Advanced Competitive Programming

國立成功大學ACM-ICPC程式競賽培訓隊
nckuacm@imslab.org

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

Week 9
Data Structure

Segment Tree, Binary Indexed Tree

先看個題目：題目 #1

- 給定一初始序列，並有 Q 筆查詢區間 $[l, r]$ 內的和
- $|序列長度| = N, Q \leq 10^5$
- $1 \leq l, r \leq N$

然後手順寫了這東西

```cpp
#include<bits/stdc++.h>

using namespace std;

int data[100005];

int main(){
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, ans = 0, l, r;
    cin >> n;
    for ( int i = 0 ; i < n ; i++ )
        cin >> data[i];
    while ( m-- ){
        cin >> l >> r;
        ans = 0;
        for ( int i = l ; i <= r ; i++ )
            ans += data[i];
        cout << ans << '\n';
    }
}
```

前綴和

前綴和

- 每一格儲存的是 $S(i) = \sum_{i=1}^{n} a_i$
- 預處理 $S(i) = a_i + S(i-1)$
- 想要知道某一個區間 $S(r) - S(l-1)$
- 預處理 $O(N)$，查詢 $O(1)$

Code

```cpp
// by. MiohitoKiri5474
#include<bits/stdc++.h>

using namespace std;

#define max N 100005

int pre[maxN];

int main(){
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, l, r;
    cin >> n >> m;
    for ( int i = 1 ; i < n ; i++ ){
        cin >> pre[i];
        pre[i] += pre[i - 1];
    }
    while ( m-- ){
        cin >> l >> r;
        cout << pre[r] - pre[l - 1] << '\n';
    }
}
```

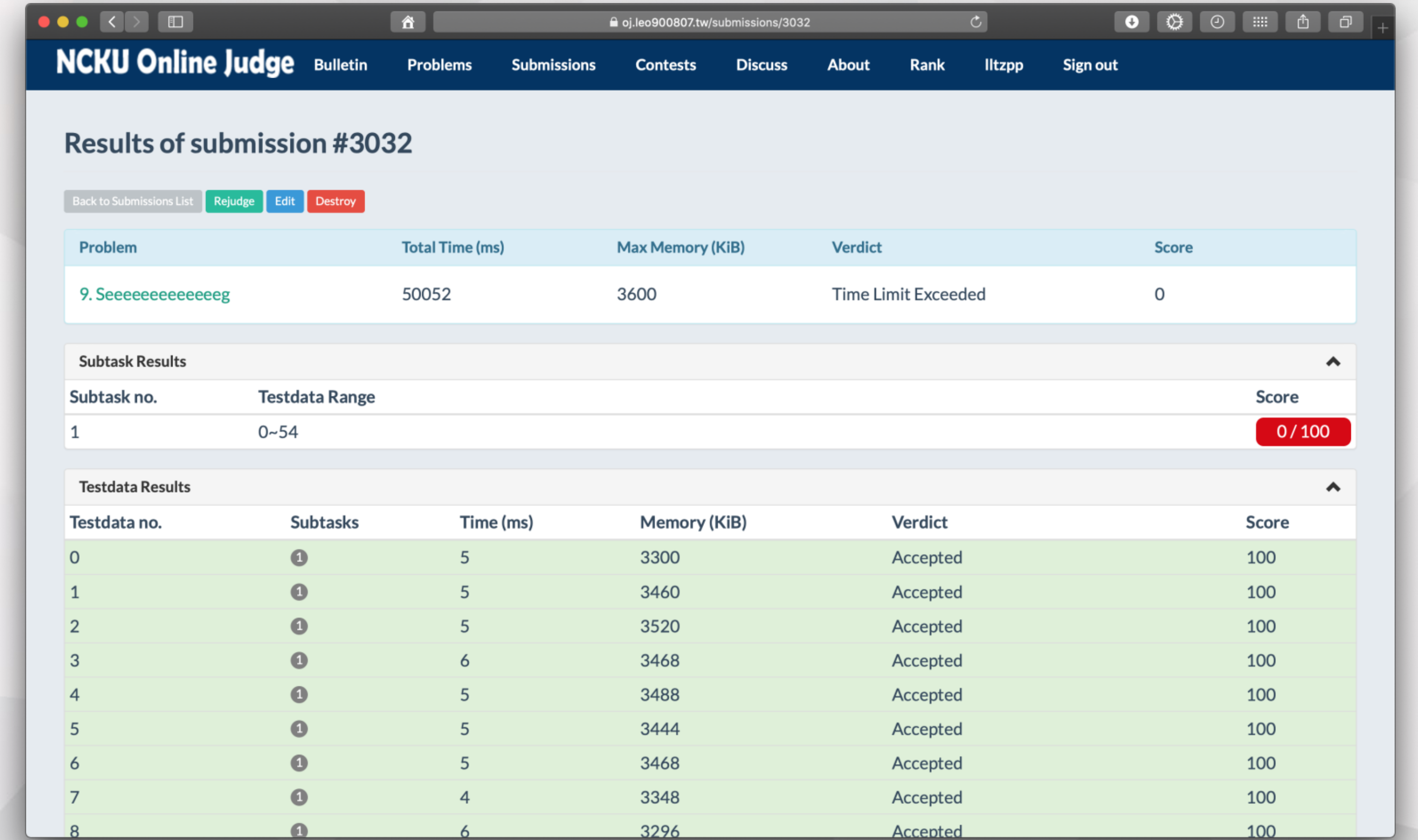
假設題目做了點變化：題目 #2

- 給定一初始序列，需要能夠執行以下兩種操作
  - 1. 修改成員的值
  - 2. 查詢區間 $[l, r]$ 內的最大值
- 序列長度 $\leq 10^5$
- 題目連結可以看這邊

於是手順刻了這東西

```cpp
#include<bits/stdc++.h>
using namespace std;
int data[100005];
int main(){
    int n, m, l, r, type, ma;
    cin >> n >> m;
    for ( int i = 0 ; i < n ; i++ )
        cin >> data[i];
    while ( m-- ){
        cin >> type >> l >> r;
        if ( type == 1 )
            data[l] = r;
        else{
            ma = -1;
            for ( int i = l ; i <= r ; i++ )
                ma = max ( ma, data[i] );
            cout << ma << '\n';
        }
    }
}
```

然後就TLE了



oj.leo900807.tw/submissions/3032

NCKU Online Judge Bulletin Problems Submissions Contests Discuss About Rank lltzpp Sign out

Results of submission #3032

Back to Submissions List | Rejudge | Edit | Destroy

Problem	Total Time (ms)	Max Memory (KiB)	Verdict	Score
9. Seeeeeeeeeeeeeg	50052	3600	Time Limit Exceeded	0

Subtask Results

Subtask no.	Testdata Range	Score
1	0~54	0 / 100

Testdata Results

Testdata no.	Subtasks	Time (ms)	Memory (KiB)	Verdict	Score
0	1	5	3300	Accepted	100
1	1	5	3460	Accepted	100
2	1	5	3520	Accepted	100
3	1	6	3468	Accepted	100
4	1	5	3488	Accepted	100
5	1	5	3444	Accepted	100
6	1	5	3468	Accepted	100
7	1	4	3348	Accepted	100
8	1	6	3296	Accepted	100

先看個複雜度

- emmm，看起來是 $O(NM)$ ，看起來要 $O(N\log N)$ 才會過

```
while ( m-- ){
    cin >> type >> l >> r;
    if ( type == 1 )
        data[l] = r;
    else{
        ma = -1;
        for ( int i = l ; i <= r ; i++ )
            ma = max ( ma, data[i] );
        cout << ma << '\n';
    }
}
```

Segment Tree

此線段樹非彼線段樹

- 如果在 google 上直接敲線段樹會出現這東西

維基台北寫作聚於每月第二個禮拜六舉行，歡迎報名參與

線段樹 [编辑]

维基百科，自由的百科全书

本文介紹的是一種儲存區間的資料結構。關於一種提供區間查詢的資料結構，請見「线段树 (区间查询)」。

此條目沒有列出任何參考或來源。(2016年9月19日)
維基百科所有的內容都應該可供查證。请协助添加来自可靠来源的引用以改善这篇条目。无法查证的内容可能被提出异议而移除。

線段樹（英語：Segment tree）是一種二元樹形資料結構，1977年由Jon Louis Bentley發明[1]，用以儲存區間或線段，並且允許快速查詢結構內包含某一點的所有區間。

一個包含n個區間的線段樹，空間複雜度為$O(n)$，查詢的時間複雜度則為$O(\log n+k)$，其中k是符合條件的區間數量。

此資料結構亦可推廣到高維度。

結構 [编辑]

本處以一維的線段樹為例。

令S是一維線段的集合。將這些線段的端點坐標由小到大排序，令其為$x_1, x_2, \cdots, x_m$。我們將被這些端點切分的每一個區間稱為「單位區間」（每個端點所在的位置會單獨成為一個單位區間），從左到右包含：

$$(-\infty, x_1), [x_1, x_1], (x_1, x_2), [x_2, x_2], \ldots, (x_{m-1}, x_m), [x_m, x_m], (x_m, +\infty)$$

線段樹的結構為一個二元樹，每個節點都代表一個坐標區間，節點N所代表的區間記為Int(N)，則其需符合以下條件：

- 其每一個葉節點，從左到右代表每個單位區間。
- 其內部節點代表的區間是其兩個兒子代表的區間之聯集。
- 每個節點（包含葉子）中有一個儲存線段的資料結構。若一個線段S的坐標區間包含Int(N)但不包含Int(parent(N))，則節點N中會儲存線段S。

參考資料 [编辑]

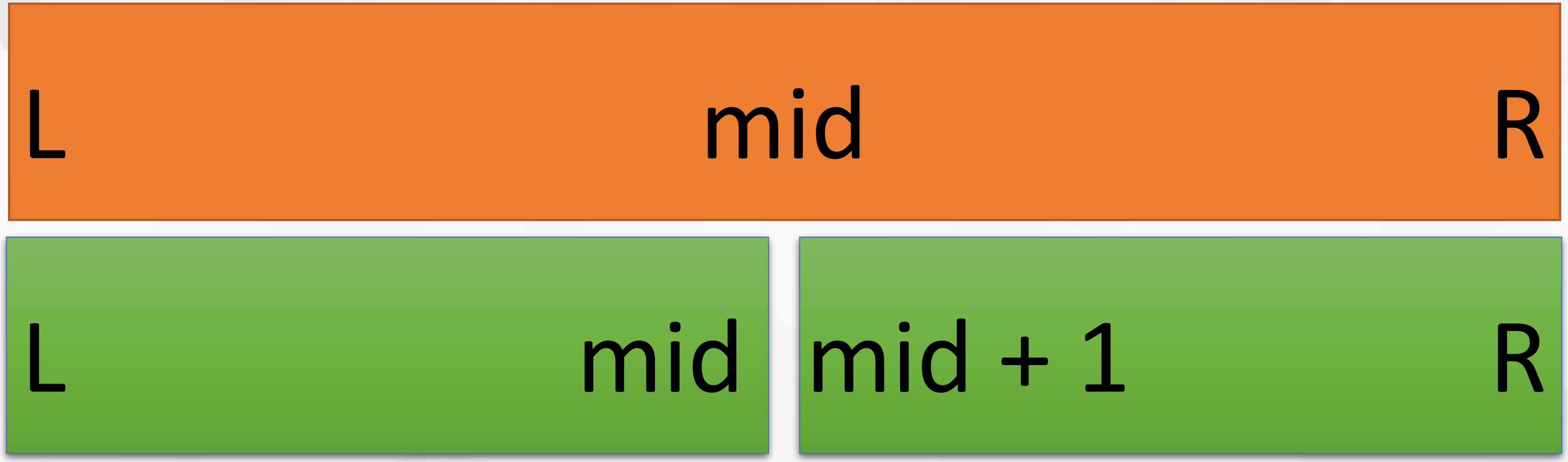
1. ^ (de Berg 等人 2000, p.229)

- de Berg, Mark; van Kreveld, Marc; Overmars, Mark; Schwarzkopf, Otfried. More Geometric Data

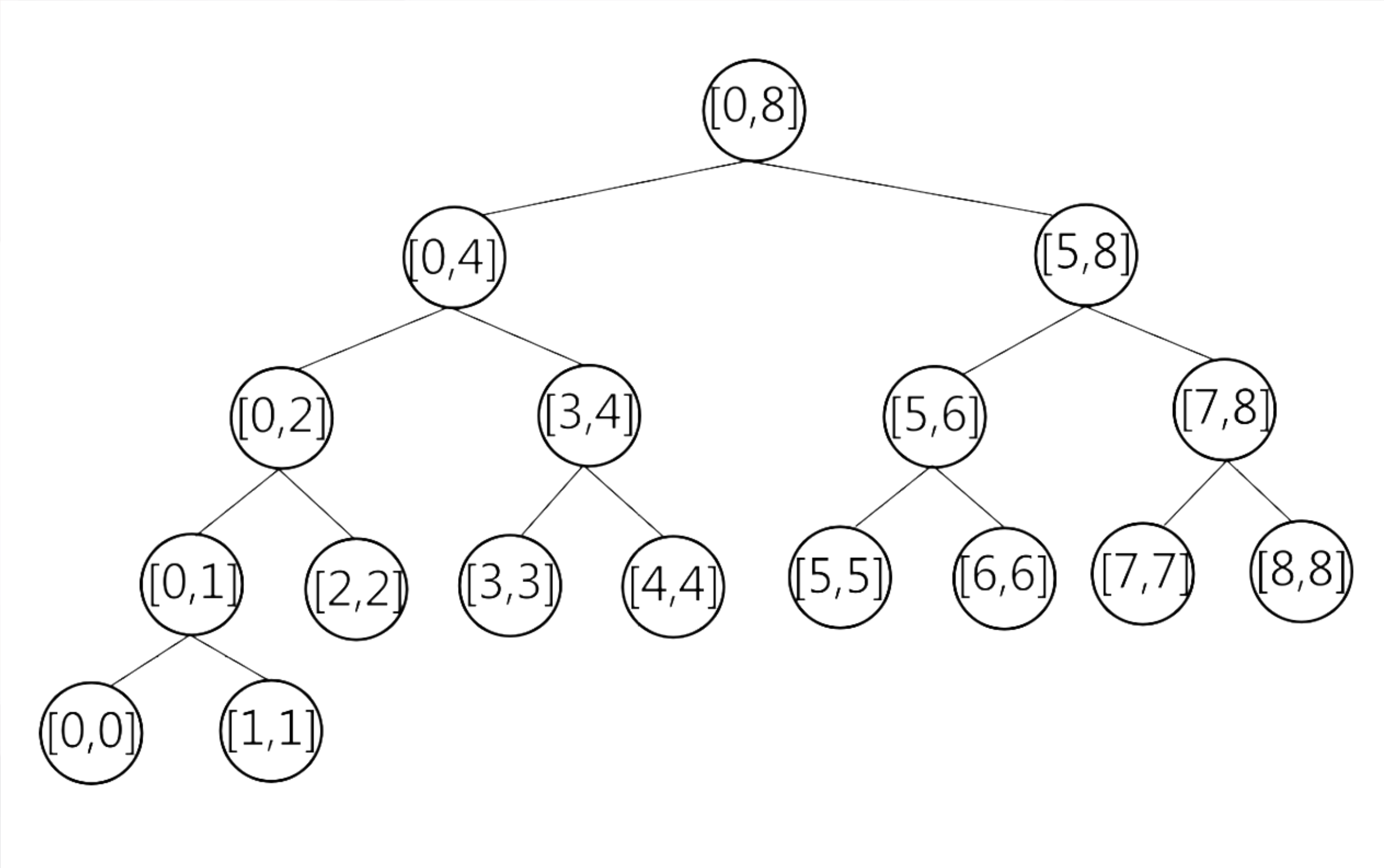
線段樹

- 由競程選手自行開發出的資料結構
- 為樹狀結構，每個節點都維護一個區間，將此區間對切往左右遞迴給左右子節點，並且合併結果
- 可以處理區間操作

好像有點難懂



直接看



Functions of Segment Tree

- build
- update (modify)
- query

update (modify)

- 更新元素用
- 查詢需修改位置位於何處，往左往右遞迴
- 假設當前區間大小為一，則更新值

Case 1

Case 2

query

- 查詢區間的答案
- 檢查區間位置來決定操作，大致上來說是往左右遞迴

Case 1

Case 2

Case 3

build

- 建構線段樹
- 把左右子區間往下遞迴
- 把左右子區間的結果合併（有點 merge sort 的概念）
- 也可以把所有點都單點修改一次啦

線段樹實作

- 指標
- 陣列

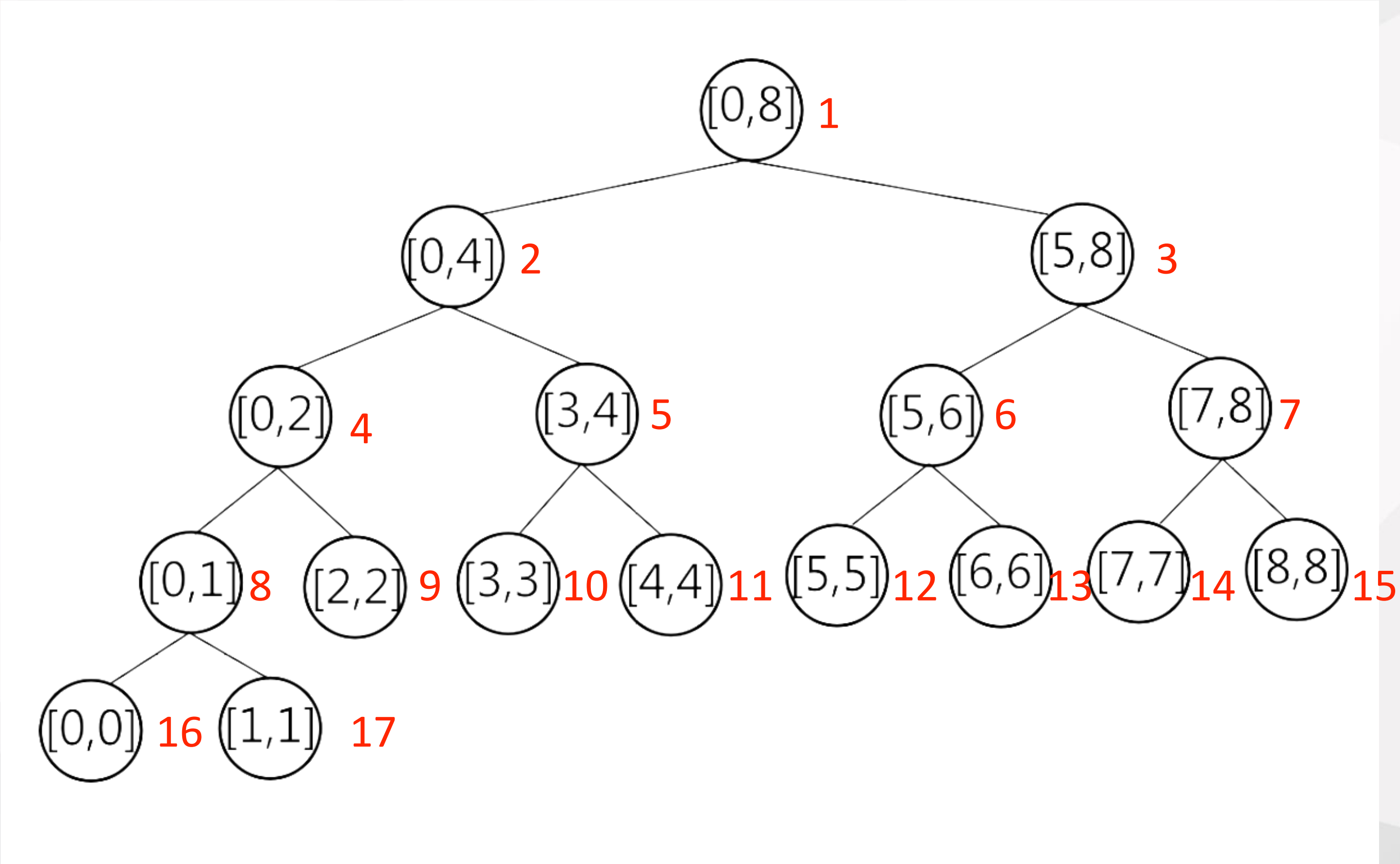
指標版本

- 每一個節點存著左子節點與右子節點的位置
- 葉節點的左子節點與右子節點皆為 NULL (nullptr in C++11 or later version)

陣列版本

- 相較於指標版本，陣列可以一次把記憶體開好開滿
- 如果現在的節點編號為 N
  - 左子節點編號為 $2N$，右邊為 $2N + 1$（編號從 1 開始的情況）
  - 左子節點編號為 $2N + 1$，右邊為 $2N + 2$（編號從 0 開始的情況）

陣列版本：編號



[0,8] 1
[0,4] 2
[5,8] 3
[0,2] 4
[3,4] 5
[5,6] 6
[7,8] 7
[0,1] 8
[2,2] 9
[3,3] 10
[4,4] 11
[5,5] 12
[6,6] 13
[7,7] 14
[8,8] 15
[0,0] 16
[1,1] 17

Code

```
void update ( int l, int r, int index, int value, int n ){
    // 當前區間左邊界、當前區間右邊界、欲更新位置、欲更新位置之值、目前節點編號
    if ( l == r )
        seg[n] = value;
    else{
        int mid = ( l + r ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        // int mid = ( l + r ) / 2, leftSon = n * 2, rightSon = leftSon + 1;
        if ( index <= mid ) // case 1
            update ( l, mid, index, value, leftSon );
        else // case 2
            update ( mid + 1, r, index, value, rightSon );

        seg[n] = max ( seg[leftSon], seg[rightSon] );
        // seg[n] = up ( seg[leftSon], seg[rightSon] );
    }
}
```

Code

```cpp
int query ( int l, int r, int nowL, int nowR, int n ){
    // 欲查詢範圍左邊界、欲查詢範圍右邊界、當前區間左邊界、當前區間右邊界、當前區間編號
    if ( l <= nowL && nowR <= r )
        return seg[n];
    int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
    if ( r <= mid ) // case 1
        return query ( l, r, nowL, mid, leftSon );
    if ( mid < l ) // case 2
        return query ( l, r, mid + 1, nowR, rightSon );

    // case 3
    return max ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
    // up ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
}
```

分析一下複雜度

- update (modify), query
  - 跟二分搜一樣的概念 → $O(\log N)$
- 如果題目有多筆（M）查詢
  - 總複雜度 → $O(M\log N)$

回頭看題目 #1的變化：題目 #3

- 給定一初始序列，並有 Q 筆操作
  1. 查詢區間 $[l, r]$ 內的和
  2. 修改其中一個元素
- $|序列長度| = N, Q \leq 10^5$
- $1 \leq l, r \leq N$

前綴和88

- 前綴和一旦修改其中一個值，就需要修改 $S(i) \sim S(n)$
  → 複雜度 $O(N)$，聽起來跟暴力差不多
- 也可以用線段樹寫，不過線段樹的 coding 量（ry

~~樹狀數組（Fenwick Tree）~~
二元索引樹（Binary Indexed Tree, BIT）

BIT

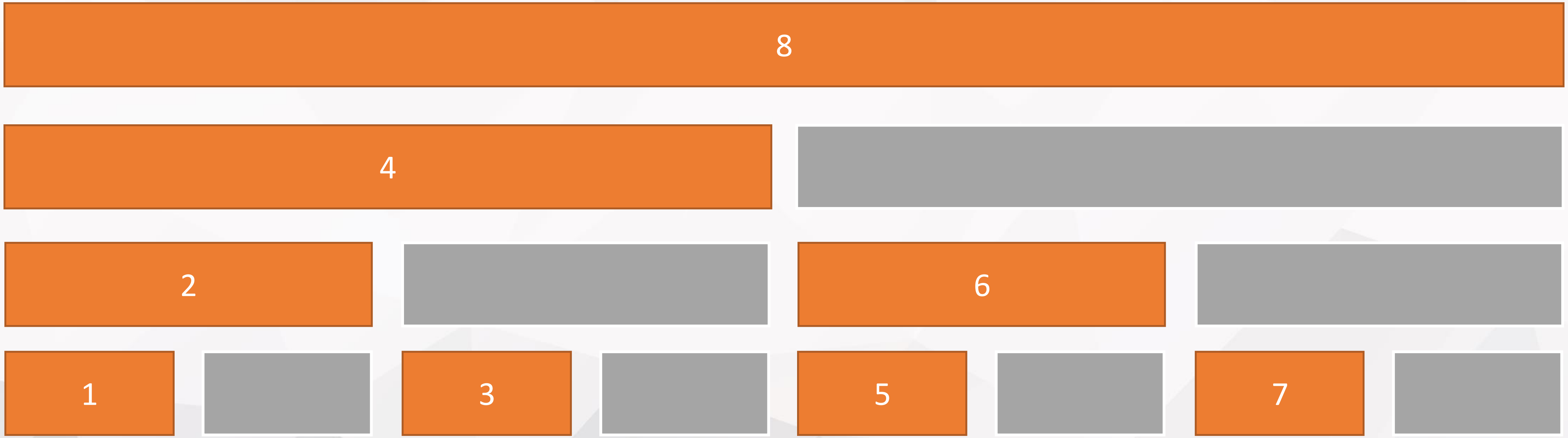
- 由 Peter M. Fenwick 於 1994 年提出
- 結構上就是把線段樹的右子樹拔掉

等等，拔掉右子樹？！

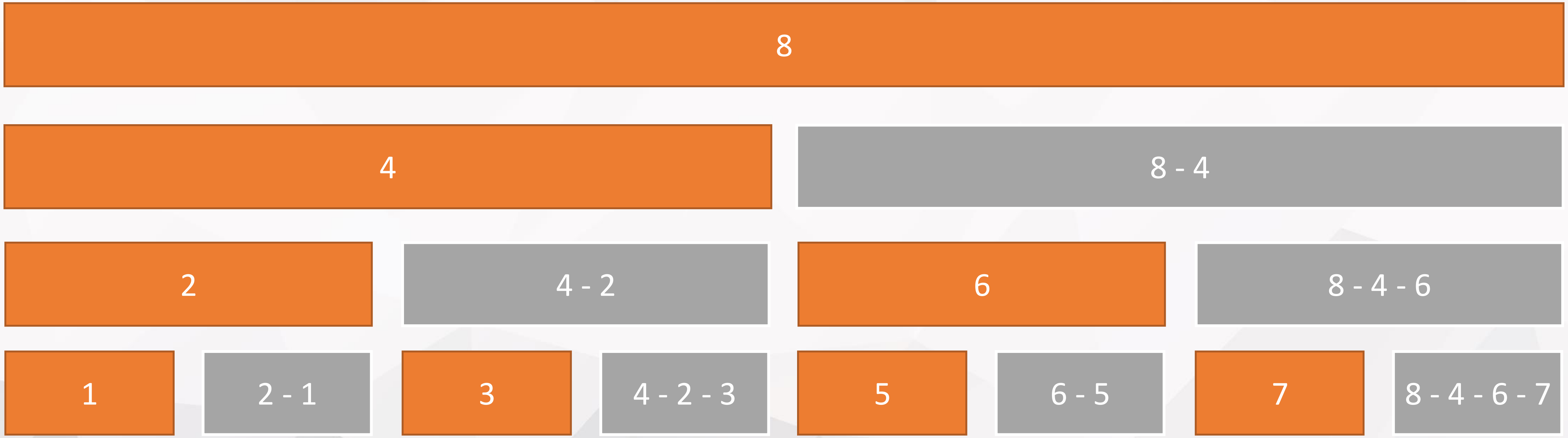
回想一下

- 由前面前綴和的觀念，我們知道 $a_i = S(i) - S(i-1)$
- 所以如果把右子樹都拔掉，也不影響

不對不對，感覺還是怪怪的



那麼這樣呢



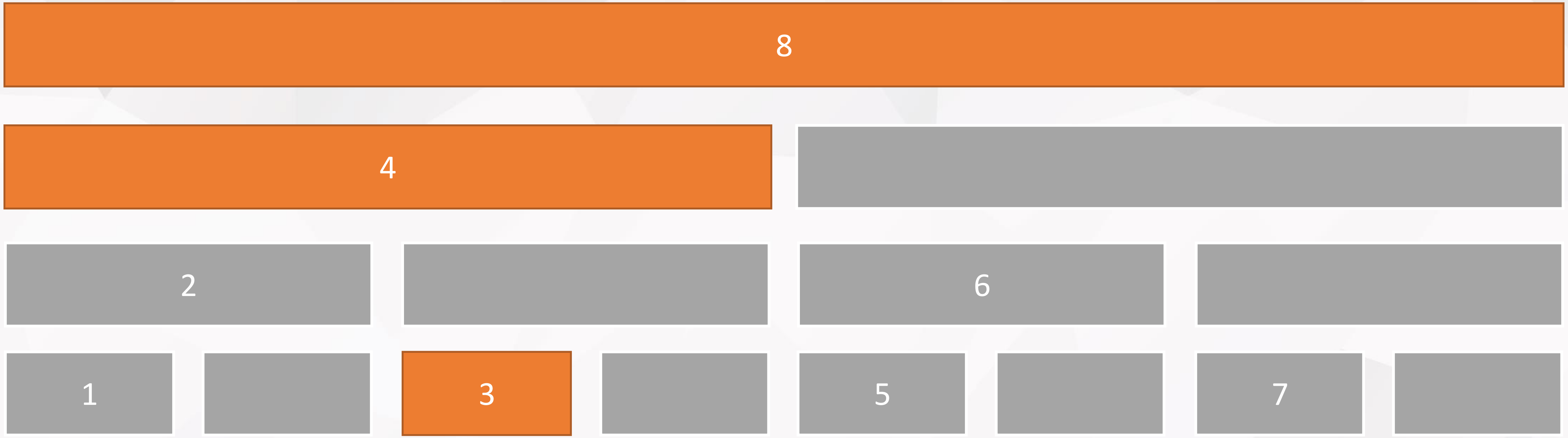
用數學表達的話

- $BIT_i = \sum_{j=i-lowbit(i)+1}^{n} a_j$
- lowbit:
  - 現在這個數字二進位中，最低位的 1 所代表的值 for example: $lowbit(1) = 1, lowbit(2) = 2, lowbit(3) = 1$
  - 在 C++ 可以用位元運算達到這件事（ i & -i ）

修改 (add)

- 將所有包含 idx 的區間加上變異量
- while (idx <= n){
  bit[idx] += delta;
  idx += idx & -idx;
  }
- 修改一個點有 $\log N$ 個區間
  → 複雜度 $O(\log N)$

修改 a_3



- $3 = (11)_2 \rightarrow 3 + lowbit(3) = (11)_2 + (1)_2 = 4(100)_2$
  $4 = (100)_2 \rightarrow 4 + lowbit(4) = (100)_2 + (100)_2 = 8(1000)_2$

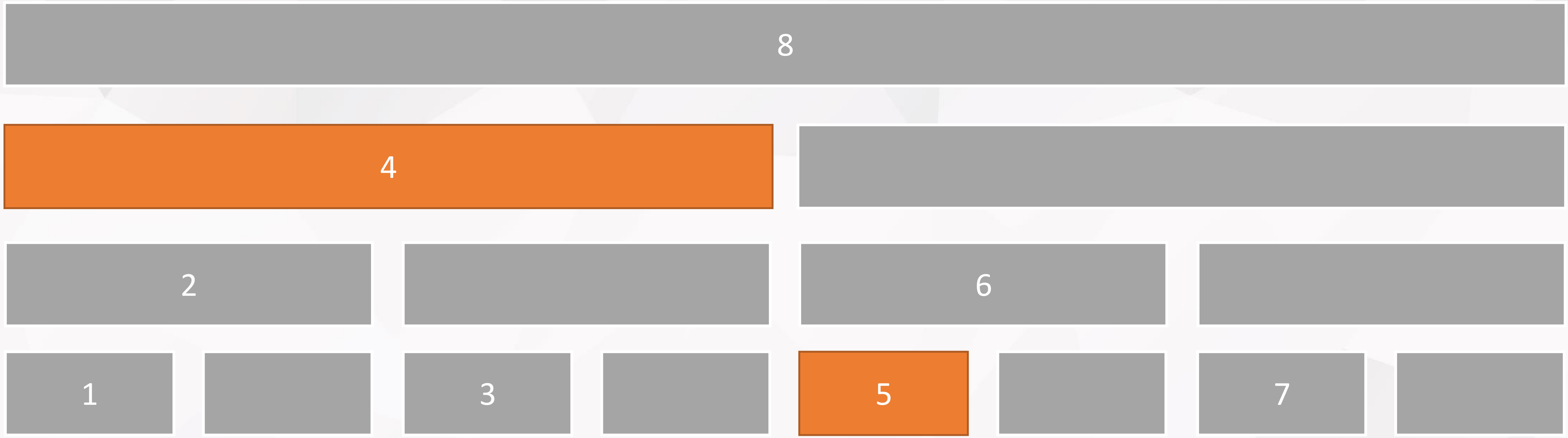
查詢 (sum)

- 將有覆蓋到的範圍都加起來
- while (idx){
  sum += bit[idx];
  idx -= idx & -idx;
  }

```
while ( idx ){
    sum += bit[idx];
    idx -= idx & -idx;
}
```

- 最多會覆蓋到 $\log N$ 個區間
  → 複雜度 $O(\log N)$

查詢 $S(5)$



- $5 = (101)_2 \rightarrow 5 - lowbit(5) = (101)_2 - (1)_2 = 4(100)_2$
  $4 = (100)_2 \rightarrow 4 - lowbit(4) = (100)_2 - (100)_2 = 0(0)_2$

完整的 code

```cpp
// by. Miohitokiri5474
#include<bits/stdc++.h>

using namespace std;

#define maxN 100005

int bit[maxN], basic[maxN];
int n;

void add ( int idx, int delta ){
    while ( idx < n ){
        bit[idx] += delta;
        idx += idx & -idx;
    }
}

int sum ( int idx ){
    int res = 0;
    while ( idx ){
        res += bit[idx];
        idx -= idx & -idx;
    }

    return res;
}
```

```cpp
int main(){
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int m, l, r, type;
    cin >> n >> m;
    for ( int i = 1 ; i <= n ; i++ ){
        cin >> basic[i];
        add ( i, basic[i] );
    }

    while ( m-- ){
        cin >> type >> l >> r;
        if ( type == 1 ){
            add ( l, r - basic[l] );
            basic[l] = r;
        }
        else
            cout << sum ( r ) - sum ( l ) << '\n';
    }
}
```

還有一個題目 #4

- 給定一初始序列，並有 Q 筆操作
  1. 查詢區間 $[l, r]$ 內的最大值
  2. 區間 $[l, r]$ 的元素加上 k
- $|序列長度| = N, Q \leq 10^5$
- $1 \leq l, r \leq N$

呃。。。對 $[l,r]$ 做單點修改

- Emmm好像不是不可行
- 那先看一下複雜度，$O(N\log N)$，可能要做 Q 次
  $\rightarrow$ 最差 $O(NQ\log N)$ 聽起來有點糟

Segment Tree with Lazy Tag

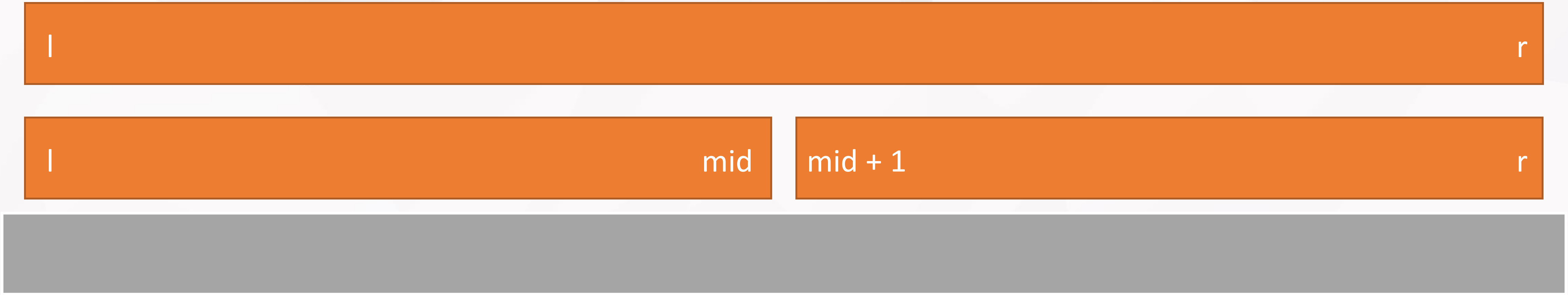
Lazy Tag

- 每次操作不是真的操作
- 只有當前節點範圍 $[nowL, nowR]$ 包含於區間修改區間 $[l, r]$ 內，才會對這個節點打上標記，否則往左與右節點遞迴
- 減少修改次數，假設現在一個區間要修改 k 次，如果狀態可以疊加那麼一次推下去的效率顯然比更新 k 次好得多

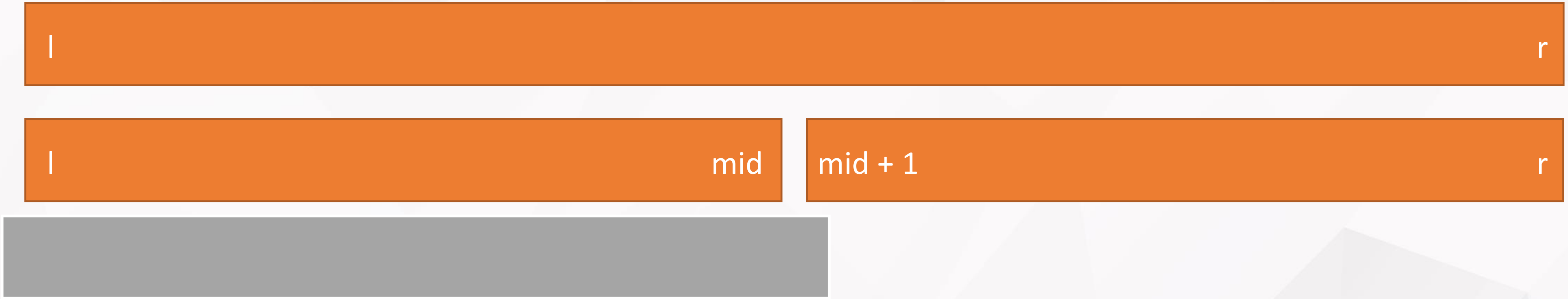
Lazy Tag 實作

- 一個節點會有兩個變數，一個是紀錄 Lazy Tag，另外一個是紀錄原本的值
- 通常個人會用 pair，first 是紀錄最大值，second 是 tag

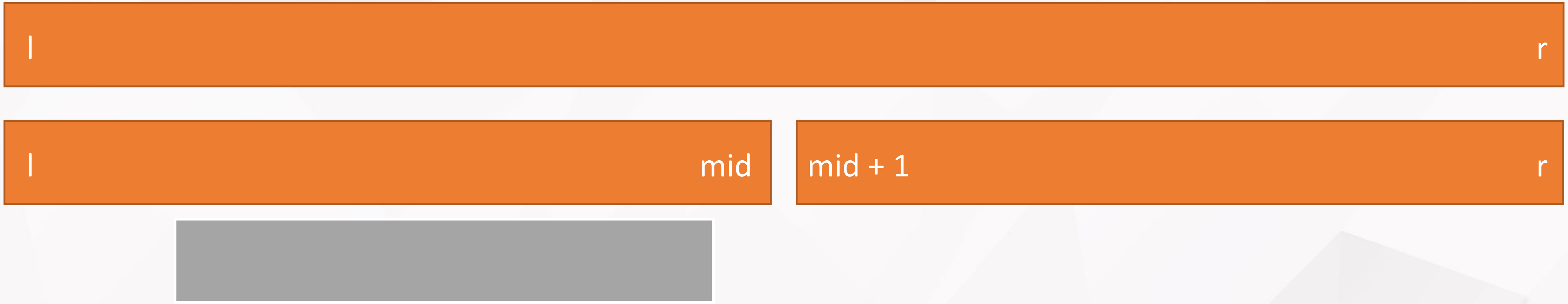
Case 1



Case 2



Case 3



push

- 將懶惰標記往下推
- 在原本的函數中，要往下一層遞迴時先把標記 push 下去

```
void push ( int n ){
    if ( seg[n].S ){
        int leftSon = n << 1, rightSon = n << 1 | 1, value = seg[n].S;
        seg[n].S = 0;
        seg[leftSon].F += value, seg[rightSon].F += value;
        seg[leftSon].S += value, seg[rightSon].S += value;
    }
}
```

區間修改 (modify)

```
void modify ( int l, int r, int nowL, int nowR, int value, int n ){
    if ( l <= nowL && nowR <= r )
        seg[n].F += value, seg[n].S += value;
    else{
        int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n );
        if ( r <= mid )
            modify ( l, r, nowL, mid, value, leftSon );
        else if ( mid < l )
            modify ( l, r, mid + 1, nowR, value, rightSon );
        else{
            modify ( l, r, nowL, mid, value, leftSon );
            modify ( l, r, mid + 1, nowR, value, rightSon );
        }
        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}
```

code

```cpp
// by. Miohitokiri5474
#include<bits/stdc++.h>

using namespace std;

#define maxN 100005
#define F first
#define S second
typedef pair < int, int > pii;

pii seg[maxN << 2];

void push ( int n ){
    if ( seg[n].S ){
        int leftSon = n << 1, rightSon = n << 1 | 1, value = seg[n].S;
        seg[n].S = 0;
        seg[leftSon].F += value, seg[rightSon].F += value;
        seg[leftSon].S += value, seg[rightSon].S += value;
    }
}

void update ( int l, int r, int index, int value, int n ){
    if ( l == r )
        seg[n].F = value;
    else{
        int mid = ( l + r ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n );
        if ( index <= mid )
            update ( l, mid, index, value, leftSon );
        else
            update ( mid + 1, r, index, value, rightSon );

        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}

int query ( int l, int r, int nowL, int nowR, int n ){
    if ( l <= nowL && nowR <= r )
        return seg[n].F;
    int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
    push ( n );
    if ( r <= mid )
        return query ( l, r, nowL, mid, leftSon );
    if ( mid < l )
        return query ( l, r, mid + 1, nowR, rightSon );
    return max ( query ( l, r, nowL, mid, leftSon ), query ( l, r, mid + 1, nowR, rightSon ) );
}

void modify ( int l, int r, int nowL, int nowR, int value, int n ){
    if ( l <= nowL && nowR <= r )
        seg[n].F += value, seg[n].S += value;
    else{
        int mid = ( nowL + nowR ) >> 1, leftSon = n << 1, rightSon = leftSon | 1;
        push ( n );
        if ( r <= mid )
            modify ( l, r, nowL, mid, value, leftSon );
        else if ( mid < l )
            modify ( l, r, mid + 1, nowR, value, rightSon );
        else{
            modify ( l, r, nowL, mid, value, leftSon );
            modify ( l, r, mid + 1, nowR, value, rightSon );
        }
        seg[n].F = max ( seg[leftSon].F, seg[rightSon].F );
    }
}
```

code (con’t)

```
int main(){
    ios::sync_with_stdio ( false );
    cin.tie ( 0 );
    cout.tie ( 0 );

    int n, m, l, r, in, type;
    cin >> n >> m;

    // build
    for ( int i = 1 ; i <= n ; i++ ){
        cin >> in;
        update ( 1, n, i, in, 1 );
    }

    while ( m-- ){
        // type 1: 單點修改
        // type 2: 區間查詢
        // type 3: 區間修改
        cin >> type >> l >> r;
        if ( type == 1 ){
            update ( 1, n, l, r, 1 );
        }
        else if ( type == 2 )
            cout << query ( l, r, 1, n, 1 ) << '\n';
        else{
            cin >> in;
            modify ( l, r, 1, n, in, 1 );
        }
    }
}
```

codes on github

- https://ppt.cc/fgNPix
- https://github.com/MiohitoKiri5474/CodesBackUp/tree/master/ncku-icpc/2020/week9

差分

問題

給你一個長度為 N 的序列，接下來有 Q 筆操作：

- 區間加值：將 $[l,r]$ 的所有數字加上 k 。
- 單點查詢：查詢序列第 x 個的值為多少。

保證所有查詢操作都在加值操作後面。

範圍： $N,Q \leq 10^5$

作法

每一次修改就暴力將所有區間內的值修改
然後再單點查詢

修改複雜度： $O(N)$
查詢複雜度： $O(1)$
總複雜度： $O(NQ)$

TLE

觀察問題

觀察一下，題目提到「所有查詢都在加值操作後面」
那是不是可以預處理完所有加值操作
再進行查詢
這時就可以利用一個技巧

作法

假設序列是 1 5 9 7 5 6 3 4
那麼可以將兩兩數字之間的差紀錄下來
我們開一個陣列 dif[N] 紀錄
dif[i] 代表 a[i + 1] – a[i] 的值，此時假設 a[0] 為 0

index	0	1	2	3	4	5	6	7	8
a	0	1	5	9	7	5	6	3	4
dif	1	4	4	-2	-2	1	-3	1	

作法

當要將區間 $[l, r]$ 的所有數字增加 k 時
就將 dif[l - 1] 的值增加 k ，將 dif[r] 的值減少 k
就完成一次修改了

index	0	1	2	3	4	5	6	7	8
a	0	1	5	9	7	5	6	3	4
dif	1	4	4	-2	-2	1	-3	1	

作法

假設將區間 [3, 6] 增加 2
則將 dif[2] 增加 2 ，將 dif[6] 減少 2
因為 a[3] 與 a[2] 的差值在修改後增加了 2
而 a[7] 與 a[6] 的差值則減少了 2 ，且其他差值不變

index	0	1	2	3	4	5	6	7	8
a	0	1	5	9	7	5	6	3	4
dif	1	4	4	-2	-2	1	-3	1	

作法

假設將區間 [3, 6] 增加 2
則將 dif[2] 增加 2 ，將 dif[6] 減少 2
因為 a[3] 與 a[2] 的差值在修改後增加了 2
而 a[7] 與 a[6] 的差值則減少了 2 ，且其他差值不變

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
a	0	1	5	11	9	7	8	3	4
dif	1	4	6	-2	-2	1	-5	1	

作法

如此一來就可以做到 $O(1)$ 修改
而且只要對 dif 陣列做一次前綴和就能將數列復原
最後只要單點查詢值就好
總複雜度： $O(N + Q)$ AC

index	0	1	2	3	4	5	6	7	8
a	0	1	5	11	9	7	8	3	4
dif	1	4	6	-2	-2	1	-5	1	

Questions?