Integer> arr, int tar) {
    int lo = 0, hi = arr.size();
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if (tar > arr.get(mid)) {
            lo = mid + 1;
```

```
            } else {
                hi = mid;
            }
        }
        return lo;
    }
```

以上就是搜索左侧边界的二分查找，等会儿会用到，其中的细节可以参见前文《二分查找详解》，这里不再赘述。

四、代码实现

这里以单个字符串 `s` 为例，对于多个字符串 `s`，可以把预处理部分抽出来。

```
boolean isSubsequence(String s, String t) {
    int m = s.length(), n = t.length();
    // 对 t 进行预处理
    ArrayList<Integer>[] index = new ArrayList[256];
    for (int i = 0; i < n; i++) {
        char c = t.charAt(i);
        if (index[c] == null)
            index[c] = new ArrayList<>();
        index[c].add(i);
    }

    // 串 t 上的指针
    int j = 0;
    // 借助 index 查找 s[i]
    for (int i = 0; i < m; i++) {
        char c = s.charAt(i);
        // 整个 t 压根儿没有字符 c
        if (index[c] == null) return false;
        int pos = left_bound(index[c], j);
        // 二分搜索区间中没有找到字符 c
        if (pos == index[c].size()) return false;
        // 向前移动指针 j
        j = index[c].get(pos) + 1;
    }
    return true;
```

```
}
```

算法执行的过程是这样的：

【pdf/mobi格式不支持GIF:%E5%AD%90%E5%BA%8F%E5%88%97/2.gif】请查看【关于本小抄及作者】章节的解决方案

可见借助二分查找，算法的效率是可以大幅提升的。