分治、DP 進階

hansonyu123

2016 年 10 月 5 日

1 分治法 (Divide and Conquer, D&Q)

分治法，字面上的意思是「分而治之」，而實作上也是如此。總地來說，分治法就是將原問題分成若干個規模較小的子問題（「分」），再將子問題的答案統整起來解決原問題（「治」）。

事實上，之前我們也看到了許多有分治法的影子的演算法了。比如說，二分搜的精髓在於它將原問題分成更小的兩個子問題，並且只計算其中一個（因為這樣就足夠了）；然而做為分治法，它的「治」的色彩又過少了。又比如說，DP 本身也是一種分治法，不過 DP 的特點在於將答案儲存以避免重複計算，這又是大多分治法所沒有的特點。所以這堂課，就讓我們來認識純真 (?) 的分治法吧！

1.1 Merge sort

雖然這並不是非常經典的分治法習題，但拿來講解分治法的精髓特別好用。

Merge sort 的想法是，先將一個長度為 n 的陣列切半，分別排序（分）。所以現在我們需要將子問題的答案合併，也就是說，我們要試圖把兩個已排序的陣列合在一起變成一個更大的已排序陣列。這個其實也不難，在 $O(n)$ 時間內就可以做到（你也可以偷懶用 STL 的 merge）。於是你就成功合併子問題的答案，解決原問題（治）了。

這是一個分治法的例子。然而最大的問題應該就是：這個複雜度如何估算呢？

假設排序 n 個東西的複雜度是 $T(n)$，那麼 $T(n)$ 其實就是排序兩個長度為 $n/2$ 的陣列的時間 $2T(n/2)$ 再加上統整答案的時間 $O(n)$。寫成數學式就是

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

光看這個遞迴式沒辦法馬上得出複雜度，所以我們需要一個定理……

1.2 主定理 (Master Theorem)

這個「主」絕對不帶有任何宗教色彩 (?)，它只是「主人」的「主」的意思。

雖然主定理的敘述不長這樣，不過我們只需要知道這麼多就好：
如果 a 是正整數，$b > 1, k \geq 0$ 而且

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^{\alpha}\log^{k} n)$$

那麼：

$$T(n) = \begin{cases} O(n^{\log_b a}) & \log_b a > \alpha \\ O(n^{\alpha}\log^{k} n) & \log_b a < \alpha \\ O(n^{\alpha}\log^{k+1} n) & \log_b a = \alpha \end{cases}$$

簡單來說，誰比較大就聽誰的話（所以才叫「主」定理）；如果一樣大，就多一個 log。以 Merge sort 為例。$T(n) = 2T(n/2) + O(n)$，這裡 $\log_2 2$ 跟 1 一樣大，所以 $T(n) = O(n \log n)$。

至於主定理的證明，其實不知道也罷，知道了也不會更好（咦）。證明只是單純地寫開加起來而已（通常這個過程都會利用一個東西叫「遞迴樹」）。不過這是大學演算法教授熱愛講的課題之一，到那個時候再無奈地 (?) 聽一下也不遲。

不過要知道主定理並不是萬能的。非常低機率會碰到子問題規模不一樣的情況（如 Median of medians），或者 $T(n) = 4T\left(\frac{n}{2}\right) + O(\frac{n^2}{\log n})$ 這種不在主定理定義範圍內的情況（據說本例解是 $T(n) = n^2 \log\log n$），此時主定理就不能用了。然而這種問題不常見，主定理通常情況還是很有用的。

1.3 Karatsuba algorithm：超越小學直式乘法

給你兩個 n 次多項式，計算出它們的乘積。
你小學的時候應該就會 $O(n^2)$ 地算出來了（因為多項式乘法就是沒有進位的直式乘法）。然而，要如何超越小學直式乘法呢？以前這問題一直困擾著各專家，直到有人提出了分治的解法，才終於突破腦殘的 $O(n^2)$。

先看一個失敗的例子吧！方便起見，假設 $n = 2k$ 是個偶數。一個很直覺的分治法就是將兩個 $2k$ 次多項式 f, g 分成 $f_1x^k + f_2, g_1x^k + g_2$ 這四個 k 次多項式。乘法的結果應該是

$$f_1g_1x^{2k} + (f_1g_2 + f_2g_1)x^k + f_2g_2$$

所以只要算出 $f_1g_2, f_1g_2, f_2g_1, f_2g_2$ 的值之後（分），就可以用 $O(n)$ 多項式加法的算出所求的答案。所以複雜度 $T(n)$ 滿足

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n)$$

因為 $\log_2 4 = 2 > 1$，主定理告訴我們 $T(n) = O(n^2)$—竟然跟小學直式乘法一樣 QQ

先別被憤怒沖昏頭 (?)，回頭仔細檢視你的算法。複雜度來源主要來自於那個萬惡的「4」，所以你希望可以不要計算 4 次子問題。事實上，你需要的只有三個東西，感覺上好像三次乘法就可以解決了。如果你成功地只用三次乘法（還有一堆加法）湊出答案的話，那麼

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

主定理告訴我們 $T(n) = n^{\log_2 3}$，比 $O(n^2)$ 還要快許多。所以接下來就是想盡辦法用三次乘法湊出答案。這其實也滿容易的。如果你算出 $(f_1 + f_2)(g_1 + g_2), f_1g_1, f_2g_2$ 三個乘法的結果的話，你可以把第一個減掉第二個跟第三個，就會留下 $(f_1g_2 + f_2g_1)$ 了。這樣你想要的都有了，自然就可以算出原問題的答案了。

實作上，為了方便切割問題，常常會令 n 是 2 的冪次。也就是說，在多項式前面補 0 使得它變成次數是二的冪次的多項式。這樣做除了大大減低實作難度以外，複雜度也不會受到影響，是個常用的手法。

扯一點點題外話。事實上多項式乘法最快可以到 $O(n \log n \log\log n)$（或甚至 $O(2^{2\log^* n} n \log n)$，其中 $\log^* n$ 是「迭代對數」），不過實作非常複雜。有興趣的人可以查「快速傅立葉變換」、「Schönhage—Strassen algorithm」和「Fürer's algorithm」。

1.4 題外話：常數優化

分治法有一個劣勢：它的常數太大了。如果你仔細地證明主定理，你會發現常數會被放大好幾倍，尤其當 $\log_b a$ 跟 α 非常相近的時候常數被放大的速率就會跟 log 愈來愈近。如果要改進常數的話，常用的方法是子問題若已經足夠小了，就直接暴力算出來。

比如說 Merge sort 中，當陣列長度小於 50（之類的）時，開始用 Insertion sort。又比如說多項式乘法在次數小於 10 時開始用小學直式乘法之類的。這個方法可以有效改進常數偏大的問題。至於「足夠小」到底是多小，submit 之後看看時間就知道了。

1.5 經典例題

說了那麼多，希望大家熟記，「分治法」就是先想如何「分」，再想如何「治」。接下來的題目只要持著這種想法基本上都能開心 AC。

1. (No judge) 實作 Karatsuba algorithm。

2. (No judge) 在 $O(n^{\log_2 7})$ 的時間內計算 $n \times n$ 的矩陣乘法。
（提示：問題的規模砍半，把八次乘法變七次。）

3. (UVa 10245) 最近點對問題：給你平面上 n 個點。求最近的那一對點的矩離。時間複雜度 $O(n \log n)$。
（提示：如何在 $O(n)$ 的時間內合併子問題？）

1.6 習題

1. (UVa 11129) 構造出一個 0 到 $n-1$ 的排列，使之無等差子序列。時間複雜度 $O(n)$ 或 $O(n \log n)$。

2. (CF 559B) 兩個長度一樣的字串 a 和 b「等價」若且唯若 a 和 b 一模一樣，或者它們的長度都是偶數，a 砍半後變成 a_1 和 a_2，b 砍半後變成 b_1 和 b_2，且 a_1, b_1、a_2, b_2 等價或 a_1, b_2、a_2, b_1。請在 $O(n^{\log_2 3})$ 的時間內判斷兩字串是否等價。
（提示：如何把四次比較減少為三次？）
（註：這題有好好的 $O(n \log n)$ 解法，而且比上述算法或甚麼 hash 之類的都好想很多，如果你一開始就想到這個解法請不要覺得意外。）

3. (No judge) 給你平面上 n 個點。找出滿足 A 在 B 的右上方的點對 (A, B) 的個數。時間複雜度 $O(n \log n)$。

4. (TIOJ 1208) 給你一個長度為 $N \leq 20000$ 的陣列，每個數都不超過 10000。請找出 $\frac{n(n+1)}{2}$ 個區間中，總和第 k 大的區間，其區間和是多少。

5. (No judge) 有一個 $n \times n$ 的方格，裡面每格都填有一個數字。這 $n \times n$ 的方格外面被一圈很大的數字包圍起來。請在 $O(n)$ 的時間內找到一個格子，它的值不比它上、下、左、右的數字來的大。

6. (IOI 2006)(TIOJ 1631) 有圖，詳見 TIOJ。
（提示：先把正方形沿主對角線切成兩個三角形。如果你構完了不知道複雜度，想想看快速排序的平均複雜度是多少。）

2 樹分治

這東西有點難塞進其它主題裡，只好放在這裡了。

樹有很好的分治結構：對於一個有根樹，把根拔掉之後，就變成許多的子樹（子問題）了。所以有關樹的許多問題都可以用分治的想法解決。然而跟前面的分治稍微不同的是，因為子樹規模在有些時候是不能控制的，或者就算能控制，複雜度也不會改進，所以計算複雜度時通常就不用主定理而是直接估計了。

2.1 經典例題：直徑

在很久以前，我們學過找樹的直徑的方法：兩次 DFS。利用樹分治，我們可以只用一次的 DFS 就完成任務，因此複雜度也是相同的 $O(V)$。我們的目標是找到兩點距離的最大值。

任取一個點當作根，那麼直徑的種類可以分成兩種：看直徑有沒有經過根。

對於沒有經過根的情況，那麼直徑一定是在根的某一棵子樹。所以只要把所有子樹（子問題）處理好之後，第一種情況就可以輕鬆解決了：全部取最大值就好。

而對於第二種情況，很容易可以發現這個時候直徑的兩個端點會是在不同子樹而且離根最遠（也就是深度最深）的兩個點。為了快速知道最深的深度，我們需要知道每棵子樹的深度，而這個也可以在同一次 DFS 處理完畢。如此一來，取子樹深度最大的兩個子樹，把答案相加就是答案了。不難估計複雜度就跟一般 DFS 一樣，是 $O(V+E)=O(V)$。

從這個例子可以看出，樹分治的精髓就是把跟根無關的完全交給子樹們處理，而跟根有關的再想辦法處理。如果跟根有關的答案在計算時需要更多資訊，再把計算這個資訊當作新的問題，同時使用樹分治就可以了。

有時候（尤其是多次詢問或者實行 DP 時）需要將子樹的答案存好，所以有時候樹分治也叫作樹 DP，不過兩者的精髓基本上是一樣的。

2.2 題外話：重心剖分

重心，就是使最大子樹最小的點。可以證明這樣的點最多兩個，而且如果有兩個的話，兩個重心一定是鄰居。而重心有一個好性質，就是其最大子樹的大小會是原本樹的大小的一半以下。如此就可以控制子問題的規模，說不定複雜度就會比亂樹分治還要來得好。

可惜的是，一般來說，合併問題都是在 $O(1)$ 的時間內解決的。如此以來，就算使用了好好的樹分治，複雜度 $T(n)$ 滿足

$$T(n)=2T\left(\frac{n}{2}\right)+O(1)$$

主定理告訴我們 $T(n)=O(n)$，也就是說複雜度跟胡亂地樹分治是一樣的。這是一件很悲劇的事情，因為光做重心剖分的預處理就會花 $O(n\log n)$ 的時間了（後面會提到）。然而在某些情況中，合併問題需要花 $O(n)$ 的時間才能解決。如果胡亂地樹分治的話，複雜度會到 $O(n^2)$。然而如果可以使用重心剖分的話，那麼複雜度 T 會滿足

$$T(n)=2T\left(\frac{n}{2}\right)+O(n)$$

主定理告訴我們此時複雜度就會是好好的 $O(n\log n)$。雖然這種技巧有點冷僻，不過要是遇上了錯過還是滿可惜的。

不過在使用的時候要注意題目的結構允不允許使用重心剖分（通常這種題目都是詢問有關於樹的「全局性質」，比如說動態詢問樹中任兩點距離之總和）。

此外，在執行重心剖分時需找到各個子樹的重心。如果重心剖分只需要做一次（靜態），你可以在重心剖分時同步找重心。如果你需要動態更新查詢，你可能需要預處理一個「重心樹」的結構：對於每個子樹的重心，記錄原本樹的重心是誰，把它當作祖先。至於重心的找法也不難，你可以使用 BFS 或者是 DFS。如果某個點的子樹很肥大，代表重心就在那個肥大的子樹中，往那裡找就對了。

最後，如果需要儲存有關各個子樹的資訊，建議以「點編號」和「子樹深度」開二維陣列儲存。由於子樹深度（也就是說是在第幾次重心剖分的子樹）不會超過 $O(\log n)$，空間複雜度會是好好的 $O(n \log n)$。

2.3 習題

1. 用樹分治的想法實作直徑找法。你可以用 TIOJ 1838 (IOI 2013) 測試你的程式是否正確。

2. (UVa 10077) 有圖，詳見 UVa。

3. (POJ 2342) 有一棵樹，每個點都有點權。請選出一些點使得點權和最大，且沒有一個點和它的父節點同時被選到。

4. (UVa 10859) 給你一個樹。請選最少的點使得對於每個邊，至少有一個端點都被選到。在選最少個點的情況下，請讓兩端點都被選到的邊最多。

5. (POJ 1741) 有一個 n 個點的樹，每條邊都有邊權。求距離小於等於 k 的點對個數。複雜度 $O(n \log^2 n)$。

3 DP 優化

在以前的 DP 中，我們都是把所有可能的候選子問題的答案都計算過一遍。然而仔細分析的話，有時候可以把一些不可能成為最佳解（成為轉移來源）的子問題給略過而不計算。這樣的方法往往可以把複雜度降低。而這種技巧就稱為「DP 優化」。

而 DP 優化中，最常見的莫過於「斜率優化」以及「四邊形優化」。當然，種類不僅限於這兩種，而這兩種優化的應用也不限於接下來提的 DP 轉移式。唯有參透精髓後應用自如才是王道。

3.1 斜率優化 (Convex Hull Optimization)

先討論比較一般的情況。如果轉移式是

$$dp[i] = f(i) + \max_{j \le R_i} \{a[j]x[i] + b[j]\}$$

其中 R_i 是遞增的，直接計算的複雜度是 $O(n^2)$。然而，可以發現有些候選人是有「有效期限」的。比如說，如果 $a[j] > a[k]$，而且有一個 i 使得 $a[j]x[i] + b[j] > a[k]x[i] + b[k]$，那麼對於所有的 $i' > i$，因為 $x[i'] \geq x[i]$，所以 $a[j]x[i'] + b[j] > a[k]x[i'] + b[k]$。所以在 i 以後，k 都不可能是轉移的來源，就可以不用考慮了。

刪去不再可能是轉移來源的候選人後，你可以把所有 $y = a[j]x + b[j]$ 在座標平面上畫出來，它們應該會是一個斜率不斷遞增的線集。如果對於每個 x，都在線集中找到最大的那條線，把它標起來，那麼看起來就會像是一個下凸包，因此這個技巧又稱為「凸包優化」(convex hull optimization)。

其實如果用凸包來想像的話會好做許多。每次要找答案時，相當於求 $x = x[i]$ 這條直線和當前凸包的交點。如果 $x[i]$ 有單調性，那就只需看以前的轉移來源還是不是最大的：跟下一個候選人比較，如果贏了就直接轉移，輸了就踢除，持續進行這個動作直到找到最佳解。如果 $x[i]$ 沒有單調性，那就必須紀錄每條線在凸包中的有效範圍，並二分搜答案在哪條線上（因為斜率遞增）。

除了維護候選人結構以外，還需要在 R_i 增加的過程不斷地加入新的候選人。因為加進去之後還得保持斜率不斷遞增，我們需要檢查是不是有人不可能再成為最佳（維護好凸包性質），所以這個資料結構必須支援二分搜和插入、移除任意一項，因此 set 是個不錯的選擇。由於每個候選人最多只有被加進和拔出一次，用 set 複雜度 $O(n \log n)$。

還有幾件需要注意的事情：

1. $a[j], b[j]$ 可能是和 $dp[j]$ 有關的。不過如果有關，那麼 R_i 一定會小於 i，所以計算完 $dp[i]$ 之後再計算 $a[i], b[i]$，加入候選人即可。

2. 別讓腦袋硬化了！max 改成 min 還是能如法炮製（變成上凸包）。

不過只會不動腦地套用是不行的。以 APIO 2010 Commando 為例：給你一個長度為 N 的正整數列以及一個二次方程 $f(x) = ax^2 + bx + c$ 且 $a < 0$。你要將 N 分成連續的很多塊，使得每塊的總和套入 f 之後的函數值之總和最大。容易看出轉移式是：

$$dp[i] = \max_{j<i} \left\{ dp[j] + f\left(\sum_{k=j+1}^{i} x[k] \right) \right\}$$

因為要一直求一段的和，我們不妨預處理前綴和 $S[j] = x[1]$ 加到 $x[j]$，接著再展開、簡化，斜率優化就慢慢成形了：

$$\begin{aligned} dp[i] &= \max_{j<i} \{dp[j] + f(S[i] - S[j])\} \\ &= \max_{j<i} \left\{dp[j] + a(S[i]^2 - 2S[i]S[j] + S[j]^2) + b(S[i] - S[j]) + c\right\} \\ &= f(S[i]) + \max_{j<i} \{(-2aS[j])S[i] + (dp[j] + f(S[j]))\} \end{aligned}$$

於是它已經變成斜率優化的型態了，可以用前述方法解決，複雜度 $O(N \log N)$。

然而這還不夠，事實上這題可以優化成 $O(N)$。關鍵就是斜率和查詢都具有單調性。

3.2 單調隊列優化

單調隊列優化可以視為斜率優化的特殊情況。當斜率和查詢（$x[i]$）都有單調性的時候，查詢和新加入一條線不再需要搜尋有效範圍或斜率，而是直接在頭或尾插入、查詢就好，因此只需要一個 deque 便能完成任務。當然，在加入時需要把不再可能是最佳解的候選人移除。因此複雜度 $O(N)$。

一個斜率單調的例子是斜率全部為零（這個時候當然單調，畢竟根本一模一樣）。回想一下有限背包問題，當時我們用 $O(NWM)$ 或 $O(NW \log M)$ 的時間做完，但用單調隊列優化可以改進為 $O(NW)$。假設第 i 件物品的重量、價值、個數分別是 $w[i], v[i], c[i]$。先寫出轉移式：

$$dp[n][m] = \max_{0 \le k \le c[n], m - w[n]k \ge 0} \{dp[n-1][m - w[n]k] + v[n]k\}$$

令 $m = w[n]i + r$，其中 $0 \le r < w[n]$：

$$\begin{aligned} dp[n][w[n]i + r] &= \max_{\max(0, i-c) \le j \le i} \{dp[n-1][w[n]j + r] + v[n](i - j)\} \\ &= v[n]i + \max_{\max(0, i-c) \le j \le i} \{dp[n-1][w[n]j + r] - v[n]j\} \end{aligned}$$

所以如果先計算好 $dp[n-1][m]$ 的話，可以發現這個轉移式可以套用單調隊列優化。因為 $\max(0, i-c), i$ 對 i 都遞增，且 $dp[n-1][w[n]j + r] - v[n]j$ 和 i 無關，也就是說斜率為零。如此，複雜度縮為 $O(NW)$。

在斜率有單調性的時候，我們還可以處理 j 的左界也有限定（而且遞增）的狀況。由於斜率和左界都單調，所以「過期」的東西一定會出現在端點，那麼就在對任何候選人做事情之前先檢查是否還在區間內，不是的話就移除就好。

3.3 四邊形優化 (Knuth Optimization)

看了學長的講義，一堆網上資源和一堆論文 orz（筆者對四邊形優化也沒很熟）。總之，如果有興趣的話，這會是一個滿大的坑。

先聲明，四邊形優化的證明還滿暴力（複雜）的，但是應用價值頗高，個人建議理解後直接記下來就好了（雖然我沒有記:q）。

對於一個雙變數函數 $w(i, j)$，我們定義以下四種性質：

1. 凹（凸）四邊形不等式（concave(convex) Monge condition）：
$\forall a < b, c < d$，$w(a,c) + w(b,d) \geq (\leq)\ w(a,d) + w(b,c)$。

2. 凹（凸）完全單調性（concave(convex) totally monotone）：
$\forall a < b, c < d$，$w(a,c) \leq (\geq)\ w(b,c) \Rightarrow w(a,d) \leq (\geq)\ w(b,d)$。

由四邊形不等式可以推出完全單調性，但是反過來則不行。因為有些人會直接把這些性質簡稱為「凹（凸）單調」，所以在看的時候要注意他們指的是哪一種。

以上的性質，如果函數 $w(i,j)$ 定義在 $i,j \in \mathbb{N}_0$，則只要驗證 $b = a+1, d = c+1$ 的情況就好了。(很重要！)

3.4 1D/1D 的凹性優化

所謂 1D/1D 的 DP 是指：

$$dp[j] = \min_{0 \leq i < j} f(i,j)\ ;\ i,j \in \mathbb{N}_0, dp[0] = k$$

注意在 $f(i,j)$ 裡面會有 $dp[i]$ 的項，所以不要忘記在 $dp[i]$ 還沒算出來之前，$f(i,j)$ 的值是未知的；另外，在 $i \geq j$ 時，$f(i,j)$ 是沒有定義的。

如果 f 符合凹完全單調性，且令 $h(j)$ 是使 $f(i,j)$ 最小的 i 值，那麼不難證明兩件事：

1. $\forall\, i' < h(j), j' \leq j$ 且 $i < j'$，$f(i',j') \geq f(i,j')$

2. $\forall\, i' > h(j), j' \geq j$ 且 $i' < j$，$f(i',j') \geq f(i,j')$

以上兩式可以直接由凹完全單調性和其逆否命題直接推導。仔細觀察可以發現 $h(j) \leq h(j-1)$，而且根據 dp 和 h 的定義，$dp[j] = f(h(j), j)$。也就是說，我們找到了 dp 陣列轉移來源的規律（不斷遞減），剩下的問題就是要如何維護這個轉移來源。

要如何善用轉移來源遞減的性質呢？可以維護一個資料結構，每一個元素是 (p, L, R)，代表在只考慮當前 f 的所有「已知項」時，p 是使 $f(i, [L,R)\ )$ 最小的 i 值。可以使元素按照 L, R 值遞增排列（因為每個 $[L,R)$ 不相交），根據轉移來源遞減，p 便會遞減。

我們發現，在剛計算完 $dp[0 \cdots j]$ 時，我們的已知項是 $f(\ [0,j), [j,n)\ )$（因為才剛算出 $dp[j]$，故不包含 j）。計算出來後，要算出 $dp[j+1]$ 前，必須要更新資料結構當中的 p 值，必須多考慮 $f(j, [j+1, n)\ )$（n 是狀態上界）。於是可以分兩個 case 討論，假設當前資料結構中 L 最小的元素為 S：

1. $f(j, j+1) \geq f(S.p, j+1)$：
代表多考慮的元素都不可能成為最佳解（否則違反轉移來源遞減），直接略過。

2. $f(j, j+1) < f(S.p, j+1)$：

代表多考慮的元素有些會成為最佳解，根據轉移來源遞減，可以知道這些元素一定會是 $f(j, [j+1, k))$，也就是最後資料結構中應該會多出 $(j, j+1, k)$ 這一項。那要如何求 k 呢？我們可以從資料結構中從 L 最小的開始看，每當 $f(j, S.R) < f(S.p, S.R)$ 就把 S 丟掉（並更新 S）。最後可分成兩種情況：

2-1. 資料結構沒東西了：$k = n$。

2-2. 資料結構有東西：可知 $k \in [S.L, S.R)$，且是令 $f(S.p, k) < f(j, k)$ 成立的最小值。於是可以二分搜 k，並更新 $S.L = k$。

可以發現如此一更新完再加入 $(j, j+1, k)$，資料結構就被維護好了（符合當初定義），也因此 $dp[j+1]$ 的轉移來源就是 $S.p$！轉移完後，因為 $f([0, j+1), j+1)$ 再也不會被用到了，因此檢查 $S.R$ 是否 $\leq j+1$，如果是就把 S 丟掉即可。

仔細觀察我們在資料結構上的操作：每一次放進資料結構的元素 L 必定最小，並且我們也只會對 L 最小的東西做事情。因此 stack 足以勝任此資料結構。

最後分析複雜度。每一個 p 值最多只會被丟進和丟出 stack 一次，共有 n 個可能的 p 值，$O(n)$；每次轉移 $O(1)$，n 次轉移，$O(n)$；每次二分搜範圍最大 n，最多 n 次二分搜，$O(n \log n)$。故總體複雜度 $O(n \log n)$，比起原本的 $O(n^2)$ 是極大的改進。

3.5 1D/1D 的凸性優化

如果 f 符合凸完全單調性，那做法和凹完全單調性的幾乎一模一樣。只是因為轉移來源會變成遞增，所以維護的資料結構會需要在 L 最小的部分轉移、丟掉用不到的範圍，並且從 L 最大的部分更新資料結構。因此這個結構在兩端都要做事情，必須使用 deque。複雜度仍然是 $O(n \log n)$。

可以發現在這兩種優化當中，時間複雜度的來源主要是二分搜。如果 $f(i, j) = dp[i] + w(i, j)$，且 $w(i, j)$ 有某些性質可以讓我們不需要二分搜集可更新資料結構，那時間複雜度可以再進一步優化成 $O(n)$。

最後，要怎麼發現這些單調性呢？如果 $f(i, j)$ 中所有包含 j 的項有完全單調，那 f 也有（顯然），例如 $f(i, j) = dp[i] + w(i, j)$ 只需要看 w 有沒有單調即可。因為完全單調不好證，所以通常去驗證四邊形不等式是否成立會是一個不錯的方法。當然，對題目有強大的直覺 (?) 也是很重要的。

3.6 2D/1D 轉換成 1D/1D

所謂的 2D/1D DP 是指：

$$dp[i][j] = \min_{i<k<j} f(i, j, dp[i][k], dp[k][j]) \; ; \; i, j \in \mathbb{N}_0, i < j, dp[i][i+1] = 0$$

2D/1D 可以每次枚舉一維之後視為 1D/1D 問題，可以令 $dp[i][j] = A_i[j-i]$：

$$A_i[j] = \min_{0<k<j} f(i, i+k, A_i[k], A_{i+k}[j-k])$$

由於 i, A_i 視為常數，令 $f'(k,j) = f(i, i+k, A_i[k], A_{i+k}[j-k])$，可以看出它變成了 1D/1D 形式。如果對於所有 i，f' 都有凹或凸完全單調性的話，可以透過把 i 由大枚舉到小（因為計算時會用到 $i+k$ 的 dp 值），套用前述的方法。於是複雜度變成 $O(n^2 \log n)$。

套用這種優化的題目較為少見，大約看看就好。

3.7 2D/1D 的凸性優化

這種優化可說是所有四邊形優化當中最常出現的一種。

如果 2D/1D DP 的「DP 表格」（$dp[i][j]$）符合凸四邊形不等式（注意只有凸完全單調是不夠的），並令 $h(i,j)$ 是使 $f(i,j,dp[i][k],dp[k][j])$ 最小的 k 值（特別地，必須設定 $h(i,i+1) = i+1$），那麼可以證明：

$$h(i,j-1) \le h(i,j) \le h(i+1,j)$$

白話文來說，就是 $dp[i][j]$ 的轉移來源必定會在 $dp[i][j-1]$ 和 $dp[i+1][j]$ 的轉移來源之間。那麼如果我們按照 $j-i$ 遞增的順序計算整個 dp 表格，容易發現完整地算完一個 $j-i$ 的所有 dp 值，最多只需要轉移 n 次，也就是總複雜度是 $O(n^2)$！

實作上要注意的是，因為 h 的初始狀態，在實際轉移時還必須考慮 $h(i,j) < j$，以避免存取到未定義的 $dp[j][j]$。另外，因為我們的計算順序，h 只需要用掉一個維度的空間（記左界即可），也可以視為一種簡單的滾動。

這種 DP 優化非常好寫，只是改一下轉移來源的範圍而已，因此重點是要如何發現 dp 表格符合這個性質。除了直覺大法和硬用 DP 表格去證凸四邊以外，還有一個特例有不錯的判斷方法：如果 $f(i,j,dp[i][k],dp[k][j]) = w(i,j) + dp[i][k] + dp[k][j]$，那只要 w 符合凸完全單調且 $w(i,i+3) \ge \max\{w(i,i+2), w(i+1,i+3)\}$，則 dp 一定符合凸四邊形不等式。另外，有些題目你可能可以用直覺發現轉移來源位在兩者之間這件事，如果懶得證明，直接寫出來丟看看也是一個很好的方法（因為很好寫）。

為了方便，以上所有敘述採用的都是左閉右開區間。少數 2D/1D 的題目用全閉區間想可能會稍微好想一點，視狀況靈活運用即可。

3.8 小結

總之 DP 優化是個很深的學問。如果有興趣的話可以自己鑽研，有許多論文都是在討論如何優化特殊條件的 DP 式計算複雜度。此外，上面講的有許多是非常特殊的 case，比

如說四邊形優化。一般來說有更多的式子可以套用這些優化，不過就不列在這裡了。如果能理解精髓，那麼看到新的式子只需要想辦法套用想法，或者想辦法證出它滿足一些你需要的條件（比如說某個東西符合某個單調性）就好了。

不過如果你的理解力不強，因為這些東西的實作複雜度沒有說特別高，你也可以直接實作然後 submit 看看有沒有 AC 啦:p

3.9 習題

1. (TIOJ 1639) 複賽題不解釋。

2. (No judge) 長度為 N 的整數序列要分成數段區間，每個區間的權重是該區間的數和乘上該區間第一個數的值。求權重和最小值。時間複雜度 $O(N \log N)$。

3. (BZOJ 1096)(ZJOI 2007) 有 N 座工廠排成一直線，第 i 座工廠離第 1 座工廠 $x[i]$ 單位長，且有 $p[i]$ 個產品。在第 i 座工廠蓋（容量無限的）倉庫需要耗費 $c[i]$。現在你需要選擇一些地方蓋倉庫，然後將所有產品送往倉庫。然而產品只能往編號大的方向移動，而且一個產品移動一單位長需要耗費 1。請求出最小耗費。時間複雜度 $O(N)$。

4. (TIOJ 1676) 有長度為 n 的正整數序列，可以把它分成若干塊長度不超過 k 的區間。令第 i 個區間的總和是 S_i、長度是 L_i，求 $\sum (S_i(i-1) - L_i^2)$ 的最大值。時間複雜度 $O(n)$。

（提示：正著不會做的話……倒過來呢？）

5. (IOI 2002 Batch Scheduling) 有一些機器，每次啟動的時候都需要花 S 秒。有 N 個任務，第 i 個任務的執行時間是 $T[i]$，費率是 $F[i]$。你可以把這 N 個任務分成幾塊，從編號小的慢慢一塊一塊丟進機器處理。假設你將第 i 到第 j 個任務在時間 t 時丟進了機器，那麼在時間 $t + S + T[i] + \cdots + T[j]$ 時第 i 到第 j 個任務會同時完成。如果第 i 個任務的完成時間是 $O[i]$，那麼總費用是 $O[1]F[1] + O[2]F[2] + \cdots + O[N]F[N]$。請計算總費用的最小值。時間複雜度 $O(n)$。

（註：本題堪稱 IOI 第一題 DP 優化。IOI 的題目可以上 wcipeg，有好用的 judge。）

6. (USACO 2012 Bookshelf) 有一個寬度為 L 的書架，還有 $N \leq 10^5$ 本書，每本書都有寬度和高度。你要把書架分層，並把書全部放進書櫃裡，每一層裡面書的編號都必須連續，且總寬度不超過 L，則該層高度為該層書中最高的一本。求書架最小高度。

7. (IOI 2013)(TIOJ 1842) 題敘太長太難懂怒略。複雜度 $O(C^2 \log R)$ 乘上更新次數。

（註：空間很緊，請犧牲一些時間換取空間的改進。）