动态规划篇

{% include alert.html text="本文作者：刘杰煌 原稿于2022年4月12日发表 本文受版权保护，转载请联系编辑部." %}

一.什么是动态规划?

首先要说明，动态规划不是一种特定的算法，动态规划不同于其他算法，不是一个一成不变的算法，动态规划其实是一种**方法**。

有些算法，比如DFS、BFS算法，你只需要去理解以后记住这个算法的模板，怎么去用，在哪些情况下用，以后遇到类似的问题，将模板转化一下就可以用了。

但动态规划不是，动态规划是一种“方法论”，这就是动态规划的迷雾所在.

wiki上面对于动态规划的定义如下：

> dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

动态规划是一种方法，用于解决**复杂问题**时将复杂问题转化为一个个**相似的小问题**，所有的问题都只需要**计算一次**，并且**储存**它们的解。

二.动规和回溯的关系?

当你研究了几道动规的题目以后，就会觉得这东西和回溯算法太相像，为什么这么说？且慢，先来看一道简单的fibonacci数列的题目入手，Fibonacci数列即

的这样一个数列。在每个人刚刚开始学习C语言的时候，老师肯定都花了很大功夫用这个例子来讲递归。给定一个数，从开始输出Fibonacci数列到第N个数。根据上面那个公式，n>=2时，F(n)只与n前两位的数有关。

即，如果我们要求F(20），我们需要知道F(19)和F(18)

```
if(x == 1 || x == 2 ){
                return 1;
        }
else{
                return fun(x - 1) + fun(x - 2);
        }
```

用动态规划的方法，我们需要一直从上往下求，直到到达我们设定的base case（基础值）。很明显，在求F（19）的时候，我们依然要求一次F（18），继续按照转移方程，从上面走到底，获得F（18）的结果。也就是说，求F（20）,其实我们可以说成我们求的是F（1）... F(19)，因为最终的**大问题**的结果来源于这些**小问题**。

你可能觉得这样很烦，F（18）我们不是求过一次了吗，现在何必再来求一次呢？如果我们第一次求到F（18）的时候能够把它结果保存起来，当要用到F（18）的时候再用我们已经储存的值不就行了吗？

其实这样的思考已经和动态规划挂钩了，只是很多时候我们没发现这是用到动态规划思想的。我们把这些结果储存起来，要用的时候直接调用，比刚刚那个递归节省了不是一点半点的时间。

那我们只需要改一下，每一次算出结果后将结果保存起来，供后面调用。

```
//动规版本

int fibo(int n){
        if(n < 1) return -1;
        int F[n+1];
        F[1] = 1;
        F[2] = 1;
        for(int i = 3; i <= n; i++){
                F[i] = F[i-1] + F[i-2];
        }
        return F[n];
}
```

这就是Fibonacci数列计算的动态规划版本了。

所以动规是一种聪明的遍历，因为在于他有记忆，他不会再去遍历已经思考过的东西

按我个人的理解来说

> ** 动态规划就是一种 将一个大问题分解为各个独立的小问题，建立一个可保存数据的结构（通常是数组）来缓存小问题已经得出的结果，并且在后面通过一个归纳的方程（即状态转移方程）复用这个结果，得出大问题的结果 的一种方法。**

如果你觉得读的很绕，那就记住关键的这几个字

> **缓存数据、复用数据、状态、转移方程、方法**

在学习的时候，我们经常会见句知意，但在动态规划这里最好不要这样，因为这似乎和“动态”打不着半点杆子。

外国版知乎Quara上面举了个很通俗的例子，很好的解释了什么是动态规划：

> Q：（1+1+1+1+1+1+1+1+1+1+1）=？
>
> A：（你一个一个数了，很慢的回答）等于11
>
> Q：（1+1+1+1+1+1+1+1+1+1+1）+1=?
>
> A：（你只看到右边多了个+1，快速回答了）等于12

为什么第二次你不需要一个一个数到底有多少个“1”在等式左边呢？因为第一次问你的时候，你数了，知道等于11。第二次在左边加了个1，对于你来讲，就是问你：11+1=？ 于是你几乎不需要停留，脱口而出：“12”

回到正题，这个例子就是在阐述**缓存数据**、**复用数据**、**转移方程**（在Fibonacci数列这个例子里就是呈现的**算术**）这些动态规划的核心。

** 二.什么问题能用动态规划？**

在这里，先不仔细解释满足最优子结构、无后效性这些术语，第一次学每个人都会对这些术语很懵，后期我们会用几道题目来向你解释这些术语的含义。

浅显的说，一个问题如果**大问题**(原问题）能够分解为一个个**小问题**，小问题可以拆分为更小的问题，同时，大问题的最优解能够通过小问题的最优解**递推**得到、小问题的答案可以由更小的问题的最优解**递推**得到（这个大与小指规模的大小），这就是满足“最优子结构”的问题。

在这里大问题的**结果**，我们只看小问题的**答案**，而**不用考虑**小问题的答案是如何得到的，这就是满足“无后效性”的问题。即

> “现在就是过去的总结，现在决定未来，未来与过去无关。”

那这个问题就可以通过动态规划解决。

“什么问题能用动态规划？”这个问题实际上有些泛化。

应该这样问：“什么问题适合用动态规划？”

那就是长期被复用，这才是关键。

如果不被长期复用，那么除去“**缓存、复用**”的步骤，这个“定义”依然是可以适用于其他情况的。

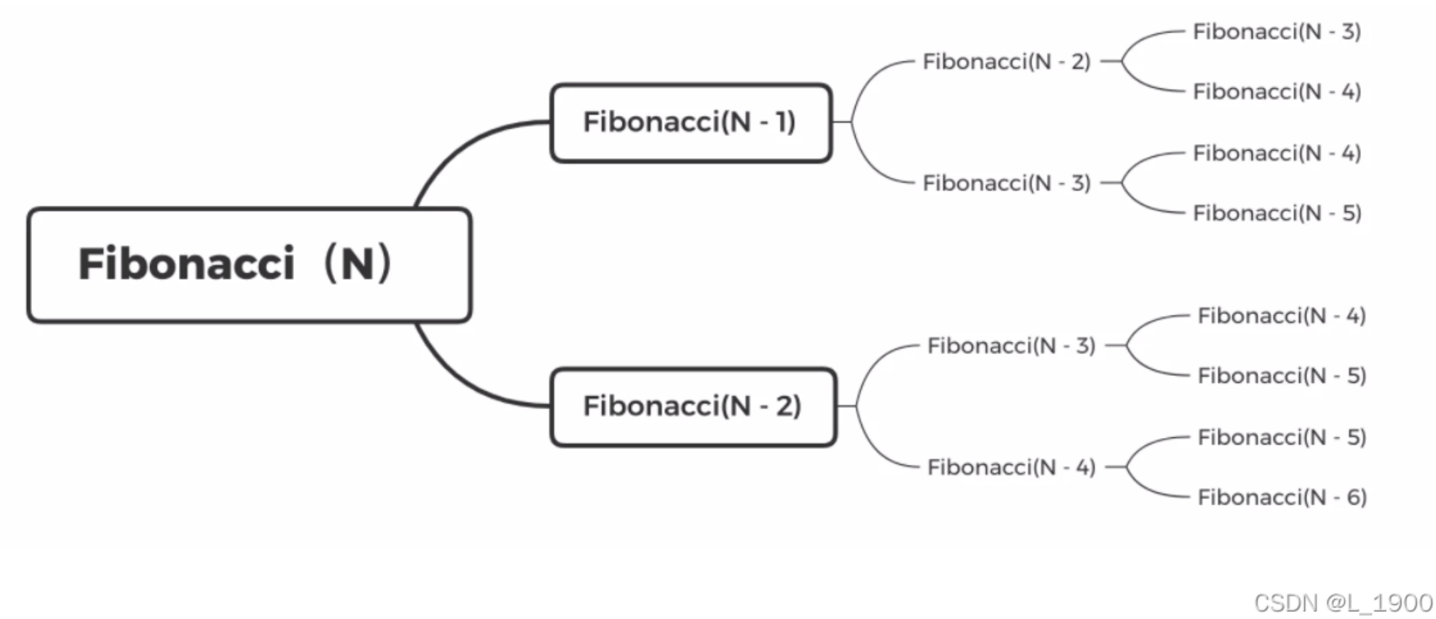
实际上很多大问题都是可以转化为小问题的，那为什么我们一定要用动态规划呢？我们用递归从上到下暴力求解不也可以吗？还减少了“**找状态转移方程**”这个麻烦的过程，我们直接把数据交给计算机，叫计算机一个一个遍历然后从中筛选取得我们想要的值就行了，要知道计算机最擅长的就是计算，计算机最不怕的就是大量的数据。

一切的一切，在于**长期被复用**。

长期被复用而不用动态规划的结果，就是计算机遍历时间无边无际，时间复杂度过高，一切算法都将毫无意义。

既然多次被复用，那么用一个结构储存当前的结果，以便于后续使用，这才是聪明人的做法。

再举一次Fibonacci数列的例子，更好的理解长期被复用带来的影响



> F(n)=F(n-1)+F(n-2)
>
> 你要求F(100)，自然要知道F(99),F(98)。
>
> 你要求F(99)，自然要知道F(98),F(97).
>
> 你要求F(98)，自然要知道F(97),F(96).
>
> ...

用递归方法，你要从F(99)求到F(1)，然后再从F(98)求到F(1)，你才能得到F（100）

用动态规划，我们第一次求到F（99）的时候就直接保存值（很明显，每一个值在Fibonacci数列里面都会调用多次），后期用到直接拿来用，这样省下的时间就将是指数级别，不是一点半点。

三.动态规划思路

> 1.明确题目里的状态
>
> 2.明确DP数组的定义
>
> 3.做出正确的状态转移方程。

每一次求出F（n），我们就把这个值保存起来，实现了缓存数据并复用。

> Fibonacci数列问题里面，我们要求的是指定数字的Fibonacci值
>
> 状态自然就是数字n
>
> 我们把F（n）定义为：数字为n时的Fibonacci值
>
> 那么根据我们的状态定义，该题正确的状态转移方程是 F(n)=F(n-1)+F(n-2)

以上任何一个环节定义错误，都会影响到下一环节的进行，这就要求我们**明确状态**、**明确DP数组的定义**，不然转移方程有可能会不再适用（脱离了“**最优子结构**”或脱离了“**无后效性**”），也就得出了错误的最终结果。

这种情况下，我们要重新考虑状态、重新考虑DP数组的定义。确保DP状态的定义满足：

> **1.最优子结构**
>
> **2.无后效性**

通过几个题目，我的一般处理思路是：

> **1.思考状态是否找对、找全？下一结果不能由当前结果递推得到（也就是说，当前的DP状态定义不满足“最优子结构”），有可能是状态不够，也许这个DP数组可以升为二维、三维甚至更多维来解决（参考 Leetcode.买卖股票问题）、或者可能要用到两个、多个状态定义来做（参考 Leetcode.乘积最大子数组）用到来储存更多状态**
>
> **状态越明确，范围越明确，从而可以从更多的状态（更为明确的范围）里面做出正确的选择，找出正确的状态转移方程。**
>
> **2.思考DP数组的定义是否正确？DP[i]储存的东西是什么？变换一下思路，定义正确的DP数组，充分仔细考虑问题条件与所求，确保储存的东西一经确定，不会受到未来（过去）的影响。（参考最长子数组和问题）**

为什么一定要保证状态的定义满足**最优子结构、无后效性**?

上面已经写了，DP数组定义不适当的话，不仅找不到合适的状态转移方程，而且在做题过程中可能会混乱状态，不知所然。

拿**最长子数组**的问题来举例

> 给你一个整数数组 nums ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
>
> 子数组 是数组中的一个连续部分。

> 示例 1:
>
> 输入：nums = [-2,1,-3,4,-1,2,1,-5,4]
> 输出：6
> 解释：连续子数组 [4,-1,2,1] 的和最大，为 6 。

如果你做了一些其他求最值的动态规划的题目，不按步骤一一考虑，按所谓"经验"，胡乱的可能直接就这样定义F[i]了：

前i个数字里面的最大子数组和，结果输出 `F[len(nums)-1]` 就行了。

看起来似乎没毛病，但当你按这个定义时，状态转移方程怎么找?

举个例子，**nums = [4,-1,5] **这个数组，当你一开始选了4的时候，max更新为4，由于下一个是-1，4-1=3，比之前的小了，于是你放弃了选择-1，放弃了继续连续，于是你又重新开始，选了[3]，max = 5。这就是鼠目寸光，如果你继续选，max可以达到4-1+5=8。

于是，我们应该通览大局，这样定义F(n)：**以n为右端点的子数组的最大和。**

于是以每一个数字为右端点的子数组和就都有了，然后我们再考虑后面的结果加不加这个值。

状态转移方程为：

关键就在于，问题问的是数组和。要么**连续**，要么**重新开始、割舍一切**。所以，最大值不一定是**F(end)**，我们应该从**F（1..end)**里面寻找最大值，即为我们要求的最大子数组和。

代码如下 (Python)：

```python
nums = [0]+[-2,1,-3,4,-1,2,1,-5,4]
n = len(nums)
dp=[0]*n
for i in range(1,n):
    dp[i] = max(nums[i],nums[i]+dp[i-1])
res = 0
for i in range(n):
    res = max(res,dp[i]);

print(res)
```

再举一个乘积最大子数组的例子，和上面最大子数组和有些类似。这道题讲了在发现我们的dp状态**不满足最优子结构的情况下**如何思考优化，实际上就是上面总结的个人的思路。

> 给你一个整数数组 nums ，请你找出数组中乘积最大的非空连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

> 测试用例的答案是一个 32-位 整数。
>
> 子数组 是数组的连续子序列。
>
> 示例 1:
>
> 输入: nums = [2,3,-2,4]
> 输出: 6
> 解释: 子数组 [2,3] 有最大乘积 6。
> 示例 2:
>
> 输入: nums = [-2,0,-1]
> 输出: 0
> 解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

此时，把**F（n）**定义为以n为右端点的子数组最大乘积。

这里情况有所不同了，这里要求的是子数组的最大乘积，不是和。你可能会想，改一下转移方程就行了，把加改为乘。

求最大和，数字相加看看哪个更大就行，但求乘积，就是单纯的两数相乘看结果吗？如果有三个元素，前两个元素的乘积是负数，第三个数也是负数，再继续乘，负负得正，这样的情况下反而会出现如果前面两个元素的值越低，三个一起的乘积就越大。比如**nums = [3，-2，-3]**，按照上面的转移方程，得到的**F(3) = 6**，但实际上应该得到的**F(3)=18**

即此时F的最优解不能由**更小的规模**的F解得出，即这里我们的DP状态F(n)**不能满足最优子结构**。

于是我们不难发现，这里应该要再引入一个新的DP状态处理存在负数乘积的情况。我们令**minn（n）**表示以n为右端点的数字最小乘积，**maxn（n）**表示以n为右端点的数字最大乘积。

分类讨论：

> 当前位置如果是一个负数的话，它的最大值来源于前一个位置的最小数*当前数，前一个位置的最小数是负数的话自然负负得正，越负越大。前一个位置的最小数是正数的话也不矛盾，我们希望正数尽可能小，以使得当前位置的最大数取得最小。正数同理。（这里可能有点绕，可以自己写一写理解一下）

此时，就能得出以n-1为右端点的最大值与最小值，然后根据nums[n]是正数还是负数来分类进行状态转移。从而得出（1...end）的所有maxn值和minn值，最后我们输出maxn（1...end）里面最大的值即为最大数组乘积。

代码如下：

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n = len(nums)
        maxn = [0]*n
        minn = [0]*n
        maxn[0],minn[0] = [nums[0],nums[0]]
        for i in range(1,n):
            if nums[i]>0:
                maxn[i] = max(nums[i],maxn[i-1]*nums[i])
                minn[i] = min(nums[i],minn[i-1]*nums[i])
            else:
                maxn[i] = max(nums[i],minn[i-1]*nums[i])
                minn[i] = min(nums[i],maxn[i-1]*nums[i])

        return max(maxn)
```

接下来通过和 **买卖股票的最佳时机 III**和**122. 买卖股票的最佳时机 II**来讲述什么是满足无后效性，以及怎么处理让其满足无后效性。

1.买卖股票最佳时机II：

> 给定一个数组 prices ，其中 prices[i] 表示股票第 i 天的价格。
>
> 在每一天，你可能会决定购买和/或出售股票。你在任何时候 最多 只能持有 一股 股票。你也可以购买它，然后在 同一天 出售。
> 返回 你能获得的 最大 利润 。
>
> 示例 1:
>
> 输入: prices = [7,1,5,3,6,4]
> 输出: 7
> 解释: 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
> 随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出, 这笔交易所能获得利润 = 6-3 = 3 。
> 示例 2:
>
> 输入: prices = [1,2,3,4,5]
> 输出: 4
> 解释: 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
> 注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。因为这样属于同时参与了多

> 笔交易，你必须在再次购买前出售掉之前的股票。
> 示例 3:
>
> 输入: prices = [7,6,4,3,1]
> 输出: 0
> 解释: 在这种情况下, 没有交易完成, 所以最大利润为 0。

2.买卖股票最佳时机III：

> 给定一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。
>
> 设计一个算法来计算你所能获取的最大利润。你最多可以完成 两笔 交易。
>
> 注意：你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
>
> 示例 1:
>
> 输入：prices = [3,3,5,0,0,3,1,4]
> 输出：6
> 解释：在第 4 天（股票价格 = 0）的时候买入，在第 6 天（股票价格 = 3）的时候卖出，这笔交易所能获得利润 = 3-0 = 3 。
> 随后，在第 7 天（股票价格 = 1）的时候买入，在第 8 天 （股票价格 = 4）的时候卖出，这笔交易所能获得利润 = 4-1 = 3 。
> 示例 2:
>
> 输入：prices = [1,2,3,4,5]
> 输出：4
> 解释：在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出, 这笔交易所能获得利润 = 5-1 = 4 。
> 注意你不能在第 1 天和第 2 天接连购买股票，之后再将它们卖出。
> 因为这样属于同时参与了多笔交易，你必须在再次购买前出售掉之前的股票。

这两道题的区别在于一个能进行不限制次数的交易，一个只能进行两次交易。

我们先来看一下第一题，能进行不限次数的交易。

由于该题卖出的前提是要持有，容易知道这道题的状态就是天数n、是否持有股票(0 or 1)。

我们将

定义为 到第n天，卖出/持有股票的最大利润，我们规定0为不持有股票，1为持有股票，那么我们要求的最大利润自然就是

很容易理解，最后一天的时候，不持有股票一定要比持有股票的利润要多。（最后一天了，手中还搁置着股票不卖，这也太傻了）

状态转移方程如下：

很容易理解，当你第n天持有股票时，有两种情况：

1.第n-1天你就已经持有股票，继承n-1的情况。

2.第n-1天你没有持有股票，你要花费第n天股票的价值将其股票收购。

当你第n天没有持有股票时，也是有两种情况：

1.第n-1天你就没有持有股票，继承n-1的情况

2.第n-1天时你持有股票，第n天你把股票出售了。

在平常做题的过程中，只要问题满足最优子结构，我们就可以按照这个思路来思考状态转移方程：

> **题目要求F（n）的情况，那么我们默认已经知道了F（0）F（n-1）的值，现在就只需要去思考：怎么用F（0）F（n-1）的元素递推出F（n）的值，然后将这个过程转化为数字语言表示出来。**

完整代码如下：

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        m = len(prices)
        dp = [[-float('inf')]*2]*m
        dp[0][0] = 0
        dp[0][1] = -prices[0]
        for i in range(1,m):
            dp[i][0] = max(dp[i-1][0],dp[i-1][1]+prices[i])
            dp[i][1] = max(dp[i-1][1],dp[i-1][0]-prices[i])

        return dp[m-1][0]
```

我们来讲讲第二题，它给了一个限定条件K：你只能出售2次。

我们还是按第一题的定义DP状态吗？很明显不行了，因为我们有一个更多的限制条件K，必须按照这个规矩行事。

现在你应该有些理解无后效性了吧？

大问题的结果，我们只看小问题的答案，而不用考虑小问题的答案是如何得到的，这就是满足“无后效性”的问题

> “现在就是过去的总结，现在决定未来，未来与过去无关。”

如果我们还是按照第一题的定义来定义第二题的DP状态，很明显就不能满足“无后效性”，因为下一个阶段的结果不仅仅取决于上一天持有与否的最大利润，还取决于上一条是如何持有的，即K还剩几次。

如果K已经满足了2，下一个阶段K就将等于3，这时就不能按照第一题的转移方程递推得到下一阶段的答案。

这时我们就要多设一个状态K，将DP数组转为三维数组来考虑，这样K就会跟着状态改变，从而进一步缩小了问题的规模，可以从更多的状态（更为明确的范围）里面做出正确的选择，找出正确的状态转移方程。

我们将dp状态定义为

（注意：此处n为天数，0 or 1代表是否持有股票，**k为最大交易次数，而不是已交易次数，因为这样定义才可以满足我们的状态转移方程，因为不保证取得最大利润时已经交易了K次**）

状态转移方程如下：