从《Cash》谈一类分治算法的应用

分治算法的基本思想是将一个规模为 N 的问题分解为 K 个规模较小的子问题，这些子问题相互独立且与原问题性质相同．求出子问题的解，就可得到原问题的解．分治算法非常基础，但是分治的思想却非常重要，本文将从今年 NOI 的一道动态规划问题 **Cash** 开始谈如何利用分治思想来解决一类与维护决策有关的问题：

例一．货币兑换(Cash)[1]

问题描述

小Y最近在一家金券交易所工作．该金券交易所只发行交易两种金券：A 纪念券（以下简称 A 券）和 B 纪念券（以下简称 B 券）每个持有金券的顾客都有 一个自己的帐户．金券的数目可以是一个实数．

每天随着市场的起伏波动，两种金券都有自己当时的价值，即每一单位金券 当天可以兑换的人民币数目．我们记录第 K 天中 A 券和 B 券的价值分别为 A_K 和 B_K（元/单位金券）

为了方便顾客，金券交易所提供了一种非常方便的交易方式：比例交易法．比例交易法分为两个方面：

A) 卖出金券：顾客提供一个**[0，100]**内的实数 **OP** 作为卖出比例，其意 义为：将 **OP%**的 A 券和 **OP%**的 B 券以当时的价值兑换为人民币；

B) 买入金券：顾客支付 **IP** 元人民币，交易所将会兑换给用户总价值为 **IP** 的金券并且满足提供给顾客的 A 券和 B 券的比例在第 K 天恰好为 $Rate_K$；

例如，假定接下来 **3** 天内的 A_k、B_k、$Rate_K$ 的变化分别为：

时间	A_k	B_k	$Rate_k$
第一天	**1**	**1**	**1**
第二天	**1**	**2**	**2**
第三天	**2**	**2**	**3**

假定在第一天时，用户手中有 **100** 元人民币但是没有任何金券． 用户可以执行以下的操作：

[1] NOI 2007，货币兑换

时间	用户操作	人民币(元)	*A* 券的数量	*B* 券的数量
开户	无	100	0	0
第一天	买入 100 元	0	50	50
第二天	卖出 50%	75	25	25
第二天	买入 60 元	15	55	40
第三天	卖出 100%	205	0	0

注意到，同一天内可以进行多次操作.

小 Y 是一个很有经济头脑的员工，通过较长时间的运作和行情测算，他已经 知道了未来 N 天内的 A 券和 B 券的价值以及 $Rate$. 他还希望能够计算出来，如 果开始时拥有 S 元钱，那么 N 天后最多能够获得多少元钱.

算法分析

不难确立动态规划的方程：

设 $f[i]$表示第 i 天将所有的钱全部兑换成 A, B 券，最多可以得到多少 A 券. 很容易可以得到一个 $O(n^2)$的算法：

```
f[1]←S * Rate[1] / (A[1] * Rate[1] + B[1])
Ans←S
For i ← 2 to n
    For j ← 1 to i-1
        x ← f[j] * A[i] + f[j] / Rate[j] * B[i]
        If x > Ans
            Then Ans ← x
    End For
    f[i] ← Ans * Rate[i] / (A[i] * Rate[i] + B[i])
End For
Print(Ans)
```

$O(n^2)$的算法显然无法胜任题目的数据规模. 我们来分析对于 i 的两个决策 j 和 k，决策 j 比决策 k 优当且仅当：

$(f[j]-f[k]) * A[i] + (f[j] / Rate[j] - f[k] / Rate[k]) * B[i] > 0$.

不妨设 $f[j] < f[k]$，$g[j] = f[j] / Rate[j]$，那么

$(g[j]-g[k]) / (f[j]-f[k]) < -a[i] / b[i]$.

这样我们就可以用平衡树以 $f[j]$为关键字来维护一个凸线，平衡树维护一个点

集$(f[i], g[i])$，$f[i]$是单调递增的，相邻两个点的斜率是单调递减的．每次在平衡树中二分查找与$-a[i] / b[i]$最接近的两点之间的斜率．

这样动态规划的时间复杂度就降低为 $O(n\log_2 n)$，但是维护凸线的平衡树实在不容易在考场中写对☹，编程复杂度高，不易调试(我的 **Splay** 代码有 **6k** 多)．这个问题看上去只能用高级数据结构来维护决策的单调性，事实上我们可以利用分治的思想来提出一个编程复杂度比较低的方法：

对于每一个 i，它的决策 j 的范围为 **1**~i-**1**．我们定义一个 **Solve** 过程：*Solve*(l, r)表示对于的 $l \le i \le r$，用 $l \le j \le i$-**1** 的决策 j 来更新 $f[i]$的值．这样我们的目标就是 *Solve*(**1**, n)：可以先 *Solve*(**1**, n/**2**)后计算出 $f[1]$ .. $f[n/2]$，那么 **1**~n/**2** 的每一个数一定是 n/**2**+**1**~n 的每个 i 的决策，用 **1**~n/**2** 的决策来更新 n/**2**+**1**~n 的 $f[i]$值后 *Solve*(n/**2**+**1**, n)．这恰好体现的是一种分治的思想：

用 **1**~n/**2** 的决策来更新 n/**2**+**1**~n 的 $f[i]$值：类似用平衡树的方法，我们可以对 **1**~n/**2** 的所有决策建立一个凸线，对 n/**2**+**1**~n 的所有 i 按照$-a[i] / b[i]$从大到小排序，凸线的斜率是单调的，$-a[i]/b[i]$也是单调的，这样我们就可以通过一遍扫描来计算出对于每一个 i 在 **1**~n/**2** 里面最优的决策 j.

现在面临的问题是如何对于一段区间$[l, r]$维护出它的凸线：由于 $f[]$值是临时计算出来的，我们只需要递归的时候利用归并排序将每一段按照 $f[]$值从小到 **12121212122111** 大排序，凸线可以临时用一个栈 $O(n)$计算得出．下面给一个分治算法的流程：

由于$-a[i] / b[i]$是已知的，不像 $f[i]$是临时计算得出的．因此可以利用归并排序预处理，计算每一段$[l, r]$按照$-a[i] / b[i]$排序后的 i.

Procedure *Solve*(l, r)

If $l = r$

Then 更新 *ans*，利用已经计算好的 l 的最优决策 k，计算 $f[l]$值，**Exit**

$Mid \leftarrow (l + r) / 2$

Solve(l, *mid* -**1**)

对$[l, mid$-**1**$]$这一段扫描一遍计算出决策的凸线，由于$[mid$+**1** .. $r]$这一段以$-a[i] / b[i]$的排序在预处理已经完成，因此只需要扫描一遍更新$[mid + 1$ .. $r]$的最优决策．

Solve(mid+1, r)

利用$[l, mid$-**1**$]$已排好序的 $f[]$值和$[mid$+**1**, $r]$已排好序的 $f[]$值归并排序将

$[l, r]$这一段按 $f[]$值排序.

End Procedure

至此，问题已经基本解决．时间复杂度为 $T(n) = 2T(n/2) + O(n)$，因此算法的时间复杂度为 $O(n\log_2 n)$，NOI2007 的测试数据最慢的测试点 0.2s，是一个相当优秀的算法.

我们比较一下分治算法和用平衡树维护决策的方法：时间复杂度均为 $O(n\log_2 n)$，空间复杂度平衡树为 $O(n)$，分治为 $O(n\log_2 n)$ (预处理-$a[i]/b[i]$需要 $O(n\log_2 n)$的空间)．但是编程复杂度却差别非常大，分治算法实现起来相当简单，对于考场来说无疑是一个非常好的方法.

在编程复杂度非常高的情况下，动态规划维护决策与分治思想很好的结合起来从而降低了编程复杂度，无疑是“柳暗花明又一村”．这种分治思想主要体现在将**不断变化**的决策转化成一个**不变**的决策集合，将**在线**转化为**离线**．下面我们再来看一个例题：

例二．Mokia[2]

问题描述

有一个 $W * W$ 的棋盘，每个格子内有一个数，初始的时候全部为 0．现在要求维护两种操作：

1) *Add*：将格子(x, y)内的数加上 A.

2) *Query*：询问矩阵(x_0, y_0, x_1, y_1)内所有格子的数的和.

数据规模：操作 1) ≤ 160000，操作 2) ≤ 10000，$1 \le W \le 2000000$.

算法分析

这个问题是 **IOI 2000 Mobile** 的加强版：**Mobile** 中 $W \le 1000$，就可以利用二树状数组在 $O(\log_2{}^2 n)$的时间复杂度内维护出操作 1)和操作 2). 这个问题中 W 很大，开二维树状数组 $O(W^2)$的空间显然吃不消，考虑使用动态空间的线段树，最多可能达到操作次数 $*(\log_2 W)^2$ 个节点，也相当大了. 考虑使用分治思想来解决问题：

将操作 1)和操作 2)按顺序看成是一个个事件，假设共有 *Tot* 个事件，

[2] Balkan Olympiad in Informatics, 2007

Tot≤170000．类似例题一，我们定义 *Solve*(*l*, *r*)表示对于每一个 *Query* 操作的事件 *i*， 将 *l* ..*i*-**1** 的 *Add* 操作的所有属于 *i* 的矩形范围内的数值累加进来．目标是 *Solve*(**1**, *n*)．

假设计算 *Solve*(*L*, *R*)，递归 **Solve**(*L*, *Mid*)，*Solve*(*Mid* + **1**, *r*)后，对 *L* .. *Mid* 的所有 *Add* 操作的数值累加到 *Mid* + **1** .. *R* 的所有匹配的 *Query* 操作的矩形中．

后面这个问题等价于：平面中有 *p* 个点，*q* 个矩形，每个点有一个权值，求每个矩形内的点的权值之和．这个问题只需要对所有的点以及矩形的左右边界进行排序，用一维树状数组或线段树在 $O((p+q)\log_2 W)$的时间复杂度即可维护得出．因此问题的总的时间复杂度为 $O(Tot*\log_2 Tot*\log_2 W)$，不会高于二维线段树的 $O(Tot*\log_2 W*\log_2 W)$的时间复杂度．

上述这个算法无论是编程复杂度还是空间复杂度都比使用二维线段树优秀，分治思想又一次得到了很好的应用．在这个问题中，利用分治思想我们将一个在线维护的问题转化成一个离线问题，将二维线段树解决的问题**降维**用一维线段树来解决，使得问题变得更加简单．

总结

【例题一】是一个数据结构维护动态规划决策的问题，【例题二】为一个数据结构维护数据信息的问题．我们巧妙地使用分治思想，将在线维护的问题转化为离线问题，将变化的数据转化为不变的数据，使得问题解决更加的简单．