基礎資料結構

yp155136

February 5, 2023

講義勘誤

- 第 19 頁範例 code 第 7 行「路徑壓縮」這個註解劃掉（我打錯了）
- 第 28 頁「實作」區第二段第三行「來找出每個數字是第幾大」中的「第幾大」改成「第幾小」
- 第 43 頁程式碼片段 1-14 的標題應該是「線段樹：區間加值、區間最大值」
- 第 60 頁習題 1-39 的題號應該是 TIOJ 1976
- 第 73 的程式碼片段 1-31
  - 第 17, 18 行中間插入 `t->rc->fa = t;`
  - 第 14, 15 行中間插入 `t->lc->fa = t;`

道歉啟示

- 範例 code 我會盡快整理好後放在 github 上的

講師簡介

- yp155136
- LYB / BBCube
  - ICPC 2021 台北站第二名
- IOI 2018 國際資訊奧林匹亞銀牌
- APIO 2018 亞太資訊奧林匹亞金牌

講師簡介

- yp155136
- LYB / BBCube
  - ICPC 2021 台北站第二名
- IOI 2018 國際資訊奧林匹亞銀牌
- APIO 2018 亞太資訊奧林匹亞金牌
- 因為想學習基礎資料結構所以當基礎資料結構講師

大綱

- 並查集
- 好用的 STL
- Sparse Table
- 基礎線段樹
- BIT
- 進階線段樹
- 樹堆

前言

- 這堂課會教大家在競程上常用的資料結構
- 學會、精熟這些資料結構，可以提昇在競程上的實力喔

並查集

並查集

- 支援合併及查詢集合的資料結構
- 具體來說，需要支援：
  1. init(n)：初始 n 個元素，各自屬於一個獨立的集合。
  2. union(a, b)：將 a 元素所在的集合以及 b 元素所在的集合合併。
  3. find(a)：詢問 a 所在的集合編號。
- 維護的集合兩兩互斥
- 來畫圖解釋吧！

並查集 – 實作方法

- 將每個集合以樹狀結構來表示
- 整個並查集會形成一個森林，而每棵樹分別代表一個集合
- 每個元素記錄自身的父節點（程式碼中的 p），而每棵樹便以根節點（$p[i] = i$ 的點）作爲代表
- 在做集合相關操作的時候，**必須要用代表點（也就是根節點）**來操作

並查集 – 範例實作

```
1  struct DisjointSet {
2    int p[maxn];
3    void init(int n) {
4      for (int i = 1; i <= n; i++)
5        p[i] = i;
6    }
7    int find(int x) { return x == p[x] ? x : find(p[x]); }
8    void join(int x, int y) { p[find(x)] = find(y); }
9  };
```

並查集 – 時間複雜度

- `init()`：就是個 $O(N)$
- `find(x)`：看起來好像很快，但其實 worst case 會到 $O(N)$ (樹是一條鏈的狀況)
- `join(x, y)`：就是個 $O(1)$
- 想想看，`find(x)` 真的需要 $O(N)$ 那麼多嗎？

並查集 – 範例實作

- `find()` 事實上就是在樹上不斷往父節點爬
- 把查詢路徑上所有人的父節點設爲根節點！
- 關心的只是每個節點屬於哪個集合，樹狀結構長什麼樣子並不重要
- `return x == p[x] ? x : p[x] = find(p[x]);`
- 經過這個改善後，複雜度**均攤**下來是 $O(N \log N)$
- **路徑壓縮**

並查集 – 範例實作

- 那 `join()` 呢？有沒有比較厲害的 join 方式可以降低複雜度？
- **union by rank**
- 維護每棵樹當前的深度
- 每次要合併時，都將深度小的接到深度大的上
- 於是，這樣樹的深度就不會超過 $O(\log n)$，`find()` 在沒有路徑壓縮下複雜度也是好的

並查集 – 範例實作

```
1  void join(int x, int y) {
2    x = find(x), y = find(y);
3    if (x == y)
4      return;
5    if (rk[x] < rk[y]) // union by rank
6      swap(x, y);
7    p[y] = x;
8    if (rk[x] == rk[y])
9      ++rk[x];
10 }
```

並查集 – 實作總結

- 什麼都不做會退化到 $O(N)$
- union by rank / 路徑壓縮只用一個的話可以讓複雜度變 $O(\log N)$
- 同場加碼：union by rank 跟路徑壓縮一起做的話，均攤時間複雜度爲 $O(\alpha(n))$，幾乎可以視為 $O(1)$！
- 同場加碼：union by size （小樹接到大樹）跟 union by rank 的效果是一樣的喔！

並查集 – 例題

題目 (經典問題（這題講義上沒有）)

給你一張 N 個點的圖 G，圖一開始沒有任何邊，以及 M 筆操作。
操作有三種：

- 加邊：給 a, b，把點 a 跟點 b 中間加上一條邊
- 詢問一：給 a, b，問 a, b 是否在同一個連通塊
- 詢問二：詢問目前的圖有多少個連通塊

- $N, M \leq 10^5$

並查集

- 並查集很常用的地方：維護圖的連通性
- 並查集的集合**維護連通塊**
- 一開始圖是空的，相當於所有點都在自己的連通塊裡面，對應到 `init(N)`！
- 有一條邊新增進來的時候，相當於把兩個連通塊變成一個連通塊
- 也相當於是並查集的 `join()` 操作！

並查集

- 詢問兩點是否連通？
- 判斷兩個點是不是在同一個集合
- `djs.find(a) == djs.find(b)`
- 詢問連通塊個數？
- 一邊 `join()` 一邊維護
- 每當 `join()` 兩個不同的集合時，連通塊個數 `-= 1`。

並查集 – 例題

題目 (Ural 1671 Anansis Cobweb)

給一張無向圖 G，N 個點、M 條邊。Q 筆操作，每筆操作破壞一條邊後問連通塊個數。

- $1 \leq N \leq 10^5$
- $1 \leq M \leq 10^5$
- $1 \leq Q \leq M$

並查集

- 但，這題是破壞邊，不是增加邊，怎麼辦？

並查集

- **時間倒流**
- 沒有被破壞的邊表示從頭到尾都沒被破壞，可以在此刻維護一個並查集
- 破壞一條邊，就相當於建造一條邊！

並查集 – 例題

題目 (POJ 1182 食物鏈)

有三類動物 A, B, C，這三類動物的食物鏈構成如下：A 吃 B，B 吃 C，C 吃 A。現有 N 個動物，編號 $1, 2, \ldots, N$。每個動物都是 A, B, C 中的一種，但並不知道是哪一種。依序給 K 條屬於以下兩種的敘述：

- 1 X Y，表示 X 和 Y 是同類
- 2 X Y，表示 X 吃 Y

然而，並不是每條描述都是正確的，有些是真話，有些是假話。如果當前的話與前面的某些真話衝突，就是假話
請判斷哪些話是真話，哪些話是假話。

並查集

- 並查集的經典變化！
- 第一種資訊，顯然可以用並查集維護
- 假設 a, b 同類，可能可以類似 `djs.join(a, b)`
- 但，第二種資訊怎麼辦？好像沒辦法直接維護ㄋㄟ

並查集

- 那就直接假設每種動物在每個分類底下的狀況！
- 把每個動物 i 分成三個資訊：動物 i 屬於第 A 類、動物 i 屬於第 B 類、動物 i 屬於第 C 類
- 假設 i_O 代表動物 i 是第 O 類的資訊
- 有了這個可以做什麼 OuO

並查集

- 我們可以用並查集維護**資訊間的因果關係**
- 一個集合裡面的資訊代表「這些資訊必須同時發生」
- 比如說，假設 (x, y) 符合第一種資訊
- 那就代表：如果 x 是 A 類，那 y 必須是 A 類，反之亦然（B, C 類同樣可以套用）
- 翻譯一下：如果 x_A 發生的話，那 y_A 也一定要發生（B, C 類同樣可以套用）
- $join(x_A, y_A), join(x_B, y_B), join(x_C, y_C)$

並查集

- 我們可以用並查集維護**資訊間的因果關係**
- 一個集合裡面的資訊代表「這些資訊必須同時發生」
- 比如說，假設 (x, y) 符合第二種資訊
- 那就代表：
  - 如果 x 是 A 類，那 y 必須是 B 類，反之亦然
  - 如果 x 是 B 類，那 y 必須是 C 類，反之亦然
  - 如果 x 是 C 類，那 y 必須是 A 類，反之亦然
- $join(x_A, y_B), join(x_B, y_C), join(x_C, y_A)$

並查集

- 我們可以用並查集維護**資訊間的因果關係**
- 一個集合裡面的資訊代表「這些資訊必須同時發生」
- 判斷真假？
- 一種判斷方法：如果 join 後發生 $find(x_A) == find(x_B)$，就代表這中間一定有假話！

並查集 – 例題

題目 (經典問題)

給一張無向圖 G，N 個點、M 條邊。
(版本一) 請你判斷這張圖是不是二分圖？
(版本二) 請你判斷這張圖存不存在長度是奇數的簡單環？

這題是上個例題的簡化版，也是一個並查集非常實用的地方ㄛ！

並查集 – 總結

- 並查集是一個維護集合的工具
- 常用在圖論上維護圖的連通性
- 有蠻酷的變形
- 有時候會跟時間線段樹搭配，歡迎參考進階資料結構的講義！

好用的 STL

好用的 STL

- STL 裡面有許多實用的東西
- 這個章節會選一些介紹，剩下的就麻煩大家自行上網搜尋ㄛ

好用的 STL – bitset 例題

題目 (NEOJ 743 數字總和)

給你 N 個數字 $a_1, a_2, \ldots, a_N$，並且滿足 $\sum_{i=1}^{N} a_i \leq M$。
現在，對於每個數字 x ，請判斷 x 可不可由 $a_1, a_2, \ldots, a_N$ 給湊出來。也就是說，能不能找到一個集合 S ，使得 $\sum_{i \in S} a_i = x$。

- $N \leq 2 \times 10^5$
- $M \leq 2 \times 10^5$

好用的 STL – bitset

- 我會 DP！

```
1  dp[0] = 1;
2  for (int i = 1; i <= n; ++i) {
3    std::cin >> a[i];
4    for (int x = m; x >= a[i]; --x) {
5      dp[x] |= dp[x - a[i]];
6    }
7  }
```

- 複雜度是 $O(NM)$，TLE
- bitset 該出場了！

好用的 STL – bitset

- 可以想像成一個大 bool 陣列
- 可以直接對這個 bool 陣列做位元操作（and, or, 左移等等）
- 可是等等，上面那個轉移式可以被寫成兩個 bool 陣列做位元操作的形式嗎？

好用的 STL – 範例實作

```cpp
bool dp_tmp[kN] = {false};
for (int x = a[i]; x <= m; ++x) { // step 1
  dp_tmp[x] = dp[x - a[i]];
}
for (int x = 0; x <= m; ++ x) { // step 2
  dp[x] = dp[x] | dp_tmp[x];
}
```

- step 1 其實就是把 dp 整個陣列往右移 $a[i]$ 格
- step 2 其實就是 dp 跟 dp_tmp 做 and 操作

好用的 STL – 範例實作

```cpp
std::bitset<kN> dp; // 宣告一個大小為 kN 的 bitset
dp[0] = 1; // dp[0] 可以直接存取 bitset 的第 0 個 bit，也可以直接寫值
for (int i = 1; i <= n; ++i) {
  std::cin >> a[i];
  dp |= (dp << a[i]); // dp = dp | (dp << a[i]);
}
```

好用的 STL – bitset

- 複雜度瞬間變成 $O(\frac{NM}{w})$，AC
- w 要看跑程式機器的 word 大小，通常不是 32 就是 64
- 官解複雜度有除 32 或 64 通常代表有用到 bitset
- bitset 的更多用法歡迎參考講義喔！

好用的 STL – 離散化

- 詳細的定義可以參考講義
- 白話文：把數字重新從 1 開始按照大小填
- $[5, -6, 3, 5, 7]$ 變成 $[3, 1, 2, 3, 4]$
- 通常離散化後的數字都會是整數
- 常見用途：把 $[1, 10^9]$ 的數字變成 $[1, N]$ 後，就可以開一個大小是 $O(N)$ 的陣列做事

好用的 STL – idea 1

- 把那 N 個數字丟進 std::map 裡面
- 再依序把數字在 map 中對應到的 value 依序填上 $[1, N]$
- 最後利用 map 把原本的數字替換成新的數字

```
1  std::map<int, int> mp;
2  for (int i : a) mp[i] = 0;
3  int idx = 0;
4  for (auto p : mp) mp[p.first] = ++idx;
5  for (int &i : a) i = mp[i];
```

- std::map 常數太大了，實作上並不是一個有效率的作法

好用的 STL – idea 1

- 把所有數字由小到大排序好
- 利用二分搜尋法，來找出每個數字是第幾小
- 比如說，$[5, -6, 3, 5, 7]$ 的 5 是第三小，就把 5 對應到 3

好用的 STL – 範例實作

```cpp
std::vector<int> v;
for (int i = 1; i <= n; ++i) {
  v.emplace_back(arr[i]);
}
std::sort(v.begin(), v.end());
for (int i = 1; i <= n; ++i) {
  arr[i] = std::lower_bound(v.begin(), v.end(), arr[i])
  ↪  - v.begin() + 1;
}
```

`std::sort` 可以用 $O(N \log N)$ 的時間來排序那 N 個數字

`std::lower_bound` 可以在 $O(\log N)$ 的時間內找出第一個不小於目標數字的位置

好用的 STL – 範例實作

```cpp
std::vector<int> v;
for (int i = 1; i <= n; ++i) {
  v.emplace_back(arr[i]);
}
std::sort(v.begin(), v.end());
v.resize(std::unique(v.begin(), v.end()) -
  ↪  v.begin());
for (int i = 1; i <= n; ++i) {
  arr[i] = std::lower_bound(v.begin(), v.end(),
    ↪  arr[i]) - v.begin() + 1;
}
```

- `std::unique` 會把所有不同的元素搬到容器的前面，並且回傳最後一個不同元素後面的位置
- 拿這個回傳值跟容器的開頭相減，就可以得到總共有多少相異的元素（resize 那行在做的事情）

好用的 STL – 離散化

- `std::sort` 可以用 $O(N \log N)$ 的時間來排序那 N 個數字
- `std::lower_bound` 可以在 $O(\log N)$ 的時間內找出第一個不小於目標數字的位置
- `std::unique` 會把所有不同的元素搬到容器的前面，並且回傳最後一個不同元素後面的位置，複雜度為 $O(N)$

好用的 STL – pbds

- 平板電視、黑魔法
- 有現成的 hash table、heap、平衡樹
- 平衡樹比 `std::set` 還強大，可以支援搜尋一個數字的 order （大小）
- 但其實不會 pbds 對競程影響不會那麼大 (?)
- 有興趣請參考講義 + 網路資料！

好用的 STL – rope

- 簡化版持久化線段樹
- 有興趣請參考講義 + 網路資料！

Sparse Table

Sparse Table

- $\langle O(f(N)), O(g(N))\rangle$ 來代表某個資料結構需要 $O(f(N))$ 預處理，並可以 $O(g(N))$ 回答某一個詢問
- 他是一個 $\langle O(N \log N), O(1)\rangle$ 處理靜態區間最大值的資料結構
- 好抽象，先來看例題！

Sparse Table – 例題

題目 (區間最大值 Range Maximum Query，RMQ)

給定一個長度為 N 的正整數序列 $a_1, \ldots, a_N$，接著有 Q 個詢問，每個詢問形如

- `1 l r`，請你回答在 $a_l, a_{l+1}, \ldots, a_r$ 當中的最大值。
- $N, Q \leq 5 \times 10^5$

靜態：序列的數值不會變動。相對應的「動態區間最大值」會在下一個章節講喔！

Sparse Table – 例題

- 我會！

```
1  int ret = 0;
2  for (int i = l; i <= r; ++i) ret = max(ret, a[i]);
3  cout << ret << '\n';
```

- 試著用 $\langle O(f(N)), O(g(N))\rangle$ 來表達看看上面演算法的複雜度

Sparse Table – 例題

- 我會！

```
1  int ret = 0;
2  for (int i = l; i <= r; ++i) ret = max(ret, a[i]);
3  cout << ret << '\n';
```

- $\langle O(1), O(N) \rangle$，TLE！

Sparse Table – 例題

- 我會！

```
1  int ans[kN][kN];
2  for (int i = 1; i <= n; ++i) {
3    ans[i][i] = a[i];
4    for (int j = i + 1; j <= n; ++j) ans[i][j] =
     ↪  max(ans[i][j - 1], a[j]);
5  }
6  cout << ans[l][r] << '\n';
```

- 試著用 $\langle O(f(N)), O(g(N))\rangle$ 來表達看看上面演算法的複雜度

Sparse Table – 例題

- 我會！

```
1  int ans[kN][kN];
2  for (int i = 1; i <= n; ++i) {
3    ans[i][i] = a[i];
4    for (int j = i + 1; j <= n; ++j) ans[i][j] =
     ↪  max(ans[i][j - 1], a[j]);
5  }
6  cout << ans[l][r] << '\n';
```

- $\langle O(N^2), O(1) \rangle$，TLE！

Sparse Table – 例題

- 兩種方法各有利弊：
  - 第一個方法的好處是預處理超快，壞處是詢問慢到哭
  - 第二個方法的好處是詢問超快，壞處是預處理慢到哭
- 有沒有辦法在預處理跟詢問達到平衡呢？
- 可以！這就是 Sparse Table 厲害的地方

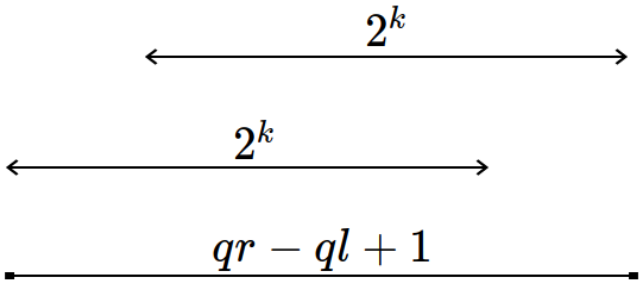
Sparse Table

- Sparse Table 希望在用相對少的預處理，來達到盡量快的詢問
- 暴雷一：詢問可以 $O(1)$
- 暴雷二：預處理是處理連續區間的最大值
- 既然是 $O(1)$ 回答詢問，就代表 $[ql, qr]$ 的答案需要用 $O(1)$ 個預處理資料回答
- 如果只用一個預處理資料的話，會變成上面 $\langle O(N^2), O(1) \rangle$
- 那如果用兩個呢？

Sparse Table

- 想法一：用 $[ql, mid]$ 跟 $[mid + 1, qr]$ 組出答案
- $mid = (ql + qr)/1$，也就是切一半的意思
- 可是這樣還是要預處理長度為 1 到 $\frac{N}{2}$ 的連續子序列的答案，複雜度還是 $O(N^2)$
- 怎麼辦？
- 靈光一閃！

Sparse Table



- 回答一個詢問 $[ql, qr]$ 最大值時，我們可以找到最大的 k 使得 $2^k \leq qr - ql + 1$
- 那麼 $[ql, qr]$ 正好可以用兩個長度 2^k 的區間的聯集湊出來！
- 也就是說，把序列每個位置做為開頭，長度是 $1, 2, 4, 8$ 等等 2 的冪次長度的子序列的最大值存起來，就可以在 $O(1)$ 的時間回答一個詢問了

Sparse Table

- 把序列每個位置做為開頭，長度是 $1, 2, 4, 8$ 等等 2 的冪次長度的子序列的最大值存起來
- 看起來我們預處理的資訊有 $O(N) \times O(\log N) = O(N \log N)$ 個
- 令 $f(i, j)$ 表示區間 $[i, i + 2^j)$ 的最大值
- 也就是說，$f(i, j)$ 是從 i 開始往後 2^j 這個連續序列的最大值

Sparse Table

- 把序列每個位置做為開頭，長度是 $1, 2, 4, 8$ 等等 2 的冪次長度的子序列的最大值存起來
- 看起來我們預處理的資訊有 $O(N) \times O(\log N) = O(N \log N)$ 個
- 令 $f(i, j)$ 表示區間 $[i, i + 2^j)$ 的最大值
- 也就是說，$f(i, j)$ 是從 i 開始往後 2^j 這個連續序列的最大值
- 要怎麼更新呢？

Spare Table

```
int f[kN][klogN];
for (int i = 0; i < kN; ++i) {
  for (int j = 0; j < klogN; ++j) {
    f[i][j] = a[i];
    for (int k = i; k < min(i + (1 << j), n); ++k) {
      f[i][j] = max(f[i][j], a[k]);
    }
  }
}
```

- 每個預處理資料都用 $O(N)$ 算答案，複雜度為 $O(N^2 \log N)$，TLE
- 有辦法**重複利用已經預處理好的資料**嗎？

Sparse Table

- 來試試看吧！
- 令 $f(i,j)$ 表示區間 $[i, i+2^j)$ 的最大值
- $j=0$ 好像沒啥辦法，就是個 $f(i,0)=a_i$
- $j=1$ 呢？好像就是 $f(i,1)=\max(a_i,a_{i+1})$，還看不太出來
- $j=2$ 呢？$f(i,2)=\max(\{a_i,a_{i+1},a_{i+2},a_{i+3}\})$
- 試著對半切切看呢？
  $f(i,2)=\max(\max(a_i,a_{i+1}),\max(a_{i+2,i+3}))$
- 咦，$\max(a_i,a_{i+1})$ 好像是 $f(i,1)$ 耶，$\max(a_{i+2},a_{i+3})$ 好像是 $f(i+2,1)$
- 長度是 4 可以被拆成 $2+2$！

Sparse Table

- 令 $f(i,j)$ 表示區間 $[i, i+2^j)$ 的最大值
- 所以其實，$j=1$ 的 $f(i,1)=\max(a_i, a_{i+1})$ 也可以寫成 $f(i,1)=\max(f(i,0), f(i+1,0))$
- 所以，假設我們已經有所有長度是 1 的答案了（就是 a 序列本身）
- 我們就會有長度是 2 的答案（每個長度是 2 的答案都可以由 $1+1$ 組出來）
- 我們就會有長度是 4 的答案（每個長度是 4 的答案都可以由 $2+2$ 組出來）

Sparse Table

- 令 $f(i,j)$ 表示區間 $[i, i+2^j)$ 的最大值
- 所以，比較大方向的來說，長度是 2^j 的區間可以由兩個長度對半（2^{j-1}）區間的答案湊出來
- 第一段的開頭是 i，第二段的開頭是 $i+2^{j-1}$
- $f(i,j) = \max(f(i,j-1), f(i+2^{j-1}, j-1))$
- 這個更新的式子是 $O(1)$ 的！也就是說，更新一個預處理資料為 $O(1)$
- 預處理的複雜度為 $O(N \log N) \times O(1) = O(N \log N)$
- $\langle O(N \log N), O(1) \rangle$

Sparse Table – 範例實作

```
1  for (int i = 0; i < n; i++)
2    mx[0][i] = arr[i];
3  for (int lg = 0; lg + 1 < maxlg; lg++) {
4    int len = 1 << lg;
5    for (int i = 0; i + len < n; i++) // 注意 i + len 不要超
     ↪  出邊界
6      mx[lg + 1][i] = std::max(mx[lg][i], mx[lg][i +
       ↪  len]);
7  }
```

Sparse Table – 範例實作

```cpp
int query(int l, int r) { // returns max([l, r])
  int lg = std::__lg(r - l + 1);
  int len = 1 << lg;
  return std::max(mx[lg][l], mx[lg][r - 1 - len]);
}
```

`std::__lg()` 這個函數會回傳該數字以 2 為底的對數的整數部份，速度比正常的 `log2()` 快上許多

Sparse Table – 總結

- $O(N \log N)$ 預處理，$O(1)$ 查詢
- **不能支援修改**
- 可以用在**重複取不會對答案造成影響的詢問**，比如說 gcd、bitwise-and/or
- 詢問很多的話可以派上用場！
- 思考：為什麼要用 2^n 劃分，用 3^n 不好嗎？

基礎線段樹

基礎線段樹 – 例題

題目 (區間最大值 Range Maximum Query，RMQ)

給定一個長度為 N 的正整數序列 $a_1, \ldots, a_N$，接著有 Q 個詢問，每個詢問形如

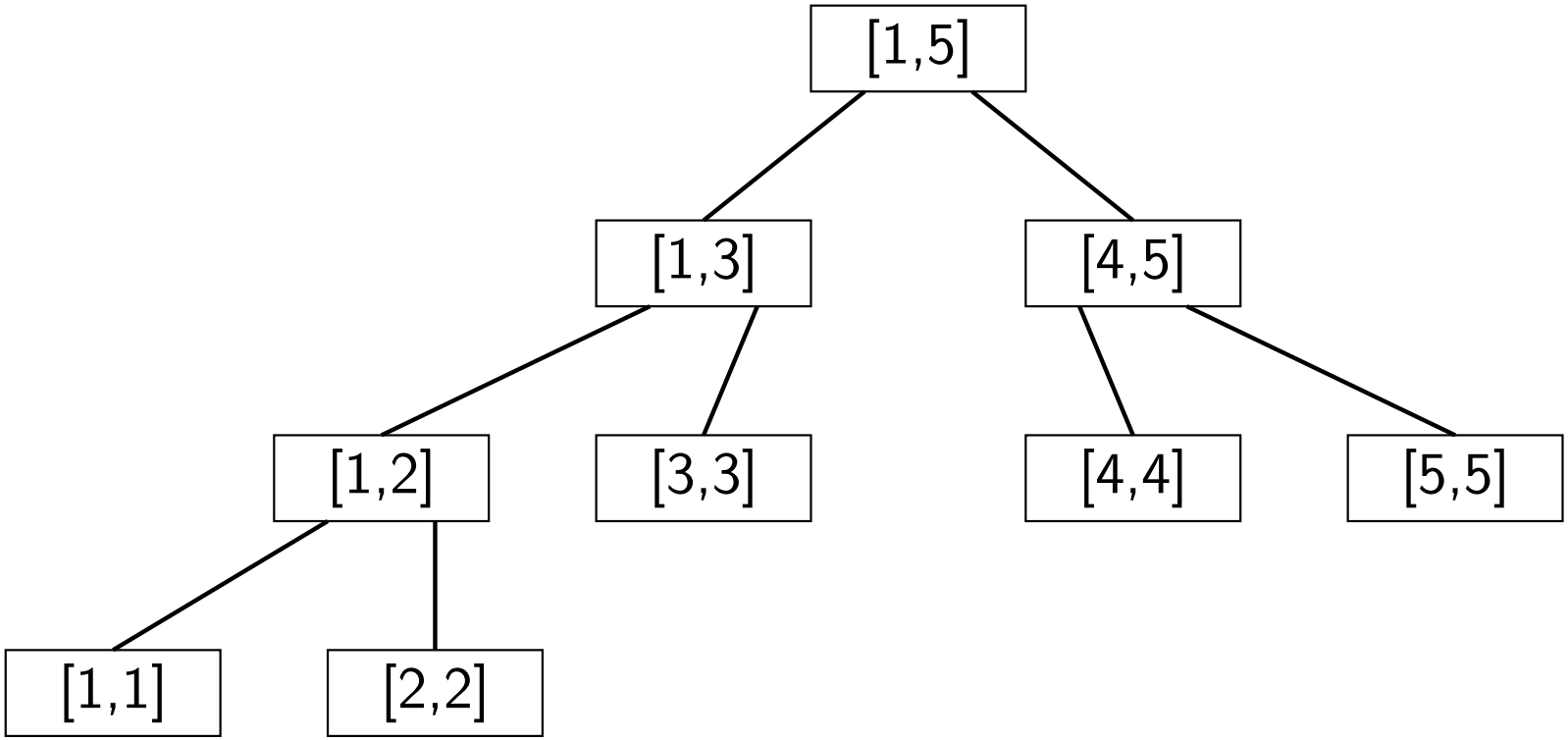
- `1 l r`，請你回答在 $a_l, a_{l+1}, \ldots, a_r$ 當中的最大值。
- `2 p x`，請你把 a_p 修改為 x。
- $N, Q \leq 5 \times 10^5$

基礎線段樹 – 例題

- 從前面的經驗來看，我們好像要對序列做一些事情，才能解決這題
- 前面介紹的 Sparse Table 已經派不上用場了
  - 因為修改一個元素會動到將近 $O(N)$ 個預處理表格的內容
- 但 Sparse Table 的精神可以給我們一些啟發：
  - 答案是從某些預處理資料組合出來的
  - 預處理資料盡量可以由本身小資料算出大資料的答案
  - 大資料底下的小資料大小大概都是大資料的一半
- 除了 Sparse Table 精神外，我們還要支援：
  - 修改一個元素的時候，讓包含這個元素的資料盡量少

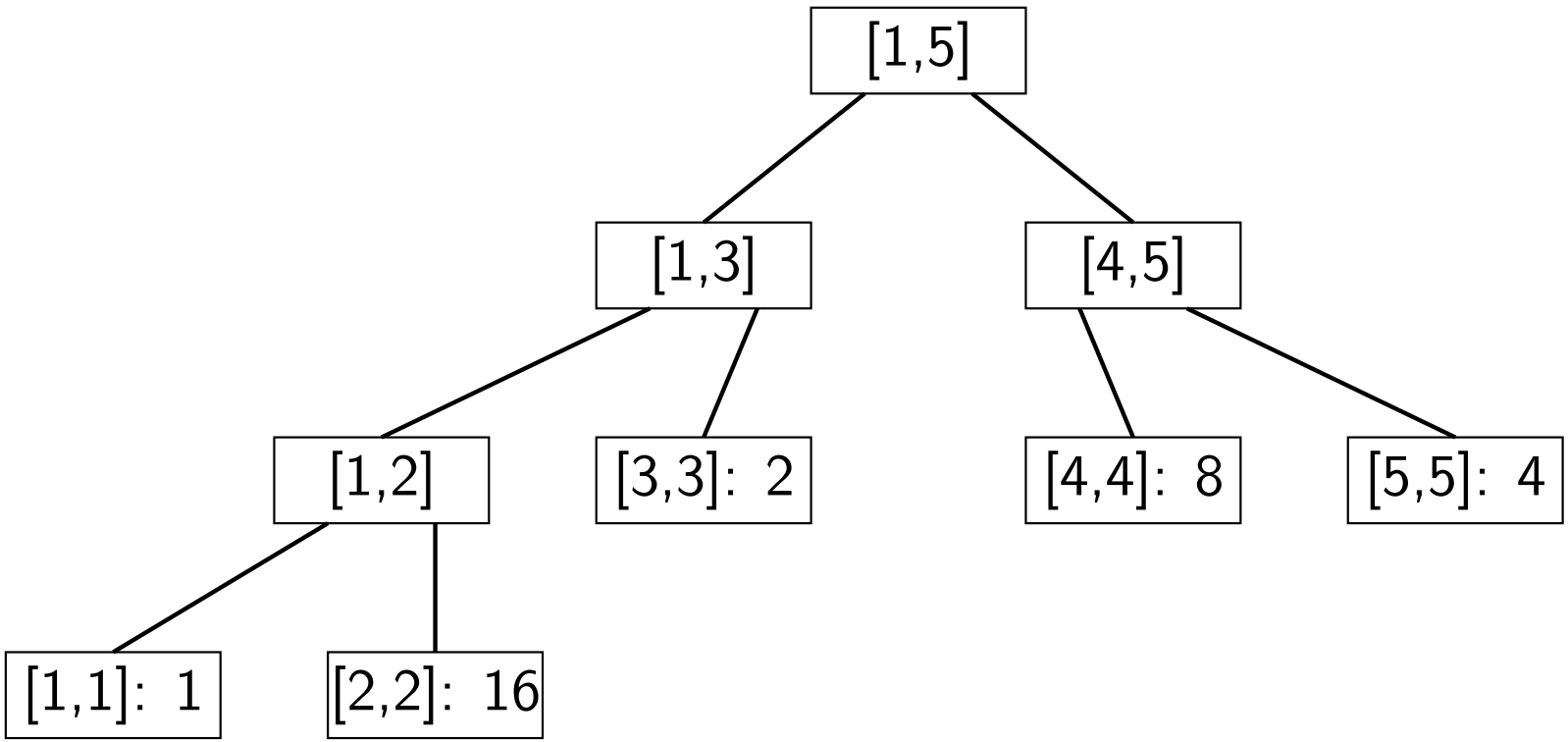
基礎線段樹 – 例題

- **大資料底下的小資料大小大概都是大資料的一半**
- 不妨試試看分治吧！來看一個長度為 5 的序列的分治過程



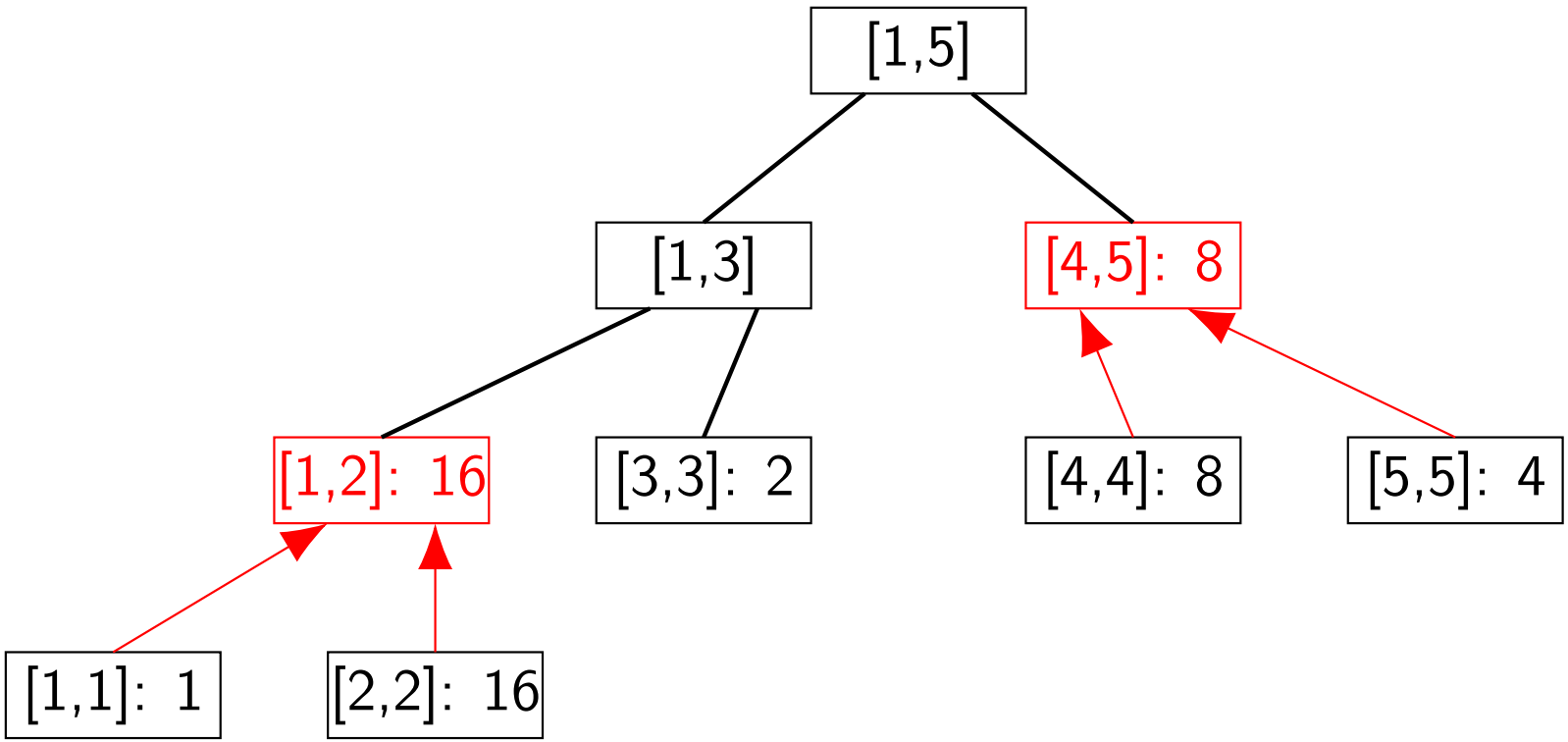
基礎線段樹

- 把這個結構存成樹，並假設每個節點都存該區間的最大值。以下假設 $a = [1, 16, 2, 8, 4]$



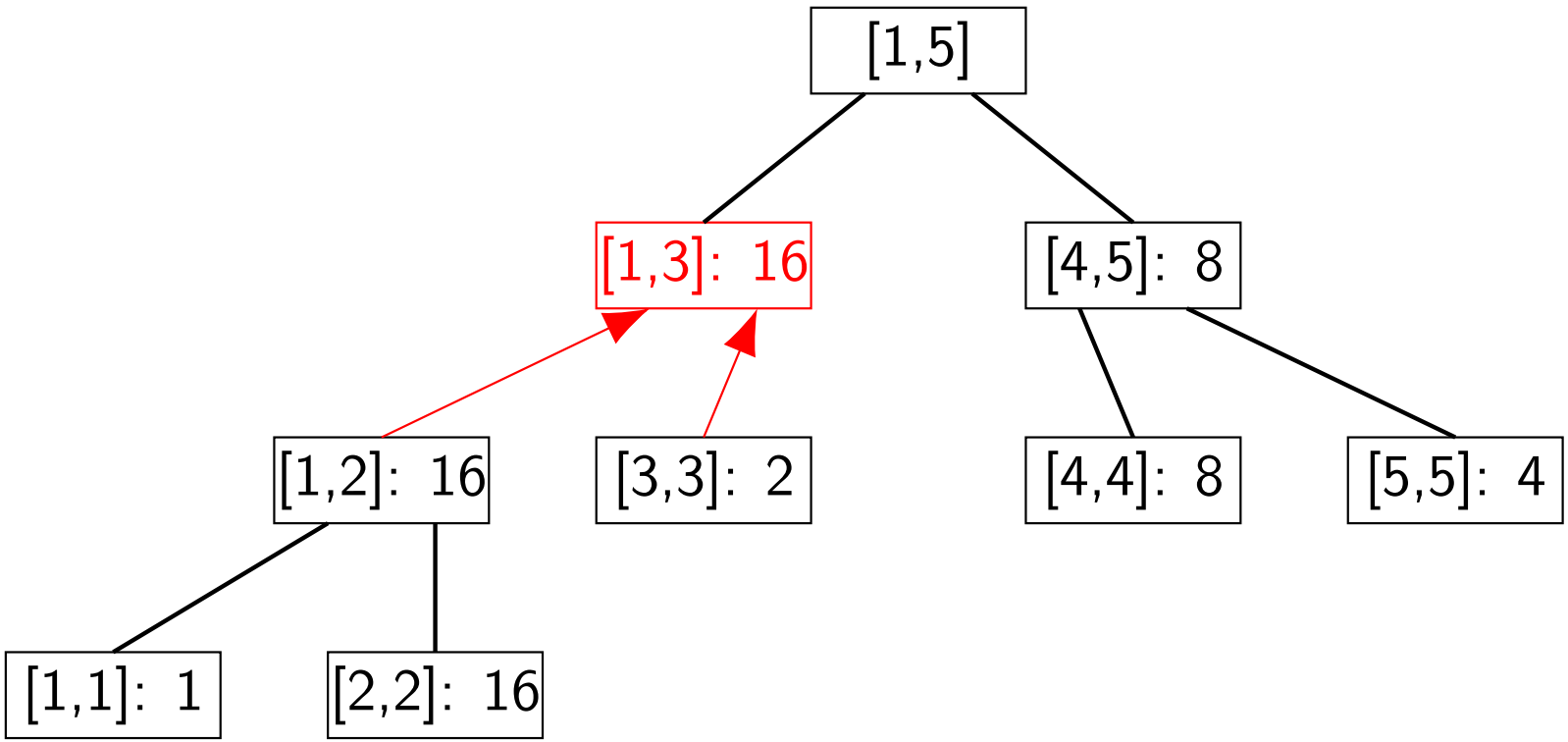
基礎線段樹

- 把這個結構存成樹，並假設每個節點都存該區間的最大值。以下假設 $a = [1, 16, 2, 8, 4]$



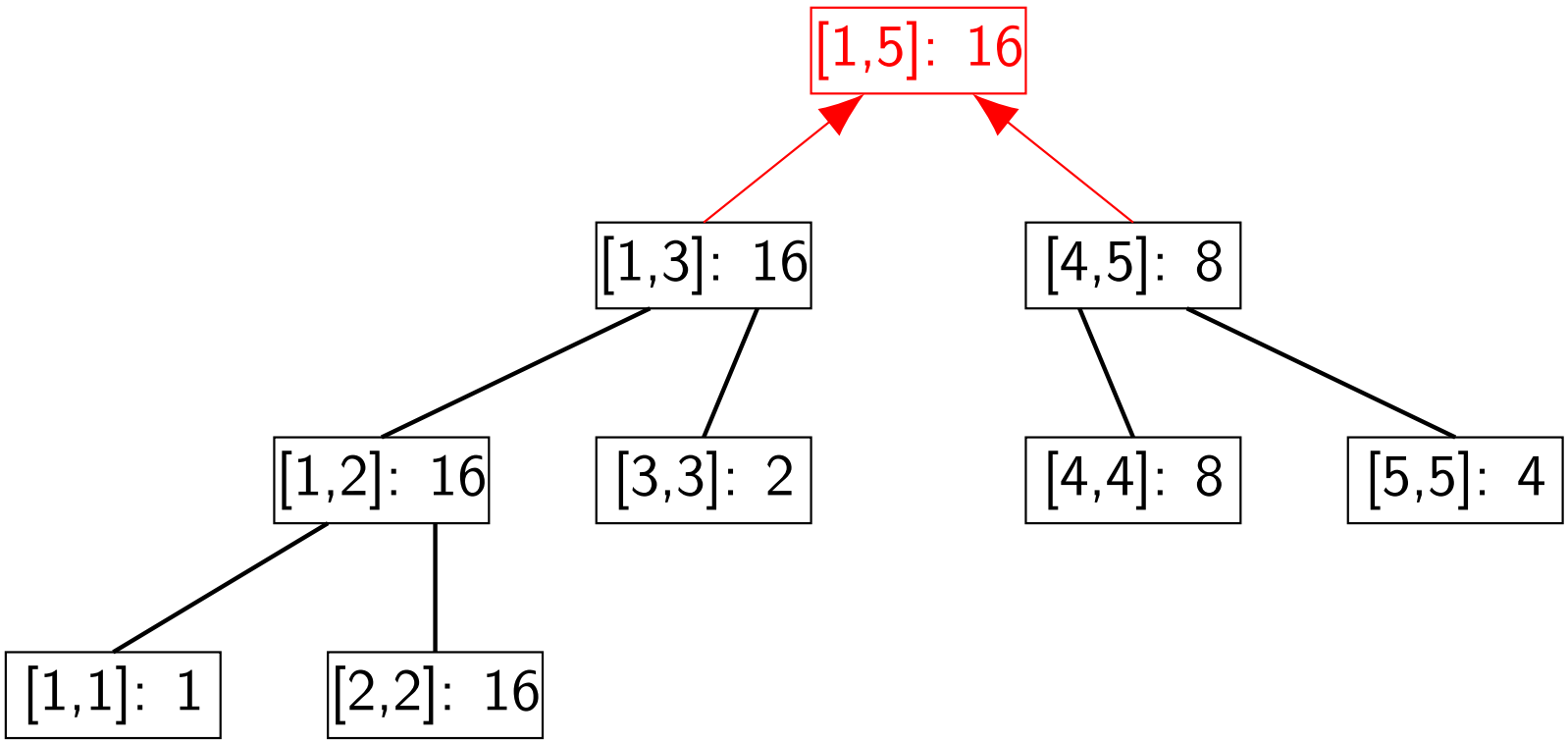
基礎線段樹

- 把這個結構存成樹，並假設每個節點都存該區間的最大值。以下假設 $a = [1, 16, 2, 8, 4]$



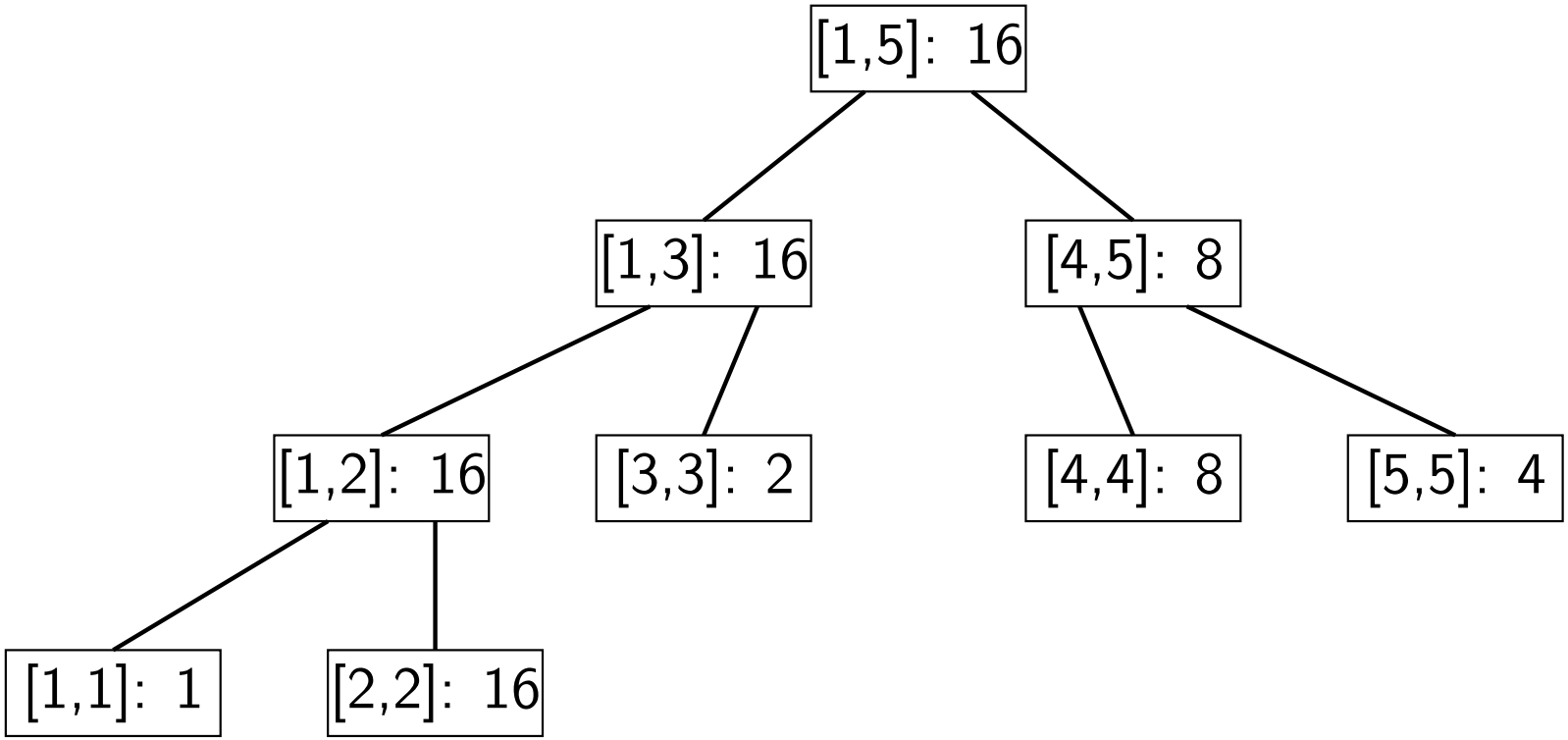
基礎線段樹

- 把這個結構存成樹，並假設每個節點都存該區間的最大值。以下假設 $a = [1, 16, 2, 8, 4]$



基礎線段樹

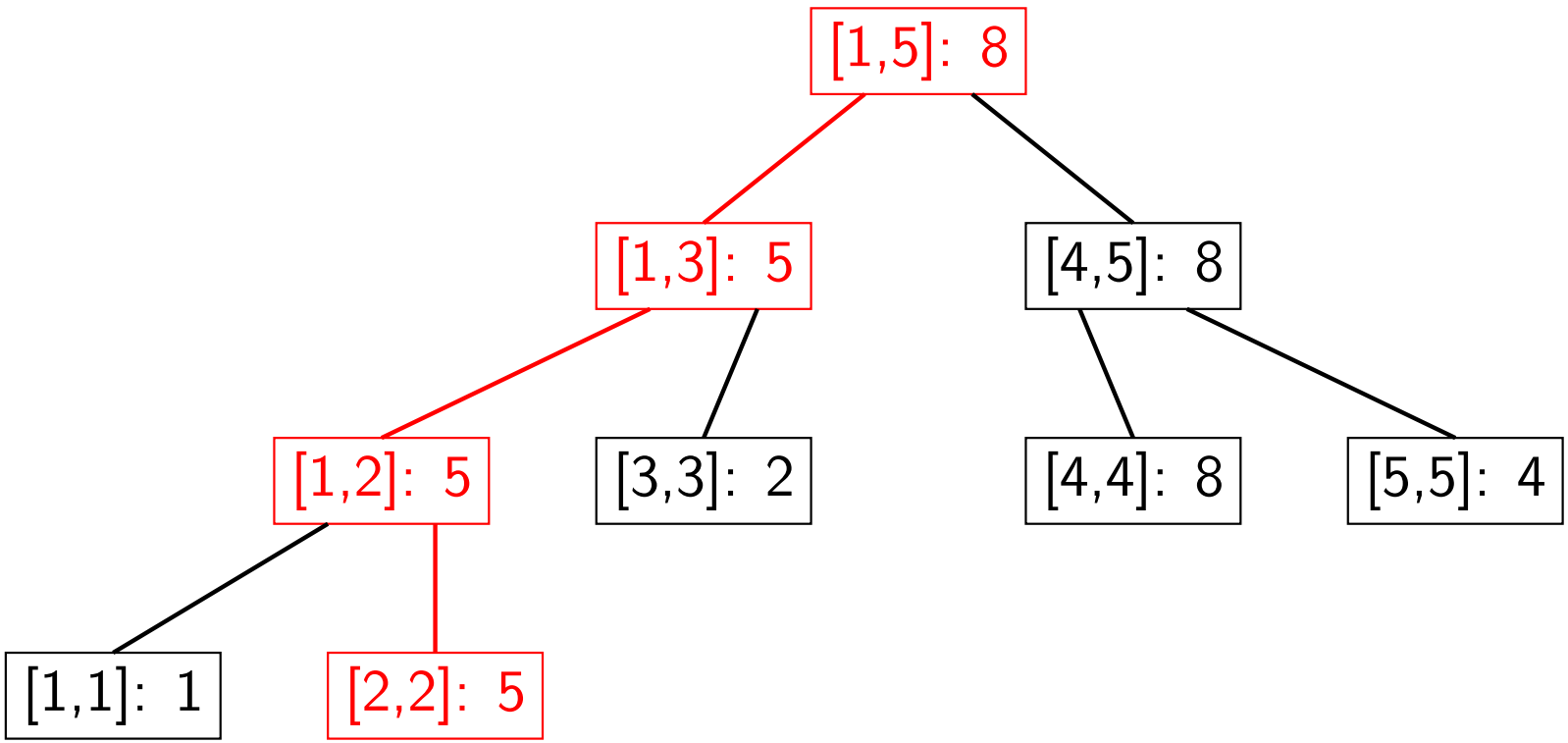
- $a = [1, 16, 2, 8, 4]$



- 答案是從某些預處理資料組合出來的：不知道可不可行
- 大資料都可以由小資料構成、小資料大小大概是大資料兩倍：可以

基礎線段樹

- $a = [1, 16, 2, 8, 4]$



- 修改一個元素的時候, 讓包含這個元素的資料盡量少：好像可以（上圖是修改 a_2 改成 5 的狀況）

基礎線段樹

- 在繼續下去之前，我們先來看看這棵樹的性質：
  - 樹的節點總共有 $2N - 1$ 個
  - 樹的高度為 $O(\log N)$
  - 他是二元樹
- 把樹上的數值填好只需要花 $O(N)$ 的時間（從葉子一路往上填，父節點的答案從左右子樹更新）
- 這種樹我們稱為「線段樹」
- 來看看怎麼把樹建出來吧！

基礎線段樹 – 範例實作

```
1  const int inf = numeric_limits<int>::max();
2  struct Node {
3    Node *lc, *rc;
4    int mx;
5    void pull() { mx = max(lc->mx, rc->mx); }
6  } *root = nullptr;
```

pull() 這個函數用途是父節點利用左右子樹的答案更新自己的答案，**每當任何子樹的值有更動時，都需要重新呼叫這個 function 更新父節點的答案**

基礎線段樹 – 範例實作

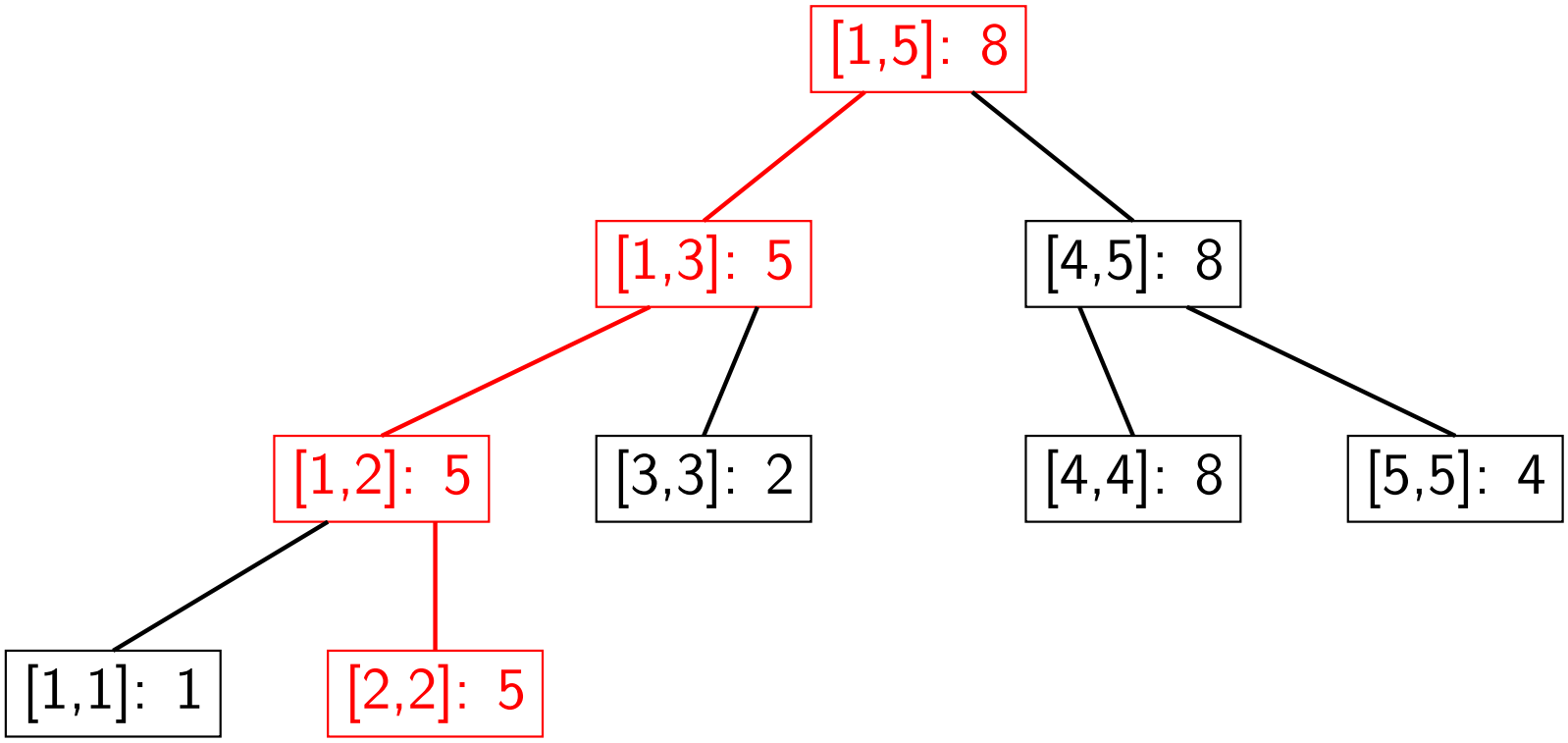
```
1  Node *build(int arr[], int l, int r) { // 回傳區間 [l, r]
   ↪ 這段區間構成的線段樹的根節點
2    Node *res = new Node();
3    if (l == r) { // 葉節點
4      res->lc = res->rc = nullptr;
5      res->mx = arr[l];
6    } else {
7      int m = (l + r) / 2;
8      res->lc = build(arr, l, m); // 把左半區間建好的樹接在自己
       ↪ 的左子樹上
9      res->rc = build(arr, m + 1, r); // 把右半區間建好的樹接
       ↪ 在自己的左子樹上
10     res->pull(); // 更新父節點資訊
11   }
12   return res;
13 }
```

基礎線段樹 – 範例實作

- 線段樹的實作會使用到大量的遞迴！
- 在遞迴線段樹節點的時候，通常會把該節點代表的區間（上述 code 的 L, R）一起遞迴下去
- 線段樹的實作有很多種版本，這裡提供的是指標版

基礎線段樹

- $a = [1, 16, 2, 8, 4]$



- 單點修改其實只會從**葉節點一路修改到根**
- 樹高是 $O(\log N)$，所以複雜度為 $O(\log N)$

基礎線段樹 – 修改二步驟

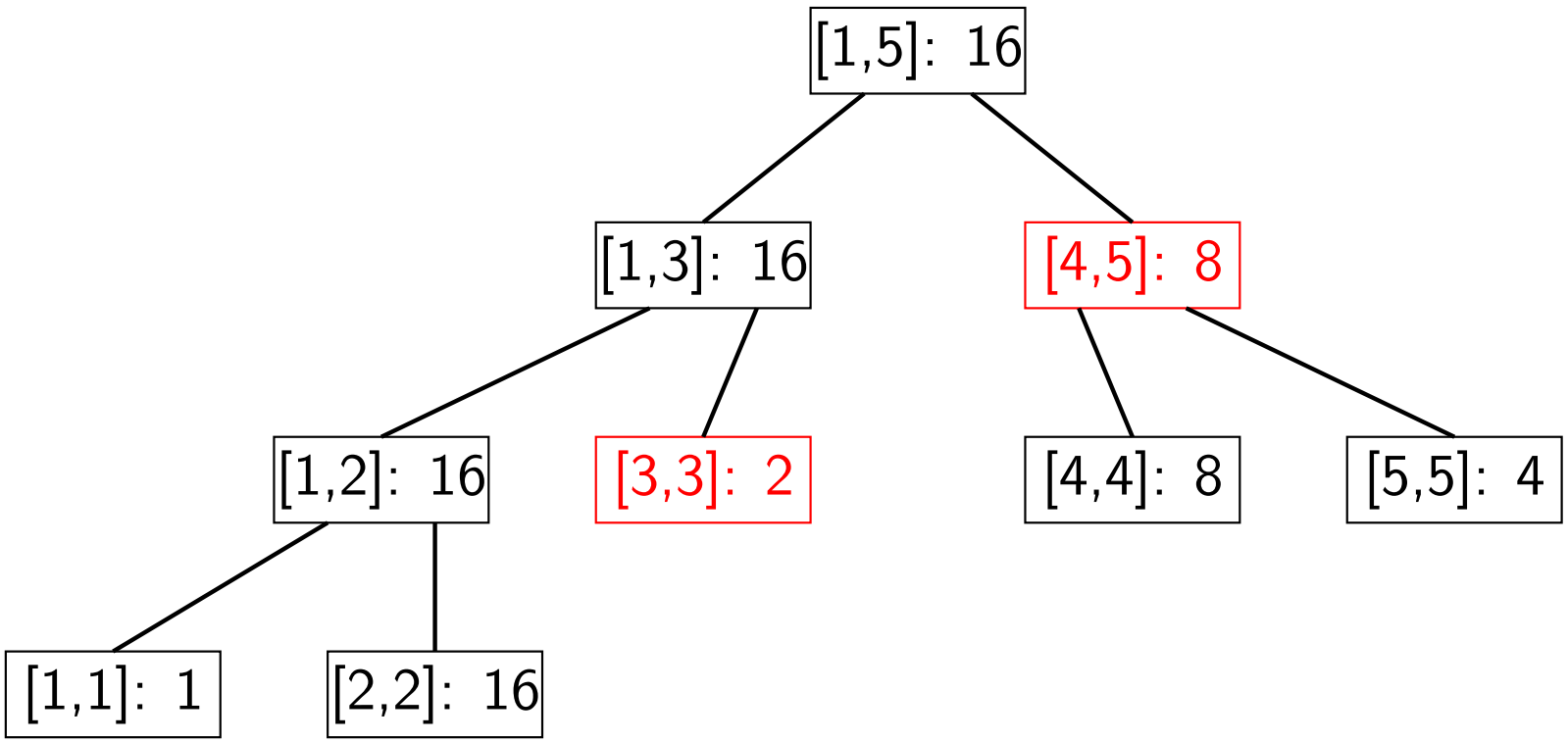
- 修改的位置是 p
- 假設目前的節點儲存的資訊是 $[L, R]$
- 修改二步驟：
  1. 如果 $[L, R]$ 區間長度為 1，可以知道這就是是一個葉節點。
  2. 否則設 $M = \lfloor \frac{L+R}{2} \rfloor$，$p$ 一定在 $[L, M]$ 或是 $[M+1, R]$ 其中之一，依照 p 與 M 的關係決定往哪個子樹遞迴。在子樹更新完之後，**更新這個節點的最大值**成為兩個子節點當中較大的。

基礎線段樹 – 範例實作

```
1  void modify(Node *nd, int val, int p, int l, int r) {
2    if (l == r) { // 葉節點
3      nd->mx = val;
4      return;
5    }
6    int m = (l + r) / 2;
7    if (p <= m) // 看修改的點是在左邊還是右邊
8      modify(nd->lc, val, p, l, m);
9    else
10     modify(nd->rc, val, p, m + 1, r);
11   nd->pull(); // IMPORTANT!!
12 }
```

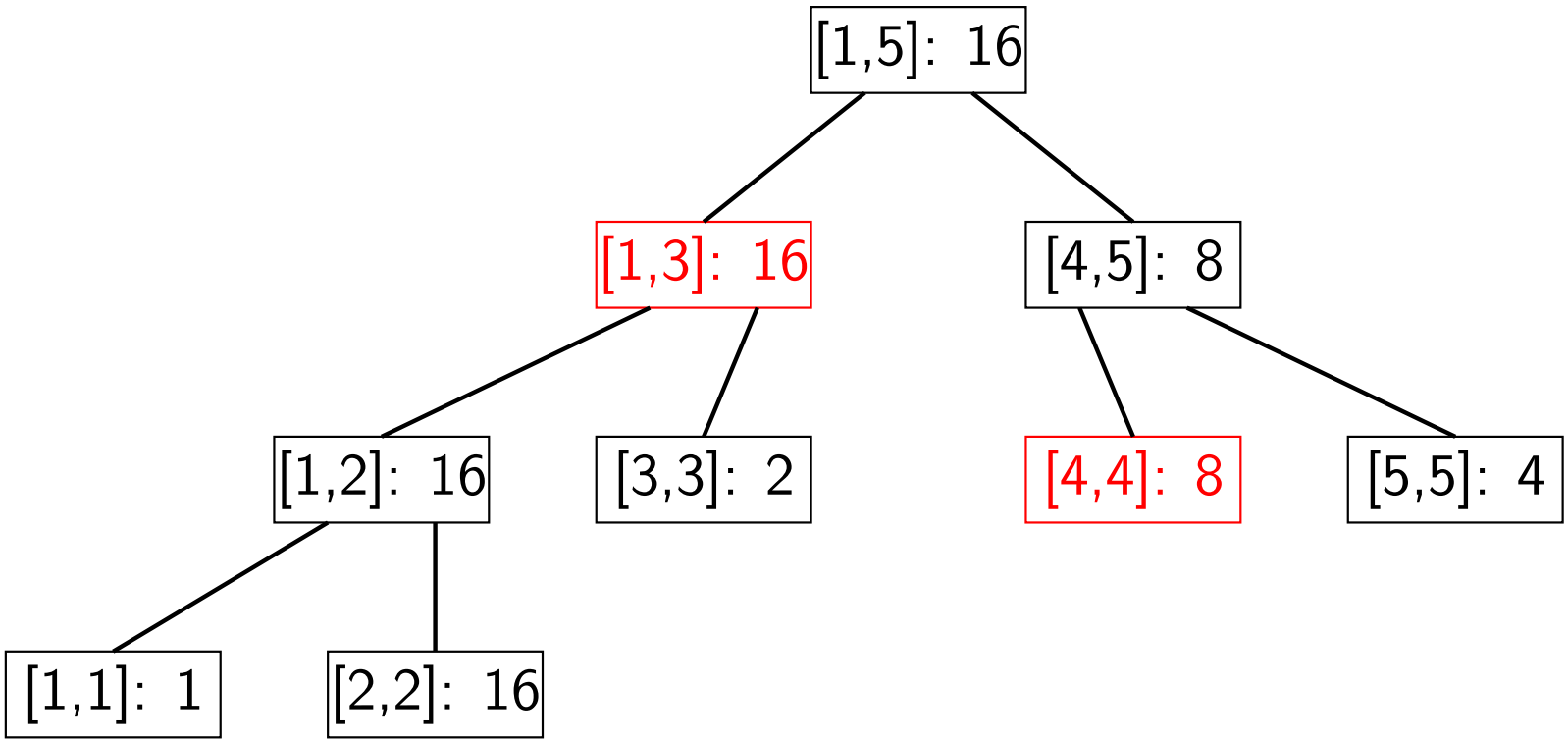
基礎線段樹 – 查詢

- 查詢 $[3, 5]$



基礎線段樹 – 查詢

- 查詢 $[1, 4]$



- 好像可以由蠻少數量的區間得到答案ㄋㄟ，可是具體是多小呢？

基礎線段樹 – 查詢

- 把選法說的具體一點：
  - 從根往下看
  - 每當看到一個被詢問區間完全包含的節點時，就直接使用該節點的答案
  - 否則就嘗試左右遞迴取出比較好的區間

基礎線段樹 – 查詢三步驟

- 目標：區間詢問 $[ql, qr]$（$ql \leq qr$）的最大值
- 假設目前的節點儲存的資訊是 $[L, R]$
- 查詢三步驟：
  1. 如果 $[L, R]$ 完整的被 $[ql, qr]$ 包含，即 $ql \leq L \leq R \leq qr$，那麼我們可以直接取用這個節點的最大值。
  2. 如果 $[L, R]$ 完全跟 $[ql, qr]$ 沒有交集，那麼可以退出遞迴了。
  3. 否則，遞迴往兩個子樹求解。

基礎線段樹 – 查詢三步驟

- 複雜度是多少呢？
- 很神奇的是，答案可以由 $O(\log N)$ 個預處理資料得到，並且從根走到 $+$ 走完這 $O(\log N)$ 個資料還是 $O(\log N)$
- 很神奇的結論！證明記不起來沒關係
- **這就是線段樹的精隨：他可以把一個大區間拆成 $O(\log N)$ 個小區間！**

基礎線段樹 – 查詢時間複雜度證明

- 先找到一個節點 M，詢問區間的所有值都在以這個節點為根的子樹裡面，而且這個點要盡量的深（白話文：盡量剛剛好包住詢問區間）
  - 從根走到 M 路上只會一直 3.
  - 用 3. 之後的左右子樹，一個會進到 3.，另外一個進到 2.
  - 樹深最多 $O(\log N)$，因此最多只會碰到 $O(\log N)$ 個節點
- 從 M 開始後，查詢區間其實就被切成左半邊跟右半邊。
- 左半邊其實是從 M 的 mid 往左的某段後綴
- 右半邊其實是從 M 的 $mid + 1$ 往右的某段前綴
- 而後綴 / 前綴的好處是，每當 3. 發生時，左右子樹一定至少有一個是 1. 或 2.
- 而 3. 在每個深度最多只會發生一次，所以整題複雜度還是 $O(\log N)$
- 整體而言，詢問區間只需要由 $O(\log N)$ 個小區間構成，而造訪這些小區間的複雜度也是 $O(\log N)$！

基礎線段樹 – 範例實作

```
1  int query(Node *nd, int ql, int qr, int l, int r) {
2    if (r < ql || l > qr)   // 完全不包含
3      return -inf;
4    if (ql <= l && r <= qr) // 完全包含
5      return nd->mx;
6    int m = (l + r) / 2;
7    return max(query(nd->lc, ql, qr, l, m), query(nd->rc,
     ↪  ql, qr, m + 1, r)); // 左右遞迴
8  }
```

BIT

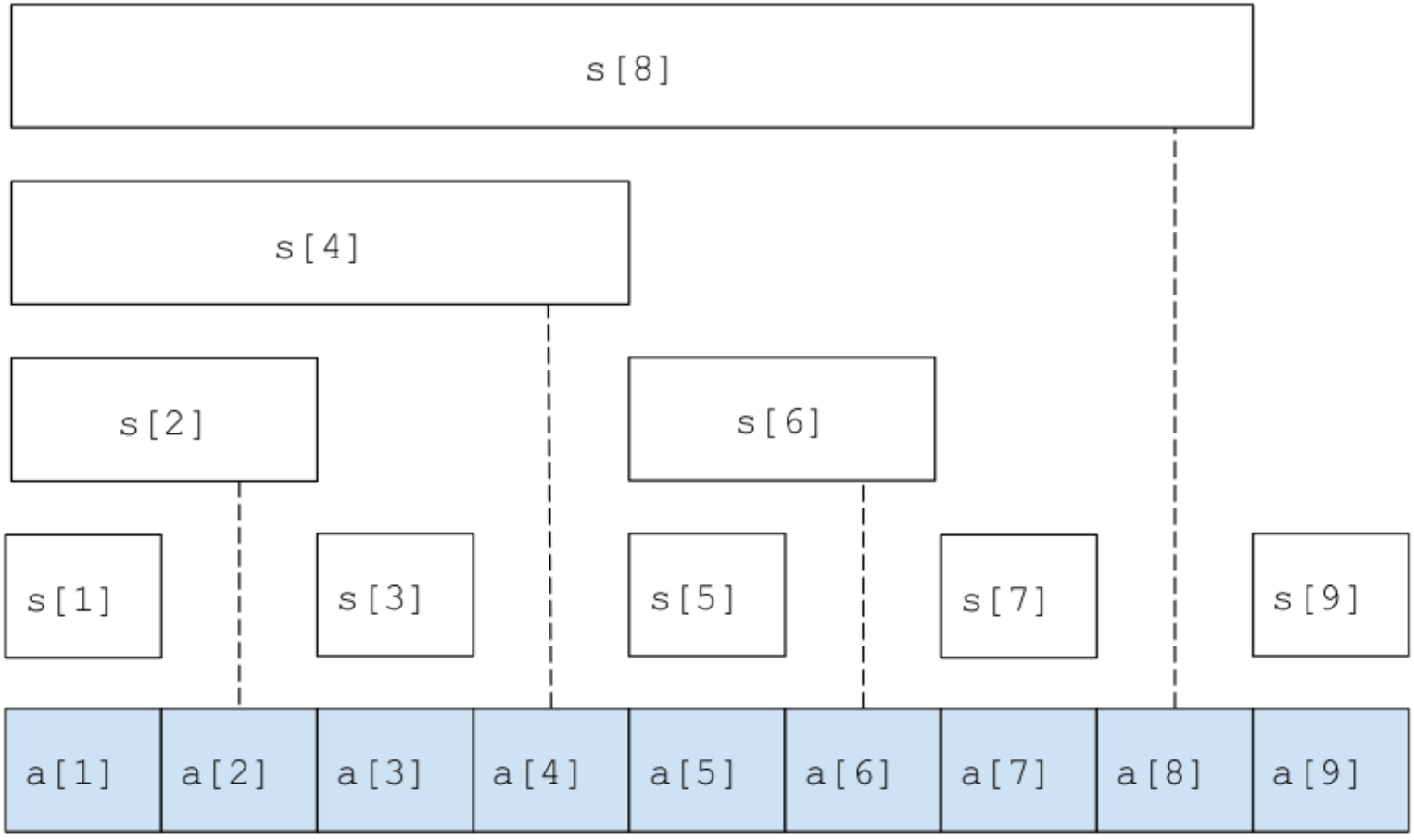
BIT

- Binary Indexed Tree
- Fenwick Tree
- 樹狀數組
- 支援操作：單點加值、前綴求和

BIT

- 我們有一個陣列 $a[1 \ldots n]$，以及另一個陣列 $s[1 \ldots n]$
- $s[i]$ 紀錄的是 $\sum_{j=i-lowbit(i)+1}^{i} a[j]$
- $lowbit(x)$ 的值是 x 寫成二進位時最小的一個 1-bit 的值
- 比如說，$44 = 101100_2, lowbit(44) = 2^2 = 4$
- $lowbit(x) = x\&(-x)$
- 好抽象，來看圖吧！

BIT



有沒有很像沒有右邊子樹的線段樹！

BIT – 前綴求和

- 假設要求 $a[1...46]$
- $= s[46] + a[1...44]$
- $= s[46] + s[44] + a[1...40]$
- 有發現什麼嗎？
- 令 $sum(x) = a[1] + a[2] + \cdots + a[x-1] + a[x]$
- 前面好像就只是 $sum(x) = s[x] + sum(x - lowbit(x))$
- 遞迴求 $sum(x)$！

BIT – 前綴求和

```
1  int sum(int id) {
2    int res = 0;
3    for (int i = id; i > 0; i -= i & -i)
4      res += s[i];
5    return res;
6  }
```

BIT – 單點加值

- 假設要把 $a[46]$ 加上 x，把有蓋到 46 的值加上去就好了
- 首先觀察發現有蓋到 46 的節點編號一定不小於 46 ，畢竟節點 i 蓋到的區間是 $(i - lowbit(i), i]$
- 其實古人的智慧告訴我們，會蓋到 i 的區間編號其實就是以下的序列：
- $\{s_0 = i, s_{k+1} = s_k + lowbit(s_k)\}$ ，一直到 $s_k > N$ 為止
  - 簡單小證明：
  - $lowbit(s_{k+1}) > lowbit(s_k)$，所以 s_{k+1} 會覆蓋到 s_k 覆蓋的範圍
  - 而 s_{k+1} 恰好是所有覆蓋 s_k 範圍中最小的那一個

BIT – 前綴求和

```
1  void upd(int id, int x) {
2    for (int i = id; i <= n; i += i & -i)
3      s[i] += x;
4  }
```

BIT – 單點加值、區間求和

- 有了這些，就可以解決一個很常見的問題了：單點加值、區間求和
- 單點加值 $a_i+=x$ 直接 `upd(i, x);` 就好
- 區間求和 $[l,r]$ 其實可以用前綴和的方式得到：`sum(r) - sum(l - 1);`

BIT – 第 K 小

- 假設 $a[x]$ 是存 x 這個數字出現過幾次
- 那 $sum(i)$ 就是 $\leq i$ 的元素有幾個
- i 是集合裡第 k 小的元素若且唯若 $sum(i-1) < k$ 且 $sum(i) \geq k$
- 二分搜！
- 直接呼叫 query 二分搜是 $O(\log^2 N)$，複雜度似乎不太好
- 用 BIT 的結構在上面枚舉 bit 二分搜！

BIT – 第 K 小

```
1  int kth(int k) {
2    int res = 0;
3    for (int i = 1 << __lg(n); i > 0; i >>= 1)
4      // 決定 res 應該在 [res, res + i - 1] 還是 [res + i,
       ↪  res + 2i - 1]
5      if (res + i <= n && dat[res + i] < k) // 左邊的個數不
       ↪  夠，往右邊走 (bit = 1)
6        k -= dat[res += i];
7      // 否則就代表左邊其實是夠的，就往左邊走 (bit = 0)
8    return res + 1;
9  }
```

BIT – BIT 與差分

- 差分序列：$d_i = a_i - a_{i-1}$
- 如果 BIT 維護差分序列的話，就可以變成支援區間加值、單點求值的資料結構
- $a_i = d_1 + d_2 + \ldots + d_i$
- 把 $a_l, \ldots, a_r$ 全部加上 x，相對應在差分數列上的變化其實只是 d_l 加上 x，d_{r+1} 扣掉 x

BIT – 總結

- 支援單點修改、前綴求和、查詢第 k 小的資料結構
- 常數很小、寫起來比線段樹簡單許多
- 實用！

進階線段樹

進階線段樹

- 來看看強大的線段樹可以做些什麼吧～
- 這裡講的技巧只是線段樹強大應用中的冰山一角，有興趣的讀者可以閱讀進階資料結構講義，或者是自行上網搜尋更加進階的應用喔！

進階線段樹 – 精妙狀態合併

- 線段樹不僅可以維護簡單的最大最小值或總和，只要能夠快速從左右子區間的資訊得知合併後的區間的所有資訊的問題，就可以用線段樹解決
- 來看例題！

題目 (區間最大連續和)

給定長度為 N 的序列 $a_1, a_2, \ldots, a_N$，接著有 Q 個形如 `l r` 的詢問，請你回答 $a_l, a_{l+1}, \ldots, a_r$ 這個序列的最大連續和。

- $N, Q \leq 5 \times 10^5$

進階線段樹 – 精妙狀態合併

- 想想看怎麼用分治法求出最大連續和！
- 在線段樹每個節點上，我們可以維護以下資訊：
  - 以區間最左端為左界的最大連續和，即最大總和的前綴。
  - 以區間最右端為右界的最大連續和，即最大總和的後綴。
  - 整個區間的最大連續和。
  - 整個區間的總和。
- 所有的區間可以分成三種：通過區間中點、完全落在左半區間、完全落在右半區間
- 中點的最大連續和一定是左半區間的最大後綴加上右半區間的最大前綴

進階線段樹 – 精妙狀態合併

```
struct Info {
  int64_t sum, lmx, rmx, mx;
};
Info combine(const Info &lhs, const Info &rhs) {
  Info res;
  res.sum = lhs.sum + rhs.sum;
  res.lmx = max(lhs.lmx, lhs.sum + rhs.lmx);
  res.rmx = max(lhs.rmx + rhs.sum, rhs.rmx);
  res.mx = max({lhs.mx, rhs.mx, lhs.rmx + rhs.lmx});
  return res;
}
```

進階線段樹 – 懶人標記

- 直接來看例題！

題目 (區間加值、區間最大值)

給定一個長度為 N 的正整數序列 $a_1, \ldots, a_N$，接著有 Q 個詢問，每個詢問形如

- `1 l r`，請輸出 $a_l, a_{l+1}, \ldots, a_r$ 當中最大的數字是多少。
- `2 l r d`，讓 $a_l, a_{l+1}, \ldots, a_r$ 全部增加 d。

- $N, Q \leq 5 \times 10^5$
- $d > 0$

進階線段樹 – 懶人標記

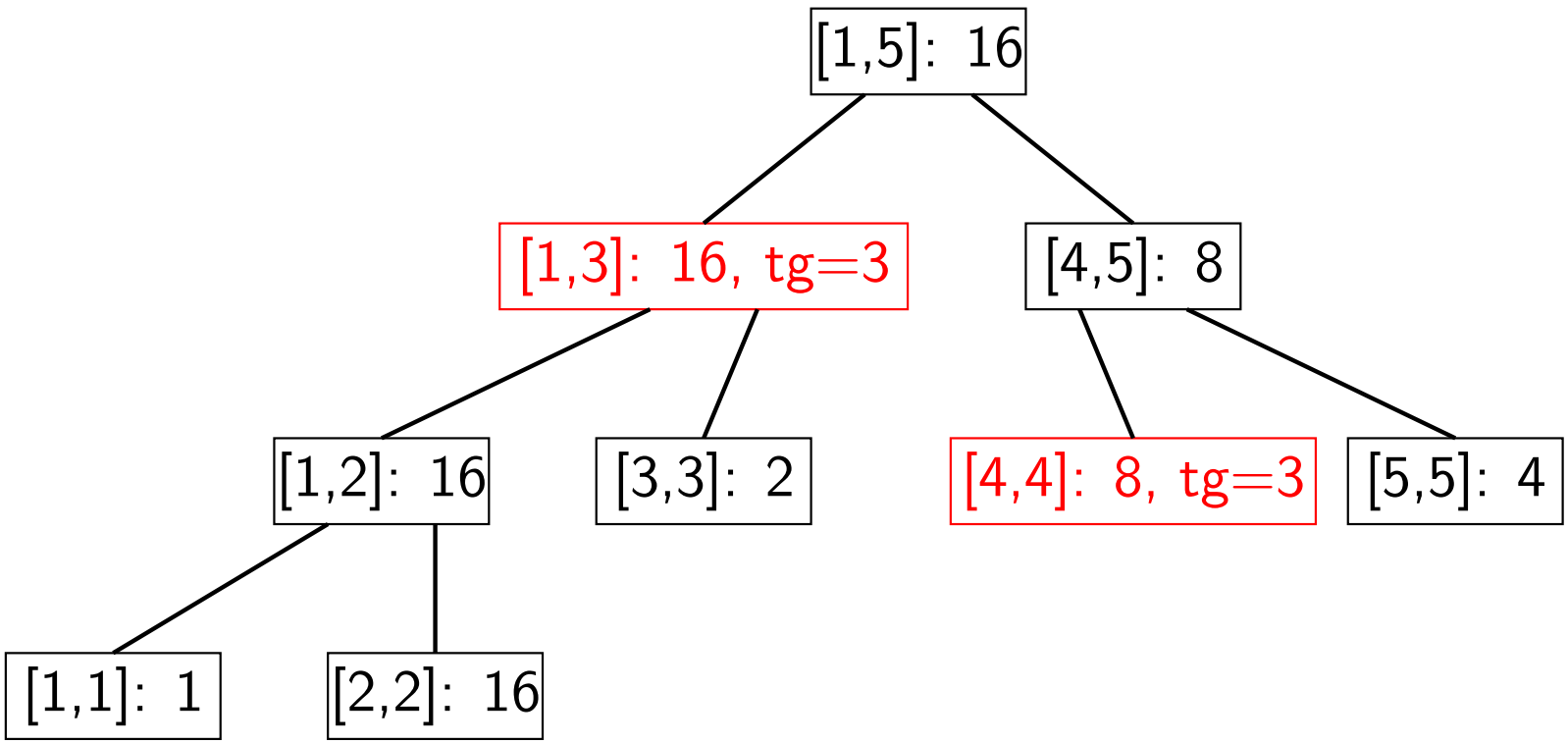
- 如果這題只是單純做 $r - l + 1$ 次單點修改的話，修改的複雜度會退化成 $O(N \log N)$
- 有沒有辦法把修改的區間拆成 $O(\log N)$ 個區間，並對每個區間做一些事情後，就達到修改的效果呢？
- 可以！

進階線段樹 – 懶人標記

- 每個節點上面存一個「標記」，表示被**這個節點對應的區間涵蓋的所有元素都必須加上某個值**
- 在區間修改時，我們就在所有區間打上這樣的「標記」
- 在打標記的時候，同時修改該節點的資訊，並且一路 pull 回去！

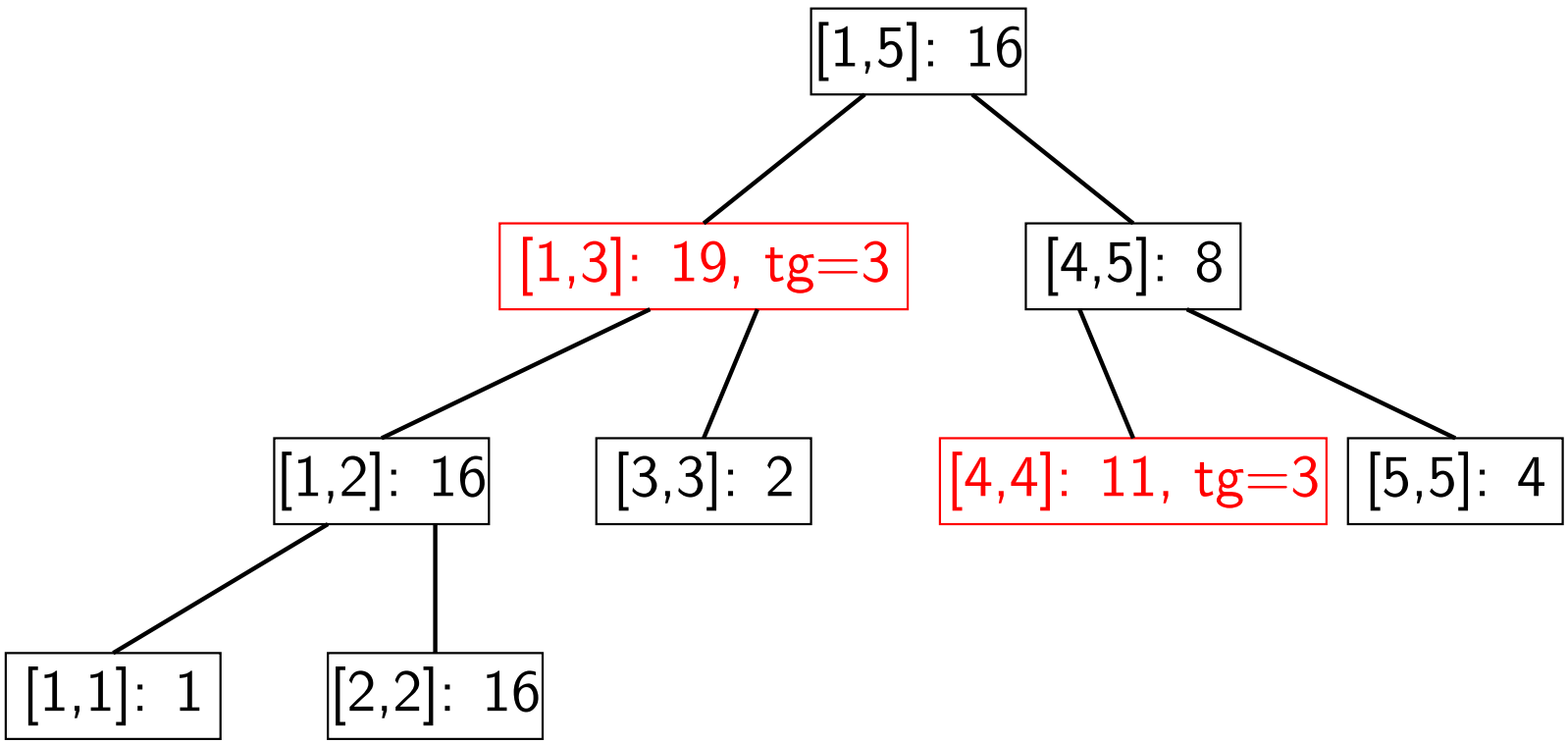
進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3
- 先把 tag 打上去！



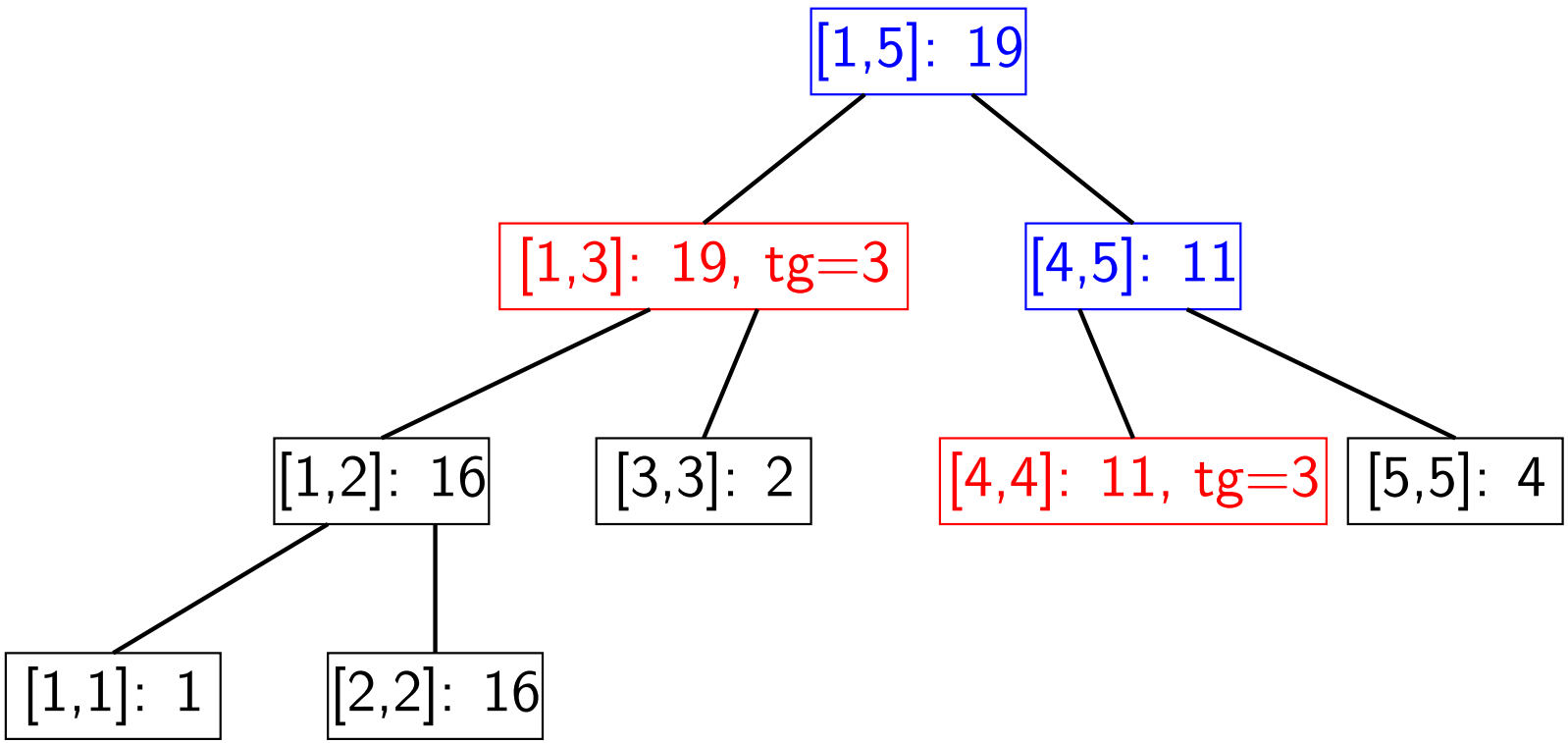
進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3
- 更新被打 tag 的區間的答案



進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3
- pull 回 root！



進階線段樹 – 懶人標記

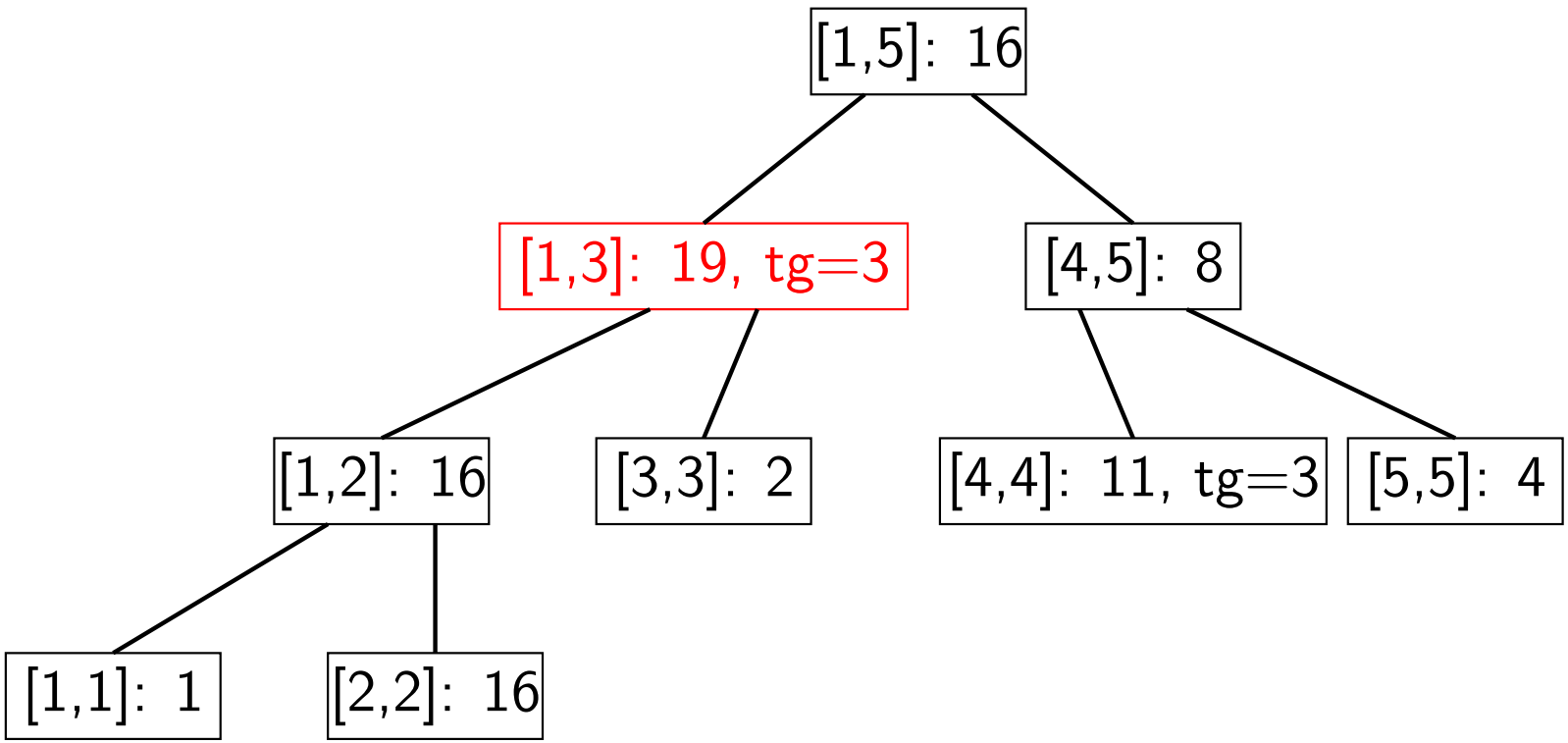
- 一些發現：
  - 有打 tag 的節點的父親們的答案都是正確的
  - tag 底下的節點還沒有被更新到，但如果沒碰到他們的話，他們其實也不用被更新 (?)
  - 有 tag 要更新答案很簡單
- 標記的關鍵：**需要的時候再去更新，沒需要就不要管他**
- 如果有兩個標記撞在一起，把他加起來就好了（標記很好合併）！
- 那，什麼時候是**需要的時候**？

進階線段樹 – 懶人標記

- 那，什麼時候是**需要的時候**？
- 注意到標記的特性：這個節點對應的區間涵蓋的所有元素都必須加上某個值
- 也就是說，如果**動到這個節點以下的節點**，就是**需要把標記往下推**
- 直接來看例子！

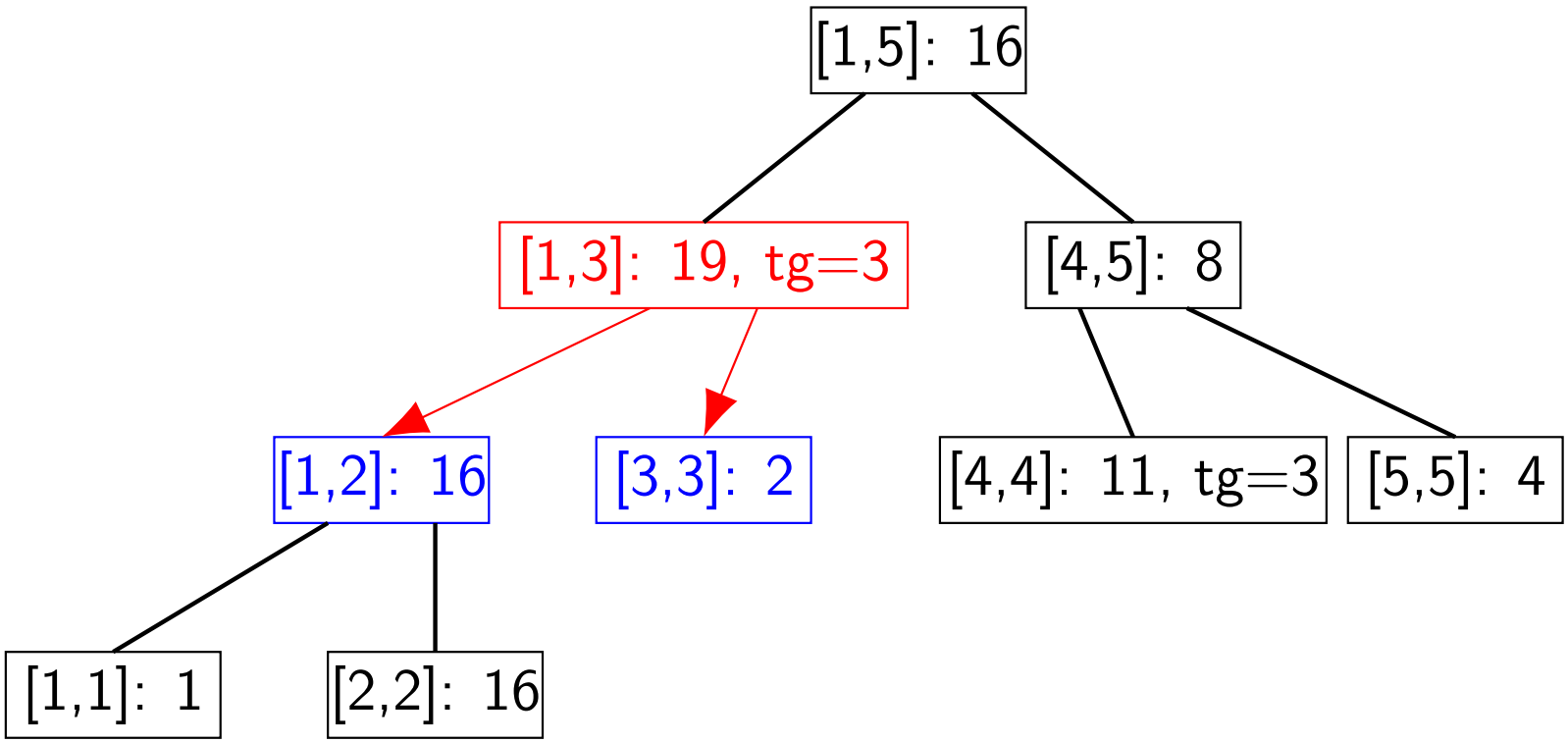
進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3，詢問 $[3, 3]$ 的最大值
- 發現我們需要走到一個有 tag 的節點的兒子



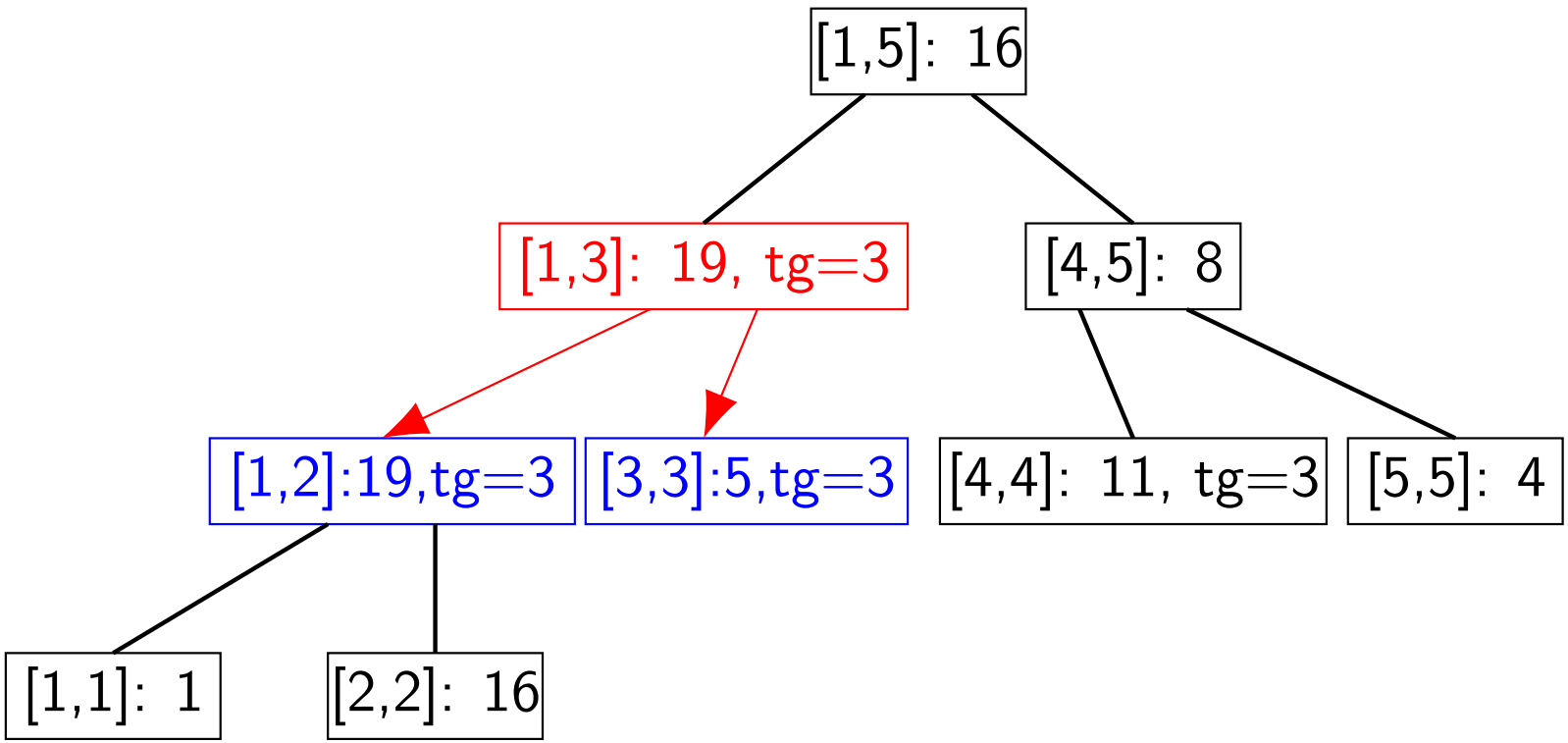
進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3，詢問 $[3, 3]$ 的最大值
- 把 tag 往下推？其實就是告訴左右孩子底下所有元素必須要加上 3



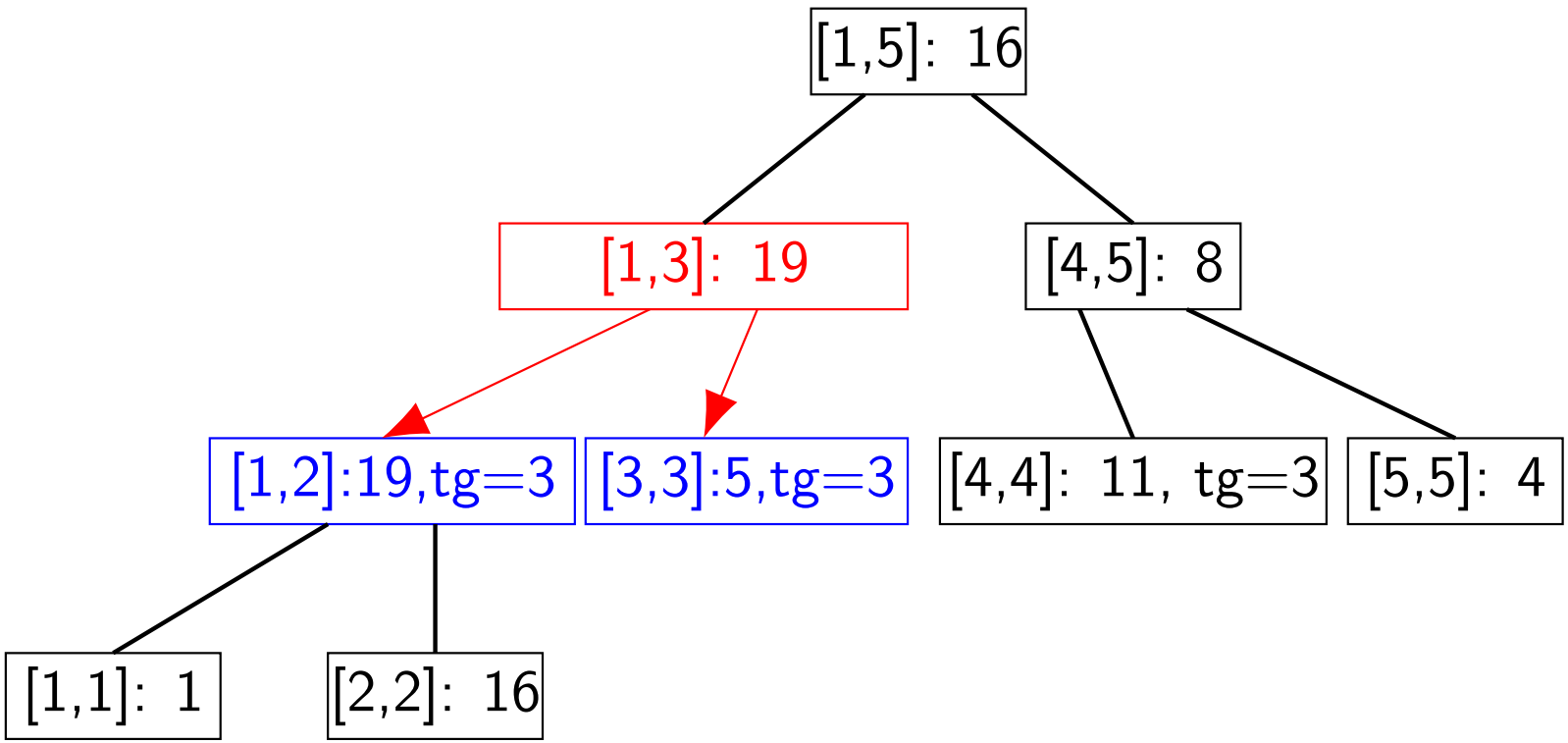
進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3，詢問 $[3, 3]$ 的最大值
- 把 tag 資訊丟給左右子樹，並且更新左右子樹的答案



進階線段樹 – 懶人標記

- $a = [1, 16, 2, 8, 4]$，並把 $[1, 4]$ 加上 3，詢問 $[3, 3]$ 的最大值
- 原本的 tag 功成身退！



進階線段樹 – 懶人標記

- 具體來說，之後某次區間修改或查詢而**造訪到某個節點時，我們才將這個節點的「標記」往下推**，把標記的資訊傳給小孩
- 推標記的複雜度是 $O(1)$，不影響原本的複雜度！
- 因為是只在必要的時候更新標記，因此這個技巧又被稱為**懶人標記**
- 我們常把推標記的函數叫做 `push();`
- 直接來看程式碼吧～

進階線段樹 – 懶人標記

```
1  struct Node {
2    Node *lc, *rc;
3    int mx, tag; // tag 左右子樹整個子樹需要加上的數值
4    void pull() { mx = max(lc->mx, rc->mx); }
5  } *root = nullptr;
```

進階線段樹 – 懶人標記

```
1  void push(Node *nd, int l, int r) {
2    // 把 nd 的 tag 往左右子樹推
3    if (l == r) nd->tag = 0;
4    if (!nd->tag) return;
5    nd->lc->tag += nd->tag; // 左子樹的 tag 跟 nd 的 tag 合
     ↪  併，這題 tag 是加上數值，因此直接加起來就可以
6    nd->lc->mx  += nd->tag; // 左子樹的最大值加上 nd 的 tag
7    nd->rc->tag += nd->tag;
8    nd->rc->mx  += nd->tag;
9    nd->tag = 0; // 清空 tag
10 }
```

進階線段樹 – 懶人標記

```
1  void modify(Node *nd, int ql, int qr, int d, int l, int
   ↪  r) {
2    if (r < ql || l > qr)
3      return;
4    if (ql <= l && r <= qr) {
5      nd->mx += d;  // 當前前點的所有數字加上 d，最大值也需要加上
       ↪  d
6      nd->tag += d; // 告訴左右子樹，最大值需要加上 d
7      return;
8    }
9    push(nd, l, r); // 需要拜訪子樹的時候，把當前點的 tag 往下推
10   int m = (l + r) / 2;
11   modify(nd->lc, ql, qr, d, l, m);
12   modify(nd->rc, ql, qr, d, m + 1, r);
13   nd->pull();
14 }
```

進階線段樹 – 懶人標記

```
1  int query(Node *nd, int ql, int qr, int l, int r) {
2    if (r < ql || l > qr)
3      return 0;
4    if (ql <= l && r <= qr)
5      return nd->mx;
6    push(nd, l, r); // 需要拜訪子樹的時候，把當前點的 tag 往下推
7    int m = (l + r) / 2;
8    return max(query(nd->lc, ql, qr, l, m), query(nd->rc,
     ↪  ql, qr, m + 1, r));
9  }
```

進階線段樹 – 懶人標記

- 使用時機？
- 標記必須要可以合併、並且能夠快速的由「標記」所儲存的資訊推出這個節點的資訊應該被修改成什麼
- 有的時候順序很重要！
- 常見出現時機：區間加值區間求和、區間加值區間取 max、區間加值區間乘值區間求和..... 等等
- 不知道大家有沒有發現，`push()` 跟 `pull()` 好像會成對出現！
- 課外閱讀：有一種標記的方式叫「永久化標記」，這種標記就會直接把懶標固定在節點上，但詢問的時候就要多處理一些東西

進階線段樹 – copy on write 線段樹

- 又稱動態開點線段樹
- 需要使用的時候再把節點開出來！
- 直接來看例題：

題目 (二維區間求和)

給一 $N \times N (N \leq 10^9)$ 二維平面初始所有值都是零，
$Q (Q \leq 10^5)$ 筆操作，操作包含兩種：

- `add x y d`: 將座標 (x, y) 的元素加上 d
- `query x1 y1 x2 y2`: 詢問所有 (x, y) 滿足 $x1 \leq x \leq x2$ 且 $y1 \leq y \leq y2$ 的總和。

進階線段樹 – copy on write 線段樹

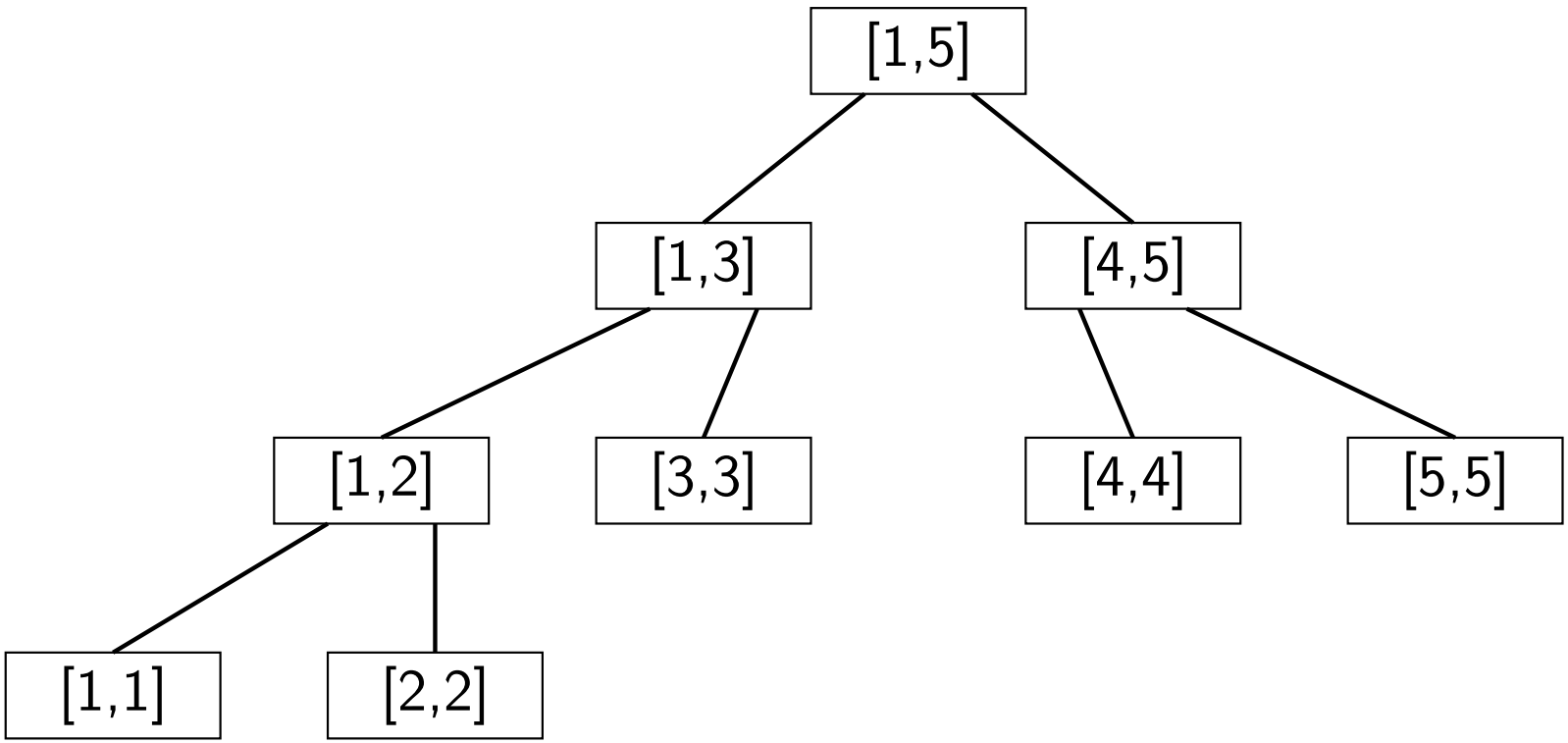
- 從題目的操作可以得知，最多只會有 10^5 個座標有值
- 若是定義在一維座標上，我們可以利用**離散化**的技巧依然用我們前面介紹的線段樹解決

進階線段樹 – copy on write 線段樹

- 然而，我們現在要處理的是二維座標
- 試著用線段樹套線段樹的方式吧！
- 等等，什麼是線段樹套線段樹？

進階線段樹 – copy on write 線段樹

- 對 x 軸開一個線段樹，x 軸相關的節點的內容是對 y 軸的線段樹！



- 比如說，$[1,3]$ 這個節點存一個 y 軸資訊的線段樹，y 軸線段樹的 $[4,5]$ 就是存 $[1..3,4..5]$ 的總和！

進階線段樹 – copy on write 線段樹

- 雖然做完離散化，還是 $O(Q^2)$ 的空間
- 但，最多只有 Q 個座標有值，因此，最多只有 $O(Q \lg^2 N)$ 個節點有值！只要能只記錄這些節點便勝利了
  - 在 x 軸的世界中會經過 $O(\log N)$ 個節點
  - 每個節點的線段樹需要經過 $O(\log N)$ 個 y 軸節點
- 來看 code 吧！

進階線段樹 – copy on write 線段樹

```
1  struct Node2 { // 第二個維度的線段樹節點
2    int val;
3    Node2 *lc, *rc;
4    Node2() {
5      val = 0;
6      lc = rc = NULL;
7    }
8  };
9  struct Node1 { // 第一個維度的線段樹節點
10   Node1 *lc, *rc;
11   Node2 *c;
12   Node1() {
13     c = NULL;
14     lc = rc = NULL;
15   }
16 };
```

進階線段樹 – copy on write 線段樹

```
1  int Val2(Node2 *node) { return node ? node->val : 0; }
2  void pull2(Node2 *node) { node->val = Val2(node->lc) +
   ↪  Val2(node->rc); }
```

進階線段樹 – copy on write 線段樹

```
1  void modify2(Node2 *node, int Y1, int Y2, int qy, int d)
   ↪  {
2    if (Y1 == Y2) {
3      node->val += d;
4      return;
5    }
6    int mid = (Y1 + Y2) >> 1;
7    if (qy <= mid) {
8      if (!node->lc) node->lc = new Node2(); // 有需要再開新
       ↪  的 y 軸點
9      modify2(node->lc, Y1, mid, qy, d);
10   } else {
11     if (!node->rc) node->rc = new Node2();
12     modify2(node->rc, mid + 1, Y2, qy, d);
13   }
14   pull2(node);
15 }
```

進階線段樹 – copy on write 線段樹

```
1  void modify1(Node1 *node, int X1, int X2, int qx, int
   ↪  qy, int d) {
2    if (!node->c) node->c = new Node2(); // 有需要再開新的 y
     ↪  軸點
3    modify2(node->c, 1, n, qy, d);
4    if (X1 == X2) {
5      return;
6    }
7    int mid = (X1 + X2) >> 1;
8    if (qx <= mid) {
9      if (!node->lc) node->lc = new Node1(); // 有需要再開新
       ↪  的 x 軸點
10     modify1(node->lc, X1, mid, qx, qy, d);
11   } else {
12     if (!node->rc) node->rc = new Node1();
13     modify1(node->rc, mid + 1, X2, qx, qy, d);
14   }
15 }
```

進階線段樹 – copy on write 線段樹

```
1  int query2(Node2 *node, int Y1, int Y2, int qy1, int
   ↪  qy2) {
2    if (qy1 > Y2 || qy2 < Y1)
3      return 0;
4    if (!node)
5      return 0; // no data!
6    if (qy1 <= Y1 && Y2 <= qy2)
7      return node->val;
8    int mid = (Y1 + Y2) >> 1;
9    return query2(node->lc, Y1, mid, qy1, qy2) +
10          query2(node->rc, mid + 1, Y2, qy1, qy2);
11 }
```

進階線段樹 – copy on write 線段樹

```
1  int query1(Node1 *node, int X1, int X2, int qx1, int
   ↪  qx2, int qy1, int qy2) {
2    if (qx1 > X2 || qx2 < X1)
3      return 0;
4    if (!node)
5      return 0; // no data!
6    if (qx1 <= X1 && X2 <= qx2)
7      return query2(node->c, 1, n, qy1, qy2);
8    int mid = (X1 + X2) >> 1;
9    return query1(node->lc, X1, mid, qx1, qx2, qy1, qy2) +
10          query1(node->rc, mid + 1, X2, qx1, qx2, qy1,
            ↪  qy2);
11 }
```

進階線段樹 – 持久化線段樹

- 修改後依舊保有歷史版本的資料結構
- 直接來看例題：

題目 (歷史版本和)

給定一個長度 N 的序列以及 M 個單點修改，接下來有 Q 次詢問，每次詢問會問你在第 m 次修改後，區間 $[l, r]$ 的總和是多少。強制在線。

- $N, Q \leq 10^5$

進階線段樹 – 持久化線段樹

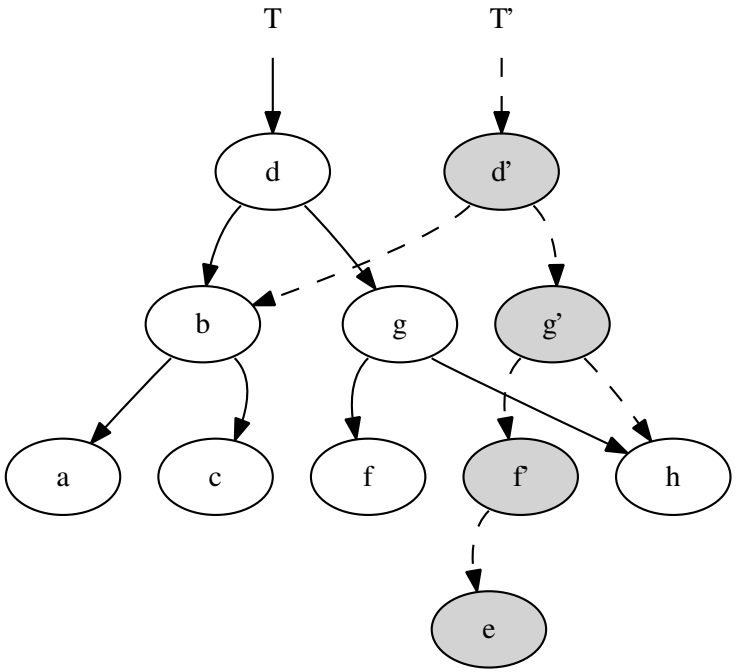
- 我會！
- 每個版本各開一個線段樹就好啦～
- 總共需要 $O(N)$ 個版本，每個版本 $O(N)$ 個點，總共 $O(N^2)$ 個點，TLE！
- 怎麼辦 OuO

進階線段樹 – 持久化線段樹

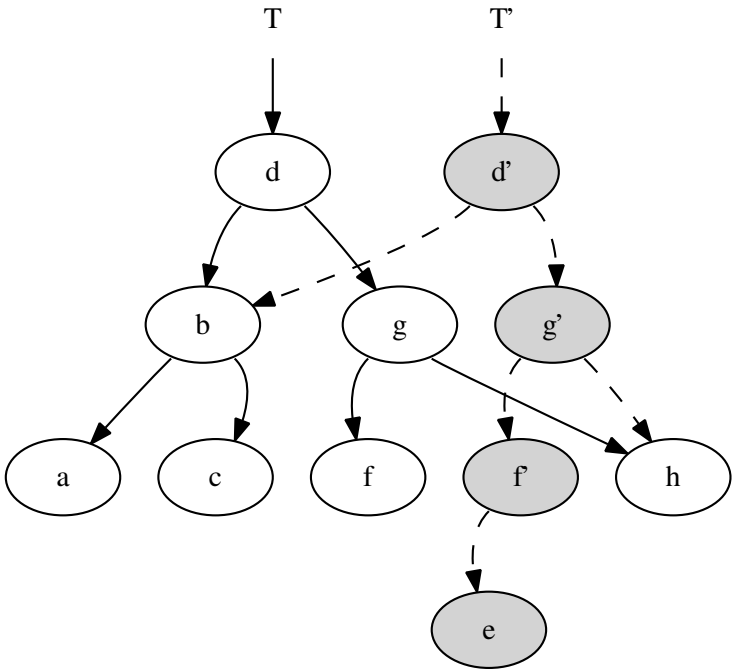
- 努力觀察一下：每次修改都只會動到 $O(\log N)$ 個節點
  - 這些節點其實就是單點修改碰到的那些點
- 那麼我們為何不能用 $O(\log N)$ 的空間來儲存兩個版本之間的差距呢？

進階線段樹 – 持久化線段樹

- 努力觀察一下：每次修改都只會動到 $O(\log N)$ 個節點
- 那麼我們為何不能用 $O(\log N)$ 的空間來儲存兩個版本之間的差距呢？
- 具體來說，我們每次想要更新一個節點的值時，會開出一個新節點儲存修改之後的資訊（不要看下面那張圖的 e 點）



進階線段樹 – 持久化線段樹



- 所以，每當需要一個新的版本的時候
- 先把舊版本根節點重新複製一遍到新版本的根節點上
- 每當要進入某個點修改時，先把該節點複製一份到新版本上
- 遞迴修改
- 修改結束後，因為新節點的子樹有變動，呼叫新節點的 `pull()`

進階線段樹 – 持久化線段樹

```
1  // get a copy of node
2  Node *getNode(Node *node) {
3    Node *tnode = new Node();
4    tnode->val = node->val;
5    tnode->lc = node->lc;
6    tnode->rc = node->rc;
7    return tnode;
8  }
```

進階線段樹 – 持久化線段樹

```
1  void modify(Node *node, Node *newNode, int L, int R, int
   ↪  i, int d) {
2    if (L == R) {
3      newNode->val += d;
4      return;
5    }
6    int mid = (L + R) >> 1;
7    if (i <= mid) {
8      newNode->lc = getNode(node->lc);
9      modify(node->lc, newNode->lc, L, mid, i, d);
10   } else {
11     newNode->rc = getNode(node->rc);
12     modify(node->rc, newNode->rc, mid + 1, R, i, d);
13   }
14   newNode->pull();
15 }
```

進階線段樹 – 持久化線段樹

- 很常運用在序列上！
- 常見建立的版本時機是：給定一個長度為 N 的序列，建立 N 個版本，第 i 個版本維護第 i 個位置的前綴（或者是後綴）的資訊

進階線段樹 – STL in 線段樹

- 線段樹的節點可以塞各式各樣的東西喔！

題目 (二維區間求和)

有 n 個目標物，每個目標物都是一個水平線段 $[s,t]$ 或是只含一點（$s=t$ 的情形），並且有各自的分數 w，所有目標物的高度（y 值）均大於 0 且互不相同（輸入按照 y 座標由大到小排序）。現在依序發射了 m 發砲彈，第 i 次會從 x 軸上的整數點 x_i 往上垂直發射砲彈，路線上第一個碰到的目標物就是擊中的目標，射中目標後砲彈與目標物一起消失，不會穿透。如果沒有擊中任何目標物，則該次射擊分數為 0。請輸出每次有擊中的目標物的分數總和。

- $n, m \leq 5 \times 10^5$
- $s, t, x_i \leq 10^6$

進階線段樹 – STL in 線段樹

- 在線段樹的每個節點當中，我們可以儲存一個 set
- set 裡面放一些完全包含這個節點對應的區間的水平線段
- 我們可以做 n 次的區間修改，在對應的節點裡面的 set 插入這個水平線段的 y 座標
- 對於每次的單點詢問，我們查看對應的葉節點到根的這條路徑上，y 座標最小的水平線段是哪一個
- 當知道哪個水平線段要被刪除之後，可以做一次區間修改，並且在對應的節點裡面的 set 刪除這個水平線段的 y 座標
- 因為每次都會造訪 $O(\log n)$ 個節點，而 std::set 插入、刪除的複雜度都是 $O(\log n)$，因此總共的複雜度會是 $((n+m)\log^2 n)$，空間複雜度則是 $O(n \log n)$

進階線段樹 – STL in 線段樹

- 事實上我們可以用一個 std::vector（或是 stack）來維護這樣的插入與刪除，因為所有刪除操作都在插入之後，如此時間複雜度可以減少一個 log
- 注意 STL 的 stack 內部預設是 deque，而空的 deque 佔的空間很多，因此在這種嵌套的資料結構特別要小心

進階線段樹 – 區間查詢與修改的對偶

- 直接來看例題！

題目 (區間取 max、單點查詢)

有一長度為 N 的序列 $a_1, a_2, \ldots, a_N$，一開始全部為 0。接下來有 M 個操作，每個操作形如 l, r, v ，表示對於所有 $l \leq i \leq r$，$a_i := \max(a_i, v)$。請你分別輸出 $a_1, a_2, \ldots, a_N$ 最後的值是多少。

- $N \leq 2 \times 10^5$
- $M \leq 5 \times 10^6$

進階線段樹 – 區間查詢與修改的對偶

- 相信在前面學過懶人標記後，大家已經知道如何用線段樹與懶人標記解決這個問題了
- 然而這樣做的複雜度是 $O(N + M \log N)$，很有可能無法在時限內通過
- 為什麼呢？因為詢問的數量太多了
- 但相對的，**修改沒那麼多**，並且**我們可以一次處理所有修改**
- 有什麼資料結構是支援快速查詢，且可以利用離線的性質呢？
- Sparse Table！

進階線段樹 – 區間查詢與修改的對偶

- 在 Sparse Table 中是把一個區間查詢拆成兩個部份重疊的區間查詢，並且兩個區間的長度為 2 的冪次
- 那麼我們也可以把區間修改拆成兩個長度為 2 的冪次的區間，並紀錄所有 $g(i, j)$ 表示 $[i, i + 2^j)$ 整個區間要跟多少取 max
- 最後再一次花 $O(N \log N)$ DP 便能得到每個單獨的位置是多少
  - 類似 sparse table 預處理的方式
  - 只是這次是從上面往下推（sparse table 預處理是下面往上推）
- 因此總複雜度為 $O(N \log N + M)$，空間複雜度是 $O(N \log N)$

進階線段樹 – 區間查詢與修改的對偶

```
1  int st[maxlg][maxn];
2  void chmax(int l, int r, int val) {
3    // for i \in [l, r), a[i] = max(a[i], val)
4    int lg = __lg(r - l);
5    int len = 1 << lg;
6    st[lg][l] = max(st[lg][l], val);
7    st[lg][r - len] = max(st[lg][r - len], val);
8  }
```

進階線段樹 – 區間查詢與修改的對偶

```
1  void dp(int n) {
2    for (int l = maxlg - 2; l >= 0; l--) {
3      int len = 1 << l;
4      for (int i = 0; i < n; i++) {
5        st[l][i] = max(st[l][i], st[l + 1][i]);
6        if (i + len < n)
7          st[l][i + len] = max(st[l][i + len], st[l +
           ↪  1][i]);
8      }
9    }
10   // ans is st[0][i]
11 }
```

進階線段樹 – 區間查詢與修改的對偶

- 網路上有一個類似的技巧被稱為 Dual Segment Tree，但比較侷限在線段樹而沒有其他推廣
- 事實上這個技巧除了 Sparse Table 外，分塊、樹上倍增法、線段樹都可以用到
- 值得注意的是線段樹一次 push 所有節點的懶標只需要 $O(N)$
  - DFS 一次整棵樹就可以了！

進階線段樹 – 線段樹上的均攤

- 直接來看例題！

題目 (區間取 max、單點查詢)

給定一個長度 N 的序列 $a_1, \ldots, a_N$，接下來有 Q 個操作：

- `A x v` 將 a_x 改為 v。
- `D l r k` 對於所有 $l \le i \le r$，將 a_i 改為 $\lfloor \frac{a_i}{k} \rfloor$。
- `S l r` 查詢區間 $[l, r]$ 的總和。

輸入的所有數字均為正整數。

- $N, Q \le 10^5$
- $1 \le a_i, v, k \le 10^9$

進階線段樹 – 線段樹上的均攤

- 本題出現了一個非常神秘的操作：區間除法取下高斯
- 我們無法快速由分母推出整個區間的總和如何變化，要怎麼處理這樣的問題呢？
- 不是整個區間都是 0，就暴力 DFS 下去！
- 這樣是好的嗎…… (?)
- 亂寫一通就發現 AC 了，怎麼會這樣！

進階線段樹 – 線段樹上的均攤

- 可以發現，每個數字被除了 $O(\log C)$ 次之後就會變成 0（當然，不包含 $k = 1$ 的情況）
- 而每個數字每除一次又最多只會讓時間增加樹高的量級
- 至於單點修改也只會讓除的次數增加 $\log C$ 次
- 因此，總複雜度便是 $O((N + Q) \log N \log C)$
- 除了除法之外，區間開根號、區間取 log 都可以用類似的技巧喔！

進階線段樹 – 搭配離線演算法

- 「離線回答的精髓」：可以事前知道所有詢問的所有參數，並且可以不按順序回答每個詢問
- 什麼意思？來看前面持久化線段樹例題的離線版本

題目 (歷史版本和 – offline)

給定一個長度 N 的序列以及 M 個單點修改，接下來有 Q 次詢問，每次詢問會問你在第 m 次修改後，區間 $[l, r]$ 的總和是多少。

- $N, Q \leq 10^5$

- 為什麼我不要乾脆一邊做修改、等到剛好第 m 次修改之後再回答詢問就好了
- 而這個就是「離線回答的精髓」：可以事前知道所有詢問的所有參數，並且可以不按順序回答每個詢問

進階線段樹 – 搭配離線演算法

- 離線演算法大致上的套路都是把詢問和修改（原本的序列也可以當成是某種修改）一起按照某種順序排序
- 接著一邊處理修改一邊處理詢問
- 並把所有詢問的答案存下來後一次回答全部
- 事實上，許多問題「在線回答」與「離線回答」的難度相差很多，也有很多知名的演算法必須在離線的問題才能使用
- 來看例題吧！

進階線段樹 – 搭配離線演算法

題目 (區間 mex)

給定一個序列 $a_1, a_2, \ldots, a_N$，接下來有 Q 個詢問，每次會給定 l, r，請你回答 $\mathrm{mex}(a_l, a_{l+1}, \ldots, a_r)$。其中 mex 的定義是一個集合中最小未出現的非負整數。

- $N, Q \leq 2 \times 10^5$

比如說，$\mathrm{mex}(1, 3) = 0$, $\mathrm{mex}(0, 1, 3) = 2$

- 題外話：這題可以用莫隊 + 值域分塊在 $O((N+Q)\sqrt{N})$ 做掉，不過不在這堂課的討論範疇（偷偷工商，可以看資芽 2021 年根號 inclass 的投影片喔）
- 好像不太清楚 `pull()` 要怎麼寫耶，怎麼辦？
- 先來考慮一個簡單一點的版本！

進階線段樹 – 搭配離線演算法

題目 (區間 mex — 簡化版)

給定一個序列 $a_1, a_2, \ldots, a_N$，接下來有 Q 個詢問，每次會給定 r，請你回答 $\mathrm{mex}(a_1, a_{l+1}, \ldots, a_r)$。其中 mex 的定義是一個集合中最小未出現的非負整數。

- $N, Q \leq 2 \times 10^5$

比如說，$\mathrm{mex}(1,3) = 0$, $\mathrm{mex}(0,1,3) = 2$

- 現在左界通通變成 1 了，但我們好像還不知道線段樹要怎麼維護？
- 在這之前中，線段樹的葉節點 $[i,i]$ 常常只用來維護 a_i 相關的資訊，讓我們來跳脫這個框架！

進階線段樹 – 搭配離線演算法

題目 (區間 mex — 簡化版)

給定一個序列 $a_1, a_2, \ldots, a_N$，接下來有 Q 個詢問，每次會給定 r，請你回答 $\mathrm{mex}(a_1, a_{1+1}, \ldots, a_r)$。其中 mex 的定義是一個集合中最小未出現的非負整數。

- $N, Q \leq 2 \times 10^5$

- 想想看我們是怎麼做 mex() 運算的
- 假設現在在回答 $[1, r]$：從 $0, 1, \ldots, N$ 開始一路往下看，看哪個數字最早沒出現在 a_1 到 a_r 中
- 也就是說，我們要關心「數值本身」的出現狀況（關心的本質），搞不好線段樹 $[i, i]$ 的葉節點要維護**數字 i 相關的資訊**

進階線段樹 – 搭配離線演算法

題目 (區間 mex — 簡化版)

給定一個序列 $a_1, a_2, \ldots, a_N$，接下來有 Q 個詢問，每次會給定 r，請你回答 $\mathrm{mex}(a_1, a_{1+1}, \ldots, a_r)$。其中 mex 的定義是一個集合中最小未出現的非負整數。

-
- 回到原本的問題，要怎麼快速的判斷一個數字有沒有出現在 a_1 到 a_r
- 觀察一：如果一個數字在序列出現很多次，我們可以**拿最左邊的點當這個數字代表**，我們可以把這些數字單獨拿出來
- $[2, 2, 0, 1, 0]$

進階線段樹 – 搭配離線演算法

- $[2, 2, 0, 1, 0]$
- 把紅色數字拿出來當作資訊用
- 數字 0 最早出現在 a_3，數字 1 最早出現在 a_4，數字 2 最早出現在 a_1，數字 3 最早出現在 a_{N+1}（不存在）
  - 在上面的例子中，$f(0) = 3, f(1) = 4, f(2) = 1, f(3) = N + 1$
- 假設我們用 $f(i)$ 來存這個資訊：$f(i)$ 代表數字 i 在 a 序列中最早出現的位置
- 有上面的資訊之後，要怎麼判斷數字 i 有沒有出現在 a_1 到 a_r 呢？
- `f(i) <= r ? "Yes" : "No"`

進階線段樹 – 搭配離線演算法

- $[2, 2, 0, 1, 0]$
- 假設我們用 $f(i)$ 來存這個資訊：$f(i)$ 代表數字 i 在 a 序列中最早出現的位置
- 有上面的資訊之後，要怎麼判斷數字 i 有沒有出現在 a_1 到 a_r 呢？
- `f(i) <= r ? "Yes" : "No"`
- 會判斷一個數字了，來回到原本的詢問：從 $0, 1, \ldots, N$ 開始一路往下看，看哪個數字最早沒出現在 a_1 到 a_r 中
- **那其實就是找到最小的 i，滿足 $f(i) > r$**
- 二分搜尋！

進階線段樹 – 搭配離線演算法

- $[2, 2, 0, 1, 0]$
- 假設我們用 $f(i)$ 來存這個資訊：$f(i)$ 代表數字 i 在 a 序列中最早出現的位置
- 原本的詢問就等價，**找到最小的 i，滿足 $f(i) > r$**
- 二分搜尋！

```
1  int bs_l = -1, bs_r = n + 1;
2  while (bs_r - bs_l > 1) {
3    int mid = (bs_l + bs_r) >> 1;
4    if (query_max(0, mid) > r) bs_r = mid;
5    else bs_l = mid;
6  }
7  // answer stored in bs_r
```

- query_max(L, R) 是詢問 $f(L)$ 到 $f(R)$ 的最大值，乍看之下可以用 Sparse Table 讓複雜度是 $O(\log N)$

進階線段樹 – 搭配離線演算法

題目 (區間 mex — 中階版)

給定一個序列 $a_1, a_2, \ldots, a_N$，接下來有 Q 個詢問，每次會給定 l, r，請你回答 $\mathrm{mex}(a_l, a_{l+1}, \ldots, a_r)$ ，但題目保證 $l \in [1, 2]$。其中 mex 的定義是一個集合中最小未出現的非負整數。

-
- 在簡化版中，我們可以用二分搜處理所有 $l = 1$ 的狀況
- 那同樣的，我們就可以**把 $l = 1$ 跟 $l = 2$** 的詢問分開，先處理所有 $l = 2$ 再處理所有 $l = 1$（個別二分搜）
- **但從 $l = 2$ 移動到 $l = 1$ 的過程中，我們需要把 $f(a_1) = 1$**
- 所以，維護 $f()$ 的資料結構要支援單點修改，**就不能用 Sparse Table，要用線段樹了**

進階線段樹 – 搭配離線演算法

```cpp
int bs_l = -1, bs_r = n + 1;
while (bs_r - bs_l > 1) {
  int mid = (bs_l + bs_r) >> 1;
  if (query_max(0, mid) > r) bs_r = mid;
  else bs_l = mid;
}
// answer stored in bs_r
```

- 但是這樣，複雜度就變成 $O(\log^2 N)$，有沒有辦法維持一個 log 呢？
- 跟 BIT 的狀況類似，**我們可以在線段樹這個結構上面二分搜！**
- 直接看 code 吧～

進階線段樹 – 搭配離線演算法

```
1  int binsearch(int k, Node *now, int l, int r) {
2    if (l == r)
3      return l;
4    int m = l + (r - l) / 2;
5    if (now->l->mx > k) // 左邊的最大值 > k，代表答案在線段樹的
     ↪  左子樹這
6      return binsearch(k, now->l, l, m);
7    else
8      return binsearch(k, now->r, m + 1, r);
9  }
```

進階線段樹 – 搭配離線演算法

- 來推廣到整個題目吧！
- 我們由 a_N 至 a_1 的順序依序掃過原本的序列，並且將其加入某個資料結構
- 在掃到 a_i 的時候，這個資料結構裡面儲存了包含順便處理左界為 i 的詢問
- 定義 f_j 表示數字 j 在 $[i, N]$ 最前面出現的位置
- 那麼 $\operatorname{mex}(i, r)$ 就是最小的數字使得 $\max_{i \le j \le r}(f_j) > r$
- 所以，我們可以用一個線段樹維護 f_j 的區間最大值，再搭配上二分搜，就能回答所有詢問了
- 二分搜可以直接在線段樹上二分搜，把複雜度壓到 $O((N + Q) \log N)$ 的複雜度
- 這種線段樹，位置 i 是存關於數值 i 的資訊，而不是序列第 i 個位置的資訊，我們常稱為這是「值域線段樹」

進階線段樹 – 區間查詢與修改的對偶

```
1  struct Node {
2    Node *l, *r;
3    int mx;
4    void pull() { mx = max(l->mx, r->mx); }
5  };
```

進階線段樹 – 區間查詢與修改的對偶

```
for (int i = 0; i < Q; i++) {
  queries.emplace_back(l, r, i);
}
sort(queries.begin(), queries.end(), greater<>());
root = build(0, N);
int it = 0;
for (int i = N; i >= 1; i--) {
  if (a[i] <= N)
    edit(a[i], i, root, 0, N);
  while (it < queries.size() && get<0>(queries[it]) ==
  ↪  i) {
    auto [l, r, qid] = queries[it++];
    ans[qid] = binsearch(r, root, 0, N);
  }
}
```

進階線段樹 – 資料結構經典題

- 直接來看例題！

題目 (K-th Number)

給長度 N 的序列 a，及 Q 筆詢問。每筆詢問給 i, j, k，求 $[i, j]$ 區間中第 k 小數。(即把 $[i, j]$ 區間由小到大排序後，第 k 個數)

- $1 \leq N \leq 10^5$
- $1 \leq Q \leq 10^5$
- $1 \leq i \leq j \leq N, k \leq (j - i + 1)$

進階線段樹 – 資料結構經典題

- 這題如果允許離線的話，可以使用整體二分在 $O((Q+N)\log N)$ 或 $O((Q+N)\log^2 N)$ 內做完，詳細部份可以參考這本講義分治這個單元
- 我們可以考慮使用值域線段樹來解決這個問題，線段樹每個位置存這個數字出現了幾次
- 而要找到第 K 小，就可以直接在值域線段樹上面二分搜就好（作法類似 BIT 第 K 小的作法，加上在前面區間 mex 操作在線段樹上二分搜的技巧）

進階線段樹 – 資料結構經典題

- 而現在，我們可以把這樣的值域線段樹外面套上一層「持久化」
- 我們開一個 N 個版本的值域線段樹，第 i 個版本存 a_1 到 a_i 項中每個數字出現過的次數
- 在進行詢問的時候，就可以利用「第 r 個版本」扣掉「第 $l-1$ 個版本」，來獲得區間 $[l, r]$ 的值域的狀況，進而在上面二分搜！
- 直接來看講義的程式碼吧～

樹堆

樹堆

- Treap
- Tree + Heap
- 樹堆是一種二元樹，是 Binary Search Tree + Heap 的綜合題
- 節點有兩個值：pri 跟 key，分別滿足**二元搜尋樹**的性質跟**堆**的性質

樹堆

- 樹性質：
  - 左子樹的 key 都小於根節點的 key
  - 左子樹的 key 都大於根節點的 key
  - 左右子樹都滿足樹性質
- 堆性質：
  - 左子樹的 pri 都小於跟節點的 pri
  - 左子樹的 pri 都小於跟節點的 pri
  - 左右子樹都滿足堆性質
- pri 是用**隨機生成**的，可以讓樹高期望上變成 $O(\log N)$

樹堆

- 兩個最重要的操作：
  - $merge(a, b)$：將兩棵樹堆 a, b 合併爲一棵樹堆，需滿足樹堆 a 的所有 key 值均小於等於樹堆 b 的所有 key 值
  - $split(t, k)$：將樹堆 t 分爲兩棵樹堆，其中第一棵樹堆中的 key 值均小於等於 k，而第二棵樹堆中的 key 值均大於 k

樹堆

```
1  struct Treap {
2    Treap *l, *r;
3    int pri, key, val;
4    Treap(int _val, int _key)
5        : val(_val), key(_key), l(NULL), r(NULL),
         ↪  pri(rand()) {}
6  };
```

樹堆 – merge

- 整個合併過程因爲堆性質的引入而拯救世界。由於**已經保證樹堆 a 的所有 key 值需小於等於樹堆 b 的所有 key 值**，我們只要顧好堆性質即可！因此，整個合併過程可以簡述如下：
- 合併三部曲：
  1. 其中一棵樹爲空，直接返回另一棵
  2. 依照堆性值決定哪個節點當根
  3. 合併尚未合併的那一邊子樹

樹堆 – merge

- 合併三部曲：
  1. 其中一棵樹爲空，直接返回另一棵
  2. 依照堆性值決定哪個節點當根
  3. 合併尚未合併的那一邊子樹

樹堆 – merge

```
1  Treap *merge(Treap *a, Treap *b) {
2    if (!a || !b) return a ? a : b;
3    if (a->pri > b->pri) {
4      a->r = merge(a->r, b);
5      return a;
6    } else {
7      b->l = merge(a, b->l);
8      return b;
9    }
10 }
```

樹堆 – split

- 在分裂的過程中，由於已經**維護好堆性質的部分**，因此，只需要依 key 值來決定要分到哪一棵樹，整個分裂過程簡述如下：
- 分裂三部曲：
  1. 樹爲空，直接返回
  2. 根節點要送給誰（依 key 值決定）
  3. 分裂尚未分裂的那一棵子樹

樹堆 – split

- 分裂三部曲：
  1. 樹爲空，直接返回
  2. 根節點要送給誰（依 key 值決定）
  3. 分裂尚未分裂的那一棵子樹

樹堆 – split

```
1  void split(Treap *t, int k, Treap *&a, Treap *&b) {
2    if (!t) a = b = NULL;
3    else if (t->key <= k) {
4      a = t;
5      split(t->r, k, a->r, b);
6    } else {
7      b = t;
8      split(t->l, k, a, b->l);
9    }
10 }
```

樹堆 – insert, remove

- insert

```
1  Treap *insert(Treap *t, int k) {
2    Treap *tl, *tr;
3    split(t, k, tl, tr);
4    return merge(tl, merge(new Treap(k), tr));
5  }
```

- remove

```
1  Treap *remove(Treap *t, int k) {
2    Treap *tl, *tr;
3    split(t, k - 1, tl, t);
4    split(t, k, t, tr);
5    // return merge(tl, tr);
6    return merge(merge(tl, t->l), merge(t->r, tr));
7  }
```

樹堆 – 例題

- 要怎麼轉換到序列上呢？
- 把 key 當作序列的 index，並且多一些變數維護題目要求的數值
- 直接來看題目！

題目 (區間最大值 Range Maximum Query，RMQ)

一開始給你 n 個數字 $a_1, a_2, \ldots, a_n$，接下來有 m 個操作，操作有兩種：

1. 將 a_p 改為 x。
2. 詢問 $a_l, a_{l+1}, \ldots, a_r$ 中的最大值。

假設所有數字均為正整數

樹堆 – 例題實作

```
1  struct Treap {
2    Treap *l, *r;
3    int pri, key; // key：陣列 index
4    int val, mx; // val: 該 index 的數值，mx：該子樹的最大值
5    Treap() {}
6    Treap(int _key, int _val) : l(NULL), r(NULL),
     ↪  pri(rand()), key(_key), val(_val), mx(_val){}
7  };
8  inline int mx(Treap *t) { return t ? t->mx : 0; }
9  inline void pull(Treap *t) { t->mx = max(mx(t->l),
   ↪  max(t->val, mx(t->r))); }
```

樹堆 – 例題實作

```
1  Treap *merge(Treap *a, Treap *b) {
2    if (!a || !b)
3      return a ? a : b;
4    if (a->pri > b->pri) {
5      a->r = merge(a->r, b);
6      pull(a); // 子樹的值有動過，更新父節點！
7      return a;
8    } else {
9      b->l = merge(a, b->l);
10     pull(b);
11     return b;
12   }
13 }
```

樹堆 – 例題實作

```
1  void split(Treap *t, int x, Treap *&a, Treap *&b) {
2    if (!t)
3      a = b = NULL;
4    else if (t->key <= x) {
5      a = t;
6      split(t->r, x, a->r, b);
7      pull(a); // 子樹的值有動過，更新父節點！
8    } else {
9      b = t;
10     split(t->l, x, a, b->l);
11     pull(b);
12   }
13 }
```

樹堆 – 例題實作

```
1  Treap *t = NULL;
2  for (int i = 1; i <= n; i++) {
3    t = merge(t, new Treap(i, a[i]));
4  }
```

樹堆 – 例題實作

- 查詢最大值

```
1  split(t, l - 1, tl, t);
2  split(t, r, t, tr);
3  printf("%d\n", mx(t));
4  t = merge(merge(tl, t), tr);
```

- 修改

```
1  split(t, p - 1, tl, t);
2  split(t, p, t, tr);
3  t->val = t->mx = x;
4  t = merge(merge(tl, t), tr);
```

樹堆 – 例題

- 還可以加懶標！

題目 (區間最大值 Range Maximum Query，RMQ)

給定一個長度為 N 的正整數序列 $a_1, \ldots, a_N$，接著有 Q 個詢問，每個詢問形如

- `1 l r`，請輸出 $a_l, a_{l+1}, \ldots, a_r$ 當中最大的數字是多少。
- `2 l r d`，讓 $a_l, a_{l+1}, \ldots, a_r$ 全部增加 d。

- $N, Q \leq 5 \times 10^5$
- $d > 0$

樹堆 – 例題實作

```
1  struct Treap {
2    Treap *l, *r;
3    int pri, key, val, big, add;
4    Treap(int _key, int _val): l(NULL), r(NULL),
     ↪  pri(rand()), key(_key), val(_val), big(_val),
     ↪  add(0){}
5  };
6  int Big(Treap *t) {
7    return t ? t->big : -inf;
8  }
9  void pull(Treap *t) {
10   t->big = max(t->val, max(Big(t->l), Big(t->r)));
11 }
```

樹堆 – 例題實作

```
1  void push(Treap *t) {
2    if (t->add == 0)
3      return;
4    if (t->l) {
5      t->l->val += t->add;
6      t->l->big += t->add;
7      t->l->add += t->add;
8    }
9    if (t->r) {
10     t->r->val += t->add;
11     t->r->big += t->add;
12     t->r->add += t->add;
13   }
14   t->add = 0;
15 }
```

樹堆 – 例題實作

```
1  Treap *merge(Treap *a, Treap *b) {
2    if (!a || !b)
3      return a ? a : b;
4    if (a->pri > b->pri) {
5      push(a); // 要前往子樹了，把懶標推下去
6      a->r = merge(a->r, b);
7      pull(a); // 子樹的值有動過，更新父節點！
8      return a;
9    } else {
10     push(b);
11     b->l = merge(a, b->l);
12     pull(b);
13     return b;
14   }
15 }
```

樹堆 – 例題實作

```
1  void split(Treap *t, int x, Treap *&a, Treap *&b) {
2    if (!t)
3      a = b = NULL;
4    else if (t->key <= x) {
5      a = t;
6      push(a); // 要前往子樹了，把懶標推下去
7      split(t->r, x, a->r, b);
8      pull(a); // 子樹的值有動過，更新父節點！
9    } else {
10     b = t;
11     push(b);
12     split(t->l, x, a, b->l);
13     pull(b);
14   }
15 }
```

樹堆 – 例題實作

```
1  int query(int l, int r) {
2    Treap *tl, *tr;
3    split(root, l - 1, tl, root);
4    split(root, r, root, tr);
5    int res = Big(root);
6    root = merge(merge(tl, root), tr);
7    return res;
8  }
```

樹堆 – 例題實作

```
1  void add(int l, int r, int d) {
2    Treap *tl, *tr;
3    split(root, l - 1, tl, root);
4    split(root, r, root, tr);
5    root->val += d;
6    root->big += d;
7    root->add += d;
8    root = merge(merge(tl, root), tr);
9  }
```

樹堆 – 維護 size

- 維護一個節點底下有多少個節點！

```
1  struct Treap {
2    Treap *l, *r;
3    int pri, size, val;
4    Treap(int _val) : val(_val), size(1), l(NULL),
     ↪  r(NULL), pri(rand()) {}
5  };
6  int Size(Treap *t) { return t ? t->size : 0; }
7  void pull(Treap *t) { t->size = 1 + Size(t->l) +
   ↪  Size(t->r); }
```

樹堆 – 維護 size

- split 就多了一種方法：現在我們可以支援把前 k 個東西切下來！

```
1  void split(Treap *t, int k, Treap *&a, Treap *&b) {
2    if (!t)
3      a = b = NULL;
4    else if (Size(t->l) + 1 <= k) {
5      a = t;
6      split(t->r, k - Size(t->l) - 1, a->r, b);
7      pull(a);
8    } else {
9      b = t;
10     split(t->l, k, a, b->l);
11     pull(b);
12   }
13 }
```

樹堆 – 維護 size

- 有了 size，在一般的序列我們就可以直接把 key 丟掉了！
- 其實 key 根本就只是**它在整個樹堆裡有幾個人比他小嘛！**
- **反正就算沒有 key，這棵樹總是有個中序走訪，我們依然可以將樹對應到序列上**
- 放下 key 人生就會豁然開朗

題目 (區間最大值 Range Maximum Query，RMQ)

給一個長度為 N 的序列 $a_1, a_2, \ldots, a_N$ 以及 Q 筆操作，操作內容形式如下：

- `1 l r`：把 a_l 到 a_r 區間反轉
- `2 l r`：求出 a_l 到 a_r 的和

樹堆 – 例題講解

- 抱著 key 不放，會讓序列 index 難以維護
- 用 size 讓人生豁然開朗！

```
1  struct Treap {
2    Treap *l, *r;
3    ll pri, size, sum, val;
4    bool tag; // 用來維護子樹是否需要翻轉（swap）
5    Treap(int _val) : l(NULL), r(NULL), pri(rand()),
     ↪  size(1), sum(_val), val(_val), tag(false) {}
6  };
```

樹堆 – 例題講解

```
1  void push(Treap *t) { // 可以把 tag 想成要把底下的區間反轉幾次
2    if (!t->tag) return;
3    if (t->l) { // 把 tag 下推到左子樹
4      swap(t->l->l, t->l->r);
5      t->l->tag ^= t->tag;
6    }
7    if (t->r) {
8      swap(t->r->l, t->r->r);
9      t->r->tag ^= t->tag;
10   }
11   t->tag = false;
12 }
```

樹堆 – 例題講解

```
1  void reverse(int l, int r) {
2    Treap *tl, *tr;
3    split(root, l - 1, tl, root);
4    split(root, r - l + 1, root, tr);
5    swap(root->l, root->r); // 區間反轉，其實就是把左右子樹對調
6    root->tag = true; // 紀錄 tag，代表往下的子樹也都需要左右對
       ↪ 調
7    root = merge(merge(tl, root), tr);
8  }
```

樹堆 – 知道自己在哪

- 有了 size 之後，感覺就可以做很多壞事，比如說：
- 給定一個 Treap 節點，詢問這個節點在中序的位置（也就是這個節點的 size）
- 要怎麼做呢？

樹堆 – 知道自己在哪

```
1  struct Treap {
2    Treap *lc, *rc, *fa;
3    int sz, pri;
4    Treap(): lc(NULL), rc(NULL), fa(NULL), sz(1),
     ↪  pri(rand()){};
5  };
6
7  int Size(Treap *t) {
8    return t ? t->sz : 0;
9  }
```

樹堆 – 知道自己在哪

```
1  void pull(Treap *t) {
2    t->fa = NULL;
3    t->sz = 1 + Size(t->lc) + Size(t->rc);
4    if (t->lc) {
5      t->lc->fa = t;
6      t->sz += t->lc->sz;
7    }
8    if (t->rc) {
9      t->rc->fa = t;
10     t->sz+= t->rc->sz;
11   }
12 }
```

樹堆 – 知道自己在哪

```
1  int get_size(Treap *node) {
2    int ret = Size(node->lc) + 1;
3    while (node->fa != NULL) {
4      if (node->fa->rc == node) {
5        ret += 1 + Size(node->fa->lc);
6      }
7      node = node->fa;
8    }
9    return ret;
10 }
```

樹堆 – 總結

- 一種利用隨機性質的平衡二元樹
- 可以解決線段樹無法處理的區間翻轉、... 等等操作
- 操作都是建議在 merge、split 上面
- warning：Treap 的常數又比線段樹大了一些，如果遇到卡常題目請斟酌使用

感謝大家

感謝大家的聆聽～有問題歡迎透過各種管道發問ㄛ