动态规划算法的优化技巧

福州第三中学　　毛子青

[关键词] 动态规划、 时间复杂度、优化、状态

[摘要]

动态规划是信息学竞赛中一种常用的程序设计方法，本文着重讨论了运用动态规划思想解题时时间效率的优化。全文分为四个部分，首先讨论了动态规划时间效率优化的可行性和必要性，接着给出了动态规划时间复杂度的决定因素，然后分别阐述了对各个决定因素的优化方法，最后总结全文。

[正文]

一、引言

动态规划是一种重要的程序设计方法，在信息学竞赛中具有广泛的应用。

使用动态规划方法解题，对于不少问题具有空间耗费大、时间效率高的特点，因此人们在研究动态规划解题时更多的注意空间复杂度的优化，运用各种技巧将空间需求控制在软硬件可以承受的范围之内。但是，也有一部分问题在使用动态规划思想解题时，时间效率并不能满足要求，而且算法仍然存在优化的余地，这时，就需要考虑时间效率的优化。

本文讨论的是在确定使用动态规划思想解题的情况下，对原有的动态规划解法的优化，以求降低算法的时间复杂度，使其能够适用于更大的规模。

二、动态规划时间复杂度的分析

使用动态规划方法解题，对于不少问题之所以具有较高的时间效率，关键在于它减少了“**冗余**”。所谓“冗余”，就是指不必要的计算或重复计算部分，算法的冗余程度是决定算法效率的关键。动态规划在将问题规模不断缩小的同时，记录已经求解过的子问题的解，充分利用求解结果，避免了反复求解同一子问题的现象，从而减少了冗余。

但是，动态规划求解问题时，仍然存在冗余。它主要包括：求解无用的子问题，对结果无意义的引用等等。

下面给出动态规划时间复杂度的决定因素：

时间复杂度=状态总数*每个状态转移的状态数*每次状态转移的时间[1]

下文就将分别讨论对这三个因素的优化。这里需要指出的是：这三者之间不是相互独立的，而是相互联系，矛盾而统一的。有时，实现了某个因素的优化，另外两个因素也随之得到了优化；有时，实现某个因素的优化却要以增大另一因素为代价。因此，这就要求我们在优化时，坚持“全局观”，实现三者的平衡。

三、动态规划时间效率的优化

3.1 减少状态总数

我们知道，动态规划的求解过程实际上就是计算所有状态值的过程，因此状态的规模直接影响到算法的时间效率。所以，减少状态总数是动态规划优化的重要部分，本节将讨论减少状态总数的一些方法。

1、改进状态表示

状态的规模与状态表示的方法密切相关，通过改进状态表示减小状态总数是应用较为普遍的一种方法。

例一、 Raucous Rockers 演唱组（USACO`96）

[问题描述]

现有 n 首由 Raucous Rockers 演唱组录制的珍贵的歌曲，计划从中选择一些歌曲来发行 m 张唱片，每张唱片至多包含 t 分钟的音乐，唱片中的歌曲不能重叠。按下面的标准进行选择：

（1） 这组唱片中的歌曲必须按照它们创作的顺序排序；

（2） 包含歌曲的总数尽可能多。

输入 n，m，t，和 n 首歌曲的长度，它们按照创作顺序排序，没有一首歌超出一张唱片的长度，而且不可能将所有歌曲的放在唱片中。输出所能包含的最多的歌曲数目。

（1≤n, m, t≤20）

[算法分析]

本题要求唱片中的歌曲必须按照它们创作顺序排序，这就满足了动态规划的无后效性要求，启发我们采用动态规划进行解题。

分析可知，该问题具有最优子结构性质，即：设最优录制方案中第 i 首歌录制的位置是从第 j 张唱片的第 k 分钟开始的，那么前 j-1 张唱片和第 j 张唱片的前 k-1 分钟是前 1..i-1 首歌的最优录制方案，也就是说，问题的最优解包含了子问题的最优解。

设 n 首歌曲按照写作顺序排序后的长度为 long[1..n]，则动态规划的状态表示描述为：

g[i, j, k]，0≤i≤n，0≤j≤m，0≤k<t，表示前 i 首歌曲，用 j 张唱片另加 k 分钟来录制，最多可以录制的歌曲数目，则问题的最优解为 g[n,m,0]。由于歌曲 i 有发行和不发行两种情况，而且还要分另加的 k 分钟是否能录制歌曲 i。这样我们可以得到如下的状态转移方程和边界条件：

当 k≥long[i]，i≥1 时：

g[i, j, k]=max{g[i-1,j,k-long[i]]，g[i-1,j,k]}

当 k<long[i]，i≥1 时：

g[i, j, k]=max{g[i-1,j-1,t-long[i]]，g[i-1,j,k]}

规划的边界条件为：

当 0≤k<t 时：g[0,0,k]=0;

我们来分析上述算法的时间复杂度，上述算法的状态总数为 O(n*m*t)，每个状态转移的状态数为 O(1)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 O(n*m*t)。由于 n，m，t 均不超过 20，所以可以满足要求。

[算法优化]

当数据规模较大时，上述算法就无法满足要求，我们来考虑通过改进状态表示提高算法的时间效率。

本题的最优目标是用给定长度的若干张唱片录制尽可能多的歌曲，这实际上等价于在录制给定数量的歌曲时尽可能少地使用唱片。所谓“尽可能少地使用唱片”，就是指使用的完整的唱片数尽可能少，或是在使用的完整的唱片数相同的情况下，另加的分钟数尽可能少。分析可知，在这样的最优目标之下，该问题同样具有最优子结构性质，即：设 D 在前 i 首歌中选取 j 首歌录制的最少唱片使用方案，那么若其中选取了第 i 首歌，则 D-{i}是在前 i-1 首歌中选取 j-1 首歌录制的最少唱片使用方案，否则 D 前 i-1 首歌中选取 j 首歌录制的最少唱片使用方案，同样，问题的最优解包含了子问题的最优解。

改进的状态表示描述为：

g[i, j]=(a, b)，0≤i≤n，0≤j≤i，0≤a≤m，0≤b≤t，表示在前 i 首歌曲中选取 j 首录制所需的最少唱片为：a 张唱片另加 b 分钟。由于第 i 首歌分为发行和不发行两种情况，这样我们可以得到如下的状态转移方程和边界条件：

g[i, j]=min{g[i-1,j]，g[i-1,j-1]+long[i]}

其中(a, b)+long[i]=(a’, b’)的计算方法为：

当 long[i]≤t-b 时： a’=a;　　b’=b+long[i];

当 long[i]>t-b 时： a’=a+1;　 b’=long[i];

规划的边界条件：

g[i,0]=(0,0)　0≤i≤n

这样题目所求的最大值是：ans=max{k| g[n, k]≤(m-1,t)}

改进后的算法，状态总数为 $O(n^2)$，每个状态转移的状态数为 O(1)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 $O(n^2)$。值得注意的是，算法的空间复杂度也由改进前的 O(m*n*t)降至优化后的 $O(n^2)$。

（程序及优化前后的运行结果比较见附件）

通过对本题的优化，我们认识到：应用不同的状态表示方法设计出的动态规划算法的性能也迥然不同。改进状态表示可以减少状态总数，进而降低算法的时间复杂度。在降低算法的时间复杂度的同时，也降低了算法的空间复杂度。因此，减少状态总数在动态规划的优化中占有重要的地位。

2、选择适当的规划方向

动态规划方法的实现中，规划方向的选择主要有两种：顺推和逆推。在有些情况下，选取不同的规划方向，程序的时间效率也有所不同。一般地，若初始状态确定，目标状态不确定，则应考虑采用顺推，反之，若目标状态确定，而初始状态不确定，就应该考虑采用逆推。那么，若是初始状态和目标状态都已确定，一般情况下顺推和逆推都可以选用，但是，能否考虑选用双向规划呢？

双向搜索的方法已为大家所熟知，它的主要思想是：在状态空间十分庞大，而初始状态和目标状态又都已确定的情况下，由于扩展的状态量是指数级增长的，于是为了减少状态的规模，分别从初始状态和目标状态两个方向进行扩展，并在两者的交汇处得到问题的解。

上述优化思想能否也应用到动态规划之中呢？来看下面这个例子。

例二、 Divide (Merc`2000)

[问题描述]

有价值分别为 1..6 的大理石各 a[1..6]块，现要将它们分成两部分，使得两部分价值和相等，问是否可以实现。其中大理石的总数不超过 20000。(英文试题详见附件)

[算法分析]

令 S=Σ(i*a[i])，若 S 为奇数，则不可能实现，否则令 Mid=S/2，则问题转化为能否从给定的大理石中选取部分大理石，使其价值和为 Mid。

这实际上是母函数问题，用动态规划求解也是等价的。

m[i, j]，0≤i≤6，0≤j≤Mid，表示能否从价值为 1..i 的大理石中选出部分大理石，使其价值和为 j，若能，则用 true 表示，否则用 false 表示。则状态转移方程为：

m[i, j]=m[i, j]　OR　m[i-1,j-i*k]　　(0≤k≤a[i])

规划的边界条件为：m[i,0]=true；　0≤i≤6

若 m[i, Mid]=true，0≤i≤6，则可以实现题目要求，否则不可能实现。

我们来分析上述算法的时间性能，上述算法中每个状态可能转移的状态数为 a[i]，每次状态转移的时间为 O(1)，而状态总数是所有值为 true 的状态的总数，实际上就是母函数中项的数目。

[算法优化]

实践发现：本题在 i 较小时，由于可选取的大理石的价值品种单一，数量也较少，因此值为 true 的状态也较少，但随着 i 的增大，大理石价值品种和数量的增多，值为 true 的状态也急剧增多，使得规划过程的速度减慢，影响了算法的时间效率。

另一方面，我们注意到我们关心的仅是能否得到价值和为 Mid 的值为 true 的状态，那么，我们能否从两个方向分别进行规划，分别求出从价值为 1..3 的大理石中选出部分大理石所能获得的所有价值和，和从价值为 4..6 的大理石中选出部分大理石所能获得的所有价值和。最后通过判断两者中是否存在和为 Mid 的价值和，由此，可以得出问题的解。

状态转移方程改进为：

当 i≤3 时：

m[i, j]=m[i, j]　OR　m[i-1,j-i*k]　　(1≤k≤a[i])

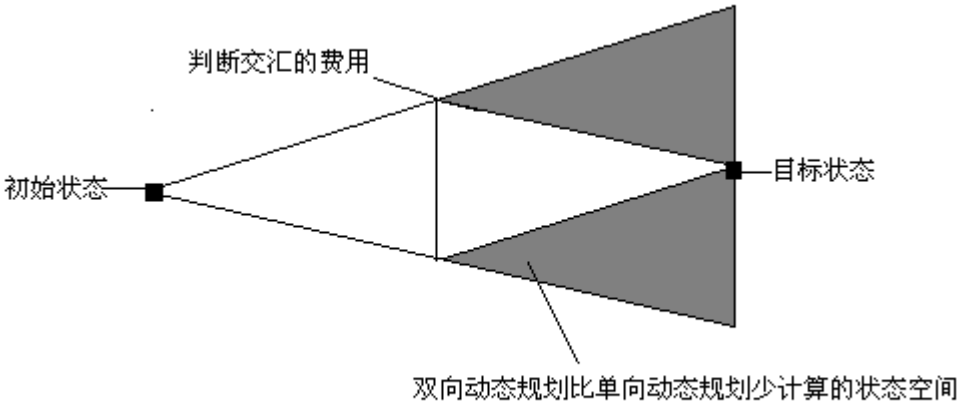
当 i>3 时：

m[i, j]=m[i, j]　OR　m[i+1,j-i*k]　　(1≤k≤a[i])

规划的边界条件为：m[i,0]=true；　0≤i≤7

这样，若存在 k，使得 m[3,k]=true, m[4,Mid-k]=true，则可以实现题目要求，否则无法实现。

（程序及优化前后的运行结果比较见附件）



从上图可以看出双向动态规划与单向动态规划在计算的状态总数上的差异。

回顾本题的优化过程可以发现：本题的实际背景与双向搜索的背景十分相似，同样有庞大的状态空间，有确定的初始状态和目标状态，状态量都迅速增长，而且可以实现交汇的判断。因此，由本题的优化过程，我们认识到，双向扩展以减少状态量的方法不仅适用于搜索，同样适用于动态规划。这种在不同解题方法中，寻找共通的属性，从而借用相同的优化思想，可以使我们不断创造出新的方法。

3.2　减少每个状态转移的状态数

在使用动态规划方法解题时，对当前状态的计算都是进行一些决策并引用相应的已经计算过的状态，这个过程称为“状态转移”。因此，每个状态可能做出的决策数，也就是每

个状态可能转移的状态数是决定动态规划算法时间复杂度的一个重要因素。本节将讨论减少每个状态可能转移的状态数的一些方法。

1、四边形不等式和决策的单调性

例三、石子合并问题(NOI`95)

[问题描述]

在一个操场上摆放着一排 n（n≤20）堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

试编程求出将 n 堆石子合并成一堆的最小得分和最大得分以及相应的合并方案。

[算法分析]

这道题是动态规划的经典应用。由于最大得分和最小得分是类似的，所以这里仅对最小得分进行讨论。设 n 堆石子依次编号为 1，2，…..，n。各堆石子数为 d[1..n]，则动态规划的状态表示为：

m[i,j]，1≤i≤j≤n，表示合并 d[i..j]所得到的最小得分，则状态转移方程和边界条件为：

m[i,j]=0　　i=j

$$m[i,j]=\min_{i<k\le j}\{m[i,k-1]+m[k,j]+\sum_{l=i}^{j}d[l]\}$$　　i<j

同时令 s[i,j]=k，表示合并的断开位置，便于在计算出最优值后构造出最优解。

上式中 $\sum_{l=i}^{j}d[l]$ 的计算，可在预处理时计算 $t[i]=\sum_{j=1}^{i}d[j]$，i=1..n；t[0]=0，则：

$$\sum_{l=i}^{j}d[l]=t[j]-t[i-1]$$

上述算法的状态总数为 $O(n^2)$，每个状态转移的状态数为 O(n)，每次状态转移的时间为 O(1)，所以总的时间复杂度为 $O(n^3)$。

[算法优化]

当函数 w[i,j]满足 $w[i,j]+w[i',j']\le w[i',j]+w[i,j'],i\le i'\le j\le j'$ 时，称 w 满足四边形不等式[2]。

当函数 w[i,j]满足 w[i′,j]≤w[i,j′] $,i\le i'\le j\le j'$ 时称 w 关于区间包含关系单调。

在石子归并问题中，令 w[i,j]= $\sum_{l=i}^{j}d[l]$，则 w[i,j]满足四边形不等式，同时由 d[i]≥0,t[i]≥0 可知 w[i,j]满足单调性。

m[i,j]=0　　i=j

$$m[i,j]=\min_{i<k\le j}\{m[i,k-1]+m[k,j]\}+w[i,j]$$　　i<j　　……①

对于满足四边形不等式的单调函数 w，可推知由递推式①定义的函数 m[i,j]也满足四边形不等式，即 $m[i,j]+m[i',j']\le m[i',j]+m[i,j'],i\le i'\le j\le j'$。这一性质可用数学归

纳法证明如下：

我们对四边形不等式中“长度”l=j′-i 进行归纳：

当 i=i′或 j=j′时，不等式显然成立。由此可知，当 l≤1 时，函数 m 满足四边形不等式。

下面分两种情形进行归纳证明：

情形 1：i<i′=j<j′

在这种情形下，四边形不等式简化为如下的反三角不等式：m[i,j]+m[j,j′] ≤m[i,j′]，设 k=max{p | m[i,j′]=m[i,p-1]+m[p,j′]+w[i,j′] }，再分两种情形 k≤j 或 k>j。下面只讨论 k≤j，k>j 的情况是类似的。

情形 1.1：k≤j，此时：

$$m[i,j]+m[j,j']\le w[i,j]+m[i,k-1]+m[k,j]+m[j,j']$$
$$\le w[i,j']+m[i,k-1]+m[k,j]+m[j,j']$$
$$\le w[i,j']+m[i,k-1]+m[k,j']$$
$$= m[i,j']$$

情形 2：i<i′<j<j′

设 y=max{p | m[i′,j]=m[i′,p-1]+m[p,j]+w[i′,j] }

z=max{p | m[i,j′]=m[i,p-1]+m[p,j′]+w[i,j′] }

仍需再分两种情形讨论，即 z≤y 或 z>y。下面只讨论 z≤y，z>y 的情况是类似的。

由 i<z≤y≤j 有：

$$m[i,j]+m[i',j']\le w[i,j]+m[i,z-1]+m[z,j]+w[i',j']+m[i',y-1]+m[y,j']$$
$$\le w[i,j']+w[i',j]+m[i',y-1]+m[i,z-1]+m[z,j]+m[y,j']$$
$$\le w[i,j']+w[i',j]+m[i',y-1]+m[i,z-1]+m[y,j]+m[z,j']$$
$$= m[i,j']+m[i',j]$$

综上所述，m[i,j]满足四边形不等式。

令 s[i,j]=max{k | m[i,j]=m[i,k-1]+m[k,j]+w[i,j] }

由函数 m[i,j]满足四边形不等式可以推出函数 s[i,j]的单调性，即

s[i,j]≤s[i,j+1]≤s[i+1,j+1],　　i≤j

当 i=j 时，单调性显然成立。因此下面只讨论 i<j 的情形。由于对称性，只要证明 s[i,j]≤s[i,j+1]。

令 m_k[i,j]=m[i,k-1]+m[k,j]+w[i,j]。要证明 s[i,j]≤s[i,j+1]，只要证明对于所有 i<k≤k′≤j 且 $m_{k'}$[i,j]≤m_k[i,j]，有：$m_{k'}$[i,j+1]≤m_k[i,j+1]。

事实上，我们可以证明一个更强的不等式

m_k[i,j]-$m_{k'}$[i,j]≤m_k[i,j+1]-$m_{k'}$[i,j+1]

也就是：　　m_k[i,j]+$m_{k'}$[i,j+1]≤m_k[i,j+1]+$m_{k'}$[i,j]

利用递推定义式将其展开整理可得：m[k,j]+m[k′,j+1]≤m[k′,j]+m[k,j+1]，这正是 k≤k′≤j<j+1 时的四边形不等式。

综上所述，当 w 满足四边形不等式时，函数 s[i,j]具有单调性。

于是，我们利用 s[i,j]的单调性，得到优化的状态转移方程为：

m[i,j]=0　　i=j

$$m[i,j]=\min_{s[i,j-1]\le k\le s[i+1,j]}\{m[i,k-1]+m[k,j]\}+w[i,j]$$　　i<j

用类似的方法可以证明，对于最大得分问题，也可采用同样的优化方法。

改进后的状态转移方程所需的计算时间为

$$O\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}(1+s[i+1,j]-s[i,j-1])\right)=O\left(\sum_{i=1}^{n-1}(n-i+s[i+1,n]-s[1,n-i])\right)=O(n^2)$$

（程序及优化前后的运行结果比较见附件）

上述方法利用四边形不等式推出最优决策的单调性，从而减少每个状态转移的状态数，降低算法的时间复杂度。

上述方法是具有普遍性的。对于状态转移方程与①式类似，且 w[i,j]满足四边形不等式的动态规划问题，都可以采用相同的优化方法，如最优二叉排序树（NOI`96）等。下面再举一例。

例四、邮局(IOI`2000)

[问题描述]

按照递增顺序给出一条直线上坐标互不相同的 n 个村庄，要求从中选择 p 个村庄建立邮局，每个村庄使用离它最近的那个邮局，使得所有村庄到各自所使用的邮局的距离总和最小。

试编程计算最小距离和，以及邮局建立方案。

[算法分析]

本题也是一道动态规划问题，详细分析请看文本附件（邮局解题报告）。将 n 个村庄按坐标递增依次编号为 1，2，……，n，各个邮局的坐标为 d[1..n]，状态表示描述为：m[i,j]表示在前 j 个村庄建立 i 个邮局的最小距离和。所以，m[p,n]即为问题的解，且状态转移方程和边界条件为：

m[1,j]=w[1,j]

$$m[i,j]=\min_{i-1\le k\le j-1}\{m[i-1,k]+w[k+1,j]\}\qquad i\leqslant j$$

其中 w[i,j]表示在 d[i..j]之间建立一个邮局的最小距离和，可以证明，当仅建立一个邮局时，最优解出现在中位数，即设建立邮局的村庄为 k，则 $k=\lfloor (i+j)/2\rfloor$或$k=\lceil (i+j)/2\rceil$，于是，我们有：

$$w[i,j]=\sum_{l=i}^{j}|d[l]-d[k]|\quad ,\qquad k=\lfloor (i+j)/2\rfloor 或 k=\lceil (i+j)/2\rceil$$

同时，令 s[i,j]=k，记录使用前 i-1 个邮局的村庄数，便于在算出最小距离和之后构造最优建立方案。

上述算法中 w[i,j]可通过 O(n)时间的预处理，在 O(1)的时间内算出，所以，该算法的状态总数为 O(n*p)，每个状态转移的状态数为 O(n)，每次状态转移的时间为 O(1)，该算法总的时间复杂度为 $O(p*n^2)$。

[算法优化]

本题的状态转移方程与①式十分相似，因此我们猜想其决策是否也满足单调性，即

s[i-1,j]≤s[i,j]≤s[i,j+1]

首先，我们来证明函数 w 满足四边形不等式，即：

$$w[i,j]+w[i',j']\le w[i',j]+w[i,j']\ \ ,i\le i'\le j\le j'$$

设 $y=\lfloor (i'+j)/2\rfloor$，$z=\lfloor (i+j')/2\rfloor$，下面分为两种情形，z≤y 或 z>y，下面仅讨论 z≤y，z>y 的情况是类似的。

由 i≤z≤y≤j 有：

$$w[i,j]+w[i',j'] \le \sum_{l=i}^{j}|d[l]-d[z]|+\sum_{l=i'}^{j'}|d[l]-d[y]|$$

$$\le \sum_{l=i}^{j}|d[l]-d[z]|+\sum_{l=i'}^{j'}|d[l]-d[y]|+\sum_{l=j+1}^{j'}|d[l]-d[z]|-\sum_{l=j+1}^{j'}|d[l]-d[y]|$$

$$=\sum_{l=i}^{j'}|d[l]-d[z]|+\sum_{l=i'}^{j}|d[l]-d[y]|$$

$$=w[i',j]+w[i,j']$$

接着，我们用数学归纳法证明函数 m 也满足四边形不等式。对四边形不等式中“长度”l=j′-i 进行归纳：

当 i=i′或 j=j′时，不等式显然成立。由此可知，当 l≤1 时，函数 m 满足四边形不等式。

下面分两种情形进行归纳证明：

情形 1：i<i′=j<j′，即 m[i,j]+m[j,j′] ≤m[i,j′]，

设 k=max{p | m[i,j′]=m[i,p-1]+m[p,j′]+w[i,j′] }，再分两种情形 k≤j 或 k>j。下面只讨论 k≤j，k>j 的情况是类似的。

$$m[i,j]+m[j,j']$$

$$\le m[i-1,k]+w[k+1,j]+m[j-1,j-1]+w[j,j']$$

$$\le m[i-1,k]+w[k+1,j']$$

$$=m[i,j']$$

情形 2：i<i′<j<j′

设　y=max{p | m[i′,j]=m[i′-1,p]+w[p+1,j] }

　　z=max{p | m[i,j′]=m[i-1,p]+w[p+1,j′] }

仍需再分两种情形讨论，即 z≤y 或 z>y。

情形 2.1，当 z≤y<j<j′时：

$$m[i,j]+m[i',j']$$

$$\le m[i'-1,y]+w[y+1,j']+m[i-1,z]+w[z+1,j]$$

$$\le m[i'-1,y]+m[i-1,z]+w[y+1,j]+w[z+1,j']$$

$$=m[i',j]+m[i,j']$$

情形 2.2，当 i-1<i′-1≤y<z<j′时：

$$m[i,j]+m[i',j']$$

$$\le m[i-1,y]+w[y+1,j]+m[i'-1,z]+w[z+1,j']$$

$$\le m[i-1,z]+m[i'-1,y]+w[y+1,j]+w[z+1,j'];$$

$$=m[i,j']+m[i',j]$$

最后，我们证明决策 s[i,j]满足单调性。

为讨论方便，令 $m_k[i,j]=m[i-1,k]+w[k+1,j]$；

我们先来证明 s[i-1,j]≤s[i,j]，只要证明对于所有 i≤k<k′<j 且 $m_{k'}[i-1,j]\le m_k[i-1,j]$，有：$m_{k'}[i,j]\le m_k[i,j]$。

类似地，我们可以证明一个更强的不等式

$$m_k[i-1,j]-m_{k'}[i-1,j]\le m_k[i,j]-m_{k'}[i,j]$$

也就是：

$$m_k[i-1,j]+m_{k'}[i,j]\le m_k[i,j]+m_{k'}[i-1,j]$$

利用递推定义式展开整理的：m[i-2,k]+m[i-1,k′]≤m[i-1,k]+m[i-2,k′]，这就是 i-2<i-1<k<k′时 m 的四边形不等式。

我们再来证明 s[i,j]≤s[i,j+1]，与上文类似，设 k<k′<j，则我们只需证明一个更强的不等式：　　$m_k[i,j]-m_{k'}[i,j]\leq m_k[i,j+1]-m_{k'}[i,j+1]$

也就是：　　$m_k[i,j]+m_{k'}[i,j+1]\leq m_k[i,j+1]+m_{k'}[i,j]$

利用递推定义式展开整理的：w[k+1,j]+w[k′+1,j+1]≤w[k+1,j+1]+w[k′+1,j]，这就是 k+1<k′+1<j<j+1 时 w 的四边形不等式。

综上所述，该问题的决策 s[i,j]具有单调性，于是优化后的状态转移方程为：

m[1,j]=w[1,j]

$$m[i,j]=\min_{s[i-1,j]\leq k\leq s[i,j+1]}\{m[i-1,k]+w[k+1,j]\}\qquad i\leqslant j$$

s[i,j]=k

同上文所述，优化后的算法时间复杂度为 O(n*p)。

（程序及优化前后的运行结果比较见附件）

四边形不等式优化的实质是对结果的充分利用。它通过分析状态值之间的特殊关系，推出了最优决策的单调性，从而在计算当前状态时，利用已经计算过的状态所做出的最优决策，减少了当前的决策量。这就启发我们，在应用动态规划解题时，不仅可以实现状态值的充分利用，也可以实现最优决策的充分利用。这实际上是从另一个角度实现了“减少冗余”。

2、决策量的优化

通过分析问题最优解所具备的性质，从而缩小有可能产生最优解的决策集合，也是减少每个状态可能转移的状态数的一种方法。

大家所熟悉的 NOI`96 中的添加号问题，正是从“所得的和最小”这一原则出发，仅在等分点的附近添加号，从而大大减少了每个状态转移的状态数，降低了算法的时间复杂度。让我们在再来看一例。

例五、石子归并的最大得分问题

[问题描述]

见例三，本例只考虑最大得分问题。

[算法分析]

设 n 堆石子依次编号为 1，2，…..，n。各堆石子数为 d[1..n]，则动态规划的状态表示为：

m[i,j]，1≤i≤j≤n，表示合并 d[i..j]所得到的最大得分，则状态转移方程和边界条件为：

m[i,j]=0　　i=j

$$m[i,j]=\max_{i<k\leq j}\{m[i,k-1]+m[k,j]+\sum_{l=i}^{j}d[l]\}\qquad i<j$$

同时令 s[i,j]=k，表示合并的断开位置，便于在计算出最优值后构造出最优解。

该算法的时间复杂度为 $O(n^3)$。

[算法优化]

仔细分析问题，可以发现：s[i,j]要么等于 i+1，要么等于 j，即：

$$\max_{i<k\leq j}\{m[i,k-1]+m[k,j]\}=\max\{m[i,j-1],m[i+1,j]\},i<j$$

证明可以采用反证法，设使 m[i,j]达到最大值的断开位置为 p，且 i+1<p<j，$y=\sum_{l=i}^{p-1}a[l]$，$z=\sum_{l=p}^{j}a[l]$，下面分为 2 种情形讨论。

情形 1、y≥z

由 p<j，可设 s[p,j]=k，则相应的合并方式可以表示为：

(　(a[i]…a[p-1])　((a[p]...a[k-1]) (a[k]..a[j]))　)

相应的得分为：$T=m[i,p-1]+m[p,k-1]+m[k,j]+y+z+z$ ……①

下面考虑另一种合并方案 s′[i,j]=k，s′[i,k]=p，表示为：

(((a[i]…a[p-1]) (a[p]...a[k-1]))　(a[k]..a[j])　)

相应的得分为：$T'=m[i,p-1]+m[p,k-1]+m[k,j]+y+y+z+\sum_{l=p}^{k-1}a[l]$ ……②

由 y≥z 可得，T<T′，这与使 m[i,j]达到最大值的断开位置为 p 的假设矛盾。

情形 2、y<z　与情形 1 类似。

于是，状态转移方程优化为：

m[i,j]=0　　i=j

$m[i,j]=\max\{m[i,j-1]+m[i+1,j]\}+\sum_{l=i}^{j}d[l]$　　i<j

优化后每个状态转移的状态数减少为 O(1)，算法总的时间复杂度也降为 $O(n^2)$。

（程序及优化前后的运行结果比较见附件）

本题的优化过程是通过对问题最优解性质的分析，找出最优决策必须满足的必要条件，这与搜索中的最优性剪枝的思想十分类似，由此我们再次看到了相同的优化思想应用于不同的算法设计方法。同时，我们也认识到：动态规划的优化必须建立在全面细致分析问题的基础上，只有深入分析问题的属性，挖掘问题的实质，才能实现算法的优化。

3、合理组织状态

在动态规划求解的过程中，需要不断地引用已经计算过的状态。因此，合理地组织已经计算出的状态有利于提高动态规划的时间效率。

例六、求最长单调上升子序列

[问题描述]

给出一个由 n 个数组成的序列 x[1..n]，找出它的最长单调上升子序列。即求最大的 m 和 $a_1,a_2\ldots\ldots,a_m$，使得 $a_1<a_2<\cdots\cdots<a_m$ 且 $x[a_1]<x[a_2]<\cdots\cdots<x[a_m]$。

[算法分析]

这也是一道动态规划的经典应用。动态规划的状态表示描述为：

m[i]，1≤i≤n，表示以 x[i]结尾的最长上升子序列的长度，则问题的解为 max{m[i],1≤i≤n}，状态转移方程和边界条件为：

m[i]=1+max{0, m[k]|x[k]<x[i] , 1≤k<i }

同时当 m[i]>1 时，令 p[i]=k，表示最优决策，以便在计算出最优值后构造最长单调上升子序列。

上述算法的状态总数为 O(n)，每个状态转移的状态数最多为 O(n)，每次状态转移的

时间为 O(1)，所以算法总的时间复杂度为 $O(n^2)$。

[算法优化]

我们先来考虑以下两种情况：

1、若 x[i]<x[j]，m[i]=m[j]，则 m[j]这个状态不必保留。因为，可以由状态 m[j]转移得到的状态 m[k] (k>j，k>i)，必有 x[k]>x[j]>x[i]，则 m[k]也能由 m[i]转移得到；另一方面，可以由状态 m[i]转移得到的状态 m[k] (k>j，k>i)，当 x[j]>x[k]>x[i]时，m[k]就无法由 m[j]转移得到。

由此可见，在所有状态值相同的状态中，只需保留最后一个元素值最小的那个状态即可。

2、若 x[i]<x[j]，m[i]>m[j]，则 m[j]这个状态不必保留。因为，可以由状态 m[j]转移得到的状态 m[k] (k>j，k>i)，必有 x[k]>x[j]>x[i]，则 m[k]也能由 m[i]转移得到，而且 m[i]>m[j]，所以 m[k]≥m[i]+1>m[j]+1，则 m[j]的状态转移是没有意义的。

综合上述两点，我们得出了状态 m[k]需要保留的必要条件：不存在 i 使得：x[i]<x[k]且 m[i]≥m[k]。于是，我们保留的状态中不存在相同的状态值，且随着状态值的增加，最后一个元素的值也是单调递增的。

也就是说，设当前保留的状态集合为 S，则 S 具有以下性质 D：

对于任意 i∈S, j∈S, i≠j 有：m[i]≠m[j]，且若 m[i]<m[j]，则 x[i]<x[j]，否则 x[i]>x[j]。

下面我们来考虑状态转移：假设当前已求出 m[1..i-1]，当前保留的状态集合为 S，下面计算 m[i]。

1、若存在状态 k∈S，使得 x[k]=x[i]，则状态 m[i]必定不需保留，不必计算。因为，不妨设 m[i]=m[j]+1，则 x[j]<x[i]=x[k]，j∈S，j≠k，所以 m[j]<m[k]，则 m[i]=m[j]+1≤m[k]，所以状态 m[i]不需保留。

2、否则，m[i]=1+max{m[j]| x[j]<x[i], j∈S}。我们注意到满足条件的 j 也满足 x[j]=max{x[k]|x[k]<x[i], k∈S}。同时我们把状态 i 加入到 S 中。

3、若 2 成立，则我们往 S 中增加了一个状态，为了保持 S 的性质，我们要对 S 进行维护，若存在状态 k∈S，使得 m[i]=m[k]，则我们有 x[i]<x[k]，且 x[k]=min{x[j]|x[j]>x[i], j∈S}。于是状态 k 应从 S 中删去。

于是，我们得到了改进后的算法：

```
For i:=1 to n do
{
   找出集合 S 中的 x 值不超过 x[i]的最大元素 k;
   if  x[k]<x[i]  then
   {
      m[i]:=m[k]+1;
      将状态 i 插入集合 S;
      找出集合 S 中的 x 值大于 x[i]的最小元素 j;
      if  m[j]=m[i]  then    将状态 j 从 S 中删去;
   }
}
```

从性质 D 和算法描述可以发现，S 实际上是以 x 值为关键字（也是以 m 值为关键字）的有序集合。若使用平衡树实现有序集合 S，则该算法的时间复杂度为 $O(n*\log_2 n)$。本题优化后，每个状态转移的状态数仅为 O(1)，而每次状态转移的时间变为 $O(\log_2 n)$，这也体现了上文所提到的优化中不同因素之间的矛盾，但从总体上看，算法的时间复杂度是降低了。

（程序及优化前后的运行结果比较见附件）

回顾本题的优化过程，首先通过分析状态之间的分析，减少需要保留的状态数，同时发现需要保留状态的单调性，从而减少了每个状态可能转移的状态数，并通过高效数据结构平衡树组织当前保留的状态，实现算法的优化。

通过对本题的优化，我们认识到减少保留的状态数，合理组织已经计算出的状态可以实现减少每个状态可能转移的状态数，同时，选取恰当的数据结构也是算法优化的一个重要原则，在下文的阐述中，还会看到借助数据结构实现算法优化。

4、细化状态转移

所谓“细化状态转移”，就是将原来的一次状态转移细化成若干次状态转移，其目的在于减少总的状态转移的次数。在优化前，问题的决策一般都是复合决策，也就是一些子决策的排列，因此决策的规模较大，每个状态可能转移的状态数也就较多，优化的方法就是将每个复合决策细化成若干个子决策，并在每个子决策后面增设一个状态，这样，后面的子决策只在前面的子决策达到最优解时才进行转移，因此在优化后，虽然，状态总数增加了，但是总的状态转移次数却减少了，算法总的复杂度也就降低了。

应该注意的是：上述优化应该满足一个条件，即原来每个复合决策的各个子决策之间也满足最优化原理和无后效性，也就是说：复合最优决策的子决策也是最优决策；前面的子决策不影响后面的子决策。

上述优化方法再一次体现了实现一个因素的优化要以增大另一个因素作为代价，但是，算法总的时间复杂度的降低才是我们的真正目的。

3.3　减少状态转移的时间

我们知道，状态转移是动态规划的基本操作，因此，减少每次状态转移所需的时间，对提高算法的时间效率具有重要的意义。

状态转移主要有两个部分构成：

1. 进行决策：通过当前状态和选取的决策计算出需要引用的状态。
2. 计算递推式：根据递推式计算当前状态值。其中主要操作是常数项的计算。

本节将分别讨论提高这两部分时间效率的一些方法。

1、减少决策时间

例七、LOSTCITY (NOI`2000)

[问题描述]

现给出一张单词表、特定的语法规则和一篇文章：

文章和单词表中只含 26 个小写英文字母 a…z。

单词表中的单词只有名词，动词和辅词这三种词性，且相同词性的单词互不相同。单词的长度均不超过 20。

语法规则可简述为：名词短语：任意个辅词前缀接上一个名词；动词短语：任意个辅词前缀接上一个动词；句子：以名词短语开头，名词短语与动词短语相间连接而成。

文章的长度不超过 1000。且已知文章是由有限个句子组成的，句子只包含有限个单词。

编程将这篇文章划分成最少的句子，在此前提之下，要求划分出的单词数最少。

[算法分析]

这是也是一道动态规划问题。我们分别用 v,u,a 表示动词，名词和副词，给出的文章用 L[1..M]表示，则状态表示描述为：

F(v,i)：表示 L 前 i 个字符划分为以动词结尾（当 i<>M 时，可带任意个辅词后缀）的最优分解方案下划分的句子数与单词数；

F(u,i)：表示 L 前 i 个字符划分为以名词结尾（当 i<>M 时，可带任意个辅词后缀）的最优分解方案下划分的句子数与单词数。

过去的分解方案仅通过最后一个非辅词的词性影响以后的决策，所以这种状态表示满足无后效性，

状态转移方程为：

F(v,i)=min{ F(n,j)+(0,1)，　L(j+1..i)为动词；
　　F(v,j)+(0,1)，　L(j+1..i)为辅词，i<>M；}
F(n,i)=min{ F(n,j)+(1,1)，　L(j+1..i)为名词；
　　F(v,j)+(0,1)，　L(j+1..i)为名词；
　　F(n,j)+(0,1)，　L(j+1..i)为辅词，i<>M；}

边界条件：F(v,0)=(1,0)；　F(n,0)=(∞，∞)；

问题的解为：min{ F(v,M), F(u,M) }；

上述算法中，状态总数为 O(M)，每个状态转移的状态数最多为 20，在进行状态转移时，需要查找 L[j+1..i]的词性，根据其词性做出相应的决策，并引用相应的状态。下面就通过不同的方法查找 L[j+1..i]的词性，比较它们的时间复杂度。

[算法实现]

设单词表的规模为 N，首先我们对单词表进行预处理，将单词按字典顺序排序并合并具有多重词性的单词。在查找词性时有以下几种方法：

方法 1、采用顺序查找法。最坏情况下需要遍历整个单词表，因此最坏情况下的时间复杂度为 O(20*N*M)，比较次数最多可达 $1000*5k*20=10^8$，当数据量较大时效率较低。

方法 2、采用二分查找法。最坏情况下的时间复杂度为 $O(20*M*\log_2 N)$，最多比较次数降为 $5k*20*\log_2 1000=10^6$，完全可以忍受。

集合查找最为有效的方法要属采用哈希表了。

方法 3、采用哈希表查找单词的词性。首先将字符串每四位折叠相加计算关键值 k，然后用双重哈希法计算哈希函数值 h(k)。采用这种方法，通过 O(N)时间的预处理构造哈希表，每次查找只需 O(1)的时间，因此，算法的时间复杂度为 O(20*M+N)=O(M)。

采用哈希表是进行集合查找的一般方法，对于以字符串为元素的集合还有更为高效的方法，即采用检索树[3]。

方法四、采用检索树查找单词的词性。由于每个状态在进行状态转移时需要查找的所有单词都是分布在同一条从树根到叶子的路径上的，因此，如果选取从树根走一条路径到叶子作为基本操作，则每个状态进行状态转移时的最多 20 次单词查找只需 O(1)的时间，另外，建立检索树需要 O(N)的时间，因此，算法总的时间复杂度虽然仍为 O(M)，但是由于时间复杂度的常数因子小于方法三，因此实际测试的速度也最快。

（程序及四种方法的运行结果的比较见附件）

从本题的优化过程可以看出：采用正确的数据结构是算法优化的重要原则，在动态规划算法的优化中也同样适用。方法 3 使用了哈希表这一高效的集合查找数据结构，方法四使用的针对性更强的检索树，使得算法的时间效率得到了提高。

2、减少计算递推式的时间

计算递推式的主要操作是对常数项的计算，因此减少计算递推式所需的时间主要是指减少计算常数项的时间。

例八、公路巡逻 (CTSC`2000)

[问题描述]

在一条没有分岔的公路上有 n（n≤50）个关口，相邻两个关口之间的距离都是 10km。所有车辆在这条公路上的最低速度为 60km/h，最高速度为 120km/h，且只能在关口出改变速度。

有 m(m≤300)辆巡逻车分别在时刻 T_i 从第 n_i 个关口出发，匀速行驶到达第 n_i+1 个关口，路上耗费时间为 t_i 秒。

两辆车相遇指他们之间发生超车现象或同时到达某个关口。

求一辆于 6 点整从第 1 个关口出发去第 n 个关口的车（称为目标车）最少会与多少辆巡逻车相遇，以及在此情况下到达第 n 个关口的最早时刻。

假设所有车辆到达关口的时刻都是整秒。

[算法分析]

本题也是用动态规划来解。问题的状态表示描述为：

F(i,T)表示在时刻 T 到达第 i 个关口的途中最少已与巡逻车相遇的次数。则状态转移方程和边界条件为：

F(i,T)=min{F(i-1,T-Tk)+w(i-1,T-Tk,T)，300≤Tk≤600}　2≤i≤n

边界条件：F(1,06:00:00)=0;

问题的解为：min{F(n,T)}

其中，函数 w(i-1,T-Tk,T)是计算目标车于时刻 T-Tk 从第 i-1 个关口出发，于时刻 T 到达第 i 个关口，途中与巡逻车相遇的次数。

下面来分析上述算法的时间复杂度，问题的阶段数为 n，第 i 个阶段的状态数为 (i-1)*300，则状态总数为：$O(\sum_{i=1}^{n}(i-1)*300)=O(300*\frac{n*(n-1)}{2})=O(150n^2)$，每个状态转移的状态数为 300，每次状态转移所需的时间关键取决与函数 w 的计算。下面比较采用不同的计算方法时，时间复杂度的差异。

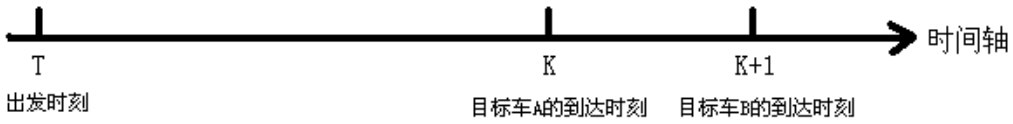
[函数 w 的计算]

方法 1、在每个决策中都进行一次计算，对所有从第 i 个关口出发的巡逻车进行判断，这样平均每次状态转移的时间为 O(1+m/n)，由 M 的最大值为 300，算法总的时间复杂度为：

$$O\left(150n^2*300*(1+\frac{m}{n})\right)=O\left(\frac{m^2n^2}{2}+\frac{m^3n}{2}\right)=O\left(m^3n\right)$$

方法 2、仔细观察状态转移方程可以发现，在对状态 F(i,T)进行转移时，所计算的函数 w 都是从第 i 个关口出发的，而且出发时刻都是 T，只是相应的到达时刻不同，我们考虑能否找出它们之间的联系，从而能够利用已经得出的结果，减少重复运算。

我们来考虑 w(i,T,k)与 w(i,T,k+1)之间的联系：



对于每辆从第 i 个关口出发的巡逻车，设其出发时刻和到达时刻分别为 Stime 和 Ttime，则：

若 Ttime<k 或 Ttime>k+1，则目标车 A、目标车 B 与该巡逻车的相遇情况相同；

若 Ttime=k，则目标车 A 与该巡逻车相遇，对目标车 B 的分析又分为：若 Stime≤T，则目标车 B 不与该巡逻车相遇，否则目标车 B 也与该巡逻车相遇；

若 Ttime=k+1,则目标车 B 与该巡逻车相遇，对目标车 A 的分析又分为：若 Stime≥T，则目标车 A 不与该巡逻车相遇，否则目标车 A 也与该巡逻车相遇；

我们令⊿[k]=w[i,T,k+1]-w[i,T,k]，由上述讨论得：

⊿[k]=G((Ttime=k+1) and (Stime≥T)) –G((Ttime=k) and (Stime≤T)).

其中函数 G(P)表示所有从 i 个关口出发，且满足条件 P 的巡逻车的数目。

这样我们就找到了函数 w 之间的联系。于是，我们在对状态 F(i,T)进行转移时，先对所有从第 i 个关口出发的巡逻车进行一次扫描，在求出 w[i,T,T+300]的同时求出⊿[T+301..T+600]，这一步的时间复杂度为 O(m/n)。在以后的状态转移中，由 w[i,T,k+1]=w[i,T,k]+⊿[k]，仅需 O(1)的时间就可以求出函数值 w，状态转移时间仅为 O(1)。则算法总的时间复杂度为：

$$O\left(150n^2*(\frac{m}{n}+300)\right)=O\left(\frac{m^2n^2}{2}+\frac{m^2n}{2}\right)=O\left(m^2n^2\right)$$

虽然，算法时间复杂度的阶并没有降低，但由于 M 的最大值为 300,N 的最大值为 50，所以实际测试中，优化的效果还是十分明显的。

（程序及两种方法的运行结果比较见附件）

本题对动态规划的优化实际上是应用了动态规划本身的思想，在计算递推式的常数项时，引进了函数⊿，利用了过去的计算结果，避免了重复计算，消除了“冗余”，从而提高算法的时间效率。上文邮局问题中函数 w 的计算也是通过预处理减少了重复计算，近来新出现的双重动态规划也是应用这个思想，利用动态规划计算递推式的常数项。可见，这种优化方法是很有普遍性的。

四、结语

本文主要从减少状态总数，减少每个状态转移的状态数和减少状态转移的时间这三个方面讨论了对动态规划时间效率的优化，同时也间接地讨论了对一般算法进行优化的方法。

在优化的过程，我认识到：对算法的优化一方面要深入分析问题的属性，挖掘问题的本质，另一方面要从原有算法的不足之处入手，不断优化、精益求精。

动态规划的算法设计具有很大的灵活性，需要具体模型具体分析。算法设计如此，算法优化也是如此，本文所述只是一些一般性的方法，许多优化技巧还需要选手们在平时的训练比赛中深入挖掘。

动态规划作为一种高效的算法，仍有许多优化的余地。不断提高算法的性能，使其适应于更大的规模，我想这是广大信息学选手共同的愿望，希望大家共同研究探讨动态规划算法的优化，这也是本文创作的初衷。

参考文献

[1] 吴文虎、 赵鹏，**1993-1996** 美国计算机程序设计竞赛试题与解析，清华大学出版社，**1999**。

[2] 吴文虎、王建德，国际国内青少年信息学（计算机）竞赛试题解析，清华大学出版社，**1997**。

[3] 傅清祥、王晓东，算法与数据结构，电子工业出版社，**1998**。

[4] 全国青少年信息学（计算机）奥林匹克分区联赛组织委员会，信息学奥林匹克（季刊），**1999.3**，**2000.2**。

附录

[1] 这个式子只是直观描述了动态规划的时间复杂度的决定因素，并不能作为普遍的计算公式。

[2] 四边形不等式是 Donald E. Knuth 从最优二叉搜索树的数据结构中提出的，这里被运用于证明动态规划中决策的单调性。

[3] 采用检索树查找字符串只要从树根出发走到叶结点即可，需要的时间正比于字符串的长度。如果哈希函数确实是随机的，那么哈希函数的值与字符串中的每一个字母都有关系。所以，计算哈希函数值的时间与检索树执行一次运算的时间大致相当。但计算出哈希函数值后还要处理冲突。因此，一般情况下，在进行字符串查找时，检索树比哈希表省时间。