1、写在前面

这几天疯狂刷leetcode的dp题，刷到我都想吐了，看到一个题就先往dp想，感觉中毒不浅，话不多说，赶紧开始总结一下我的成长~~历程~~心得。

2、我对dp的看法和理解

- 首先引入《算法笔记》里的一句话:

  > 动态规划是一种非常精妙的算法思想，他没有**固定的写法、极其灵活，常常需要具体问题具体分析**。

  没错正式这种具体问题具体分析的设定让我刷到吐了
- 平时在思考动态规划的往往是利用一个小小的公式：**递推 = 递归 + 记忆化**（下面会赘述我是怎么利用这个公式思考的）
- 虽然他是这么说，但是我还是觉得dp的是有套路可循的，例如给dp分类 *（背包dp、树形dp、区间dp……）*，经典题的记忆 *（爬楼梯、树塔问题、LIS……）*
- 一般来说，我思考dp都是从结果往前想的（即从末尾开始想）
- **dp的基本使用条件：** 1. 拥有重叠子问题、 2. 最优子结构
- **dp思考三步走：** 1. 状态定义 → 2.列状态转移方程 → 验证方程
- 我觉得的动规成长路线：小白 → 利用小公式辅助建立dp → 正常dp三步走思维思考 → 大佬 （~~我还没触及~~ 将来一定达到(☉ꇴ☉)/）

3、典型例题

①爬楼梯（leetcode #70）

题意

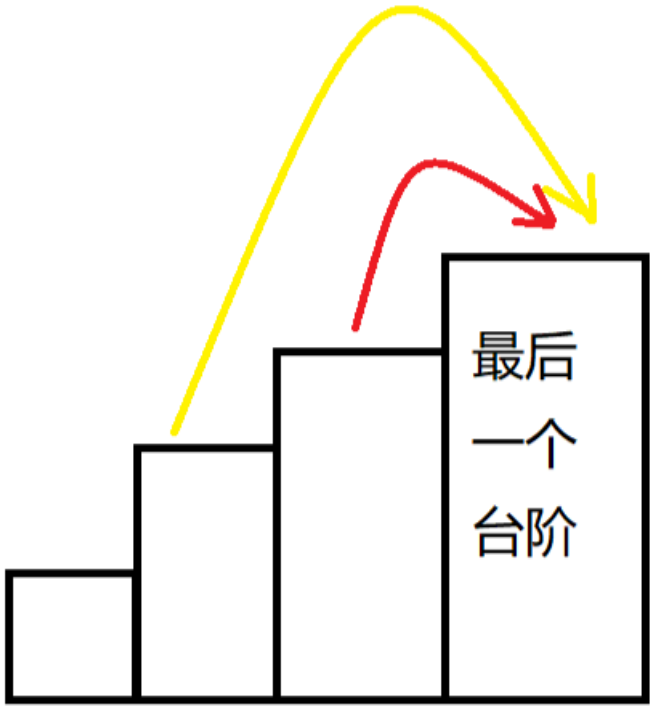
> 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
> 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢？
> 注意：给定 n 是一个正整数。
>
> 示例 1：
> 输入： 2
> 输出： 2
> 解释： 有两种方法可以爬到楼顶。
> 1. 1 阶 + 1 阶

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

分析

（抛开动规，假设我们是小白）我的思考过程，利用上述提到的小小公式（递推 = 递归 + 记忆化）

- 首先：从末尾结果开始想，如果我们要走到最后一个台阶,那么我们就有两种走到这个台阶的方法：一是从这一个台阶的前一个台阶上来、二是从这个台阶的前第二个台阶上来。如图：

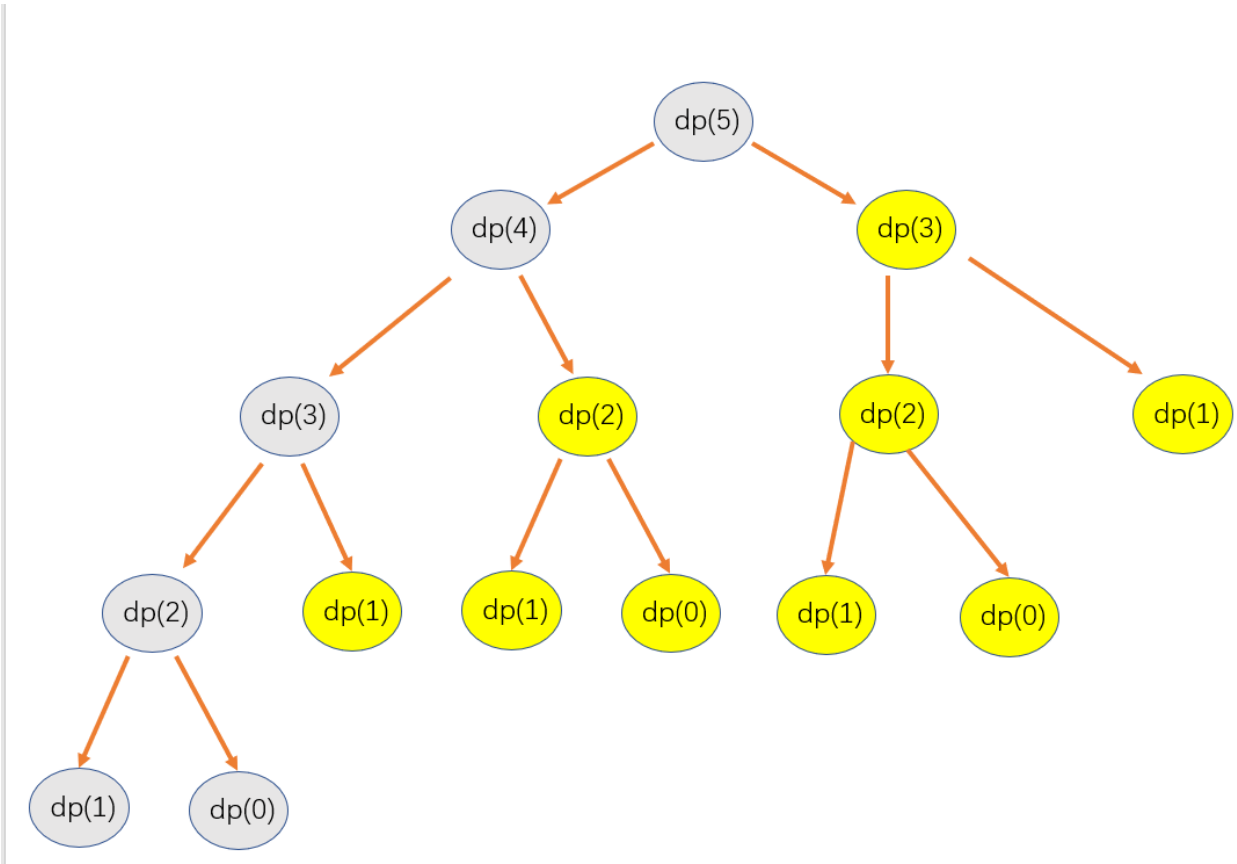


- 这样我们就很容易想到到达**最后一个台阶的方法数**为**到前一个台阶的方法数**加上**到前第二个台阶的方法数**，可能很拗口。看下面的公式就懂了。
- 设 **dp(n)** 为到台阶n的总方法数，这样很容易就能写出这个方法数的公式为：

- 不难写出递归代码

```
int dp(int n){
    if(n == 1 || n == 0)n
        return 1;
    return dp(n - 1) + dp(n - 2);
}
```

- 仔细分析一下这个代码，时间复杂度不难发现式O(2^n)，例如假设我们求的是dp(5)

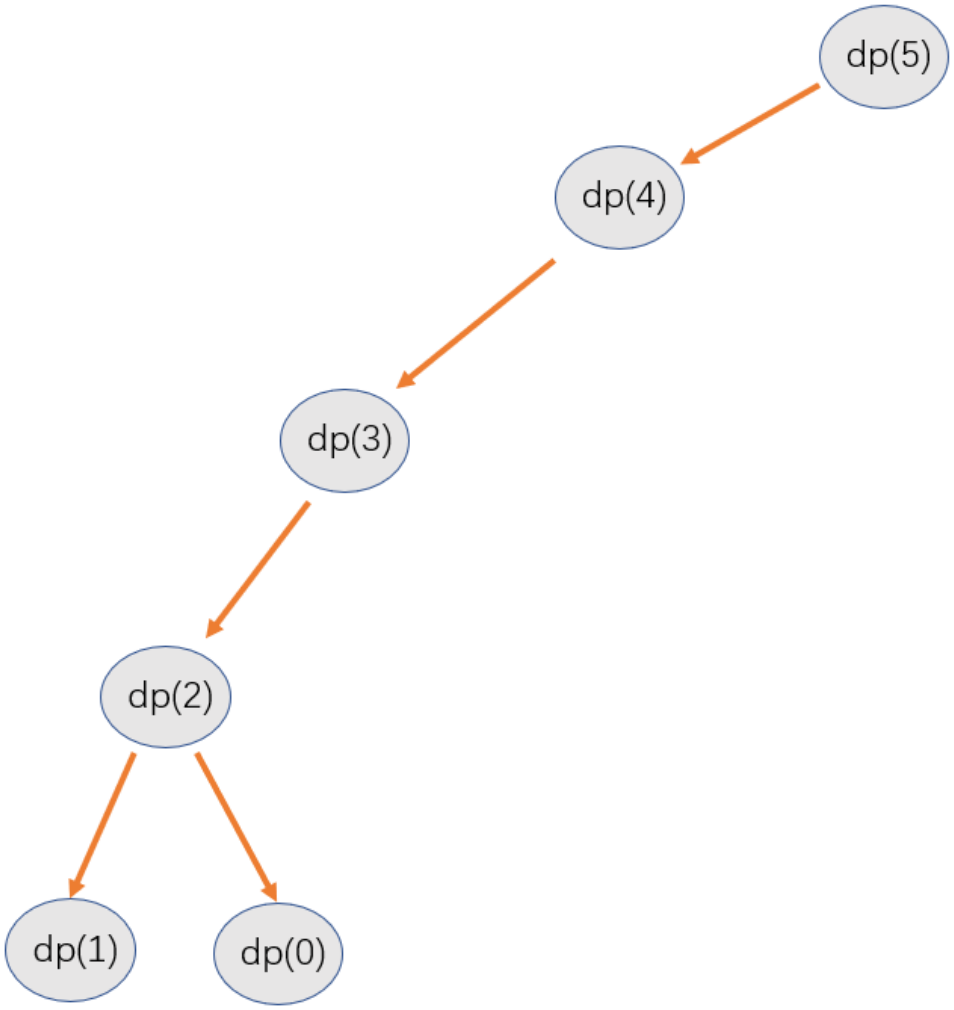


- 不难发现有很多重复计算(黄色部分)，这里采取了一个比较好的优化方法--记忆化递归（是不是觉得很接近那个小公式了？别急看代码）

```
const int Max = 50;
int memor[Max] = {0};

int dp(int n) {
    if (n == 1 || n == 0) return 1;
    if (memor[n] != 0) return memor[n];
    memor[n] = dp(n - 1) + dp(n - 2);
    return memor[n];
}
```

- 时间复杂度降到了O(n),就是变成这样



两者时间对比：

```
普通递归计算dp(45) : 3464 ms
记忆化递归计算dp(45) : 1 ms
```

天壤之别！！！

- 回过头来看这题，这题是一个典型的动规问题，那么我是怎么利用那个小公式来思考这个题的状态方程的呢，很简单就是按照刚刚的思路走一遍。
- 不难发现我们按照那个思路走，其实已经写出了转移状态方程，就是下面这个，并且这个方程的含义就是我们递归时的含义，这就相当于一下完成了**dp三步走**的前两步。

- 接下来的一步就是检查（验证）方程，**首先**我们要思考这道题适不适合用动规的方法来写，这就需要我们验证这个题是否满足**1.有重叠子问题，2.有最优子结构**。*重叠子问题*的话，刚刚分析递归的时候已经发现了，并且优化后不会出现重复计算子问题；*最优子结构*就是分析没有个状态（即dp(n)）是否是最优的解，在一开始分析题目时不难发现每一个台阶的步数只取决于他前两个台阶的步数。

- 验证方程的最后一步，设定状态边界（我们总不可能让他一直计算下去吧），其实这道题的边界就是我们一开始的递归边界。
- 到了这里dp的代码就不难了,递归式自顶向下地计算，而我们动规就是自顶向下地思考，自底向上地计算，多说无益，上代码

```
const int Max = 50;
int dp[Max], n;

int main() {
    scanf("%d", &n);
    dp[0] = 1, dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    printf("%d", dp[n]);
    return 0;
}
```

- 时间复杂度同样是O(n),虽然还有更快O(logn)的方式解这题，~~但我们现在是小白呀~~，下次再聊。

小结

- 通过这道题我们从小白晋级到了会用小公式来思考dp。
- 1、首先从末尾结果开始思考
- 2、思考怎么递归
- 3、再来就是写出递归的的方程
- 4、分析递归，去除重叠问题
- 5、转换成dp思想，直接走dp三步走的第三步，检查是否状态方程，若不满足，回到1看看有没有另一种更好的办法（一般不会出现这种情况),当然还有另一种方法就是在原有的基础上再次优化（这些都是后话）；若满足直接dp走你。

未完待续.....