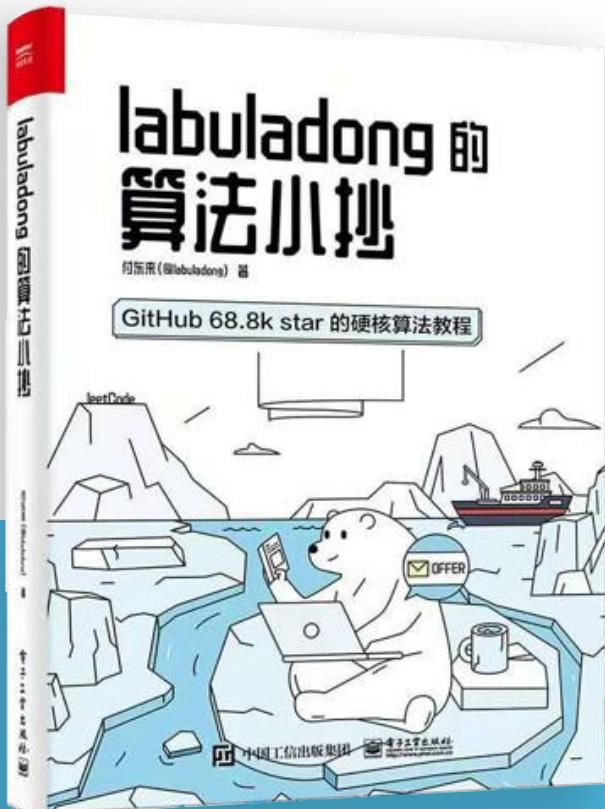


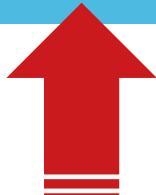
《labuladong的算法小抄》2020年12月正式出版，
请在labuladong公众号或[online book](#)关注最新信息。



搞定算法题 拿大厂Offer

Github 68.8k star 的硬核算法教程

憋出内伤，出了本



本PDF是《labuladong算法小抄》的简版，欢迎大家扫码购买。

Table of Contents

开篇词	1.1
第零章、必读系列	1.2
学习算法和刷题的框架思维	1.2.1
动态规划解题套路框架	1.2.2
回溯算法解题套路框架	1.2.3
BFS 算法解题套路框架	1.2.4
我写了首诗，让你闭着眼睛也能写对二分搜索	1.2.5
我写了首诗，把滑动窗口算法变成了默写题	
一个方法团灭 LeetCode 打家劫舍问题	1.2.7
	1.2.6
第一章、动态规划系列	2.1
动态规划答疑篇	2.1.1
动态规划设计：最大子数组	2.1.2
经典动态规划：0-1 背包问题	2.1.3
经典动态规划：完全背包问题	2.1.4
经典动态规划：编辑距离	2.1.5
经典动态规划：高楼扔鸡蛋（进阶）	2.1.6
动态规划之正则表达	2.1.7
第二章、数据结构系列	3.1
LRU算法详解	3.1.1
二叉搜索树操作集锦	3.1.2
如何计算完全二叉树的节点数	3.1.3
递归反转链表的一部分	3.1.4
队列实现栈 栈实现队列	3.1.5

学习算法和刷题的框架思维

第三章、算法思维系列	4.1
回溯算法团灭子集、排列、组合问题	4.1.1
回溯算法最佳实践：括号生成	4.1.2
信封嵌套问题	4.1.3
几个反直觉的概率问题	4.1.4

学习算法和刷题的思路指南

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。

学习算法和刷题的框架思维

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

学习算法和刷题的框架思维

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]
    }
}
```

学习算法和刷题的框架思维

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：

学习算法和刷题的框架思维



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。

学习算法和刷题的框架思维

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> inMap) {
    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 - 1,
                          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，本质上该算法也是一个前序遍历。

学习算法和刷题的框架思维

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。

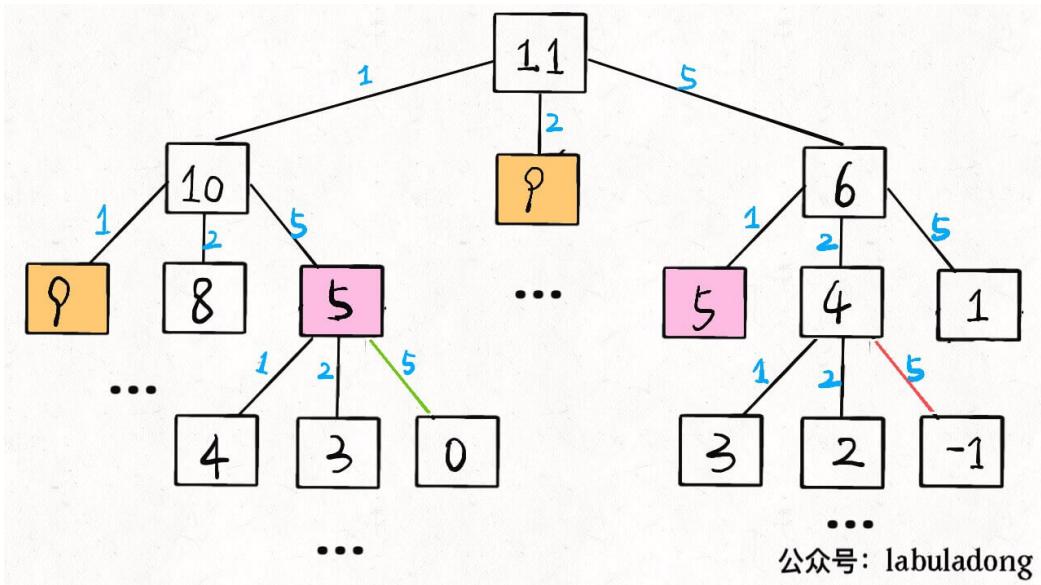
你看，Hard 难度的题目不过如此，而且还这么有规律可循，只要把框架写出来，然后往相应的位置加东西就行了，这不就是思路吗。

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：

学习算法和刷题的框架思维



```
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)
```

学习算法和刷题的框架思维

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。

学习算法和刷题的框架思维

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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。

学习算法和刷题的框架思维



微信搜一搜

Q labuladong

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！

学习算法和刷题的框架思维

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

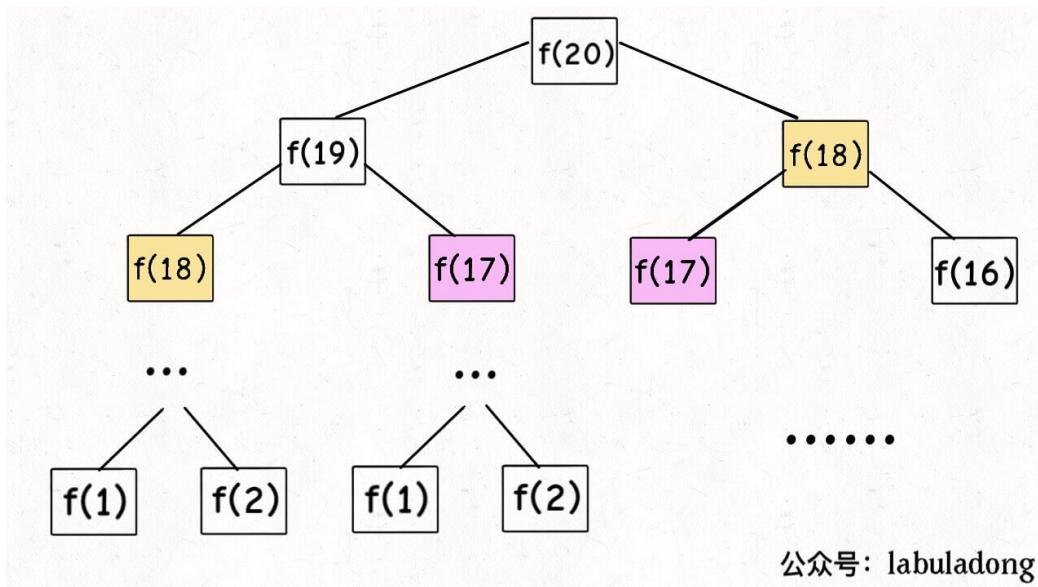
学习算法和刷题的框架思维

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。

学习算法和刷题的框架思维

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。既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。

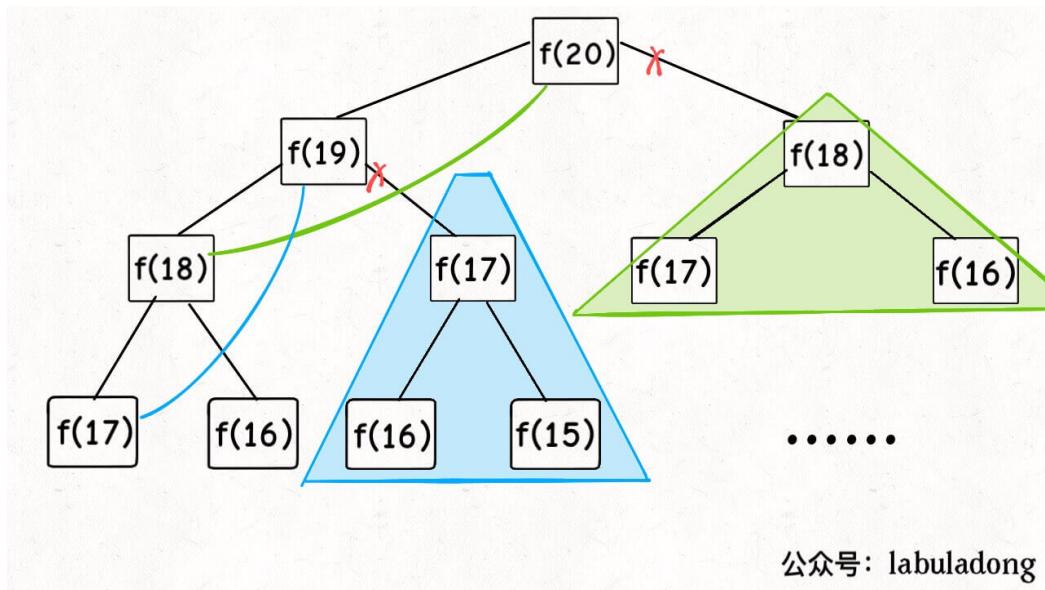
一般使用一个数组充当这个「备忘录」，当然你也可以使用哈希表（字典），思想都是一样的。

学习算法和刷题的框架思维

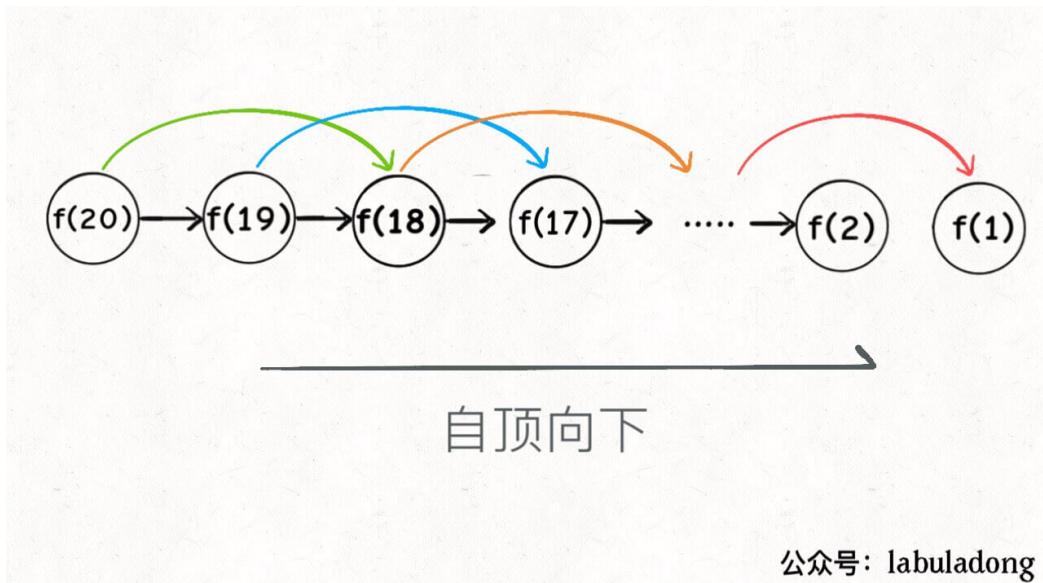
```
int fib(int N) {
    if (N < 1) return 0;
    // 备忘录全初始化为 0
    vector<int> memo(N + 1, 0);
    // 初始化最简情况
    return helper(memo, N);
}

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」。

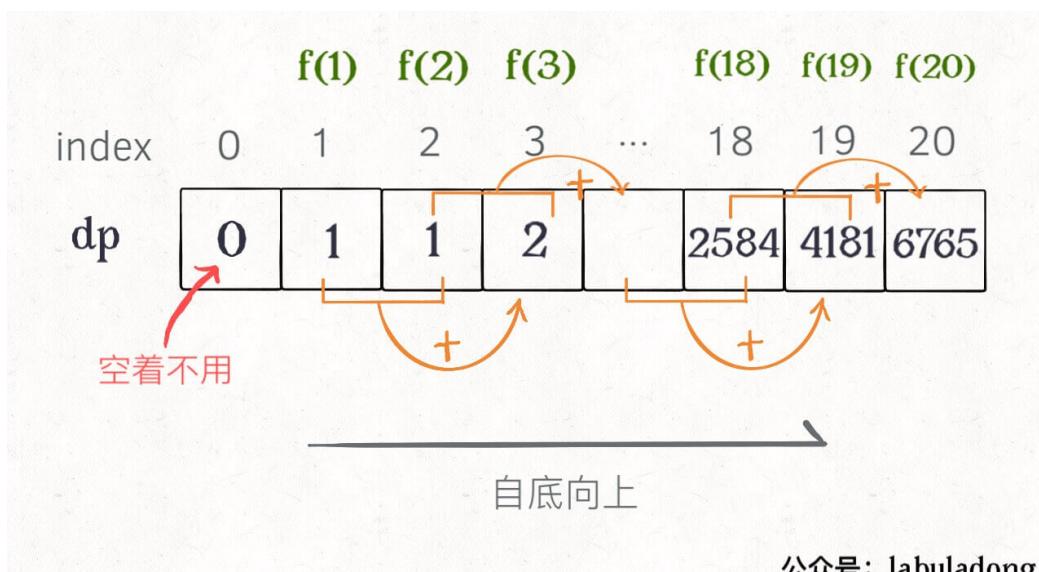
啥叫「自顶向下」？注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说 $f(20)$ ，向下逐渐分解规模，直到 $f(1)$ 和 $f(2)$ 触底，然后逐层返回答案，这就叫「自顶向下」。

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，

学习算法和刷题的框架思维

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 $\text{return } f(n - 1) + f(n - 2)$, $\text{dp}[i] = \text{dp}[i - 1] + \text{dp}[i - 2]$, 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，就是把所有可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

学习算法和刷题的框架思维

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约，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`。

学习算法和刷题的框架思维

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：

学习算法和刷题的框架思维

```
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)

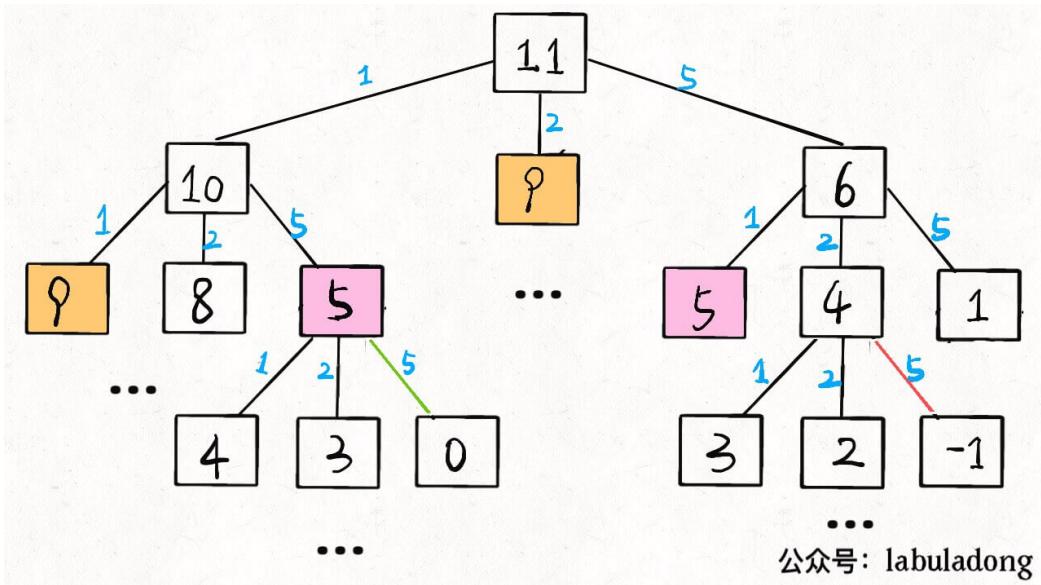
        return res if res != float('INF') else -1

    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：

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

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 \times 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

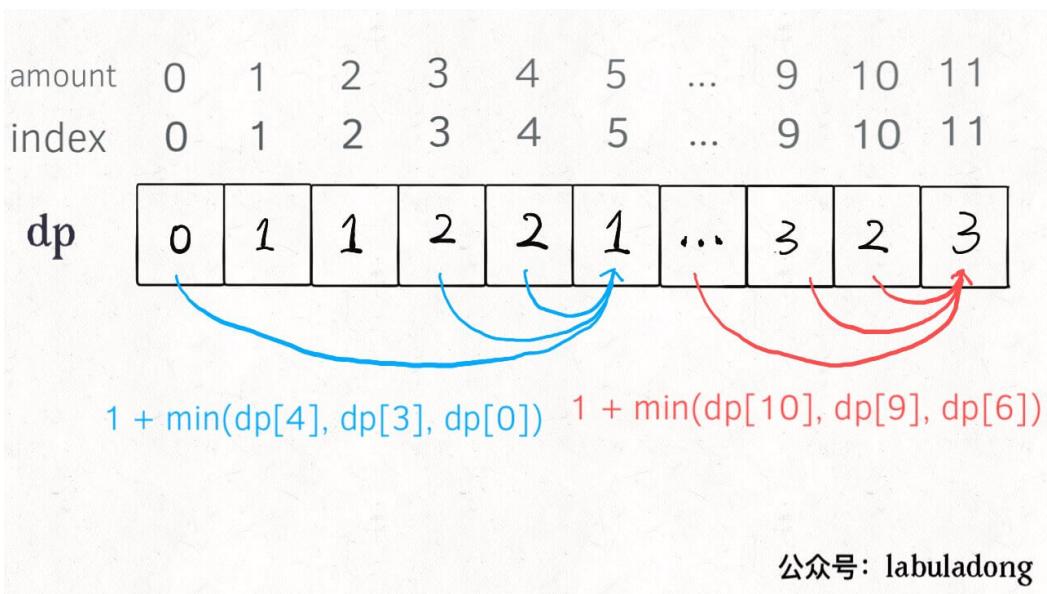
当然，我们也可以自底向上使用 dp table 来消除重叠子问题，`dp` 数组的定义和刚才 `dp` 函数类似，定义也是一样的：

`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];
}

```



公众号: labuladong

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？

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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

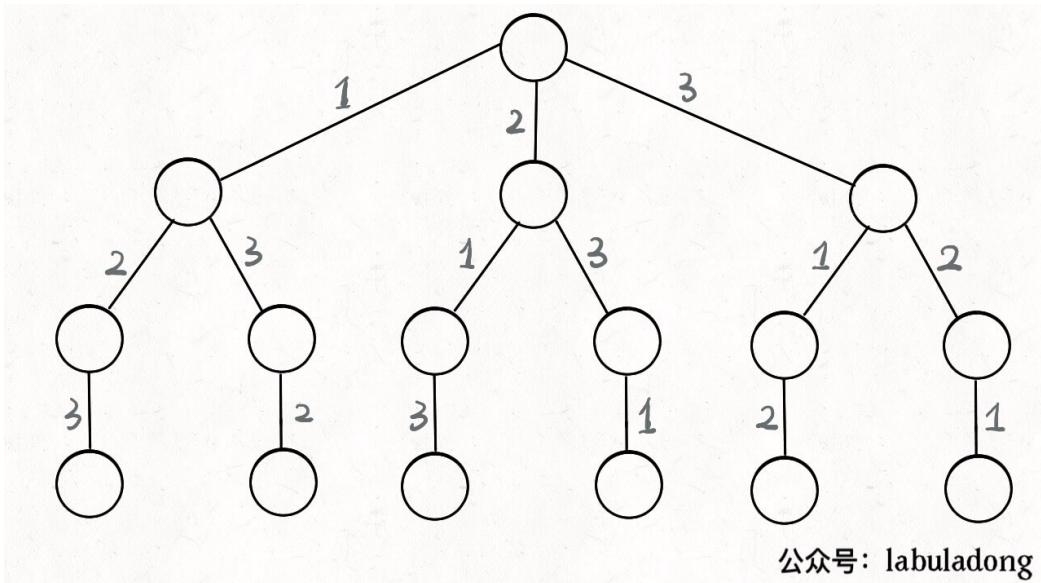
我们在高中的时候就做过排列组合的数学题，我们也知道 n 个不重复的数，全排列共有 $n!$ 个。

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：

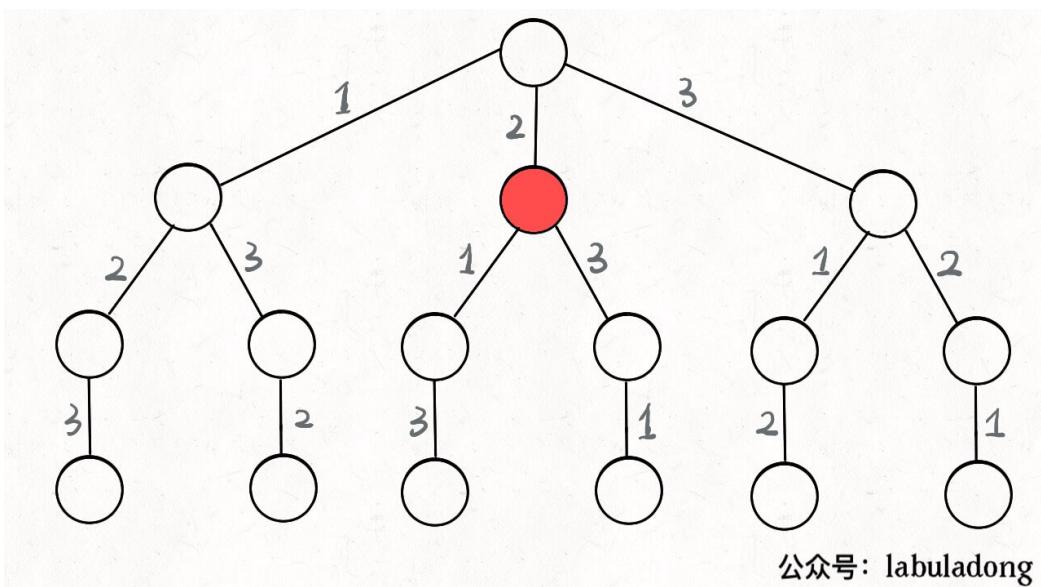
先固定第一位为 1，然后第二位可以是 2，那么第三位只能是 3；然后可以把第二位变成 3，第三位就只能是 2 了；然后就只能变化第一位，变成 2，然后再穷举后两位……

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」。

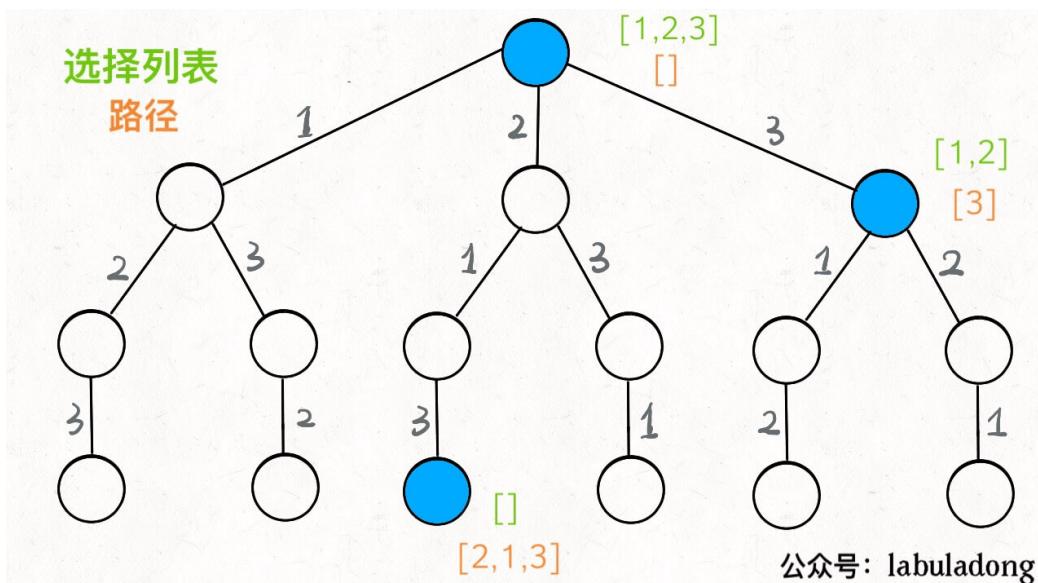
为啥说这是决策树呢，因为你在每个节点上其实都在做决策。比如说你站在下图的红色节点上：



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：

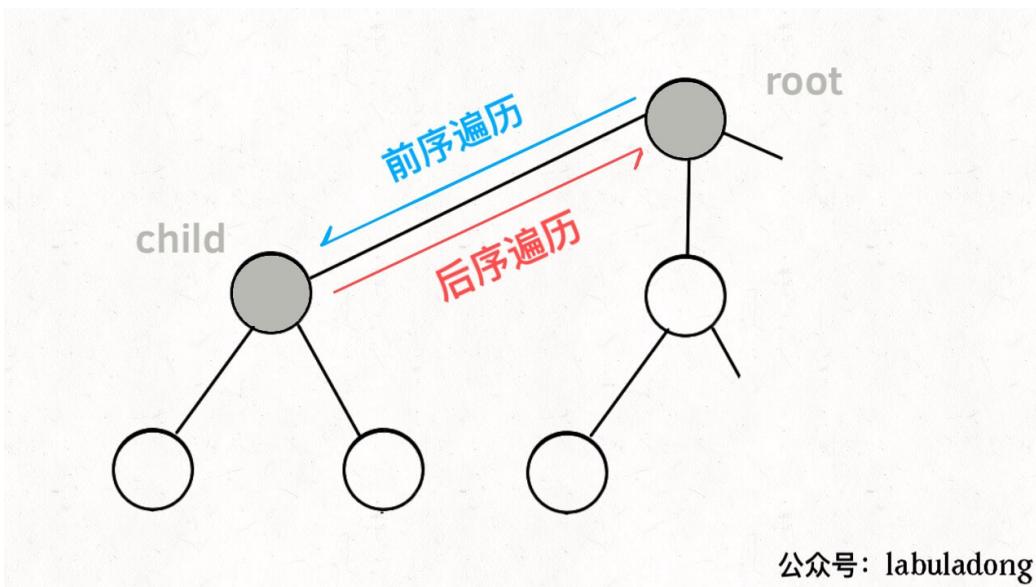


我们定义的 `backtrack` 函数其实就像一个指针，在这棵树上游走，同时要正确维护每个节点的属性，每当走到树的底层，其「路径」就是一个全排列。

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：

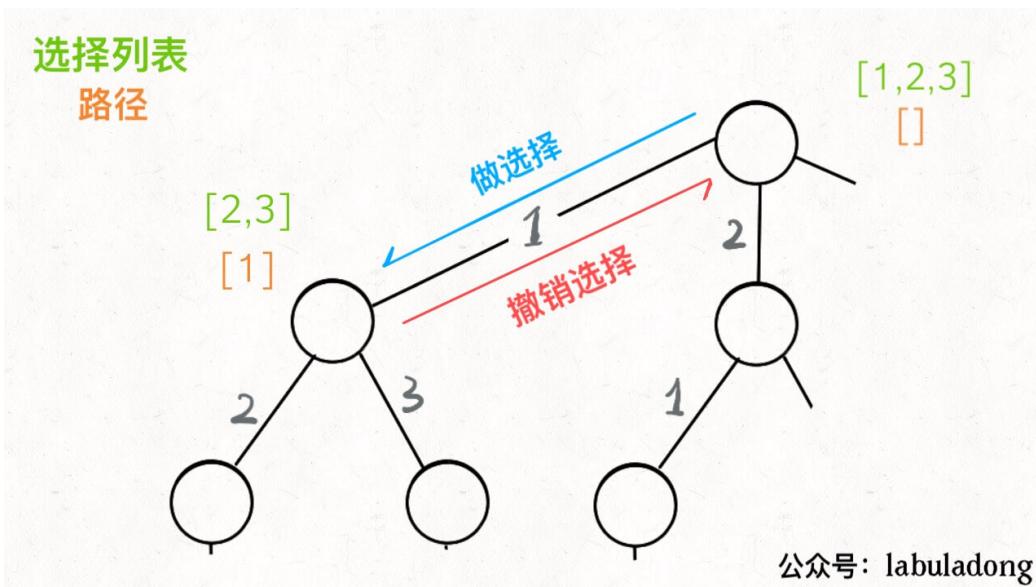
```
void traverse(TreeNode root) {  
    for (TreeNode child : root.children)  
        // 前序遍历需要的操作  
        traverse(child);  
        // 后序遍历需要的操作  
}
```

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？

学习算法和刷题的框架思维

```
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：

学习算法和刷题的框架思维

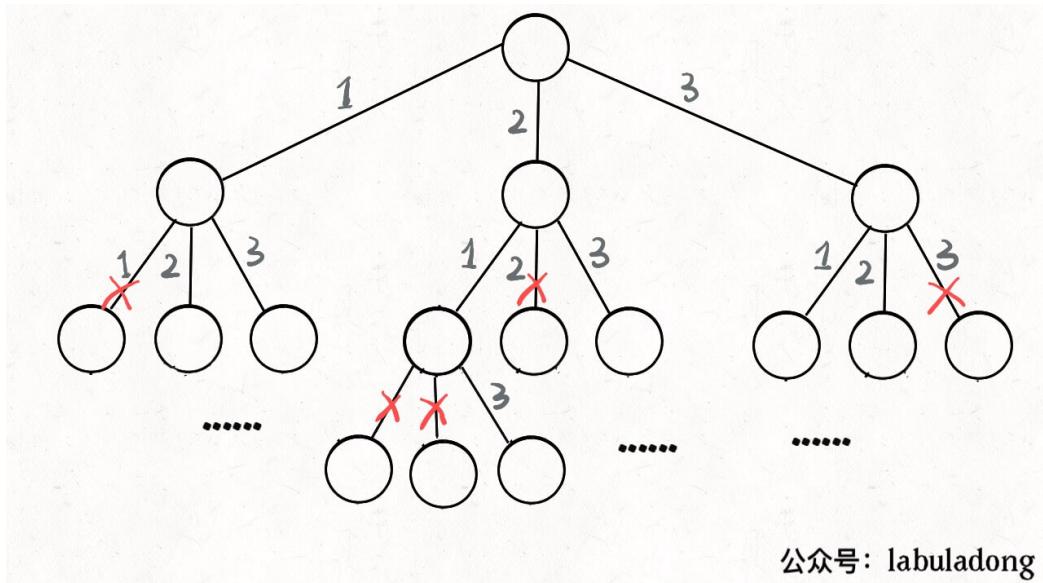
```
List<List<Integer>> res = new LinkedList<>();

/* 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 $N \times N$ 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。

学习算法和刷题的框架思维

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;
    }

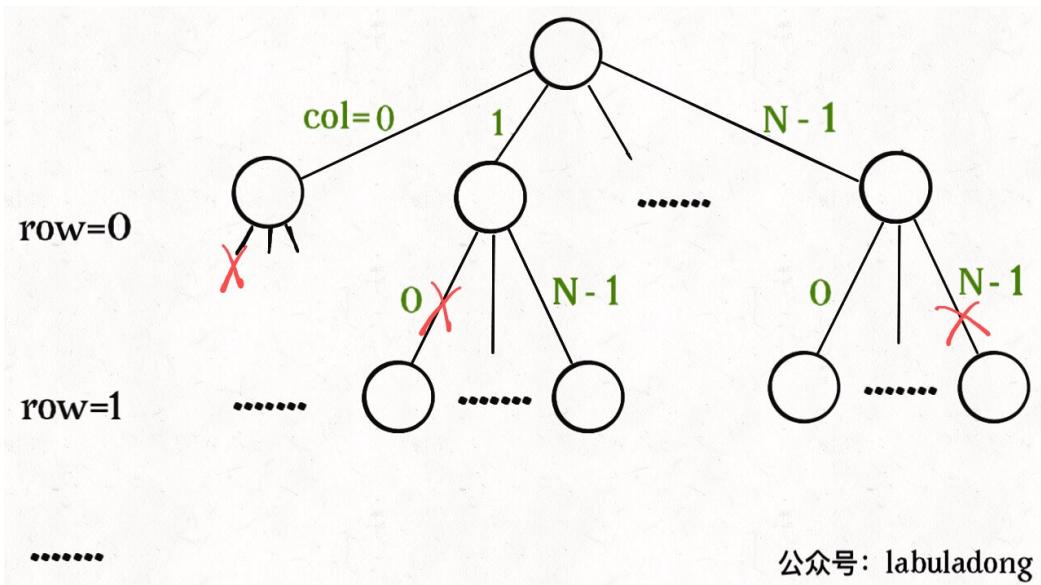
    int n = board[row].size();
    for (int col = 0; col < n; col++) {
        // 排除不合法选择
        if (!isValid(board, row, col))
            continue;
        // 做选择
        board[row][col] = 'Q';
        // 进入下一行决策
        backtrack(board, row + 1);
        // 撤销选择
        board[row][col] = '.';
    }
}
```

学习算法和刷题的框架思维

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：

学习算法和刷题的框架思维

```
// 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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

BFS 算法框架套路详解

后台有很多人问起 BFS 和 DFS 的框架，今天就来说说吧。

首先，你要说 labuladong 没写过 BFS 框架，这话没错，今天写个框架你背住就完事儿了。但要是说没写过 DFS 框架，那你还真是说错了，[其实 DFS 算法就是回溯算法](#)，我们前文[回溯算法框架套路详解](#)就写过了，而且写得不是一般得好，建议好好复习。

BFS 的核心思想应该不难理解的，就是把一些问题抽象成图，从一个点开始，向四周开始扩散。一般来说，我们写 BFS 算法都是用「队列」这种数据结构，每次将一个节点周围的所有节点加入队列。

BFS 相对 DFS 的最主要的区别是：**BFS 找到的路径一定是最短的，但代价就是空间复杂度比 DFS 大很多**，至于为什么，我们后面介绍了框架就很容易看出来了。

本文就由浅入深写两道 BFS 的典型题目，分别是「二叉树的最小高度」和「打开密码锁的最少步数」，手把手教你怎么写 BFS 算法。

一、算法框架

要说框架的话，我们先举例一下 BFS 出现的常见场景好吧，问题的本质就是让你在一幅「图」中找到从起点 `start` 到终点 `target` 的最近距离，这个例子听起来很枯燥，但是 **BFS 算法问题其实都是在干这个事儿**，把枯燥的本质搞清楚了，再去欣赏各种问题的包装才能胸有成竹嘛。

学习算法和刷题的框架思维

这个广义的描述可以有各种变体，比如走迷宫，有的格子是围墙不能走，从起点到终点的最短距离是多少？如果这个迷宫带「传送门」可以瞬间传送呢？

再比如说两个单词，要求你通过某些替换，把其中一个变成另一个，每次只能替换一个字符，最少要替换几次？

再比如说连连看游戏，两个方块消除的条件不仅仅是图案相同，还得保证两个方块之间的最短连线不能多于两个拐点。你玩连连看，点击两个坐标，游戏是如何判断它俩的最短连线有几个拐点的？

再比如……

净整些花里胡哨的，这些问题都没啥奇技淫巧，本质上就是一幅「图」，让你从一个起点，走到终点，问最短路径。这就是BFS的本质，框架搞清楚了直接默写就好。



别给我整

这些的

记住下面这个框架就OK了：

```
// 计算从起点 start 到终点 target 的最近距离
int BFS(Node start, Node target) {
    Queue<Node> q; // 核心数据结构
    Set<Node> visited; // 避免走回头路

    q.offer(start); // 将起点加入队列
    visited.add(start);
    int step = 0; // 记录扩散的步数

    while (q not empty) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            Node cur = q.poll();
            /* 划重点：这里判断是否到达终点 */
            if (cur is target)
                return step;
            /* 将 cur 的相邻节点加入队列 */
            for (Node x : cur.adj())
                if (x not in visited) {
                    q.offer(x);
                    visited.add(x);
                }
        }
        /* 划重点：更新步数在这里 */
        step++;
    }
}
```

队列 `q` 就不说了，BFS 的核心数据结构；`cur.adj()` 泛指 `cur` 相邻的节点，比如说二维数组中，`cur` 上下左右四面的位置就是相邻节点；`visited` 的主要作用是防止走回头路，大部分时候都是必须的，但是像一般的二叉树结构，没有子节点到父节点的指针，不会走回头路就不需要 `visited`。

二、二叉树的最小高度

先来个简单的问题实践一下 BFS 框架吧，判断一棵二叉树的最小高度，这也是 LeetCode 第 111 题，看一下题目：

111. 二叉树的最小深度

难度 简单 242 收藏 分享 切换为英文

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3, 9, 20, null, null, 15, 7]，



返回它的最小深度 2.

怎么套到 BFS 的框架里呢？首先明确一下起点 `start` 和终点 `target` 是什么，怎么判断到达了终点？

显然起点就是 `root` 根节点，终点就是最靠近根节点的那个「叶子节点」嘛，叶子节点就是两个子节点都是 `null` 的节点：

```
if (cur.left == null && cur.right == null)  
    // 到达叶子节点
```

那么，按照我们上述的框架稍加改造来写解法即可：

```
int minDepth(TreeNode root) {
    if (root == null) return 0;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);
    // root 本身就是一层，depth 初始化为 1
    int depth = 1;

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向四周扩散 */
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            /* 判断是否到达终点 */
            if (cur.left == null && cur.right == null)
                return depth;
            /* 将 cur 的相邻节点加入队列 */
            if (cur.left != null)
                q.offer(cur.left);
            if (cur.right != null)
                q.offer(cur.right);
        }
        /* 这里增加步数 */
        depth++;
    }
    return depth;
}
```

二叉树是很简单的数据结构，我想上述代码你应该可以理解的吧，其实其他复杂问题都是这个框架的变形，再探讨复杂问题之前，我们解答两个问题：

1、为什么 BFS 可以找到最短距离，DFS 不行吗？

首先，你看 BFS 的逻辑，`depth` 每增加一次，队列中的所有节点都向前迈一步，这保证了第一次到达终点的时候，走的步数是最少的。

DFS 不能找最短路径吗？其实也是可以的，但是时间复杂度相对高很多。你想啊，DFS 实际上是靠递归的堆栈记录走过的路径，你要找到最短路径，肯定得把二叉树中所有树权都探索完才能对比出最短的路径有多长对不对？而 BFS 借助队列做到一次一步「齐头并进」，是可以在不遍历完整棵树的条件下找到最短距离的。

形象点说，DFS 是线，BFS 是面；DFS 是单打独斗，BFS 是集体行动。这个应该比较容易理解吧。

2、既然 BFS 那么好，为啥 DFS 还要存在？

BFS 可以找到最短距离，但是空间复杂度高，而 DFS 的空间复杂度较低。

还是拿刚才我们处理二叉树问题的例子，假设给你的这个二叉树是满二叉树，节点数为 N ，对于 DFS 算法来说，空间复杂度无非就是递归堆栈，最坏情况下顶多就是树的高度，也就是 $O(\log N)$ 。

但是你想想 BFS 算法，队列中每次都会储存着二叉树一层的节点，这样的话最坏情况下空间复杂度应该是树的最底层节点的数量，也就是 $N/2$ ，用 Big O 表示的话也就是 $O(N)$ 。

由此观之，BFS 还是有代价的，一般来说在找最短路径的时候使用 BFS，其他时候还是 DFS 使用得多一些（主要是递归代码好写）。

好了，现在你对 BFS 了解得足够多了，下面来一道难一点的题目，深化一下框架的理解吧。

三、解开密码锁的最少次数

这道 LeetCode 题目是第 752 题，比较有意思：

学习算法和刷题的框架思维

752. 打开转盘锁

难度 中等 例 97 收藏 分享 切换为英文 关注 反馈

你有一个带有四个圆形拨轮的转盘锁。每个拨轮都有10个数字：'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'。每个拨轮可以自由旋转：例如把 '9' 变为 '0'， '0' 变为 '9'。每次旋转都只能旋转一个拨轮的一位数字。

锁的初始数字为 '0000'，一个代表四个拨轮的数字的字符串。

列表 `deadends` 包含了一组死亡数字，一旦拨轮的数字和列表里的任何一个元素相同，这个锁将会被永久锁定，无法再被旋转。

字符串 `target` 代表可以解锁的数字，你需要给出最小的旋转次数，如果无论如何不能解锁，返回 -1。

示例 1：

```
输入: deadends = ["0201","0101","0102","1212","2002"], target = "0202"
输出: 6
解释:
可能的移动序列为 "0000" -> "1000" -> "1100" -> "1200" -> "1201" -> "1202" ->
"0202"。
注意 "0000" -> "0001" -> "0002" -> "0102" -> "0202" 这样的序列是不能解锁的,
因为当拨动到 "0102" 时这个锁就会被锁定。
```

示例 2：

```
输入: deadends = ["8888"], target = "0009"
输出: 1
解释:
把最后一位反向旋转一次即可 "0000" -> "0009"。
```

示例 3：

```
输入: deadends = ["8887","8889","8878","8898","8788","8988","7888","9888"],
target = "8888"
输出: -1
解释:
无法旋转到目标数字且不被锁定。
```

题目中描述的就是我们生活中常见的那种密码锁，若果没有任何约束，最少的拨动次数很好算，就像我们平时开密码锁那样直奔密码拨就行了。

但现在的难点就在于，不能出现 `deadends`，应该如何计算出最少的转动次数呢？

第一步，我们不管所有的限制条件，不管 `deadends` 和 `target` 的限制，就思考一个问题：如果让你设计一个算法，穷举所有可能的密码组合，你怎么做？

穷举呗，再简单一点，如果你只转一下锁，有几种可能？总共有 4 个位置，每个位置可以向上转，也可以向下转，也就是有 8 种可能对吧。

比如说从 "0000" 开始，转一次，可以穷举出 "1000", "9000", "0100", "0900"… 共 8 种密码。然后，再以这 8 种密码作为基础，对每个密码再转一下，穷举出所有可能…

仔细想想，这就可以抽象成一幅图，每个节点有 8 个相邻的节点，又让你求最短距离，这不就是典型的 BFS 嘛，框架就可以派上用场了，先写出一个「简陋」的 BFS 框架代码再说别的：

学习算法和刷题的框架思维

```
// 将 s[j] 向上拨动一次
String plusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '9')
        ch[j] = '0';
    else
        ch[j] += 1;
    return new String(ch);
}

// 将 s[i] 向下拨动一次
String minusOne(String s, int j) {
    char[] ch = s.toCharArray();
    if (ch[j] == '0')
        ch[j] = '9';
    else
        ch[j] -= 1;
    return new String(ch);
}

// BFS 框架，打印出所有可能的密码
void BFS(String target) {
    Queue<String> q = new LinkedList<>();
    q.offer("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向周围扩散 */
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();
            /* 判断是否到达终点 */
            System.out.println(cur);

            /* 将一个节点的相邻节点加入队列 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                String down = minusOne(cur, j);
                q.offer(up);
                q.offer(down);
            }
        }
    }
    /* 在这里增加步数 */
}
```

```
    }  
    return;  
}
```

PS：这段代码当然有很多问题，但是我们做算法题肯定不是一蹴而就的，而是从简陋到完美的。不要完美主义，咱要慢慢来，好不。

这段 BFS 代码已经能够穷举所有可能的密码组合了，但是显然不能完成题目，有如下问题需要解决：

- 1、会走回头路。比如说我们从 "0000" 拨到 "1000"，但是等从队列拿出 "1000" 时，还会拨出一个 "0000"，这样的话会产生死循环。
- 2、没有终止条件，按照题目要求，我们找到 target 就应该结束并返回拨动的次数。
- 3、没有对 deadends 的处理，按道理这些「死亡密码」是不能出现的，也就是说你遇到这些密码的时候需要跳过。

如果你能够看懂上面那段代码，真得给你鼓掌，只要按照 BFS 框架在对应的位置稍作修改即可修复这些问题：

学习算法和刷题的框架思维

```
int openLock(String[] deadends, String target) {
    // 记录需要跳过的死亡密码
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 记录已经穷举过的密码，防止走回头路
    Set<String> visited = new HashSet<>();
    Queue<String> q = new LinkedList<>();
    // 从起点开始启动广度优先搜索
    int step = 0;
    q.offer("0000");
    visited.add("0000");

    while (!q.isEmpty()) {
        int sz = q.size();
        /* 将当前队列中的所有节点向周围扩散 */
        for (int i = 0; i < sz; i++) {
            String cur = q.poll();

            /* 判断是否到达终点 */
            if (deads.contains(cur))
                continue;
            if (cur.equals(target))
                return step;

            /* 将一个节点的未遍历相邻节点加入队列 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up)) {
                    q.offer(up);
                    visited.add(up);
                }
                String down = minusOne(cur, j);
                if (!visited.contains(down)) {
                    q.offer(down);
                    visited.add(down);
                }
            }
        }
        /* 在这里增加步数 */
        step++;
    }
}
```

```
// 如果穷举完都没找到目标密码，那就是找不到了  
return -1;  
}
```

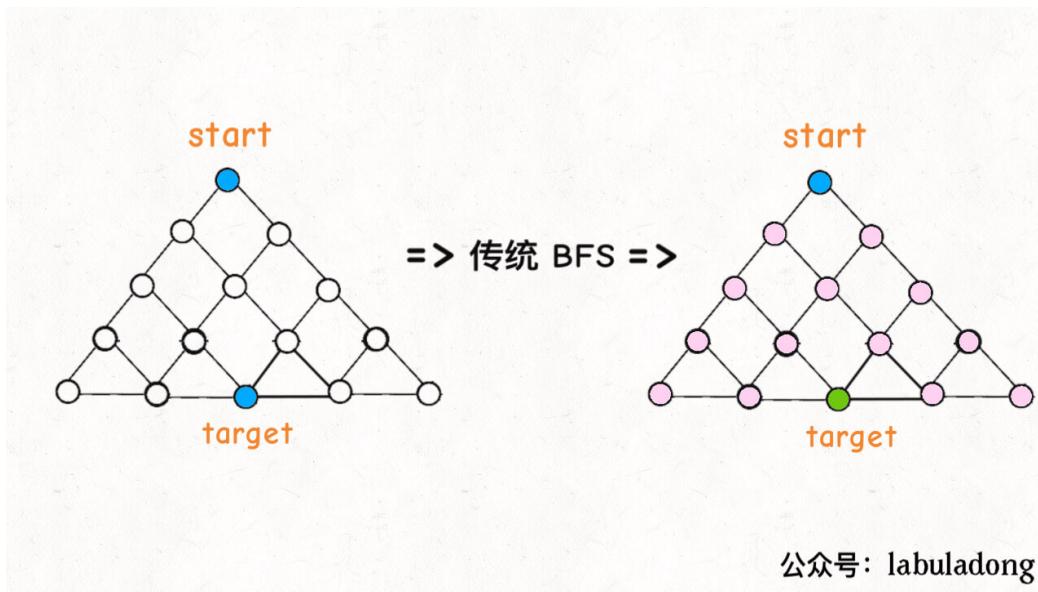
至此，我们就解决这道题目了。有一个比较小的优化：可以不需要 `dead` 这个哈希集合，可以直接将这些元素初始化到 `visited` 集合中，效果是一样的，可能更加优雅一些。

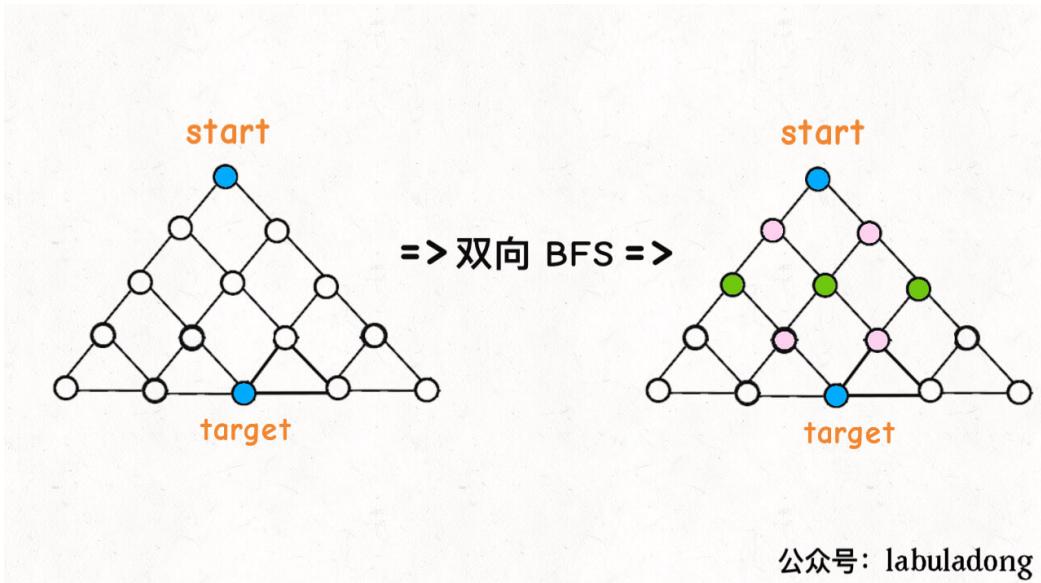
四、双向 BFS 优化

你以为到这里 BFS 算法就结束了？恰恰相反。BFS 算法还有一种稍微高级一点的优化思路：**双向 BFS**，可以进一步提高算法的效率。

篇幅所限，这里就提一下区别：传统的 **BFS** 框架就是从起点开始向四周扩散，遇到终点时停止；而**双向 BFS** 则是从起点和终点同时开始扩散，当两边有交集的时候停止。

为什么这样能够提升效率呢？其实从 Big O 表示法分析算法复杂度的话，它俩的最坏复杂度都是 $O(N)$ ，但是实际上双向 BFS 确实会快一些，我给你画两张图看一眼就明白了：





图示中的树形结构，如果终点在最底部，按照传统 BFS 算法的策略，会把整棵树的节点都搜索一遍，最后找到 target；而双向 BFS 其实只遍历了半棵树就出现了交集，也就是找到了最短距离。从这个例子可以直观地感受到，双向 BFS 是要比传统 BFS 高效的。

不过，双向 BFS 也有局限，因为你必须知道终点在哪里。比如我们刚才讨论的二叉树最小高度的问题，你一开始根本就不知道终点在哪里，也就无法使用双向 BFS；但是第二个密码锁的问题，是可以使用双向 BFS 算法来提高效率的，代码稍加修改即可：

学习算法和刷题的框架思维

```
int openLock(String[] deadends, String target) {
    Set<String> deads = new HashSet<>();
    for (String s : deadends) deads.add(s);
    // 用集合不用队列，可以快速判断元素是否存在
    Set<String> q1 = new HashSet<>();
    Set<String> q2 = new HashSet<>();
    Set<String> visited = new HashSet<>();

    int step = 0;
    q1.add("0000");
    q2.add(target);

    while (!q1.isEmpty() && !q2.isEmpty()) {
        // 哈希集合在遍历的过程中不能修改，用 temp 存储扩散结果
        Set<String> temp = new HashSet<>();

        /* 将 q1 中的所有节点向周围扩散 */
        for (String cur : q1) {
            /* 判断是否到达终点 */
            if (deads.contains(cur))
                continue;
            if (q2.contains(cur))
                return step;
            visited.add(cur);

            /* 将一个节点的未遍历相邻节点加入集合 */
            for (int j = 0; j < 4; j++) {
                String up = plusOne(cur, j);
                if (!visited.contains(up))
                    temp.add(up);
                String down = minusOne(cur, j);
                if (!visited.contains(down))
                    temp.add(down);
            }
        }
        /* 在这里增加步数 */
        step++;
        // temp 相当于 q1
        // 这里交换 q1 q2，下一轮 while 就是扩散 q2
        q1 = q2;
        q2 = temp;
```

学习算法和刷题的框架思维

```
    }
    return -1;
}
```

双向 BFS 还是遵循 BFS 算法框架的，只是不再使用队列，而是使用 **HashSet** 方便快速判断两个集合是否有交集。

另外的一个技巧点就是 **while** 循环的最后交换 **q1** 和 **q2** 的内容，所以只要默认扩散 **q1** 就相当于轮流扩散 **q1** 和 **q2**。

其实双向 BFS 还有一个优化，就是在 **while** 循环开始时做一个判断：

```
// ...
while (!q1.isEmpty() && !q2.isEmpty()) {
    if (q1.size() > q2.size()) {
        // 交换 q1 和 q2
        temp = q1;
        q1 = q2;
        q2 = temp;
    }
    // ...
```

为什么这是一个优化呢？

因为按照 BFS 的逻辑，队列（集合）中的元素越多，扩散之后新的队列（集合）中的元素就越多；在双向 BFS 算法中，如果我们每次都选择一个较小的集合进行扩散，那么占用的空间增长速度就会慢一些，效率就会高一些。

不过话说回来，无论传统 **BFS** 还是双向 **BFS**，无论做不做优化，从 **Big O** 衡量标准来看，时间复杂度都是一样的，只能说双向 BFS 是一种 trick，算法运行的速度会相对快一点，掌握

学习算法和刷题的框架思维

不掌握其实都无所谓。最关键的是把 BFS 通用框架记下来，反正所有 BFS 算法都可以用它套出解法。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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 $\log N$ 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。

二分查找并不简单，Knuth 大佬（发明 KMP 算法的那位）都说二分查找：**思路很简单，细节是魔鬼**。很多人喜欢拿整型溢出的 bug 说事儿，但是二分查找真正的坑根本就不是那个细节问题，而是在于到底要给 `mid` 加一还是减一，`while` 里到底用 `<=` 还是 `<`。

你要是没有正确理解这些细节，写二分肯定就是玄学编程，有没有 bug 只能靠菩萨保佑。**我特意写了一首诗来歌颂该算法，概括本文的主要内容，建议保存：**

学习算法和刷题的框架思维

二分搜索升天词

作者：labuladong

二分搜索不好记，左右边界让人迷。
小于等于变小于，mid 加一又减一。
就算这样还没完，return 应否再 -1 ?
信心满满刷力扣，AC 比率二十一。
我本将心向明月，奈何明月照沟渠！
问君能有几多愁？恰似深情喂了狗。

labuladong从天降，一同手撕算法题。
赠君一法写二分，不用拜佛与念经。
管他左侧还右侧，搜索区间定乾坤。

搜索一个元素时，搜索区间两端闭。
while 条件带等号，否则需要打补丁。
if 相等就返回，其他的事甭操心。
mid 必须加减一，因为区间两端闭。
while 结束就凉了，凄凄惨惨返 -1。

搜索左右边界时，搜索区间要阐明。
左闭右开最常见，其余逻辑便自明：
while要用小于号，这样才能不漏掉。
if 相等别返回，利用 mid 锁边界。
mid 加一或减一？要看区间开或闭。
while 结束不算完，因为你还没返回。
索引可能出边界，if 检查保平安。

左闭右开最常见，难道常见就合理？
labuladong不信邪，偏要改成两端闭。
搜索区间记于心，或开或闭有何异？
二分搜索三变体，逻辑统一容易记。
一套框架改两行，胜过千言和万语。

此等神人何处寻？全靠缘分不可期！
labuladong公众号，开启算法新天地。
关注标星加分享，“下次一定”不可取。

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 = left + (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` 时需要防止溢出，代码中 `left + (right - left) / 2` 就和 `(left + right) / 2` 的结果相同，但是有效防止了 `left` 和 `right` 太大直接相加导致溢出。

一、寻找一个数（基本的二分搜索）

学习算法和刷题的框架思维

这个场景是最简单的，肯定也是大家最熟悉的，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 = left + (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;
```

学习算法和刷题的框架思维

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，下一步应该去搜索哪里呢？

当然是去搜索 `[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` 为 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

以下是最常见的代码形式，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` 中是 `<` 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。

PS：这里先要说一个搜索左右边界和上面这个算法的一个区别，也是很多读者问的：刚才的 `right` 不是 `nums.length - 1` 吗，为啥这里非要写成 `nums.length` 使得「搜索区间」变成左闭右开呢？

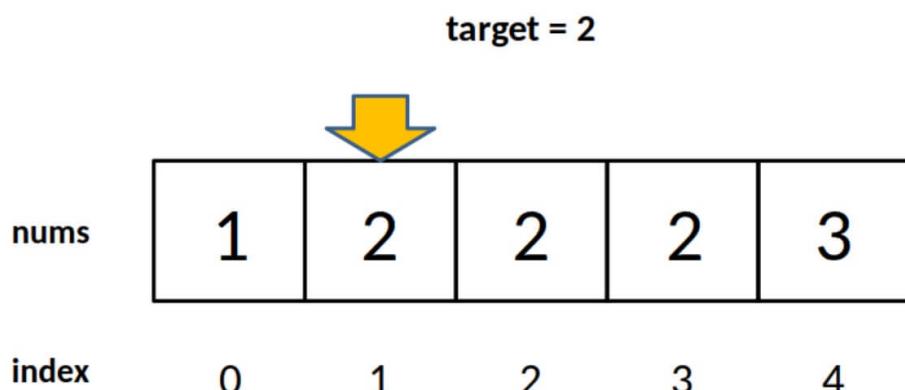
因为对于搜索左右侧边界的二分查找，这种写法比较普遍，我就拿这种写法举例了，保证你以后遇到这类代码可以理解。你非要用两端都闭的写法反而更简单，我会在后面写相关的代

学习算法和刷题的框架思维

码，把三种二分搜索都用一种两端都闭的写法统一起来，你耐心往后看就行了。

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：



公众号: labuladong

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 = [2,3,5,7]`，`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;
```

3、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`。

6、能不能想办法把 `right` 变成 `nums.length - 1`，也就是继续使用两边都闭的「搜索区间」？这样就可以和第一种二分搜索在某种程度上统一起来了。

学习算法和刷题的框架思维

答：当然可以，只要你明白了「搜索区间」这个概念，就能有效避免漏掉元素，随便你怎么改都行。下面我们严格根据逻辑来修改：

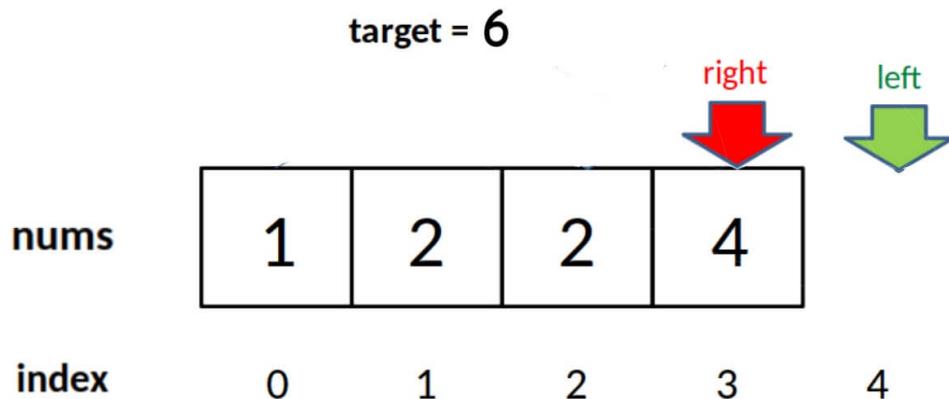
因为你非要让搜索区间两端都闭，所以 `right` 应该初始化为 `nums.length - 1`，`while` 的终止条件应该是 `left == right + 1`，也就是其中应该用 `<=`：

```
int left_bound(int[] nums, int target) {
    // 搜索区间为 [left, right]
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        // if else ...
    }
}
```

因为搜索区间是两端都闭的，且现在是搜索左侧边界，所以 `left` 和 `right` 的更新逻辑如下：

```
if (nums[mid] < target) {
    // 搜索区间变为 [mid+1, right]
    left = mid + 1;
} else if (nums[mid] > target) {
    // 搜索区间变为 [left, mid-1]
    right = mid - 1;
} else if (nums[mid] == target) {
    // 收缩右侧边界
    right = mid - 1;
}
```

由于 `while` 的退出条件是 `left == right + 1`，所以当 `target` 比 `nums` 中所有元素都大时，会存在以下情况使得索引起越界：



公众号: labuladong

因此，最后返回结果的代码应该检查越界情况：

```
if (left >= nums.length || nums[left] != target)
    return -1;
return left;
```

至此，整个算法就写完了，完整代码如下：

```
int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    // 搜索区间为 [left, right]
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            // 搜索区间变为 [mid+1, right]
            left = mid + 1;
        } else if (nums[mid] > target) {
            // 搜索区间变为 [left, mid-1]
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 收缩右侧边界
            right = mid - 1;
        }
    }
    // 检查出界情况
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}
```

这样就和第一种二分搜索算法统一了，都是两端都闭的「搜索区间」，而且最后返回的也是 `left` 变量的值。只要把住二分搜索的逻辑，两种形式大家看自己喜欢哪种记哪种吧。

三、寻找右侧边界的二分查找

类似寻找左侧边界的算法，这里也会提供两种写法，还是先写常见的左闭右开的写法，只有两处和搜索左侧边界不同，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。

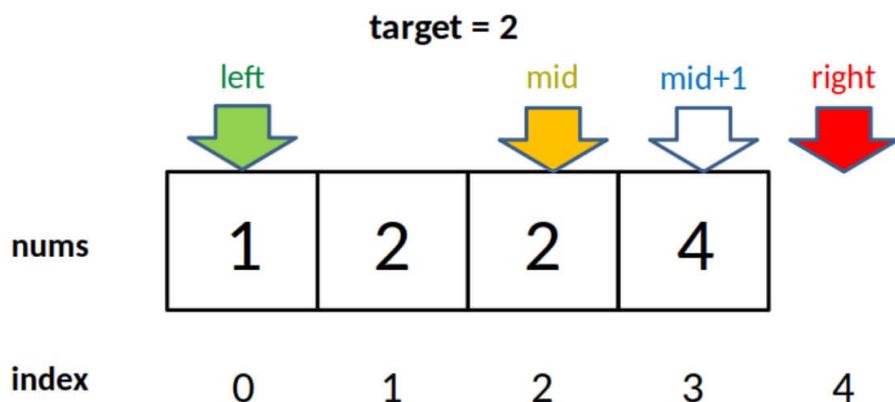
2、为什么最后返回 `left - 1` 而不像左侧边界的函数，返回 `left`？而且我觉得这里既然是搜索右侧边界，应该返回 `right` 才对。

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。

学习算法和刷题的框架思维

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
```



公众号: labuladong

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。

3、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：

学习算法和刷题的框架思维

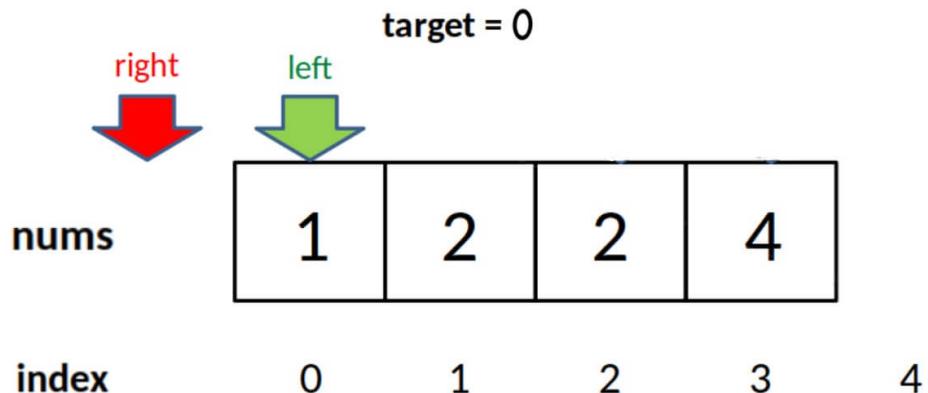
```
while (left < right) {
    // ...
}
if (left == 0) return -1;
return nums[left-1] == target ? (left-1) : -1;
```

4、是否也可以把这个算法的「搜索区间」也统一成两端都闭的形式呢？这样这三个写法就完全统一了，以后就可以闭着眼睛写出来了。

答：当然可以，类似搜索左侧边界的统一写法，其实只要改两个地方就行了：

```
int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 这里改成收缩左侧边界即可
            left = mid + 1;
        }
    }
    // 这里改为检查 right 越界的情况，见下图
    if (right < 0 || nums[right] != target)
        return -1;
    return right;
}
```

当 `target` 比所有元素都小时，`right` 会被减到 -1，所以需要在最后防止越界：



公众号: labuladong

至此，搜索右侧边界的二分查找的两种写法也完成了，其实将「搜索区间」统一成两端都闭反而更容易记忆，你说是吧？

四、逻辑统一

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

对于寻找左右边界的二分搜索，常见的手法是使用左闭右开的
「搜索区间」，我们还根据逻辑将「搜索区间」全都统一成了
两端都闭，便于记忆，只要修改两处即可变化出三种写法：

学习算法和刷题的框架思维

```
int binary_search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while(left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 直接返回
            return mid;
        }
    }
    // 直接返回
    return -1;
}

int left_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) {
            left = mid + 1;
        } else if (nums[mid] > target) {
            right = mid - 1;
        } else if (nums[mid] == target) {
            // 别返回, 锁定左侧边界
            right = mid - 1;
        }
    }
    // 最后要检查 left 越界的情况
    if (left >= nums.length || nums[left] != target)
        return -1;
    return left;
}

int right_bound(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
```

学习算法和刷题的框架思维

```
if (nums[mid] < target) {
    left = mid + 1;
} else if (nums[mid] > target) {
    right = mid - 1;
} else if (nums[mid] == target) {
    // 别返回，锁定右侧边界
    left = mid + 1;
}
}

// 最后要检查 right 越界的情况
if (right < 0 || nums[right] != target)
    return -1;
return right;
}
```

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、如需定义左闭右开的「搜索区间」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。
- 4、如果将「搜索区间」全都统一成两端都闭，好记，只要稍改 `nums[mid] == target` 条件处的代码和返回的逻辑即可，推荐拿小本本记下，作为二分搜索模板。

学习算法和刷题的框架思维

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或
[online book](#) 获取最新信息，后台回复「进群」可进刷题群，
labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

滑动窗口算法框架

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

[76.最小覆盖子串](#)

[567.字符串的排列](#)

[438.找到字符串中所有字母异位词](#)

[3.无重复字符的最长子串](#)

鉴于前文 [二分搜索框架详解](#) 的那首《二分搜索升天词》很受好评，并在民间广为流传，成为安睡助眠的一剂良方，今天在滑动窗口算法框架中，我再次编写一首小诗来歌颂滑动窗口算法的伟大：

滑动窗口防滑记
作者：labuladong

链表子串数组题，用双指针别犹豫。
双指针家三兄弟，各个都是万人迷。

快慢指针最神奇，链表操作无压力。
归并排序找中点，链表成环搞判定。

左右指针最常见，左右两端相向行。
反转数组要靠它，二分搜索是弟弟。

滑动窗口老猛男，子串问题全靠它。
左右指针滑窗口，一前一后齐头进。
自称十年老司机，怎料农村道路滑。
一不小心滑到了，鼻青脸肿少颗牙。
算法思想很简单，出了bug想升天。

labuladong稳若狗，一套框架不翻车。
一路漂移带闪电，算法变成默写题。
此车还有副驾驶，文末别忘瞧一瞧。

关于双指针的快慢指针和左右指针的用法，可以参见前文 [双指针技巧汇总](#)，本文就解决一类最难掌握的双指针技巧：滑动窗口技巧。总结出一套框架，可以保你闭着眼睛都能写出正确的解法。

学习算法和刷题的框架思维

说起滑动窗口算法，很多读者都会头疼。这个算法技巧的思路非常简单，就是维护一个窗口，不断滑动，然后更新答案么。LeetCode 上有起码 10 道运用滑动窗口算法的题目，难度都是中等和困难。该算法的大致逻辑如下：

```
int left = 0, right = 0;

while (right < s.size()) {
    // 增大窗口
    window.add(s[right]);
    right++;

    while (window needs shrink) {
        // 缩小窗口
        window.remove(s[left]);
        left++;
    }
}
```

这个算法技巧的时间复杂度是 $O(N)$ ，比字符串暴力算法要高效得多。

其实困扰大家的，不是算法的思路，而是各种细节问题。比如说如何向窗口中添加新元素，如何缩小窗口，在窗口滑动的那个阶段更新结果。即便你明白了这些细节，也容易出 bug，找 bug 还不知道怎么找，真的挺让人心烦的。

所以今天我就写一套滑动窗口算法的代码框架，我连再哪里做输出 debug 都给你写好了，以后遇到相关的问题，你就默写出来如下框架然后改三个地方就行，还不会出 bug：

由于格式原因，本文只能在 labuladong 公众号查看，关注后可直接搜索本站内容：

学习算法和刷题的框架思维



团灭 LeetCode 打家劫舍问题



微信搜一搜

Q labuladong

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

[198.打家劫舍](#)

[213.打家劫舍II](#)

[337.打家劫舍III](#)

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。

由于格式原因，本文只能在 labuladong 公众号查看，关注后可直接搜索本站内容：

学习算法和刷题的框架思维



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：



动态规划答疑篇



微信搜一搜

Q labuladong

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。
-

由于格式原因，本文只能在 labuladong 公众号查看，关注后可直接搜索本站内容：



动态规划之背包问题



微信搜一搜

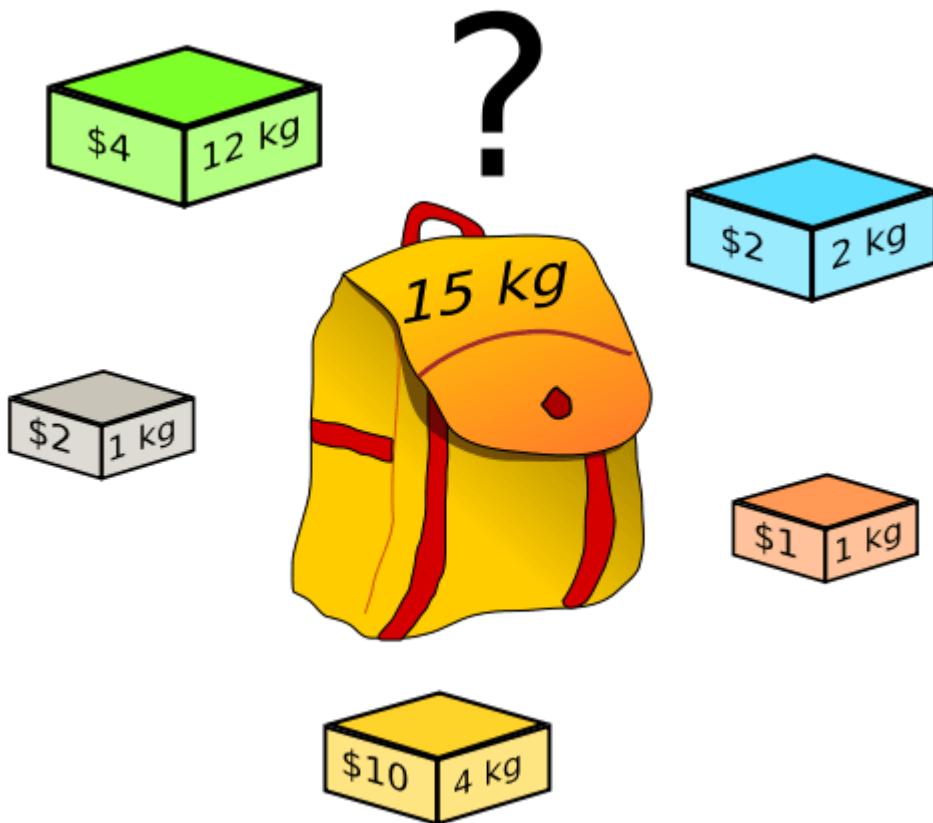
Q labuladong

后台天天有人问背包问题，这个问题其实不难啊，如果我们号动态规划系列的十几篇文章你都看过，借助框架，遇到背包问题可以说是手到擒来好吧。无非就是状态 + 选择，也没啥特别之处嘛。

今天就来说一下背包问题吧，就讨论最常说的 0-1 背包问题。

描述：

给你一个可装载重量为 w 的背包和 N 个物品，每个物品有重量和价值两个属性。其中第 i 个物品的重量为 $wt[i]$ ，价值为 $val[i]$ ，现在让你用这个背包装物品，最多能装的价值是多少？



举个简单的例子，输入如下：

```
N = 3, W = 4  
wt = [2, 1, 3]  
val = [4, 2, 3]
```

算法返回 6，选择前两件物品装进背包，总重量 3 小于 w ，可以获得最大价值 6。

题目就是这么简单，一个典型的动态规划问题。这个题目中的物品不可以分割，要么装进包里，要么不装，不能说切成两块装一半。这就是 0-1 背包这个名词的来历。

解决这个问题没有什么排序之类巧妙的方法，只能穷举所有可能，根据我们「动态规划详解」中的套路，直接走流程就行了。

学习算法和刷题的框架思维

由于格式原因，本文只能在 labuladong 公众号查看，关注后
可直接搜索本站内容：



背包问题之零钱兑换

零钱兑换 2 是另一种典型背包问题的变体，我们前文已经讲了 [经典动态规划：0-1 背包问题](#) 和 [背包问题变体：相等子集分割](#)。

希望你已经看过前两篇文章，看过了动态规划和背包问题的套路，这篇继续按照背包问题的套路，列举一个背包问题的变形。

本文聊的是 LeetCode 第 518 题 Coin Change 2，题目如下：

518. 零钱兑换 II

难度 中等 122 收藏 分享 切换为英文 关注 反馈

给定不同面额的硬币和一个总金额。写出函数来计算可以凑成总金额的硬币组合数。假设每一种面额的硬币有无限个。

示例 1:

```
输入: amount = 5, coins = [1, 2, 5]
输出: 4
解释: 有四种方式可以凑成总金额:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2:

```
输入: amount = 3, coins = [2]
输出: 0
解释: 只用面额2的硬币不能凑成总金额3。
```

```
int change(int amount, int[] coins);
```

PS：至于 Coin Change 1，在我们前文 [动态规划套路详解](#) 写过。

我们可以把这个问题转化为背包问题的描述形式：

学习算法和刷题的框架思维

有一个背包，最大容量为 `amount`，有一系列物品 `coins`，每个物品的重量为 `coins[i]`，**每个物品的数量无限**。请问有多少种方法，能够把背包恰好装满？

这个问题和我们前面讲过的两个背包问题，有一个最大的区别就是，每个物品的数量是无限的，这也就是传说中的「**完全背包问题**」，没啥高大上的，无非就是状态转移方程有一点变化而已。

下面就以背包问题的描述形式，继续按照流程来分析。

解题思路

第一步要明确两点，「状态」和「选择」。

状态有两个，就是「背包的容量」和「可选择的物品」，选择就是「装进背包」或者「不装进背包」嘛，背包问题的套路都是这样。

明白了状态和选择，动态规划问题基本上就解决了，只要往这个框架套就完事儿了：

```
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][...] = 计算(选择1, 选择2...)
```

第二步要明确 `dp` 数组的定义。

首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 `dp` 数组。

`dp[i][j]` 的定义如下：

学习算法和刷题的框架思维

若只使用前 i 个物品，当背包容量为 j 时，有 $dp[i][j]$ 种方法可以装满背包。

换句话说，翻译回我们题目的意思就是：

若只使用 $coins$ 中的前 i 个硬币的面值，若想凑出金额 j ，有 $dp[i][j]$ 种凑法。

经过以上的定义，可以得到：

base case 为 $dp[0][..] = 0$, $dp[..][0] = 1$ 。因为如果不使用任何硬币面值，就无法凑出任何金额；如果凑出的目标金额为 0，那么“无为而治”就是唯一的一种凑法。

我们最终想得到的答案就是 $dp[N][amount]$ ，其中 N 为 $coins$ 数组的大小。

大致的伪码思路如下：

```
int dp[N+1][amount+1]
dp[0][..] = 0
dp[..][0] = 1

for i in [1..N]:
    for j in [1..amount]:
        把物品 i 装进背包,
        不把物品 i 装进背包
return dp[N][amount]
```

第三步，根据「选择」，思考状态转移的逻辑。

注意，我们这个问题的特殊点在于物品的数量是无限的，所以这里和之前写的背包问题文章有所不同。

如果你不把这第 i 个物品装入背包，也就是说你不使用 $\text{coins}[i]$ 这个面值的硬币，那么凑出面额 j 的方法数 $\text{dp}[i][j]$ 应该等于 $\text{dp}[i-1][j]$ ，继承之前的结果。

如果你把这第 i 个物品装入了背包，也就是说你使用 $\text{coins}[i]$ 这个面值的硬币，那么 $\text{dp}[i][j]$ 应该等于 $\text{dp}[i][j - \text{coins}[i-1]]$ 。

首先由于 i 是从 1 开始的，所以 coins 的索引是 $i-1$ 时表示第 i 个硬币的面值。

$\text{dp}[i][j - \text{coins}[i-1]]$ 也不难理解，如果你决定使用这个面值的硬币，那么就应该关注如何凑出金额 $j - \text{coins}[i-1]$ 。

比如说，你想用面值为 2 的硬币凑出金额 5，那么如果你知道了凑出金额 3 的方法，再加上一枚面额为 2 的硬币，不就可以凑出 5 了嘛。

综上就是两种选择，而我们想求的 $\text{dp}[i][j]$ 是「共有多少种凑法」，所以 $\text{dp}[i][j]$ 的值应该是以上两种选择的结果之和：

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= amount; j++) {
        if (j - coins[i-1] >= 0)
            dp[i][j] = dp[i - 1][j]
                        + dp[i][j - coins[i-1]];
}
return dp[N][W]
```

最后一步，把伪码翻译成代码，处理一些边界情况。

我用 Java 写的代码，把上面的思路完全翻译了一遍，并且处理了一些边界问题：

学习算法和刷题的框架思维

```
int change(int amount, int[] coins) {  
    int n = coins.length;  
    int[][] dp = amount int[n + 1][amount + 1];  
    // base case  
    for (int i = 0; i <= n; i++)  
        dp[i][0] = 1;  
  
    for (int i = 1; i <= n; i++) {  
        for (int j = 1; j <= amount; j++)  
            if (j - coins[i-1] >= 0)  
                dp[i][j] = dp[i - 1][j]  
                    + dp[i][j - coins[i-1]];  
            else  
                dp[i][j] = dp[i - 1][j];  
    }  
    return dp[n][amount];  
}
```

而且，我们通过观察可以发现，`dp` 数组的转移只和 `dp[i]` [...] 和 `dp[i-1][...]` 有关，所以可以压缩状态，进一步降低算法的空间复杂度：

```
int change(int amount, int[] coins) {  
    int n = coins.length;  
    int[] dp = new int[amount + 1];  
    dp[0] = 1; // base case  
    for (int i = 0; i < n; i++)  
        for (int j = 1; j <= amount; j++)  
            if (j - coins[i] >= 0)  
                dp[j] = dp[j] + dp[j-coins[i]];  
  
    return dp[amount];  
}
```

这个解法和之前的思路完全相同，将二维 `dp` 数组压缩为一维，时间复杂度 $O(N*amount)$ ，空间复杂度 $O(amount)$ 。

至此，这道零钱兑换问题也通过背包问题的框架解决了。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或
[online book](#) 获取最新信息，后台回复「进群」可进刷题群，
labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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：

学习算法和刷题的框架思维

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。

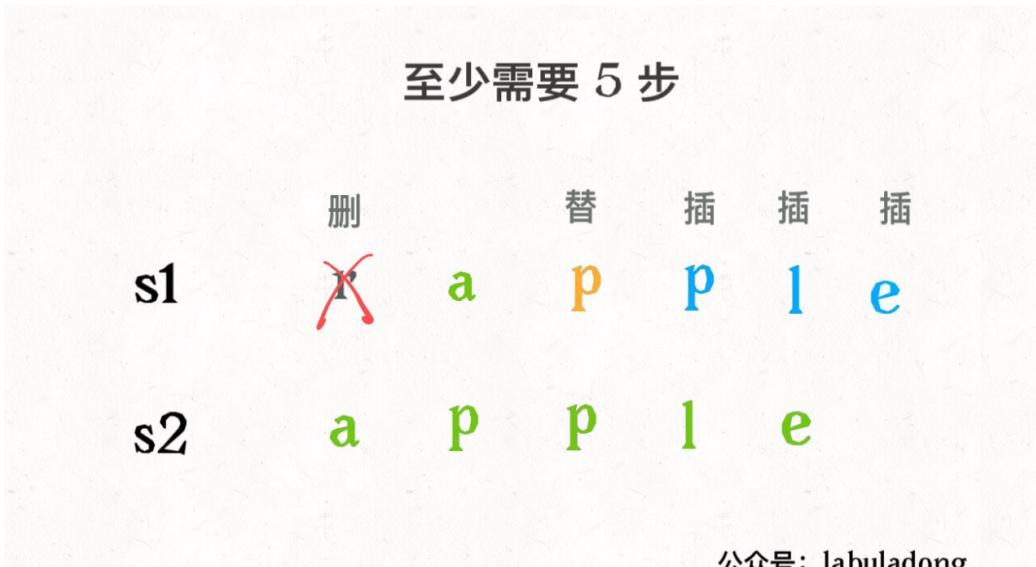
一、思路

编辑距离问题就是给我们两个字符串 s_1 和 s_2 ，只能用三种操作，让我们把 s_1 变成 s_2 ，求最少的操作数。需要明确的是，不管是把 s_1 变成 s_2 还是反过来，结果都是一样的，所以后文就以 s_1 变成 s_2 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。

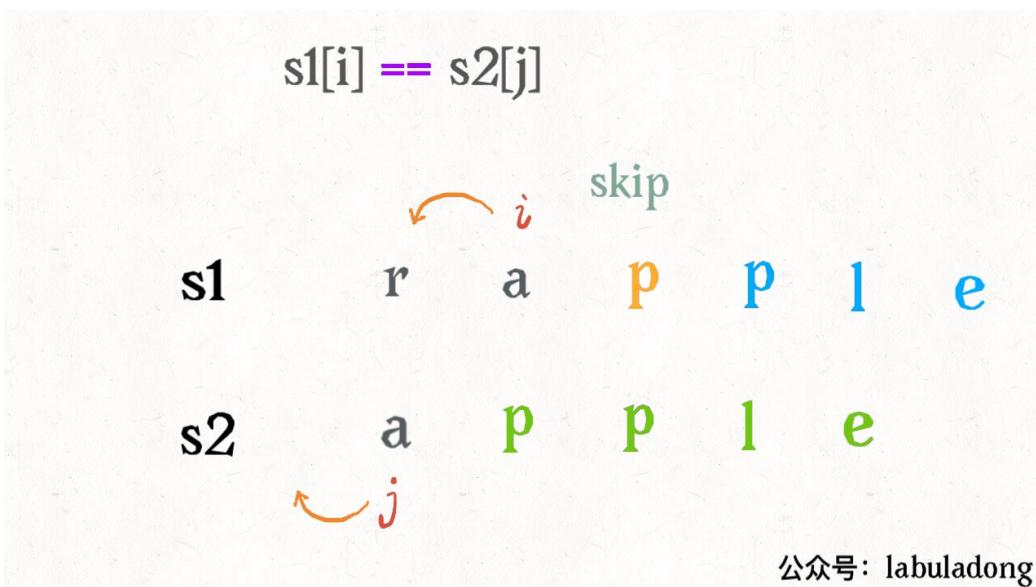
设两个字符串分别为 "rad" 和 "apple"，为了把 s_1 变成 s_2 ，算法会这样进行：

【PDF格式无法显示GIF文件 editDistance/edit.gif，可移步公众号查看】



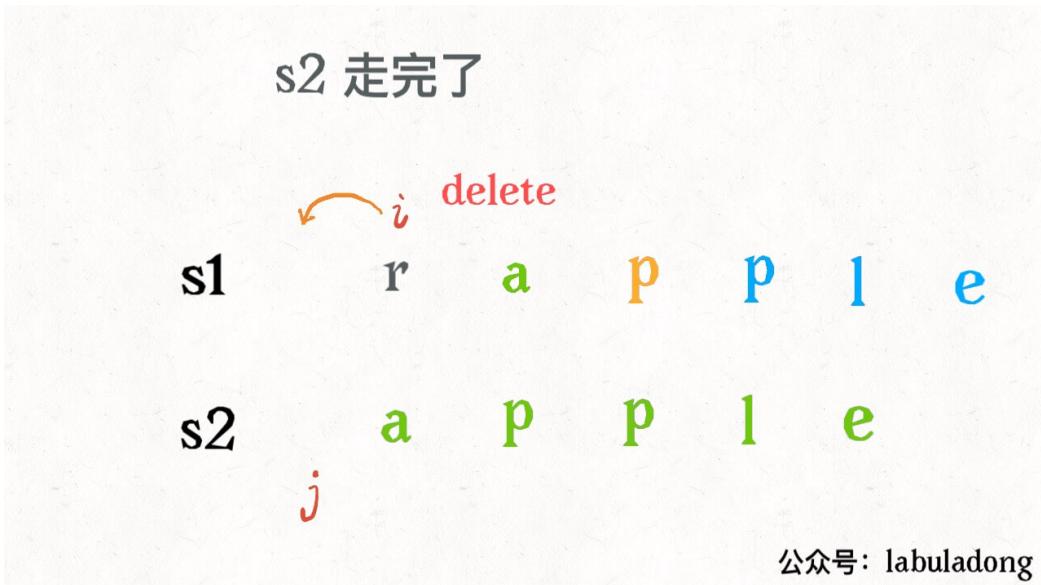
请记住这个 GIF 过程，这样就能算出编辑距离。关键在于如何做出正确的操作，稍后会讲。

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）。比如这种情况：



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：

学习算法和刷题的框架思维

```
if s1[i] == s2[j]:  
    哪都别做 (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：

```
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) # 哪都不做  
        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。

学习算法和刷题的框架思维

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：

```
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 $s1[i] \neq s2[j]$ ，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格式无法显示GIF文件 editDistance/insert.gif，可移步公众号查看】

```
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
```

学习算法和刷题的框架思维

【PDF格式无法显示GIF文件 editDistance/delete.gif, 可移步公众号查看】

```
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格式无法显示GIF文件 editDistance/replace.gif, 可移步公众号查看】

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

学习算法和刷题的框架思维

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：

s1 \ s2	" "	a	p	p	l	e
" "	0	1	2	3	4	5
r	1	1	2	3	4	5
a	2	1	2	3	4	5
d	3	2	2	3	4	5

学习算法和刷题的框架思维

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
```

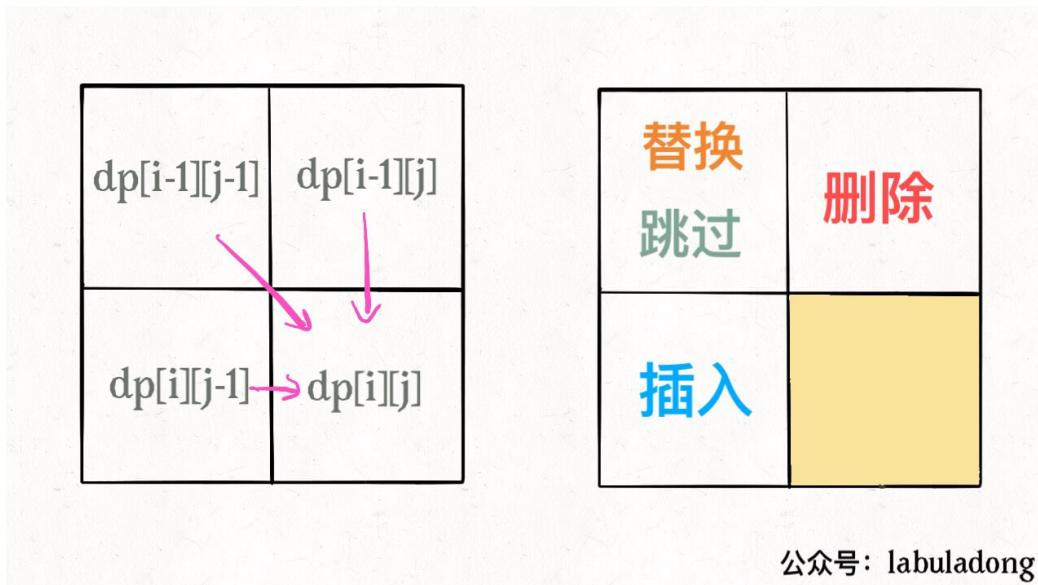
dp 函数的 base case 是 `i, j` 等于 -1，而数组索引至少是 0，所以 dp 数组会偏移一位。

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 = 0; i <= m; i++)  
        dp[i][0] = i;  
    for (int j = 0; 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：



公众号: labuladong

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：

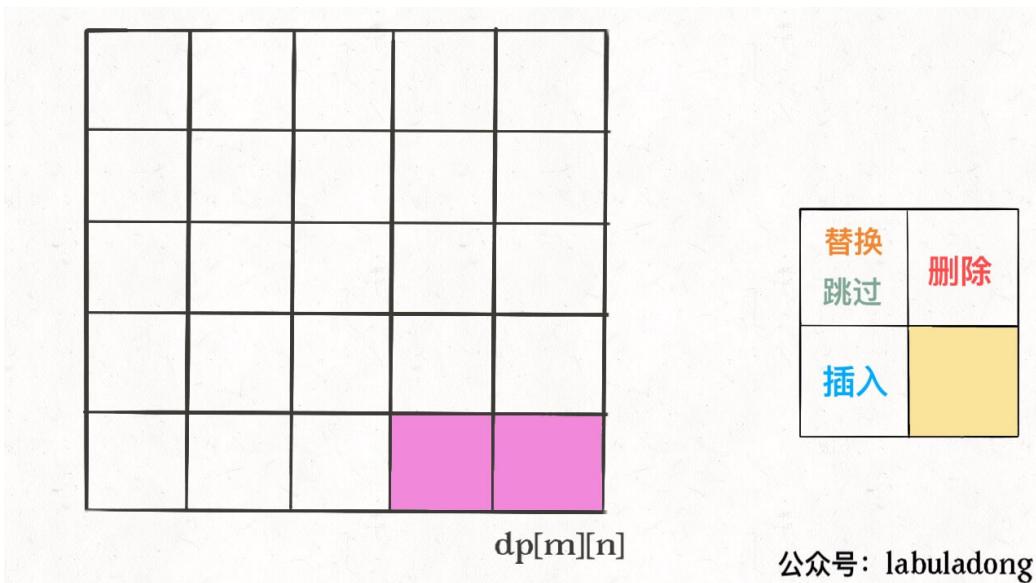
```
// int[][] dp;
Node[][] dp;

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
}
```

学习算法和刷题的框架思维

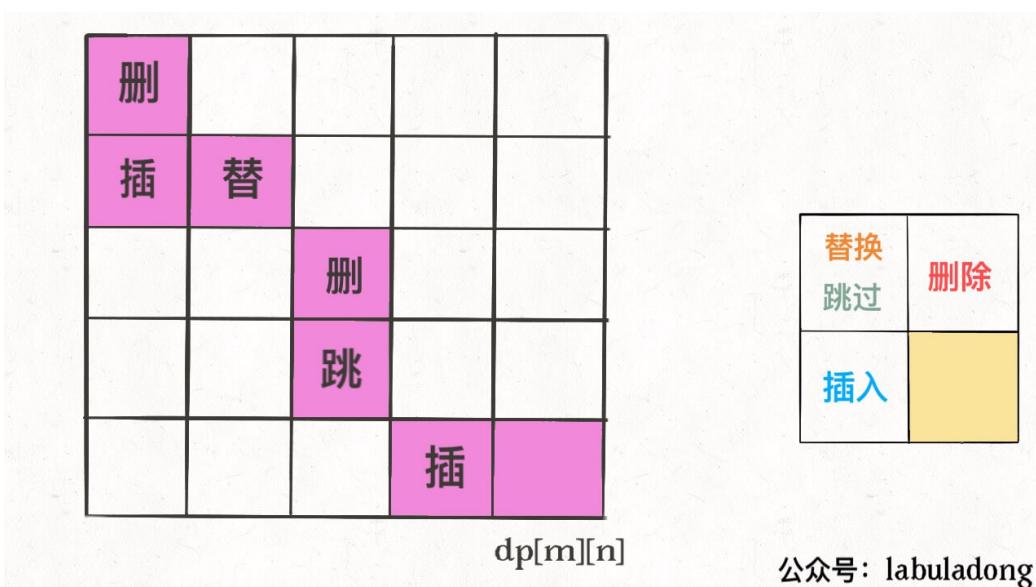
`val` 属性就是之前的 `dp` 数组的数值，`choice` 属性代表操作。在做最优选择时，顺便把操作记录下来，然后就从结果反推具体操作。

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：



公众号: labuladong

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。



公众号: labuladong

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或
[online book](#) 获取最新信息，后台回复「进群」可进刷题群，
labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

经典动态规划问题：高楼扔鸡蛋 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 $1 \leq i \leq N$ 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；

学习算法和刷题的框架思维

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
```

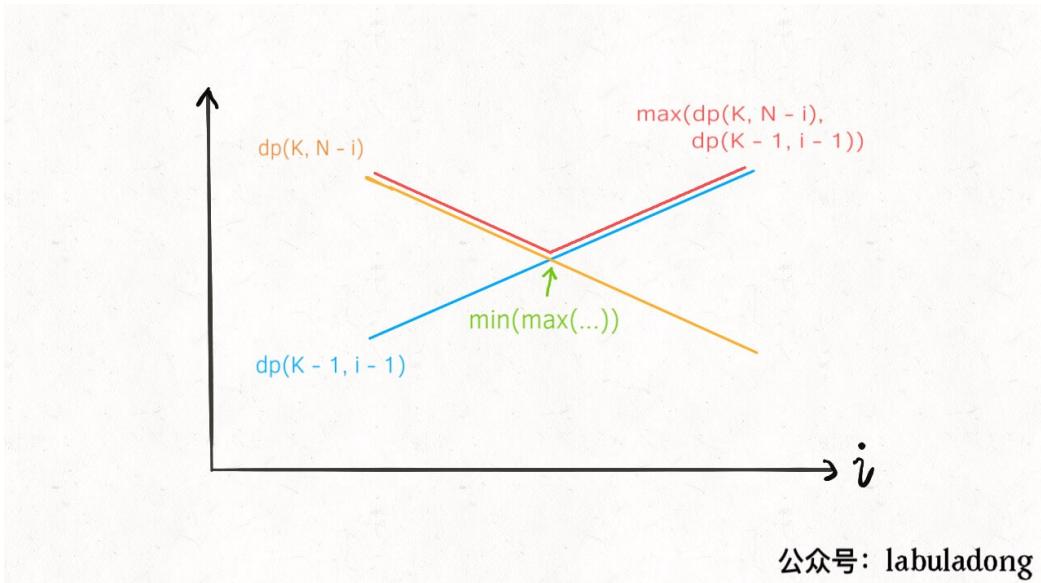
这个 for 循环就是下面这个状态转移方程的具体代码实现：

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

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：



公众号: labuladong

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);  
}
```

学习算法和刷题的框架思维

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：

学习算法和刷题的框架思维

```
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。**

学习算法和刷题的框架思维

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 $O(\log N)$ 。

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 $O(K \cdot N \cdot \log N)$ ，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。

学习算法和刷题的框架思维

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？

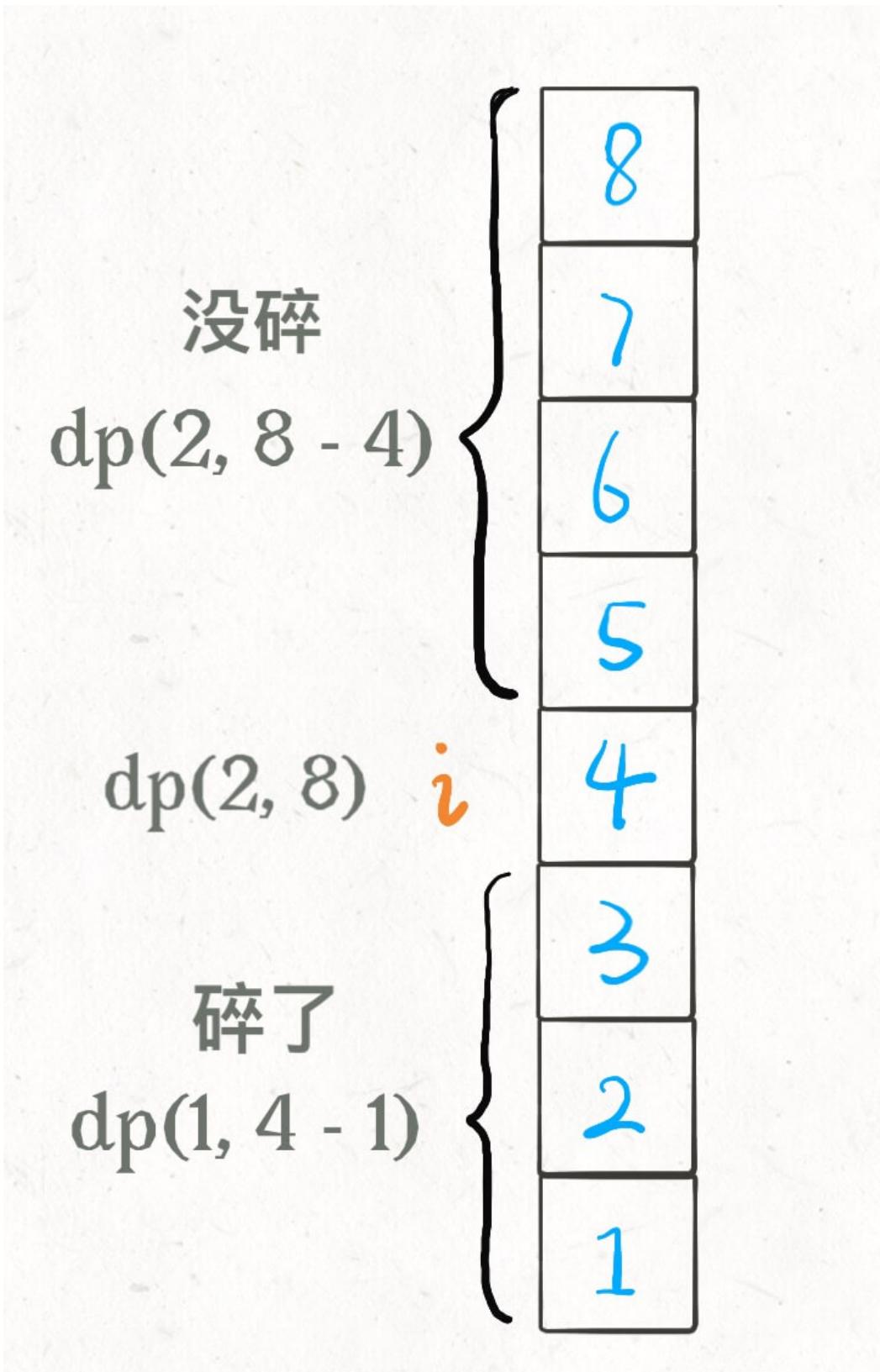
我们最终要求的其实是扔鸡蛋次数 m ，但是这时候 m 在状态之中而不是 `dp` 数组的结果，可以这样处理：

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

学习算法和刷题的框架思维

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：



公众号：labuladong

学习算法和刷题的框架思维

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）。

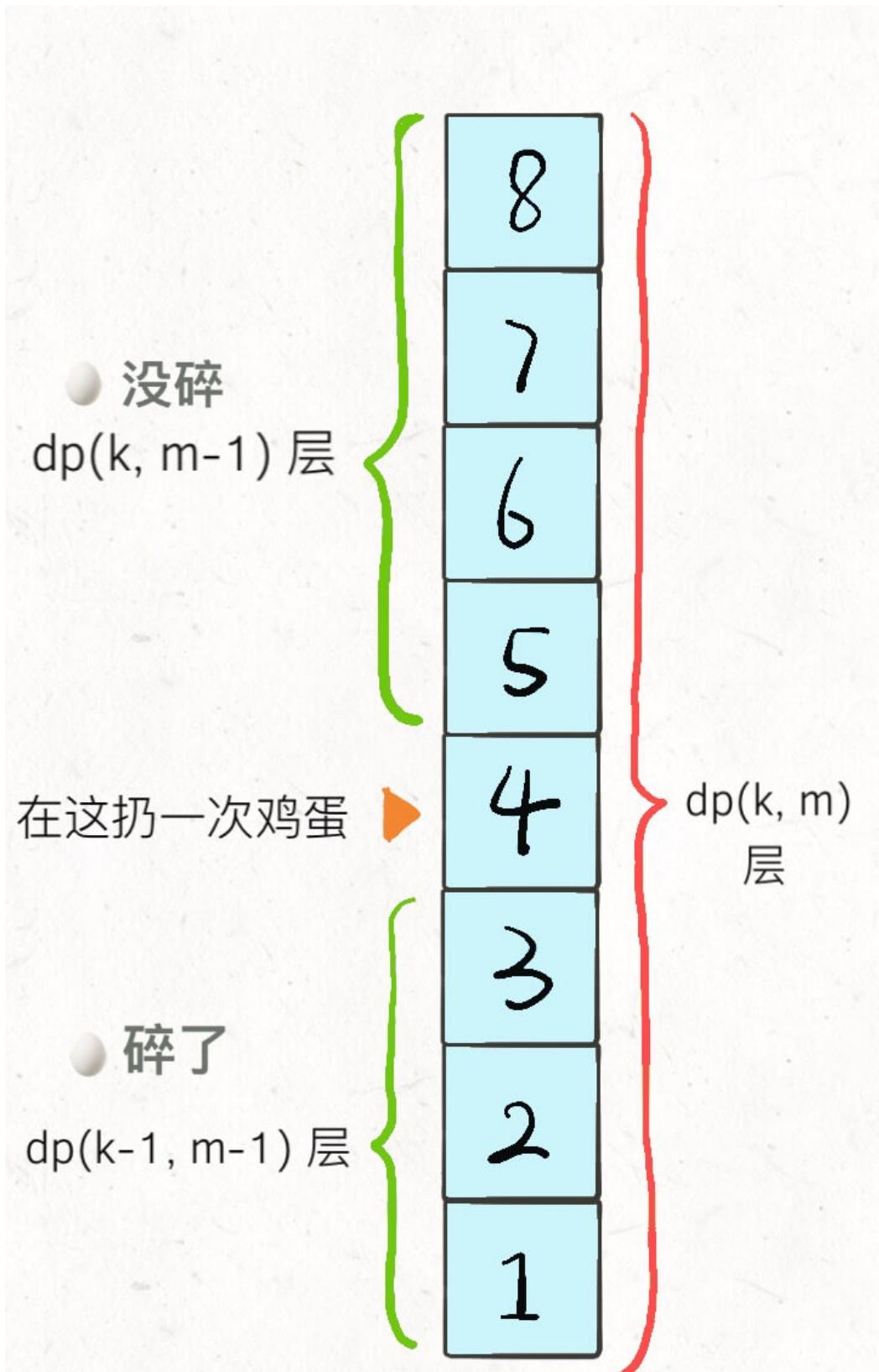
根据这个特点，可以写出下面的状态转移方程：

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

$dp[k][m - 1]$ 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。



学习算法和刷题的框架思维

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 $O(K \log N)$ ，我们可以设

`g(k, m) =`

算了算了，打住吧。我觉得我们能够写出 $O(K * N * \log N)$ 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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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成最终的答案。

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：

学习算法和刷题的框架思维

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

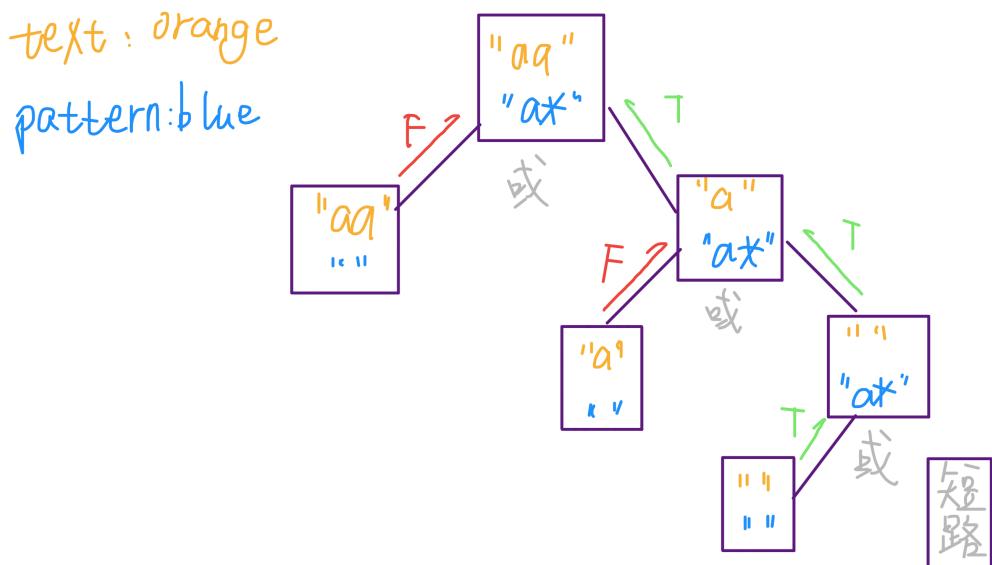
星号通配符可以让前一个字符重复任意次数，包括零次。那到底是重复几次呢？这似乎有点困难，不过不要着急，我们起码可以把框架的搭建再进一步：

```
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。

学习算法和刷题的框架思维

```
# 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？

学习算法和刷题的框架思维

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 $f(n) \rightarrow \#1 \rightarrow \#1$ ，二是 $f(n) \rightarrow \#2$ 。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 $dp(i, j) \rightarrow \#3 \rightarrow \#3$ ，二是 $dp(i, j) \rightarrow \#1 \rightarrow \#2 \rightarrow \#2$ 。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

学习算法和刷题的框架思维

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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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：



LRU算法详解

一、什么是 LRU 算法

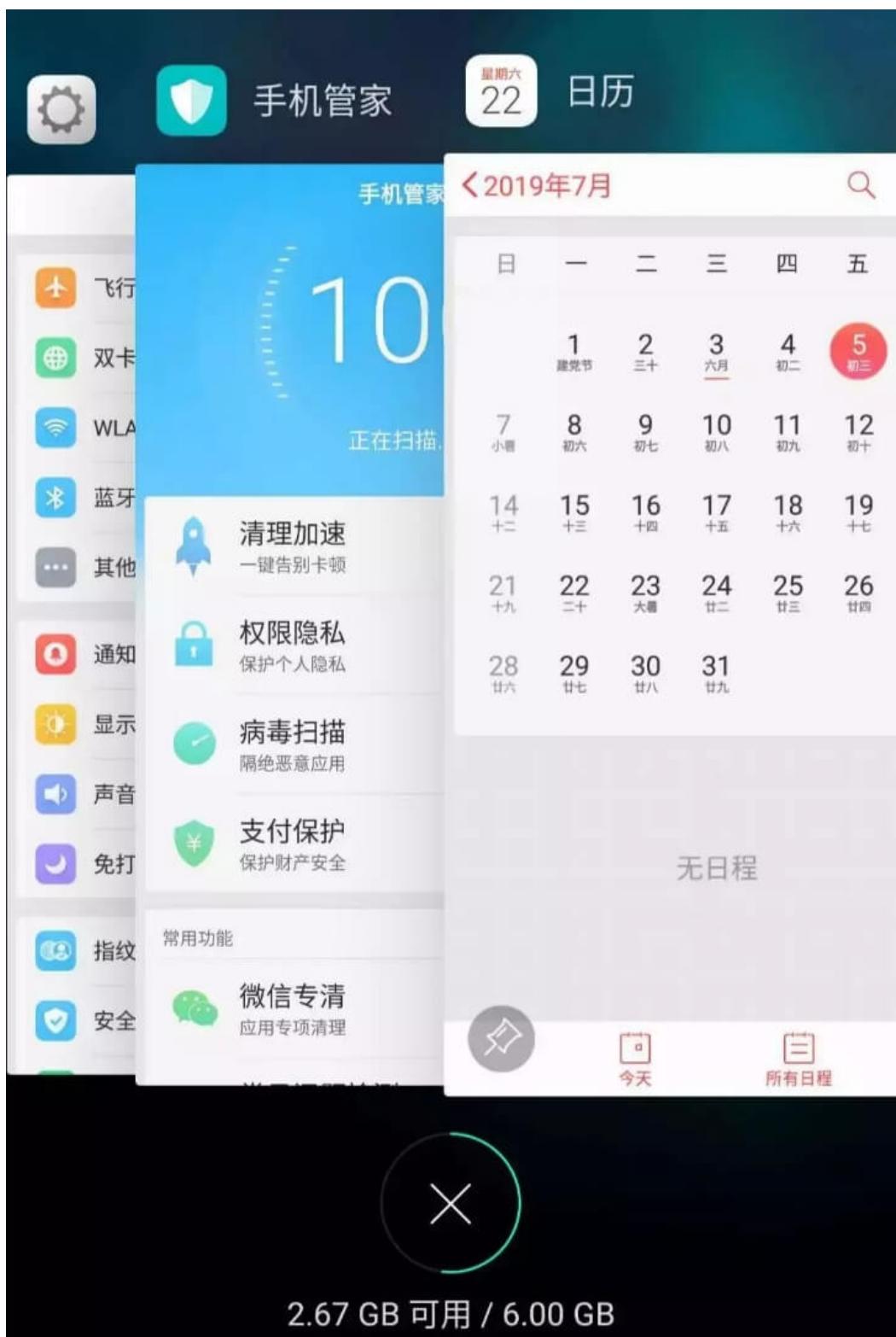
就是一种缓存淘汰策略。

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「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。

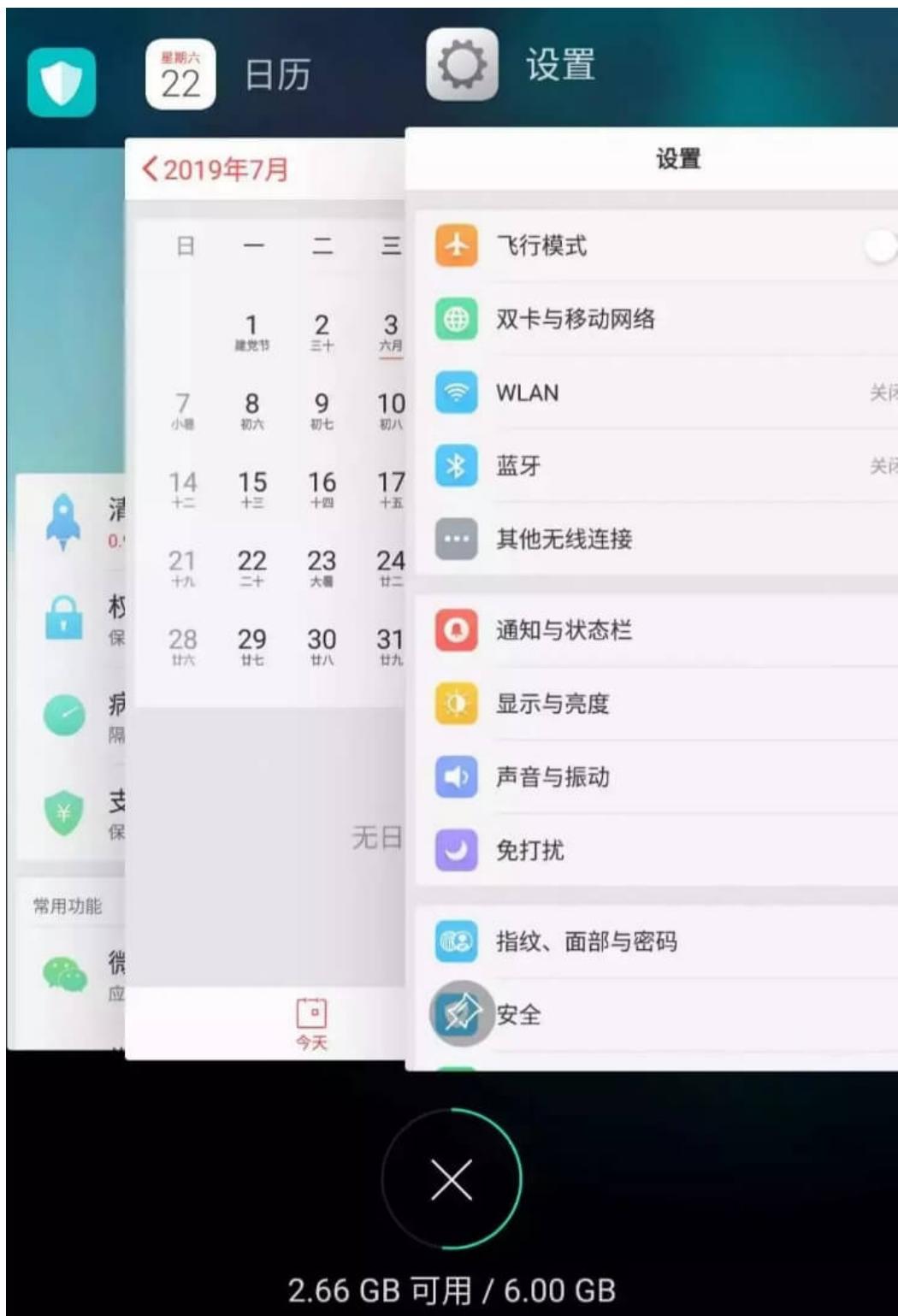
举个简单的例子，安卓手机都可以把软件放到后台运行，比如我先后打开了「设置」「手机管家」「日历」，那么现在他们在后台排列的顺序是这样的：

学习算法和刷题的框架思维



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：

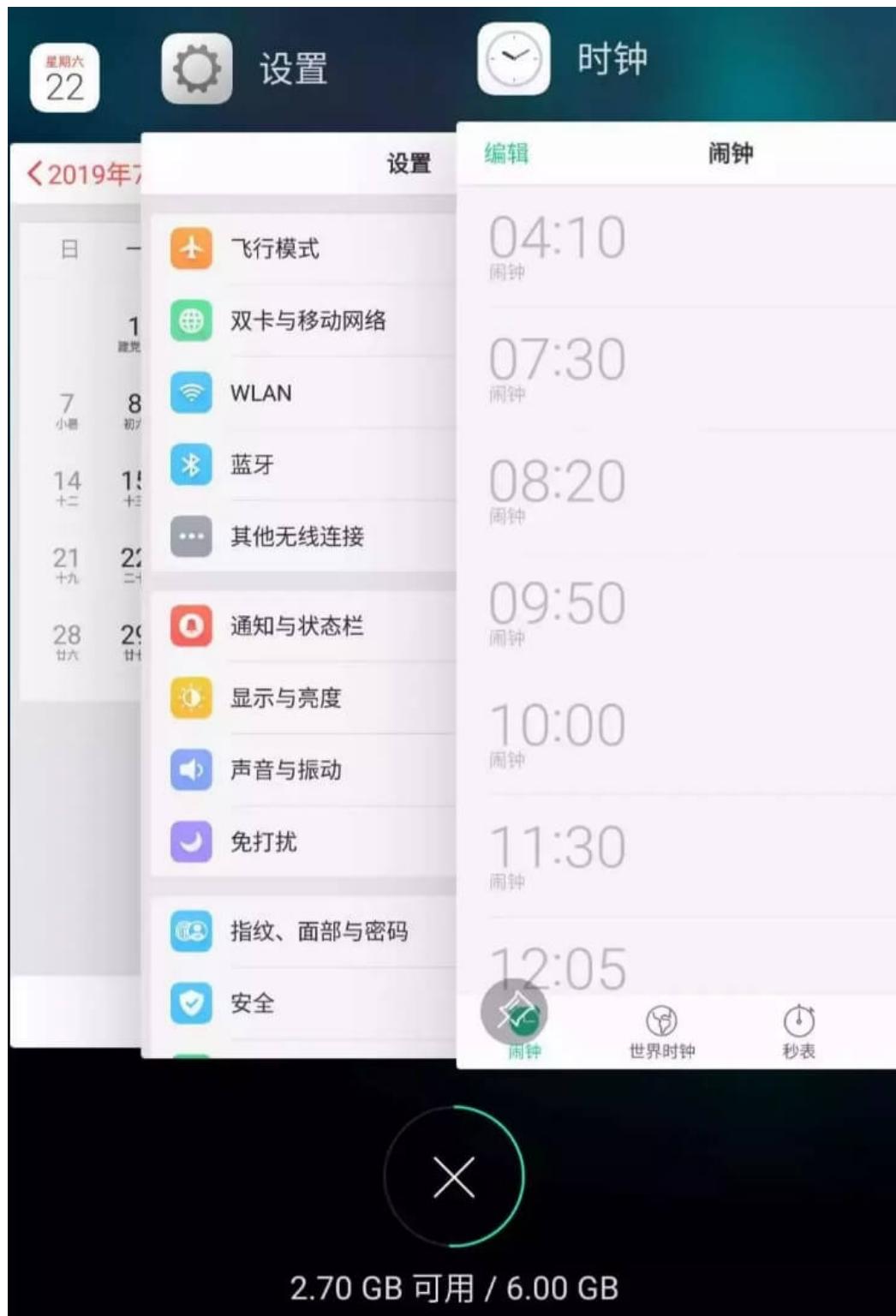
学习算法和刷题的框架思维



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？

学习算法和刷题的框架思维

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：



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

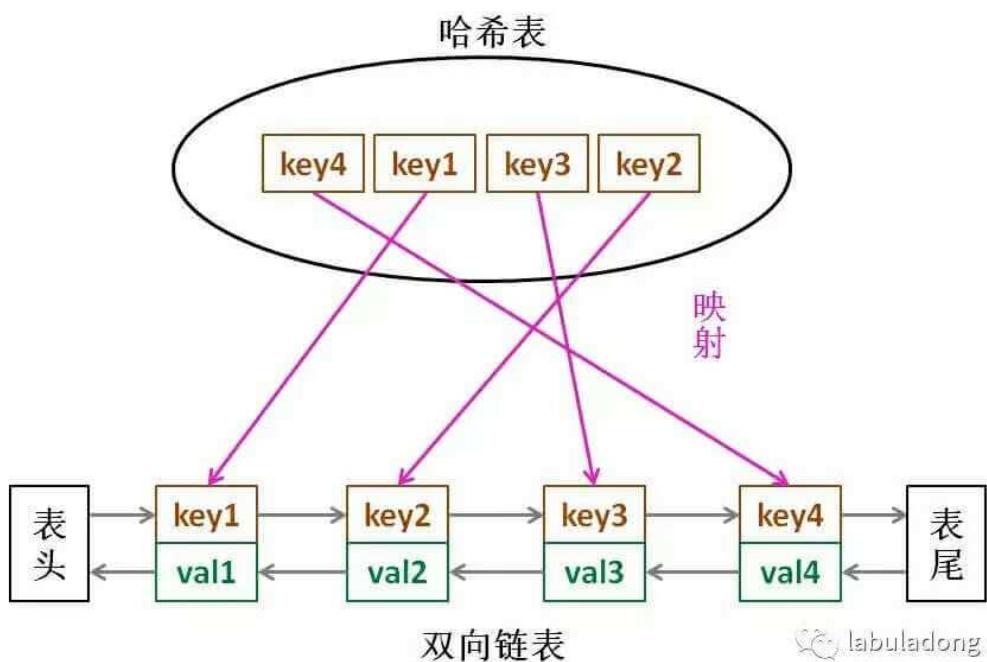
分析上面的操作过程，要让 put 和 get 方法的时间复杂度为 $O(1)$ ，我们可以总结出 cache 这个数据结构必要的条件：查找快，插入快，删除快，有顺序之分。

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();  
}
```

学习算法和刷题的框架思维

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：

学习算法和刷题的框架思维

```
// 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：

学习算法和刷题的框架思维

```
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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

学习算法和刷题的框架思维

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);  
}
```

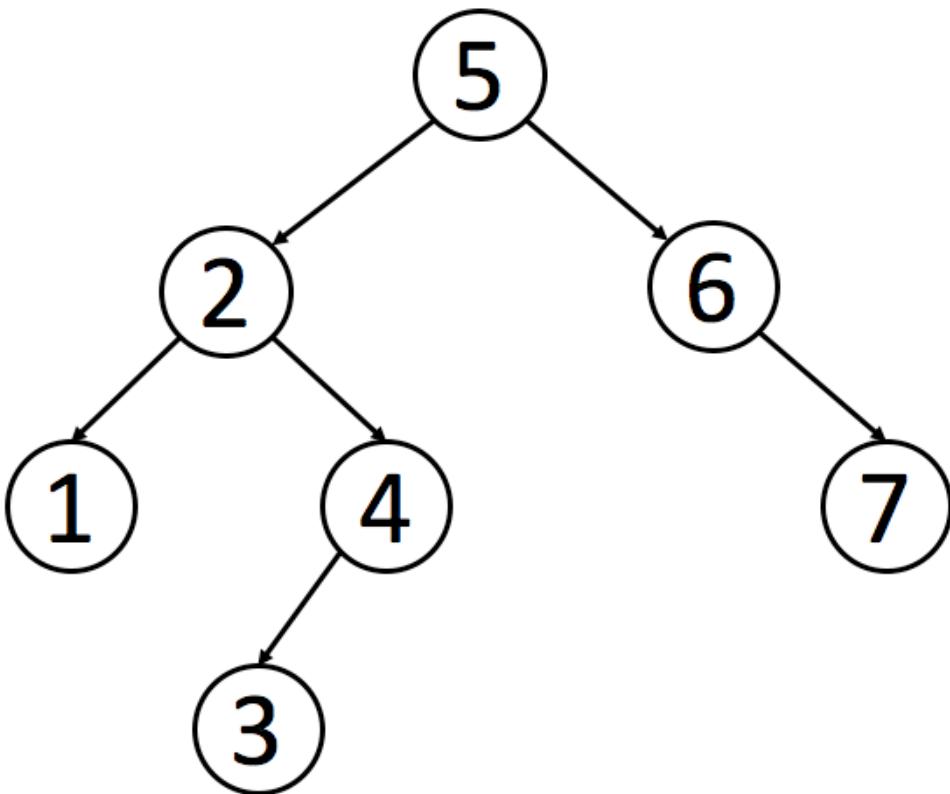
2. 如何判断两棵二叉树是否完全相同？

```
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。

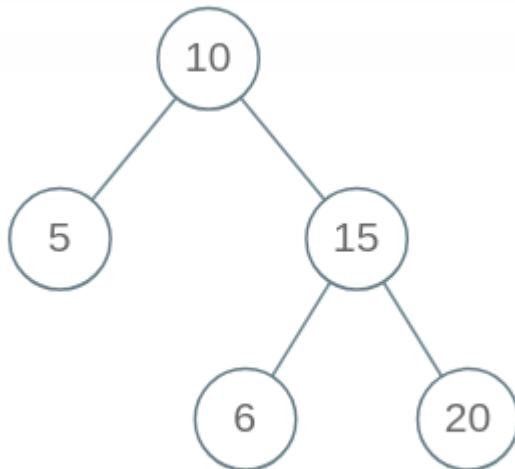
零、判断 BST 的合法性

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) ret  
    if (root.right != null && root.val >= root.right.val) i  
  
    return isValidBST(root.left)  
        && isValidBST(root.right);  
}
```

学习算法和刷题的框架思维

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：

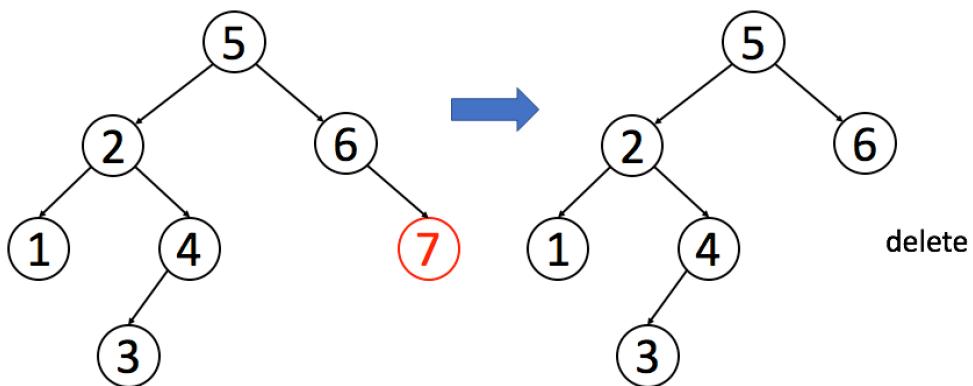
```
TreeNode deleteNode(TreeNode root, int key) {  
    if (root.val == key) {  
        // 找到啦, 进行删除  
    } else if (root.val > key) {  
        root.left = deleteNode(root.left, key);  
    } else if (root.val < key) {  
        root.right = deleteNode(root.right, key);  
    }  
    return root;  
}
```

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

Case 1: No Child



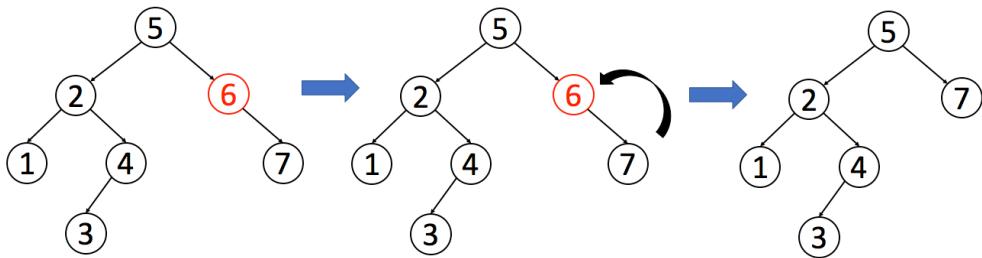
```
if (root.left == null && root.right == null)  
    return null;
```

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。

学习算法和刷题的框架思维

图片来自 LeetCode

Case 2: One Child

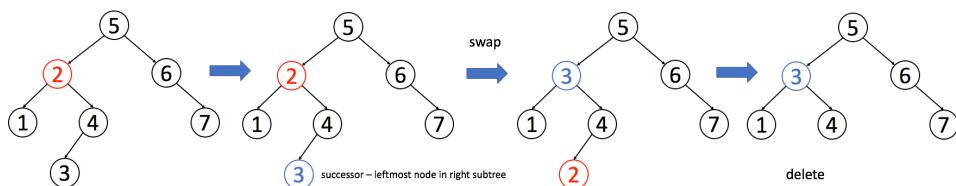


```
// 排除了情况 1 之后
if (root.left == null) return root.right;
if (root.right == null) return root.left;
```

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

Case 3: Two Children



```
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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

快速计算完全二叉树的节点

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

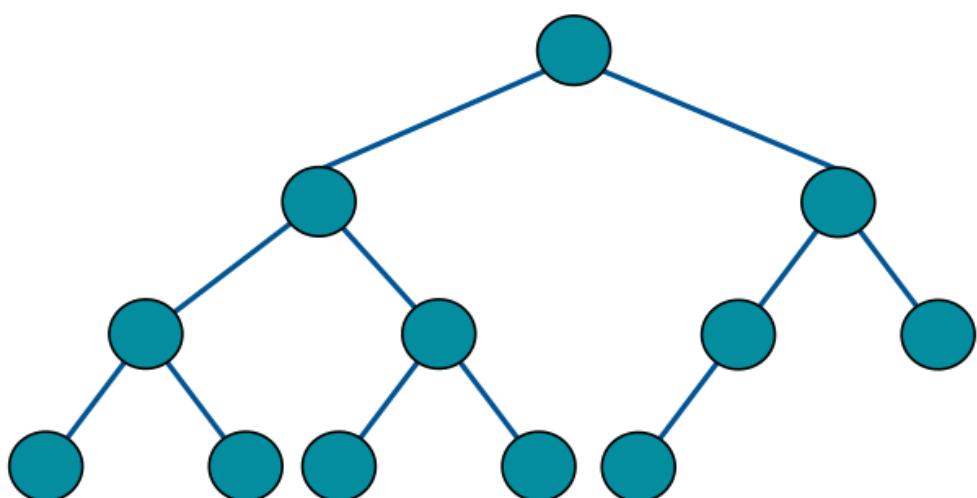
[222. 完全二叉树的节点个数](#)

如果让你数一下一棵普通二叉树有多少个节点，这很简单，只要在二叉树的遍历框架上加一点代码就行了。

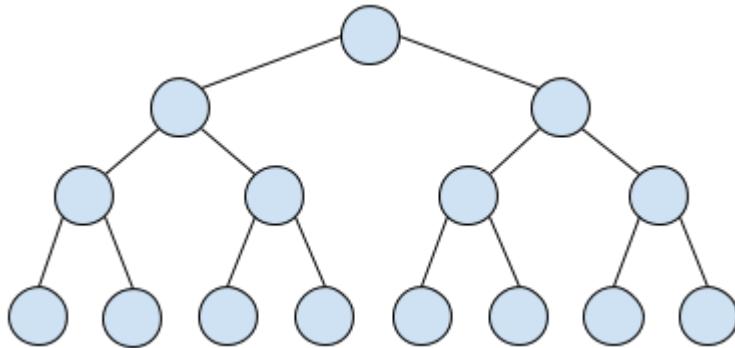
但是，如果给你一棵完全二叉树，让你计算它的节点个数，你会不会？算法的时间复杂度是多少？这个算法的时间复杂度应该是 $O(\log N * \log N)$ ，如果你心中的算法没有达到高效，那么本文就是给你写的。

首先要明确一下两个关于二叉树的名词「完全二叉树」和「满二叉树」。

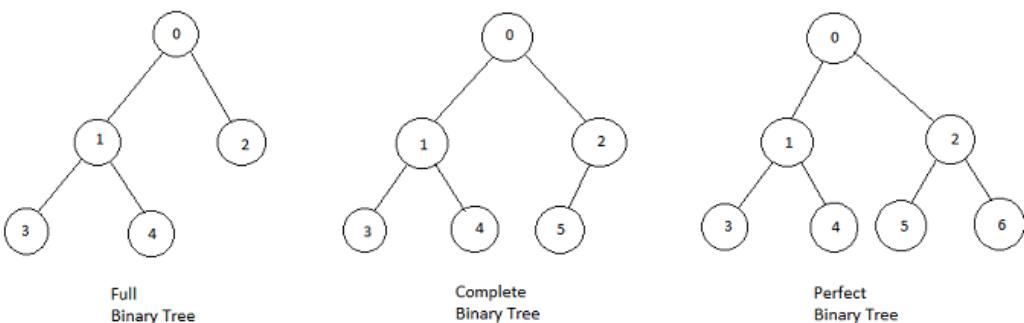
我们说的完全二叉树如下图，每一层都是紧凑靠左排列的：



我们说的**满二叉树**如下图，是一种特殊的完全二叉树，每层都是是满的，像一个稳定的三角形：



说句题外话，关于这两个定义，中文语境和英文语境似乎有点区别，我们说的完全二叉树对应英文 Complete Binary Tree，没有问题。但是我们说的满二叉树对应英文 Perfect Binary Tree，而英文中的 Full Binary Tree 是指一棵二叉树的所有节点要么没有孩子节点，要么有两个孩子节点。如下：



以上定义出自 wikipedia，这里就是顺便一提，其实名词叫什么都无所谓，重要的是算法操作。本文就按我们中文的语境，记住「**满二叉树**」和「**完全二叉树**」的区别，等会会用到。

由于格式原因，本文只能在 labuladong 公众号查看，关注后可直接搜索本站内容：

学习算法和刷题的框架思维



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：

$1 \leq m \leq n \leq$ 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;
}
```

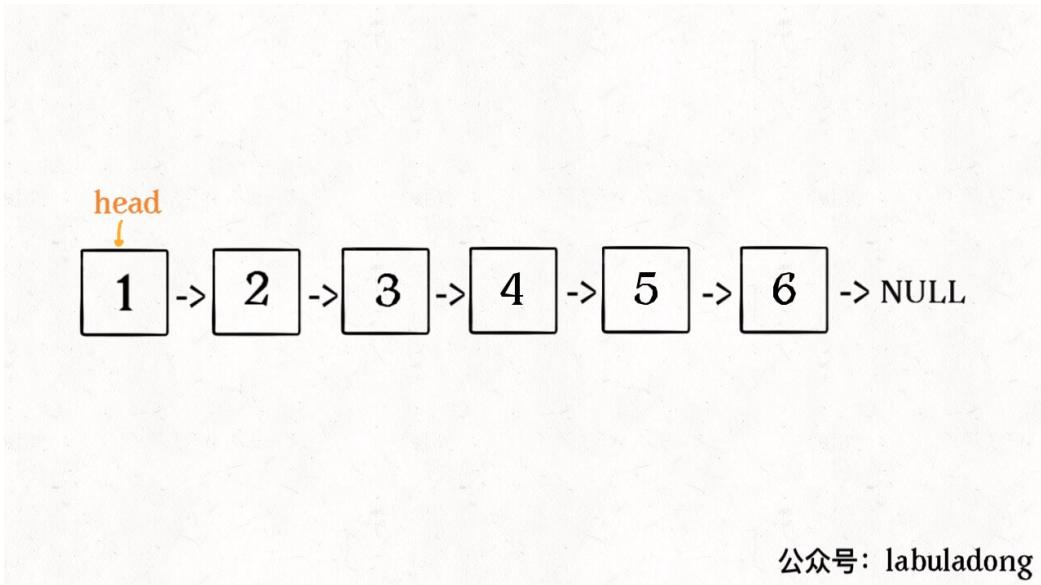
看起来是不是感觉不知所云，完全不能理解这样为什么能够反转链表？这就对了，这个算法常常拿来显示递归的巧妙和优美，我们下面来详细解释一下这段代码。

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：

学习算法和刷题的框架思维

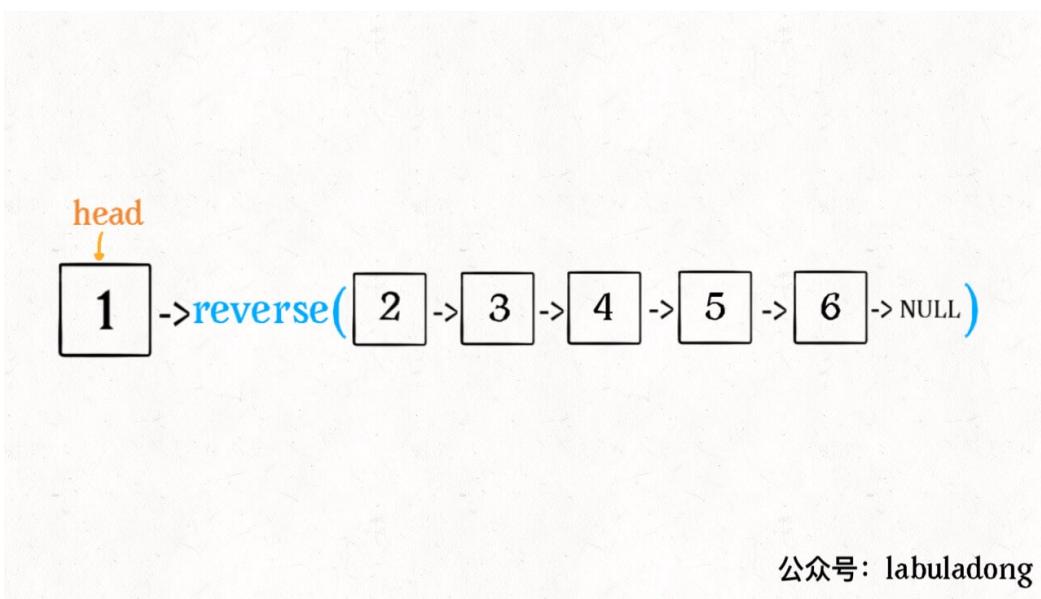


公众号: labuladong

那么输入 `reverse(head)` 后，会在这里进行递归：

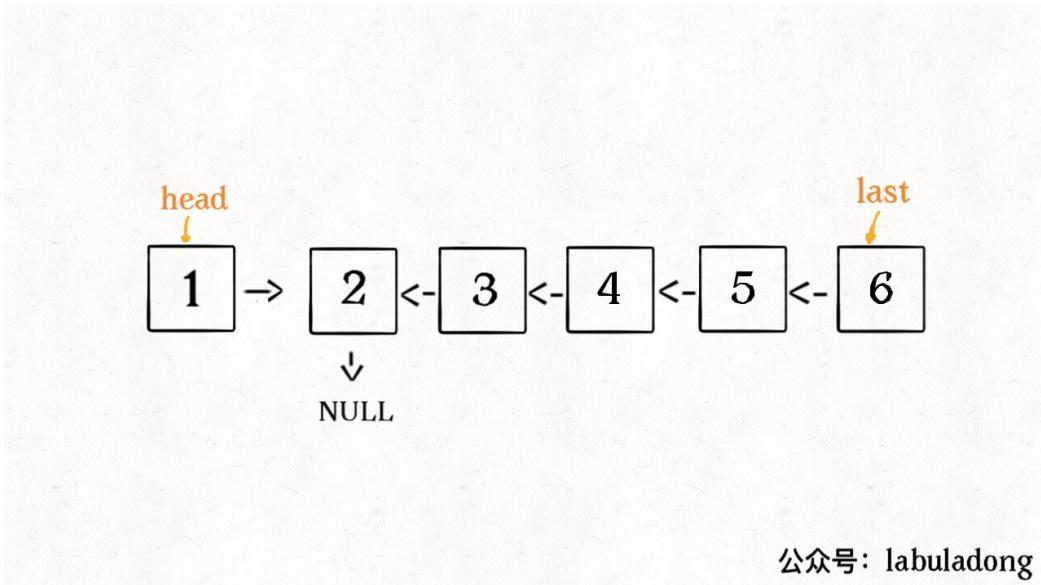
```
ListNode last = reverse(head.next);
```

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：



公众号: labuladong

这个 `reverse(head.next)` 执行完成后，整个链表就成了这样：

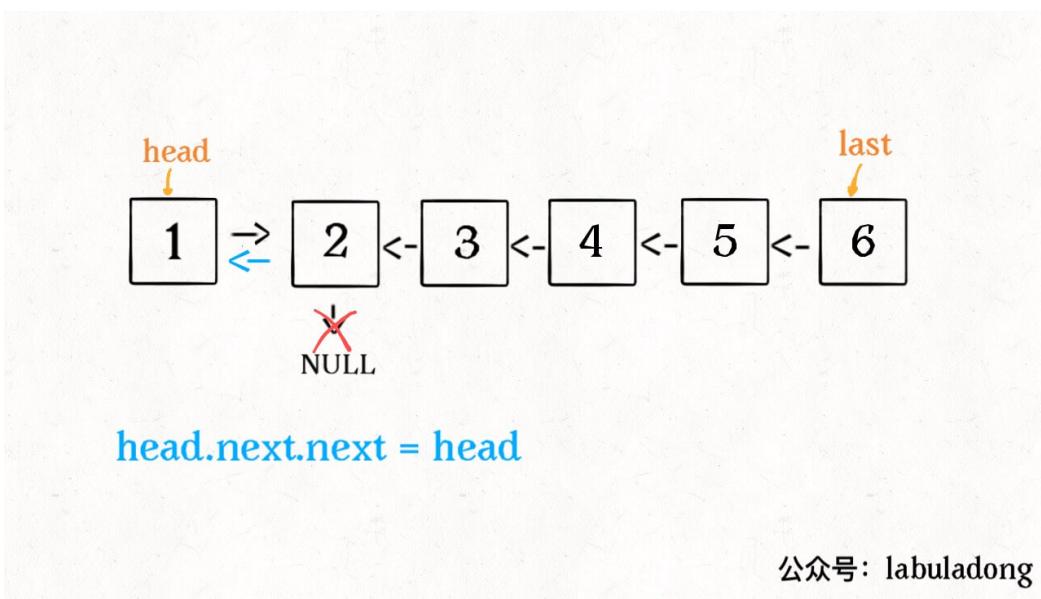


公众号: labuladong

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：

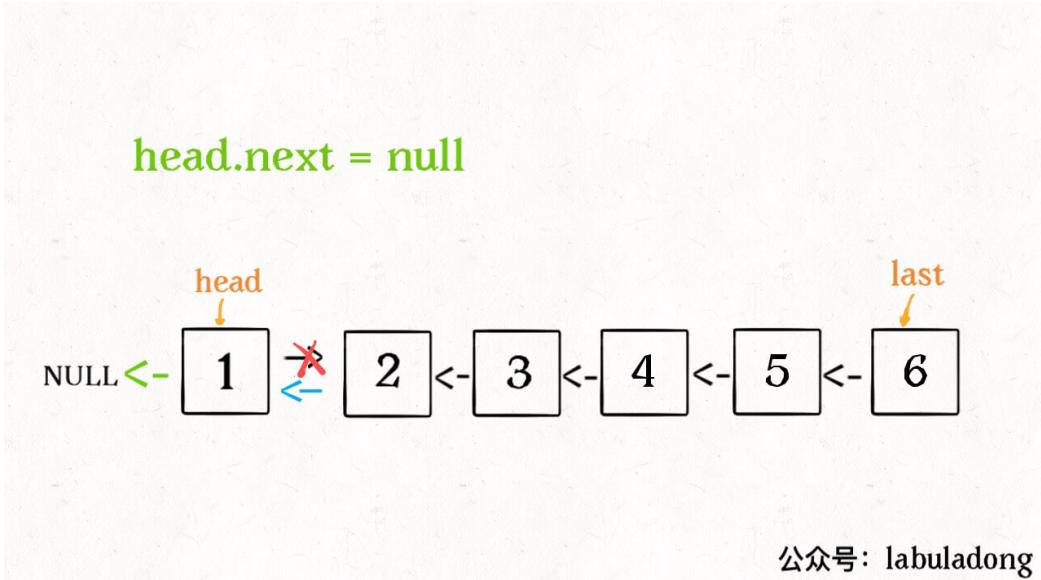
```
head.next.next = head;
```



公众号: labuladong

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;
```

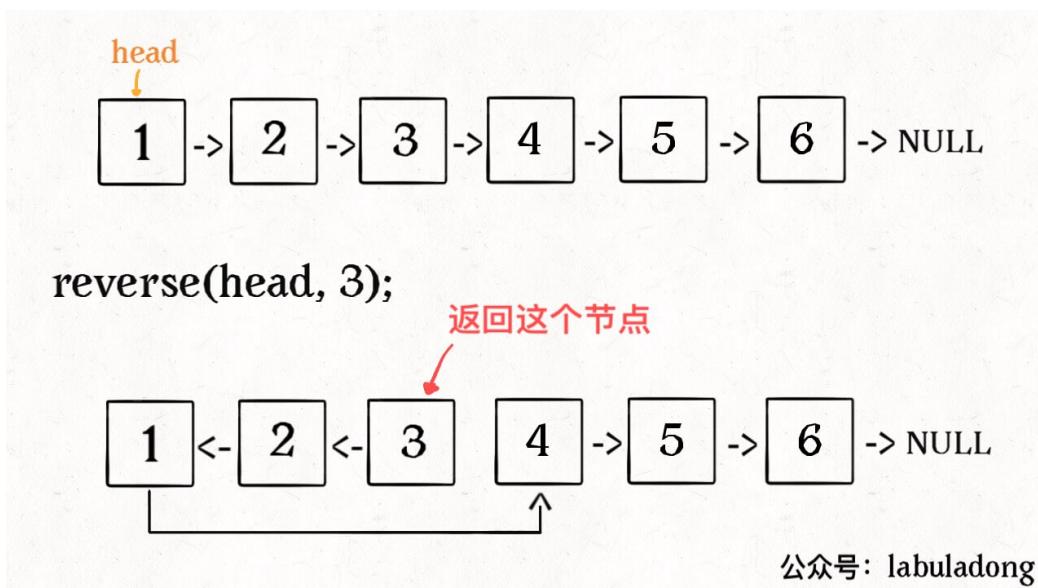
理解了这两点后，我们就可以进一步深入了，接下来的问题其实都是在这个算法上的扩展。

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)`：



公众号：labuladong

解决思路和反转整个链表差不多，只要稍加修改即可：

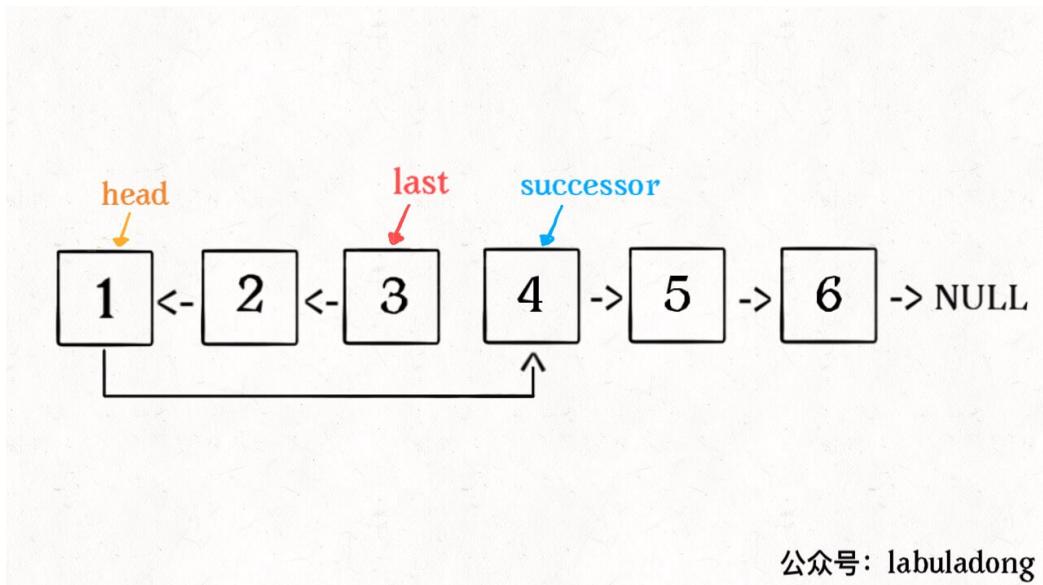
```
ListNode successor = null; // 后驱节点

// 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。



公众号: labuladong

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 $m \neq 1$ 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。

学习算法和刷题的框架思维

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或
[online book](#) 获取最新信息，后台回复「进群」可进刷题群，
labuladong 带你搞定 LeetCode。

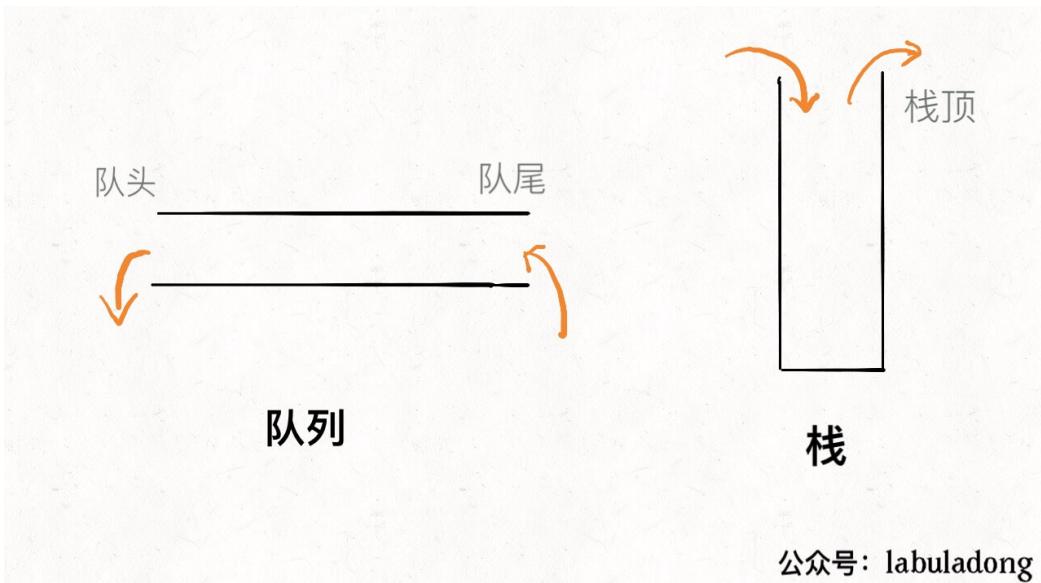


微信搜一搜

Q labuladong

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」。

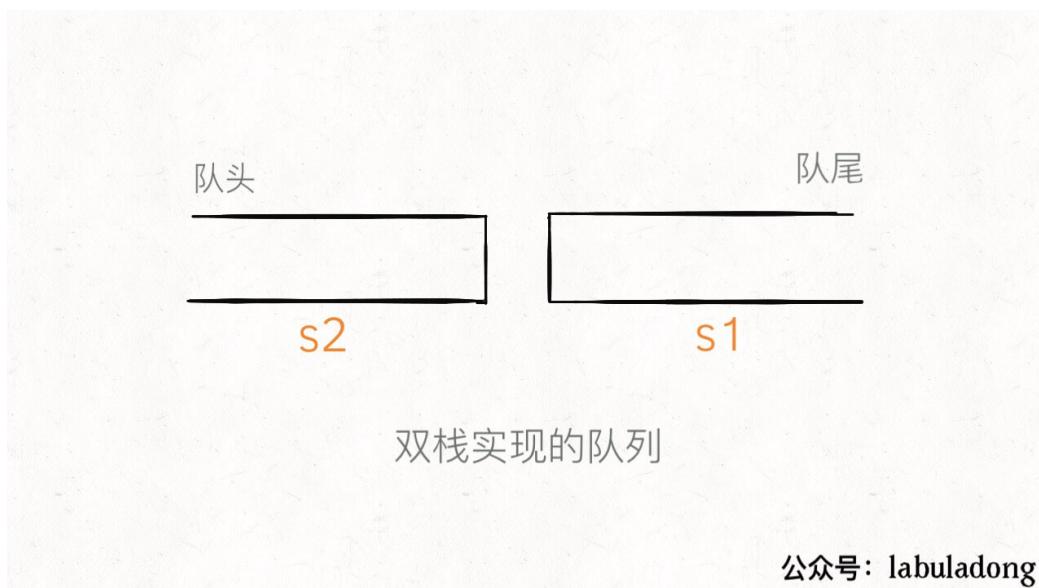
一、用栈实现队列

首先，队列的 API 如下：

学习算法和刷题的框架思维

```
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）：



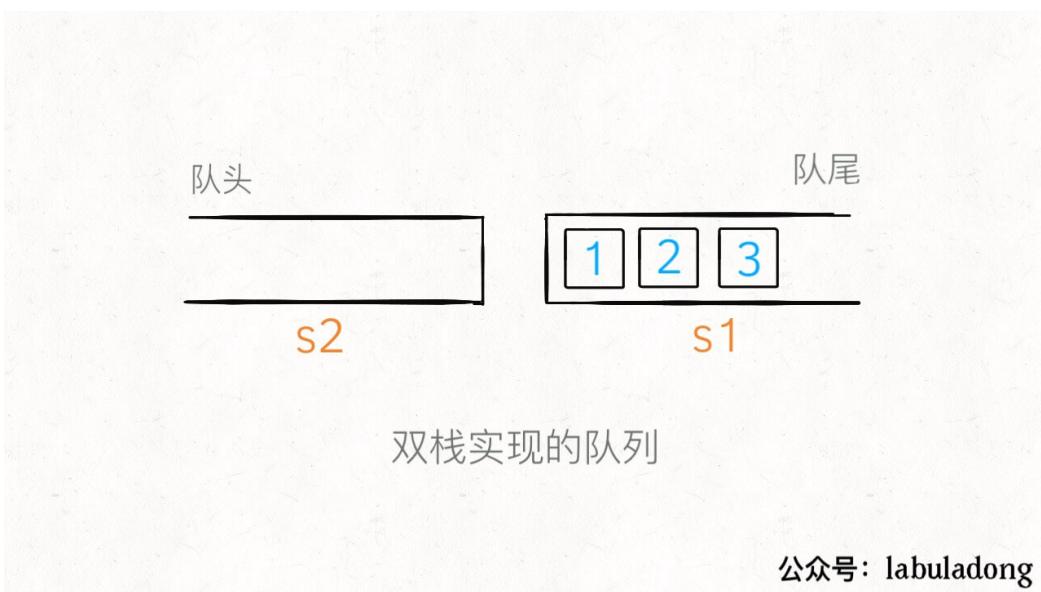
公众号: labuladong

学习算法和刷题的框架思维

```
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：



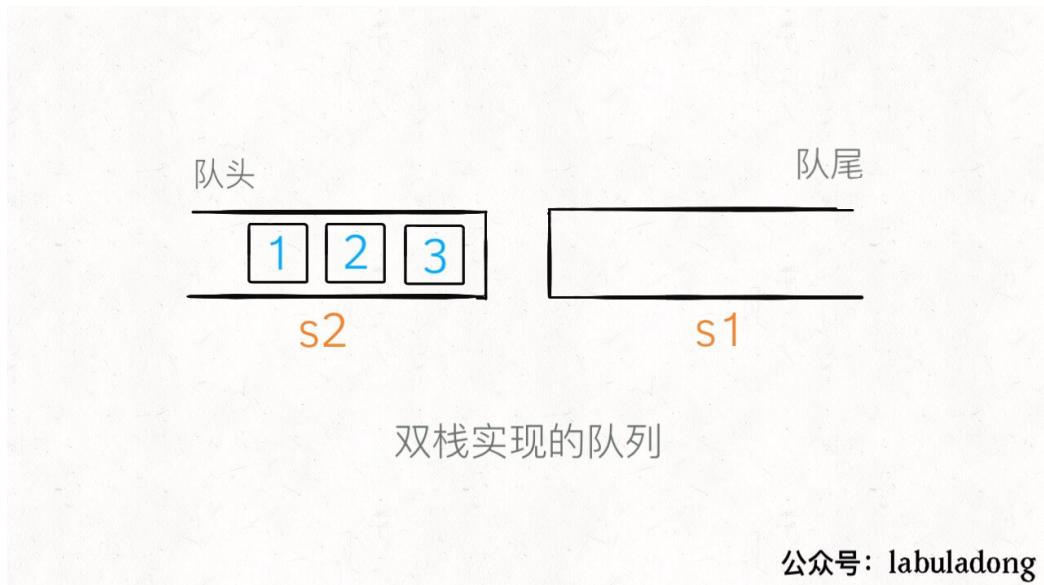
公众号: labuladong

```
/** 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`

学习算法和刷题的框架思维

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。



公众号: labuladong

```
/** 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：

学习算法和刷题的框架思维

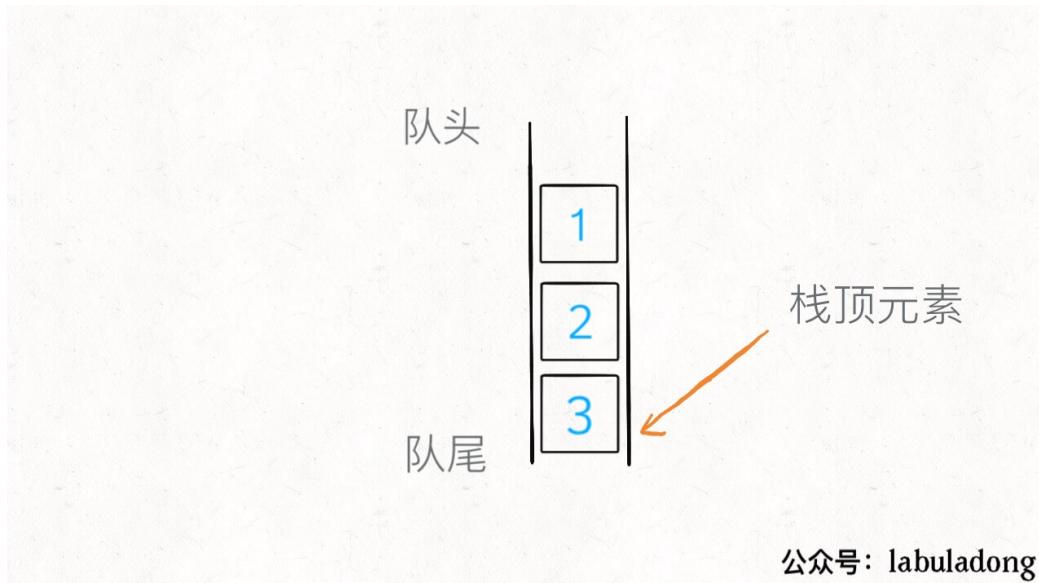
```
class MyStack {  
  
    /** 添加元素到栈顶 */  
    public void push(int x);  
  
    /** 删除栈顶的元素并返回 */  
    public int pop();  
  
    /** 返回栈顶元素 */  
    public int top();  
  
    /** 判断栈是否为空 */  
    public boolean empty();  
}
```

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：

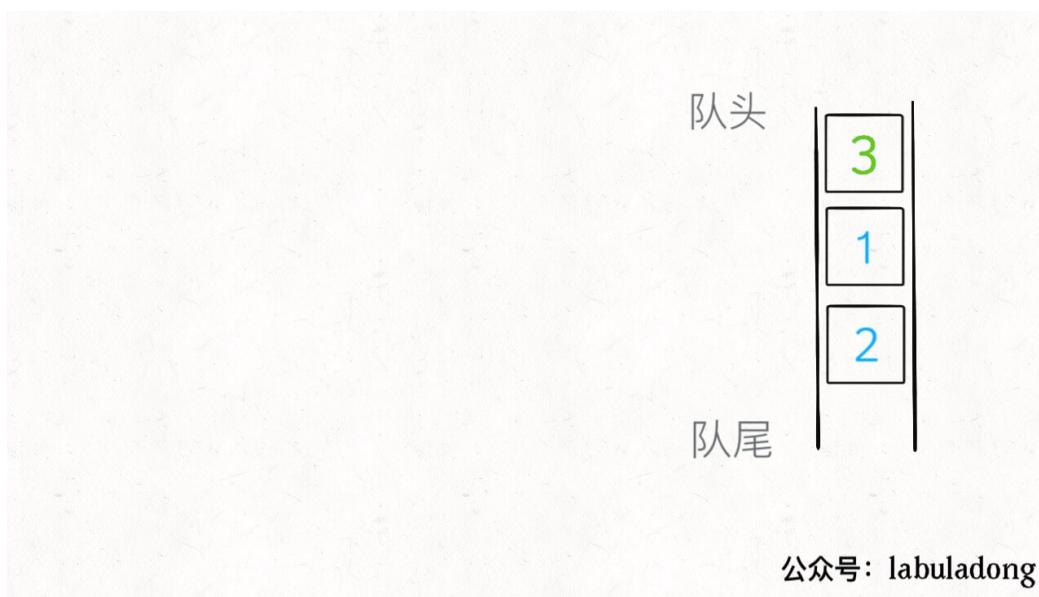
```
class MyStack {  
    Queue<Integer> q = new LinkedList<>();  
    int top_elem = 0;  
  
    /** 添加元素到栈顶 */  
    public void push(int x) {  
        // x 是队列的队尾，是栈的栈顶  
        q.offer(x);  
        top_elem = x;  
    }  
  
    /** 返回栈顶元素 */  
    public int top() {  
        return top_elem;  
    }  
}
```

学习算法和刷题的框架思维

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：



学习算法和刷题的框架思维

```
/** 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：

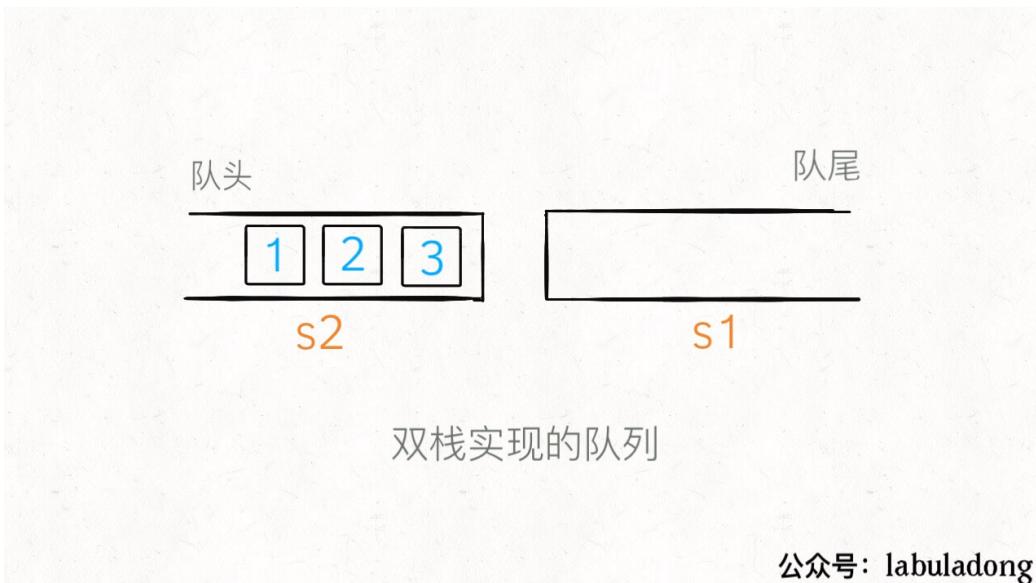
```
/** 删除栈顶的元素并返回 */
public int pop() {
    int size = q.size();
    // 留下队尾 2 个元素
    while (size > 2) {
        q.offer(q.poll());
        size--;
    }
    // 记录新的队尾元素
    top_elem = q.peek();
    q.offer(q.poll());
    // 删除之前的队尾元素
    return q.poll();
}
```

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。



公众号: labuladong

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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。

学习算法和刷题的框架思维



微信搜一搜

Q labuladong

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：



回溯算法之子集、排列、组合问题

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

[78.子集](#)

[46.全排列](#)

[77.组合](#)

今天就来聊三道考察频率高，而且容易让人搞混的算法问题，分别是求子集（subset），求排列（permutation），求组合（combination）。

这几个问题都可以用回溯算法模板解决，同时子集问题还可以用数学归纳思想解决。读者可以记住这几个问题的回溯套路，就不怕搞不清了。

由于格式原因，本文只能在 labuladong 公众号查看，关注后可直接搜索本站内容：

学习算法和刷题的框架思维



合法括号生成算法

括号问题可以简单分成两类，一类是前文写过的[括号的合法性判断](#)，一类是合法括号的生成。对于括号合法性的判断，主要是借助「栈」这种数据结构，而对于括号的生成，一般都要利用回溯递归的思想。

关于回溯算法，我们前文写过一篇[回溯算法套路框架详解](#) 反响非常好，读本文前应该读过那篇文章，这样你就能够进一步了解回溯算法的框架使用方法了。

回到正题，括号生成算法是 LeetCode 第 22 题，要求如下：

请你写一个算法，输入是一个正整数 `n`，输出是 `n` 对儿括号的所有合法组合，函数签名如下：

```
vector<string> generateParenthesis(int n);
```

比如说，输入 `n=3`，输出为如下 5 个字符串：

"((()))", "(()())", "((())()", "()(())", "()()()"

有关括号问题，你只要记住一个性质，思路就很容易想出来：

1、一个「合法」括号组合的左括号数量一定等于右括号数量，这个很好理解。

2、对于一个「合法」的括号字符串组合 `p`，必然对于任何 `0 <= i < len(p)` 都有：子串 `p[0..i]` 中左括号的数量都大于或等于右括号的数量。

学习算法和刷题的框架思维

如果不跟你说这个性质，可能不太容易发现，但是稍微想一下，其实很容易理解，因为从左往右算的话，肯定是左括号多嘛，到最后左右括号数量相等，说明这个括号组合是合法的。

反之，比如这个括号组合 `)())`，前几个子串都是右括号多于左括号，显然不是合法的括号组合。

下面就来手把手实践一下回溯算法框架。

回溯思路

明白了合法括号的性质，如何把这道题和回溯算法扯上关系呢？

算法输入一个整数 `n`，让你计算 `n` 对儿括号能组成几种合法的括号组合，可以改写成如下问题：

现在有 `2n` 个位置，每个位置可以放置字符 `(` 或者 `)`，组成的所有括号组合中，有多少个是合法的？

这个命题和题目的意思完全是一样的对吧，那么我们先想想如何得到全部 $2^{(2n)}$ 种组合，然后再根据我们刚才总结出的合法括号组合的性质筛选出合法的组合，不就完事儿了？

如何得到所有的组合呢？这就是标准的暴力穷举回溯框架啊，我们前文 [回溯算法套路框架详解](#) 都总结过了：

学习算法和刷题的框架思维

```
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
```

那么对于我们的需求，如何打印所有括号组合呢？套一下框架就出来了，伪码如下：

```
void backtrack(int n, int i, string& track) {
    // i 代表当前的位置，共 2n 个位置
    // 穷举到最后一个位置了，得到一个长度为 2n 组合
    if (i == 2 * n) {
        print(track);
        return;
    }

    // 对于每个位置可以是左括号或者右括号两种选择
    for choice in ['(', ')'] {
        track.push(choice); // 做选择
        // 穷举下一个位置
        backtrack(n, i + 1, track);
        track.pop(choice); // 撤销选择
    }
}
```

那么，现在能够打印所有括号组合了，如何从它们中筛选出合法的括号组合呢？很简单，加几个条件进行「剪枝」就行了。

对于 $2n$ 个位置，必然有 n 个左括号， n 个右括号，所以我们不是简单的记录穷举位置 i ，而是用 $left$ 记录还可以使用多少个左括号，用 $right$ 记录还可以使用多少个右括

学习算法和刷题的框架思维

号，这样就可以通过刚才总结的合法括号规律进行筛选了：

```
vector<string> generateParenthesis(int n) {
    if (n == 0) return {};
    // 记录所有合法的括号组合
    vector<string> res;
    // 回溯过程中的路径
    string track;
    // 可用的左括号和右括号数量初始化为 n
    backtrack(n, n, track, res);
    return res;
}

// 可用的左括号数量为 left 个, 可用的右括号数量为 right 个
void backtrack(int left, int right,
               string& track, vector<string>& res) {
    // 若左括号剩下的多, 说明不合法
    if (right < left) return;
    // 数量小于 0 肯定是不合法的
    if (left < 0 || right < 0) return;
    // 当所有括号都恰好用完时, 得到一个合法的括号组合
    if (left == 0 && right == 0) {
        res.push_back(track);
        return;
    }

    // 尝试放一个左括号
    track.push_back('('); // 选择
    backtrack(left - 1, right, track, res);
    track.pop_back(); // 撤消选择

    // 尝试放一个右括号
    track.push_back(')'); // 选择
    backtrack(left, right - 1, track, res);
    track.pop_back(); // 撤消选择
}
```

这样，我们的算法就完成了，算法的复杂度是多少呢？这个比较难分析，对于递归相关的算法，时间复杂度这样计算（递归次数）*（递归函数本身的时间复杂度）。

`backtrack` 就是我们的递归函数，其中没有任何 `for` 循环代码，所以递归函数本身的时间复杂度是 $O(1)$ ，但关键是这个函数的递归次数是多少？换句话说，给定一个 `n`，`backtrack` 函数递归被调用了多少次？

我们前面怎么分析动态规划算法的递归次数的？主要是看「状态」的个数对吧。其实回溯算法和动态规划的本质都是穷举，只不过动态规划存在「重叠子问题」可以优化，而回溯算法不存在而已。

所以说这里也可以用「状态」这个概念，对于 `backtrack` 函数，状态有三个，分别是 `left`, `right`, `track`，这三个变量的所有组合个数就是 `backtrack` 函数的状态个数（调用次数）。

`left` 和 `right` 的组合好办，他俩取值就是 $0 \sim n$ 嘛，组合起来也就 n^2 种而已；这个 `track` 的长度虽然取在 $0 \sim 2n$ ，但对于每一个长度，它还有指数级的括号组合，这个是不好算的。

说了这么多，就是想让大家知道这个算法的复杂度是指数级，而且不好算，这里就不具体展开了，是 $\frac{4^n}{\sqrt{n}}$ ，有兴趣的读者可以搜索一下「卡特兰数」相关的知识了解一下这个复杂度是怎么算的。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。

学习算法和刷题的框架思维



微信搜一搜

Q labuladong

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：

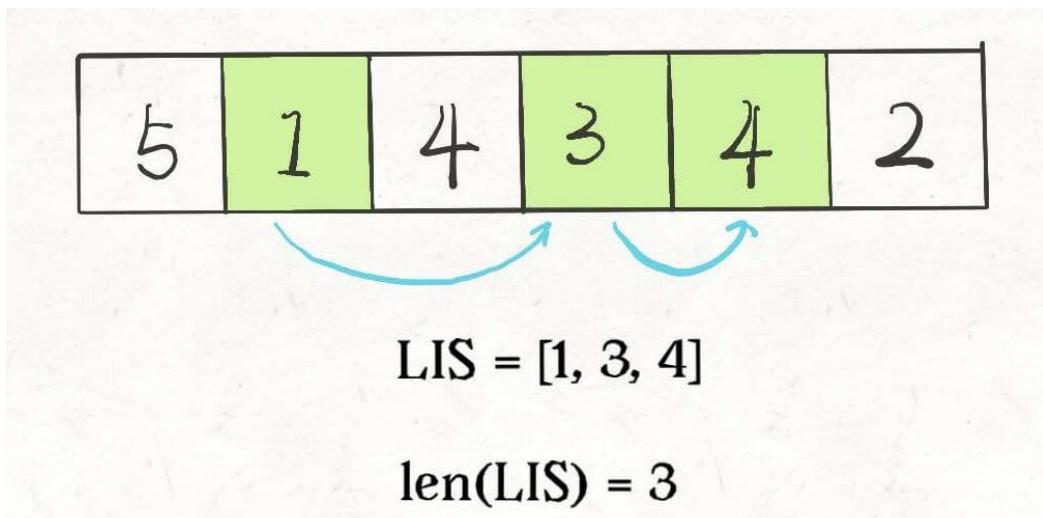
输入：envelopes = [[5, 4], [6, 4], [6, 7], [2, 3]]

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（Longest 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 $w \times h$ 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：

宽度 w 高度 h

[1 , 8]

[2 , 3]

[5 , 4]

[5 , 2]

[6 , 7]

[6 , 4]



学习算法和刷题的框架思维

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 $O(N \log N)$ ，因为排序和计算 LIS 各需要 $O(N \log N)$ 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。

学习算法和刷题的框架思维

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。

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或 [online book](#) 获取最新信息，后台回复「进群」可进刷题群，labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong

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，其中一个孩子是男孩，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

学习算法和刷题的框架思维

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：

$$1 - P(22 \text{ 个人都和我的生日不同}) = 1 - (364/365)^{22} = 0.06$$

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%？

学习算法和刷题的框架思维

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%：

$$P(\text{两次能中奖}) = P(\text{第一次就中了}) + P(\text{第一次没中但第二次中了}) = 1/2 + 1/2 * 1/2 = 75\%$$

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：

$$P(A') = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \cdots \times \frac{343}{365}$$

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。

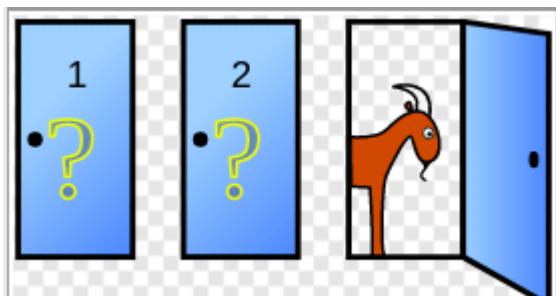
三、三门问题

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？

学习算法和刷题的框架思维

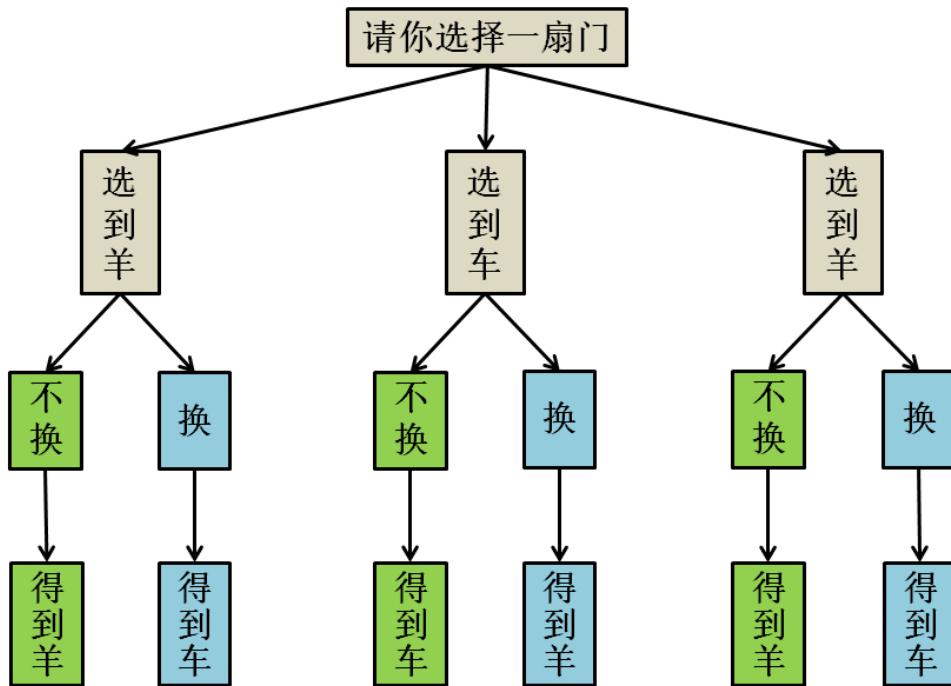
为了防止第一次看到这个问题的读者迷惑，再具体描述一下这个问题：

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。

学习算法和刷题的框架思维

本小抄将在 2020 年 12 月出版，关注 labuladong 公众号或
[online book](#) 获取最新信息，后台回复「进群」可进刷题群，
labuladong 带你搞定 LeetCode。



微信搜一搜

Q labuladong