动态规划到底在讲什么?

今日目标:

1：能够说出贪心的特点，局限性以及贪心失效后怎么办

2：能够说出记忆化递归的求解思路

3：能够说出动态规划问题的特征，什么是最优子结构

4：能够说出如何定义状态转移方程

5：能够说出动态规划的求解思路和代码模板

6：完成零钱兑换和0-1背包问题的求解

1、初识动态规划

动态规划（英语：Dynamic programming，简称DP）是一种在数学、管理科学、计算机科学、经济学和生物信息学中使用的，通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。

参考维基百科对动态规划的定义: In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure.

核心观点是: 通过以递归的方式将其分解为更简单的子问题来简化复杂的问题,如果可以通过将问题分解为子问题然后递归地找到子问题的最优解来最佳地解决问题，则可以说它具有最优子结构。

动态规划一般用于去解决**最优化系列**的问题。

1.1、从贪心说起

1.1.1、贪心的特点

所谓贪心算法，就是指它的每一步计算作出的都是在当前看起来最好的选择，也就是说它所作出的选择只是在某种意义上的局部最优选择，并不从整体最优考虑。在这里把这两种选择的思路称作**局部最优解**和**整体最优解**。

像之前讲到的：322. 零钱兑换

贪心的思路就是每次都用最大的面额，然后用次大的，.....

```
class Solution {
    //贪心思路:每次都找局部最优解
    public int coinChange(int[] coins, int amount) {
        //定义组合总数
        int count = 0;
        Arrays.sort(coins);
        for (int i=coins.length-1;i>=0;i--) {
            //计算该面额能用几次
```

```
            int currentCount = amount / coins[i];
            if (currentCount==0) {
                continue;
            }
            //该面额用完后剩余的面额
            amount = amount % (currentCount * coins[i]) ;
            //统计所用硬币个数
            count+= currentCount;
            if (amount ==0) {
                return count;
            }
        }
        return -1;
    }
}
```

1.1.2、贪心的局限性

从零钱兑换的例子中可以看出，如果coins=[10,9,1],amount=18，贪心算法并不能得到最终的最优解，

这就是贪心算法所谓的局部最优导致的问题，因为我们每一步都尽量多地使用面值最大的硬币，因为这样数量肯定最小，但是有的时候我们就进入了死胡同，就好比上面这个例子。

所谓**局部最优**，就是只考虑“当前”的最大利益，既不向前多看一步，也不向后多看一步，导致每次都只用当前阶段的最优解。那么如果纯粹采用这种策略我们就永远无法达到**整体最优**，也就无法求得题目的答案了。

虽然纯粹的贪心算法作用有限，但是这种求解**局部最优**的思路在方向上肯定是对的，毕竟所谓的**整体最优**肯定是从很多个局部最优中选择出来的，因此所有最优化问题的基础都是贪心算法。

1.1.3、贪心失效后怎么办

在硬币找零问题中，当贪心失效后我们在贪心的基础上加入了失败后的回溯，稍微牺牲一点当前利益，仅仅是希望通过下一个硬币面值的局部最优达到最终可行的整体最优。

```
class Solution {
    public void sort (int[] coins) {
        //为了方便先Arrays.sort,然后倒置
        Arrays.sort(coins);
        int n = coins.length-1;
        int temp;
        for (int i=0;i<=(n-1)/2;i++) {
            temp = coins[i];
            coins[i] = coins[n-i];
            coins[n-i] = temp;
        }
    }
    public int coinChange(int[] coins, int amount) {
        //对coins从大到小排序
        sort(coins);
        dfs(coins,amount,0,0);
        if (minCount == Integer.MAX_VALUE) {
            return -1;
```

```
        }
        return minCount;
    }

    //全局最优解使用的最少硬币数量
    int minCount = Integer.MAX_VALUE;
    public void dfs (int[] coins,int amount,int selectedCount,int
startIndex) {
        //如果amount==0则终止
        if (amount==0) {
            if (selectedCount < minCount) {
                minCount = selectedCount;
            }
            return;
        }
        if (startIndex >= coins.length) {
            return;
        }
        //计算当前硬币最多能使用多少次
        int maxCount = amount / coins[startIndex];
        //从选择列表[0,maxCount]中选择,然后递归到下一个硬币
        for (int i=maxCount;i>=0 && i+selectedCount < minCount ;i--) {
            //选择i个该硬币后剩余的额度
            int resAmount = amount - i*coins[startIndex];
            //drill down
            dfs(coins,resAmount,selectedCount+i,startIndex+1);

        }
    }
}
```

> 整体的思路是：穷举所有组合情况，过程中仍然每次找最大的以期望更快找到最优解，并且通过剪枝减少穷举次数，最终在所有满足条件的组合中找到最优解.

所有贪心的思路就是我们最优化求解的根本思想，所有的方法只不过是针对贪心思路的改进和优化而已。回溯(递归)解决的是正确性问题，而动态规划则是解决时间复杂度的问题。

1.1.4、从最优化问题到递归

为什么最优化问题跟递归有关系?两个方面

1: 在求解最优化问题的时候，我们经常会用到**回溯**这个策略。在硬币找零这个问题里，具体说就是如果遇到已经无法求解的组合，那么我们就往回退一步，修改上一个面值的硬币数量，然后再尝试新的组合。递归这种形式，正是赋予了回溯这种可以回退一步的能力：它通过堆栈保存了上一步的当前状态。因此，如果想要用回溯的策略来解决问题，那么递归应该是你的首选方法。所以说，回溯在最优化问题中有多么重要，递归也就有多么重要。

2: 枚举与递归, 最优组合的求解策略,如果想得到最优组合，那么最简单直接的方法肯定就是**枚举**。枚举就是直接求出所有满足条件的组合，然后看看这些组合是否能得到最大值或者最小值。

枚举本身很简单，就是把所有组合都遍历一遍即可。问题就是如何得到这些组合呢？这就需要我们通过一些策略来生成所有满足条件的组合, 而**递归**正是得到这些组合的方法。最优化问题使用递归来处理是非常清晰的，递归是搜索组合的一种非常直观的思路。

比如:322. 零钱兑换,直接递归枚举搜索所有组合

1、是否有重复子问题：能够兑换成功其实就是amount变化到0的过程，中间的每一个过程我们都是在coins中选择硬币来完成，所以具备重复性

2、递推公式如何表示

```
f(x)= min( f(x-c) +1  ),x>0,f(x-c)!=-1,x代表amount值,c代表coins中的硬币金额
其中f(0)=0,
如果x<0则f(x) =-1;
```

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        return dfs(coins,amount);
    }

    //返回当前面额amount需要使用的硬币个数
    public int dfs(int[] coins,int amount) {
        //terminal
        if (amount==0) {
            return 0;
        }
        //定义面额组合成功所需的最少硬币数量
        int minCount = Integer.MAX_VALUE;

        for (int i=0;i<coins.length;i++) {
            //当前硬币面额大于剩余额度,当前硬币不可用
            if (coins[i]  > amount ) {
                continue;
            }
            //使用一次当前硬币,再接着选
            int rearCount = dfs(coins,amount-coins[i]);//返回后面组合的硬币数量
            if (rearCount ==-1 ) {
               //说明这种组合不可行,跳过
               continue;
            }

            //组合可行,记录当前组合使用的硬币数量
            int currentCount = rearCount + 1;
            if (currentCount < minCount) {
                minCount = currentCount;
            }
        }
        //如果都没有可用的组合
        if (minCount == Integer.MAX_VALUE) {
            return -1;
        }
        return minCount;
    }
}
```

当然,提交未AC,原因是超时!

当然这跟递归过程中出现的一些问题有关系,我们现在的递归属于朴素递归,说白了就是暴力递归.

1.2、记忆化递归（备忘录）

1.2.1、朴素递归的弊端

1、代码的可读性和可调试性差

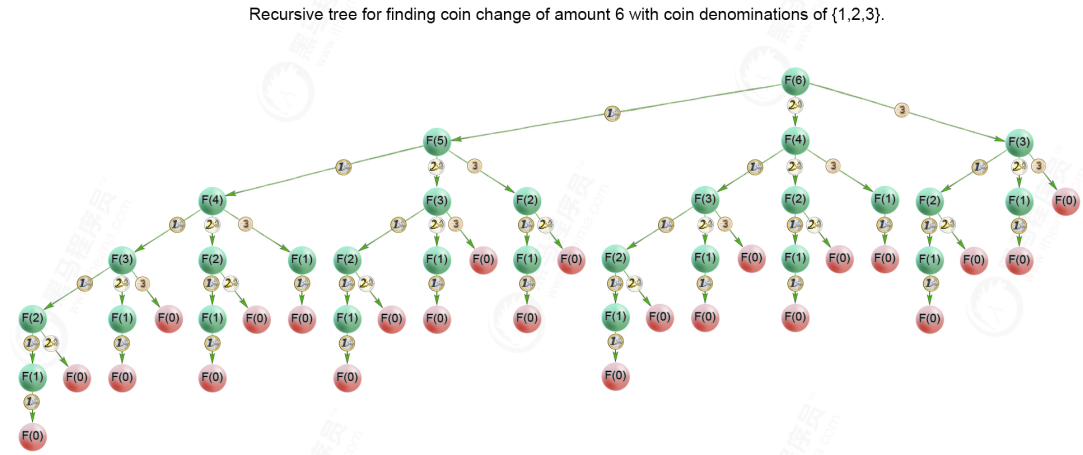
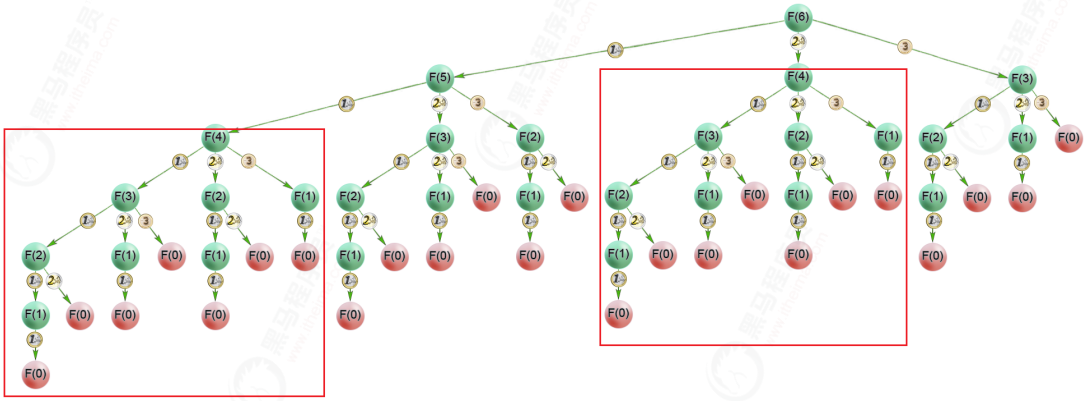
虽然递归在数学意义上非常直观，但是如果问题过于复杂，画递归状态树的时候，如果分支极多，那么很多人就很难继续在脑海中模拟出整个求解过程了，当然，我们并不推荐用人肉递归的方式在脑海中模拟整个过程。

另外，一旦程序出现 bug，当你想尝试去调试的时候，就会发现这样的代码几乎没有调试的可能性，而且这种问题在数据规模很大的情况下尤为明显。

2、性能低下

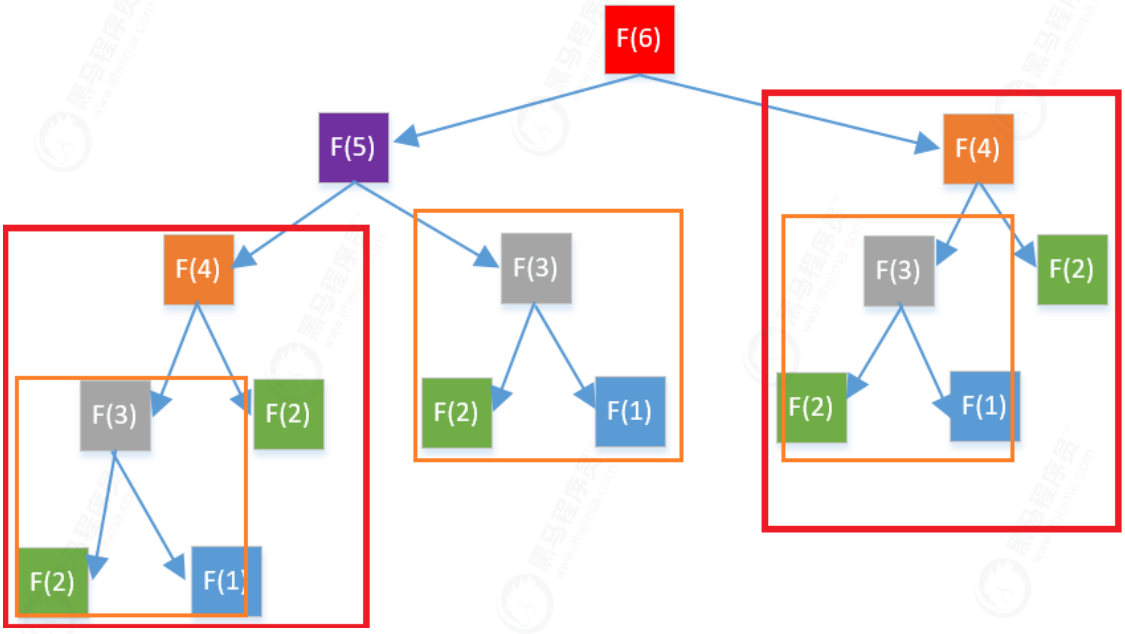
使用递归穷举所有组合结果一般性能很低下，时间复杂度有时候能达到指数级别，如果数据量特别大带来的性能问题就是灾难级别的。

而导致递归性能暴跌的一般主要原因是因为在递归的过程中会产生大量的**重复计算**，也就是一个问题存在**重叠子问题**。

> 比如对于：322. 零钱兑换就存在重叠子问题，该题目画出的递归状态树如下
>
> 
>
>
> 我们发现其中的F(1),F(2),F(3)均被计算了很多次，**另外对于F(4)所在的子树，他们的中间求解过程是完全一样的，我们称之为重叠子问题。**
>
> 
>
>
> 还比如之前作过的一道：剑指 Offer 10- I. 斐波那契数列
>
> 按照朴素的递归解法

```java
class Solution {
public int fib(int n) {
    //终止条件
    if (n < 2) {
        return n;
    }else {
        return fib(n-1) % 1000000007 + fib(n-2) % 1000000007;
    }
}
}
```

如果我们画出其递归的状态树，如下：



我们也能发现，这其中不仅有很多重复计算，更是有很多的**重叠子问题**。

其实动态规划的核心思想就是通过先解决子问题的最优解来推导原问题的最优解。

那如何优化朴素的递归呢?

1.2.2、如何剪枝与优化

1、剪枝

把一些明显不符合最终结果的分支提前剪掉，分支数量减少了，递归的效率自然就高了，这就是所谓的剪枝优化。这个一般通过在递归过程中添加判断条件来实现。

2、参考贪心

递归过程中可以仿照贪心，从整个搜索策略上来调整，先考虑最优解的情况，如果不行再找次优解。

> 比如322. 零钱兑换中我们在递归搜索所有解的过程中每次都先找面额最大的，并且每次都选最大数量，如果递归搜索不匹配，回溯后我们再减少数量再进行搜索。

3、**记忆化递归（备忘录）**

由于递归的性质决定了它本身是**自顶向下（自上而下）**的，也就是一个大问题被逐层分解成子问题，直到终止的子问题，然后随着子问题的不断解决最终使得原问题得到解决。在逐层递归的过程中可能会产生一些重复的分支，也就是前面讲的重复计算和重叠子问题。

为了消除重叠子问题，即消灭重复计算的过程。我们可以创建一个**备忘录（memorization）（即缓存）**，在每次计算出某个子问题的答案后，将这个临时的中间结果记录到备忘录里，然后再返回。

接着，每当遇到一个子问题时，我们不是按照原有的思路开始对子问题进行递归求解，而是先去这个备忘录中查询一下。如果发现之前已经解决过这个子问题了，那么就直接把答案取出来复用，没有必要再递归下去耗时的计算了。

对于备忘录，你可以考虑使用以下两种数据结构：

1、数组（Array），通常对于简单的问题来说，使用一维数组就足够了。在后续的课程中，也许要用到更为复杂的状态存储过程，届时要使用更高维度（二维甚至三维）的数组来存储状态。

2、哈希表（Hash table），如果存储的状态不能直接通过索引找到需要的值（比如斐波那契数列问题，你就可以直接通过数组的索引确定其对应子问题的解是否存在，如果存在你就拿出来直接使用），比如你使用了更高级的数据结构而非简单的数字索引，那么你还可以考虑使用哈希表，即字典来存储中间状态，来避免重复计算的问题。

使用记忆化递归来优化322. 零钱兑换朴素递归的代码如下

```java
class Solution {
    public int coinChange(int[] coins, int amount) {
        //有[0,amount]共amount+1个子问题,其中amount[0]=0;
        int[] memo = new int[amount+1];
        Arrays.fill(memo,-2);//如果存的是-1代表组合不可用
        return dfs(coins,amount,memo);
    }

    //返回当前面额amount需要使用的硬币个数
    public int dfs(int[] coins,int amount,int[] memo) {
        //先从缓存中拿子问题的解
        if (memo[amount] != -2) {
            return memo[amount];
        }

        //terminal
        if (amount==0) {
            memo[amount]=0;
            return 0;
        }
        //定义面额组合成功所需的最少硬币数量
        int minCount = Integer.MAX_VALUE;

        for (int i=0;i<coins.length;i++) {
            //当前硬币面额大于剩余额度,当前硬币不可用
            if (coins[i]  > amount ) {
                continue;
            }
            //使用一次当前硬币,再接着选
            int rearCount = dfs(coins,amount-coins[i],memo);//返回后面组合的硬币数量
            if (rearCount ==-1 ) {
               //说明这种组合不可行,跳过
               continue;
            }

            //组合可行,记录当前组合使用的硬币数量
```

```
            int currentCount = rearCount + 1;
            if (currentCount < minCount) {
                minCount = currentCount;
            }
        }
        //如果都没有可用的组合
        if (minCount == Integer.MAX_VALUE) {
            memo[amount] = -1;
            return -1;
        }
        //记录该子问题的解
        memo[amount] = minCount;
        return minCount;
    }
}
```

通过备忘录，我们避免了重复计算，即避免重复计算那些已经计算过的子问题。重叠子问题处理模式。

那这种利用重叠子问题的缓存来提升速度的方法是不是万灵药呢？有一句老话，叫计算机中不存在“银弹”，也就是说没有任何一种方法能够解决世界上的所有问题，通过备忘录的思想来处理重叠子问题的方法亦是如此。

1、因为有些问题是不存在重叠子问题的，比如八皇后问题。既然没有重叠子问题，那么通过备忘录来对其优化加速，又从何谈起呢？

2、有些问题虽然看起来像包含“重叠子问题”的子问题，但是这类子问题可能具有后效性，但我们追求的是**无后效性**。所谓无后效性，指的是在通过 A 阶段的子问题推导 B 阶段的子问题的时候，我们不需要回过头去再根据 B 阶段的子问题重新推导 A 阶段的子问题，即子问题之间的依赖是单向性的。所以说，如果一个问题可以通过重叠子问题缓存进行优化，那么它肯定都能被画成一棵树。希望你能牢记这些限制，不然可能抓破头皮都没法解决问题，最后陷入死胡同。

1.2.3、总结

备忘录的思想极为重要，特别是当求解的问题包含重叠子问题时，只要面试的问题包含重复计算，你就应该考虑使用备忘录来对算法时间复杂度进行简化，具体来说，备忘录解法可以归纳为：

1、用数组或哈希表来缓存已解的子问题答案，并使用自顶向下的递归顺序递归数据；

2、基于递归实现，与暴力递归的区别在于备忘录为每个求解过的子问题建立了备忘录（缓存）；

3、为每个子问题的初始记录存入一个特殊的值，表示该子问题尚未求解（像求解硬币找零问题中的初始值-2）；

4、在求解过程中，从备忘录中查询。如果未找到或是特殊值，表示未求解；否则取出该子问题的答案，直接返回。

与此同时，在求解最优解问题的时候，画出基本的递归树结构，能极大地降低问题的难度。因此在解决此类问题的时候要尝试使用这个方法。

含有备忘录的递归算法已经与动态规划思想十分相似了，从效率上说也是如此。没错！备忘录让我们实现了对算法时间复杂度的“降维打击”（一般会从指数级别到线性级别），这与贪心算法到递归的进步程度不同，这是真正意义上的动态规划思维：我们考虑了整体最优；在计算的过程中保存计算当中的状态，并在后续的计算中复用之前保存的状态。记住使用备忘录来优化你的算法时间复杂度，它是提高算法效率的高级手段。我们距真正的动态规划咫尺之遥，除了重叠子问题，我们还需要理解动态规划中的**最优子结构**，**状态转移方程。**

1.3、动态规划解决硬币找零

对于322. 零钱兑换真正的动态规划解法与前面讲到的备忘录记忆化递归到底有什么区别呢?

1.3.1、动归问题的特征

我们曾不止一次提到**重叠子问**题，并在上一节对其做了深入探讨。其实，重叠子问题是考虑一个问题是否为动态规划问题的先决条件，除此之外，我还提到了无后效性，所以总结下来动态规划问题一定具备以下三个特征：

1、**重叠子问题**：在穷举的过程中（比如通过递归），存在重复计算的现象；

2、**无后效性**：子问题之间的依赖是单向性的，某阶段状态一旦确定，就不受后续决策的影响

3、**最优子结构**：子问题之间必须相互独立，或者说后续的计算可以通过前面的状态推导出来（能通过子问题的最优解推导出原问题的最优解）

1.3.2、什么是最优子结构?

1、什么叫子问题之间必须相互独立?

> 场景1：
>
> 苹果原价：8元/斤，梨原价：7元/斤；但是现在做促销，苹果：5元/斤，梨：4元/斤。
>
> 问题1：以最低的价格购买5斤苹果，3斤梨。
>
> 方案1：显然两种水果的促销价格相互独立，互不影响，我们直接购买即可，都能享受到两种水果的促销价格。
>
> 场景2：苹果原价：8元/斤，梨原价：7元/斤；但是现在做促销，苹果：5元/斤，梨：4元/斤，但是两种水果的折扣不能同时享用。
>
> 问题2：以最低的价格购买5斤苹果，3斤梨。
>
> 方案2：此时不能同时以最低的苹果价格和最低的香蕉价格享受到最低折扣了。为了使价格最低，我们选择苹果按促销价，梨用原价。
>
> 总结：在方案二中，因为子问题并不独立，苹果和香蕉的折扣价格无法同时达到最优，这时最优子结构被破坏

2、什么叫做后续的计算可以通过前面的状态推导?

即：能通过子问题的最优解推导出原问题的最优解

> 还是在刚刚的场景1中：如果你准备购买了 5 斤折扣苹果，那么这个价格（即子问题）就被确定了，继续在购物车追加 3 斤折扣香蕉的订单，只需要在刚才的价格上追加折扣香蕉的价格，就是最低的总价格（即答案）

或者再举一个场景

> A : "2+2+2+2+2=? 请问这个等式的值是多少? "

> B : "计算ing 。。。。。。结果为10 "
>
> A : "那如果在等式左边写上 1+ ，此时等式的值为多少? "
>
> B : "quickly 结果为11 "
>
> A : "你怎么这么快就知道答案了"
>
> A : "只要在10的基础上加1就行了 "
>
> A : "所以你不用重新计算因为你记住了第一个等式的值为10 ,动态规划算法也可以说是'记住求过的解来节省时间'

最后，让我们回到硬币找零的问题上来，它满足最优子结构吗?

答案是满足，假设有两种面值的硬币coins=[3,5]，目标兑换金额为 amount=11。原问题是求这种情况下求最少兑换的硬币数?

如果你知道凑出 amount=6 最少硬币数为 “2”（注意，这是一个子问题），那么你只需要再加 “1” 枚面值为 5 的硬币就可以得到原问题的答案，即 2 + 1 = 3。

原问题并没有限定硬币数量，你应该可以看出这些子问题之间没有互相制约的情况，它们之间是互相独立的。因此，硬币找零问题满足最优子结构，可以使用动态规划思想来进行求解。

1.3.3、状态转移方程

当动态规划最终落到实处，就是一个**状态转移方程**，这是一个吓唬人的名词。没关系，其实我们已经具备了写出这个方程的所有工具。现在我们一起看看如何写出这个状态转移方程。

1、首先，任何穷举算法（包括递归在内）都需要一个**终止条件。**那么对于硬币找零问题来说，终止条件是什么呢？当剩余的金额为 0 时结束穷举，因为这时不需要任何硬币就已经凑出目标金额了。在动态规划中，我们将其**称之为初始化状态。**

2、找出子问题与原问题之间会发生变化的变量。原问题指定了硬币的面值，同时没有限定硬币的数量，因此它们俩无法作为“变量”。唯独剩余需要兑换的金额amount是变化的，因此在这个题目中，唯一的变量是目标兑换金额amount。在动态规划中，我们将其称之为**状态/状态参数。**同时，你应该注意到了，这个状态在不断逼近初始化状态。而这个不断逼近的过程，叫做**状态转移。**

3、当我们确定了状态，那么什么操作会改变状态，并让它不断逼近初始化状态呢？每当我们挑一枚硬币，用来凑零钱，就会改变状态。在动态规划中，我们将其称之为**决策/选择。**

总之：构造了一个初始化状态 -> 确定状态参数 -> 设计决策的思路。我们可以写出这个状态转移方程了。通常情况下，状态转移方程的参数就是状态转移过程中的变量，即状态参数。而函数的返回值就是答案，在该题目里是最少兑换的硬币数。

> 我在这里先用递归形式（伪代码形式）描述一下状态转移的过程

```
DP(coins, amount) {
res = MAX
for c in coins
// 作出决策，找到需要硬币最少的那个结果
res = min(res, 1 + DP(coins, amount-c)) // 递归调用

if res == MAX
return -1

return res
}
```

按这个逻辑，状态转移方程写出来是这样的：

$$DP(n) = \begin{cases} 0, n=0 \\ -1, n<0 \\ \min\{1+DP(n-c)\}，c代表coins中硬币面额 \end{cases}$$

```
f(n) = min ( 1+ dp(n-c) ),c是coins中硬币面额,n代表兑换的金额amount
其中：f(0) = 0,
如果n<0,则f(n)=-1
```

不知你是否发现，这个状态转移方程与刚开始定义递归的递推公式是一样的。

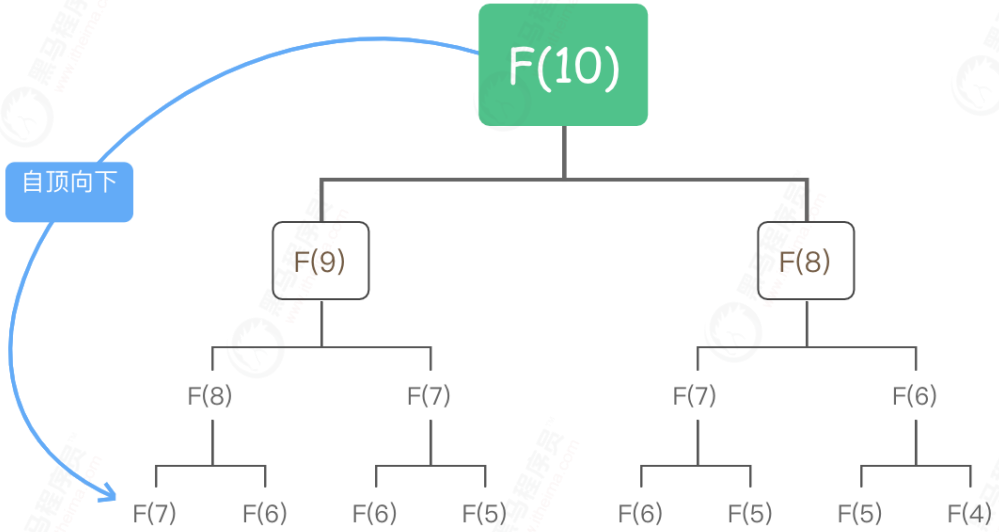
在上一节为了优化递归中的重叠子问题，设计了一个缓存用于存储重叠子问题的结果，避免重复计算已经计算过的子问题。而这个缓存其实就是存储了动态规划里的状态信息。

因此，带备忘录的递归算法与你现在看到的动态规划解法之间，有着密不可分的关系。它们要解决的核心问题是一样的，即消除重叠子问题的重复计算。

事实上，**带备忘录的递归算法也是一种动态规划解法。**但是，我们并不把这种方法作为动态规划面试题的常规解法，为什么呢？这是递归带来的固有问题。

1、首先，从理论上说，虽然带备忘录的递归算法与动态规划解法的时间复杂度是相同规模的，但在计算机编程的世界里，递归是依赖于函数调用的，而每一次函数调用的代价非常高昂。递归调用是需要基于系统堆栈才能实现的。而对于基于堆栈的函数调用来说，在每一次调用的时候都会发生环境变量的保存和还原，因此会带来比较高的额外时间成本。这是无法通过时间复杂度分析直接表现出来的，表现在空间复杂度上，递归的空间复杂度跟递归的深度有关系。

2、更重要的是，即便我们不考虑函数调用带来的开销，递归本身的处理方式是自顶向下的。比如斐波拉契数列用递归处理的递归状态树如下



每次都需要查询子问题是否已经被计算过，如果该子问题已经被计算过，则直接返回备忘录中的记录。也就是说，在带备忘录的递归解法中，无论如何都要多处理一个分支逻辑，只不过这个分支的子分支是不需要进行处理的。这样的话，我们就可以预想到，如果遇到子问题分支非常多，那么肉眼可见的额外时间开销在所难免。我们不希望把时间浪费在递归本身带来的性能损耗上。

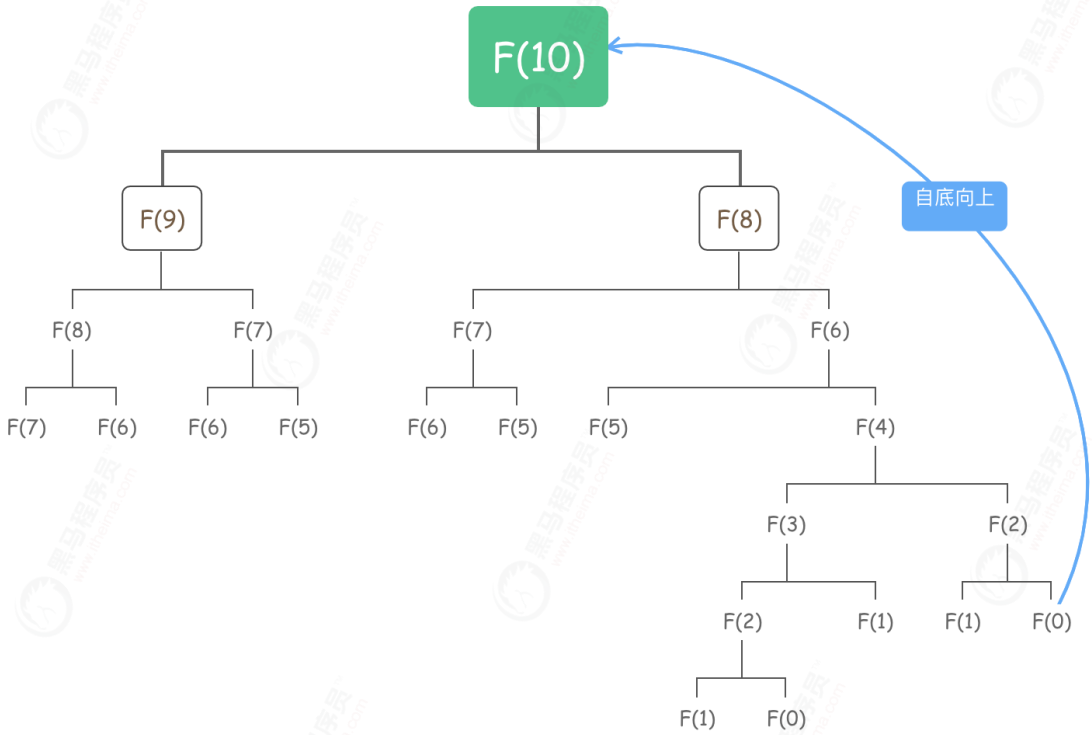
那么，有什么好的办法来规避这个问题呢？我们需要设计一种新的缓存方式，并考虑使用迭代来替换递归，这也是动态规划算法真正的核心。

1.3.4、状态缓存与循环

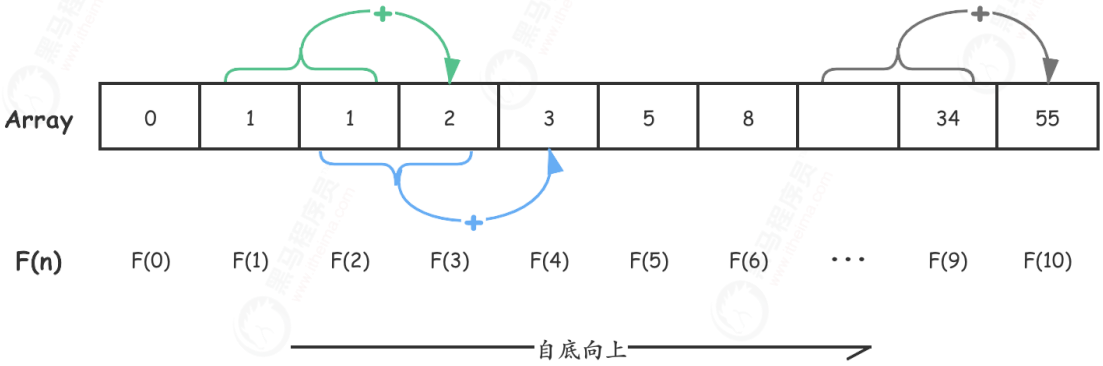
在带备忘录的递归算法中，每次都需要查询子问题是否已经被计算过。针对这一问题，我们可以思考一下，是否有方法可以不去检查子问题的处理情况呢？在执行 A 问题的时候，确保 A 的所有子问题一定已经计算完毕了。仔细一想，这不就是把处理方向倒过来用**自底向上**嘛。

回顾一下自顶向下的方法，我们的思路是从目标问题开始，不断将大问题拆解成子问题，然后再继续不断拆解子问题，直到子问题不可拆解为止。通过备忘录就可以知道哪些子问题已经被计算过了，从而提升求解速度。那么如果要自底向上，我们是不是可以首**先求出所有的子问题，然后通过底层的子问题向上求解更大的问题**

比如下图中表示了斐波拉契数列自底向上的处理方式



从解路径的角度看动态规划的自底向上处理方式，那么它的形式可以用一个数组来进行表示，而这个数组事实上就是实际的备忘录存储结构



这样有一个好处，当求解大问题的时候，我们已经可以确保该问题依赖的所有子问题都已经计算过了，那么我们就无需检查子问题是否已经求解，而是直接从缓存中取出子问题的解。通过自底向上，我们完美地解决掉了递归中由于"试探"带来的性能损耗。

这也就是**动态规划的核心，先计算子问题，再由子问题计算父问题。**

1.3.5、求解零钱兑换

1、确定状态参数和选择

> 原问题：

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

子问题：

```
输入：coins = [1, 2, 5], amount = 9
输出：3
解释：9 = 5 + 2 + 2
```

我们发现在原问题和子问题之间发生变化的是兑换金额amount

因此兑换金额 `amount` 就是状态参数，而选择也很简单，从 `coins` 中选择硬币会使得 `amount` 发送变化。

2、定义dp数组的含义：

`dp[i]` 的含义是：如果能从 `coins` 中找到相应的硬币凑成总金额 `i`，那么 `dp[i]` 则为所需的最少的硬币个数。

注意：

1、对于每一个兑换金额 `i` 都有两种可能，一种能从 `coins` 中找到硬币凑成，一种是不能从 `coins` 中找到硬币凑成。

2、如果能凑成则 `dp[i]` 代表所需的最少硬币个数。

因此在这里我们给 `dp` 数组中先填充一个不可能的值，比如 `-1,Integer.MAX_VALUE,Integer.MIN_VALUE` 等，后续状态转移的时候如果 `dp[i]` 能凑成功，则修改 `dp[i]` 的值。

coins = [1, 2, 5] amount = 11

i	0	1	2	3	4	5	6	7	8	9	10	11
dp[i]	0	1	1	2	2	1	2	2	3	3	2	3

在本题中，我们最终要返回的就是 `dp[11]`

3、确定初始状态：

当兑换金额 `i=0` 时，所需最少硬币个数为0，因此初始状态 `dp[0]=0`。

4、状态转移逻辑：

动态规划的核心，先计算子问题，再由子问题推导计算父问题；也就是说在我知道了 `dp[i-1]` 及更小子问题的值后我如何根据这些计算 `dp[i]`：也就是说：

可以理解为现在已经有了以下这些条件

能从 `coins` 中找到凑成金额 `i-1` 的最少硬币个数为 `dp[i-1]`，现在要凑成金额 `i`，

能从 `coins` 中找到凑成金额 `i-2` 的最少硬币个数为 `dp[i-2]`，现在要凑成金额 `i`，

.....

能从 `coins` 中找到凑成金额 `0` 的最少硬币个数为 `dp[0]`，现在要凑成金额 `i`，

如何凑成金额 i 呢？答案很简单：

从 coins 中依次选择硬币 coin，能跟它正好凑成金额 i 的是 i-coin，那凑成金额 i-coin 的最少硬币已经算出来了，就是子问题 dp[i-coin]，在它的基础上我们选择了一个金额为 coin 的硬币，

故：dp[i]=dp[i-coin]+1

当然 coins 中有很多金额的硬币可选择，每选择一个都会得到一个 dp[i] 的值，我们要求的是最小值，所以：dp[i]=min(dp[i],dp[i-coin]+1)

翻译成代码为：

```
for (int coin:coins) {
    if (coin <= i) {//代表该硬币可选
        dp[i] = min (dp[i],dp[i-coin]+1);
    }
}
```

5、代码实现

```
class Solution {
    public int coinChange(int[] coins, int amount) {
        //构造dp数组(缓存中间的状态信息/子问题的最优解/状态参数的值)
        int[] dp = new int[amount+1];
        //初始填充一个不可能的值,对应面额amount最多的组合就是amount个1的组合，需要amount个硬币,所以amount+1是不可能的
        Arrays.fill(dp,amount+1);
        //初始状态赋值
        dp[0] = 0;//amount=0无需硬币组合

        // 对每种状态的每种取值进行遍历

        //从子问题开始求解,推导到大问题，外层循环遍历所有状态的所有取值
        for (int i=1;i<=amount;i++) {
            //对每个子问题i，在coins中找能满足子问题i的众多解中的最优解(最少硬币数量)，内层 for 循环在求所有选择的最小值
            for (int coin:coins) {
                if (coin <= i) { //coin>子问题i，该硬币无法构成最终解,直接跳过
                /*
                    1:直接从dp中拿出求解问题i所依赖的子问题(i-coin)的最优解，
                    2:dp[i-coin] + 1即当前子问题i选择该coin的最优解
                    3:dp[i] = Math.min(dp[i],dp[i-coin]+1);是在众多解中选出一个最优的作为子问题i的最优解。

                */
                   dp[i] = Math.min(dp[i],dp[i-coin]+1);
                }
            }

        }
        return (dp[amount]==amount+1) ? -1:dp[amount];
    }
}
```

1.3.6、通用的动态规划

掌握了如何使用标准的动态规划来解决硬币找零问题后，我们需要将相关的解法来进行推广，以此推导出一套解决动态规划面试问题的通用框架，或者说套路。注意，这里是一个经验总结。

动态规划问题的核心是写出正确的状态转移方程，为了写出它，我们要先确定以下几点：

1、状态参数：找出子问题与原问题之间会发生变化的变量。在硬币找零问题中，这个状态只有一个，就是剩余的目标兑换金额 amount；

一般来说，状态转移方程的**核心参数**就是**状态**。

2、决策/选择：改变状态，让状态不断逼近初始化状态的行为。在硬币找零问题中，挑一枚硬币，用来凑零钱，就会改变状态，状态不断向初始化状态的变化过程就是**状态转移**。

3、定义 dp 数组/函数的含义。我们需要**自底向上**地使用备忘录来消除重叠子问题，构造一个备忘录（在硬币找零问题中的dp数组。为了通用，我们以后都将其称之为 DP table），**dp中存储的值就是我们想要的结果**。

dp数组跟我们的状态参数有关，状态参数不止一个，一般dp数组是多维的。

4、初始化状态：由于动态规划是根据已经计算好的子问题推广到更大问题上去的，因此我们需要一个“原点”作为计算的开端。在硬币找零问题中，这个初始化状态是 dp[0]=0；

5、状态转移逻辑：自下而上的解决方式就是，思考如何根据子问题的解去推导父问题的解，在该题目中就是如何根据 `dp[i-1]` 及更小的子问题的解，来推导 `dp[i]`

通过这样几个简单步骤，我们就能写出状态转移方程：

$$DP(n) = \begin{cases} 0, n = 0 \\ -1, n < 0 \\ min(DP(n), 1 + DP(n-c)), c \in values \end{cases}$$

由于是经验，因此它在 90% 以上的情况下都是有效的，而且易于理解。

在此也给出解决动态规划的通用框架如下：

```
//定义dp，可以是数组/哈希
int[][]....

// 初始化 base case
dp[0][0][...] = base

//进行状态转移
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][...] = 求最值(选择1，选择2...)

```

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，不那么容易穷举完整。

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。

子序列系列

子序列：在原数据序列中删除某些数据（也可以不删）后的新数据序列，会破坏原有数据的连续性。

子串：是原有数据串的一个子部分，

子序列类型的问题，如果穷举基本上复杂度都很高，指数级别，而动态规划算法做的就是穷举 + 剪枝，它俩天生一对儿。所以可以说只要涉及子序列类型的问题，十有八九都需要动态规划来解决，能把复杂度降低到 `n^2`

300. 最长上升子序列

字节，美团，快手最近面试，300. 最长上升子序列

通过前面对动态规划思想的讲述，发现状态转移的逻辑部分不太容易想到，其实**状态转移逻辑的思考有一个技巧叫做：数学归纳法思想**

> 如果我们想通过数学归纳法证明一个结论：
>
> 1、先假设这个结论在 k<n 时成立，
>
> 2、然后想办法证明 k=n 的时候此结论也成立。如果能够证明出来，那么就说明这个结论对于 k 等于任何数都成立。
>
> 类似的，我们设计动态规划算法，不是需要一个 dp 数组吗？我们可以假设 `dp[0...i-1]` 都已经被算出来了，然后问自己：怎么通过这些结果算出 `dp[i]`？

对于最长上升子序列这道题，我们的算法分析如下：

首先最长上升子序列，是一定要保证严格上升特性。

> 比如nums=[1,3,3,4,6,8,7]，但[1,3,3,4,6,7]并不满足，[1,3,4,6,7]才是。

1、状态参数：

```
>原问题，求nums=[1,3,3,4,6,8,7]的最长上升子序列长度，
>
>子问题，求nums=[1,3]的最长上升子序列长度
```

我们发现nums数组的下标 `i` 是这中间的变量。

2、决策/选择

从子问题到原问题的过程（从左到有遍历nums）中状态会发生变化

3、定义dp数组的含义

重要：大部分情况下我们题目要返回的数据就是dp数组中存储的值，或者通过dp数组中存储的值处理后可得到。

我们的定义是这样的：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度。

根据这个定义，我们的最终结果（子序列的最大长度）应该是 dp 数组中的最大值。

```
//构造dp数组,对dp[i]而言也是取各组合中的最大值
.........构造dp数组.........
//返回dp中的最大值
int res = 0;
for (int i = 0; i < dp.length; i++) {
 res = Math.max(res, dp[i]);
}
return res;
```

4、初始化状态

nums长度为0，则最长上升子序列长度为0，nums长度1（只有一个元素），则最长上升子序列1

5、状态转移逻辑

我们说动态规划的一个思想就是先解决子问题，然后由子问题去推导大问题的解决；这个推导过程我们可以用数学归纳法思想：

如果我们已经知道了子问题，`dp[0] ~ dp[5]` 的值了，如何通过这些值去推导出`dp[5]`

index	0	1	2	3	4	5
nums[i]	1	4	3	4	2	3
dp[i]	1	2	2	3	2	?

根据 dp 数组的定义，现在想求 dp[5] 的值，也就是想求以 nums[5] 为结尾的最长递增子序列。

nums[5] = 3，既然是递增子序列，我们只要找到前面那些结尾比 3 小的子序列，然后把 3 接到最后，就可以形成一个新的递增子序列，而且这个新的子序列长度加一。

当然，可能形成很多种新的子序列，但是我们只要最长的，把最长子序列的长度作为 dp[5] 的值即可。

```
for (int j=0;j<=i;j++) {
    if (nums[j] < nums[i]) {
        dp[i] = Math.max(dp[i],dp[j]+1);
    }
}
```

6、代码实现

```
class Solution {
    public int lengthOfLIS(int[] nums) {
```

```
        //定义dp
        int[] dp = new int[nums.length];
        //初始化,dp[0]=1,其实任意dp[i]初始化值都可以等于1,因为都可以将自己当作子序列
        Arrays.fill(dp,1);
        //状态转移
        for (int i=0;i<nums.length;i++) {
            for (int j=0;j<i;j++) {
                if (nums[j] < nums[i]) {
                    //从dp[i]所有组合中找最长的长度
                    dp[i] = Math.max(dp[i],dp[j] +1);
                }
            }
        }

        //返回dp[i]中的最大值
        int max = dp[0];
        for (int i=1;i<dp.length;i++) {
            max = Math.max(max,dp[i]);
        }
        return max;
    }
}

//简化版本
class Solution {
    public int lengthOfLIS(int[] nums) {

        //定义dp
        int[] dp = new int[nums.length];
        //初始化,dp[0]=1,其实任意dp[i]初始化值都可以等于1,因为都可以将自己当作子序列
        Arrays.fill(dp,1);
        int max=0;
        //状态转移
        for (int i=0;i<nums.length;i++) {
            for (int j=0;j<i;j++) {
                if (nums[j] < nums[i]) {
                    //从dp[i]所有组合中找最长的长度
                    dp[i] = Math.max(dp[i],dp[j] +1);
                }
            }
            max = Math.max(max,dp[i]);
        }
        return max;
    }
}
```

> 复杂度：时间复杂度O(N^2)，空间复杂度O(N)

7、总结

1、一定明确 dp 数组所存数据的含义。这非常重要，如果不得当或者不够清晰，会阻碍之后的步骤。

2、根据 dp 数组的定义，运用数学归纳法的思想，假设 dp[0...i-1] 都已知，想办法求出 dp[i]

3、如果无法完成第二步，很可能就是 dp 数组的定义不够恰当，需要重新定义 dp 数组的含义；或者可能是 dp 数组存储的信息还不够，不足以推出下一步的答案，需要把 dp 数组扩大成二维数组甚至三维数组。