Fenwick tree (Binary indexed tree)

線段樹可以動態的在 $O(\log n)$ 時間內詢問一個區間內的元素總合，以及修改其中一個元素的值。然而線段樹這個資料結構有點大，且遞迴時也有些複雜，有沒有另外的資料結構可以處理這種問題呢？我們發現，「區間 $[a, b]$ 的總合」可以轉成「區間 $[1, b]$ 的總合」減掉「區間 $[1, a-1]$ 的總合」，因此我們可以把問題簡化成：如何快速的求得一個陣列的前綴和，以及更改其中一個元素的值？

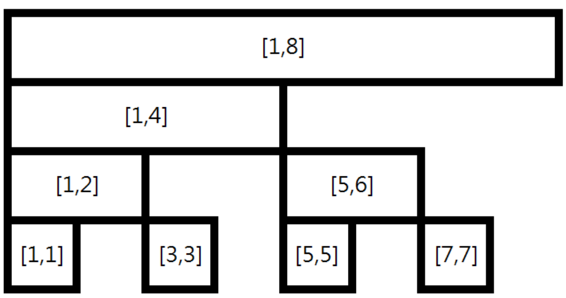
Fenwick tree 是一個處理前綴時很有效率的資料結構，他也被稱為 Binary indexed tree，或簡稱 BIT，在中國被稱做樹狀數组。以下我們都簡稱它為 BIT。和線段樹相同，對於一個長度為 n 的陣列，BIT 可以在 $O(n)$ 的時間初始化，在 $O(\log n)$ 時間詢問一個前綴的訊息 (例如前綴和)，以及在 $O(\log n)$ 的時間修改其中一個值。雖然複雜度和線段樹相同，但 BIT 的時間常數比線段樹小得多 (時間複雜度會把常數拿掉，但實際上還是有影響的)，空間也比較小 (只需要多開一個大小恰好為 n 的陣列即可)，程式碼也非常精簡 (初始化 + 修改 + 詢問，全部約 20 行)。一般來說，如果問題的性質可以使用 BIT，效率會和使用線段樹有明顯的差別。當然，BIT 的缺點就是有些問題無法轉為前綴之間的運算，例如區間 $[a, b]$ 的最大值就無法由區間 $[1, b]$ 和 $[1, a-1]$ 的最大值算出，這時候就無法使用它了。以下將詳細介紹 BIT 的操作。

首先，我們先介紹一個函數：$\text{lowbit}(x)$，表示 x 在二進位表示時，最接近 LSB(Least Significant Bit)，也就是最靠右邊的 1 所對應的值。例如十進位下的數字 $6_{(10)}$，在二進位的寫法是 $110_{(2)}$，其中的兩個 1 分別表示 2^2 和 2^1，因此 $\text{lowbit}(6) = 2^1 = 2$。同理，$20_{(10)} = 10100_{(2)}$，所以 $\text{lowbit}(20) = 2^2 = 4$。lowbit 函數可以用位元運算在 $O(1)$ 時間內得到，在作業中會再多做說明。

和線段樹一樣，BIT 先記錄了一些區間的訊息，並由這些預先處理過的區間拼出真正詢問的區間，而在單點修改時，也要另外修改包含該點的區間。假設原始陣列長度為 n，且索引值為 1 至 n(1-based)，線段樹會記錄大約 $2n$ 至 $4n$ 個區間 (這也和實作方法有關)，而 BIT 只需記錄恰好 n 個區間。BIT 的這 n 個區間的右界即是 1 到 n，若一個區間的右界為 x，其左界就是 $x - \text{lowbit}(x) + 1$。意即，BIT 的區間集合為

$$\{\ [x - \text{lowbit}(x) + 1, x] \mid 1 \le x \le n\ \}$$

畫成圖之後可以發現，BIT 其實是棵刪除掉一些區間的線段樹，如下圖所示：



因為每個範圍的右界都不同，我們可以只使用右界來表示這些區間，並以 $\text{range}(x)$ 表示，也就是

$$\text{range}(x) = [x - \text{lowbit}(x) + 1, x]$$

以下我們用前綴和當例子：已知一個長度為 n 的陣列 `arr`，我們希望建立一顆 BIT 的陣列 `bit`，並滿足 $\texttt{bit}[x]$ 為 `arr` 陣列中相對應範圍的元素和，即：

$$\texttt{bit}[x] = \sum_{i \in \text{range}(x)} \texttt{arr}[i] = \sum_{i=x-\text{lowbit}(x)+1}^{x} \texttt{arr}[i]$$

定義完 $\text{range}(x)$ 和 $\texttt{bit}[x]$ 之後，我們討論三種操作：詢問，單點修改，以及初始化時，如何快速的得到答案或完成操作。

1. 詢問前綴和
   詢問 $[1, x]$ 的區間和很容易，因為該區間可以拆成 $[1, x-\text{lowbit}(x)]$ 和 $[x-\text{lowbit}(x)+1, x]$ 兩個區間和的加總。區間 $[x - \text{lowbit}(x) + 1, x]$ 的和就是 $\texttt{bit}[x]$，另外一個區間則遞迴處理。由於 $x - \text{lowbit}(x)$ 在二進位表示法中會比 x 少一個 1，因此最多遞迴 $\lceil \log_2 x \rceil$ 次之後就會終止，詢問的複雜度也就是 $O(\log n)$。(雖然概念上另外一個區間是遞迴處理，實作時只要使用迴圈就可以了，這也是 BIT 常數比較小的原因之一。)

```
int query(int x) {
    int sum = 0;
    for(int i = x; i > 0; i -= lowbit(i))
        sum += bit[i];
    return sum;
}
```

2. 單點修改
   將 $\texttt{arr}[x]$ 的值增加 val 之後，`bit` 陣列中所有對應區間包含 x 的值都要改變。由 BIT 的結構圖可以觀察到，需要更新的區間為 $\text{range}(x), \text{range}(x + \text{lowbit}(x)), \cdots$。因為每往上遞迴一次，對應區間大小就會變為原本的 2 倍，因此最多遞迴 $\lceil \log_2 n \rceil$ 次之後就會終止，單點修改的複雜度也就是 $O(\log n)$。

```
void update(int x, int val) {
    for(int i = x; i <= n; i += lowbit(i))
        bit[i] += val;
}
```

3. 初始化
   和詢問的操作類似，我們可以使用已經計算好的 $\texttt{bit}[x]$ 值來計算未知的 $\texttt{bit}[x]$ 值。令 x 由小到大計算 $\texttt{bit}[x]$，由定義，

$$\texttt{bit}[x] = \sum_{i=x-\text{lowbit}(x)+1}^{x} \texttt{arr}[i] = \texttt{arr}[x] + \sum_{i=x-\text{lowbit}(x)+1}^{x-1} \texttt{arr}[i]$$

除了 $\texttt{arr}[x]$ 這項以外，區間內其餘元素的和可以用前面已經算好的 bit 較快的得到答案，如 $\texttt{bit}[8] = \texttt{arr}[8] + \texttt{bit}[7] + \texttt{bit}[6] + \texttt{bit}[4]$。此做法雖然看起來計算單一個 $\texttt{bit}[x]$ 的時間複雜度不是 $O(1)$，但是總時間複雜度計算之後是 $O(n)$。演算法程式碼如下：

```
void init(int n) {
    for(int x = 1; x <= n; ++x) {
        bit[x] = arr[x];
        int y = x - lowbit(x);
        for(int i = x-1; i > y; i -= lowbit(i))
            bit[x] += bit[i];
    }
}
```

另一個做法則是計算完 $\texttt{bit}[x]$ 之後，「主動」往上更新上一層的值。該做法和上一個做法原理完全一樣，由程式碼就可以輕易看出這是個 $O(n)$ 的演算法：

```
void init2(int n) {
    for(int x=1; x<=n; ++x)
        bit[x] = 0;
    for(int x=1; x<=n; ++x) {
        bit[x] += arr[x];
        int y = x + lowbit(x);
        if(y <= n) arr[y] += arr[x];
    }
}
```

習題

1. (10 pts) 請證明在二補數系統下 (即 $-x \equiv \sim x + 1$)，$\text{lowbit}(x) = x \& (-x)$。(假設 $x > 0$)。

2. 有一個初始化 BIT 的方法是一開始先把 `bit` 陣列歸零，並對於 `arr` 陣列中的每個元素都呼叫一次 update(x, `arr`$[x]$)，如以下程式碼所示：

```
void init3(int n) {
    for(int x=1; x<=n; ++x)
        bit[x] = 0;
    for(int x=1; x<=n; ++x)
        update(x, arr[x]);
}
```

這個演算法的時間複雜度顯然是 $O(n \log n)$，然而這只是一個上界，實際上也許並不會那麼差，因為更新時不一定會改到滿滿的 $\log n$ 個區間，也許均攤後也是 $O(n)$ 呢！對此我們需要更細的計算一下，以下將證明該做法的複雜度是 $\Omega(n \log n)$，也就是均攤之後還是比較差。

為了方便起見，先假設 $n = 2^m$，其中 m 為非負整數。令 $f(m)$ 表示當 `arr` 長度為 2^m 時，用這個初始化做法需要更新幾次區間，也就是操作次數。由 BIT 的結構可以發現，大小為 2^m 的 BIT 最上層一定是區間 $[1, 2^m]$，且不論更改哪個位置的值，這個區間一定會被更改到。將最上層的區間拿走之後，下面剩下的剛好是大小為 $2^{m-1}, 2^{m-2}, \cdots, 2^1, 2^0$ 的 BIT，於是就得到

$$f(m) = 2^m + \sum_{k=0}^{m-1} f(k), \quad f(0) = 1$$

(a) (10 pts) 請證明：$\sum_{i=0}^{n} i \cdot 2^{i-1} = (n-1) \cdot 2^n + 1$。

(b) (20 pts) 請證明：$f(m) \geq m \cdot 2^{m-1}$, for $m \geq 0$。(即：$f(m) \geq \frac{n}{2} \cdot \log_2 n$, for $n = 2^m, m \geq 0$)

3. 考慮一個無修改的區間最大值問題，雖然一段區間的最小值無法由兩個前綴的最小值得到，但是有個人提出了以下使用 BIT 的作法：

(1) `bit`$[x]$ 存的值為 range(x) 中的最大值

(2) 使用 query(a, b) 遞迴詢問一段區間 $[a, b]$ 的最大值，實作方式為

$$\text{query}(a,b) = \begin{cases} -\infty & , \text{ if } a > b \\ \max(\texttt{bit}[b], \text{query}(a, b - \text{lowbit}(b))) & , \text{ if } b - \text{lowbit}(b) + 1 \geq a \\ \max(\texttt{arr}[b], \text{query}(a, b-1)) & , \text{ otherwise} \end{cases}$$

對於該做法，請回答下列問題：

(a) (10 pts) 請給出一組會讓時間複雜度超過 $O(\log n)$ 的詢問。

(b) (15 pts) 請問該做法中，詢問的時間複雜度為何？

4. 給定原始陣列 `arr`，我們定義一個差分陣列 `dif`，滿足

$$\texttt{dif}[x] = \begin{cases} \texttt{arr}[1] & \text{, if } x = 1 \\ \texttt{arr}[x] - \texttt{arr}[x-1] & \text{, if } x \neq 1 \end{cases}$$

接著再定義另外一個陣列 `dif2`，滿足

$$\texttt{dif2}[x] = \texttt{dif}[x] \times x$$

並且定義以下函式，時間複雜度皆為 $O(\log n)$，可以想像內部就是用 BIT 實作的：

- $\text{query}(\texttt{dif}, x)$：回傳 $\sum_{i=1}^{x} \texttt{dif}[i]$ 的值
- $\text{query}(\texttt{dif2}, x)$：回傳 $\sum_{i=1}^{x} \texttt{dif2}[i]$ 的值
- $\text{update}(\texttt{dif}, x, \text{val})$：將 $\texttt{dif}[x]$ 的值加上 val
- $\text{update}(\texttt{dif2}, x, \text{val})$：將 $\texttt{dif2}[x]$ 的值加上 val

請回答以下問題：

(a) (10 pts) 請問如何使用給定的函式，在 $O(\log n)$ 時間內得到 $\texttt{arr}[x]$ 的值？

(b) (15 pts) 現在將 `arr` 陣列中的一段區間 $[a, b]$ 內的數都加上 val，請問如何在 $O(\log n)$ 時間內，使用給定的函式改變 `dif` 和 `dif2` 陣列以滿足定義？

(c) (15 pts) 請以 `dif` 陣列表示 `arr` 陣列的前綴和，也就是 $\sum_{i=1}^{x} \texttt{arr}[i]$。(答案中可以有很多項 $\texttt{dif}[i]$，甚至包含 $\sum$，只要能表示就可以了。)

(d) (15 pts) 請使用給定的函式，在 $O(\log n)$ 時間內得到 $\sum_{i=1}^{x} \texttt{arr}[i]$。