資料結構進階

Yihda Yol

2016 年 10 月 17 日

1 前言

這一次我們將會聚焦在一些關於序列的資料結構。通常這種問題會是給你一個序列，並且可能會對它做一些操作（可能是詢問或修改），並對於每次的詢問輸出答案。

這種題目有分「在線」或「離線」兩種做法，在線就是你每接到一個操作就立刻把所有該更新的東西更新好並回答詢問，離線就是把所有的操作都吃進來之後再一次回答所有的詢問。

為了方便，以下的複雜度將以 $\langle O(f(N)), O(g(N))\rangle$ 表示，代表預處理的時間複雜度是 $O(f(N))$，每次操作則是 $O(g(N))$。

2 Sparse Table

Sparse Table 是用來解決區間最小值問題（Range Minimum Query, RMQ）的資料結構。

RMQ 問題十分簡單易懂：給你一個序列 $A_{[0,N)}$，要求詢問 $A_{[i,j)}$ 中所有元素的最小值。如果用暴力法求，複雜度是 $\langle O(1), O(N)\rangle$。然而我們可以對原數列做一些預處理，來達到更好的查詢複雜度。

Sparse Table 的預處理是建立一個二維陣列 D，其中 $D_{i,j}$ 代表編號在 $[j, j+2^i)$ 範圍內所有數的最小值。這樣一來空間複雜度 $O(N \log N)$。

有沒有覺得這種技巧似曾相識？之前提到樹的最低共同祖先時，有介紹過「倍增法」，這裡的概念其實差不多。可以發現 $D_{i+1,j} = \min\{D_{i,j}, D_{i,j+2^i}\}$，於是整個預處理的複雜度是 $O(N \log N)$。如果要查詢 $[l, r)$，先找出 $p = \lfloor \log_2(r-l) \rfloor$，可以發現答案會是 $\min\{D_{p,l}, D_{p,r-2^p}\}$。如此，複雜度是 $\langle O(N \log N), O(1)\rangle$。注意求 p 要在 $O(1)$ 內做完，方法對每個範圍內的 i 預處理 $\lfloor \log_2 i \rfloor$。

其實 RMQ 的詢問的「最小值」可以換成有交換律、結合律且滿足 $f(x,x) = x$ 的任意二元運算 f。

顯然這種資料結構允許在線。然而，這種資料結構無法更新序列的數值：每次更新會影響到 D 中包含該位置的值，而這樣的值有 $O(N)$ 個，時間複雜度 $O(N)$。

順帶一提，RMQ 問題有 $\langle O(N), O(1)\rangle$ 解，方法是利用笛卡爾樹（中序遍歷結果是原數列的樹）的性質和把原數列分成大小為 $\frac{\log N}{4}$ 的塊。然而因為實作過於複雜而不實用。

3 樹狀樹組（Fenwick Tree, BIT）

樹狀樹組，英文叫做 Fenwick Tree 或 Binary Indexed Tree，又簡稱 BIT，由 Fenwick 在 1994 年提出，用來解決「動態前綴和」問題。

動態前綴和問題就是給你一個序列 $A_{[1,N]}$，要求支援兩種操作：把 A_i 改成 $A_i + k$（修改），或詢問 $\sum_{i=1}^{j} A_i$ 的值（查詢）。（為了介紹方便，本節改用 1-base 與全閉區間。）

暴力法的時間複雜度是 $\langle O(1), O(1), O(N)\rangle$ 或者 $\langle O(N), O(N), O(1)\rangle$（維護前綴和），分別是預處理、修改和查詢的複雜度。這兩種做法都讓一種操作很快而另一種很慢。那有沒有折衷的方法呢？

我們可以沿用 sparse table 的想法：令 $p = \lfloor \log_2 j \rfloor$，把查詢的區間拆成 $A_{[1,2^p]}, A_{[2^p+1,j]}$ 兩個部分，再對後面那個區間遞迴拆下去，可以發現這樣的拆法相當於把 j 的二進位表示中所有 1 的位拆出來，因此最多只會拆成 $1 + \log_2 j$ 份。如果對於每一份的結果都可以 $O(1)$ 回答，那查詢的複雜度便是 $O(\log N)$。

仔細觀察可以發現：對於任何 j 的任何一份分拆 $A_{[l,r]}$，$l-1$ 和 r 的二進位表示法只有一個位元 q 不一樣，而且比 q 更低的位元必然都是 0。也就是說，對於任何分拆，右界 r 必定唯一對應到一個左界 l，且 $l-1$ 是 r 二進位表示法中最低位的一個 1 換成 0 代表的數。於是，我們維護一個數列 D，其中 D_r 代表 $\sum_{i=l}^{r} A_i$。如此空間複雜度是 $O(N)$。

最後檢查一下修改的影響：D 中包含 A_k 的項是 $D_k, D_{f(k)}, D_{f(f(k))}, \cdots$，其中 $f(k)$ 是把 k 的二進位表示法中最低位的 $01\cdots1$ 換成 $10\cdots0$，因此總共也不超過 $1 + \log_2 N$ 個。

剩下的問題就是要如何計算分拆結果與 $f(k)$。查詢時是要把最低位的 1 換成 0，更新時是要把最低位的 $01\cdots1$ 換成 $10\cdots0$，因此若以 lowbit(k) 代表 k 只留下最低位的一個 1 代表的數，那兩種操作就分別是減掉和加上 lowbit(k) 了。

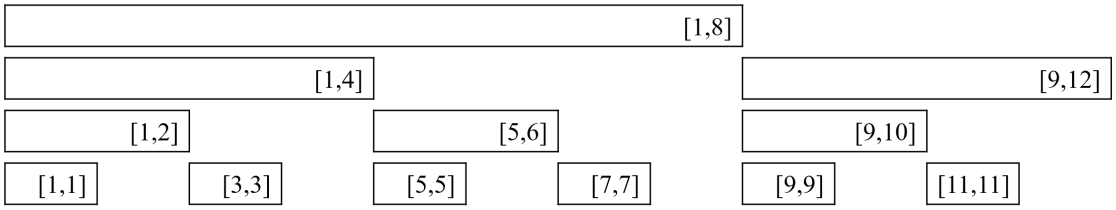
計算 lowbit(k) 的方法，當然可以二分搜最低位 $O(\log\log N)$，但是可以證明 lowbit$(k) = k$ & $((\sim k) + 1)$（& 是 bitwise-and、$\sim$ 是 bitwise-not），因此達到 $O(1)$。（由於電腦以二補數表達負數的特性，可以直接寫作 k & -k。）

預處理的部分，當然可以從一個全 0 的序列一個一個數修改上去，$O(N \log N)$。但是其實可以達到 $O(N)$，一種方法是從原數列開始把間距為 2^i 的值相加，另一種則是處理好前綴和後從後往前扣掉前 lowbit 個位置的值。

故 BIT 的複雜度為 $\langle O(N), O(\log N)\rangle$，允許在線，是極好寫且常數極小的資料結構。

其實「前綴和」的「和」（加法）可以改成任何一種符合交換律和結合律的二元運算 f。容易發現，如果這種二元運算存在反運算 f'（滿足 $f(f'(x,y),y)=x$ 的運算，例如對於加法，其反運算是減法），那麼更新便可以將數改成任何值，查詢時也可以查詢任意區間而不限於前綴。

至於為甚麼 BIT 的名子有「Tree」呢？把它每一個元素代表的區域畫出來，會發現它的結構其實就是一顆二元樹，如下圖。



3.1 高維版本

BIT 可以推廣到高維度，所查詢的便是 $(1,1,\cdots,1)$ 到 $(a_1,a_2,\cdots,a_D)$ 的總和。方法也很簡單，就是想像 BIT 裡面每一個元素都是一個 BIT，如此套 D 層。如果每個維度的大小為 N，那複雜度就是 $\langle O(N^D), O(\log^D N)\rangle$；如果有逆運算而要查詢任意區間，因為要使用排容原理計算，查詢複雜度變成 $O(2^D \log^D N)$。

3.2 習題

1. (No judge) 給你一個長度 N 的序列，要求兩種操作：把一個範圍的數全部加上 k，或查詢第 i 個值是多少。$\langle O(N), O(\log N)\rangle$。

2. (TIOJ 1907) 初賽題不解釋，請嘗試用 BIT 實作 LIS。

3. (TIOJ 1080) 請嘗試用 BIT 實作逆序數對。

4. (TIOJ 1228) 複賽題不解釋。

5. (TIOJ 1869) 裸二維 BIT。

6. (TIOJ 1483) 有個 $R \times C$ 的網格，每格都有一臺電腦，你要從左上角走到右下角，只能走捷徑，過程中可以檢查一些電腦。每個電腦都會有「糟糕度」，如果這臺電腦的糟糕度比前面所有檢查過的糟糕度都還高，你就可以檢查這臺電腦。求你在至少檢查一臺電腦的前提下，被你檢查的電腦「糟糕度序列」可以有幾種。

7. (TIOJ 1345)(IOI 2007) 給定一維、二維或三維空間中的 $N \le 10^5$ 個格子點，求有幾個點對的曼哈頓距離（所有維度座標值差的絕對值的和）不超過 D。一維、二維、三維的任意座標值分別不超過 $7.5 \times 10^6, 75000, 75$。

8. (TIOJ 1403) 有 $N \leq 2.5 \times 10^5$ 臺車，每臺車都在數線上同向行駛。給定每臺車的初始速度和初始位置，求總共會有幾次超車，以及前 10^4 次超車是誰超過誰。保證不會有兩個超車事件同時發生。

4 線段樹

線段樹是競賽選手發明的一種資料結構，正式出現於 2001 年（雖然之前已經有類似的想法出現，例如 BIT），之所以存在是因為 coding 複雜度不高，但是在實際應用基本上沒什麼用處，也沒有正式文獻。因此，「線段樹」（segment tree）此名其實為非正式名稱，更糟糕的是還跟一個早就存在的資料結構撞名了（詳細可以自己查），所以如果是在非資訊競賽的場合，千萬不要亂用這個名稱。

線段樹可以看成是 BIT 的推廣。BIT 因為只需要查詢前綴，所以觀察其結構（見前頁），有很多區間被省略了（如 [3,4]、[5,8] 等），也因此可以完美地塞進一維陣列裡。線段樹的想法就更為直接，如果把那些省略的區間全部補回來，就變成一棵貨真價實的二元樹。如果要查詢任意區間，就不需依賴逆運算，可以直接從現有的區間湊出來，時空複雜度也不改變。

既然如此，為了簡化實作，可以稍稍改變實作方法：相對於 BIT 的由下往上建立（bottom-up），一般採用更為直觀的由上往下（top-down）。將原數列當作根結點，接著將數列切半，分別當作左右子樹，再對切半後的數列遞迴下去，切到區間長度只剩下 1 為止，該葉節點的值便是原數列。至於如何知道每個非葉節點的值呢？可以在遞迴回來的時候一併處理：把該節點兩子樹的值合併起來（例如求和就是加起來）。這樣便在 $O(N)$ 的時間完成了線段樹的建立。

修改也是一樣的道理：遞迴到代表該位置的葉節點，修改葉節點的值，回來的時候照樣更新所有非葉節點的值。因為線段樹的高度是 $O(\log N)$，這也是修改的時間複雜度。

查詢的方法也類似。用相同的方法從根遞迴下去，若欲查詢的區間是 $[ql, qr)$，中途遇到的每個節點都會代表一個區間 $[l, r)$，則會有三種情況：

1. $ql \leq l$ 且 $r \leq qr$：此區間被欲查詢區間完全包含，直接把該節點的值合併到答案裡。

2. $r \leq ql$ 或 $qr \leq l$：此區間與欲查詢區間互斥，忽略此節點。

3. 其它：對兩子樹遞迴下去。

這樣一來，總共只會考慮到 $O(\log N)$ 個節點，時間複雜度 $O(\log N)$。

綜上，若合併的複雜度是 $O(1)$，線段樹的複雜度是 $\langle O(N), O(\log N)\rangle$。因為查詢的答案完全用存在的區間合併而成，不要求反運算，我們也因此解決了帶修改的 RMQ 問題。

因為線段樹十分重要，所以在這裡多提一些實作細節。

線段樹的實作方法有三種：指標型、偽指標型和陣列型。指標型是每個節點開個 struct，裡面有指向左右子節點的指標；偽指標型是把指標換成陣列的索引值；陣列型則是利用線段樹是平衡二元樹的性質，令位置 1 代表根節點，位置 x 的左右節點分別是 $2x$ 和 $2x+1$。這三種實作方式的空間和直觀程度都是由大到小（因為指標通常占用 8 byte 的空間，是 int 的兩倍），因此以下皆以指標型舉例，其它方法可以視需要自行類推。

首先，因為每次對線段樹做事時，遞迴的方式永遠都相同，因此每個節點所代表的區間不應該儲存在節點中，而是跟著遞迴的動作一起算，可以省下大量的空間。

其次，查詢時要仔細思考要怎麼處理「區間互斥」這個情況，在簡單的問題例如 RMQ，可以透過回傳一個不影響答案的值（inf）解決，但是有些時候沒辦法如此做，就需要在遞迴下去前先判斷區間是否互斥，實作時需要注意。

最後，「將兩個子節點的資訊合併為當前節點的資訊」這個動作常稱作「pull」。通常會將它直接寫成節點的成員函數，呼叫起來比較方便。

以下是帶修改 RMQ 的指標型線段樹，有一些常見的 coding 技巧，可以自行觀察。

Algorithm 1: Dynamic RMQ Using Pointer-based Segment Tree in C++

```
1  struct node {
2      int val;
3      node *l, *r;
4      node(int v = 0) : val(v) {} //在建構式初始化 struct 的成員
5      node(node* l, node* r) : l(l), r(r) { pull(); } //撞名無妨
6      void pull() { val = min(l->val, r->val); }
7  };    //l->val 就是 (*l).val，注意. 的優先順序比 * 還要高
8
9  int v[N]; //原數列
10 node* build(int l, int r) {
11     if (l + 1 == r) return new node(v[l]);
12     int mid = (l + r) / 2;
13     return new node(build(l, mid), build(mid, r));
14 }
15 void modify(node* a, int l, int r, int pos, int k) { //把 pos 位置的值換成 k
16     if (l + 1 == r) { a->val = k; return; }
17     int mid = (l + r) / 2;
18     if (pos < mid) modify(a->l, l, mid, pos, k);
19     else modify(a->r, mid, r, pos, k);
20     a->pull();
21 }
22 int query(node* a, int l, int r, int ql, int qr) { //查詢 [ql, qr) 範圍的最小值
23     if (r <= ql || qr <= l) return inf;
24     if (ql <= l && r <= qr) return a->val;
25     int mid = (l + r) / 2;
26     return min(query(a->l, l, mid, ql, qr), query(a->r, mid, r, ql, qr));
27 }
```

線段樹是個極度靈活的資料結構，也因此在資訊競賽界廣受歡迎。每個節點不一定只能存一個值，可以存很多個，甚至可以存一整個資料結構。可以注意到，如果每個節點都存一個相當於該節點長度大小的資料結構，空間複雜度是 $O(N \log N)$；建立和修改線段樹的複雜度取決於 pull 的複雜度，通常用主定理即可計算。

以下幾個小節即是線段樹常見的變化方式。

4.1 懶標記

現在線段樹已經可以提供區間查詢和單點修改的操作了，但是如果一次修改很多個點，複雜度還是爛掉。能不能將其「進化」成區間修改的版本呢？答案是只要區間修改符合以下三個性質，那線段樹就可以實現：

1. 兩個對於相同區間的區間修改，要可以快速合併成相同區間且相同型式的區間修改。

2. 一個對於 $[l, r)$ 的區間修改，對於任意 $l < m < r$，都要可以快速拆成對於 $[l, m)$ 和 $[m, r)$ 相同型式的區間修改。

3. 對於線段樹任意節點和任意對於該節點的區間修改，都要可以僅利用該節點的資訊就計算出該節點修改後會變成甚麼樣子。

這麼一來，只要對每個節點維護「這個節點需要進行甚麼樣的區間修改」，實作區間修改便如同查詢，分成三種狀況考慮即可：區間完全包含時，把當前節點已有的區間修改與當前的修改合併（性質 1.）；互斥時不用做事；遞迴下去時則要將區間修改拆成兩個（性質 2.）。如此便可在 $O(\log N)$ 時間內完成區間修改。

維護的這個東西通常被稱為「懶標記」（lazy tag），因為只記錄這個節點要進行的修改，並沒有實際去修改它，等到真正使用到的時候再處理，可以看成「偷懶」。

也因為在區間修改時並沒有更新答案，因此在任何操作（修改與查詢）時剛抵達一個節點時，如果當前節點有懶標記，必須更新當前節點的資訊（性質 3.）候才能使用；另外它的子樹也尚未更新，故還要將懶標記分給它的兩個子樹（性質 2.）。以上兩個動作可以併成一個操作，通常稱為「push」，一樣可以寫成成員函數以簡化程式。

懶標記還有另一種實作方式：在打懶標記的同時更新節點資訊，而 push 的時候是先把懶標記下推，並更新子節點資訊。如此就只需要在遞迴造訪子樹前 push 即可，也避免了 push 時額外判斷是否為葉節點的麻煩。

事實上，大部分的區間修改，只要在節點維護正確的資訊，以及想好區間修改要用甚麼樣的懶標記記錄，幾乎都能符合上列性質。例如，若動態 RMQ 的區間修改是把一個區間每個值都加 k，那懶標記就只要存一個數，代表這個區間要加多少。又或者，若動態區間和的區間修改是「將該區間的第 i 個數加上 ik」，那懶標記就存兩個數 a, b，代表「此區間第 i 個數要加 $ai + b$」，也能符合上列性質。

4.2 動態加點

有些題目會要求很大的 N 值（例如 10^9），並且給定一個初始狀態（例如全 0），再開始修改與查詢 Q 次。如果允許離線的話，可以把所有操作全部讀進來離散化處理。若是強制在線，顯然如果一開始就把所有節點開好會徹底 MLE+TLE 掉。

但不難發現，其實序列中有很多位置會保持初始狀態，而就算有區間修改，也有很多會保持區間修改的狀態，因此修改時再把節點 new 出來即可。如此若在查詢時遇到一個空子樹，代表那個區間的樹全部都處於初始狀態（或懶標記所代表的狀態），可以直接計算答案。如此，每次修改最多增加 $O(\log N)$ 個節點（無論單點或區間），而查詢則不會新增節點，總空間與時間都是 $O(Q \log N)$。

這種線段樹不能用陣列實作。建議除非要狂壓記憶體，否則用指標線段樹就好了。

這裡附帶一個 C++ 小技巧：如果一開始先把記憶體空間開好，那在 new 節點的時候可以寫「new(某指標) 型別名稱」，那麼這個新的東西就會被建立在指定的記憶體位置，可以方便快速回收記憶體。如果一開始開的記憶體空間是個陣列，那就可以輕鬆地寫出偽指標型線段樹；透過對 node 加載 new 運算子，並在裡面自訂一個 ptr 型別並定義-> 運算子，就可以讓偽指標型線段樹的寫法變得跟指標型一模一樣。

這個技巧可以套用在任何用到指標的資料結構，尤其是在持久化資料結構（之後會提到）上更加方便。以下附上程式碼：

Algorithm 2: Segment Tree, Using int as Pointer in C++

```
1  struct node {
2      static node mem[K]; //預先開好記憶體
3      static int top;
4      struct ptr {
5          ptr() {}
6          ptr(const node* s) : c(s − mem) {}
7          int c; //在陣列的位置
8          node* operator−>() { return mem + c; }
9      } l, r; //左右子樹的偽指標
10     void* operator new(size_t x) { return mem + top++; } //可額外初始化或重載
11 } node::mem[K]; //不要忘記要把 static 變數宣告出來
12 //現在 node::ptr 就可以當成 node 的指標了
13 int node::top; //將這個設為 0 即可重新利用記憶體
```

4.3 高維版本

線段樹也可以像 BIT 一樣推廣成高維度，方法也一樣，就是線段樹的每個節點是一棵線段樹，如此套 D 層，此時每個節點代表的便是一個矩形（或 D 維矩形）區域，複雜度便是 $\langle O(N^D), O(\log^D N)\rangle$。

但是有一件必須要記住的事情：高維的線段樹不能同時提供區間修改與區間查詢兩個功能，只能擇一。原因很簡單，就是會出現兩個節點 A, B，其代表的區域有相交，但是這兩個節點卻不是父子關係，因此如果對 A 區間修改，按照遞迴策略，不會更新到 B 所代表的值，因此直接查詢 B 便會發生錯誤。一個簡潔的例子是 2×2 的二維線段樹，$A = [(0,0),(1,2)), B = [(0,0),(2,1))$。

另外，如果是區間修改單點查詢也需要注意，查詢 (x, y) 時必須將第一維包含 x 的所有節點都進入第二維查詢 y，並將所有區間修改對 y 造成的影響累加。

如果要同時具有區間修改和區間查詢的功能，以二維為例，就要使用另一種「四叉版」線段樹。方法是直接把每個節點切成四塊（在兩維各切一刀），每塊遞迴下去，用相同的方法建立、查詢、修改。仔細分析會發現，這種資料結構的區間修改和區間查詢複雜度只能達到 $O(N)$，但是相較於暴力的 $O(N^2)$ 或一維暴力一維線段樹的 $O(N \log N)$，仍是一種改進。

4.4 笛卡爾樹

前面有提到線段樹沒有甚麼實際應用，其實原因不難理解。對於每一個序列，都存在中序遍歷是它的二元樹，這種樹稱為笛卡爾樹，也就是序列轉換成的樹。只要能讓這棵樹是平衡的，我們便可以在這棵樹上面用相同的複雜度，實踐套用在線段樹上的所有技巧，因此這棵樹擁有所有線段樹的功能。更甚者，如果利用平衡二元搜尋樹的方法平衡這棵樹，那麼這個序列甚至可以支援任意增刪元素、分裂、重組等線段樹無法做到的操作。一個功能比較少而且常數又比較大（明顯地，線段樹多了一倍的節點）的東西自然沒有甚麼應用價值。

之後的課程中會介紹到「樹堆」（treap）這種資料結構，其實就只是一種平衡二元搜尋樹的方法，再透過樹與序列的對應關係，利用所有線段樹的技巧，實現一個更有彈性的序列結構。

4.5 習題

1. (TIOJ 1224) 求平面上 $N \leq 10^5$ 個矩形覆蓋的總面積。

2. (TIOJ 1408) 現在有 $C \leq 10^5$ 個時刻，每個時刻可以處於忙或不忙兩種狀態。給定 $N \leq 10^5$ 個條件，代表在時刻 $[A_i, B_i)$ 中最少有 C_i 個時刻要處於忙的狀態。求若要滿足所有條件，最少要有幾個時刻處於忙的狀態。

3. (IOI 2013)(TIOJ 1836) 有一個 $R \times C$ 的格子，一開始每個格子的數字都是 0。有兩種操作：詢問一個矩形區域所有數的最大公因數，共 N_Q 次；或改變一個位置的數，共 N_U 次。對於每個詢問輸出答案，強制在線（雖然 TIOJ 沒有）。$R, C \leq 10^9$，$N_U \leq 22000, N_Q \leq 2.5 \times 10^5$。