資料結構（二）Data Structure

陳俊安 Colten

2022 新化高中 x 嘉義高中 x 薇閣高中 資研社暑期培訓營隊

2022.07.13

資料結構（二）

- 倍增法
- 倍增法二分搜
- 稀疏表 Sparse Table
- Policy Based Data Structure
- Binary Indexed Tree (Fenwick Tree)
- BIT 解動態規劃問題
- BIT 解分治問題
- BIT 上二分搜與 BIT 上倍增法二分搜
- 線段樹 Segment Tree
- 線段樹與懶惰標記
- 線段樹上不太一樣的區間合併

原本還有要教的東西

- 線段樹上二分搜
- 動態開點線段樹
- 可回朔 Disjoint Set
- 時間分治線段樹

雖然真的很想分享更多的知識讓大家知道，但時間有限，這邊就挑幾個比較重要的東西，希望大家能在這堂課收穫很多

使用前說明書

- 以下內容分治講師與動態規劃（二）講師可能會有非常強烈的副作用，希望你們不要高血壓
- 在這邊會教你們怎麼把之前某些教過的分治題或動態規劃題用資料結構做出來
- 很多人可能曾經都很想學怎麼自己實作一個大資結，像是線段樹這些東西，但又覺得很難
- 我會盡量用最簡單的方式帶大家理解這些資料結構的原理跟實作方法
- 建議大家今天下午就先去實作看看最基本的裸題，看看能不能刻出一個完整能用的資料結構，有其他時間再來寫進階題吧！
- 資料結構是一門藝術，讓我們一起欣賞吧

可能會需要知道的圖論術語

- 簡單路徑： 一條沒有任何點被重複走過的邊
- 樹： 任意兩點恰好只有一條簡單路徑的圖
- 根結點： 樹上深度最小的那一個點
- 父結點： 子結點的父親
- 左子結點： 父結點的左兒子
- 右子結點： 父結點的右兒子

可能會需要的知識

- **離散化：** 簡單來說就是將所有數字改以數字大小的排名取代，以此讓我們可以順利的將資訊以索引值的方式放置在陣列裡面
- **例如：** $\{10, 20, 30\} = \{1, 2, 3\}$

離散化的實作（二分搜）

```
19    int n;
20    cin >> n;
21
22    vector<int> a(n),b(n);
23    for(int i=0;i<n;i++) cin >> a[i] , b[i] = a[i];
24
25    sort(b.begin(),b.end());
26
27    b.resize( unique(b.begin(),b.end()) - b.begin() );
28
29    for(int i=0;i<n;i++)
30    {
31        a[i] = lower_bound(b.begin(),b.end(),a[i]) - b.begin() + 1;
32    }
33
34    for(int i=0;i<n;i++) cout << a[i] << " ";
```

區間離散化

- 如果現在給你的東西是一個區間，則也是可以進行離散化的，只是一開始要將區間的左界跟右界都一起丟進 `vector` 裡面重新賦予離散化後的值
- 如此一來區間左界與右界的值就可以有效的變小
- 舉例： $[10, 10^9]$ 與 $[3, 10]$ 這兩個區間一起離散化後分別會變成 $[2, 4]$ 與 $[1, 2]$

區間離散化

```
19      int n;
20      cin >> n;
21
22      vector<int> l(n),r(n),b;
23      for(int i=0;i<n;i++)
24      {
25          cin >> l[i] >> r[i];
26
27          b.push_back(l[i]);
28          b.push_back(r[i]);
29      }
30
31      sort(b.begin(),b.end());
32
33      b.resize( unique(b.begin(),b.end()) - b.begin() );
34
35      for(int i=0;i<n;i++)
36      {
37          l[i] = lower_bound(b.begin(),b.end(),l[i]) - b.begin() + 1;
38          r[i] = lower_bound(b.begin(),b.end(),r[i]) - b.begin() + 1;
39      }
40
41      for(int i=0;i<n;i++) cout << l[i] << " " << r[i] << "\n";
```

倍增法

倍增法

- 其實我不知道為什麼其他講師說倍增法是屬於這個單元的
- 概念跟動態規劃非常相似，在某些操作的先後順序不影響最後結果的題目可以使用
- 大部分的常見題目通常都是以 2 的冪次來倍增

倍增法

CSES Problem Set Planets Queries I

現在有 n 個行星，每一個行星上有一個數字 a_i $(1 \leq a_i \leq n)$，表示如果從第 i 個行星移動一步會傳送到 a_i，現在有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每組查詢將詢問從行星 x 移動 k 步會到哪一個行星

倍增法

CSES Problem Set Planets Queries I

現在有 n 個行星，每一個行星上有一個數字 a_i $(1 \leq a_i \leq n)$，表示如果從第 i 個行星移動一步會傳送到 a_i，現在有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每組查詢將詢問從行星 x 移動 k 步會到哪一個行星

- 如果現在走了 2^k 步，那麼其實等價於先從起點走 2^{k-1} 步到達另外一個點 P，再從 P 走 2^{k-1} 步，到達的點就是最後所在的點
- $2^{k-1} + 2^{k-1} = 2 \times 2^{k-1} = 2^k$
- 基於這樣的概念，我們就可以使用倍增來完成

倍增法

- 定義 dp[i][k] 表示從行星 i 開始，移動 2^k 步會到達的行星

倍增法

- 定義 dp[i][k] 表示從行星 i 開始，移動 2^k 步會到達的行星
- 一開始輸入 a_i 時，先把 $dp[i][0] = a_i$
- 因為一開始我們就可以知道從第 i 個星球走 $2^0 = 1$ 步會到 a_i
- 而我們必須要先設定好初始的狀態，才能依靠倍增的方式把表一層一層的疊出來

倍增法

- 接下來我們就可以試著列出轉移狀態
- 如果我們從 i 開始出發走 2^k 步，那麼我們會到達的點就會是 $dp[i][k-1]$

倍增法

- 接下來我們就可以試著列出轉移狀態
- 如果我們從 i 開始出發走 2^k 步，那麼如果我們先走 2^{k-1} 步會到達的點就會是 $dp[i][k-1]$
- 那麼剩下的 2^{k-1} 步，就變成我們要從 $dp[i][k-1]$ 出發，走 2^{k-1} 步

倍增法

- 接下來我們就可以試著列出轉移狀態
- 如果我們從 i 開始出發走 2^k 步，那麼如果我們先走 2^{k-1} 步會到達的點就會是 $dp[i][k-1]$
- 那麼剩下的 2^{k-1} 步，就變成我們要從 $dp[i][k-1]$ 出發，走 2^{k-1} 步
- 因此我們可以得到轉移式 `dp[i][k] = dp[dp[i][k-1]][k-1]`

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!!（我都叫他跳跳二分搜）

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!! (我都叫他跳跳二分搜)
- 倍增法二分搜的核心概念是，能跳到多遠的地方就跳到多遠，整個過程就真的是跳來跳去，所以我都叫他跳跳二分搜

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!! (我都叫他跳跳二分搜)
- 倍增法二分搜的核心概念是，能跳到多遠的地方就跳到多遠，整個過程就真的是跳來跳去，所以我都叫他跳跳二分搜
- 不過我來詳細拆解一次要怎麼用跳跳二分搜跳出答案來

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!!（我都叫他跳跳二分搜）
- 倍增法二分搜的核心概念是，能跳到多遠的地方就跳到多遠，整個過程就真的是跳來跳去，所以我都叫他跳跳二分搜
- 不過我來詳細拆解一次要怎麼用跳跳二分搜跳出答案來
- 假設要走 5 步，而 5 被轉成二進位的結果是 101，這一個 101 跟 2 的冪次有什麼關係呢？

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!! (我都叫他跳跳二分搜)
- 倍增法二分搜的核心概念是，能跳到多遠的地方就跳到多遠，整個過程就真的是跳來跳去，所以我都叫他跳跳二分搜
- 不過我來詳細拆解一次要怎麼用跳跳二分搜跳出答案來
- 假設要走 5 步，而 5 被轉成二進位的結果是 101，這一個 101 跟 2 的冪次有什麼關係呢？
- $5_{(10)} = 101_{(2)} = 2^0 + 2^2$

倍增法

- 照著 `dp[i][k] = dp[dp[i][k-1]][k-1]` 這一個轉移式建出所有的表之後，剩下的問題就是處理查詢了
- 究竟為什麼有了這一個表我們就可以在非常快的時間查詢出答案了呢？
- 用倍增法二分搜來搜出答案 !!!（我都叫他跳跳二分搜）
- 倍增法二分搜的核心概念是，能跳到多遠的地方就跳到多遠，整個過程就真的是跳來跳去，所以我都叫他跳跳二分搜
- 不過我來詳細拆解一次要怎麼用跳跳二分搜跳出答案來
- 假設要走 5 步，而 5 被轉成二進位的結果是 101，這一個 101 跟 2 的冪次有什麼關係呢？
- $5_{(10)} = 101_{(2)} = 2^0 + 2^2$
- 每一個 k 都能被拆成像這樣的二進位，所以如果現在是要從 12 這一個星球移動 5 次，就等價於從 12 開始，先移動 2^2 次，再移動 2^0 次，而由於我們有預先處理好這樣的表格，所以我們就可以計算出最後會停留的位置

倍增法

- 我們來估計一下時間複雜度吧
- 我們預處理表格必須至少預處理出 2 的 $\log k$ 的冪次的資訊，才足夠應付查詢，而每一個冪次都需要枚舉 n 個起點填入資訊，因此建倍增表的時間複雜度是 $O(n \log k)$

倍增法

- 我們來估計一下時間複雜度吧
- 我們預處理表格必須至少預處理出 2 的 $\log k$ 的冪次的資訊，才足夠應付查詢，而每一個冪次都需要枚舉 n 個起點填入資訊，因此建倍增表的時間複雜度是 $O(n \log k)$
- 查詢時我們最多會跳 $\log k$ 次，而總共有 q 次查詢，因此查詢的時間複雜度會是 $O(q \log k)$
- 整體時間複雜度： $O(n \log k + q \log k)$

參考程式碼

```
25      for(int i=1;i≤31;i++)
26      {
27          for(int j=1;j≤n;j++)
28          {
29              dp[j][i] = dp[dp[j][i-1]][i-1];
30          }
31      }
32
33      while(q--)
34      {
35          int a,b;
36          cin >> a >> b;
37
38          int point = a;
39          for(int i=0;i≤31;i++)
40          {
41              if( ( b & ( 1 << i ) ) ≠ 0 )
42              {
43                  point = dp[point][i];
44              }
45          }
46
47          cout << point << "\n";
```

倍增法

- 倍增法只能做這種跳來跳去的題目嗎？
- 我們來看下一題

倍增法

CSES Problem Set Cyclic Array

給一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列與一個正整數 k $(1 \leq k \leq 10^{18})$，現在你可以將這一個序列分成數個子陣列，每一個子陣列內元素的總和不能超過 k，請問最少只需要分幾個子陣列即可達成題目要求

倍增法

CSES Problem Set Cyclic Array

給一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列與一個正整數 k $(1 \leq k \leq 10^{18})$，現在你可以將這一個序列分成數個子陣列，每一個子陣列內元素的總和不能超過 k，請問最少只需要分幾個子陣列即可達成題目要求

- 看起來跟原本倍增法完全不一樣了對吧，但這題是可以使用倍增法解決的！

倍增法

- 倍增法的題目也很常出現在這種環狀陣列上

倍增法

- 倍增法的題目也很常出現在這種環狀陣列上
- 我們先假設使用一個子陣列就表示跳躍 1 次
- 定義 `dp1[i][k]` 表示從 i 這一個位置開始跳躍 2^k 次，最遠可以跳到的位置（換句話說，在這個位置上時，如果再多跳一個就必須多使用一個子陣列了）

倍增法

- 倍增法的題目也很常出現在這種環狀陣列上
- 我們先假設使用一個子陣列就表示跳躍 1 次
- 定義 dp1[i][k] 表示從 i 這一個位置開始跳躍 2^k 次，最遠可以跳到的位置（換句話說，在這個位置上時，如果再多跳一個就必須多使用一個子陣列了）
- 定義 dp2[i][k] 表示從 i 這個位置往後跳 2^k 次，所有子陣列的元素總和（需要這一個的原因是，如果沒有多維護這一個，單純用第一個 dp 跳的話，有可能會發生跳過頭的情況（多跳），因此多維護一個 dp2 來控制不要跳過頭）

倍增法

- 倍增法的題目也很常出現在這種環狀陣列上
- 我們先假設使用一個子陣列就表示跳躍 1 次
- 定義 dp1[i][k] 表示從 i 這一個位置開始跳躍 2^k 次，最遠可以跳到的位置（換句話說，在這個位置上時，如果再多跳一個就必須多使用一個子陣列了）
- 定義 dp2[i][k] 表示從 i 這個位置往後跳 2^k 次，所有子陣列的元素總和（需要這一個的原因是，如果沒有多維護這一個，單純用第一個 dp 跳的話，有可能會發生跳過頭的情況（多跳），因此多維護一個 dp2 來控制不要跳過頭）
- 倍增法剛開始最需要的東西就是初始狀態，有了這一個我們才能疊出資訊，我們應該怎麼紀錄初始狀態呢？

倍增法

- 倍增法的題目也很常出現在這種環狀陣列上
- 我們先假設使用一個子陣列就表示跳躍 1 次
- 定義 dp1[i][k] 表示從 i 這一個位置開始跳躍 2^k 次，最遠可以跳到的位置（換句話說，在這個位置上時，如果再多跳一個就必須多使用一個子陣列了）
- 定義 dp2[i][k] 表示從 i 這個位置往後跳 2^k 次，所有子陣列的元素總和（需要這一個的原因是，如果沒有多維護這一個，單純用第一個 dp 跳的話，有可能會發生跳過頭的情況（多跳），因此多維護一個 dp2 來控制不要跳過頭）
- 倍增法剛開始最需要的東西就是初始狀態，有了這一個我們才能疊出資訊，我們應該怎麼紀錄初始狀態呢？
- 雙指針（Two Pointers）！就可以找出所有的初始狀態了，記得小心指針移動過頭的問題

倍增法

- 有了初始狀態接下來就來講講轉移式吧！

倍增法

- 有了初始狀態接下來就來講講轉移式吧！
- 如果現在 5 這一個位置最遠可以跳到 12，換句話說 $\sum_{i=5}^{12} a_i \leq k$ 但是 $\sum_{i=5}^{13} a_i > k$
- 那麼表示初始狀態 `dp[5][0] = 12`，那麼如果現在 12 最遠可以往右跳回 2，也就是說初始狀態 `dp[12][0] = 2`

倍增法

- 有了初始狀態接下來就來講講轉移式吧！
- 如果現在 5 這一個位置最遠可以跳到 12，換句話說 $\sum_{i=5}^{13} a_i \leq k$ 但是 $\sum_{i=5}^{12} a_i > k$
- 那麼表示初始狀態 `dp1[5][0] = 12`，那麼如果現在 12 最遠可以往右跳回 2，也就是說初始狀態 `dp1[12][0] = 2`
- 那麼也就表示說如果從 5 出發，使用兩個子陣列（跳 2 次）最遠可以跳回 2，也就意味著 `dp1[5][1] = 2`
- 因為 5 先跳了 1 次到了 12，再從 12 跳了 1 次跳到 2
- 所以我們可以推導出 `dp1[i][k] = dp1[dp1[i][k-1]][k-1]`

倍增法

- 有了初始狀態接下來就來講講轉移式吧！
- 如果現在 5 這一個位置最遠可以跳到 12，換句話說 $\sum_{i=5}^{13} a_i \leq k$ 但是 $\sum_{i=5}^{12} a_i > k$
- 那麼表示初始狀態 `dp1[5][0] = 12`，那麼如果現在 12 最遠可以往右跳回 2，也就是說初始狀態 `dp1[12][0] = 2`
- 那麼也就表示說如果從 5 出發，使用兩個子陣列（跳 2 次）最遠可以跳回 2，也就意味著 `dp1[5][1] = 2`
- 因為 5 先跳了 1 次到了 12，再從 12 跳了 1 次跳到 2
- 所以我們可以推導出 `dp1[i][k] = dp1[dp1[i][k-1]][k-1]`
- 而所有子陣列的總和的轉移式也就會是
- `dp2[i][k] = dp2[i][k-1] + dp2[dp1[i][k-1]][k-1]`

倍增法

- `dp1[i][k] = dp1[dp1[i][k-1]][k-1]`
- `dp2[i][k] = dp2[i][k-1] + dp2[dp1[i][k-1]][k-1]`
- 有了這兩個轉移式我們就可以寫兩層迴圈把會使用到的資訊都透過這兩個轉移式疊出來

倍增法

- `dp1[i][k] = dp1[dp1[i][k-1]][k-1]`
- `dp2[i][k] = dp2[i][k-1] + dp2[dp1[i][k-1]][k-1]`
- 有了這兩個轉移式我們就可以寫兩層迴圈把會使用到的資訊都透過這兩個轉移式疊出來
- 在轉移的時候，要小心 `dp2` 發生 `overflow` 的問題，因為轉移的時候有些情況會跳過頭很多，這樣一直累加就會發生 `overflow`（連 `long long` 都裝不下），因此當 `dp2[i][k]` 的值超過 $10^9 \times 2 \times 10^5$ 時，其實我們就可以都把它紀錄成 $10^9 \times 2 \times 10^5 + 1$ 就好了（一定要比你陣列的所有元素和還要大，不然你在最後找答案的時候他可能會發生都選擇這種跳過頭的當作你的答案）

倍增法

- 最後找答案的時候，我們必須枚舉起點，並對每一個起點所需要的子陣列數量做倍增法二分搜

倍增法

- 最後找答案的時候，我們必須枚舉起點，並對每一個起點所需要的子陣列數量做倍增法二分搜
- 在倍增法二分搜跳答案的時候，務必注意不要跳過頭，當跳的時候如果會發生總和超過原本陣列總合的情況則這一個情況是不能跳的，務必特別小心
- 最後只要盡可能的跳，跳到最後如果發現，跳的總和還沒有跟陣列總和一樣，表示有剩下一些還沒有跳完，針對這種情況最後答案的數量要多加 1

稀疏表 Sparse Table

Sparse Table

- 理念是倍增法的一種資料結構
- 在不帶有修改的情況下時，如果查詢的東西是那種區間重疊計算不影響結果的東西（像是 max,min,gcd）則就可以使用 Sparse Table
- Sparse Table 在建表時需要 $O(n \log n)$ 的時間，但在查詢是驚人的 $O(1)$

區間 min

給定一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列 a，有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每一組查詢將查詢一個區間 $[L_i, R_i]$ 的最小值，請設計一個擁有足夠效率的演算法完成

區間 min

給定一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列 a，有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每一組查詢將查詢一個區間 $[L_i, R_i]$ 的最小值，請設計一個擁有足夠效率的演算法完成

- 會線段樹的學員我知道你們在想什麼，但是先冷靜

Sparse Table

區間 min

給定一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列 a，有 q $(1 \leq q \leq 2 \times 10^5)$ 組查詢，每一組查詢將查詢一個區間 $[L_i, R_i]$ 的最小值，請設計一個擁有足夠效率的演算法完成

- 會線段樹的學員我知道你們在想什麼，但是先冷靜
- 我們來看看 Sparse Table 在這題的發揮吧

Sparse Table

- 定義 dp[i][k] 表示以 i 當作起點，往後延伸共 2^k 個元素的最小值（起點本身也算一個元素，所以 dp[5][0] 指的就是區間 [5,5]，而 dp[5][2] 則是 [5,8]）

Sparse Table

- 定義 `dp[i][k]` 表示以 i 當作起點，往後延伸共 2^k 個元素的最小值（起點本身也算一個元素，所以 `dp[5][0]` 指的就是區間 `[5,5]`，而 `dp[5][2]` 則是 `[5,8]`）
- 接下來我們想想看，如果我們想要算出 `dp[i][k]`，那麼 `dp[i][k]` 的結果會是 $[i, i+2^k-1]$ 這一個區間的最小值，那其實就等價於我們先求區間 $[i, i+2^{k-1}-1]$ 的最小值，再把這一個最小值跟區間 $[i+2^{k-1}, i+2^k-1]$ 的最小值再取一次 `min`，就會是答案

Sparse Table

- 定義 `dp[i][k]` 表示以 i 當作起點，往後延伸共 2^k 個元素的最小值（起點本身也算一個元素，所以 `dp[5][0]` 指的就是區間 `[5,5]`，而 `dp[5][2]` 則是 `[5,8]`）
- 接下來我們想想看，如果我們想要算出 `dp[i][k]`，那麼 `dp[i][k]` 的結果會是 $[i, i+2^k-1]$ 這一個區間的最小值，那其實就等價於我們先求區間 $[i, i+2^{k-1}-1]$ 的最小值，再把這一個最小值跟區間 $[i+2^{k-1}, i+2^k-1]$ 的最小值再取一次 `min`，就會是答案
- 因此就可以列出轉移式 $dp[i][k] = \texttt{min}(dp[i][k-1], dp[i+2^{k-1}][k-1])$

Sparse Table

- 定義 `dp[i][k]` 表示以 i 當作起點，往後延伸共 2^k 個元素的最小值（起點本身也算一個元素，所以 `dp[5][0]` 指的就是區間 `[5,5]`，而 `dp[5][2]` 則是 `[5,8]`）
- 接下來我們想想看，如果我們想要算出 `dp[i][k]`，那麼 `dp[i][k]` 的結果會是 $[i, i+2^k-1]$ 這一個區間的最小值，那其實就等價於我們先求區間 $[i, i+2^{k-1}-1]$ 的最小值，再把這一個最小值跟區間 $[i+2^{k-1}, i+2^k-1]$ 的最小值再取一次 `min`，就會是答案
- 因此就可以列出轉移式 $dp[i][k] = \texttt{min}(dp[i][k-1], dp[i+2^{k-1}][k-1])$
- 記得轉移的時候小心 $i+2^{k-1}$ `overflow` 的問題

Sparse Table

- 有了這一個表之後我們就可以 $O(1)$ 查詢區間最小值了

Sparse Table

- 有了這一個表之後我們就可以 $O(1)$ 查詢區間最小值了
- 假設查詢的區間是 $[L, R]$，那麼我們則必須要先知道 $R - L + 1$（也就是區間長度）的 $log_2 r - l + 1$ 是多少

Sparse Table

- 有了這一個表之後我們就可以 $O(1)$ 查詢區間最小值了
- 假設查詢的區間是 $[L, R]$，那麼我們則必須要先知道 $R - L + 1$（也就是區間長度）的 $log_2 r - l + 1$ 是多少
- 假設 $T = \lfloor log_2 r - l + 1 \rfloor$
- 如此一來我們就可以先取得 $[L, 2^T]$ 的最小值，再透過 $dp[R - 2^T + 1][T]$ 取得 $[R - 2^T + 1, R]$ 的最小值，再將這兩個區間的最小值一起再取一次最小值，就可以得到區間 `[L,R]` 的最小值了
- 即使這兩個區間有些部分會有重疊，但由於我們要取得的資訊是最小值，因此重疊的部分是不會影響答案的

Sparse Table

- 有了這一個表之後我們就可以 $O(1)$ 查詢區間最小值了
- 假設查詢的區間是 $[L, R]$，那麼我們則必須要先知道 $R - L + 1$（也就是區間長度）的 $log_2 r - l + 1$ 是多少
- 假設 $T = \lfloor log_2 r - l + 1 \rfloor$
- 如此一來我們就可以先取得 $[L, 2^T]$ 的最小值，再透過 $dp[R - 2^T + 1][T]$ 取得 $[R - 2^T + 1, R]$ 的最小值，再將這兩個區間的最小值一起再取一次最小值，就可以得到區間 `[L,R]` 的最小值了
- 即使這兩個區間有些部分會有重疊，但由於我們要取得的資訊是最小值，因此重疊的部分是不會影響答案的
- 統整一下，因此如果要查詢一個區間 $[L, R]$ 的最小值，這一個最小值就會是 $\texttt{min}(dp[L][T], dp[R - 2^T + 1][T]$

Sparse Table

- 有了這一個表之後我們就可以 $O(1)$ 查詢區間最小值了
- 假設查詢的區間是 $[L, R]$，那麼我們則必須要先知道 $R - L + 1$（也就是區間長度）的 $log_2(r - l + 1)$ 是多少
- 假設 $T = \lfloor log_2(r - l + 1) \rfloor$
- 如此一來我們就可以先取得 $[L, 2^T]$ 的最小值，再透過 $dp[R - 2^T + 1][T]$ 取得 $[R - 2^T + 1, R]$ 的最小值，再將這兩個區間的最小值一起再取一次最小值，就可以得到區間 `[L,R]` 的最小值了
- 即使這兩個區間有些部分會有重疊，但由於我們要取得的資訊是最小值，因此重疊的部分是不會影響答案的
- 統整一下，因此如果要查詢一個區間 $[L, R]$ 的最小值，這一個最小值就會是 $\texttt{min}(dp[L][T], dp[R - 2^T + 1][T])$
- 而區間 `gcd,max` 的作法也一樣，只是把 `min` 改成你要的東西就可以了！聽起來很棒對吧

參考程式碼

- 雖然圖片裡面沒有出現，但記得在輸入的時候把 `dp[i][0] = a[i]`
- 要有初始狀態才能疊倍增

```cpp
int dp[200005][31];

void sparse_table(int n)
{
    for(int i=1;i≤30;i++)
    {
        for(int k=1;k+(1LL≪(i-1))≤n;k++)
        {
            dp[k][i] = min(dp[k][i-1],dp[k+(1LL≪(i-1))][i-1]);
        }
    }

    return;
}
int query_min(int l,int r)
{
    int idx = __lg( r - l + 1 ); // 這一個東西可以幫助我們知道 log_2(r-l+1) 是多少

    return min( dp[l][idx] , dp[r-(1LL≪idx)+1][idx] );
}
```

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

- 如果 a_i 與 a_{i+1} 我們要找到一個數字 P 滿足 $a_i \equiv a_{i+1} \pmod P$，那我們其中一種選擇就會是 $|a_i - a_{i+1}|$

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

- 如果 a_i 與 a_{i+1} 我們要找到一個數字 P 滿足 $a_i \equiv a_{i+1} \pmod P$，那我們其中一種選擇就會是 $|a_i - a_{i+1}|$
- 為什麼呢？你可以想像，如果現在 $a_i \bmod P$ 的結果是 Q，那麼當你把 $a_i + P$ 或 $a_i - P$ 其實就等同於你直接跳了一次循環

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

- 如果 a_i 與 a_{i+1} 我們要找到一個數字 P 滿足 $a_i \equiv a_{i+1} \pmod P$，那我們其中一種選擇就會是 $|a_i - a_{i+1}|$
- 為什麼呢？你可以想像，如果現在 $a_i \bmod P$ 的結果是 Q，那麼當你把 $a_i + P$ 或 $a_i - P$ 其實就等同於你直接跳了一次循環
- 舉例來說，如果餘數的循環是 {0,1,2,3,4,5,0,1,2,3,4,5...}，那麼當你一次跳 5 步就等同於跳一次完整的循環，所以兩個數字的餘數會相等

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

- 如果 a_i 與 a_{i+1} 我們要找到一個數字 P 滿足 $a_i \equiv a_{i+1} \pmod P$，那我們其中一種選擇就會是 $|a_i - a_{i+1}|$
- 為什麼呢？你可以想像，如果現在 $a_i \bmod P$ 的結果是 Q，那麼當你把 $a_i + P$ 或 $a_i - P$ 其實就等同於你直接跳了一次循環
- 舉例來說，如果餘數的循環是 {0,1,2,3,4,5,0,1,2,3,4,5...}，那麼當你一次跳 5 步就等同於跳一次完整的循環，所以兩個數字的餘數會相等

Sparse Table

Codeforces Round #736 (Div. 1) pB. Integers Have Friends

定義一個區間 $[L, R]$ 內的元素如果滿足 $a_L \bmod m = a_{L+1} \bmod m = ... = a_R \bmod m$ 且 $m \geq 2$ 則稱為一個朋友群，現在給定一組長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請你找到這一個序列中所有朋友群的最大長度

- 如果 a_i 與 a_{i+1} 我們要找到一個數字 P 滿足 $a_i \equiv a_{i+1} \pmod P$，那我們其中一種選擇就會是 $|a_i - a_{i+1}|$
- 為什麼呢？你可以想像，如果現在 $a_i \bmod P$ 的結果是 Q，那麼當你把 $a_i + P$ 或 $a_i - P$ 其實就等同於你直接跳了一次循環
- 舉例來說，如果餘數的循環是 {0,1,2,3,4,5,0,1,2,3,4,5...}，那麼當你一次跳 5 步就等同於跳一次完整的循環，所以兩個數字的餘數會相等

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同
- 但是現在我們的序列長度不一定是 2，有可能很大，那麼這麼長的序列我們要怎麼知道要用哪一個餘數呢？

Sparse Table

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同
- 但是現在我們的序列長度不一定是 2，有可能很大，那麼這麼長的序列我們要怎麼知道要用哪一個餘數呢？
- 想想看，如果現在有元素 {10,88,130}，也就表示 88 - 10 = 78 會讓 10 與 89 同餘、130 - 88 = 42 會讓 80 與 130 同餘，那麼我們選擇哪一個數字會讓 10,88,130 同餘呢？

Sparse Table

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同
- 但是現在我們的序列長度不一定是 2，有可能很大，那麼這麼長的序列我們要怎麼知道要用哪一個餘數呢？
- 想想看，如果現在有元素 {10,88,130}，也就表示 88 - 10 = 78 會讓 10 與 89 同餘、130 - 88 = 42 會讓 80 與 130 同餘，那麼我們選擇哪一個數字會讓 10,88,130 同餘呢？
- 想想看，如果現在可以用 12 這一個餘數，那麼我們可以用 4 這一個餘數嗎？

Sparse Table

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同
- 但是現在我們的序列長度不一定是 2，有可能很大，那麼這麼長的序列我們要怎麼知道要用哪一個餘數呢？
- 想想看，如果現在有元素 {10,88,130}，也就表示 88 - 10 = 78 會讓 10 與 89 同餘、130 - 88 = 42 會讓 80 與 130 同餘，那麼我們選擇哪一個數字會讓 10,88,130 同餘呢？
- 想想看，如果現在可以用 12 這一個餘數，那麼我們可以用 4 這一個餘數嗎？
- 答案是可以的，我們就只是把原本 12 一次循環拆成三等份，變成每 4 個一次循環而已，所以關鍵就是在如果要把大的餘數換成小的，那麼大數字必須是小的數的倍數

Sparse Table

- 也就是說我們必須先知道每個相鄰的兩個數字的差的值，我們才能知道每一個相鄰的元素需要用哪一個數字來讓他們取餘數的結果相同
- 但是現在我們的序列長度不一定是 2，有可能很大，那麼這麼長的序列我們要怎麼知道要用哪一個餘數呢？
- 想想看，如果現在有元素 {10,88,130}，也就表示 88 - 10 = 78 會讓 10 與 89 同餘、130 - 88 = 42 會讓 80 與 130 同餘，那麼我們選擇哪一個數字會讓 10,88,130 同餘呢？
- 想想看，如果現在可以用 12 這一個餘數，那麼我們可以用 4 這一個餘數嗎？
- 答案是可以的，我們就只是把原本 12 一次循環拆成三等份，變成每 4 個一次循環而已，所以關鍵就是在如果要把大的餘數換成小的，那麼大數字必須是小的數的倍數
- 那麼也就是說如果要讓 10,88,130 同餘，我們必須找到其中一個數字 K，這一個 K 要是 78 與 42 的因數（換句話說，48 與 78 要是 K 的倍數）

- 那麼我們要怎麼確定我們找不找的到一個大於 1（題目要求），且是 78 與 42 的因數的數字呢？

Sparse Table

- 那麼我們要怎麼確定我們找不找的到一個大於 1（題目要求），且是 78 與 42 的因數的數字呢？
- 直接看最大公因數是不是 1 就知道我們找不找的到了呀！

Sparse Table

- 那麼我們要怎麼確定我們找不找的到一個大於 1（題目要求），且是 78 與 42 的因數的數字呢？
- 直接看最大公因數是不是 1 就知道我們找不找的到了呀！
- 那現在問題又被我們巧妙的轉換成了一個區間 gcd 的問題
- 我們可以利用雙指針（Two Pointers）來搜出答案，不斷的延伸右指針直到區間 gcd 變成 1 的時候就表示不能再延長了，就更新答案並把左指針移動一個位置枚舉出答案，這一題就完成了！

Policy Based Data Structure (pbds)

pbds

- aka 平板電視（pbds）
- 如果你有聽到有打競程的人在講平板電視，那麼就是在說這一個資料結構
- 這邊我們只會介紹最常見的兩種操作

- 使用前你會需要以下兩個標頭檔

pbds

- 使用前你會需要以下兩個標頭檔
- `ext/pb_ds/assoc_container.hpp`
- `ext/pb_ds/tree_policy.hpp`
- 以上兩個標頭檔可以用下面這個統一取代
- `ext/pb_ds/detail/standard_policies.hpp`
- 但有些電腦可能不能使用這一個標頭檔，是因為在你標頭檔的資料夾中，這一個檔案名稱因為 windows 在搞所以名稱是錯的，要修正你必須自己找到那一個地方把它修成正確的

pbds

- 使用前你會需要以下兩個標頭檔
- `ext/pb_ds/assoc_container.hpp`
- `ext/pb_ds/tree_policy.hpp`
- 以上兩個標頭檔可以用下面這個統一取代
- `ext/pb_ds/detail/standard_policies.hpp`
- 但有些電腦可能不能使用這一個標頭檔，是因為在你標頭檔的資料夾中，這一個檔案名稱因為 `windows` 在搞所以名稱是錯的，要修正你必須自己找到那一個地方把它修成正確的
- 不過懶得修的話就用上面那兩個就好了

以及你會需要 `using namespace __gnu_pbds;` 會讓你接下來平板電視的體驗更加順利

- 使用平板電視之前你必須告訴程式你要用平板電視了，所以跟變數一樣，必須宣告

- 使用平板電視之前你必須告訴程式你要用平板電視了，所以跟變數一樣，必須宣告
- 宣告很簡單，相信大家一定都會

pbds

- 使用平板電視之前你必須告訴程式你要用平板電視了，所以跟變數一樣，必須宣告
- 宣告很簡單，相信大家一定都會

```
tree<Type,null_type,less<Type>,
rb_tree_tag,tree_order_statistics_node_update>
```

- 使用平板電視之前你必須告訴程式你要用平板電視了，所以跟變數一樣，必須宣告
- 宣告很簡單，相信大家一定都會

```
tree<Type,null_type,less<Type>,
rb_tree_tag,tree_order_statistics_node_update>
```

- 看起來很長，如果每一次宣告都要打這一大串就太麻煩了，所以建議你可以自己用 `typedef` 幫他重新命名一次，我會在 `github` 附上範例

- 基本上操作跟 set 一樣，要加入元素用 insert 移除用 erase，只是接下來我們要介紹其他兩個黑魔法

- 基本上操作跟 set 一樣，只是接下來我們要介紹其他兩個黑魔法
- .find_by_order(k) 找到當前所有元素中第 k - 1 小的元素是誰

pbds

- 基本上操作跟 set 一樣，只是接下來我們要介紹其他兩個黑魔法
- .find_by_order(k) 找到當前所有元素中第 k - 1 小的元素的迭代器位置
- .order_of_key(k) 找到當前這裡面的東西有幾個小於 k
- 性質跟 set 一樣，重複元素只會出現 1 次
- 複雜度都是帶一個 log，但常數很大，用太多平板電視小心會出事

第 k 大連續和

給你一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請問這一個序列的第 k 大連續和是多少

第 k 大連續和

給你一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請問這一個序列的第 k 大連續和是多少

- 你說分治講師有教怎麼用分治做嗎？

第 k 大連續和

給你一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請問這一個序列的第 k 大連續和是多少

- 你說分治講師有教怎麼用分治做嗎？
- 小孩子才使用分治，我們來用平板電視

第 k 大連續和

給你一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請問這一個序列的第 k 大連續和是多少

- 你說分治講師有教怎麼用分治做嗎？
- 小孩子才使用分治，我們來用平板電視
- 我們一樣二分搜答案，不知道大家還記不記得，如果我想要知道一個序列有幾個區間比我枚舉的這一個數字還要大，那就表示說假設我當前的前綴和如果是 P，那我就必須要知道前面有幾個前綴和可以拿 P 去減掉他們，而且結果是大於等於 `mid` 的

第 k 大連續和

給你一個長度為 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，請問這一個序列的第 k 大連續和是多少

- 假設當前這一個位置的前綴和為 P，等同於我們要知道前面有幾個前綴和 $\leq P - mid$，就可以知道在這一個位置時，有幾個區間的連續和會 $\leq mid$ 了，那你要找幾個數字 $\leq P - mid$ 就可以輕鬆的用平板電視搞定
- 而第 k 大的連續和至少會有 k 個連續和會 $\geq$ 它，如此一來就可以判斷接下來要往左邊還是右邊找

```
29          int l = -2e9 , r = 2e9;
30
31          while( l ≤ r )
32          {
33              int mid = l + ( r - l ) / 2;
34
35              ordered_set s; // 記得 typedef 那邊要調成 pair
36
37              s.insert({0,0});
38
39              int total = 0;
40
41              for(int i=1;i≤n;i++)
42              {
43                  int num = s.order_of_key( { b[i] - mid + 1 , -1 } );
44                  total += num;
45                  s.insert({b[i],i});
46              }
47
48              if( total ≥ k ) l = mid + 1;
49              else r = mid - 1;
50          }
51
52          cout ≪ r ≪ "\n";
```

BIT（Binary Indexed Tree）樹狀數組

BIT

- 真正的名字其實叫做 Fenwick Tree
- 但大部分的人都叫他 BIT
- 一個輕量級的資料結構，實作量非常少，有些時候可以取代線段樹
- 通常用於維護前綴的資料
- BIT 的內容不會跟大家講詳細的原理，詳細的原理大家可以去參考吳邦一教授寫的一個 BIT 的有趣故事，因為講原理會需要很多的時間 QQ

BIT 是一個用二進位概念維護的資料結構，每一個節點 i 表示的區間會是 $[i - lowbit(i) + 1, i]$ 這一個區間

BIT 是一個用二進位概念維護的資料結構，每一個節點 i 表示的區間會是 $[i - lowbit(i) + 1, i]$ 這一個區間

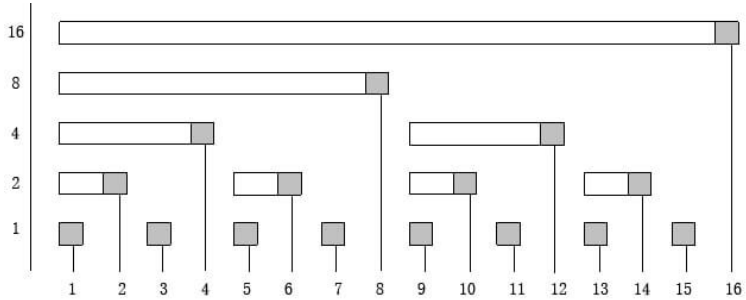
- lowbit 指的是一個數字被轉成二進位之後，從右邊數到左邊第一個是 1 的位置所表示的數字
- 範例：$4_{(10)} = 100_{(2)}$ 的 lowbit 是 4，因為第一個 1 出現在表示 2^2 的那一個位置
- 怎麼取得一個數字的 lowbit 呢？

BIT

BIT 是一個用二進位概念維護的資料結構，每一個節點 i 表示的區間會是 $[i - lowbit(i) + 1, i]$ 這一個區間

- lowbit 指的是一個數字被轉成二進位之後，從右邊數到左邊第一個是 1 的位置所表示的數字
- 範例：$4_{(10)} = 100_{(2)}$ 的 lowbit 是 4，因為第一個 1 出現在表示 2^2 的那一個位置
- 怎麼取得一個數字的 lowbit 呢？
- i 的 lowbit 等價於 $i\& - i$
- 因為 $-i$ 其實就是 i 轉成二進位後把那些二進位 1 轉 0、0 轉 1 之後再 $+1$ 的結果
- 也就表示原本 i 是 0 的位置都被換成了 1，第一個 1 的位置變成了 0，這個時候因為 $+1$ 變成第一個 1 的位置又變回 1 了，但前面都變 0
- 而在第一個 1 的位置左邊的位數因為不受到 $+1$ 的影響，AND 出來的結果皆為 0，唯獨這一個位置的二進位數字是一樣的，所以就可用這樣的方法求出 lowbit

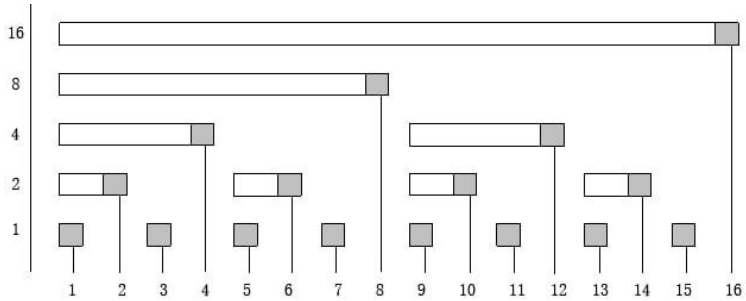
BIT



單點加值、區間查詢

單點加值、區間查詢

- 先講怎麼加值，如果你要在 idx 加上 val，那麼你會需要把在 BIT 上所有跟這一個位置有關的資訊都加上 val，那我們觀察一下 BIT 的長相
- 實際上在 BIT 的結構當中，所有區間會涵蓋到 idx 的節點會是 idx 一直不斷的 + lowbit（在圖上看起來會有一種一直往上跳的感覺）

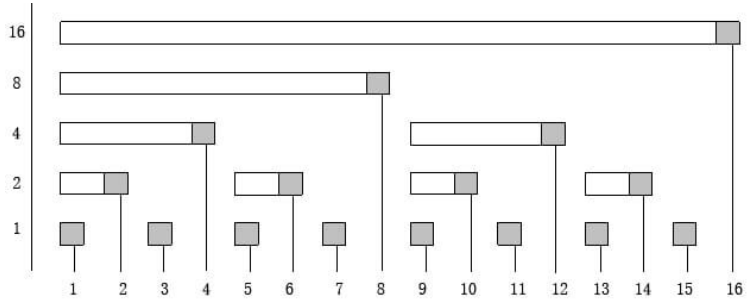


單點加值、區間查詢

```
18 void update_add(int idx,int val)
19 {
20     while( idx ≤ 2e5 )
21     {
22         bit[idx] += val;
23         idx += ( idx & -idx );
24     }
25
26     return;
27 }
```

單點加值、區間查詢

- 那由於 BIT 是一個維護前綴的資料結構，我們要查詢 [1,idx] 的結果的話其實就等同於說我們要找到不重疊且覆蓋 [1,idx] 的所有資訊，那我們觀察一下 BIT 會發現，我們只要一直把 idx 減掉 lowbit 就可以計算出答案了



單點加值、區間查詢

```
28 int query(int idx)
29 {
30     int total = 0;
31     while( idx > 0 )
32     {
33         total += bit[idx];
34         idx -= ( idx & -idx );
35     }
36
37     return total;
38 }
```

綜合以上，一個支援單點加值、區間和查詢的 BIT 就完成了

單點加值、區間查詢

- 那如果我現在想要單點改值有辦法嗎？

單點加值、區間查詢

- 那如果我現在想要單點改值有辦法嗎？
- 可以，如果當前要改的位置的值是 P，那麼你想要把它改成 T，其實就等價於你把這一個位置加值 $T - P$
- 如此一來改值的操作就被我們巧妙地換成加值了呢

單點加值、區間查詢

CSES Problem Set Dynamic Range Sum Queries

給你一組長度 n $(1 \leq n \leq 2 \times 10^5)$ 的序列，接下來最多有 2×10^5 次操作，第一種操作是把位置 P 這個位置的值改成 u，第二種操作是查詢區間 $[L, R]$ 的和

單點加值、區間查詢

```
18      int n,q;
19      cin >> n >> q;
20      vector <int> a(n);
21      for(int i=0;i<n;i++)
22      {
23          cin >> a[i];
24          update(i+1,a[i]); // 把東西更新到 bit 上
25      }
26      while(q--)
27      {
28          int input;
29
30          cin >> input;
31
32          if( input == 1 )
33          {
34              int x,y;
35              cin >> x >> y;
36              int u = query(x) - query(x-1); // 先找到這一個位置的值
37
38              update(x,y-u); // 把改值操作利用差分的概念轉成加值操作
39          }
40          else
41          {
42              int x,y;
43              cin >> x >> y;
44              cout << query(y) - query(x-1) << "\n"; // 前綴和 [L,R] = [1,R] - [1,L-1]
45          }
46      }
```

BIT 與前綴最大值跟前綴最小值

BIT 與前綴最大值跟前綴最小值

- 用相同的實作方式把 + 改成取 max 或 min 在 BIT 就變成了取前綴最大值與最小值了
- 不過在 BIT 取最大值與最小值要特別注意一件事情，先從最大值來說，在單點改值的時候，該點的值要改的數字只能比當前還要更大，否則無法使用 BIT
- 最小值也同理，要改的數字只能比當前還要更小，否則無法使用 BIT

BIT 與前綴最大值跟前綴最小值

- 為什麼會這樣呢？
- 試著想想，如果我們原本 idx 這一個點是 5 且這一個 5 是所有數字當中最小的，而當被我們改成 8 了，那麼更新時因為 5 比 8 小不會更新，可是 5 這個數字已經被我們改掉了，所以前綴最小值不會是 5
- 大家使用 BIT 的時候務必要特別留意這一個問題

區間加值、單點查詢

區間加值、單點查詢

- 想要方便的在 BIT 上使用區間加值就只能付出單點查詢的代價
- 其實是有方法做到區間加值區間查詢，但非常麻煩

區間加值、單點查詢

- 想要方便的在 BIT 上使用區間加值就只能付出單點查詢的代價
- 其實是有方法做到區間加值區間查詢，但非常麻煩
- 區間加值的做法有兩種，一種是差分，另外一種是我不知道該取什麼名字的神奇方法
- 差分的方法比較好理解，這邊就只介紹差分的做法，另外一種作法是利用 BIT 的性質反著做

區間加值、單點查詢

- 差分的想法是，如果我們要在 [L,R] 加上 val
- 那麼就表示在 L 這一個點時，元素的值會多 val，而在 R + 1 這一個點時，元素的值會少 val
- 查詢 idx 的值時，就等同於查詢 [1,idx] 這一個區間的總和，就會是第 idx 這一個位置的值

區間加值、單點查詢

```
17 void update(int idx,int val) // 單點加值
18 {
19     while( idx ≤ 2e5 )
20     {
21         bit[idx] += val;
22         idx += ( idx & -idx );
23     }
24
25     return;
26 }
27 void modify(int l,int r,int val) // 區間加值
28 {
29     update(l,val);
30     update(r+1,-val);
31 }
32 int query(int idx) // 單點查詢
33 {
34     int total = 0;
35     while( idx > 0 )
36     {
37         total += bit[idx];
38         idx -= ( idx & -idx );
39     }
40
41     return total;
42 }
```

BIT 操作的時間複雜度

BIT 操作的時間複雜度

- 更新跟查詢基本上我們都會算 $O(\log n)$
- 不過其實，很多時候 BIT 跳的程度甚至會不到 $\log n$，是一個很恐怖的資料結構

BIT 與初始化

BIT 與初始化

- 使用 BIT 時，大家最常犯的錯誤就是如果要重複使用 BIT 的話會忘記初始化
- 所以在多筆測資的這種題目，大家要重複使用 BIT 前務必先把它初始化（把要用到的索引值初始化成 0）

BIT 解動態規劃問題

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \leq N \leq 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \leq L_i \leq R_i \leq 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \leq w_i \leq 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \leq N \leq 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \leq L_i \leq R_i \leq 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \leq w_i \leq 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

- 我們先定義出轉移式，定義 `dp[i]` 表示在時間點 $1 \sim i$ 的最大快樂度

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \leq N \leq 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \leq L_i \leq R_i \leq 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \leq w_i \leq 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \leq N \leq 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \leq L_i \leq R_i \leq 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \leq w_i \leq 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

- 我們先定義出轉移式，定義 `dp[i]` 表示在時間點 $1 \sim i$ 的最大快樂度
- 對於第 i 場比賽來說，$dp[R_i] = \max(dp_1, ..., dp_{L_i-1}) + w_i$

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \le N \le 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \le L_i \le R_i \le 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \le w_i \le 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

- 我們先定義出轉移式，定義 `dp[i]` 表示在時間點 $1 \sim i$ 的最大快樂度
- 對於第 i 場比賽來說，$dp[R_i] = \max(dp_1, ..., dp_{L_i-1}) + w_i$
- 你有發現什麼了嗎？前面那一段式子就是在求前綴的最大值！所以我們可以用 BIT 來維護
- 除此之外，觀察轉移式會發現，在轉移時我們一定都必須把 L_i 之前的最佳結果計算完，因此我們必須先照左界或右界先排序後再依序轉移

BIT 解動態規劃問題

2021 YTP 決賽第七題 - 壘球大師（20 分）

接下來會有 N $(1 \le N \le 10^6)$ 場比賽，每一場比賽的舉辦日期是 $L_i \sim R_i$ $(1 \le L_i \le R_i \le 10^9)$，在參加第 i 場比賽後將得到 w_i $(1 \le w_i \le 10^9)$ 的快樂度，不過如果有多場比賽舉辦日期有重疊，則只能選擇其中一場比賽參加，也就是說參加了某一場比賽後直到比賽結束之前都不能參加任何比賽，請你設計一個程式找到快樂度的最大值

- 我們先定義出轉移式，定義 `dp[i]` 表示在時間點 $1 \sim i$ 的最大快樂度
- 對於第 i 場比賽來說，$dp[R_i] = \max(dp_1, ..., dp_{L_i-1}) + w_i$
- 你有發現什麼了嗎？前面那一段式子就是在求前綴的最大值！所以我們可以用 BIT 來維護
- 除此之外，觀察轉移式會發現，在轉移時我們一定都必須把 L_i 之前的最佳結果計算完，因此我們必須先照左界或右界先排序後再依序轉移
- 而且由於 L_i, R_i 最大到 10^9，所以我們必須先做離散化，把數值變小，然後就可以使用 BIT 了

BIT 解動態規劃問題

```
49    vector<point> a(n+1);
50    vector <int> b;
51
52    for(int i=1;i≤n;i++)
53    {
54        cin >> a[i].l >> a[i].r >> a[i].p;
55
56        b.push_back(a[i].l);
57        b.push_back(a[i].r);
58    }
59
60    sort(b.begin(),b.end());
61    b.resize( unique(b.begin(),b.end()) - b.begin() );
62    for(int i=1;i≤n;i++)
63    {
64        a[i].l = lower_bound(b.begin(),b.end(),a[i].l) - b.begin() + 1;
65        a[i].r = lower_bound(b.begin(),b.end(),a[i].r) - b.begin() + 1;
66    }
67
68    sort(a.begin(),a.end(),cmp);
```

BIT 解動態規劃問題

```cpp
struct point{
    int l,r,p;
};
bool cmp(point &a,point &b)
{
    return a.l < b.l;
}
void update(int idx,int val)
{
    while( idx ≤ 4e5 )
    {
        bit[idx] = max(bit[idx],val) , idx += ( idx & -idx );
    }
    return;
}
int query(int idx)
{
    int ans = 0;

    while( idx > 0 )
    {
        ans = max(ans,bit[idx]) , idx -= ( idx & -idx );
    }
    return ans;
}
```

BIT 解動態規劃問題

```
69
70      for(int i=1;i≤n;i++)
71      {
72          int num = query(a[i].l-1);
73
74          int num2 = num + a[i].p; // 如果要比這一場比賽的最大快樂度
75
76          update(a[i].r,dp[i]); // 把這一個資訊更新到 a[i].r
77      }
78
79      cout << query(n*2) << "\n"; // 因為離散化後最多會有 2n 個時間點，所以要找 [1,2n] 的最大值
```

最長遞增子序列

給定一組長度 N $(1 \leq N \leq 10^5)$ 序列，請你求出他的最常遞增子序列

最長遞增子序列

給定一組長度 N $(1 \leq N \leq 10^5)$ 序列，請你求出他的最常遞增子序列

- 還記得轉移式嗎？

最長遞增子序列

給定一組長度 N $(1 \leq N \leq 10^5)$ 序列，請你求出他的最常遞增子序列

- 還記得轉移式嗎？
- 定義 $dp[i]$ 表示 `[1,i]` 之間的最長遞增子序列長度

最長遞增子序列

給定一組長度 N $(1 \leq N \leq 10^5)$ 序列，請你求出他的最常遞增子序列

- 還記得轉移式嗎？
- 定義 $dp[i]$ 表示 `[1,i]` 之間的最長遞增子序列長度
- $dp[i] = \texttt{max}(dp[1], dp[2], ..., dp[i-1]) + 1$

最長遞增子序列

給定一組長度 N $(1 \leq N \leq 10^5)$ 序列，請你求出他的最常遞增子序列

- 還記得轉移式嗎？
- 定義 $dp[i]$ 表示 [1,i] 之間的最長遞增子序列長度
- $dp[i] = \max(dp[1], dp[2], ..., dp[i-1]) + 1$
- 發現了嗎，又是前綴 max !!! 所以我們就可以用 BIT 解 ><

BIT 解分治問題

逆序數對

給定一個長度為 n $(1 \leq 2 \times 10^5)$ 的序列，請你找出有幾組 (i, j) $(1 \leq i < j \leq n)$ 滿足 $a_i > a_j$

BIT 解分治問題

逆序數對

給定一個長度為 n $(1 \leq 2 \times 10^5)$ 的序列，請你找出有幾組 (i, j) $(1 \leq i < j \leq n)$ 滿足 $a_i > a_j$

- 對於每一個位置 i，我們要計算在 i 的左邊有幾個數字會跟 i 形成逆序數對
- 其實這就等同於，我們對值域來開 BIT，把每一個位置的值標記在 BIT 上（如果值是 20，就在 BIT 20 的位置上 +1）

BIT 解分治問題

逆序數對

給定一個長度為 n $(1 \leq 2 \times 10^5)$ 的序列，請你找出有幾組 (i, j) $(1 \leq i < j \leq n)$ 滿足 $a_i > a_j$

- 對於每一個位置 i，我們要計算在 i 的左邊有幾個數字會跟 i 形成逆序數對
- 其實這就等同於，我們對值域來開 BIT，把每一個位置的值標記在 BIT 上（如果值是 20，就在 BIT 20 的位置上 +1）
- 如果現在 $i = 12$，那麼現在問題就被轉換成我們要知道前面有幾個數字比 a_i 還要大
- 比 a_i 還要大的數字數量 = 當前的數字數量 - 比 a_i 還要小或相等的數字數量
- 算完 i 的答案之後，就把 BIT 上 a_i 這一個位置 $+1$，重複以上步驟就可以不用分治算出逆序數對了

逆序數對

給定一個長度為 n $(1 \leq 2 \times 10^5)$ 的序列，請你找出有幾組 (i, j) $(1 \leq i < j \leq n)$ 滿足 $a_i > a_j$

- 不過這邊其實可以做一些小優化，如果我們從 n 做回 1，並且對於 i 我們去找 i 的右邊有幾個數字可以跟他變成逆序數對的話，就等同於我們要知道右邊有幾個數字比 a_i 還要小
- 而且由於我們是倒著做回來，所以我們只需要使用一次查詢就可以算出數量了，整體來說會比原本的兩次查詢還要稍微好一點

BIT 上二分搜與 BIT 上倍增法二分搜

帶有更新的第 k 大的數字

第 k 大的數字

請你在帶有更新的操作下，針對每一筆查詢找到當前序列中第 k 大的數字

帶有更新的第 k 大的數字

第 k 大的數字

請你在帶有更新的操作下，針對每一筆查詢找到當前序列中第 k 大的數字

- 我們照值域，把東西打在 BIT 上
- $\{10, 20, 20, 30, 40\}$ 就等同於我們把 10 這個位置 $+1$、20 這個位置加兩次 1 以此類推 . . .

帶有更新的第 k 大的數字

第 k 大的數字

請你在帶有更新的操作下，針對每一筆查詢找到當前序列中第 k 大的數字

- 我們照值域，把東西打在 BIT 上
- $\{10, 20, 20, 30, 40\}$ 就等同於我們把 10 這個位置 $+1$、20 這個位置加兩次 1 以此類推 ...
- 接下來我們去想想，第 k 大的數字在這一個序列會有什麼樣的特質
- 特質就是，在一個長度為 n 的序列中，數字 $\geq$ 這一個數字的數量一定至少會有 k 個甚至更多（因為數字可能會有重複）

帶有更新的第 k 大的數字

第 k 大的數字

請你在帶有更新的操作下，針對每一筆查詢找到當前序列中第 k 大的數字

- 我們照值域，把東西打在 BIT 上
- $\{10, 20, 20, 30, 40\}$ 就等同於我們把 10 這個位置 $+1$、20 這個位置加兩次 1 以此類推 ...
- 接下來我們去想想，第 k 大的數字在這一個序列會有什麼樣的特質
- 特質就是，在一個長度為 n 的序列中，數字 $\geq$ 這一個數字的數量一定至少會有 k 個甚至更多（因為數字可能會有重複）
- 接下來我們可以做一件事情叫做對答案二分搜，去檢查有幾個數字 $\geq mid$，如果數量超過 k 個，你就可以非常確定第 k 大的數字一定 $\geq mid$，否則，就表示 mid 太大了，往小一點的答案去找

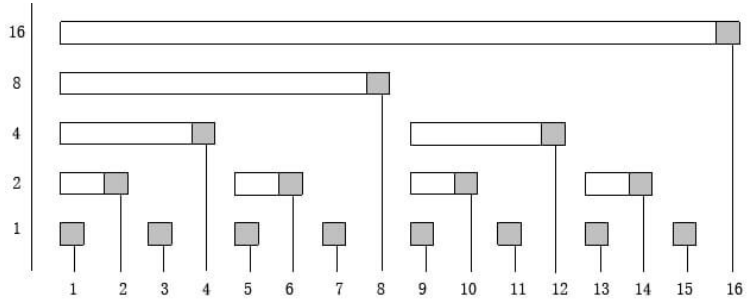
帶有更新的第 k 大的數字

```cpp
int binary_search(int k) // 找到第 k 大的數字
{
    int l = 0 , r = 2e5; // 如果數字很大或是有負數，必須先離散化，最大到 2e5 只是一個假設

    while( l ≤ r )
    {
        int mid = ( l + r ) / 2;
        int num = n - query(mid-1); // 全部 - 比 mid 還小的數字數量 = ≥ mid 的數字個數

        if( num ≥ k ) l = mid + 1;
        else r = mid - 1;
    }

    return r;
}
```

查詢的時間複雜度會是 $O(\log^2 n)$

BIT 上倍增法二分搜

在講 BIT 上倍增法二分搜之前，我們觀察看看 BIT 的結構
你會發現，它的結構就是把每一段長度 2^n 再細切成更多的部分，BIT 上的每一個節點的長度都是 2 的冪次
所以在 BIT 上二分搜時我們可以使用倍增法二分搜，核心的概念也是一樣的
可以跳多遠就先跳多遠



BIT 上倍增法二分搜

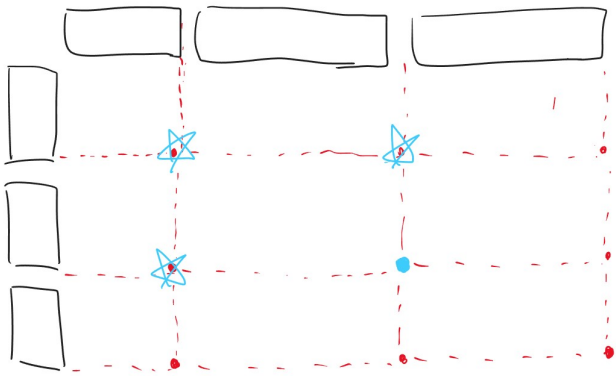
```cpp
int binary_search(int k) // 找到第 k 大的數字
{
    int idx = 0 , answer = 0 , total = 0;

    for(int i=30;i≥0;i--)
    {
        if( (1LL << i) ≤ 2e5 )
        {
            if( total + bit[idx+(1LL<<i)] < n - k + 1 ) // 至少會有 n - k + 1 個數字 ≤ 第 k 大的數字
            {
                total += bit[idx+(1LL<<i)];
                answer += ( 1LL << i );
                idx += ( 1LL << i );
            }
        }
    }

    // 為什麼我們只跳到答案的前一個，為的是避免跳超過

    return answer + 1;
}
```

而這樣子的倍增法二分搜會從原本一般二分搜的 $O(\log^2 n)$ 優化成 $O(\log n)$，原因是倍增法二分搜不需要依賴 BIT 的查詢功能

二維 BIT

二維 BIT

- 可以用來求從原點到某個點圍出來的矩形上所有元素的和，寫法跟一般的 BIT 一樣，只是變成了二維的狀態



二維 BIT

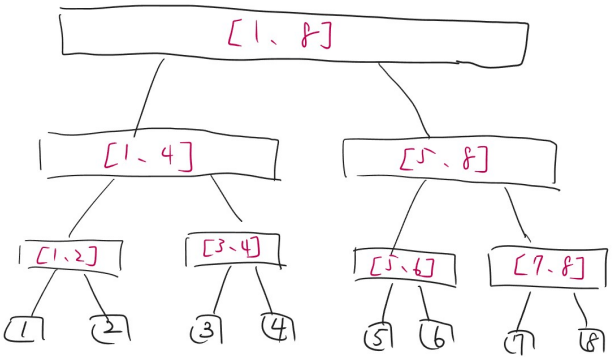
```
 9 void update(int x,int y,int val)
10 {
11     while( x ≤ 1000 )
12     {
13         int idx = y;
14 
15         while( idx ≤ 1000 )
16         {
17             bit[x][idx] += val;
18             idx += ( idx & -idx );
19         }
20 
21         x += ( x & -x );
22     }
23 }
24 int query(int x,int y)
25 {
26     int total = 0;
27     while( x > 0 )
28     {
29         int idx = y;
30         while( idx > 0 )
31         {
32             total += bit[x][idx];
33 
34             idx -= ( idx & -idx );
35         }
36 
37         x -= ( x & -x );
38     }
39 
40     return total;
41 }
```

線段樹 Segment Tree

線段樹 Segment Tree

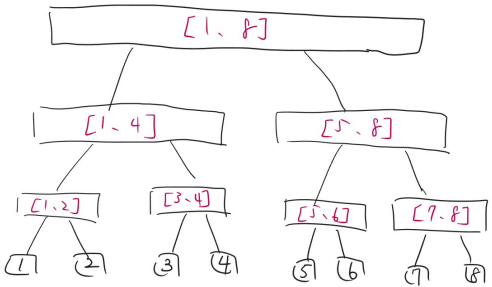
- 一個大家都多少有聽過的一個資料結構
- 其理念是分治
- 實作量大，但可以幫我們完成超級多事情
- 可以做到區間改值/加值、單點改值/加值、區間查詢、單點查詢

線段樹 Segment Tree



build

- 分治的概念，如果我們要算出當前 [L,R] 的答案，那麼我們必須先知道 [L,MID] 跟 [MID+1,R] 的結果，並將這兩個結點的結果合併
- 因此實作上會使用遞迴，當我們要取得當前區間的答案，就往左右去遞迴取得左右子區間的答案
- 通常我們在線段樹的結點編號，根結點的編號會是 1，左半區間的編號會是父結點的編號乘上 2，而右半區間的結點編號會是父結點的編號乘上 2 再加 1
- 時間複雜度 $O(n \log n)$

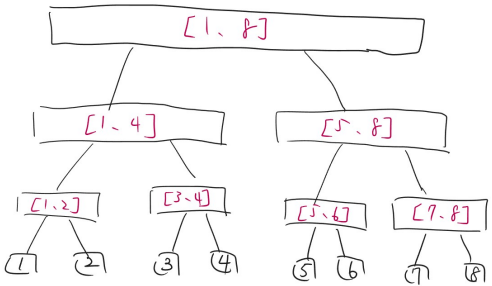


build

```cpp
int seg[200005*4]; // 要開大約 4 倍的結點
int a[200005];

void build(int index,int l,int r)
{
    if( l == r ) // 已經遞迴到底了，停止遞迴，回去合併
    {
        seg[index] = a[l];
        return;
    }

    int mid = ( l + r ) / 2;

    build(index*2,l,mid); // 取得左半區間的答案
    build(index*2+1,mid+1,r); // 取得右半區間的答案

    seg[index] = seg[index*2] + seg[index*2+1]; // 合併

    return;
}
```

update

- 如果我們要做單點更新
- 我們就必須要遞迴到表示那一個點的葉節點去
- 因此每一次單點更新時間複雜度都是 $O(\log n)$

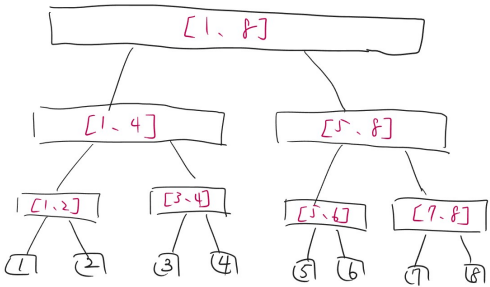


update

```
22 void update(int pos,int val,int index,int l,int r) // 將 pos 這個位置改成 val
23 {
24     if( l == r )
25     {
26         seg[index] = val; // 找到 pos 的位置了，更新
27         return;
28     }
29
30     int mid = ( l + r ) / 2;
31
32     if( r ≤ mid ) update(pos,val,index*2,l,mid); // 要更新的點在左邊，往左遞迴
33     else update(pos,val,index*2+1,mid+1,r);       // 要更新的點在右邊，往右遞迴
34
35     seg[index] = seg[index*2] + seg[index*2+1];
36     // 因為更新節點的資訊，要重新合併最新資訊
37 }
```

query

- 區間查詢 [L,R] 時，如果發現當前線段樹的這一個節點已經被包含在查詢範圍內，就直接回傳該節點的資訊
- 否則如果要查詢的區間全部位於左半部，就往左半部遞迴、全部都位於右半部就往右半部遞迴、介於左邊與右邊之間則必須兩邊都遞迴
- 時間複雜度為 $O(\log n)$



query

```
38 int query(int ql,int qr,int index,int l,int r) // 查詢 [ql,qr] 的總和
39 {
40     if( ql ≤ l && r ≤ qr ) return seg[index]; // 這一個節點的資訊完全被包含在查詢的區間內
41
42     int mid = ( l + r ) / 2;
43
44     if( qr ≤ mid ) return query(ql,qr,index*2,l,mid); // 要找的區間都在左邊
45     else if( ql > mid ) return query(ql,qr,index*2+1,mid+1,r); // 要找的區間都在右邊
46     else return query(ql,qr,index*2,l,mid) + query(ql,qr,index*2+1,mid+1,r); // 介於兩邊
47 }
```

線段樹 Segment Tree

- 在合併的部分自己把 + 改成取 max 或 min 就自然變成一個區間求最大值或最小值的線段樹了
- 而要取最大公因數也是可以的，一樣就是把合併的部分改成求 gcd
- 現在你會種出一棵最基本的線段樹了，好耶好耶

區間改值/加值的懶惰標記

區間改值與加值的懶惰標記

- 可能有些人已經想到，如果要做區間改值，就跟查詢很像，遞迴到每一個葉子去
- 不過區間改值與加值不像查詢一樣，可以在發現這一個區間的資訊都會被使用到就直接回傳該節點的資訊結束遞迴
- 而是必須每一個都硬生生的跑到葉節點去，如此一來時間複雜度甚至會比暴力修改還要差 . . .

區間改值與加值的懶惰標記

- 可能有些人已經想到，如果要做區間改值，就跟查詢很像，遞迴到每一個葉子去
- 不過區間改值與加值不像查詢一樣，可以在發現這一個區間的資訊都會被使用到就直接回傳該節點的資訊結束遞迴
- 而是必須每一個都硬生生的跑到葉節點去，如此一來時間複雜度甚至會比暴力修改還要差 . . .
- 所以我們需要一個東西，叫做懶惰標記 ! ! !
- 我們來看看什麼是懶惰標記吧！

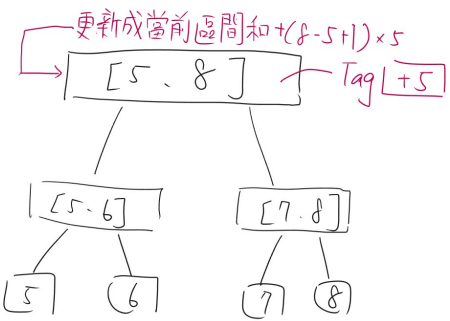
懶惰標記

- 懶惰標記的想法是，假設我們現在想要做區間加值，那麼假設現在這一個線段樹的節點是被要加值的區間完全包含的話，那其實我們可以直接預期到這一個節點會變成什麼（會變成當前總和加上區間大小 $\times$ 要加值的數字）
- 因此我們就直接把這一個節點的值改成當前總和加上區間大小 $\times$ 要加值的數字，並額外紀錄這一個節點上打上一個加值的標記，接著就直接結束遞迴
- 如此一來你會發現，遞迴的終止條件就跟查詢一模一樣了
- 不過原本打在那一個節點的標記是要做什麼的呢，為什麼這樣可以讓我們不用往下更新呢？

懶惰標記

- 假設現在我在查詢時用到帶有標記的這一個節點，那就表示我接下來非常有可能會使用到這一個節點的左右子樹，這一個節點的左右子樹也分別表示其完整個一個區間
- 之前打上了標記是表示那一個區間的元素都要被加上一個值，但現在我們只把父節點給更新成加值後的結果
- 那接下來我要用到他的左子節點跟右子節點了，那我就一併跟查詢的遞迴一起把標記往下推下去
- 這樣子跟查詢的遞迴一起連帶做之前的區間加值的想法就非常的懶惰，等同於說我們既然要更新這一個區間那就乾脆等到我真的要用到他的時候我再實際把它更新，如此一來複雜度就會是好的

懶惰標記



[5、8] +5

懶惰標記

```cpp
int seg[200005*4],tag[200005*4];

void push_tag(int index,int l,int r)
{
    if( tag[index] =/= 0 )
    {
        int mid = ( l + r ) / 2;

        seg[index*2] += ( mid - l + 1 ) * tag[index]; // 更新左半結果
        seg[index*2+1] += ( r - mid ) * tag[index]; // 更新右半的結果

        tag[index*2] = tag[index*2+1] = tag[index]; // 標記下推

        tag[index] = 0; // 原本這一個點的標記已經下推完成，把它初始化
    }

    return;
}
```

懶惰標記

```
20 void modify(int ql,int qr,int val,int index,int l,int r)
21 {
22     if( l ≠ r ) push_tag(index,l,r);
23
24     if( ql ≤ l && r ≤ qr )
25     {
26         seg[index] += ( r - l + 1 ) * val;
27         tag[index] += val;
28
29         return;
30     }
31
32     int mid = ( l + r ) / 2;
33
34     if( qr ≤ mid ) modify(ql,qr,val,index*2,l,mid);
35     else if( ql > mid ) modify(ql,qr,val,index*2+1,r);
36     else
37     {
38         modify(ql,qr,val,index*2,l,mid);
39         modify(ql,qr,val,index*2+1,r);
40     }
41
42     seg[index] = seg[index*2] + seg[index*2+1];
43 }
```

懶惰標記

```
int query(int ql,int qr,int index,int l,int r)
{
    if( l ≠ r ) push_tag(index,l,r);

    int mid = ( l + r ) / 2;

    if( ql ≤ l && r ≤ qr ) return seg[index];
    else if( qr ≤ mid ) return query(ql,qr,index*2,l,mid);
    else if( ql > mid ) return query(ql,qr,index*2+1,mid+1,r);
    else return query(ql,qr,val,index*2,l,mid) + query(ql,qr,index*2+1,mid+1,r);
}
```

懶惰標記

- 不過要特別注意的是，合併的東西必須滿足結合律的性質才適用懶惰標記
- 除了滿足結合律之外，修改的東西還必須是可預測性的東西才可以，否則你會不知道這一個區間被修改後會變成什麼樣子

線段樹上不太一樣的區間合併

線段樹上不太一樣的區間合併

區間最大連續和（帶修改）

多筆查詢與修改元素，查詢每一次修改完後當前序列的區間最大連續和

線段樹上不太一樣的區間合併

區間最大連續和 （帶修改）

多筆查詢與修改元素，查詢每一次修改完後當前序列的區間最大連續和

- 大家還記得分治的做法嗎，其實 . . .　就只是把分治的做法套用到線段樹上來哦 . . .
- 我們一起來看看吧！

線段樹上不太一樣的區間合併

- 還記得分治的精神嗎？要求一整個好難哦，那我先求一部分的，把每一部分的東西都算完再合併出答案

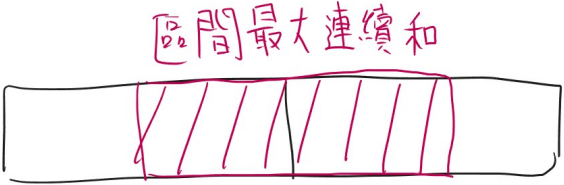
線段樹上不太一樣的區間合併

- 還記得分治的精神嗎？要求一整個好難哦，那我先求一部分的，把每一部分的東西都算完再合併出答案
- 我們先到底層算出每一個區間的以下東西
- 區間的前綴最大連續和
- 區間的後綴最大連續和
- 區間的最大連續和
- 區間的總和

線段樹上不太一樣的區間合併

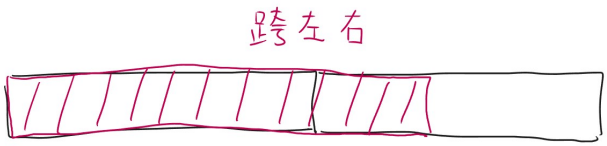
- 接下來我們要做的事情是維護好我們的這些資訊
- 在我們合併一個大區間的時候，這一個大區間的區間最大連續和的答案要怎麼合併出來呢？
- 假設左半部區間的最大連續和是 a
- 假設右半部區間的最大連續和是 b
- 假設左半部的後綴最大連續和是 c
- 假設右半部的前綴最大連序和是 d
- 答案就會是 $\max(a, b, c + d)$
- 因為這一個大區間的最大連續和只有三種可能
- 一種是落在左半部
- 一種是落在右半部
- 最後一種情況則是落在左半部也落在右半部

答案跨越左右的例子



線段樹上不太一樣的區間合併

- 那麼一個大區間的前綴最大連續和就只有兩種情況了
- 第一種是前綴最大連續和就是原本左半部的前綴最大連續和
- 第二種則是前綴最大連續和跨越到了右半部，因此前綴最大連序和變成了左半部的總和 + 右半部的前綴最大連序和



線段樹上不太一樣的區間合併

- 那麼大區間的後綴最大連續和就以此類推
- 一樣是兩種情況
- 一種是後綴最大連續和依舊是右半部的後綴最大連序和
- 另外一種則是橫跨兩半部，右半部的總和 + 左半部的後綴最大連續和

線段樹上不太一樣的區間合併

- 綜合以上，你會發現我們知道了要怎麼從小區間合併出大區間，而且線段樹能幫助我們支援修改，如此一來就可以利用線段樹來維護區間最大連續和了！

參考程式碼

```cpp
struct node{
    int ans, left, right, sum,mx;
};

node seg[200005*4];
int a[200005];

node combine(node a, node b){ // 如果自己寫一個合併的 function 的話，程式碼會比較乾淨
    node c;
    c.ans = max({a.ans, b.ans, a.right+b.left});
    c.left = max(a.sum+b.left, a.left);
    c.right = max(a.right+b.sum, b.right);
    c.sum = a.sum + b.sum;

    return c;
}

// 下面都跟一般線段樹一樣，只是把合併的部分丟到上面去處理
```