ACM 奖获得者谈基础动态规划

基础动态规划浅淡

一、前言

许多人会这样问，动态规划到底是一种什么样的算法呢？在我看来，动态规划并不能说是一种特定的算法，而更多的被认为是一种思想，一种解决问题的手段。那动态规划到底难不难呢？难！！！作为算法竞赛的“宠儿”，动态规划很大程度上是对同学们建模分析能力的考察，而这种建模分析能力的形成，来自长期艰苦的训练，以及当时的灵感。那动态规划是否就遥不可及了呢？当然不是！！！对于基础的动态规划，我们完全可以在短时间内学会如何去思考和解决。

但冰冻三尺非一日之寒，任何东西的学习过程都需要付出艰苦的努力，本文的初衷旨在帮助大家理解基本的动态规划模型，助大家校招一臂之力，但对于动态规划的内容却只是浅尝则止。如果大家想要学习更高深的姿势，最好的方法还是在实际当中去不断地探索和总结。文中的每道例题我都给出了详细的题解和代码，希望大家在阅读过程中先自己独立思考，在无法独立完成的情况下在参照题解和AC代码。

二、关于动态规划

关于动态规划，我想大家听得最多的两个名词大概就是最优子结构和重叠子问题了吧。⑴最优子结构主要是指问题的最优解包含着子问题的最优解，即不管前面的策略的如何，此后的决策必须是基于当前状态（由上次决策产生）的最优决策。⑵重叠子问题是指每次递归产生的问题并不总是新问题，有些问题被反复计算多次，对每个子问题只计算一次，然后将其保存起来，以后在遇到同样的问题就直接引用，不必直接求解。

在这里，为了让大家更好的理解动态规划，我们并不关注概念本身，而是通过讲解一些经典模型，让大家能够更好的理解状态和状态转移，从而能够更好的学会去思考动态规划相关问题。

三、动态规划经典模型

0-1背包模型

首先我们通过一个经典的问题来带着大家看看什么是重叠子问题。

给定一个可以容纳重量为W的背包，现在有n种物品，每种物品有且仅有一个，同时每种物品的重量和价值分别为 w_i 和 v_i 。现在我们需要从这些物品当中挑选总重量不超过W的物品，要求选出所有挑选方案中价值总和最大的一种。

输入:两个整数n, W,分别表示物品种数和背包容量，然后接下来n行，每行2个整数，分别为 w_i 和 v_i ，表示每个物品的重量和价值。

输出:

一个整数，表示背包可以容下物品的最大价值。

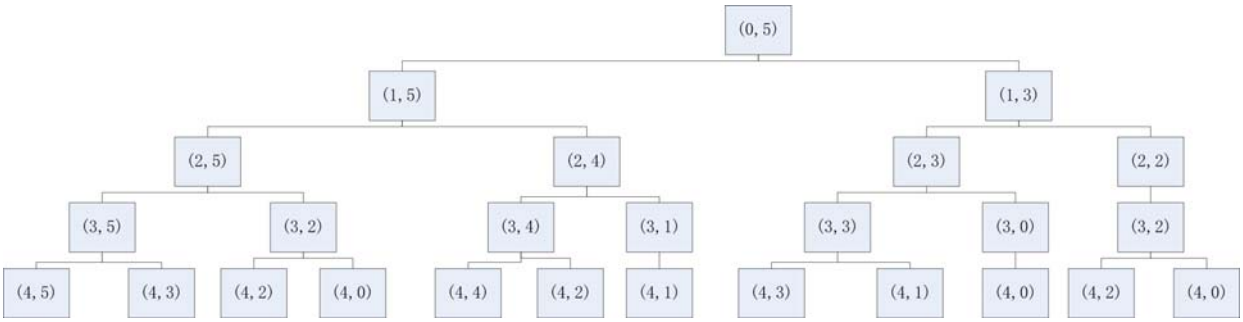
数据范围:

$1 \leq n \leq 100$
$1 \leq w_i, v_i \leq 100$
$1 \leq W \leq 10000$
样例：
Input:
4 5
2 3
1 2
3 4
2 2
Output:
7

分析：当我第一次拿到这个题目的时候我肯定会去想，这很简答呀，暴搜不就好了，把每个物品放进去，如果发现不合适在拿出来，最后总是能够找到一个放法使得其价值是最大的，于是我写出了下面的代码。

```
const int maxn=100+10;
int n,W;
int w[maxn],v[maxn];
int dfs(int i,int j){    //从第i个物品开始，挑选小于j的部分
    int res;
    if(i==n){    //已经没有剩余物品了
        res=0;
    }else if(j<w[i]){    //无法挑选这个物品
        res=dfs(i+1,j);
    }else{
        //挑选和不挑选两种情况都要尝试一下
        res=max(dfs(i+1,j),dfs(i+1,j-w[i])+v[i]);
    }
    return res;
}
```


那我们来分析一下这个算法，每个物品都有两种状态，取或者是不取，那n个物品，总共就有2^n种状态，那当n=100时，肯定是无法承受的复杂度，那怎么办，我们需要对算法做优化，怎么优化呢？下面我就来为大家介绍一下。



很显然，如图所示，图中第4层左右子树的(3,2)整个子树部分是被重复计算的，这像不像我们所说的重叠子结构呢？那我们能不能想种方法，把中间计算过的结果存下来，下次

使用的时候不用再去计算，而是直接调用呢？这样效率就会有很大的提升！！！

```
const int maxn=100+10;
int n,W;
int w[maxn],v[maxn];
int dp[maxn][maxn];
memset(dp,-1,sizeof(dp));
int dfs(int i,int j){    //从第i个物品开始，挑选小于j的部分
    if(dp[i][j]>=0)     //计算过就直接调用
        return dp[i][j];
    int res;
    if(i==n){    //已经没有剩余物品了
        res=0;
    }else if(j<w[i]){    //无法挑选这个物品
        res=dfs(i+1,j);
    }else{
        //挑选和不挑选两种情况都要尝试一下
        res=max(dfs(i+1,j),dfs(i+1,j-w[i])+v[i]);
    }
    return dp[i][j]=res;  //将它记录在数组中
}
```

因为dp[i][j]的状态总共只有$n \times W$种,所以整个的复杂度就是$O(n \times W)$，是不是在效率上有了很大的优化呢？纪录中间过程的方法是不是很有效呢？我们给这种方法一个名称，叫做记忆化搜索，在一定意义上，它就是动态规划！！！

接着我们可以把上面的方法用递推的方式改写一下，就得到了我们熟悉的01背包问题。

```
const int maxn=100+10;
int n,W;
int w[maxn],v[maxn];
int dp[maxn][maxn];
void solve()
{
    for(int i=1;i<=n;i++){
        for(int j=W;j>=w[i];j--){
            dp[i][j]=min(dp[i-1][j],dp[i-1][j-w[i]]+v[i]);
        }
    }
}
```

通过上面的介绍，我想大家基本上都理解01背包问题了吧，那下面我们再来看它的一个变形。

金明的预算方案

分析：题目意思应该不难理解， 唯一需要注意的如果我们要买附件的话，必须得先买该类的主件，然后才能购买该类的附件。那我们在想一想这题可否抽象成01背包模型呢？

当然是可以的。怎么抽象呢？这题我们可以看成依赖型01背包问题，对于每个主件，他的挂载可以有4种情况：(1)一个都不挂；(2)只挂1号附件；(3)只挂2号附件；(4)同时挂1号和2号附件。所以，这个是否我们就用数组v1[i]和p1[i]分别表示1号挂件的价格和重要度，同理用v2[i]和p2[i]分别表示2号挂件的价格和重要度，用v[i]和p[i]表示主件的价格和重要度。最后的答案就是4种情况的最大值。建议大家自己先做一做，如果有想不明白的，可以看看我的代码。

View my code

学会了如何计算依赖型的背包，那下面让我们再来学习一下如何去做含有判定关系的背包。

劲歌金曲

题意分析：因为是英文题，所以我在这里做一下翻译。一般我们到KTV通常会点一首歌，就是《劲歌金曲》，为什么呢？因为KTV在时间到的时候不会立马把歌切掉，而是会等这首歌放完。例如，我们在15秒的时候去唱一首2分钟的歌，实际上我们就多唱了105秒，融合了很多歌曲的劲歌金曲长达11分18秒，如果唱这首歌，相当于多唱了663秒！假设我们现在在KTV，距离结束还有t秒。你决定唱接下来的n首歌，在结束之前再唱一次《劲歌金曲》，使得唱的歌数量尽量多的情况下尽量晚的离开KTV。
分析：首先我们不要被数据范围吓到，因为数据范围是“纸老虎”，虽然说 $t \leq 10^9$，但是题目限定了每首歌长度不超过3分钟，而 $t \leq 180n + 678$。所以t并不是一个很大的数。同时我们需要知道一个问题，01背包是可以填满的，也就是说这n首歌选完之后有可能正好把时间耗完，这和题意是不符合的，所以我们应该让t-1。好了，准备工作都做完之后我们就可以来做这道题了。我们需要求的是在歌曲尽量多的情况下还要尽量完离开KTV，这时我们就应该用两个数组dp1和dp2来做，一个数组纪录歌曲数量，一个纪录离开时间。显然，我们需要在满足第一个数组的条件下，再去满足第二个数组，也就是说在01背包的时候我们只有在保证第一个条件尽量大的情形下再去保证第二个条件。这样我们就可以很轻松解决这个问题了。有不清楚的同学可以看看我的代码。

View my code

好了，通过上面的讲解我相信聪明的你对于01背包问题一定已经比较了解了，那就让我们来看看下面的问题吧。

最长不下降子序列模型

给定一个含有n个整数序列，$b_1, b_2, \ldots\ldots, b_n$,如果对于$i_1 < i_2 < \ldots < i_m$，有$b_{i1} \leq b_{i2} \ldots\ldots \leq b_{im}$,则称存在一个长度为m的不下降子序列。什么意思呢？让我们通过一个样例来解释一下这个过程，现在有如下序列:

13 7 9 16 38 24 37 18 44 19 21 22 63 15

对于下标$i_1 = 1, i_2 = 4, i_3 = 5, i_4 = 9, i_5 = 13$,满足$13 < 16 < 38 < 44 < 63$,所以不下降子序列长度为5。

对于下标：

$i_1 = 2$，$i_2 = 3$，$i_3 = 4$，$i_4 = 8$，$i_5 = 10$，$i_6 = 11, i_7 = 12, i_8 = 13$, 有$7 < 9 < 16 < 18 < 19 < 21 < 22 < 63$,所以不下降子序列的长度为8。

我们要做的就是从所有的不下降子序列当中选出长度最长的一个。下面我们来看看如何分析这个题目：

首先，子序列可以理解为删除0个或者多个数，其他数的顺序保持不变，那我们需要求的就是在满足条件的情况下被删除数最少的那一种。在这儿我们定义dp[i]为以i结尾的最长上升子序列的长度，前面我们说过，我们说过，我们重视状态和状态转移，而忽略那些生涩的定义。于是我们需要思考dp[i]由哪个状态转化而来呢？答案很简单嘛，我们思考，什么数可以作为i的前一个数呢？当然是在区间[1,i-1]当中不大于b_i的数。所以我们的状态转移方程也就可以很轻松的写出来了。

$dp[i] = max(0, dp[j] | j < i, b_j < b_i) + 1$，为什么这样写？因为$b_i$可以作为所有在他之前比他小的数的下一个元素，那我们当然从中选择长度最长的一个了。这是不是很像我们所说的最优子结构呢？哈哈，这样，这个问题我们就解决了。最后dp[n]就是我们要求的。

通过这个问题告诉我们在解决动态规划问题的时候，我们更多的应该关注状态和状态转移。那对于最长不下降子序列问题你是否掌握了呢？下面我们来看看他的一些变式。

导弹拦截

题意分析:这是一道很经典的题目，题目不仅仅要求我们求出最长下降子序列，同时还要求我们记录求解过程，也就是我们常说的需要记录路径，因为第2问，问我们需要多少个系统才能完成拦截。那这个题目又该如何去做呢？

分析：基于前面的讲解，我们已经知道如何求解最长不上升子序列了(最长不下降改改就好)，那我们怎么去记录路径呢？我们可以这样，用一个pre[]数组纪录他从哪里来，也就是纪录他的前一个元素是哪一个，最后我们在纪录一下每次的最后最长不下降子序列结束的那个元素，这样顺着最后那个元素往前去寻找，就可以很容易得出我们需要的路径。在本题中，我们可以通过不断求最长不上升子序列，然后每次我们把路径上的点删除了，直到最后把序列当中的元素全部删完，看需要几次操作，也就是我们要求的答案。为了方便代码的编写，这儿我们可以用一个vector来维护。大家可以试着编写一下这题的代码，如果有什么问题，可以参看我的代码。

View my Code

学会了如何纪录最长不下降子序列的路径，下面让我们看看另外一个相关的问题，有时候最长不下降子序列问题可以变为求”先升后降”或者是”先降后升”的问题。什么意思呢？也就是我们需要对序列的前一半求最长不下降子序列，对序列的后一半求最长不上升子序列，最后我们需要的结果是二者和的最大值，还是让我们再来看一个例题吧。

合唱队形

分析：题目要求我们求出$T_i < T_2 <...< T_i > T_{i+1} >...> T_k$的最长序列，怎么求呢？首先对于序列，我们有必要求出他的最长上升子序列，因为无论如何，我们都知道序列的前半部分是呈上升趋势的。而对于后半部分，我们需要求最长下降子序列？有人会问，这样怎么能够保证最大呢？别急，在这儿我们就需要改变一下动态规划的方向了。让dp[i]表示从第i个到第n个最长下降子序列的长度，那dp[i]这个状态就由区间[i+1,n]当中比i小的数转移过来，同理参照上面，我们选择其中长度最大的一组，这儿的求解需要我们逆推。如此，二者的和的最大值就是我们要求的最长合唱队形。具体的实现过程，可以参看我的代码。

View my code

最长不下降子序列还有一种$O(nlogn)$的做法，但改做法不利用路径的纪录，我会在以后的chat中详细讲述改做法，在这里不做展开了，感兴趣的同学可以查阅相关资料进行学习。

区间动态规划模型

与区间相关的动态规划问题也是我们经常遇到的，这类问题看似灵活多变，但只要掌握了其中的一些规律，这类问题并不会成为成为无法突破的“瓶颈”。这类问题的解法一般是设出一个表示状态的DP，一般是二维的。然后将问题划分为两个子问题，也就是将一段区间分成左右两个区间。最后将左右两个区间合并到整个区间。也就是我们所说的把局部最优解合并成为全局最优解。下面让我们来看一个例子。

最优矩阵链乘:

一个$n \times m$的矩阵由n行m列公$n \times m$个数排列而成。两个矩阵A和B可以相乘当且仅当A的列数等于B的行数。一个$n \times M$的矩阵乘以一个$m \times p$的矩阵，等于一个$n \times p$的矩阵，运算量为nmp。矩阵的乘法不满足分配律，但是矩阵乘法满足结合律，因此对于$A \times B \times C$既可以按顺序$(A \times B) \times C$进行，也可以按$A \times (B \times C)$进行。假设A、B、C分别是2×3，3×4，4×5的，则$(A \times B) \times C$运算量为$2 \times 3 \times 4 + 2 \times 4 \times 5 = 64$,$A \times (B \times C)$的运算量为$3 \times 4 \times 5 + 2 \times 3 \times 5 = 90$。

显然第一种顺序更节省运算量。给出一个n个矩阵组成的序列，设计一种方法把它们依次乘起来，使得总的运算量最小。假设第i个矩阵A_i是$p_{i-1} \times p_i$的。

分析：本题任务是设计一个表达式。在表达式中，一定有一个”最后一次乘法”，我们不妨设为第k个乘号，则在此之前已经算出了$P = A_1 \times A_2 \times \ldots \times A_k$和$Q = A_{k+1} \times A_{k+2} \times \ldots \times A_n$。显然P和Q的过程互相不干预，而且无论按照什么样的顺序，P和Q的值都不会发生改变，因此只需让P和Q分别按照最优的方案计算即可，这是不是就是我们常说的最优子结构呢？为了计算P的最优方案，还需要继续枚举$P = A_1 \times A_2 \times \ldots \times A_k$的最后一次乘法，把它分成两部分。不难发现，无论怎么分，在任意时候我们需要处理的子问题都是形如把$A_i, A_{i+1}, \ldots, A_j$乘起来需要多少次乘法？接着我们可以根据上面的定义来写状态转移方程了，首先我们定义dp[i][j]为从i到j需要做多少次乘法，则根据我们上面的分析，我们知道需要把区间(i,j)分为两部分(i,k)和(k+1,j),我们最后的答案为这两部分的和。所以我们就不难写出状态转移方程:

$dp[i][j] = min(dp[i][k] + dp[k+1][j] + p_{i-1}p_kp_j)$,边界是dp[i][i]=0。这样我们就能够在$O(n^3)$的复杂度内解决这个问题。最后dp[1][n]就是所求。区间动态规划问题大抵就是上面的套路，你明白了么？下面还是让我们来看一个实际的例子吧。

切木棍

题意分析：因为是英文题目，我在这里大致为大家翻译一下。有一根长度为L的木棍，棍子上有n个切割点的位置(按照从小到大排列)。你的任务就是在这些切割点的位置处把棍子切成n+1部分，使得总切割费用最小。每次切割的费用等于被切割的木棍长度。例如L=10；，切割点为2，4，7.如果按照2，4，7的顺序，费用为10+8+6=24，如果按照4，2，7的顺序，费用为10+4+6=20。

分析：这个题目是不是跟我们刚才那个题目很像呢？没错，这类题看上去都是那样的似曾相识。那我们是否可以沿用前面的模型呢？当然没问题，但是这个题有一个需要注意的，在我们的第0个切分点是0，第n+1个切分点应该设为L，为什么这样做？因为后面在算切割费用的时候会用到。接下来就上我们的模型吧，设dp[i][j]为切割小木棍i~j的费用，则根据前面的分析我们可以知道，这时需要把区间分为两部分。则$dp[i][j] = min(dp[i][k] + dp[k][j] + a[j] - a[i] | i < k < j)$，注意在这里$a[j] - a[i]$代表第一刀的费用。在切完这一刀之后，小木棍立马变成了i~k,k~j两部分。状态转移方程由此可以得到。所以最后dp[0][n+1]就是所求。那么你会做这题了么？如果还有什么不明白的，可以参看我的代码。

View my code

通过上面的过程你是否对区间动态规划问题有一点感觉了呢？那就让我们趁热打铁，再练习一题，完全拿下它吧。

括号序列

题意分析：同样因为这题是英文题，我再为大家翻译一下。定义如下的序列是好的括号序列:

1. 空序列是好的括号序列。

2. 如果S是好的括号序列，那么(s)或者[s]都是好的括号序列。

3. 如果A和B都是好的括号序列，则AB也是好的括号序列。

现在输入一个由不超过100个字符组成的括号序列，仅有[、]、(、)构成，添加尽量少的括号，使其能够成为一个好的括号序列。并输出最终的结果。如果有多解，输出任意一个即可。

分析：这题看上去似乎跟前面的题目不太一样，但是当我们仔细分析下来，发现其实和前面的题目如出一辙。首先我们设字符串S最少需要添加d(S)个括号会变成一个好的括号序列。那么就会出现如下两种情况。

1. 如果S形如(S')或者[S']，则转移到d(S')。

2. 如果有至少两个字符，则可以分成AB，转移到d(A)+d(B)。

边界情况是:S为空时,d(S)=0,S为单个字符时，d(S)=1。

这里有一个特别需要注意的就是无论S是否满足第一条，都要尝试第二种转移，为什么呢？让我们来看一个例子，比如:S=[][],按照规则1的话，我们会转移到状态][，则这时需要添加的括号是两个，但是按照题意，我们最少需要添加的括号是0个，所以答案错误。

那么根据第2条规则，我们可以把串分成A和B两部分，是不是很像我们的区间动态规划呢？没错，这就是我们的区间动态规划。用dp[i][j]表示子串s[i~j]至少需要添加几个括号，则dp[i][j]=min(dp[i][k]+dp[k+1][j]),哈哈，似乎又回到了我们的模型上面。这样这个问题就解决了大半了。剩下部分我们要怎么解决呢？就是路径打印嘛，在这里我们选择一种“笨办法”，我们之前不是已经计算过dp[][]数组了嘛，那我们就仿照dp数组的过程再走一遍，就可以很容易打印出路径。具体实现可以参看我的代码

View my code

通过上面的学习，你应该对区间动态规划模型有所了解了吧。

四、结束语

好了，弱菜今天给大家带来的东西就是这些了。但动态规划博大精深，要想成为一个真正的动态规划高手，仅仅掌握我今天讲的内容是远远不够的。这篇文章旨在让大家对动态规划有个大致的概念，以及加深大家对动态规划的基本模型的理解，希望在校招当中出现类似的动态规划问题时，大家能够迎刃而解。

如果您对文中内容有任何问题或者您对算法感兴趣，欢迎跟我交流。后期我预计还会推出动态规划进阶和其他算法相关的专题，如果大家有什么好的建议或者有什么想学习的专题，也欢迎跟我留言。我的邮箱是：wanghan19940509@gmail.com。最后祝愿大家工作顺利，学习进步，前程似锦。

参考资料

- 《算法竞赛入门经典(第二版)》
- 《挑战程序设计竞赛》
- 《朱全民动态规划》
- 《背包九讲》