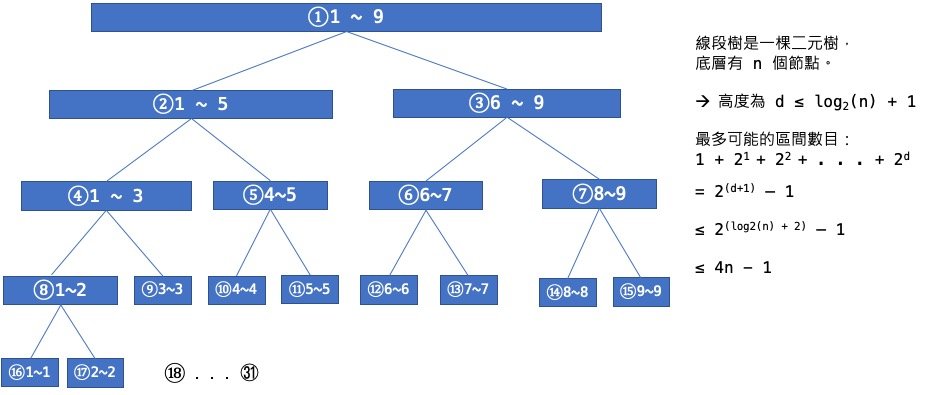
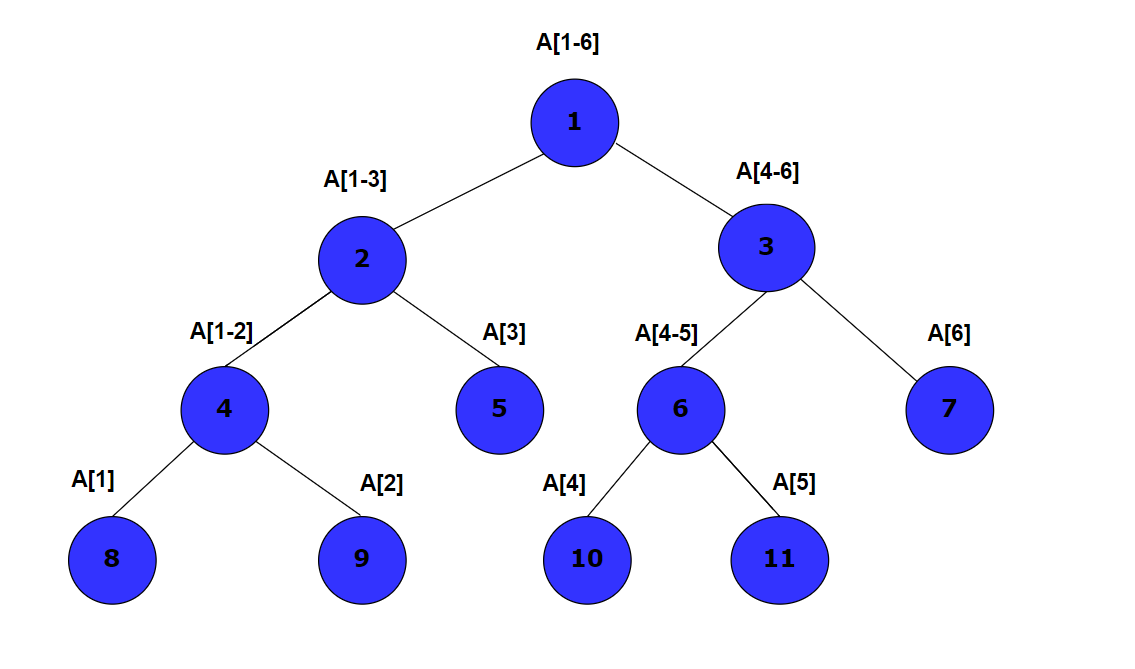
# 線段樹Segment Tree

## 概念

先將答案分塊預先計算好，每次查詢至多遞迴LogN層，達到經過O(N)預處理後，每次查詢O(LogN)的複雜度





## **運用**

主要解決**RMQ**問題

**RMQ** : Range Maximum/Minimum Query

題目多次查詢區間 [ L , R ]的資訊

要查詢的資訊需滿足「結合律」。

**可以運用的場合：**

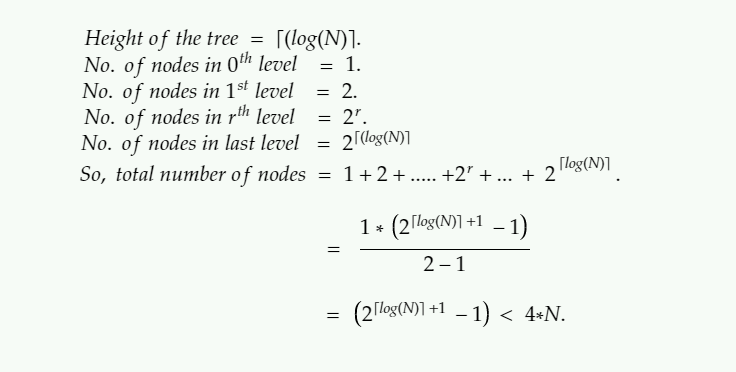
* 區間和
* 區間最大
* 區間最小
* 區間XOR
* 區間GCD

## 架構

### 空間複雜度

陣列版本需要開 **4倍** 記憶體空間

證明如下：



屬於線性空間複雜度，4×Size仍是O(N)空間雜度

### 時間複雜度

因為每次都切 2 塊 ， 直到區塊大小為 1 ， 所以會有 logN 層

而因為最多遞迴 logN 層查詢 ， 每次查詢或修改的時間複雜度都是 O(logN)

與 naive 作法相比（ 每次查詢都需要O(N)

時間複雜度從O(Q×N)下降為O(Q×LogN)

## 實做

### 細節

有很多版本的線段樹，寫法因人而異

像是除了陣列版本還有指標版本的線段樹（ 應該說最早的線段樹是用指標來實做，但是因為方便可以使用陣列的方式去代 new & delete 的操作，但是因為線段樹是既有的資料結構，所以以偽線段樹稱之。雖然大家已將兩個東西混為一談了，只會說「指標版本」或「陣列版本」的線段樹）

還有 0-based index 或 1-based index

兩個者個差別是：

* 0-based index

0-based index 的 left-child-index 會是： i\*2+1

0-based index 的 right-child-index 會是： i\*2+2

* 1-based index

1-based index 的 left-child-index 會是： i\*2

1-based index 的 right-child-index 會是： i\*2+1

查詢範圍也分成：

左閉右閉： [L,R]

左閉右開： [L,R)

都跟二分搜一樣，大家寫法不盡相同，細節有注意到就好

### 預先配置記憶體

*// original array*

int arr[MAX\_N];

*// segment tree*

int T[MAX\_N<<2];

MAX\_N 是題目限制的最大範圍

T 陣列存的是線段樹的節點資訊，透過先前的證明得知需要開 **4倍** 記憶體防止溢位

（ MAX\_N<<2 代表MAX\_N\*4

### 子節點編號 child index

這邊實做的是 1-based index 陣列版本的線段樹

*// left child idx*

inline int L\_idx(int i){

return i<<1;

}

*// right child idx*

inline int R\_idx(int i){

return i<<1|1;

}

其中 i<<1 代表 i\*2，而 i<<1|1 則是 i\*2+1 ，是透過 bitwise operation 稍微加快運算速度

### 建構 Build

void Build(int idx,int L,int R){

if(L==R){

T[idx] = arr[L];

*// cin>>T[ idx ];*

return ;

}

int mid = (L+R)>>1;

Build( L\_idx(idx) , L , mid );

Build( R\_idx(idx) , mid+1,R );

*// pull*

T[ idx ] = max( T[ L\_idx(idx) ] , T[ R\_idx(idx) ] );

}

一直遞迴將區間大小為 1 的 base case填好，並記得將子節點的資訊紀錄到當前節點(其中標注pull的部份)

遞迴參數中L R 紀錄的是當前區間的範圍（ 代表現在這個區間就是[L,R]

**左子區間**紀錄的是 **[L,mid]** ，**右子區間**紀錄的是 **[mid+1,R]**

另外可以看到註解的 cin>>T[ idx ]，也可以直接呼叫 Build 將資料讀入線段樹的節點中，可以少開原先的 arr 陣列

使用方式：

Build(1,1,n);

*// Build( idx , L ,R )*

因為實做的是 1-based index，所以一開始的Node index是1，而Left也是1，Right則是測資的範圍

### 查詢 Query

這邊以區間最大值為例

int Query(int idx,int L,int R,int Q\_L ,int Q\_R ){

*// in current range*

if( Q\_L<=L && R<=Q\_R ) return T[ idx ];

*// out of range*

if( L>Q\_R || R<Q\_L ) return -INF;

*// recursion down*

int mid = (L+R)>>1;

return max( Query(L\_idx(idx) , L ,mid ,Q\_L , Q\_R ) ,

Query(R\_idx(idx) , mid+1 ,R ,Q\_L ,Q\_R ) );

}

遞迴參數中：

* idx： 當前線段樹節點編號
* L & R：當前區間的左界與右界
* Q\_L & Q\_R：查詢的左界與右界

在遞迴過程中，會遇到以下**3種**情況：

1. 查詢範圍在區間範圍內

if( Q\_L<=L && R<=Q\_R ) return T[ idx ];

從線段來看：

------{Q\_L----[L------R]------Q\_R}------

因為節點代表的整個區間都在查詢的範圍內，所以可以直接回傳當前節點的資訊

1. 查詢範圍不再區間內

if( L>Q\_R || R<Q\_L ) return -INF;

也就是以下這 2 種情況：（ 用線段來表示

* ---{Q\_L----Q\_R}---[L----R]-----
* ---[L----R]-----{Q\_L----Q\_R}---

查詢的範圍和當前的區間沒有交集，所以不需要再繼續遞迴

1. 查詢範圍與區間有部份交集

*// recursion down*

int mid = (L+R)>>1;

return max( Query(L\_idx(idx) , L ,mid ,Q\_L , Q\_R ) ,

Query(R\_idx(idx) , mid+1 ,R ,Q\_L ,Q\_R ) );

在排除以上 2 種情況後，只剩下部份有交集的情況，所以就向左、右子節點遞迴查詢，得出答案

用線段來表示:

* ---{Q\_L-----[L---Q\_R}---R]-----
* ---[L----{Q\_L---R]------Q\_R}---

### 單點修改 Update

void Update(int idx,int L,int R ,int pos ,int val){

if( L==R ){

T[ idx ] = val;

return ;

}

int mid = (L+R)>>1;

*// in left part*

if( pos<= mid ){

Update( L\_idx(idx) , L ,mid , pos ,val );

}

else{ *// right part*

Update( R\_idx(idx) , mid+1 , R , pos ,val );

}

*// pull*

T[ idx ] = max( T[ L\_idx(idx) ] , T[ R\_idx(idx) ] );

}

遞迴參數中：

* idx： 當前線段樹節點編號
* L & R：當前區間的左界與右界
* pos：要修改的位置
* val：要修改的值

與 Build() 的過程相似，但是在向下遞迴的過程中，需要檢查要修改的位置是在左節點還是右節點，並記得將子節點的資訊更新到當前節點中(其中標注pull的部份)

## 其他

這邊實做的是單點修改的陣列版本

另外可以使用指標實做還有區間修改、動態開點、高維線段樹等版本（ 但因為較複雜，這邊不進行介紹

而其他與RMQ相關的資料結構還有Binary Index Tree、Sparse Table等

有不同限制與優缺點，有興趣可以額外研究

## 例題

[區間極值查詢](https://zerojudge.tw/ShowProblem?problemid=d539)

[區間 XOR + 單點修改](https://zerojudge.tw/ShowProblem?problemid=c651)

### Referece

（ Baltic OI 2001: Mars Maps ）

<https://www.oi.edu.pl/static/attachment/20110713/boi-2001.pdf>

<https://web.ntnu.edu.tw/~algo/Sequence2.html#3>

<https://cp.wiwiho.me/segment-tree/>

<https://yuihuang.com/segment-tree/>

<https://oi-wiki.org/topic/rmq/>

<https://oi-wiki.org/ds/seg/>

<https://medium.com/nybles/understanding-range-queries-and-updates-segment-tree-lazy-propagation-and-mos-algorithm-d2cd2f6586d8>