神奇的 K 线

【算法分析】

假设给出的序列是 a。对于每个 $a[i]$有 3 种决策，<u>删除</u>它、<u>保留</u>它或者<u>修改</u>它。

很容易想到 DP。可以先确定一个大致的状态为 $f[i][j]$，含义为将 a 中前 i 个元素通过一系列操作对应到新序列中的前 j 个元素的最小代价。但是这有一个问题，无法确定新序列中第 j 个元素的确切值。一种解决方案是给状态添加一维。

> 算法 1：
>
> DP，状态 $f[i][j][k]$表示原序列的前 i 个对应于新序列的前 j 个，并且新序列的第 j 个数是 k。转移方程 $f[i][j][k]=\min\{f[i-1][j-1][k-p[j-1]]$(若 $a[i]=k$)，$f[i-1][j-1][k-p[j-1]]+modify$，$f[i-1][j][k]+delete\}$。
>
> 时间复杂度 $O(n^2 max)$，max 表示数列可能的最大值。期望得分 10~40。

另一种解决方案是修改 $f[i][j]$的含义，规定 a 中第 i 个元素被保留，得到新序列中第 j 个元素的最小代价。

> 算法 2：
>
> DP，状态 $f[i][j]$（$j \leqslant i$）表示新序列中保留 $a[i]$，并且 $a[i]$是新序列中的第 j 个元素的最小代价。状态转移：$f[i][j]=\min\{f[i'][j']+modify*(j-1-j')+delete*((i-j)-(i'-j'))\}$，其中 $j'<j$ 且 $i'-j' \leqslant i-j$ 且 $a[i]-a[i']=\sum_{k=j'}^{j-1} p[k]$。答案就是 $\max\{f[i][j]+\min\{modify,delete\}*(n-i)\}$。时间复杂度 $O(n^4)$，期望得分 30。

注意到根据现在 $f[i][j]$的定义，已经可以确定要从 $a[1]$~$a[i]$中删除$(i-j)$个元素了，那么此时，$f[i][j]$中存的方案就一定是保留的元素个数最多的方案了。于是可以进一步改变 $f[i][j]$的含义，用 $f[i][j]$记录不是最小代价，而是最多保留的个数。于是得到算法 3：

> 算法 3：
>
> DP，状态 $f[i][j]$（$j \leqslant i$）表示新序列中保留 $a[i]$，并且 $a[i]$是新序列中的第 j 个元素的情况下，原序列中最多能保留多少个元素。状态转移：$f[i][j]=\max\{f[i'][j']+1\}$，其中 $j'<j$ 且 $i'-j' \leqslant i-j$ 且 $a[i]-a[i']=\sum_{k=j'}^{j-1} p[k]$。答案就是 $\min\{modify*(j-f[i][j])+delete*(i-j)+\min\{modify,delete\}*(n-i)\}$。时间复杂度 $O(n^4)$，期望得分 30。

这样转移方程就简洁很多了。

<u>**假如已经确定了 $a[i]$被保留，且 $a[i]$在新序列中是第 j 个元素，那么新序列中的第一个元素就是**$a[i]-\sum_{k=1}^{j-1} p[k]$**，设为 *first*[i][j]。**</u>

由于转移只发生在 *first* 相等的状态之间，据此可以得到算法 4。

算法 4：

将 *first* 不同的状态分开计算。这样在寻找前继状态时就不会找到大量 *first*[*i*’][*j*’]≠*first*[*i*][*j*]的状态了。在 *first* 中的每种数个数都不多时效果较好。时间复杂度 O(n^4)，期望得分 30~60。

将 *f* 列成一个表格：

j / i	1	2	3	4	5	6	7
1							
2							
3							
4							
5							
6							
7							

图中灰色格子的状态是无用的。蓝色格子代表状态 *f*[*i*][*j*]，能转移到它的状态在红色区域中，可以发现红色格子组成一个平行四边形。据此，我们可以得到两种算法：

算法 5：

还是**将 *first* 不同的状态分开计算**。按照优先(*i*-*j*)从小到大，(*i*-*j*)相同则 *j* 从小到大的顺序 dp。这样在计算 *f*[*i*][*j*]（蓝色格子）时，所有的前继状态（红色格子）都已经被算出，并且第 *j*’列中已经被计算的状态 *f*[*i*’][*j*’]都满足 *i*’-*j*’≤*i*-*j*。对于同一列中已经被计算的状态，只需要保存最优的那个即可。这样在计算一个状态 *f*[*i*][*j*]时，只需要扫一遍 1~(*j*-1)列中的最优状态，取其中最优的即可；计算好地 *f*[*i*][*j*]后，再用它更新第 *j* 列的最优状态。这样转移的复杂度降为 O(*n*)，总时间复杂度降为 O(n^3)，期望得分 60。

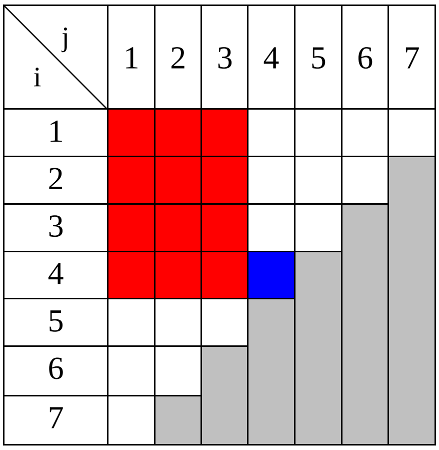
由于每次转移要找的列是 1~(*j*-1)列，是连续的，所以可以用线段树来维护连续列的最优状态。转移复杂度就降为 O(log*n*)，总时间复杂度降为 O(n^2log*n*)，期望得分 100。标程用的是这种算法。

算法 6：

类似于算法 5，不过按照优先 *j* 从小到大，*j* 相同则 *i* **从大到小**的顺序 dp。定义斜线 *k* 为所有 *i*-*j*=*k* 的状态 f[*i*][*j*]组成的状态集合。线段树维护的是每一条斜线上的最优状态。计算状态 *f*[*i*][*j*]时，只需要用斜线 1~(*i*-*j*)中的最优状态来转移即可。时间复杂度 O(n^2log*n*)，期望得分 100。

为什么还要将 *first* 不同的分开呢？因为如果一起做，那么对于每一个 *first* 都要开一个数组来记录最优状态，空间不够。

其实还可以将状态 *f*[*i*][*j*]的含义改为前(*i*+*j*)个，删除 *i* 个，留下（保留或修改）*j* 个，并且第(*i*+*j*)个保留。这样转移的区域就从一个非矩形的平行四边形变成了一个矩形（如下图）。虽然和前面的状态是等价的，但是更直观了。



【小结】

这道题的关键部分有两个。第一就是发现 *first* 不同的状态之间不能互相转移，于是将它们分开 DP，没有这一点是很难进行之后的优化的。第二就是将状态用表格形式列出来，发现转移的区域是一个平行四边形，这就将很难组织起来状态用一个数据结构就很容易维护了，这个方法也很容易推广。