动态规划方法论

动态规划是算法面试中的一个“大IP”，同时也是很多同学的心头痛。本节致力于用舒服的姿势帮助大家克服这块心病，因此开篇不能急于怼知识点，要先讲讲方法。

在笔者看来，对于动态规划的学习，最重要的是找到一个正确的学习切入点：如果你是一个对相关理论一无所知的初学者，自然不能急于一上来就生吞“模型”、“状态转移方程”等高端概念——大家谨记，动态规划是一种**思想**，所谓思想，就是**非常好用，好用到爆的套路**。我们学习一种思想，**重要的是建立起对它的感性认知，而不是反复咀嚼那些对现在的你来说还非常生硬的文字概念——从抽象去理解抽象是意淫，从具体去理解抽象才是学习。**

本节将会延续小册一贯的讲解风格：首先带大家一起解决一个实际的问题，然后逐步复盘问题的解决方案，最后从解决方案中提取出动态规划相关的概念、模型和技巧，实现对号入座。

> 从前面一系列章节的学习反馈中，笔者观察到一部分同学的阅读习惯非常“薄情”——打开小册只为做题，做完就溜，讲解部分基本是不看的。 这里想要提醒大家的是，题目本身不仅仅是命题点，更是素材、是教具，大家最终要关注到的还是题目背后的思想和方法。因此希望同学们能多给自己一点时间、多一些耐心去反刍和吸收知识。

从“爬楼梯”问题说起

> 题目描述：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

思路分析与编码实现

这道题目有两个关键的特征：

1. 要求你给出达成某个目的的**解法个数**
2. 不要求你给出每一种解法对应的具体路径

这样的问题，往往可以用动态规划进行求解（这个结论大家先记下来，后面我们会有很多验证它的机会）。

Step1：递归思想分析问题

基于动态规划的思想来做题，我们首先要想到的思维工具就是“倒着分析问题”。“倒着分析问题”分两步走：

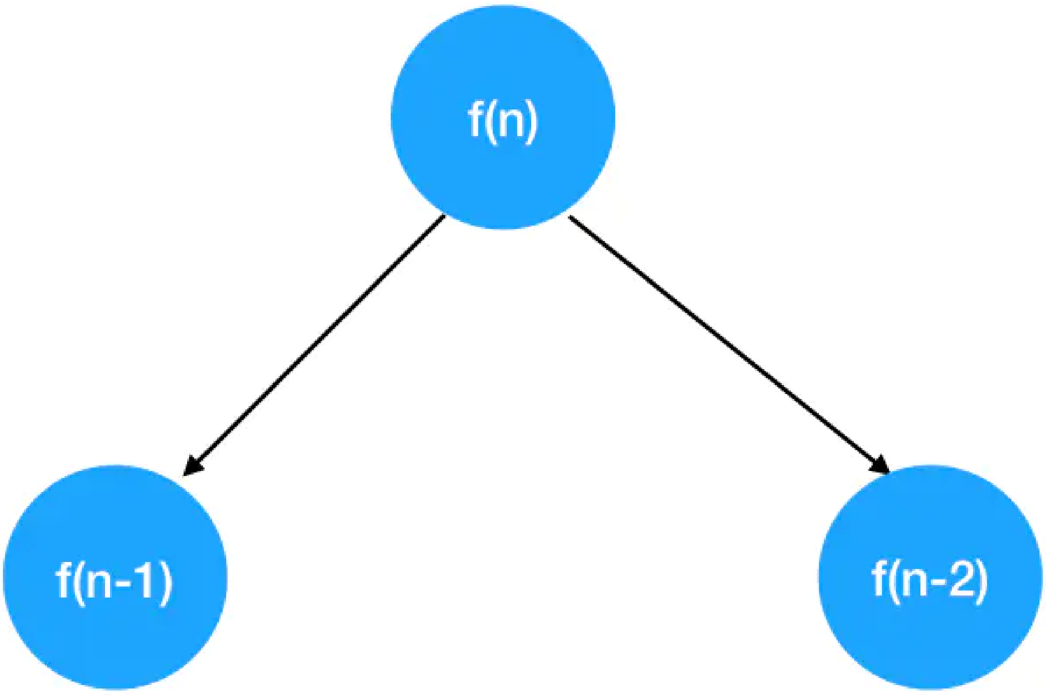
1. 定位到问题的终点
2. 站在终点这个视角，思考后退的可能性

在这道题里，“问题的终点”指的就是走到第 `n` 阶楼梯这个目标对应的路径数，我们把它记为 `f(n)` 。

那么站在第 `n` 阶楼梯这个视角， 有哪些后退的可能性呢？按照题目中的要求，一次只能后退 1 步或者 2 步。因此可以定位到从第 `n` 阶楼梯只能后退到第 `n-1` 或者第 `n-2` 阶。我们把抵达第 `n-1` 阶楼梯对应的路径数记为 `f(n-1)` ，把抵达第 `n-2` 阶楼梯对应的路径数记为 `f(n-2)` ，不难得出以下关系：

```
f(n) = f(n-1) + f(n-2)
```

这个关系用树形结构表示会更加形象



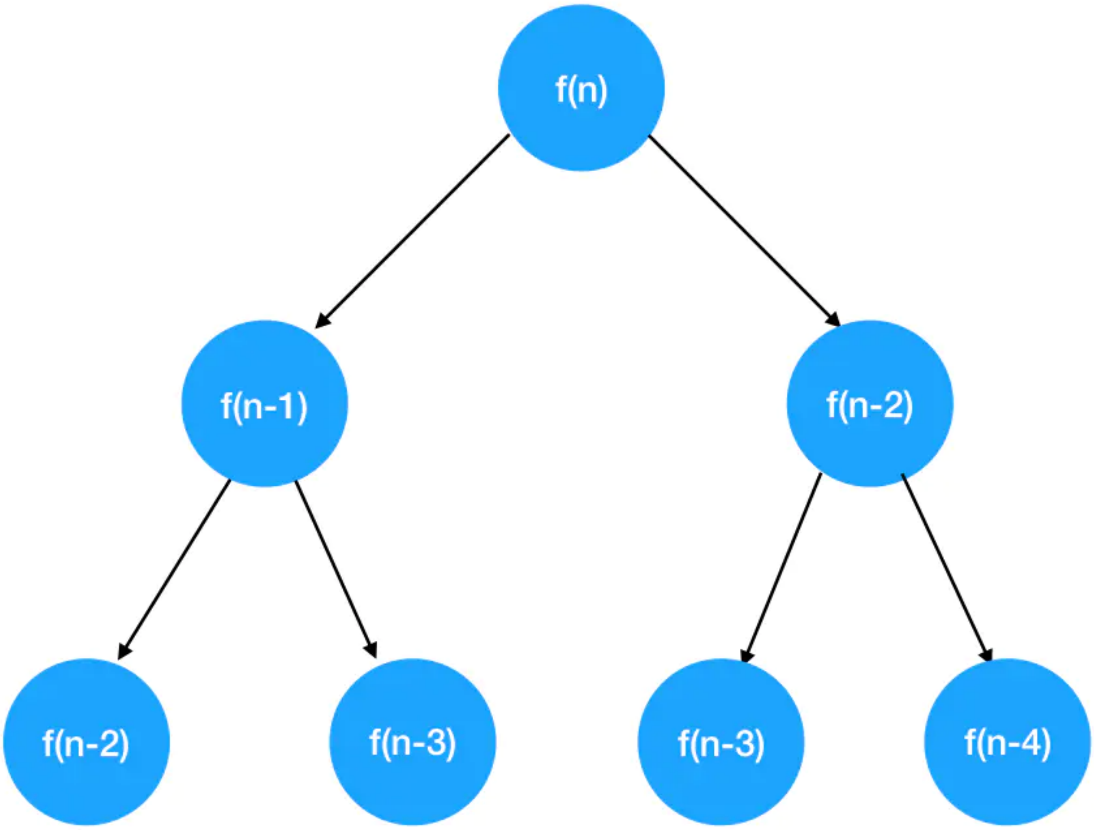
现在我们不难看出，要想求出 `f(n)` ，必须求出 `f(n-1)` 和 `f(n-2)` （我们假设 `n` 是一个大于 5 的数字）。
接下来站在第 `n-1` 阶台阶上，思考后退的姿势，也无非只能是退到 `n-1-1` 层台阶 或 `n-1-2` 层台阶上，所以 `f(n-1)` 和 `f(n-2)` 、 `f(n-3)` 间同样具有以下关系：

```
f(n-1) = f(n-2) + f(n-3)
```

同理， `f(n-2)` 也可以按照同样的规则进行拆分：

```
f(n-2) = f(n-3) + f(n-4)
```

现在我们的树结构渐渐丰满起来了：



随着拆分的进行，一定会有一个时刻，求解到了 `f(1)` 或 `f(2)` 。按照题设规则，第 1 阶楼梯只能走 1 步抵达，第 2 阶楼梯可以走 1 步或者走 2 步抵达，因此我们不难得出 `f(1)` 和 `f(2)` 的值：

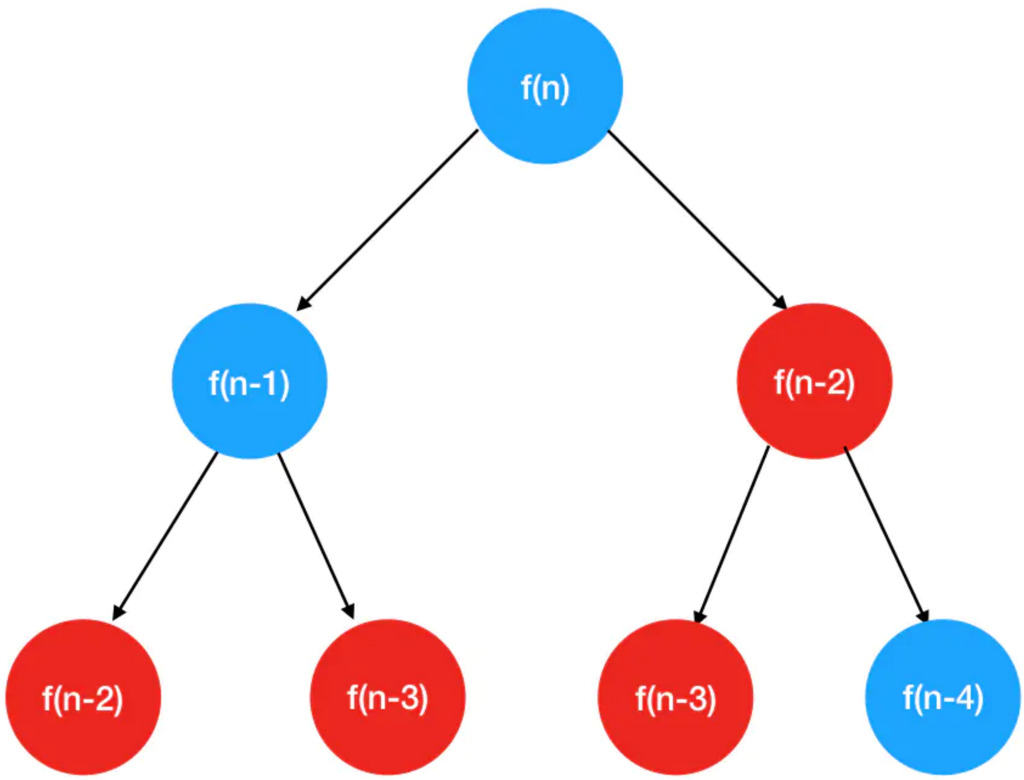
```
f(1) = 1
f(2) = 2
```

我们在学习递归与回溯思想的时候，曾经给大家强调过，遇到“树形思维模型”，就要想办法往递归上靠。这道题明显用到了树形思维模型，有着明确的重复内容(不断地按照 `f(n) = f(n-1) + f(n-2)` 的规则拆分），同时有着明确的边界条件(遇到 `f(1)` 或 `f(2)` 就可以返回了)，因此我们不难写出其对应的递归解法代码：

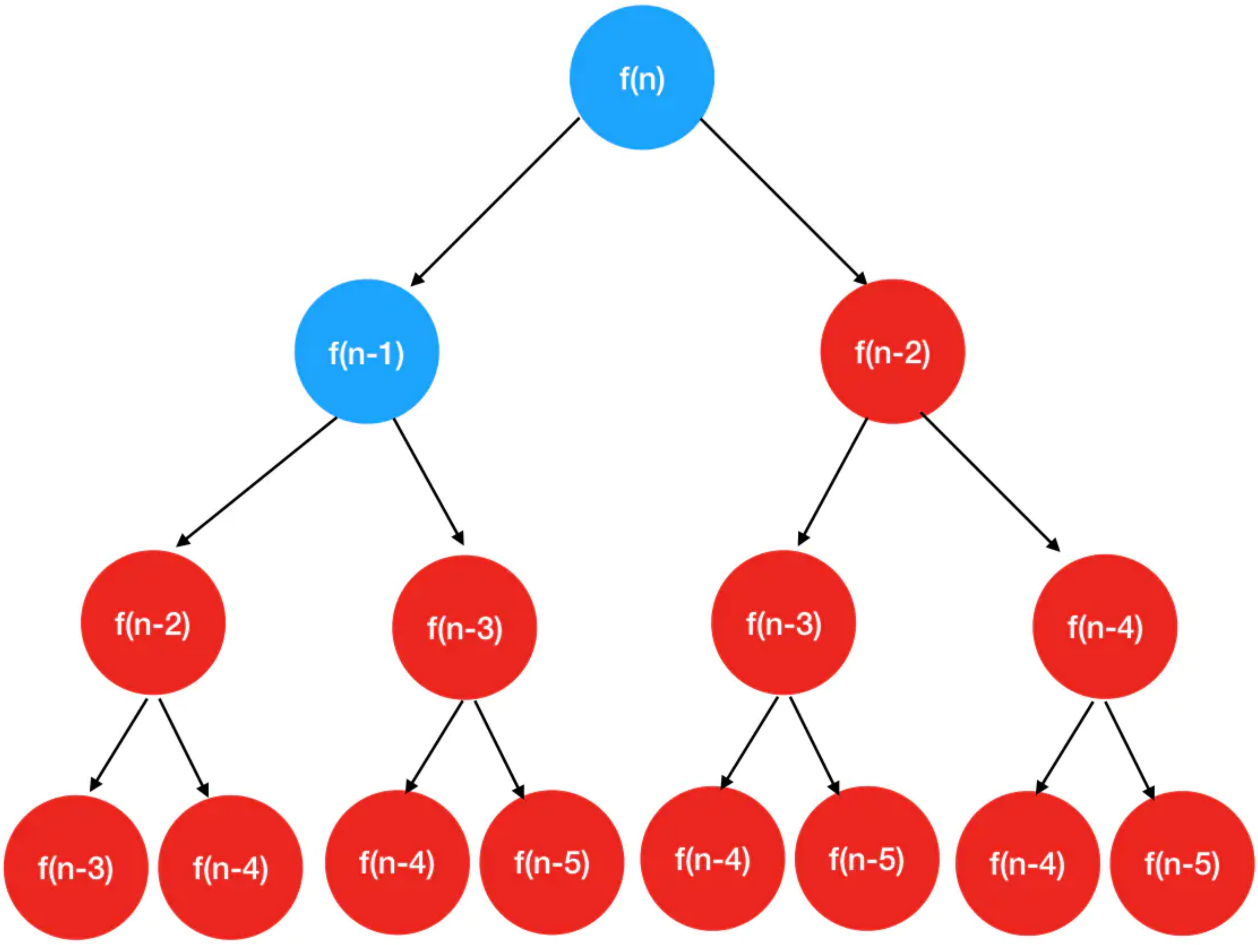
```
/**
* @param {number} n
* @return {number}
*/
const climbStairs = function(n) {
    // 处理递归边界
    if(n === 1) {
        return 1
    }
```

```
    if(n === 2){
        return 2
    }
    // 递归计算
    return climbStairs(n-1) + climbStairs(n-2)
};
```

但是这个解法问题比较大，丢进 OJ 会直接超时。我们一起来看看原因，回到我们上面这张树形结构图上来：



这次我把 `f(n-2)` 和 `f(n-3)` 给标红了。大家不难看出，我们在图中对 `f(n-2)` 和 `f(n-3)` 进行了重复的计算。事实上，随着我们递归层级的加深，这个重复的问题会越来越严重：



（图上标红的均为发生过重复计算的结点）

Step2：记忆化搜索来提效

重复计算带来了时间效率上的问题，要想解决这类问题，最直接的思路就是用空间换时间，也就是想办法记住之前已经求解过的结果。这里我们只需要定义一个数组：

```
const f = []
```

每计算出一个 `f(n)` 的值，都把它塞进 `f` 数组里。下次要用到这个值的时候，直接取出来就行了：

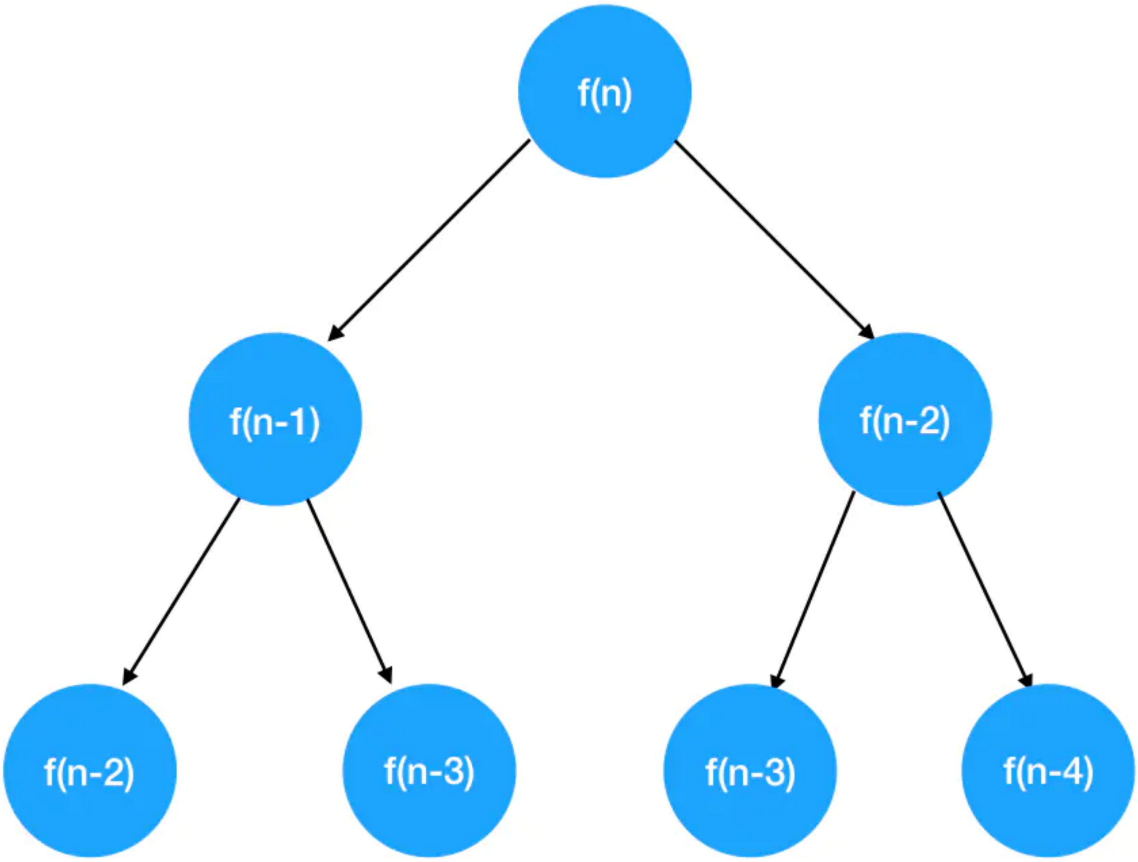
```
/**
* @param {number} n
* @return {number}
*/
// 定义记忆数组 f
const f = []
const climbStairs = function(n) {
```

```
    if(n==1) {
        return 1
    }
    if(n==2) {
        return 2
    }
    // 若f[n]不存在，则进行计算
    if(f[n]===undefined)  f[n] = climbStairs(n-1) + climbStairs(n-2)
    // 若f[n]已经求解过，直接返回
    return f[n]
  };
```

以上这种在递归的过程中，不断保存已经计算出的结果，从而避免重复计算的手法，叫做**记忆化搜索**。
对于一些实用派的面试官来说，“记忆化搜索”和“动态规划”没有区别，它们都能够以不错的效率帮我们达到同样的目的。这种情况下，上面这个答案就足够了。
但是还有一部分面试官，比较讲究，善于咀嚼理论概念。他会告诉你记忆化搜索和动态规划是两个东西，别想糊弄哥，哥要的是动态规划的解法。
行吧，就给你动态规划的解法。

Step3：记忆化搜索转化为动态规划

要想完成记忆化搜索与动态规划之间的转化，首先要清楚两者间的区别。
先说记忆化搜索，记忆化搜索可以理解为优化过后的递归。递归往往可以基于树形思维模型来做，以这道题为例：



我们基于树形思维模型来解题时，实际上是站在了一个比较大的未知数量级（也就是最终的那个 `n` ），来不断进行拆分，最终拆回较小的已知数量级（ `f(1)` 、 `f(2)` ）。这个过程是一个明显的**自顶向下**的过程。

动态规划则恰恰相反，是一个**自底向上**的过程。它要求我们站在**已知**的角度，通过定位**已知**和**未知**之间的关系，一步一步向前推导，进而求解出未知的值。
在这道题中，已知 `f(1)` 和 `f(2)` 的值，要求解未知的 `f(n)` ，我们唯一的抓手就是这个等价关系：

```
f(n) = f(n-1) + f(n-2)
```

以 `f(1)` 和 `f(2)` 为起点，不断求和，循环递增 `n` 的值，我们就能够求出 `f(n)` 了：

```
/**
* @param {number} n
* @return {number}
*/
const climbStairs = function(n) {
    // 初始化状态数组
    const f = [];
```

```
    // 初始化已知值
    f[1] = 1;
    f[2] = 2;
    // 动态更新每一层楼梯对应的结果
    for(let i = 3;i <= n;i++){
        f[i] = f[i-2] + f[i-1];
    }
    // 返回目标值
    return f[n];
};
```

以上便是这道题的动态规划解法。

从题解思路看动态规划

下面我们基于这个题解的过程，站在专业的角度来重新认识一下动态规划。

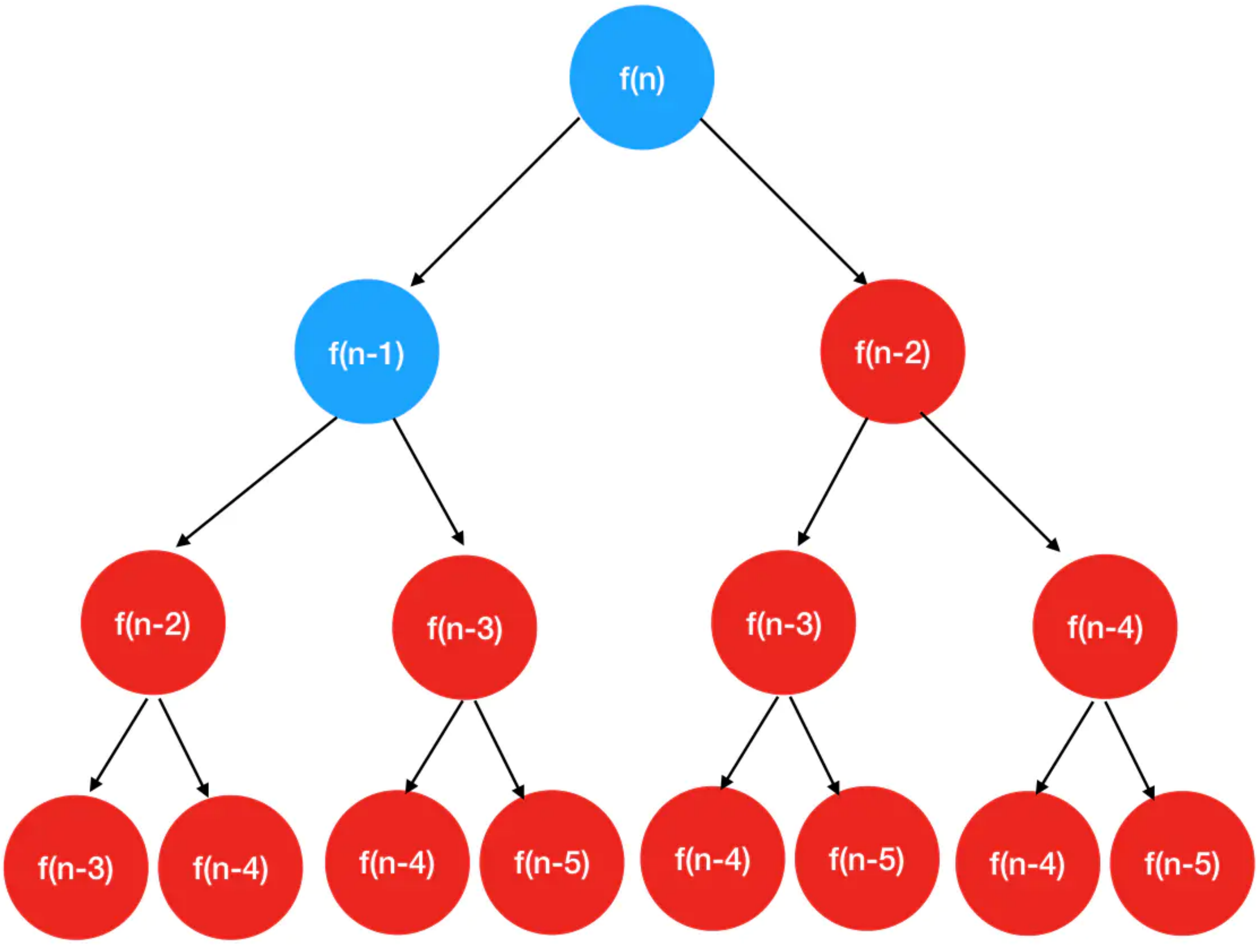
前面咱们在排序专题学过“分治”思想，提到了“子问题”这个概念。分治问题的核心思想是：把一个问题分解为相互独立的子问题，逐个解决子问题后，再组合子问题的答案，就得到了问题的最终解。

动态规划的思想和“分治”有点相似。不同之处在于，“分治”思想中，各个子问题之间是独立的：比如说归并排序中，子数组之间的排序并不互相影响。而动态规划划分出的子问题，往往是相互依赖、相互影响的。

什么样的题应该用动态规划来做？我们要抓以下两个关键特征：

- 最优子结构
- 重叠子问题

拿这道题的分析过程来说：



最优子结构，它指的是问题的最优解包含着子问题的最优解——不管前面的决策如何，此后的状态必须是基于当前状态（由上次决策产生）的最优决策。就这道题来说， `f(n)` 和 `f(n-1)` 、 `f(n-2)` 之间的关系印证了这一点（这玩意儿叫状态转移方程，大家记一下）。

重叠子问题，它指的是在递归的过程中，出现了反复计算的情况。就这道题来说，图上标红的一系列重复计算的结点印证了这一点。
因此，这道题适合用动态规划来做。

动态规划问题的分析技巧

现在，大家理解了动态规划的概念，明确了其"自底向上"的脑回路特征。但在实际做题过程中，"自底向上"分析问题往往不是最舒服的解题姿势，按照这个脑回路去想问题，容易拧巴。

什么姿势不拧巴?
递归!

你现在回过头去看看咱们前面递归+记忆化搜索那一通操作，你觉得拧巴吗？不拧巴！舒服不？相当舒服了——只要你掌握了递归与回溯，就不难分析出图上的树形思维模型和递归边界条件，**树形思维模型将帮助我们更迅速地定位到状态转移关系，边界条件往往对应的就是已知子问题的解**；基于树形思维模型，结合一下记忆化搜索，难么？不难，谁还不会初始化个记忆数组了呢；最后再把递归往迭代那么一转，答案不就有了么!

当然，咱们上面一通吹牛逼都只是为了衬托递归思路分析下来有多么爽，并不是说动态规划有多么简单。实际上，动态规划可复杂了，递归+记忆化搜索的思想只是帮助我们简化问题，但并不能送佛送到西。说到底，还是得靠我们自己。
动态规划到底复杂在什么地方，这里我先预告一下：

1. 状态转移方程不好确定
2. 已知的状态可能不明显
3. 递归转迭代，一部分同学可能不知道怎么转（这个就是纯粹的编程基础问题了，多写多练哈）

多的也没法说了，大家后面慢慢体会吧：）。
总结一下，对于动态规划，笔者建议大家优先选择这样的分析路径：

1. 递归思想明确树形思维模型：找到问题终点，思考倒退的姿势，往往可以帮助你更快速地明确**状态间的关系**
2. 结合记忆化搜索，明确**状态转移方程**
3. 递归代码转化为迭代表达（这一步不一定是必要的，1、2本身为思维路径，而并非代码实现。若你成长为熟手，2中分析出来的状态转移方程可以直接往循环里塞，根本不需要转换）。

“最值”型问题典范：如何优雅地找硬币

> 题目描述：给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 -1。

> 示例1：
> 输入: coins = [1, 2, 5], amount = 11
> 输出: 3
> 解释: 11 = 5 + 5 + 1

> 示例2：
> 输入: coins = [2], amount = 3
> 输出: -1

> 提示：最值问题是动态规划的常见对口题型，见到最值问题，应该想到动态规划

思路分析

现在思维工具已经给到大家了，详细的步骤我就不啰嗦了。我直接讲难点：这道题对于初学者来说，难的是状态转移方程的明确。

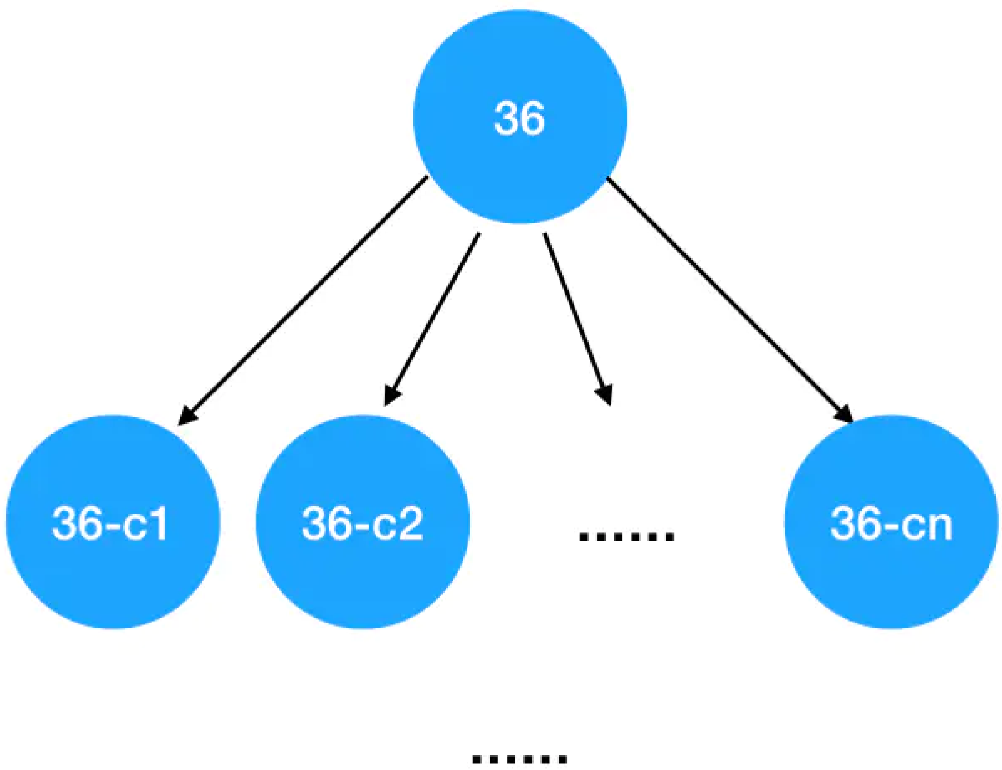
要明确状态转移关系，我们依然是借助“倒推”的思想：解决爬楼梯问题时，我们首先思考的是站在第 `n` 阶楼梯上的后退姿势。这道题也一样，我们需要思考的是站在 `amount` 这个组合结果上的“后退姿势”—— 我

们可以假装此时手里已经有了 36 美分，只是不清楚硬币的个数，把“如何凑到36”的问题转化为“如何从36减到0”的问题。

硬币的英文是 coin，因此我们这里用 c1、c2、c3......cn 分别来表示题目中给到我们的第 1-n 个硬币。现在我如果从 36 美分的总额中拿走一个硬币，那么有以下几种可能：

```
拿走 c1
拿走 c2
拿走 c3
......
拿走 cn
```

重复往前推导这个“拿走”的过程，我们可以得到以下的树形思维模型：



假如用 f(x）表示每一个总额数字对应的最少硬币数，那么我们可以得到以下的对应关系：

```
f(36) = Math.min(f(36-c1)+1,f(36-c2)+1,f(36-c3)+1......f(36-cn)+1)
```

这套对应关系，就是本题的状态转移方程。

找出了状态转移方程，我们接下来需要思考的是递归的边界条件：在什么情况下，我的“后退”（实际是做减法）可以停下来？这里需要考虑的是硬币总额为 0 的情况，这种情况对应的硬币个数毫无疑问也会是 0，因而不需要任何的回溯计算。由此我们就得到了一个已知的最基本的子问题的结果：

```
f[0] = 0
```

现在，明确了状态转移方程，明确了已知子问题的解，我们来写代码：

编码实现

```
const coinChange = function(coins, amount) {
    // 用于保存每个目标总额对应的最小硬币个数
    const f = []
    // 提前定义已知情况
    f[0] = 0
    // 遍历 [1, amount] 这个区间的硬币总额
    for(let i=1;i<=amount;i++) {
        // 求的是最小值，因此我们预设为无穷大，确保它一定会被更小的数更新
        f[i] = Infinity
        // 循环遍历每个可用硬币的面额
        for(let j=0;j<coins.length;j++) {
            // 若硬币面额小于目标总额，则问题成立
            if(i-coins[j]>=0) {
                // 状态转移方程
                f[i] = Math.min(f[i],f[i-coins[j]]+1)
            }
        }
    }
    // 若目标总额对应的解为无穷大，则意味着没有一个符合条件的硬币总数来更新它，本题无
    if(f[amount]===Infinity) {
        return -1
    }
    // 若有解，直接返回解的内容
    return f[amount]
};
```

小结

经过本节的讲解，相信大家已经对动态规划的概念和通用解题模板有了掌握。但仅仅依靠这些，可能还不足以支撑起你全部的底气——动态规划问题千姿百态，有着繁多的题型分支。在下一节，我们就将围绕这些分支中考察频率最高的一部分，提取出通用的解题模型，帮助大家更进一步。