

# 动态规划套路详解

---

下面通过对斐波那契数列和这道凑零钱问题详解动态规划。

动态规划算法似乎是一种很高深莫测的算法，你会在一些面试或算法书籍的高级技巧部分看到相关内容，什么状态转移方程，重叠子问题，最优子结构等高大上的词汇也可能让你望而却步。

而且，当你去看用动态规划解决某个问题的代码时，你会觉得这样解决问题竟然如此巧妙，但却难以理解，你可能惊讶于人家是怎么想到这种解法的。

实际上，动态规划是一种常见的「算法设计技巧」，并没有什么高深莫测，至于各种高大上的术语，那是吓唬别人用的，只要你亲自体验几把，这些名词的含义其实显而易见，再简单不过了。

至于为什么最终的解法看起来如此精妙，是因为动态规划遵循一套固定的流程：递归的暴力解法 -> 带备忘录的递归解法 -> 非递归的动态规划解法。这个过程是层层递进的解决问题的过程，你如果没有前面的铺垫，直接看最终的非递归动态规划解法，当然会觉得牛逼而不可及了。

当然，见的多了，思考多了，是可以一步写出非递归的动态规划解法的。任何技巧都需要练习，我们先遵循这个流程走，算法设计也就这些套路，除此之外，真的没啥高深的。

以下，先通过两个比较简单的例子：斐波那契和凑零钱问题，揭开动态规划的神秘面纱，描述上述三个流程。后续还会写几篇文章探讨如何使用动态规划技巧解决比较复杂的经典问题。

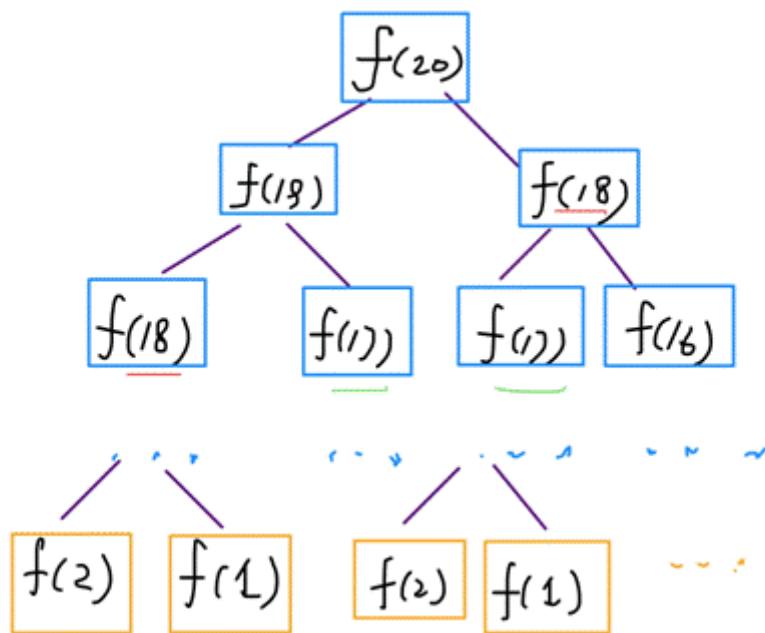
首先，第一个快被举烂了的例子，斐波那契数列。请读者不要嫌弃这个例子简单，因为简单的例子才能让你把精力充分集中在算法背后的通用思想和技巧上，而不会被那些隐晦的细节问题搞的莫名其妙。后续，困难的例子有的是。

## 步骤一、暴力的递归算法

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

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

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



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

递归算法的时间复杂度怎么计算？子问题个数乘以解决一个子问题需要的时间。

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

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

所以，这个算法的时间复杂度为  $O(2^n)$ ，指数级别，爆炸。

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

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

## 步骤二、带备忘录的递归解法

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

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

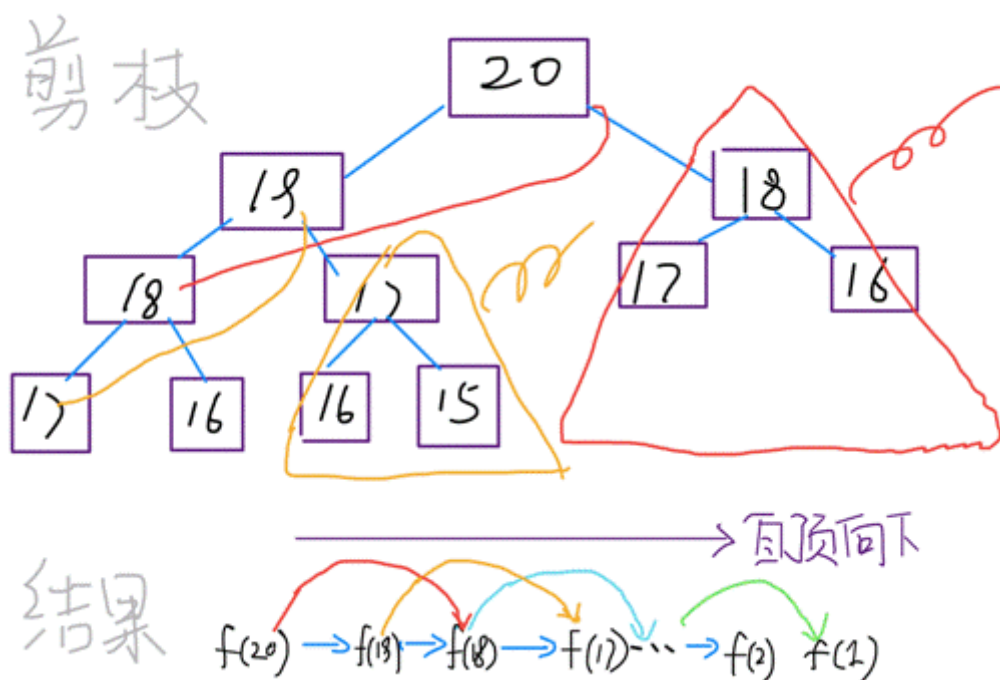
```

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

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

```

现在，画出递归树，你就知道「备忘录」到底做了什么



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

递归算法的时间复杂度怎么算？子问题个数乘以解决一个子问题需要的时间。

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

解决一个子问题的时间，同上，没有什么循环，时间为  $O(1)$ 。

所以，本算法的时间复杂度是  $O(n)$ 。比起暴力算法，是降维打击。

至此，带备忘录的递归解法的效率已经和动态规划一样了。实际上，这种解法和动态规划的思想已经差不多了，只不过这种方法叫做「自顶向下」，动态规划叫做「自底向上」。

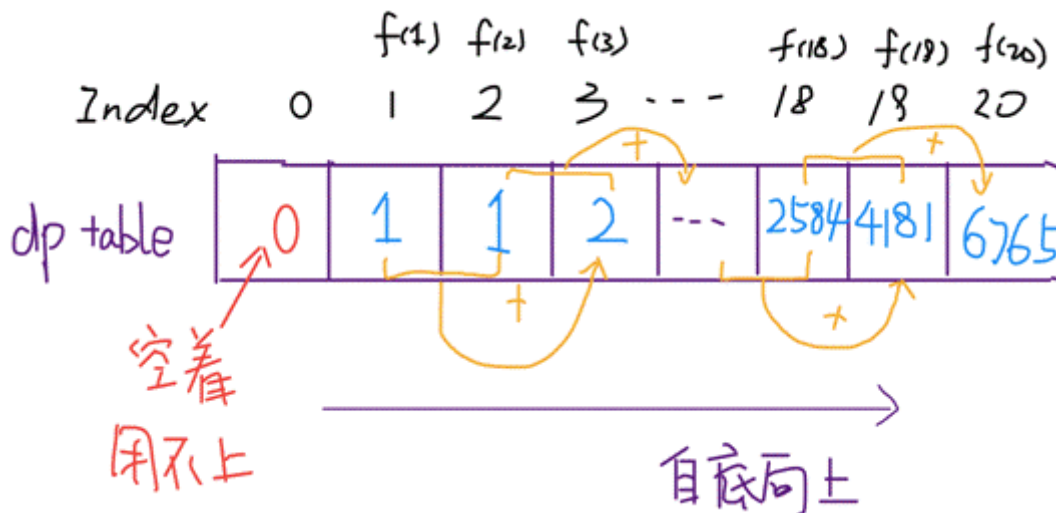
**「自顶向下」**：注意我们刚才画的递归树（或者说图），是从上向下延伸，都是从一个规模较大的原问题比如说  $f(20)$ ，向下逐渐分解规模，直到  $f(1)$  和  $f(2)$  触底，然后逐层返回答案，这就叫「自顶向下」。

「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。

### 步骤三、动态规划

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);  
    dp[1] = dp[2] = 1;  
    for (int i = 3; i <= N; i++)  
        dp[i] = dp[i - 1] + dp[i - 2];  
    return dp[N];  
}
```



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

这里,引出「动态转移方程」这个名词,实际上就是描述问题结构的数学形式:

# 斐波那契数列

$$f(n) = \begin{cases} 1, & \text{当 } n=1 \text{ 或 } n=2 \\ f(n-1) + f(n-2), & \text{其它} \end{cases} \quad (n > 0)$$

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

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

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

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

```
int fib(int n) {
    if (n < 2) return n;
    int prev = 0, curr = 1;
    for (int i = 0; i < n - 1; i++) {
        int sum = prev + curr;
        prev = curr;
        curr = sum;
    }
    return curr;
}
```

---

## 「最优子结构」

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，以上旨在演示算法设计螺旋上升的过程。当问题中要求求一个最优解或在代码中看到循环和 `max`、`min` 等函数时，十有八九，需要动态规划大显身手。



下面，看第二个例子，凑零钱问题，有了上面的详细铺垫，这个问题会很快解决。

题目：给你  $k$  种面值的硬币，面值分别为  $c_1, c_2 \dots c_k$ ，再给一个总金额  $n$ ，问你最少需要几枚硬币凑出这个金额，如果不可能凑出，则回答  $-1$ 。

比如说， $k = 3$ ，面值分别为  $1, 2, 5$ ，总金额  $n = 11$ ，那么最少需要  $3$  枚硬币，即  $11 = 5 + 5 + 1$ 。下面走流程。

## 一、暴力解法

首先是最困难的一步，写出状态转移方程，这个问题比较好写：

凑零钱 (用  $f$  代替名称 coinChange)

$$f(n) = \begin{cases} 0 & , \text{当 } n=0 \\ 1 + \min \{ f(n - c_i) \mid i \in [1, k] \} \end{cases}$$

$n$  代表目标金额， $c_i$  为硬币面值。

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，显然子问题之间没有相互制约，而是互相独立的。所以这个状态转移方程是可以得到正确答案的。

之后就没啥难点了，按照方程写暴力递归算法即可。

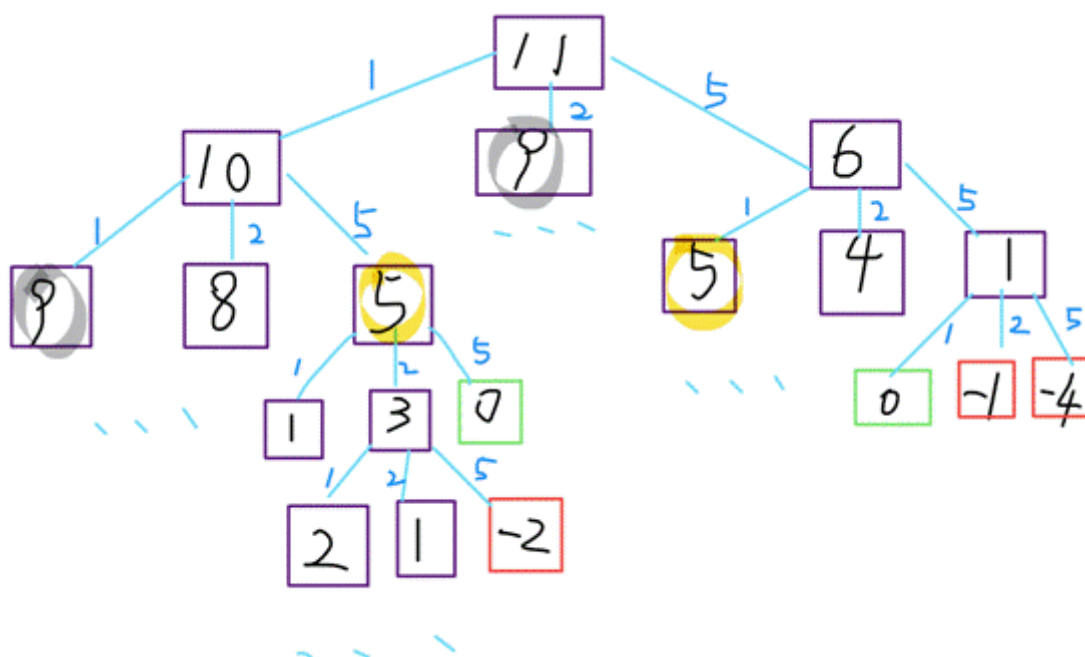
```
int coinChange(vector<int>& coins, int amount) {
    if (amount == 0) return 0;
    int ans = INT_MAX;
    for (int coin : coins) {
```

```

// 金额不可达
if (amount - coin < 0) continue;
int subProb = coinChange(coins, amount - coin);
// 子问题无解
if (subProb == -1) continue;
ans = min(ans, subProb + 1);
}
return ans == INT_MAX ? -1 : ans;
}

```

画出递归树：



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  $O(k \cdot n^k)$ ，指数级别。

## 二、带备忘录的递归算法

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  $O(kn)$

```

int coinChange(vector<int>& coins, int amount) {
// 备忘录初始化为 -2
vector<int> memo(amount + 1, -2);
return helper(coins, amount, memo);
}

```

```

int helper(vector<int>& coins, int amount, vector<int>& memo) {
if (amount == 0) return 0;

```

```

if (memo[amount] != -2) return memo[amount];
int ans = INT_MAX;
for (int coin : coins) {
    // 金额不可达
    if (amount - coin < 0) continue;
    int subProb = helper(coins, amount - coin, memo);
    // 子问题无解
    if (subProb == -1) continue;
    ans = min(ans, subProb + 1);
}
// 记录本轮答案
memo[amount] = (ans == INT_MAX) ? -1 : ans;
return memo[amount];
}

```

### 三.动态规划

```

int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int i = 1; i <= amount; i++) {
        for (int coin : coins)
            if (coin <= i)
                dp[i] = min(dp[i], dp[i - coin] + 1);
    }
    return dp[amount] > amount ? -1 : dp[amount];
}

```

