Leetcode

70 Climbing Stairs

> 类型：DP基础题

> Time Complexity O(N)

> Space Complexity O(1)

这道题作为DP的启蒙题(拥有非常明显的重叠子结构)，我在这详细的讲一讲LC大神们的答案是如何从一个毫无优化的做法，效率极低的递归解法，慢慢的进化到他们现在的答案，也给不会DP思路的同学补一补基础。

Top-Down

这道题自顶向下的思考：如果要爬到n台阶，有两种可能性:

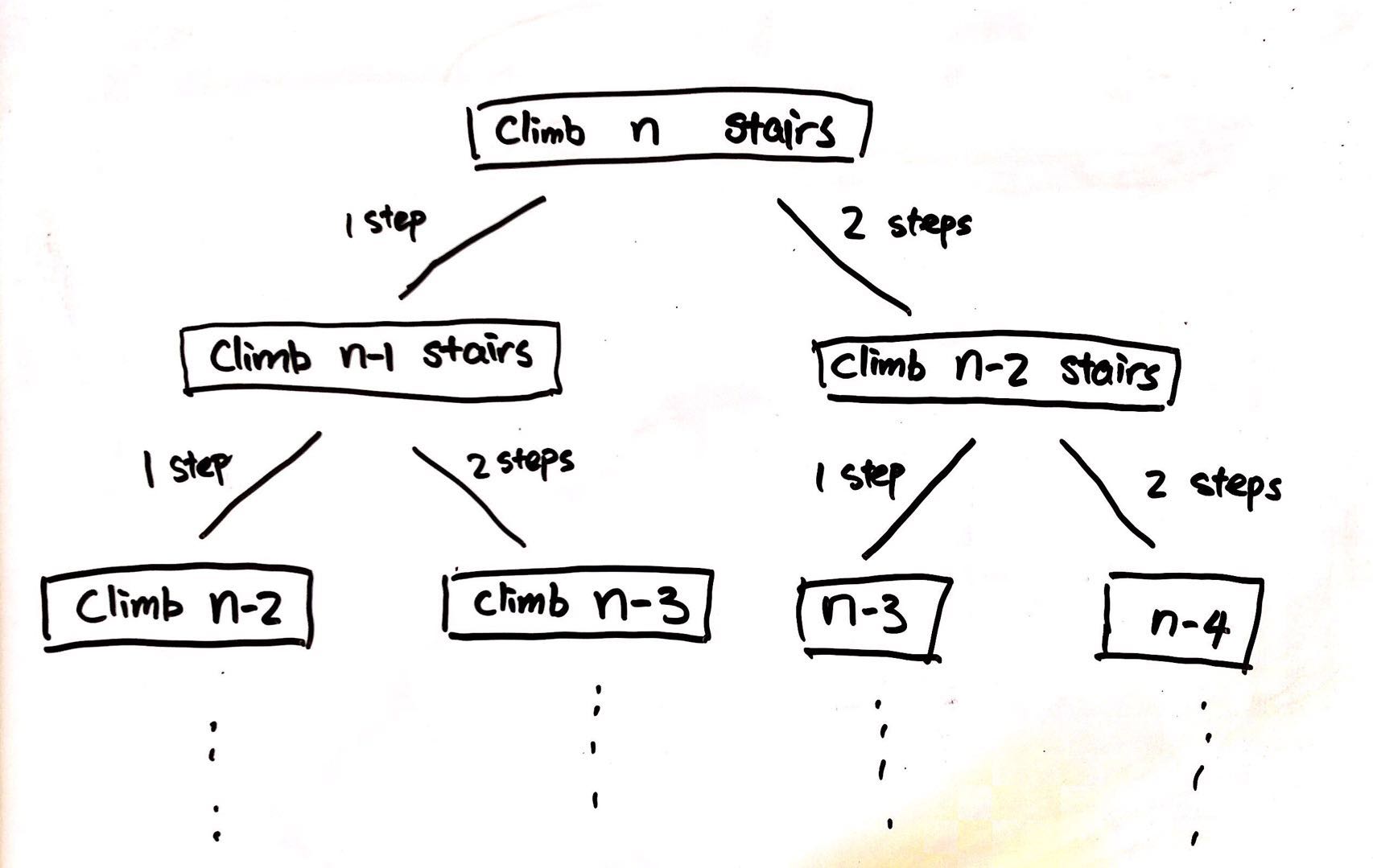
1. 在n-1的台阶处爬一层台阶
2. 在n-2的台阶处爬两层台阶

继续向下延伸思考，到达每一次层一共有几种方法这个问题就变成了2个子问题：

1. 到达n-1层台阶有几种方法
2. 到达n-2层台阶有几种方法

之后对返回子问题之和即可。

具体可以看下图。



根据以上的思路得到下面的代码

Solution 1: Recursion (TLE)

class Solution(object):

def climbStairs(self, n):

if n == 1: return 1

if n == 2: return 2

return self.climbStairs(n - 1) + self.climbStairs(n - 2)

TLE原因：

以上代码实现之所以会TLE，是因为递归的时候出现了很多次重复的运算。就如上图显示的爬n-2层的计算出现了2次，这种重复计算随着input的增大，会出现的越来越多，时间复杂度也会将以指数的级别上升。

优化思路：

这时候的思路为：如果能够将之前的计算好了的结果存起来，之后如果遇到重复计算直接调用结果，效率将会从之前的指数时间复杂度，变成O(N)的时间复杂度。

实现方法：

有了思路，实现方面则开辟一个长度为N的数组，将其中的值全部赋值成-1，具体为什么要用-1，是因为这一类问题一般都会要你求多少种可能，在现实生活中，基本不会要你去求负数种可能，所以这里-1可以成为一个很好的递归条件/出口。  
然后只要我们的数组任然满足arr[n] == -1，说明我们在n层的数没有被更新，换句话说就是我们还在向下递归或者等待返回值的过程中，所以我们继续递归直到n-1和n-2的值返回上来。这里注意我们递归的底层，也就是arr[0]和arr[1]，我们要对起始条件进行初始化：arr[0], arr[1] = 1, 2

Solution 2: Top-Down (using array)

class Solution(object):

def climbStairs(self, n):

if n == 1: return 1

res = [-1 for i in range(n)]

res[0], res[1] = 1, 2

return self.dp(n-1, res)

def dp(self, n, res):

if res[n] == -1:

res[n] = self.dp(n - 1, res) + self.dp(n - 2, res)

return res[n]

这样是不是还是很抽象？我们可以举个例子，当n = 4的时候，我们在每一层返回之前打印一下当前的数组的值：

# print(n+1, res)

(2, [1, 2, -1, -1])

(1, [1, 2, -1, -1])

(3, [1, 2, 3, -1])

(2, [1, 2, 3, -1])

(4, [1, 2, 3, 5])

大家看到了，我们先从第4层开始递归，当递归到了1和2层的base case的时候，开始进行返回的计算，  
当到了第3层，将1和2层加起来，得到3，继续返回  
当到了第4层，将2和3层加起来，得到了5，这时候res[n] = 5，则满足出口条件，进行返回。

这就是Top-Down的思路，从大化小，思路就和DFS Tree中的分制一样。

Solution 2.5: Top-Down (using hash table)

这种的话就没有太多的篇幅，无脑塞哈希表就行。

class Solution(object):

def \_\_init\_\_(self):

self.memo = {0 : 0, 1 : 1, 2 : 2}

def climbStairs(self, n):

if n in self.memo:

return self.memo[n]

left = self.climbStairs(n - 1)

self.memo[n - 1] = left

right = self.climbStairs(n - 2)

self.memo[n - 2] = right

return left + right

or

class Solution(object):

def \_\_init\_\_(self):

self.memo = {0 : 0, 1 : 1, 2 : 2}

def climbStairs(self, n):

if n in self.memo:

return self.memo[n]

res = self.climbStairs(n - 1) + self.climbStairs(n - 2)

self.memo[n] = res

return res

Bottom-Up

Bottom-Up的思路则相反。我们不从n层向下考虑，而是解决我们每一层向上的子问题。在每一层，我们其实只需要知道在当前节点，我们的n-1和n-2的值是多少即可。

当我们有了第1层和第2层的base case，我们则可以直接从base case向上推得第3层，第4层以及第n层的答案，具体实现如下：

Solution 3: Bottom-Up (using array)

class Solution(object):

def climbStairs(self, n):

if n == 1: return 1

res = [0 for i in range(n)]

res[0], res[1] = 1, 2

for i in range(2, n):

res[i] = res[i-1] + res[i-2]

return res[-1]

这种方法的使用的时候，我们发现其实在每一次更新当前的数的时候，我们最终返回的是最后一次更新的数，而我们的dependency是n-1 和n-2中的值，我们根本不需要一个数组去储存，直接用两个函数即可。所以底下为Bottom-Up的优化：

Solution 3: Bottom-Up (Constant Space)

class Solution(object):

def climbStairs(self, n):

if n == 1: return 1

a, b = 1, 2

for \_ in range(2, n):

a, b = b, a + b

return b

公瑾在刷题，加入我? : <https://github.com/yuzhoujr/leetcode/projects/1>