資料結構超階、離線演算法

hansonyu123

2016 年 11 月 21 日

1 前言

這次要介紹的東西有樹堆、持久化結構、莫隊、整體二分搜以及操作分治，共有五個主題（這是一個筆者又要爆肝的 tempo(?)）。

雖然看起來這次要講的東西跟 DP 優化那堂課一樣雜，但是這些東西基本上能應付的問題都非常相像，甚至可以說是一體兩面，所以不要被這海量的主題數嚇到了。

1.1 簡介

在介紹線段樹時，已經有介紹過笛卡兒樹了。然而實踐笛卡兒樹時需要自己寫一棵自平衡二元樹，而大部分的自平衡二元樹都要用到高 coding 複雜度的「旋轉」操作（包含 1989 年提出的旋轉式 treap）。因此，分裂/合併式樹堆（merge-split treap）這一資料結構就在近幾年誕生了。Merge-split treap 有許多優點：易實作，期望複雜度 $O(\log n)$，並且有高度的靈活度（這也是為什麼它和線段樹一樣熱門）。基於這些緣故，如果你不想寫 splay tree（大陸人愛用的自平衡二元樹）的話，treap 是一定要會的。

然而使用線段樹、treap 這類資料結構時，有時候可能會需要類似 ctrl+Z（復原）的功能，或者說需要直接看之前的資料結構狀態如何，所以需要存下每個瞬間的狀態。將資料結構複製好幾份顯然不是明智之舉，所以人們想到了利用這些資料結構保證的複雜度衍生出的技巧：持久化。雖然持久化的出發點在這，但持久化能處理的問題卻遠超過它的出發點。

隨著資料結構支援的東西愈來愈多，它們的實作複雜度以及常數大小也隨之增高。為此，幾種偷懶方法就誕生了。其一即是放棄支援在線詢問以求較小實作複雜度的離線演算法。雖然離線演算法一開始都只是為了偷懶，但一樣也可以解決一些別的方法不支援的題目。

總而言之，這些東西基本上都是處理類似的題目的方法，但應用都有一些不同。

接著就讓我們一個一個認識吧！

2 樹堆（Treap）

2.1 簡介

Treap 這一詞其實是結合了 tree 和 heap 兩字所產生的。顧名思義，treap 這一個資料結構應該要同時具備二元搜尋樹（tree）和堆（heap）的性質：每一個節點都存有兩個值 (a, b)，每個節點有零到兩個子節點，中序遍歷這棵 treap 時節點的 a 恰好由小到大遍歷（tree），而每個節點的 b 都比它的兩個子節點的 b 還要大（heap）。滿足這種性質的資料結構我們就稱之為 treap。然而光從 treap 的定義是看不出其用處的，接下來將說明它的主要用途。

要將 treap 投入應用之前，我們需要想想 a 跟 b 應該要是什麼。

因為 a 滿足二元搜尋樹的性質，所以理所當然地我們會希望 a 就是鍵值（key）。至於 b，則是保持樹平衡的重要角色。

事實上，b 有幾個性質，可以幫助維持樹的平衡：

1. 給定 n 個節點的 a、b 的大小關係，那麼這棵 treap 的形狀唯一。

2.（重點）給定 n 個節點的 a，在 n 個節點的 b 都隨機的前提下（也就是 treap 的形狀隨機），任一個選定的節點的期望深度為 $O(\log n)$。

這兩件事實引導我們將 b 取為一個隨機的數。而 2. 在 treap 的複雜度扮演著極重要的角色。

要注意的是雖然我們已經想好 a 跟 b 是什麼了，但這個 treap 上面還可以存更多的東西：你甚至可以把它當線段樹（笛卡兒樹）用，額外存個值（或懶人標）之類的。這樣的性質造就了 treap 的靈活度。

接下來的問題是：treap 應該要支援什麼操作？合理的想法是至少要能支援二元搜尋樹該有的功能：$O(\log n)$ 插入、尋找和刪除元素。那又要如何實作呢？

2.2 Merge-split Treap

在競賽中通常採用的實作方法是 merge-split treap。儘管旋轉式的 treap 常數較優，但 merge-split treap 的實作複雜度是顯著地低（大約和線段樹差不多）。除此之外，merge-split treap 可以做到旋轉式 treap 做不到的事情，就是將 treap 分割以及合併。

Merge-split treap 最重要的兩個功能即為 merge 和 split。merge 是指將一棵鍵值全小於等於 k 的 treap 和另一棵鍵值全大於 k 的 treap 合併，而 split 是指將一棵 treap 拆成兩棵 treap，一棵鍵值全小於等於 k，另一棵鍵值全大於 k。也就是說，merge 和 split 互為逆操作。

至於實作方法也很簡單。直接附上虛擬碼吧。（以下 key 代表前述的 a，pri 代表前述的 b。）

Algorithm 1: Merge-split treap

```
node* merge(node* a, node* b) //將根節點為 a 和 b 的 treap 合併
{
    if (!a) return b;
    if (!b) return a; //記得處理 base case
    if(a->pri > b->pri) { //a 的 pri 比較大，應該當 b 的祖先
        a->r = merge(a->r, b); //b 的 key 較大，應和 a 的右子樹合併
        return a;
    }
    b->l = merge(a, b->l); //a 的 key 較小，應和 b 的左子樹合併
    return b;
}

void split(node* s, int k, node*& a, node*& b) //依照 k 將 treap 分割至 a, b
{
    if (!s) a = b = nullptr; //base case
    else if (s->key <= k) { //s 的 key 較小，故 s 和其左子樹都在左邊
        a = s;
        split(s->r, k, a->r, b); //只剩右子樹要分割
    }
    else {
        b = s;
        split(s->l, k, a, b->l); //只剩左子樹要分割
    }
}
```

可見複雜度為 O(深度)，故期望複雜度 $O(\log n)$。

有了 merge 和 split 後，插入和刪除就變得很顯然了。如果要插入鍵值為 k 的東西的話，先把 treap 分成小於 k 和大於 k 的兩半，然後把鍵值為 k 的節點當作一棵 treap，將三棵 treap 合併即可。刪除也是類似的，先把 treap 分成三棵，將中間（鍵值為 k）的刪除，再將剩下的兩棵合併回來。這也是為什麼 merge-split treap 的常數比旋轉式 treap 還大。至於搜尋，只要直接二分搜就好了，不需要 merge 或 split。

2.3 應用

Treap 在處理序列問題時是一個有力的工具。以下面這個例題來說：

一開始有一個序列。每次可以將序列分成兩半，將兩半交換，再接回去。在進行如次操作的同時詢問當前序列某一位的值。

第一個想法是，把序列拆成兩半，再接回去的這種圖像和 merge-split treap 的實作方法非常相像，所以可以試著用 merge-split treap 處理。每個節點的 key 代表它在序列的位

置，再多存一個 val 代表值。而每次的操作就是把 treap 依照某個 k 分割後，將兩棵 treap 交換，再 merge 起來。

然而這樣做的時候會遇到一個困難，就是將兩個 treap 交換之後，一棵 treap 的 key 需要全部加 k，另一棵 treap 的 key 需要全部減 k。然而每次都對所有節點實施加減是不切實際的事。

但可以發現，當 key 存的是序列中的位置時，這棵 treap 其實就是原序列的笛卡兒樹，也因此 key 這個值顯得有些多餘。原本只有在 split 的時候用到 key，但既然我們要求的是「從序列位置 k 的地方把這棵樹分成兩半」，那麼只要對每個節點記錄以該節點為根的子樹大小，split 時拿左子樹大小和分割位置比較就好。維護子樹大小也很簡單，如線段樹一般實作 pull 函數，在 merge 和 split 時更新即可。需要注意的是 pull 前一定要記得判斷子樹是否為空，不然會 segmentation fault。

當然維護 size 的技巧用途非常廣泛，不只方便對序列做各種變換，也可以應付多樣的查詢。

其實也可以在每個節點多存一個懶人標 tag，代表以它為根的子樹的 key 都需要加上 tag。如此一來，便可以利用線段樹裡懶人標的思想 $O(1)$ 完成區間加值（因為一定是加在根節點）。然而這個方法有點大材小用。

由此可見 treap 的用途是非常廣泛的。事實上，利用這些思想可以解決很多問題。

2.4 小技巧

如果再仔細看一下 treap 的實作，會發現 pri 只用在 merge，用途是以兩棵樹的根 pri 的大小關係，決定兩棵要合併的樹應該由誰來當作祖先。

既然只是比較 pri 的大小，存下 pri 值似乎有些浪費。既然所有節點的 pri 值皆是隨機，且根節點的 pri 是所有 pri 中的最大值，可以發現若兩棵樹的大小分別是 S_A, S_B，那麼 A 的 pri 大於 B 的 pri 的機率約是 $\frac{S_A}{S_A+S_B}$，如此便可以略去 pri 值，在 merge 時用上述的機率隨機即可，便省下了一些記憶體空間，代價是計算隨機數的次數變多。

2.5 習題

1. (TIOJ 1332) 給你一個二元搜尋樹的插入順序，找出每個人的父節點是誰。複雜度 $O(n)$。

（註：這題用 treap 做會 TLE 的。這題的用意是證明性質 1.。）

2. (No judge) 有 n 個數，在線單點修改以及查詢 n 個數中第 k 大的數。

3. (ZJ a063) 有一個長度為 n 的序列以及 m 次操作。每次操作可以把某個區間的數反轉。將最後的結果輸出。複雜度 $O(n + m \log n)$。

（提示：懶人標。）

4. (IOI 2013)(TIOJ 1836) 在線段樹的講義中出現過了，題敘略。

（二維線段樹就算用動態開點，沒有常數優化也會 MLE。由於 treap 的空間比線段樹好，因此把第二維換成 treap 即可。）

3 持久化

3.1 簡介

持久化即是將一個資料結構的許多「歷史版本」存起來。基於這個緣由，有些人把線段樹的持久化版本叫做「版本樹」，大陸人的說法是「主席樹」。

當然每次做事就把整個資料結構全部複製一份絕對是個又 TLE 又 MLE 的悲劇，因此需要聰明地複製。

以線段樹為例，可以發現每次修改的時候最多只會動到 $O(\log n)$ 個節點。為了這 $O(\log n)$ 個被修改的點就大費周章地複製一次這種事情不會有人想做的。然而我又需要兩個版本，要怎麼辦呢？既然大部分的點根本跟之前一樣，那不是就共用就可以了嗎？

事實上持久化實作起來就是秉持著共用的想法，只有被修改到的點才需要新增。以線段樹而言，因為每次最多只動到 $O(\log n)$ 個點，所以每個新版本只會新增 $O(\log n)$ 個節點，這就可以接受了。

至於如何看各個版本的狀況呢？其實不難發現，因為根節點一定會被動到，所以每次都會新增一個根節點。每個版本就可以用該版本的根節點做為代表。

3.2 複雜度

大部分情況下，持久化並不影響時間複雜度，而空間複雜度就只是多了修改到的點數而已，通常和修改的時間複雜度相同。用這種想法不難估計空間複雜度。

然而有些結構持久化後複雜度會變差。其中一例是並查集。仔細思考的話會發現，這是因為均攤分析不再有效：如果在某個版本下修改很耗時間，就一直戳那個版本，複雜度就爛了。於是，並查集的 compression 不再有意義，餘下的只剩 union by rank，複雜度退化為 $O(\log n)$。（其實我想得到的複雜度會退化的可持久化資料結構也就只有這個了……）

3.3 應用

舉一個萬年老例子：序列帶修改詢問區間第 k 小，強制在線。

先來想想靜態（不帶修改）的例子。合理的想法是對答案二分搜，所以需要快速知道一個給定區間比一個數小的個數有多少個。這件事情只需要利用線段樹，每個節點維護該區間的數由小排到大的數列即可。建立的方法就用 merge sort 的方法建立即可。時間複雜

度 $O(n\log n + q\log^2 n\log c)$，空間複雜度 $O(n\log n)$。

能不能改進呢？仔細思考發現問題在計算一個區間內比某數還小的數字個數太花時間了。如果能快速處理這問題的話，複雜度也許能改善。而這其實也跟處理前綴區間比某數還小的數字個數等價。如果有 n 棵線段樹，維護 $[0,i)$ 區間中每個數出現的次數，那麼就可以用「前綴轉區間和」的精神，利用第 l 棵和第 r 棵線段樹求出 $[l,r)$ 區間中每個數出現的次數。於是只需要在線段樹上二分搜答案即可。而第 $i+1$ 棵線段樹其實是在第 i 棵線段樹修改一個點的值後得到的，所以可以用持久化線段樹處理這個問題。時間複雜度 $O((n+q)\log n)$，空間複雜度 $O(n\log n)$。

靜態的問題處理完了，接著就想辦法轉換成動態。剛剛因為是靜態的，所以可以用前綴和轉為區間和。前綴和的動態版本，自然就會聯想到 BIT。於是可以用「BIT 套持久化動態開點線段樹」解決，時間複雜度 $O((n+q)\log n\log c)$（注意在線修改不能離散化），空間複雜度 $O((n+q)\log c)$ 解決動態的問題了。

使用一般的資料結構很難做到如此高難度的事情，由此可見可持久化資料結構的威力所在。

3.4 適用的資料結構

基本上只要是樹狀結構都可以持久化。因此，線段樹、treap、trie、並查集（祖先當做父節點）、左偏樹（一種可合併 heap），甚至是連結串列（可以當作樹的退化）都可以持久化。其中複雜度會退化的只有並查集。至於 treap 持久化後複雜度並不會退化（當然仍是期望複雜度），但要注意的是不同於 treap，持久化 treap 的空間複雜度也是期望複雜度。

3.5 Smart Pointer

雖然可持久化結構的記憶體已經改進過了，但若不丟掉不需要的點的話，還是可能被邪惡的記憶體限制陰。但最麻煩的是，要將某個版本捨棄時不能直接把它的子樹刪除：它已經被共用了。此時，幫助回收記憶體的技巧：smart pointer 就可以幫忙解決這個問題。

基本的想法就是，對每個節點都設一個計數器，代表目前被幾個人指到了。如果沒人指向它的話即刪除該節點。實作上有許多小細節需要注意，這要自己寫了才會發現，建議大家找一個沒事的週末寫 smart pointer，相信這樣週末就會泡湯了 (?)。

通常 smart pointer 的實作方法是實作一個指標型別替代掉 node*，重載 operator*、operator-> 和 operator new（如同之前介紹的偽指標型線段樹），再兼任計數器的角色。如果不重載運算子的話，coding 會變得很恐怖。

順帶一提，C++11 自己有 smart pointer 可用，但其效率極差，所以建議必要之時還是自己實作吧。

3.6 習題

1. (TIOJ 1840) 跟 3.3 提到的例題一樣。不要被邪惡的題目敘述騙了 (?)。

2. (IOI 2012)(TIOJ 1271) 每次可以在字串後加上一個字元，或 undo 最近的 k 個指令（undo 本身也要可以被 undo），或詢問當前字串的某一位字元。複雜度 $O(n \log n)$。
（提示：其實這題沒有你想像中的那麼困難，不寫這些資料結構也做得出來。）

3. (UVa 12538) 每次可以在字串後加上一個字串或移去後面一部分的字串。詢問會要求你印出某個版本的子字串。

4. (No judge) 有一個字串。每次可以將某個子字串複製後插入到字串的某處。每當字串長度超過某個限制 k 以後，就把限制長度以後的字元全部捨棄。將最後的結果輸出。如果操作數 m，複雜度 $O(k + m \log k)$。

5. (TIOJ 1920) 有一個長度為 $N \leq 10^4$ 的序列，對於 $Q \leq 10^6$ 次詢問 (l, r, x)，輸出 x xor $[k, r)$ 內所有數 xor 的最大值，其中 $l \leq k < r$。

4 淺談離線演算法

所謂的離線就是在知道所有要回答的詢問後才開始計算答案。乍看之下離線並沒有什麼好處，但事實上如果題目允許離線的話會多出非常多的作法。

在介紹常用的離線演算法之前，讓我們先看看一個非常原初的離線想法。讓我們再度回到已經討論到爛掉的 RMQ 問題。

之前我們已經給出了若干種作法：sparse table、線段樹、euler tour 轉 LCA 等等的。然而因為 max 沒有反運算，BIT 並不支援此類問題。不過如果題目允許離線的話，便可以用 BIT 解決。

首先，將所有的詢問依照左界由大到小排列。接著，在固定一個 l 時，我們希望有一個資料結構能幫我們快速計算 $c_i = \max[l, i)$（這裡 i 小於等於 l 時就假設是負無限大）。如此一來，就可以將左界是 l 的詢問的答案全部計算出來。接著的問題就是，當 l 逐漸變小時，該如何維護這個資料結構。不難發現，如果令 $b_i = \begin{cases} -\infty, & \text{if } i \leq l \\ a_i, & \text{if } i > l \end{cases}$，那麼 c_i 其實就是 b 的前綴 max。而 l 變小 1 的時候只是將 b_l 變為 a_l（也就是和 a_l 取 max），因此可以用 BIT 解決這個問題。

雖然在這題中我們只是用一個相對好寫的資料結構解決了本來就可以用同樣複雜度在線處理的問題，但之後就可以慢慢看到離線演算法的厲害之處了。

4.1 習題

1. (Codeforces 524E) $n \times m$ ($n, m \leq 10^5$) 的棋盤上有 $k \leq 2 \times 10^5$ 個城堡。之後有 $q \leq 2 \times 10^5$ 筆詢問，對於每筆詢問都要回答某個矩形內的城堡的攻擊範圍是不是涵蓋了那個矩形。允許離線。

(提示：把條件換成更好驗證的等價敘述，再把對行的條件以及對列的條件分開做。)

2. (No judge)(2015 TOI 四模) 有一個長度為 n 的序列，每個數都不超過 c。有 q 筆詢問，對於每筆詢問需回答某個區間內的最小公倍數。答案模 $10^9 + 7$ 後輸出，允許離線。複雜度 $O(q \log q + n \log n \log c)$。

5 莫隊算法（Mo's Algorithm）

前面我們已經看到，如果我們將詢問用某種方法「排好」，說不定就能用一種特殊的方法將它們解決掉了。

假設我們對於兩個「相近的」詢問，可以其中一個用不多的時間算出另一個的答案，那麼說不定可以將所有詢問排成兩兩都不會太遠的順序，可以節省時間複雜度。具體來說，如果已知一個詢問 $[l, r)$ 的答案，可以用 $O(f(n))$ 的時間算出 $[l \pm 1, r \pm 1)$ 的答案，那麼對於兩個詢問 $[l_1, r_1), [l_2, r_2)$，就可以用 $O((|l_1 - l_2| + |r_1 - r_2|) f(n))$ 的時間以一個詢問的答案算出另一個詢問的答案。不難發現，$|l_1 - l_2| + |r_1 - r_2|$ 就是 (l_1, r_1) 到 (l_2, r_2) 的曼哈頓距離，所以可以用曼哈頓最小生成樹的演算法（可自行 google）算出最優的計算方法。複雜度 $O(q \log q)$，找出來的最小生成樹權重和最差到 $O(n\sqrt{q})$。

然而是不是有更簡單，複雜度相同的方法用好的順序計算答案呢？我們試著用分塊的想法解決這個問題。

如果將整個序列分塊，使得 k 個元素為一塊，總共有 $\frac{n}{k}$ 塊。依照某種神奇的依據將所有詢問排列：先比較左界在哪一塊，左界在比較前面的人放在前面；如果左界在同一塊，那麼就依照右界排序。計算答案的順序就直接由前往後計算。

那麼依這種順序計算答案的複雜度會是多少呢？因為左界和右界每變動 1，複雜度就會多 $O(f(n))$，所以我們只需要分別看左界跟右界變動的距離有多少。

對於左界，在同一塊內每次最多變動 k，而在不同塊之間移動距離的總和最多 n，所以變動的距離總和最多 $qk + n$。對於右界，因為在同一塊內都是遞增排列的，所以一塊的變動距離總和最多 n，有 $\frac{n}{k}$ 塊，所以總和最多 $\frac{n^2}{k}$。故總變動量是 $O(\frac{n^2}{k} + qk + n)$，算幾不等式告訴我們當 $k = O(\frac{n}{\sqrt{q}})$ 時總變動量最小，為 $O(n\sqrt{q})$。這和曼哈頓最小生成樹的結果是一樣的，所以通常都採用此種寫法。

雖然莫隊算法通常是用來處理不帶修改的問題的，但對於帶修改的問題，大陸人也研發出了莫隊算法。想法大概是把時間加到第三維，只要時間可以「逆流」就可以用類似的方法找到 $O(n^{\frac{5}{3}})$ 的莫隊算法。

5.1 應用

通常只要 $f(n)$ 小小的就可以考慮使用莫隊算法。我們以區間眾數問題為例：給定一個序列。找出給定區間的眾數出現的次數。

這裡採用莫隊算法。考慮先離散化後，維護兩個陣列，C_k 代表當前 k 這個數出現幾次，S_k 代表當前區間中出現 k 次的數總共有幾種。可以發現，對於區間多一個數/少一個數，都可以用 $O(1)$ 的時間完成維護，因此套用莫隊算法後複雜度為 $O(q \log q + n\sqrt{q})$。

可以發現一般的資料結構難以處理此類問題，由此可見離線演算法的威力。

5.2 習題

1. (ZJ b417) 有一個長度為 $n \leq 10^5$ 的序列以及 $m \leq 10^6$ 的詢問。對每筆詢問，輸出區間內眾數出現次數以及眾數個數。允許離線。

2. (BZOJ 3585) 有長度為 n 的序列及 q 筆詢問。對每筆詢問，輸出區間內的 mex 值（不在區間內的最小非負整數）。允許離線。複雜度 $O(n\sqrt{q} + q\sqrt{n})$。
（註：直接使用 heap 類資料結構能達到 $O(n\sqrt{q}\log n)$。使用線段樹能做到離線 $O((n+q)\log n)$，用線段樹套持久化線段樹能做到在線帶修改 $O((n+q)\log^2 n)$。）

6 整體二分搜

通常我們在二分搜時都會有一些代價。如果有多筆詢問，對每筆詢問都要二分搜的話，代價就會累加起來，實在是有點糟糕。如果要節省時間的話，不妨將所有人「一起」二分搜，共用（以節省）代價。這就是整體二分搜的精髓。

雖然整體二分搜通常都被拿來替代可持久化資料結構，但還是有些題目用整體二分的複雜度較優，甚至是其它資料結構無法處理的。

我們來看一個替代持久化線段樹的例子。再回到帶修改區間第 k 小的那題。

同樣地，先考慮靜態的問題。先將原序列離散化。對於第一次二分搜，我們需要先計算好每一個前綴區間內小於等於 x 的數有幾個，然後對每個詢問計算落在該區間的數有幾個。如果超過詢問的 k 了就代表要的答案在小於 x 的那邊，不然就是大於的那邊。如此一來，我們就可以將詢問分成兩部分。而對於原序列的數來說，它們的影響力只有其中一邊，所以也可以分成兩部分。接著再對兩個部分持續二分搜即可。

然而二分搜時遇到一個問題，就是每次計算前綴區間內小於等於 x 的數的個數花太多時間了。如果把第一次二分搜的資訊遞給右邊的第二次二分搜，那麼只需做幾次的「修改」就可以維護好前綴和了，所以我們採用 BIT 以支援這種操作。

接著是整體二分搜的複雜度分析：對於每個詢問，最多只會二分搜 $O(\log n)$ 次，每次二分搜的時候要花 $O(\log n)$ 的時間確定要被分到哪去，故詢問總時間複雜度 $O(q\log^2 n)$。而對於原序列的每個數，最多只會被 $O(\log n)$ 次二分搜用到，每次被用到的時候都要花 $O(\log n)$ 的時間修改 BIT，故對複雜度貢獻 $O(n\log^2 n)$。總複雜度 $O((n+q)\log^2 n)$。通常整體二分搜的複雜度分析也長得差不多像這樣。

雖然靜態的表現比持久化線段樹差，但不難發現上面的作法可以直接套到動態的問題。關鍵在於將「把 a 修改成 b」看作「刪除 a」以及「加入 b」兩個分開來的操作以確保和它們有關的二分搜最多只有 $O(\log n)$ 個。如此一來，每次二分搜時只要照著時間順序修改以及詢問就不會出問題了。複雜度仍是 $O((n+q)\log n\log(n+q))$（離散化的複雜度變大了一些，通常沒影響），和持久化線段樹的表現差不多。然而整體二分搜只需要寫 BIT，在實作上比持久化線段樹還要容易許多，正因為如此才會被廣泛使用在競賽當中。

值得一提的是上述作法的空間複雜度是 $O(n+q)$，比持久化線段樹還要少了許多。有些題目會藉由卡記憶體「強迫離線」。

6.1 複雜度

上述作法的空間複雜度是 $O(n)$ 的主要原因是這些「修改」是可以還原的，所以右邊二分搜完之後把它們對 BIT 的影響全部還原，再對左邊二分搜即可。但是如果是用不可還原的資料結構（比如說並查集），那麼就需要在每一層暫存被動到的值它們原本的值是什麼，使得空間複雜度提高。

除此之外，整體二分搜套並查集的複雜度會比持久化並查集還要好。這是因為均攤分析在整體二分搜仍然成立，所以複雜度不會退化。

6.2 習題

1. (TIOJ 1840) 請試著用整體二分搜實作這題。

2. (POI XVIII Stage III) 有 m 個太空站和 n 個國家，太空站編號為 1 到 m 並且呈環狀排列，每個太空站都屬於一個國家（可能有一個國家擁有許多太空站，也有可能有國家沒有任和太空站）。現在有 Q 次事件發生，第 i 次事件會讓 $[L_i, R_i]$ 中每一個太空站都獲得 a_i 顆隕石（因為環狀排列的關係，有可能 $L_i > R_i$）。每個國家都有一個隕石收集數量的目標值 p_i，輸出 n 行代表每個國家是在哪一次事件發生後達到其收集目標，若 Q 次事件之後還沒有達到目標則輸出 NIE。$n, m \le 3\times 10^5; a_i, p_i \le 10^9$。

（註：雖然這題可以用持久化線段樹解決，但是原題的記憶體限制很緊，所以寫持久化會 MLE 掉。）

3. (No judge)(2015 TOI 二模) 同上題，但是太空站改為直線排列，並且事件發生時是讓每一個在該範圍有太空站的國家都獲得 a_i 顆隕石。$n, m \leq 10^5; a_i, p_i \leq 10^9$。

7 操作分治

操作分治（又名 CDQ 分治），和整體二分搜一樣都是大陸人用來偷懶不寫可持久化資料結構的技巧。其精髓如同其名，即是對其「操作」進行「分治」。

以前我們都是試圖將 $n+1$ 維問題降為 n 維問題。操作分治的精髓較為不同，其想法是對一個 n 維問題，將時間視為新的維度，將原問題轉為 $n+1$ 維問題後，對時間一維進行分治（即為操作分治一名的由來）以及對 n 維中的某一維排序，以把問題從 $n+1$ 維問題再度降回 $n-1$ 維問題。

儘管這個想法看起來莫名其妙，但在處理帶修改的離線問題時非常有效。套一句大陸人的話，「操作分治主要用途是结合其它几种方法来解决高维偏序问题（竞赛中常见的主要是三维偏序问题）」。翻成繁體（白話）文來說就是原本是帶修改的二維問題（加上時間變成三維）。

實作上，操作分治通常都分為三步：

1. 計算 [l,mid) 的答案

2. 計算 [mid,r) 的答案

3. 計算 [l,mid) 中的操作對 [mid,r) 中的詢問的貢獻

有時候視情況 2. 和 3. 的順序可以反過來。實際用一個例子來說明：

在一個二維平面上，每次可以加入某一個帶權點或者詢問某個點的左上方所有點的權重和。計算每個詢問的答案。

直接用二維 BIT 或線段樹顯然只有 MLE 一途，所以我們試圖用操作分治簡化問題以採用較為簡單的資料結構。

首先，加入時間一維。所以每個「操作」都可以看作是三維空間的點，要嘛是帶權點，要嘛是詢問。接下來，直接對時間一維分治。要計算 [l,r) 中所有詢問的答案，就先計算 [l,mid),[mid,r) 的答案。最後再來看前者對後者詢問的貢獻。可以發現問題轉變為二維的靜態問題：二維平面上給定了一些點，每次詢問某個點的左上方的權重和。

轉變為二維問題後，我們發現如果其中一維，比如說 x，已經排好序的話，那麼這個問題就變成一維動態詢問前綴和的問題了，也就是說可以使用 BIT 解決（當然離

散化是不可避免的）。因此計算 [l,mid) 對 [mid,r) 的貢獻可以在 $O((r-l)\log q)$ 的時間處理完畢，這裡 q 是詢問總數。而將 x 排序好的方法可以直接用 merge sort 以避免複雜度多一個 log，達到總複雜度 $O(q\log q)$。而單就用 BIT 計算貢獻的部分，時間複雜度為 $T(n)=2T(n/2)+O(n\log q)$。主定理告訴我們 $T(n)=O(n\log n\log q)$，故總複雜度 $O(n+q\log^2 q)$。

可以發現上面這作法的精髓完完全全就是經典的分治，只是處理的題目稍微不同，用到的輔助工具也稍微先進了一點。

另外，分治的過程中同時進行 merge sort 是分治常用手法，在操作分治也不例外。雖然通常用不用 merge sort 不會影響到總複雜度（因為在合併的過程中複雜度偏高了一點），但養成寫 merge sort 的好習慣是降低常數的好開始。

7.1 習題

1. (Balkan OI 2007)(BZOJ 1176) 一個 $N\times N$ 的矩陣，初始值皆為 S，M 次增加一個點的值並 Q 次詢問一個子矩陣的數總和。允許離線。$N\le 2\times 10^6, Q\le 10^4, M\le 1.6\times 10^5$。
（提示：能不能等價成前面的那題呢？）

2. (BZOJ 3295) 有一個 1 到 $n\le 10^5$ 的排列以及一些操作。每次操作都會把序列中的某個數移去。請在每次操作結束後計算出當前逆序數對數。允許離線。
（註：這題也可利用持久化線段樹解決。）

8 結語

九次的培訓、四次練習賽、兩次校內賽以及一次北市賽就這樣過去了。我們也把我們會的大部分的東西（甚至是我們也不太會的東西）濃縮成精華寫在這八份講義裡了。或許第一次聽完之後無法完全吸收（廢話），但只要你願意花時間把九次的講義盡可能地多吸收點，多練習 coding，相信你一定會大幅度進步的。

至於最佳的吸收講義的時間，當然就是寒假啦！「想當初我寒假 TLE 了人生第一題 POI(?)，寫了人生第一棵平衡樹（AVL 樹）(咦)，還學會了 Kruskal、Prim 演算法，以及終於知道 KMP 不是播放器了 (?)。就是因為有這個寒假，我才有基礎通過……校隊補選 (?)。」

上面這個國手的故事告訴我們：寒假可以做的事很多，如果增進自己的編程以及思考演算法的能力是你想做的事的話，千萬不要讓寒假就這樣過去了。

最後，希望大家接下來的三次練習賽多多加油，全國賽加油，還有更重要的是，TOI 加加油，挽回建中名譽！