# 算法精品讲解(2)——DP问题入门

原链接：

https://blog.csdn.net/xdnxl/article/details/126735355?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522E34329E1-FD75-42C2-A113-DC133B594248%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=E34329E1-FD75-42C2-A113-DC133B594248&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-126735355-null-null.142^v100^pc\_search\_result\_base

前言

DP入门介绍：

先画个饼吧哈哈。看完之后，你也可以。

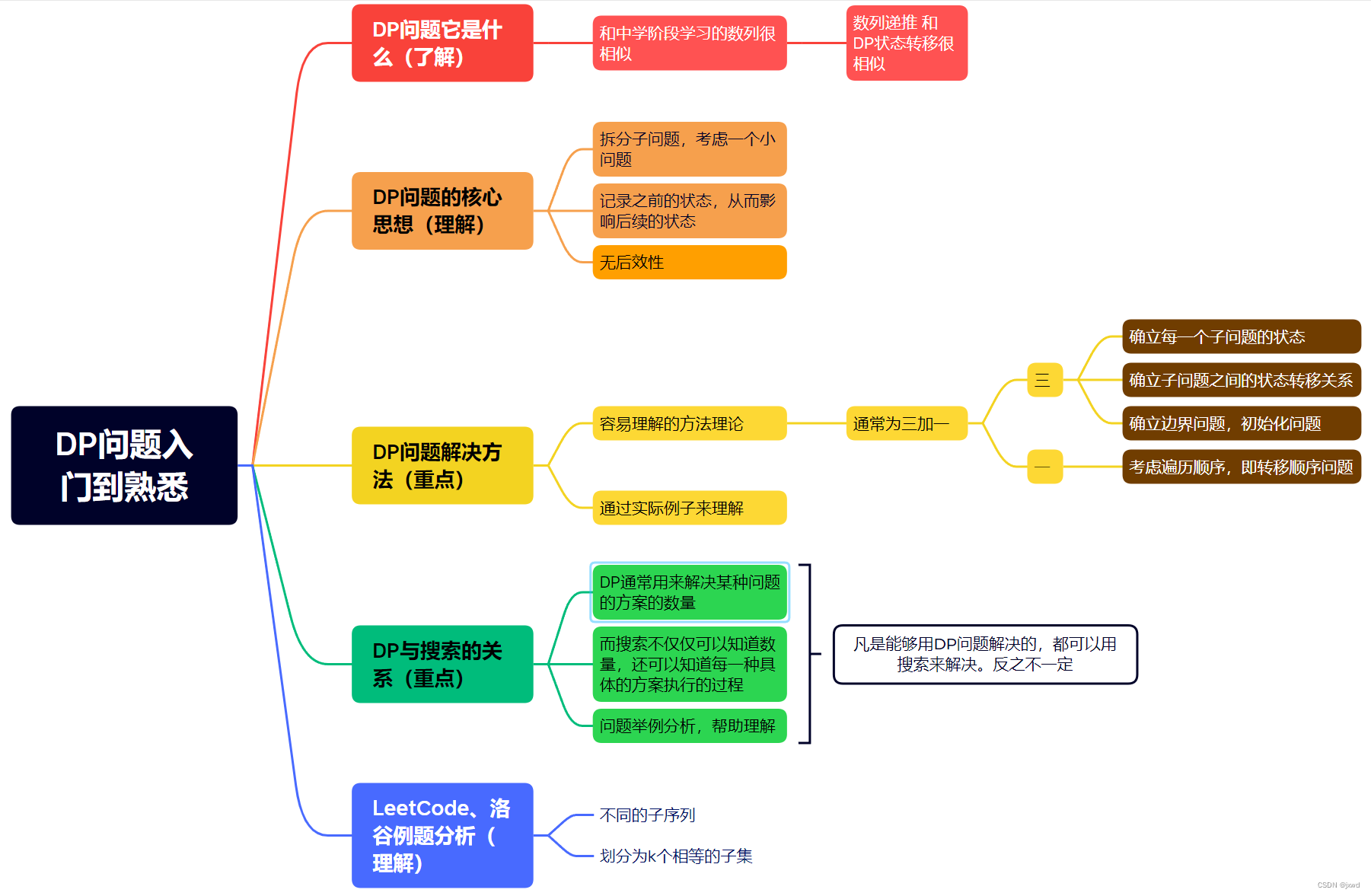
在本节内容中，我们将讲述DP问题的基本思考方法、基本逻辑要素。也就是DP问题的基本入门。

在本章节中，我们也将讲述我们通过一些题目所得出的思想方法，但大多数都是为了讨论DP的入门问题、基本问题、基本思考方法。对于DP问题的分类讨论，我们会在下一节中去进行详细讲解。

如下，是我们本节将要讲述的内容。我们将结合着LeetCode、[洛谷](https://so.csdn.net/so/search?q=%E6%B4%9B%E8%B0%B7&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/xdnxl/article/details/_blank)和CodeForce上面的一些习题来作为我们的参考。难度循序渐进，直到大于等于LeetCode Hard。

文章首发于wx公众h【自学编程村】

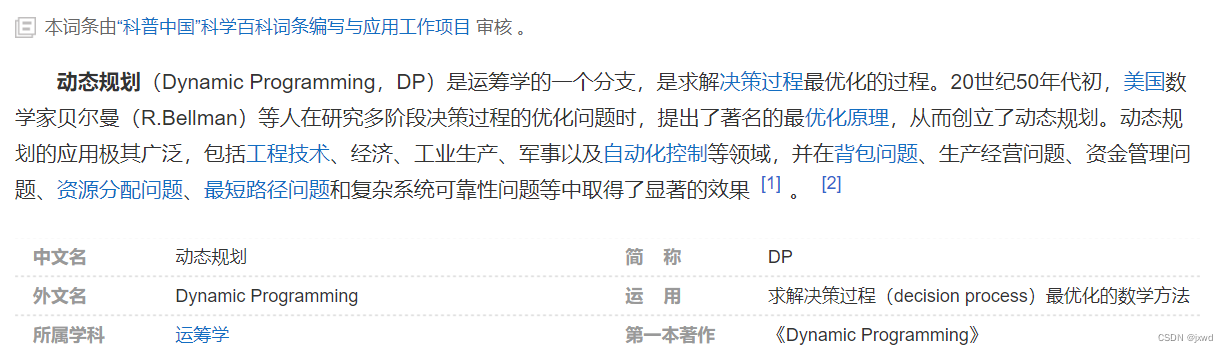
（原创By 自学编程村·阿祥/CSDN jxwd）



 我们下面就进入正文。

DP问题它是什么（了解）

大名鼎鼎的DP问题，实际上就是Dynamic programming。就是我们通常所说的[动态规划](https://so.csdn.net/so/search?q=%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/xdnxl/article/details/_blank)，简称DP。



由科学百科，我们简单了解下。

说人话，实际上DP问题，即动态规划问题，就是将一个大的问题，不断拆分为一个个小的问题。通过小的问题，并且在DP问题中，一个小的问题结果，是会影响后面的问题的结果。

注意，这里每一个小的问题，可以一个孤立的状态，也可以是前面所有的问题共同得到的结果。这个结果可能是贪心得到，也可能是由组合得到。

是不是觉得有一点晕或者比较难懂呢？哈哈不用慌，那是因为咱只是用文字去进行抽象地描述了。下面，我们将来具体地为大家探讨这一类问题。并结合着例题，帮助大家来理解。

从中学的例题谈起

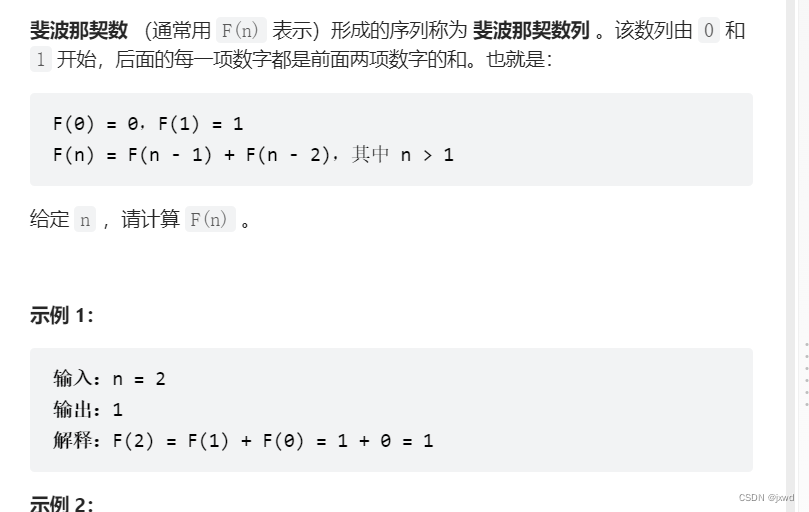
我们来举个简单的例子。

我们以中学时候学过的[斐波那契数列](https://leetcode.cn/problems/fibonacci-number/" \o "斐波那契数列)来作为举例（所谓的斐波那契数列，就是类似于：0，1，1，2，3，5，8，13...这样的数列，除了前两项，后面的每一项都是前面两项的和）。我们用LeetCode来作为平台来讲解这一问题。

[509. 斐波那契数 - 力扣（LeetCode）https://leetcode.cn/problems/fibonacci-number/](https://leetcode.cn/problems/fibonacci-number/" \o "509. 斐波那契数 - 力扣（LeetCode）)

我们主要来通过这样一个例子来理解何为DP。

我们为了方便同学们看题目，将题目截图放在下面:



 对于每一个位置的数，都是受前面的数所影响所得到。

我们可以暂时不用关心是怎么影响得到的，在不同的题目可能会有不同的方式。

我们只需要一个这样的概念：每一个数，都是由前面的数影响所得到（可以不是一个数影响得到）。

比如，在斐波那契这道题当中，F(5)，它是由F(4)和F(3)影响所得到；

一般情况而言，对于F(k)，它是由F(k-1)和F(k-2)得到.......这样，一直往前递推，直到推到临界条件，即初始条件（就是F(0)和F(1)）

我们再来看，我们用F(k)得到的数是用来干嘛的呢？实际上，它还有一种作用。可以认为是用来记录的，即记录我当前的数值，然后当后面有的数需要我来为它计算、为它推波助澜的时候，我通过我之前记录过的状态（即数值），继续计算后面的数。

所以，DP问题确实是将大问题拆分成为小问题。但是，这些小问题和小问题之间是存在联系的。一般我们都可以通过递推关系来去找到它们之间的联系。

但是需要注意的是，这里的递推关系，并不是我们中学时代所学的一项与前一项的数量关系或者是一项与前面几项的数量关系。它应当是认为是一项与前面所有元素整体的关系。我们可以将其抽象为一个关系，然后将其记录在前一项或者前几项的元素内。并且，一旦我将其存了起来，那么在后面的推导过程中，这个元素的值它是怎么来的？我就一点也不关心了。这就叫做无后效性。

我们再来结合刚刚的斐波那契数列，重新审视一下。

可能这道题过于简单，所以不容易发现这个点。

实际上，一个F(k)，和前面所有项的元素都有关系。是有前k-1个元素累积作用得到。而该作用的方式，在这道题中，为F(k) = F(k-1)+F(k-2)，而所作用的结果，就放在F(k)里。并且在后面的推导过程中，F(k)是怎么来的？我就一点也不关心了（这就叫无后效性）。我只知道，它是前k个数的某种作用得到的结果，怎么作用的得到的呢，那这我就不关心了（关于后效性我们下面也会有例题来详细探讨）。

但是，不是所有的作用方式都是这么简单。它可能是由前面的某些数的组合得到（下文有例题），也有可能是由前面的某些数求得最优解得到（下文有例题）等等。这个时候，其就会和贪心、排列组合等知识组合到了一起。而如何找到这样一个作用方式，正是dp问题的关键。

一般而言，每一种独立的结果我们称之为状态。而我们将作用方式称之为状态转移方程。

所以，我们可以这样形象地认为DP：

动态规划实际上就像一个多米诺骨牌一样。从第一张牌开始往后翻滚，每一张牌实际上都是一种状态。我们给定了一个初始条件，就是给了第一张牌一个动力。将牌与牌之间的位置距离摆好，就是确定状态与状态之间的转移方程，将所要要求的骨牌推到，就得到了我们想要的结果。

需要强调的是，和骨牌有些区别的是，实际中的转移不应当是看成一个牌对另一个牌的作用，应当看成是该牌前面所有的牌对另一个牌的作用。

再来说一下，DP问题的核心思想（理解）

实际上，在上述的斐波那契的例子当中我们已经全部提到过了。我们在这里抽象总结一下，然后我们在下面会再结合例子，和具体的解题方法一道，进行进一步的讲解。

我们现阶段可以将其分成三个：

1、拆分子问题；将大问题拆分成小问题。

2、记录状态；每一个子问题得到的状态一般都记录在了一个dp数组当中，或者是通过栈、队列等其他的方式记录。

3、DP无后效性。按照一定顺序去求解拆分的小问题。当小问题的状态或者结果得到了之后，我们就再也不去关心它是怎么样得到的了。

DP问题的解决方法

先说方法论：

总的来说，我将其总结为3+1步走。

三：

1、首先我们将一个大的、整体的问题拆开，拆成小的问题。那么我们就要确定，每一个小的问题解决之后所得到的结果表示的是什么。即确定每一个小的问题表示的状态。比如，dp[i]表示前i个数的某种方法的个数，再比如，dp[i][j]表示某个字符串从i到j是不是回文串等等。就是说，你要明白，你的每一张多米诺骨牌代表的是什么东西。

2、确立每一个状态之间的转移关系。多米诺骨牌立好之后，是不是要想着这些骨牌要怎么摆放的问题？那么状态转移就是可以理解为用某种方法，将骨牌摆好。比如，dp[i] = dp[i-1]+dp[i-2]，复杂一点的，可能还有类似于dp[i] = max(dp[i],dp[i-dp[j]]) (j = 0, 1, 2...,i - dp[j] >= 0)这样。

3、确立初始状态。就是给多米诺骨牌一个初始的动力，让它可以触发效应，一个一个全部倒下。最最常见的，就比如dp[0] = 1 这样子的。也叫确立边界条件。

一：

指在求解问题当中的遍历顺序。这点和上述的“三”是紧密相关的。也就是说，你要明白你的骨牌是怎样一个个倒下的（是从后往前？还是从前往后？）最典型的，我们后面在讲到背包问题，进行空间优化的时候，就会看出遍历顺序所带来的重要作用。

再说具体的例子

例一：

我们再来看这样一个例子：

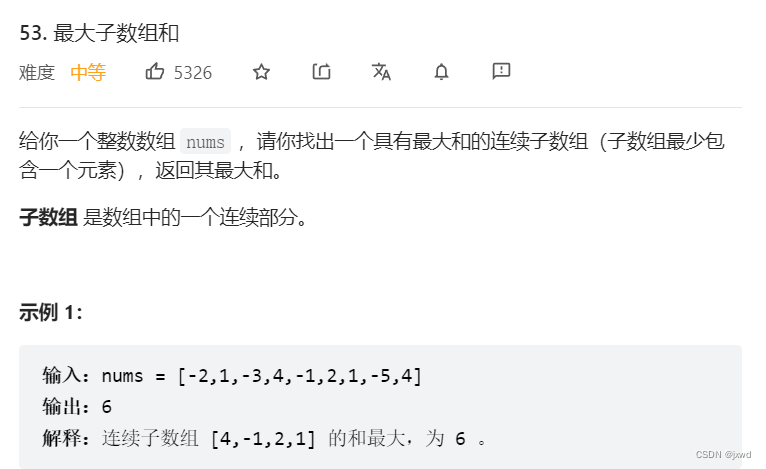
[53. 最大子数组和 - 力扣（LeetCode）https://leetcode.cn/problems/maximum-subarray/](https://leetcode.cn/problems/maximum-subarray/" \o "53. 最大子数组和 - 力扣（LeetCode）)通过这个例子，我们来正式入门动态规划。

这道题，实际上是dp入门的板子题，或者说是经典例题，它包含、体现出了动态规划的一些基本的特性。

在这道题中，我们将会体会到动态规划问题的子问题拆分、子问题最优解、无后效性等特性。

同时，我们也将通过这样一道题，给大家讲述最经典的3+1的动态规划求解步骤。

下面这个是题目截图：



 好。我们按照上述的三加一步走战略，在一边求解的过程当中，一边来感受思想。

第一步：拆分子问题。并明确每一个dp[i]个体代表什么样的状态。

首先，我们确定一下我们的子问题是什么。或者通俗一点来说，我们如果开辟一个以为数组，每一个数组元素代表着什么。也就是dp[i]可以代表着什么。

其实很简单。题意中是让求数组nums的最大子数组和，那我们的dp[i]拆分出来就是求数组下标从0到i所组成的新数组当中的最大子数组和，而为了便于下面递推的时候形成一个完整的子数组，我们这里的dp[i]准确来说是把下标为i的元素算上的最大子数组和。我们举个例子，比如i == 3，那么dp[3]就可以是说它是原来nums数组下标从0到3这样一个范围构成的一个数组 的最大子数组和。

OK，到此为止，第一步，拆分子问题，确立所表示的状态是什么结束。

这样以后，我们就有了一个个多米诺骨牌，那么我们接下来就需要按照一定的顺序逻辑将其摆放好。也就是我们的第二步，最为关键的一步，确立转移方程。

第二步：确立每一个dp[i]之间的关系

在本题中，我们可以看到，实际上dp[i]的更新，我们根据无后效性，即这个数值更新完后就再也不关心它是怎么来的了，所以我们知道它只能和小于i的部分或者大于i的部分的一侧产生关系。我们除非是要逆序遍历，否则一般都是和小于i的一侧产生关系，本题也是这样，也比较简单。

那说了半天，到底存在着什么样的关系呢？注意到这个时候，我们手里面是有dp[i-1]的，就是下标从0到i-1的新数组当中的最大子数组和（还是因为无后效性，我们这个时候并不需要关心这里的dp[i-1]是怎么得来的，直接用就可以了）

那这样的话，我们的思路就简单了。就是说，我们在遍历到第i个元素的时候，对于dp[i]而言，我们的第i个元素一定要被选上（这个原因我们下面还会具体来说）。那么现在，要么把前面的dp[i-1]都加上，要么就不加，加还是不加怎么选择？很简单，哪个大就选哪个。所以就是dp[i] = max(dp[i-1] + nums[i], nums[i]);

我们刚刚说的可能有些抽象。我们其实还可以这样去想：dp[i-1] + nums[i] 如果比nums[i]大，说明前面的那一坨（dp[i-1]，至于dp[i-1]怎么得来的我不管，这也就是无后效性）还有点作用，那我就把它要着。如果dp[i-1] + nums[i] 还没有nums[i]大，那dp[i-1]就是个拖后腿的，我还要它干嘛？直接把它扔掉就行。

所以，我们就得到状态转移方程：

dp[i] = max(dp[i-1] + nums[i], nums[i]);

针对本题，我们来讲解一个大家可能会产生的疑问：

我们每一个dp[i]为什么一定要把第i个元素选上？也就是说，为什么dp[i] = dp[i-1]这样一种情况没有算进去？

答案很简单：因为没有必要。而且会破坏子数组的连续性。一个一个来解释。我们在搞清这个问题之前，需要明白的是我们这题的最终答案在哪里？最后的答案是需要遍历一整个dp数组，然后找到那个最大的数，即是我们的答案。原因是我们的dp[i]是以第i个元素结尾（也就是把第i个元素算上，并且这个时候第i个元素恰好在数组末尾）的最大子数组和，而i是从下标0-nums.size()，所以就是说，我们把原数组nums每一个元素为结尾的情况的最大子数组和都考虑了一遍，记录在dp数组当中，然后最后再遍历一遍dp数组，找出每一种情况值的最大值，这样的话，我们用这样的方式，就可以把所有的情况都考虑上了，并且还能在这些所有的情况中找出最大值。第二点，就是如果第i个元素不选，会破坏子数组的连续性。怎么说？很简单的道理，试想，如果你第i个不选，那么你第i+1个是不是想选都选不上了？就是说，第i+1个元素要么是自己单着，要么是dp[i+1] = dp[i-1]（这是没有选择的情况），没有dp[i+1] = dp[i] + nums[i+1]这种将该元素添加进上一个子数组里面这种情况了，因为第i个元素没有选，第i+1个元素和前面的元素构不成一个连续的数组。所以这样，我们会少考虑一种情况。同时，我们也不可以保证在递推的过程中数据具有无后效性了。

所以综上，由于不选上第i个元素，会让子数组不连续，而一定选上第i个元素，我们又通过多遍历一遍的方式将所有的情况都考虑了进去。一正一反的理由，得到结论：把第i个元素一定选上，才是可行的。

如果你还是不能理解，那建议可以举几个例子来试试。

第三步：确立初始状态

这个初始状态很简单的。需要注意的是，它的题目要求是子数组不可以为空，至少得要有一个元素。而我们这里可以考虑dp[0] = nums[0]这样初始化即可（注意：确立初始化状态和确立遍历顺序是息息相关的，我们下面会说到）

这样以后，我们三步走战略就结束。

我们再来看下遍历顺序的问题。也就是3+1中的1。

一：

一般情况下，我们对于一维数组，而且在更新dp数组的过程中就只遍历一遍，所以直接正向遍历就可以了。但是需要注意的是，它和初始状态的确立的关系是很大的。还是想象多米诺骨牌的例子，我要从前往后推，那我肯定是要在最前面的那张牌上给它一个初始动力，而如果要从后往前推，那就是要在最后面的那张牌上给它一个初始动力。很容易理解的。

我们本题直接就正向遍历就行。

好，至此，我们分析完毕，来看代码，结合代码再过一遍3+1步走：

C++:

class Solution {

public:

int maxSubArray(vector<int>& nums) {

int size = nums.size();

int\* dp = new int[size+5]; //动态开辟数组，也可以静态开辟，也可以用vector

//同时明确每一个dp[i]代表的是什么状态

dp[0] = nums[0]; //初始化

for(int i = 1;i < size;i++){

dp[i] = max(dp[i-1] + nums[i], nums[i]); //每一个dp[i]之间的状态转移方程

}

int Max = INT\_MIN; //遍历找到最大的那个子数组和

for(int i = 0;i < size;i++){

Max = max(Max, dp[i]);

}

return Max;

}

};

其中dp问题在实际求解的过程当中，最花时间的和最难的，就是在三步走的第二步，即确立状态转移方程。如果这一步成果解决，那么其他的步骤基本也就迎刃而解了。

我们为了再次熟悉一下这种方式，我们再来举一个例子：

例二：

[198. 打家劫舍 - 力扣（LeetCode）https://leetcode.cn/problems/house-robber/](https://leetcode.cn/problems/house-robber/" \o "198. 打家劫舍 - 力扣（LeetCode）)



 这个题目实际上比上面的题目还要简单一点，用三步走求解非常快（由于遍历顺序一般情况下都是正向遍历，我们这里就不作为重点来去说了）

我们这次就说的比较简洁了。

第一步：确立单个dp所表示的状态。

这里，我们就让dp[i]表示原数组前i个数得到的最大钱数（即在原数组下标从0到i能偷得的最大钱数）

第二步，确立状态转移方程。

思考一下，对于一个dp[i]的第i个屋子，还是两种情况，偷或者不偷。同时根据无后效性，如果要偷，那第i-1个就不能偷，故应当加上前面i-2个的屋子偷的钱数。那么dp[i]就是nums[i] + dp[i-2]；那么如果不偷第i个屋子，那dp[i]就是前i-1个的，即dp[i-1]。这两种情况哪种情况的数值大选哪种(体现出子结构要选最优)。故状态转移方程为：

dp[i] = max(dp[i-2] + nums[i], dp[i-1]);

第三步，初始化，确立边界。

这里我们想要递推起来，需要初始化两个值，即dp[0] = nums[0]，dp[1] = max(nums[0],nums[1])

需要注意的是，这里的dp[1]需要讨论一下原数组nums的个数再写，因为原数组可能就只有一个元素。

然后遍历顺序是从左到右（从前到后）

然后就可以写代码啦：

class Solution {

public:

int rob(vector<int>& nums) {

int dp[100+5] = {0}; //由题意可知，这里的个数比较少，所以可以直接静态定义就行

//并需要明确每一个dp值所表示的状态是什么

int size = nums.size();

if(size == 1) return nums[0];//讨论一下个数

dp[0] = nums[0];dp[1] = max(nums[0],nums[1]);//初始化

for(int i = 2;i < size;i++){

dp[i] = max(dp[i-2] + nums[i],dp[i-1]); //状态转移

}

return dp[size-1]; //结果

}

};

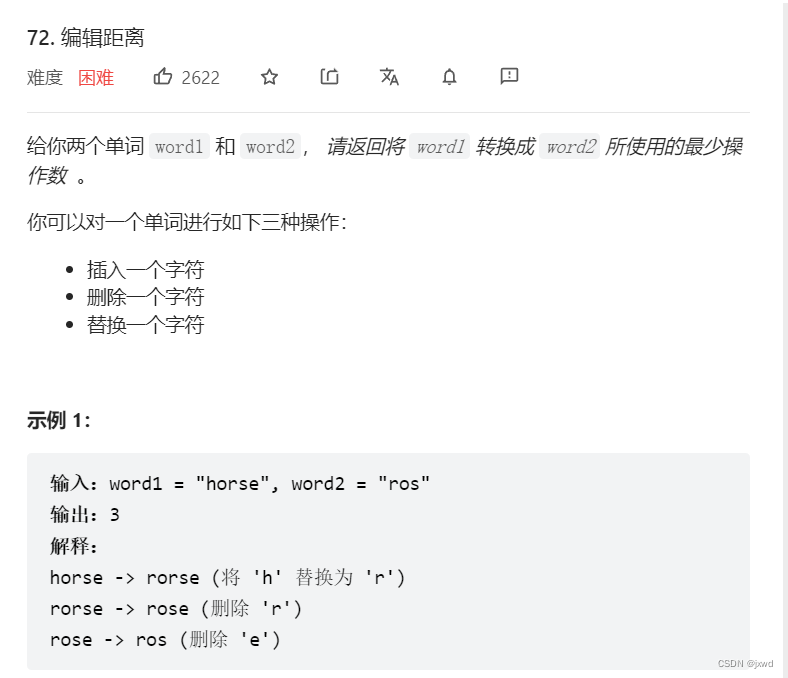
IMG_278

这道题就结束啦。

我们再来看一道稍稍难一点的

例三：

[72. 编辑距离 - 力扣（LeetCode）https://leetcode.cn/problems/edit-distance/](https://leetcode.cn/problems/edit-distance/" \o "72. 编辑距离 - 力扣（LeetCode）)



 我们结合这道题，说说画图的思想，然后你会感受到，dp实际上还是很暴力的。

我们首先思考分析一下。本题是要研究两个字符串之间的关系。

那么，我们可以这样来研究问题：

1、用dp[i][j]表示word1中前i个元素想要和word2中前j个元素匹配，所要进行的增删改的次数。

2、那么这样的话，我们的状态转移方程是这样的：如果word1[i] == word2[j]，那么dp[i][j] = dp[i-1][j-1]，很容易理解。如果word1[i] != word2[j]，dp[i][j]如果是从dp[i][j-1]变化而来，那就是删除；如果是从dp[i-1][j]变化而来，那就是增加；如果是从dp[i-1][j-1]变化而来，那就是修改。

那么想要得到最小的数值，直接就dp[i][j] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])+1 （就是谁小就选谁）

所以，状态转移方程就是：

if(word1[i] == word2[j])dp[i][j] = dp[i-1][j-1];

else dp[i-1][j-1] = min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1]) + 1;

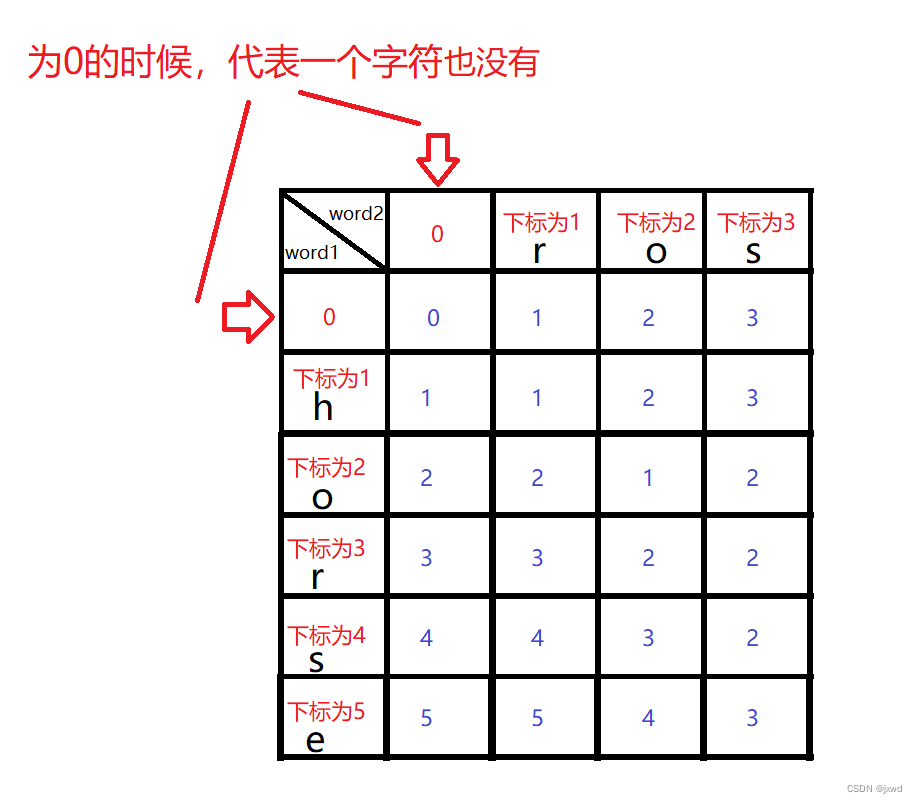
至于为啥  dp[i][j]如果是从dp[i][j-1]变化而来，那就是删除；如果是从dp[i-1][j]变化而来，那就是增加；如果是从dp[i-1][j-1]变化而来，那就是修改呢？实际上这样的题目类型有很多，熟悉了之后就比较容易接受和理解了。

3、确立初始化条件：这里我们注意到，初始化我们应当初始化一行和一列。因为需要考虑一个字符都没有的情况。初始化实际上也就是初始化这种情况。

对于第一横行dp[0][i]应当全部执行的是删除操作，dp[0][i]的值应该为i；

同理，对于第一列行dp[i][0]应当全部执行为添加操作，dp[i][0]的值应该也为i；

如果你是刚刚接触这样的题，可以通过画图来理解。我们就以原题的样例为例子来说一下。



 如上图，要求dp[1][1]，它是min(dp[0][0],dp[1][0],dp[0][1])+1，显然它的最终结果是1。

其他的同理就行了。

我们将代码附上：

class Solution {

public:

int minDistance(string word1, string word2) {

//dp[i][j] 表示的是word1中的前i个字母转换为word2中的前j个字母所需的最小次数

//dp[i][j] = dp[i-1][j] + 1(删);dp[i][j-1] + 1（增）;dp[i-1][j-1]（改）;dp[i-1][j-1]

//dp[0][0] = 0; dp[i][0] = i;dp[0][i] = i;

int dp[500+5][500+5] = {0};

int size1 = word1.size();

int size2 = word2.size();

for(int i = 0;i <= size1;i++){

dp[i][0] = i;

}

for(int i = 0;i <= size2;i++){

dp[0][i] = i;

}

for(int i = 1;i <= size1;i++){

for(int j = 1;j <= size2;j++){

if(word1[i-1] == word2[j-1]){

dp[i][j] = dp[i-1][j-1];

}

else{

dp[i][j] = min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1]) + 1;

}

}

}

return dp[size1][size2];

}

};

 好。关于这一点，我们暂时就介绍到这里。

我们下面说说DP和搜索有什么样的关系。

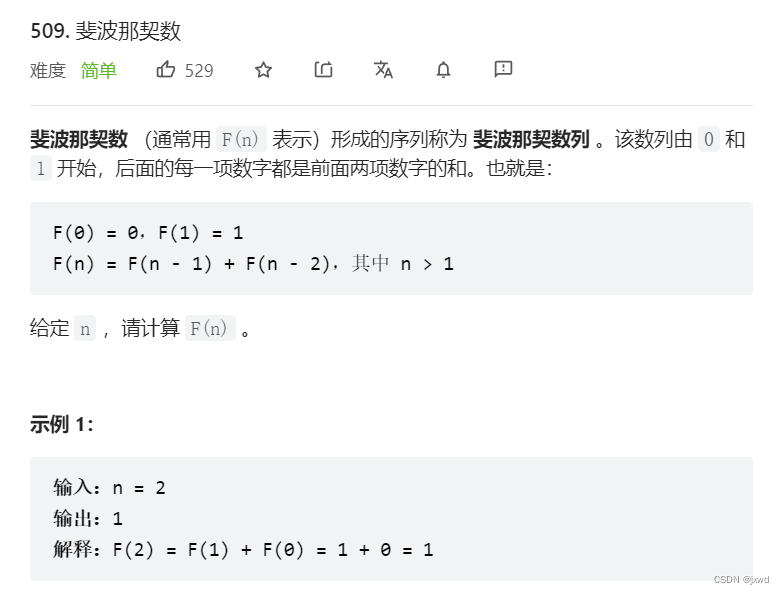
DP和搜索的关系

大家都知道，搜索中有一种搜索叫做记忆化搜索，即可以通俗的理解为DFS+剪枝。

我们来讲一讲它们之间有什么样的关系。

实际上，DP和记忆化搜索本质上是一个东西。都是在一边遍历的过程当中、考虑所有情况的过程当中，一边记录，这样以后从而减少重复计算。只是记录的方式不同。在一般情况下，我们能用DP解决的问题，能可以用搜索来去解决。反之则不一定，因为一般我们是要用到了记忆化搜索才好向DP的方法转换。

我们举一个最最最最简单的例子(还是刚刚的那个例子)——斐波那契数列

[509. 斐波那契数 - 力扣（LeetCode）https://leetcode.cn/problems/fibonacci-number/](https://leetcode.cn/problems/fibonacci-number/" \o "509. 斐波那契数 - 力扣（LeetCode）)

 像这样的一道题，就是典型的，我们既可以用DP，又可以用搜索。

一般情况下，你最先能够想到哪一种，就用哪一种。就是哪一种好想、好写，就用哪一种。

由于本题两种方法都比较简单，我们就通过这道题，来感受一下这个本题两种方法之间的差异和相同的共同之处。

这两种方法的代码我认为也不需要再做过多的解释了。

首先DP：

class Solution {

public:

int fib(int n) {

int dp[30+5] = {0};

dp[1] = 1;

for(int i = 2;i <= n;i++){

dp[i] = dp[i-1] + dp[i-2];

}

return dp[n];

}

};

当然了，你也可以继续去优化它。优化成只有三个变量滚动的形式。不过由于本题本来就比较简单，就不去细说了。

然后记忆化搜索：

class Solution {

public:

int v[30+5] = {0};

int fib(int n) {

if(n == 1)return 1;

if(n == 0) return 0;

if(v[n] != 0) return v[n]; //记忆化的过程

v[n-1] = fib(n-1);

v[n-2] = fib(n-2);

return v[n-1] + v[n-2];

}

};

我们可以看到，实际上我们这两种思路都是致一的。核心都是拆分成子问题，然后通过适当的记录结果，达到简化计算的目的。

所以，一般来说，能够用DP解决的问题，都可以用搜索来解决。

具体用DP还是用搜索，还是那句话，哪一种好想，就用哪一种方法。

注意到，我们还需要注意的是，我们这里所举的是一个非常非常简单的例子。关于这两种思想之间所存在的更多的关联，还是希望建议能够在具体的做题过程当中去总结、去感悟更好。否则，只用抽象的道理，理解起来也会有些生硬。

结语

实际上，到目前为止，我们也已经讲解了几个例题了。我们在最后，就不再为大家继续讲解新的例题了。理由很简单，因为动态规划的题目是有规律可循的，可以将其按照知识点，或者是按照题型归类。而笔者将会在下一篇姊妹篇中，为大家带来按照知识点来讲解的动态规划。而动态规划的提醒大家可以通过多多做一些练习，并将其加以总结，就会对这一块有着越来越深刻的印象和认识。本篇文章的定位只是帮助大家认识DP，将大家领进门，还远远没有能够达到能够熟练掌握的地步。但是大家至少在看完本篇文章之后，能够对所有的DP问题有着清晰的架构了，并且在之后看题解的过程中、做题的过程中也会有自己的一些套路和思考在了。不至于一窍不通、是一个门外汉了。

想要熟练掌握，该怎么办呢？

答：熟悉了DP的几种常见类型之后，做题。对于知识点的总结可以帮上忙，但是做题、训练，这一块还是要靠自己呀~