實戰營進階講義

第零章 前言

進階講義主要涵蓋一些較為複雜、或應用情境相對少見的演算法與資料結構。這些主題在實務中出現頻率不高，但在 LeetCode 或相關競賽題目中仍有可能遇到。學員可以依照自身需求與學習進度，決定要深入到什麼程度。大多數情況下，只要了解當遇到這類需求時，有對應的演算法或資料結構能高效解決問題即可。

第壹章 線段樹(Segment tree)

在課程講義的 **Prefix Sum** 章節中，我們已經學到如何快速計算區間總和。Prefix Sum 的確能有效解決「區間加總」的問題，但它也存在一些侷限。例如，若我們需要查詢區間的最大值或最小值，Prefix Sum 就無法幫上忙，因為最大值/最小值無法像加總一樣，透過簡單的扣除來計算。

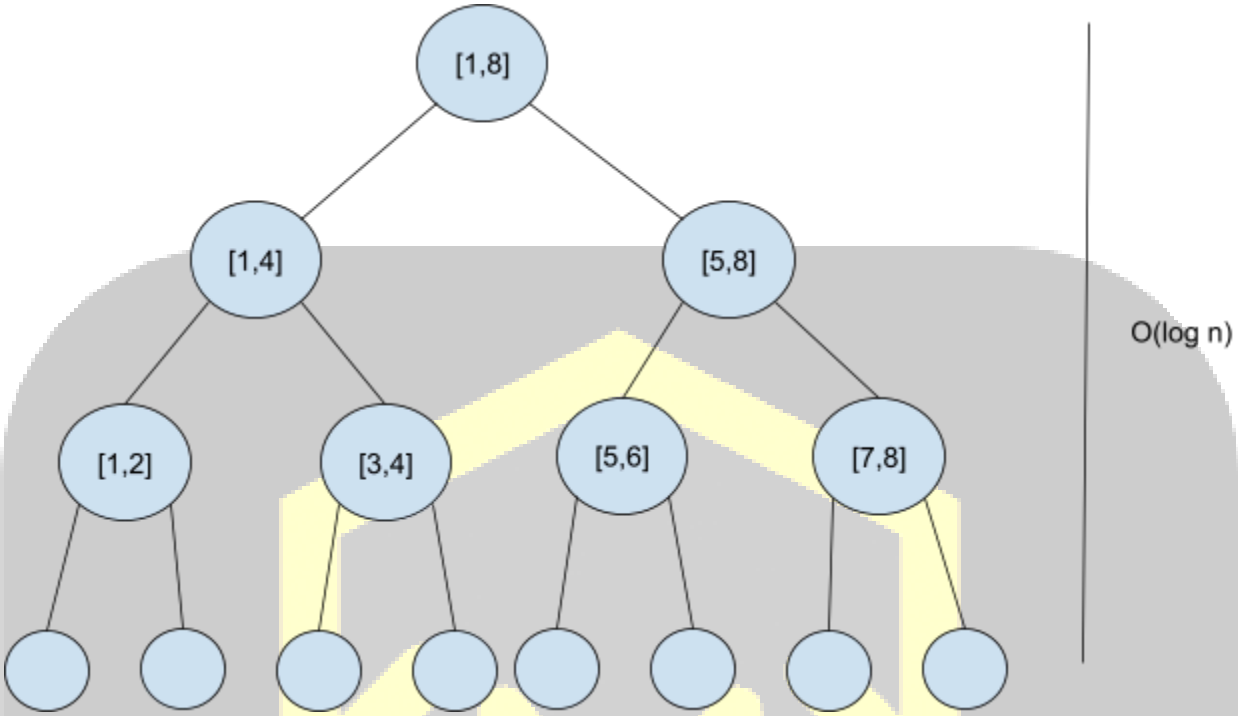
為了解決這個問題，我們可以換個角度思考：既然無法用「扣」的方式，那麼是否能將陣列拆分成數個區間段落，並在查詢時把相關段落的資訊「合併」起來？例如，雖然「區間最大值」不能用扣法計算，但卻可以透過「合併每個段落的最大值」來得到結果。這種想法正是 線段樹 **(Segment Tree)** 的基礎。

線段樹是一種樹狀結構，能在 **O(log n)** 的時間內完成以下操作：

1. 區間查詢：快速得到某段區間的資訊（例如區間總和、最大值、最小值等）。
2. 單點更新：若陣列中某個位置的值改變，也能在 O(log n) 的時間內完成更新，並確保後續的區間查詢結果正確。

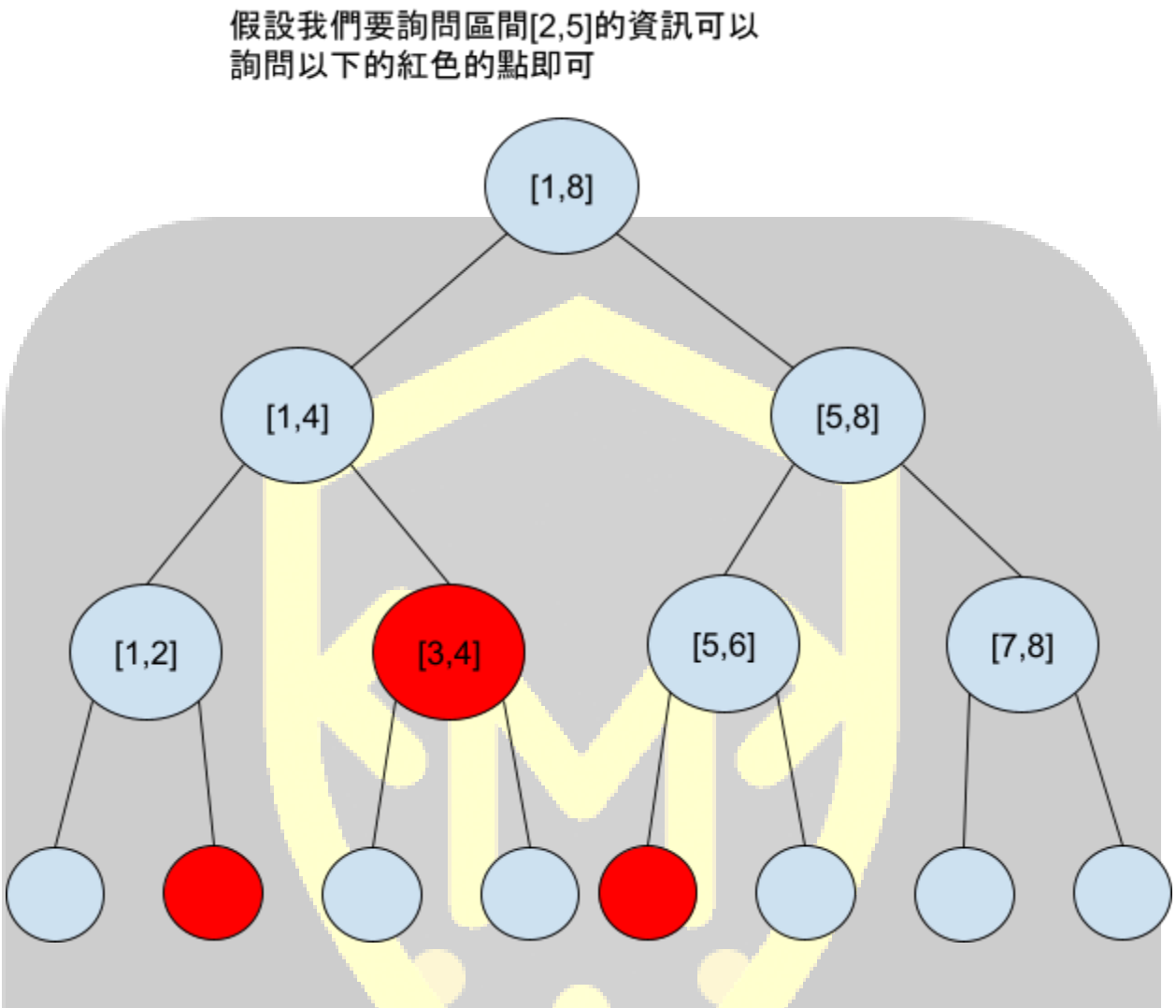
線段樹的實作思路

1. 建構 **(Build)**
    - 將原始陣列不斷劃分為左右兩半，直到劃分到單一元素為止。
    - 每個節點會存放一個「區間」的資訊，例如 `[L, R]` 區間的總和或最大值。
    - 可以看到每層節點數量翻倍，但儲存的資料量是上一個節點的一半，直到最後一個節點只儲存單一一個節點的資訊，因此樹高最多O(log n)



2. 查詢 **(Query)**

   - 若所查詢的區間剛好與節點的區間對應，直接回傳節點值。
   - 若不完全重疊，則往子節點遞迴查詢，並將結果合併。
   - 可以發現在每一層我最多只需要詢問兩個點的資料即可，若存在一個層需要問到三個點，中間的那個點一定可以跟左邊的點或是右邊的點合併，由他們的上一層節點提供答案



3. 更新 **(Update)**
    - 當陣列中某個元素改變時，沿著樹往下更新相關節點。
    - 由於每一層最多只需處理一個節點，因此更新時間也是 O(log n)。

總結來說，線段樹是一種能兼顧「高效查詢」與「快速更新」的資料結構，非常適合用來解決涉及 區間操作 的問題。

以下程式碼使用類似link list的方式實作，每一個節點會有兩個link node(left, right)，分別代表他的左子樹及右子樹

```
class Node {
     Node left, right;
     int leftBound, rightBound;
     int maxVal;
```

```
        Node(int leftBound, int rightBound) {
            this.leftBound = leftBound;
            this.rightBound = rightBound;
            left = right = null;
            maxVal = 0;
        }
    }

    Node root;

    //藉由下方節點的資訊來更新當前節點的資訊
    private void pull(Node n) {
        n.maxVal = Math.max(n.left.maxVal, n.right.maxVal);
    }

    private void build(Node n, int[] v) {
        if (n.leftBound == n.rightBound) {
            // 葉節點對應原陣列的一個值
            n.maxVal = v[n.leftBoun];
            return;
        }
        int mid = (n.leftBound + n.rightBound) / 2;
        n.left = new Node(n.leftBound, mid);
        n.right = new Node(mid + 1, n.rightBound);
        build(n.left, v);
        build(n.right, v);
        pull(n);
    }

    private int queryMax(Node n, int l, int r) {
        // 因為這個節點完全在我們詢問的區間內，因此回傳這個節點的答案
        if (n.leftBound >= l && n.rightBound <= r) {
            return n.maxVal;
        }
        // 因為這個節點完全在我們詢問的範圍外面，因此回傳一個最小值以避免影響答
案
        if (n.rightBound < l || n.leftBound > r) {
            return Integer.MIN_VALUE;
        }
        // 區間部分重疊，往子節點查詢
        return Math.max(query(n.left, l, r), query(n.right, l, r));
    }
```

第貳章 Floyd–Warshall 演算法

用來求 所有點對之間最短路徑 **(All-Pairs Shortest Path)** 的經典 DP 演算法。

核心思想：

逐步把節點 k 當成「中繼點」去嘗試更新每對 (i, j) 的最短路徑。

即：

```
dis[i][j] = Math.min(dis[i][j], dis[i][k] + dis[k][j]);
```

該演算法整個實作非常簡單，只需要三層迴圈即可

```
for (int k = 0; k < n; k++) {        // 中繼點
    for (int i = 0; i < n; i++) {    // 起點
        for (int j = 0; j < n; j++) { // 終點
            dis[i][j] = Math.min(dis[i][j], dis[i][k] + dis[k][j]);
        }
    }
}
```

為什麼 k 必須放在最外層？

Floyd–Warshall 是一種 DP：

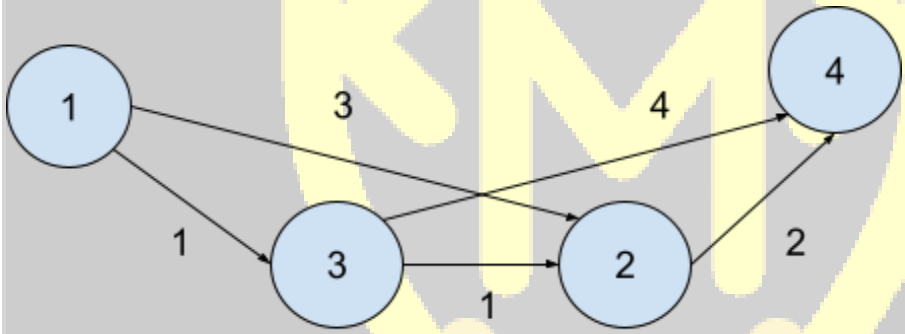
定義 dp 的 state `dp[i][j][k]` 表示：

只允許用節點 **0~k** 當中繼點(也就是這條路徑中非頭尾的點)時，**i → j** 的最短距離

transition：

```
dp[i][j][k] = min(
    dp[i][j][k-1],                    // 不用 k 當中繼點
    dp[i][k][k-1] + dp[k][j][k-1] // 使用 k 當中繼點
)
```

這時可能會想說若i->k的最短路目前還不是最短，可能之後可以藉由i->x->k去更新，那可以發現i->k->j這個原先的最短路也可以藉由i->x->j去做更新，因為x->j時假設他的最短路x->k->j，那他們就已經在前面拿k當中繼點時更新過了。



以上面的例子來看
我們窮舉到2的時候會更新
dp[1][4][2]=5
dp[3][4][2]=3
這時當我們窮舉3時
dp[1][4][3]就會使用到dp[1][3][2]跟dp[3][4][2]，而dp[3][4][2]在使用2當中繼點時已經更新成最佳解(包含2的路)了

因此，外層一定是 k代表我們一輪一輪加入新的“中繼點候選”。

第二層的 i 及第三層的 j

當固定 k 時，我們要嘗試：

所有起點 **i**，所有終點 **j** 看看能不能利用 k 讓路徑更短。

這就是第二、第三層迴圈的來源：

- 第二層：遍歷所有的 起點 **i**
- 第三層：遍歷所有的 終點 **j**

而實際在寫該演算法的時候，可以不用寫出第三維的陣列，原因很簡單，因為從transition可以觀察到 dp[i][j][k] 當我們 k 往後到 k+1 時，就用不到了，而我們可以保證dp[i][j]一定不會比原本ij的最短路短。且上述的三維 dp 最終可得到正確答案，因此就算我們省掉第三維陣列直接在前兩維的 dp state 更新，該陣列也會得到最終答案。