NCKU Programming Contest Training Course

2018/06/03

Lin Yun Wen

Binary Indexed Tree 、Segment Tree

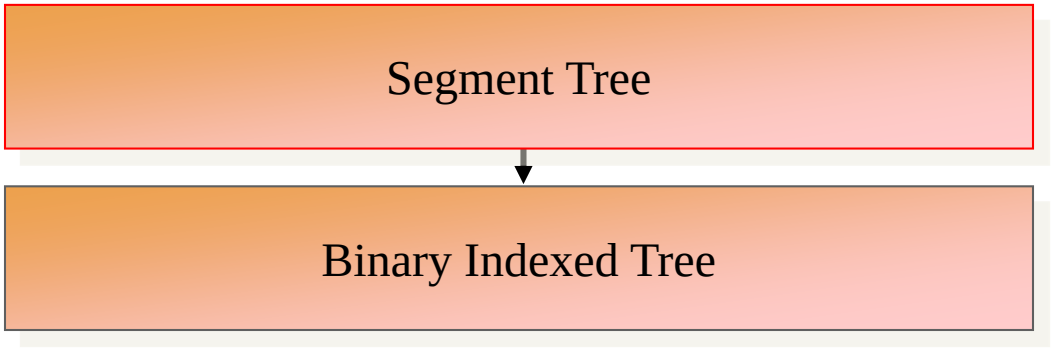
Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

made by electron & free999 & a711186

Sequence

- Sequence
  - 數學上的概念為「一連串的數字」，中文譯作「數列」
  - Ex. 5, 5, 6, 6, 520
  - 資料結構
    - Array
    - List
    - Binary Search Tree
    - **Binary Indexed Tree**
    - **Segment Tree**
    - Sparse Table
    - Cartesian Tree

Outline

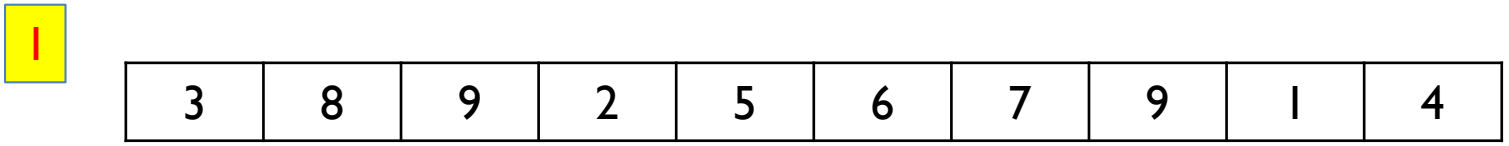


made by electron & free999 & a711186

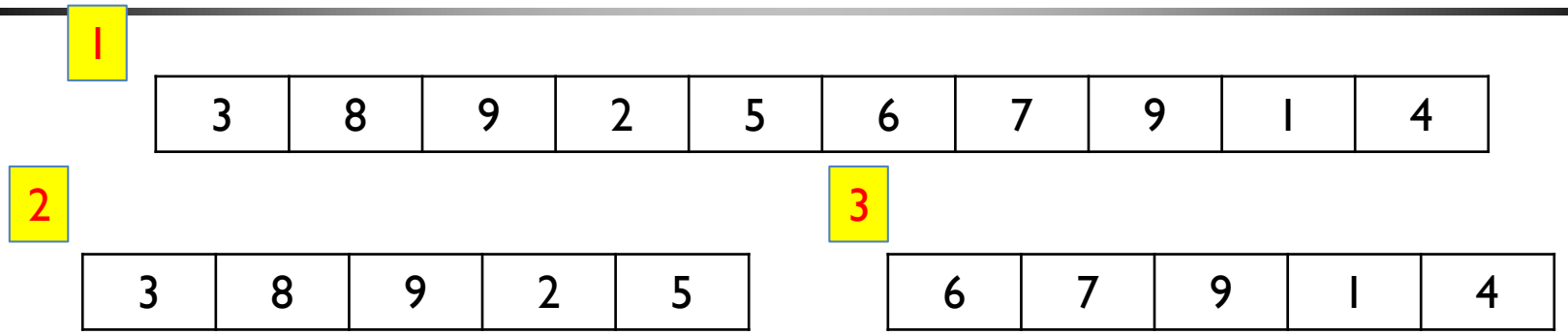
Segment Tree

- Operation
  - 可以快速算出任意區間的總和或查詢、更新數值
  - 可以插入、刪除數值。
- Naïve Solution
  - 陣列長度 = N, Query 數 = Q
  - 求區間總和，時間複雜度 **O(QN)**…
- Segment Tree
  - 有根樹狀結構
  - 每個節點記載一個區間[L,R]的資訊，且都有兩個孩子，左節點記載[L, $\frac{L+R}{2}$]，右節點記載[$\frac{L+R}{2}$+1, R]

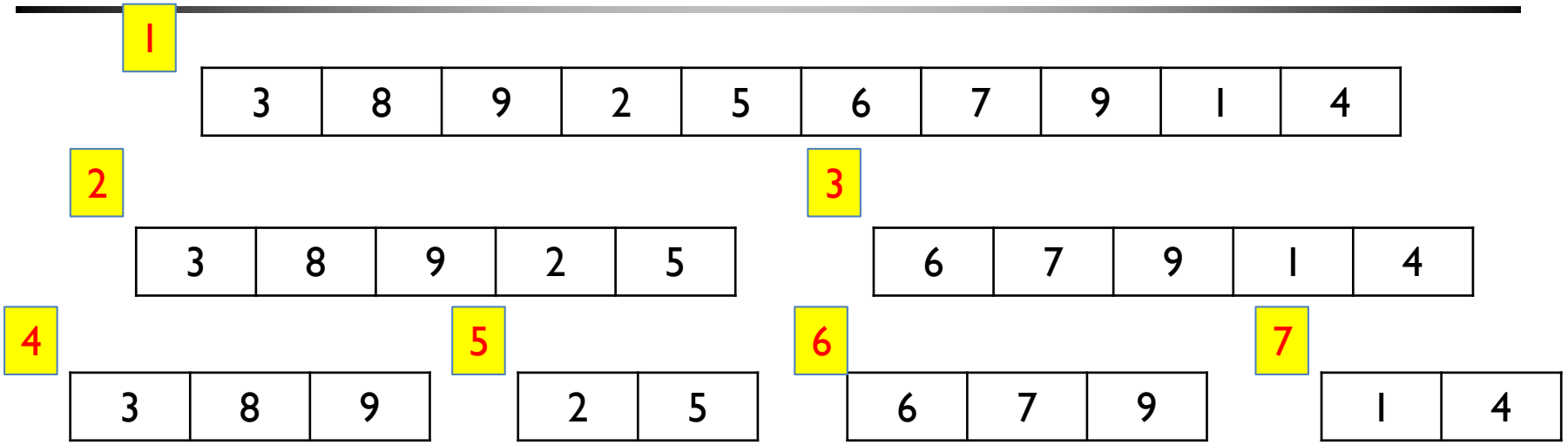
Segment Tree



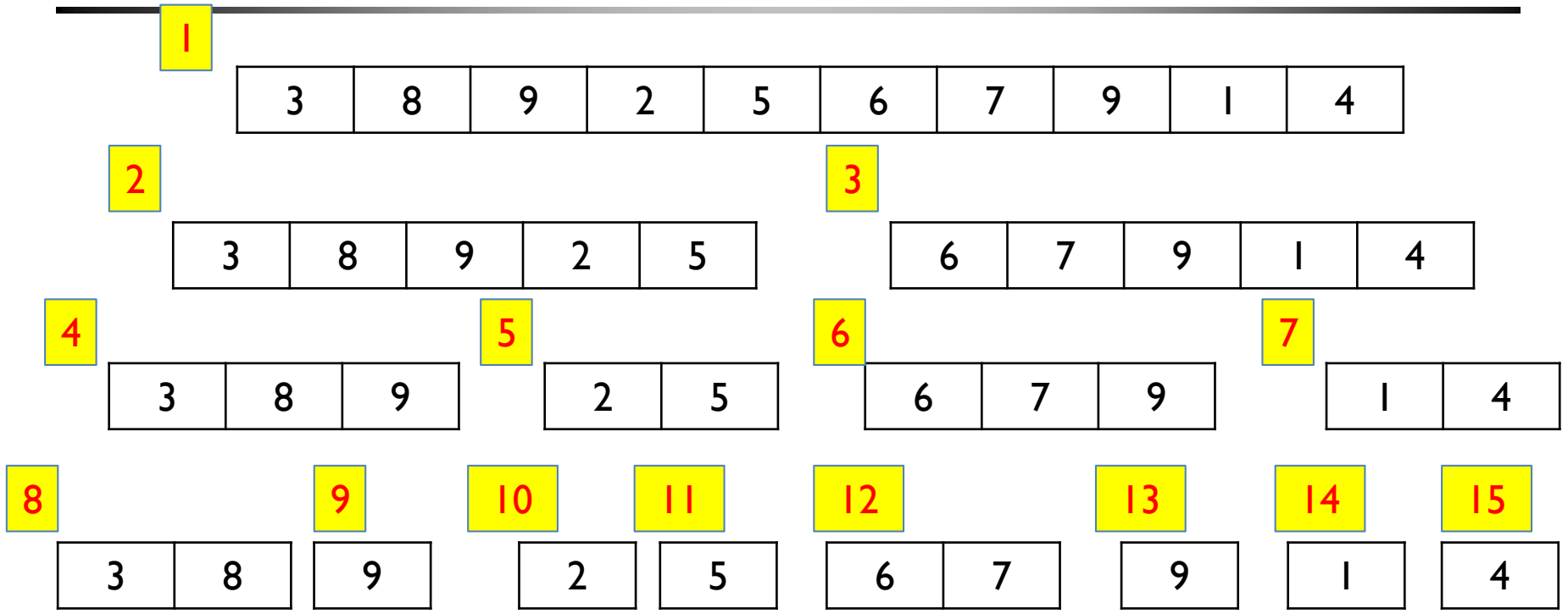
Segment Tree



Segment Tree

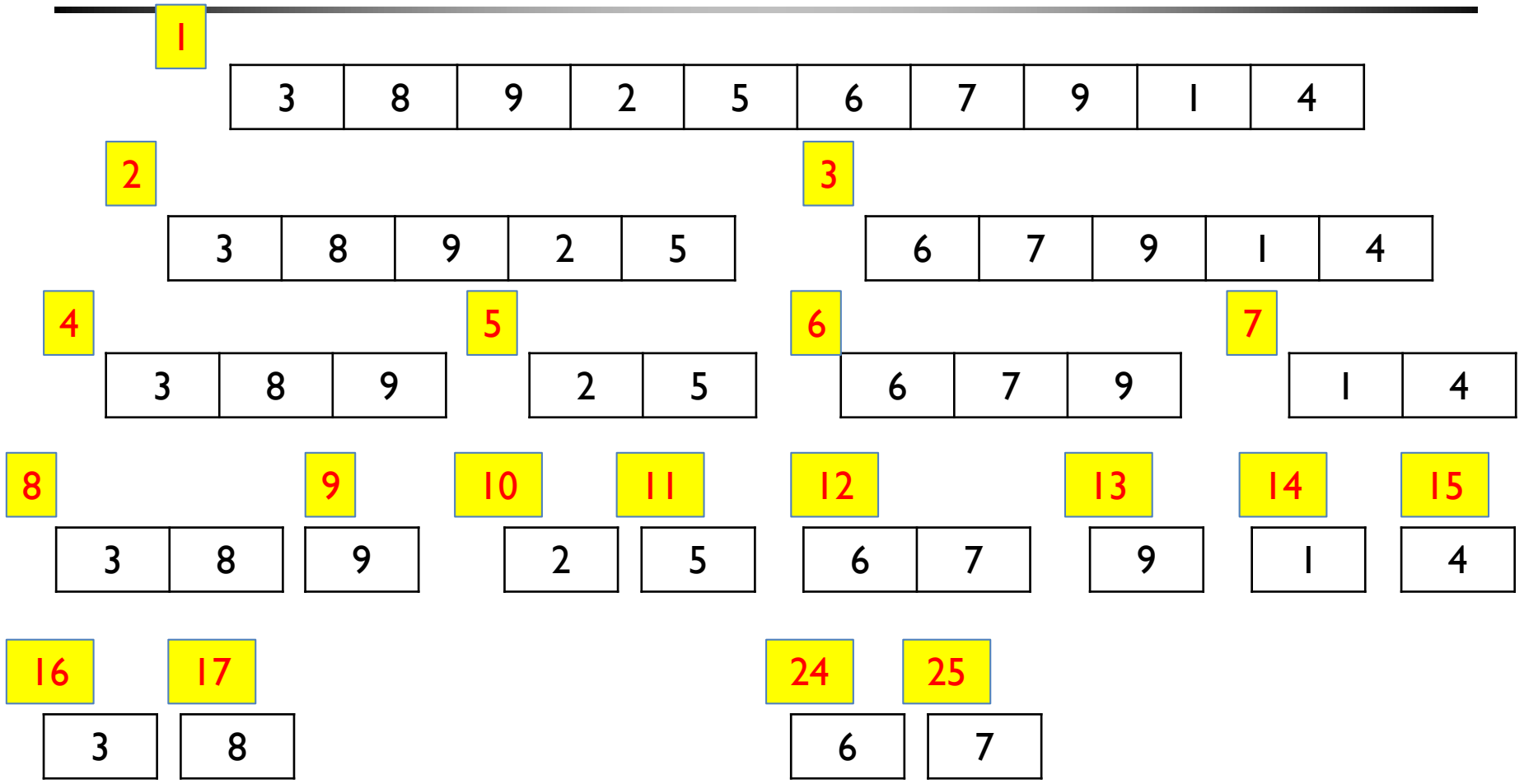


Segment Tree



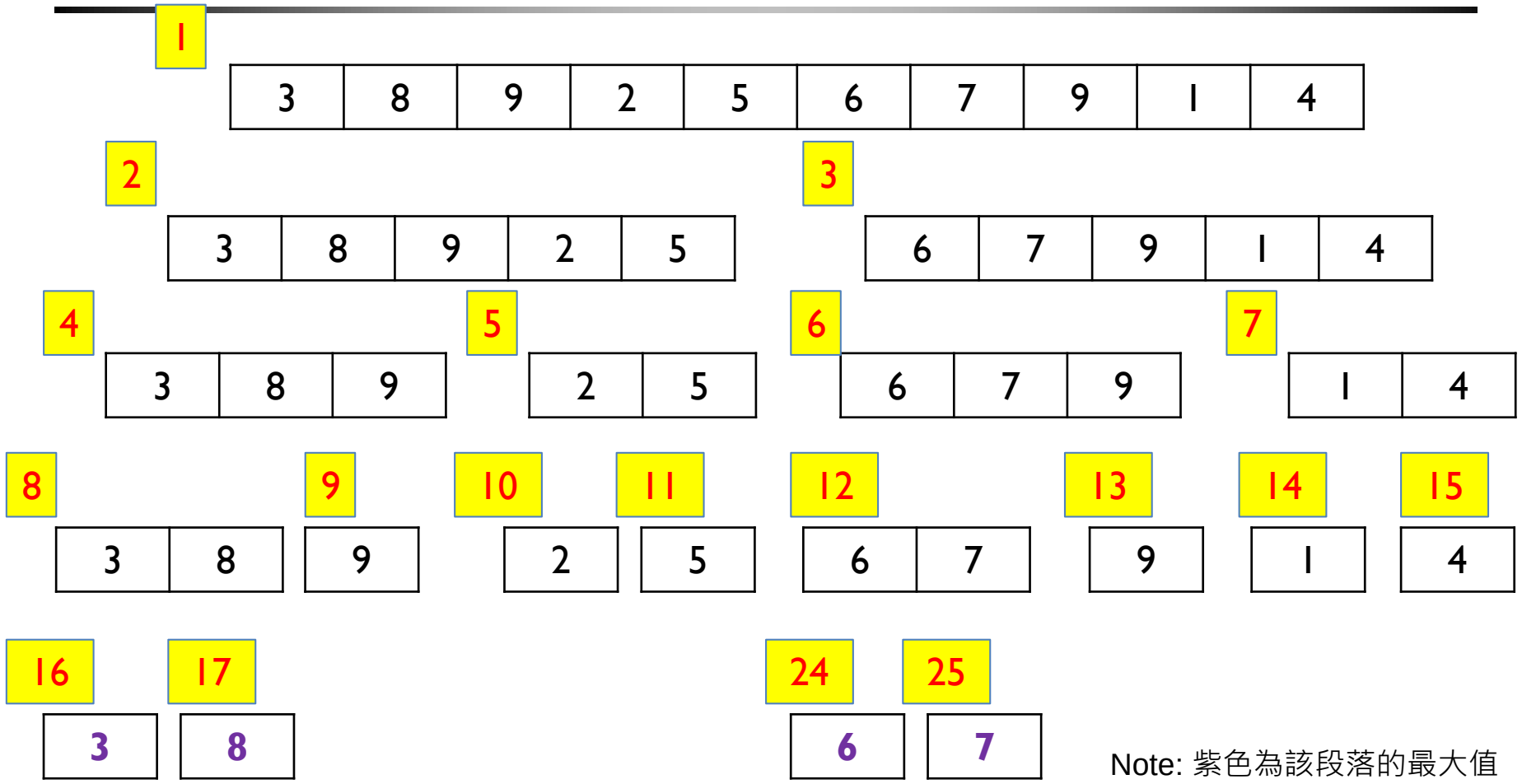
made by electron & free999 & a711186

Segment Tree



made by electron & free999 & a711186

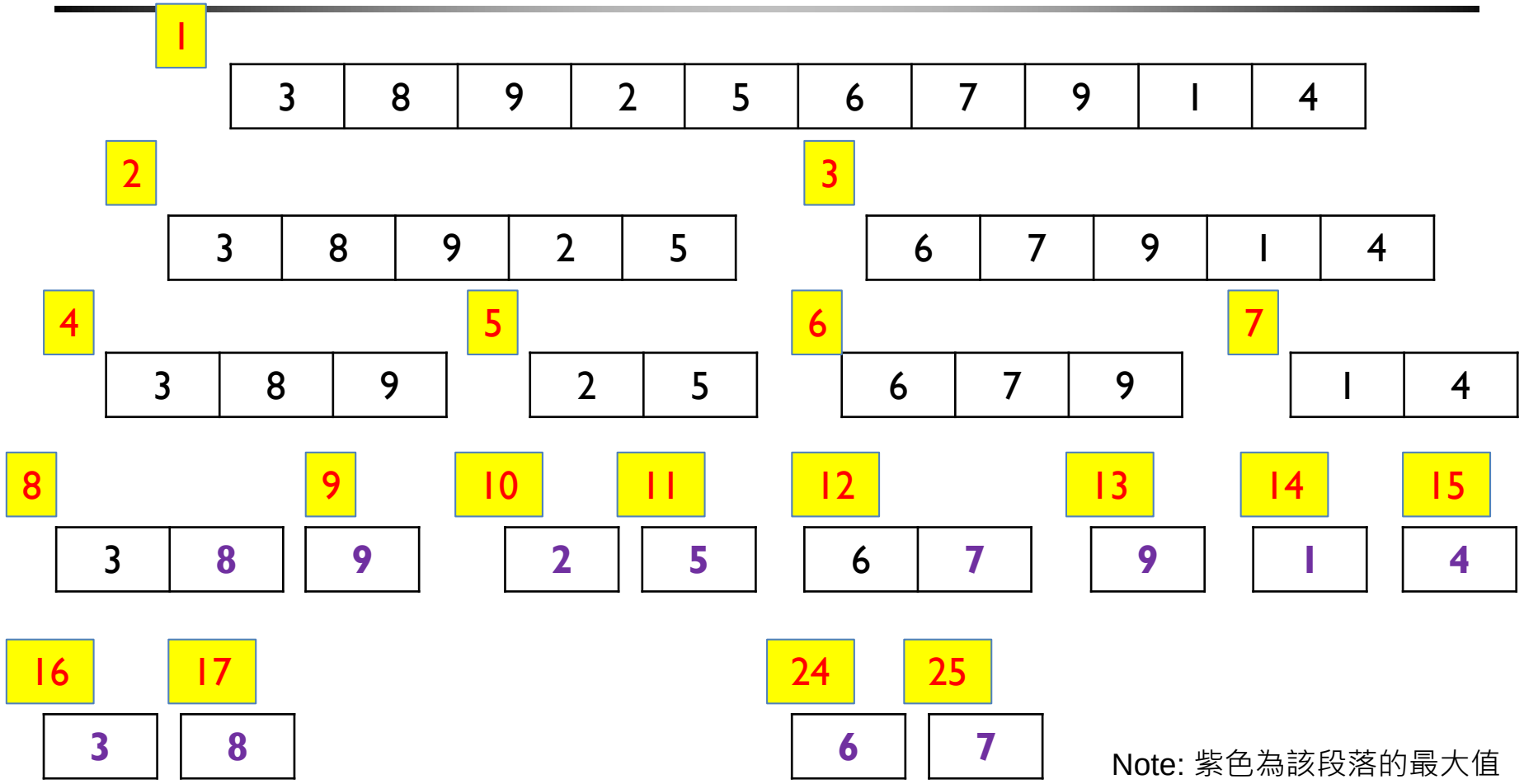
Segment Tree



Note: 紫色為該段落的最大值

made by electron & free999 & a711186

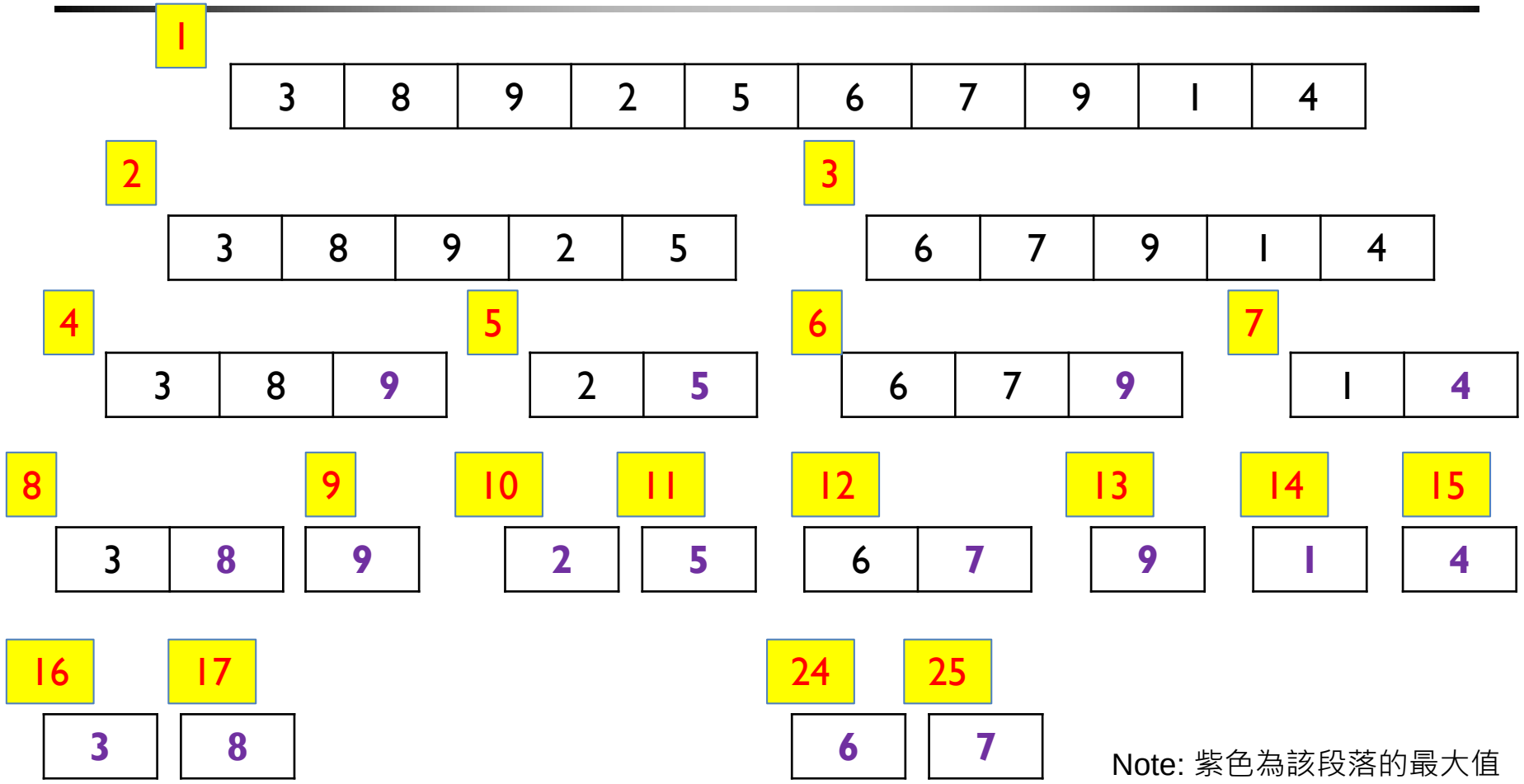
Segment Tree



Note: 紫色為該段落的最大值

made by electron & free999 & a711186

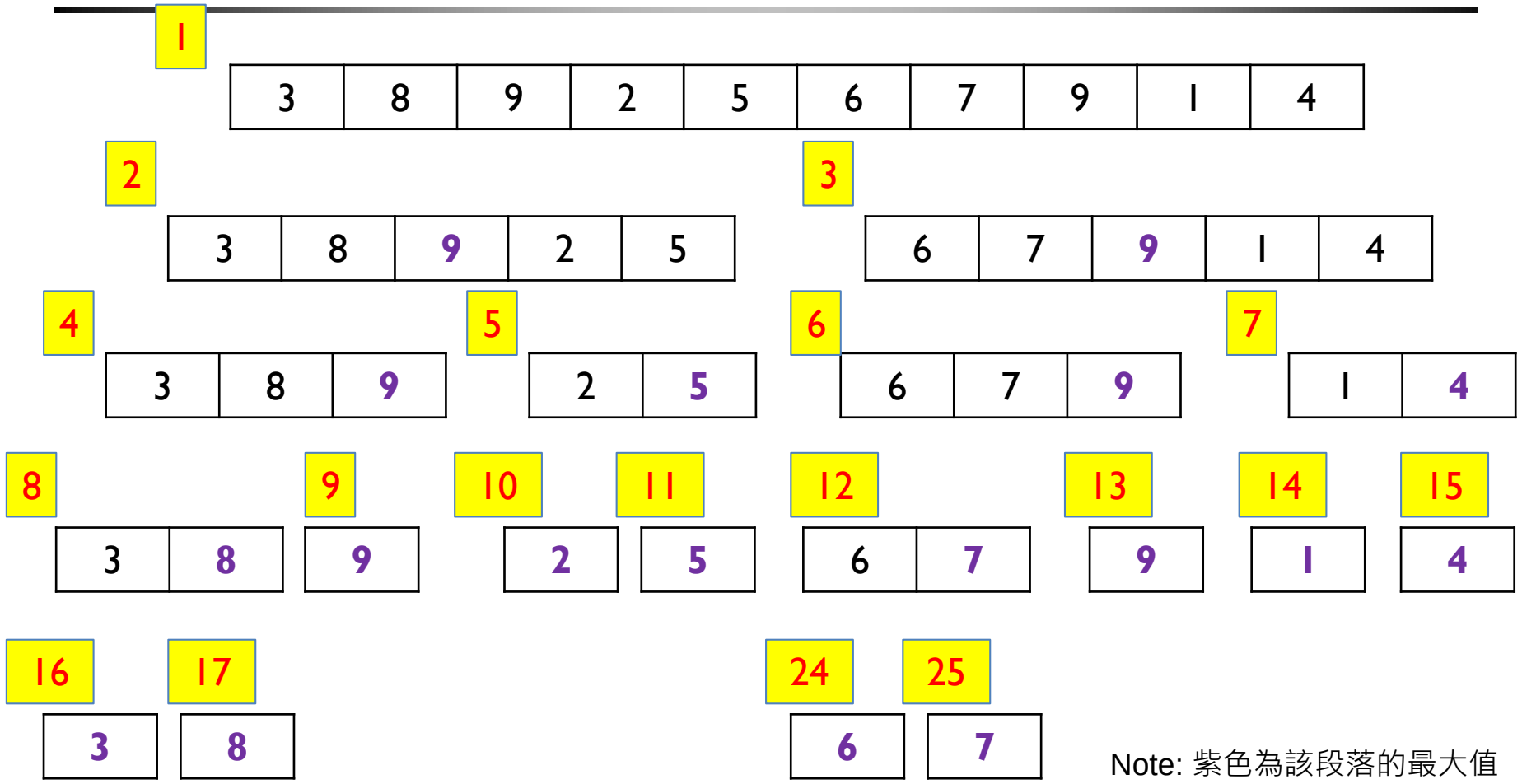
Segment Tree



Note: 紫色為該段落的最大值

made by electron & free999 & a711186

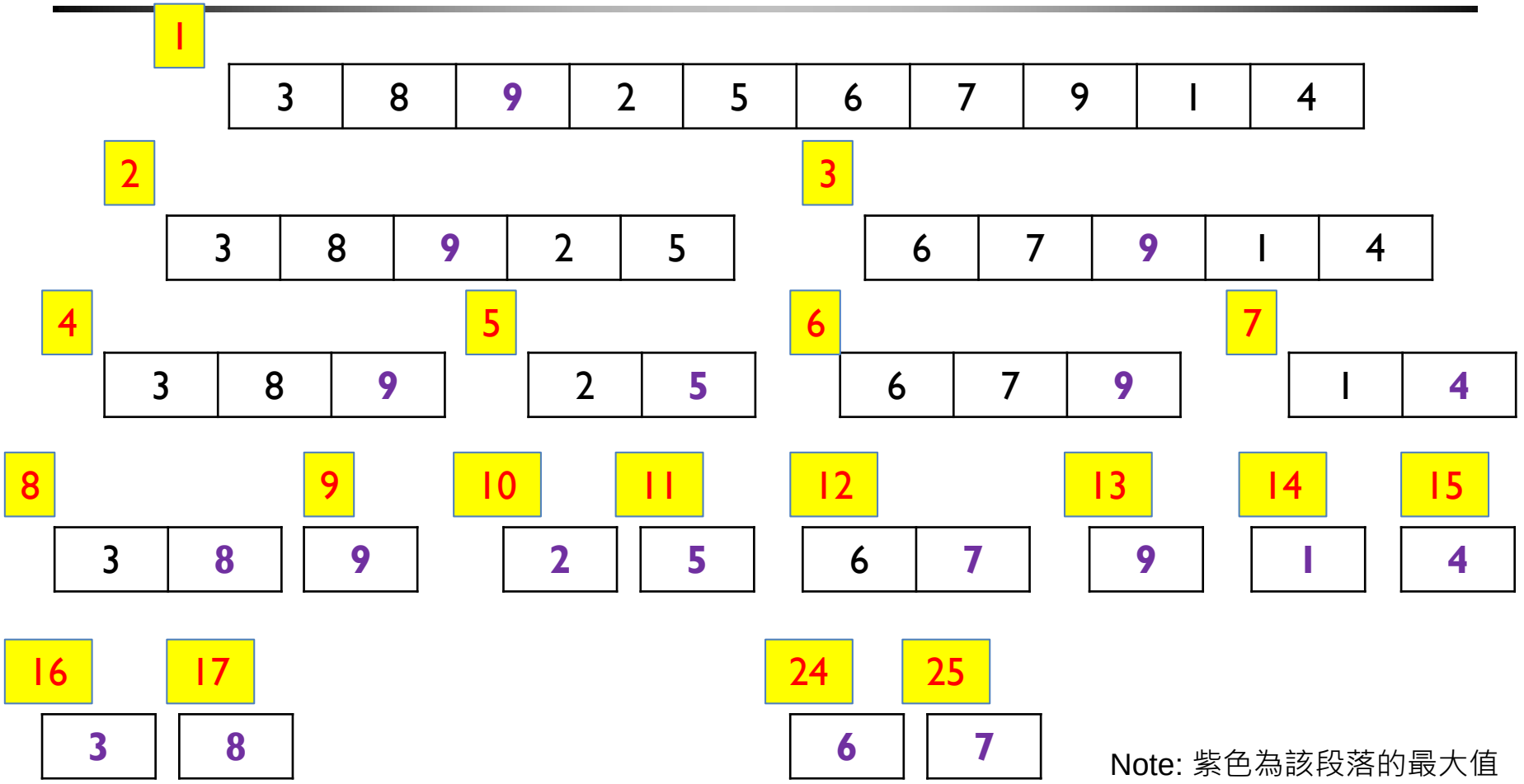
Segment Tree



Note: 紫色為該段落的最大值

made by electron & free999 & a711186

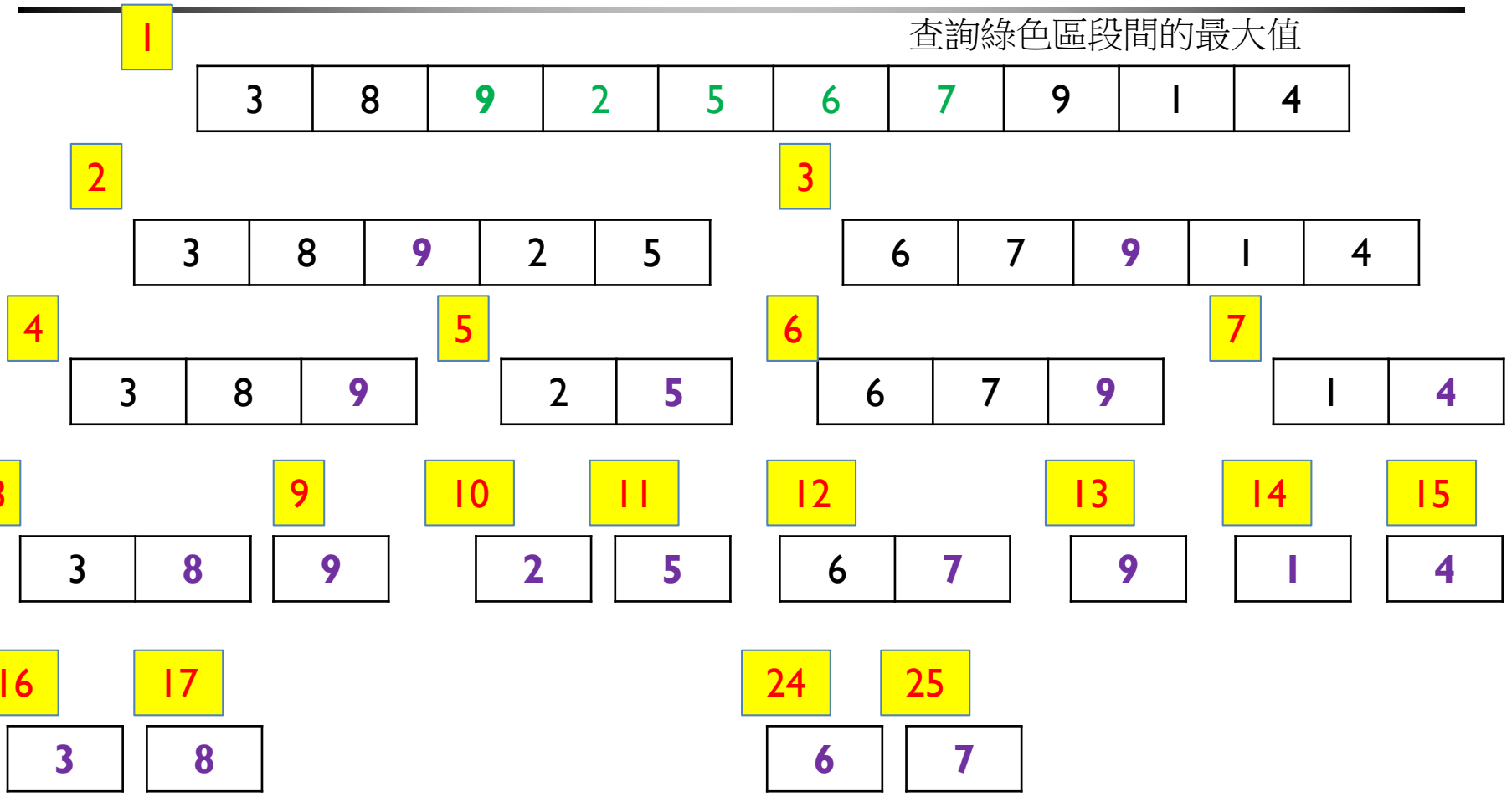
Segment Tree



Note: 紫色為該段落的最大值

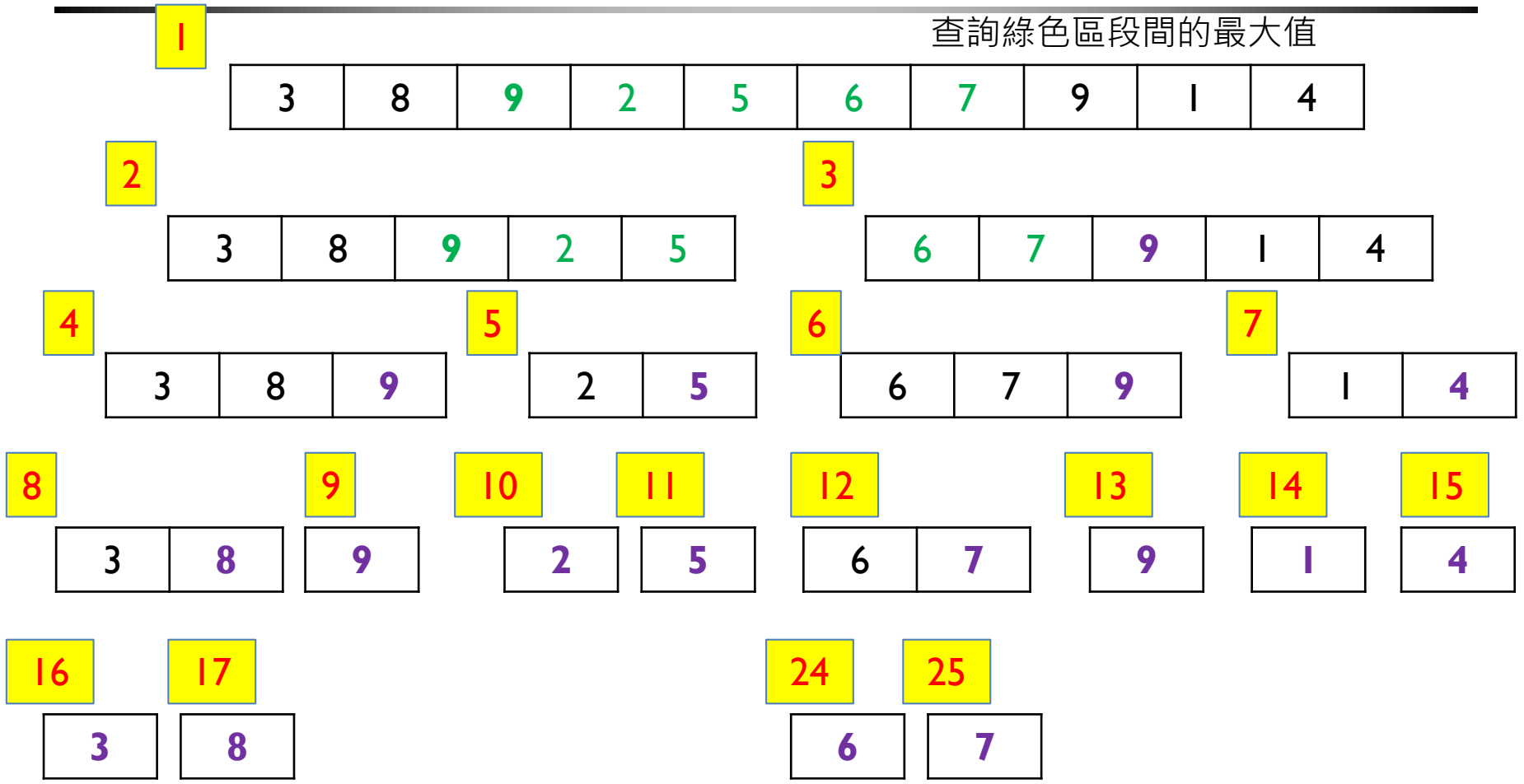
made by electron & free999 & a711186

Segment Tree



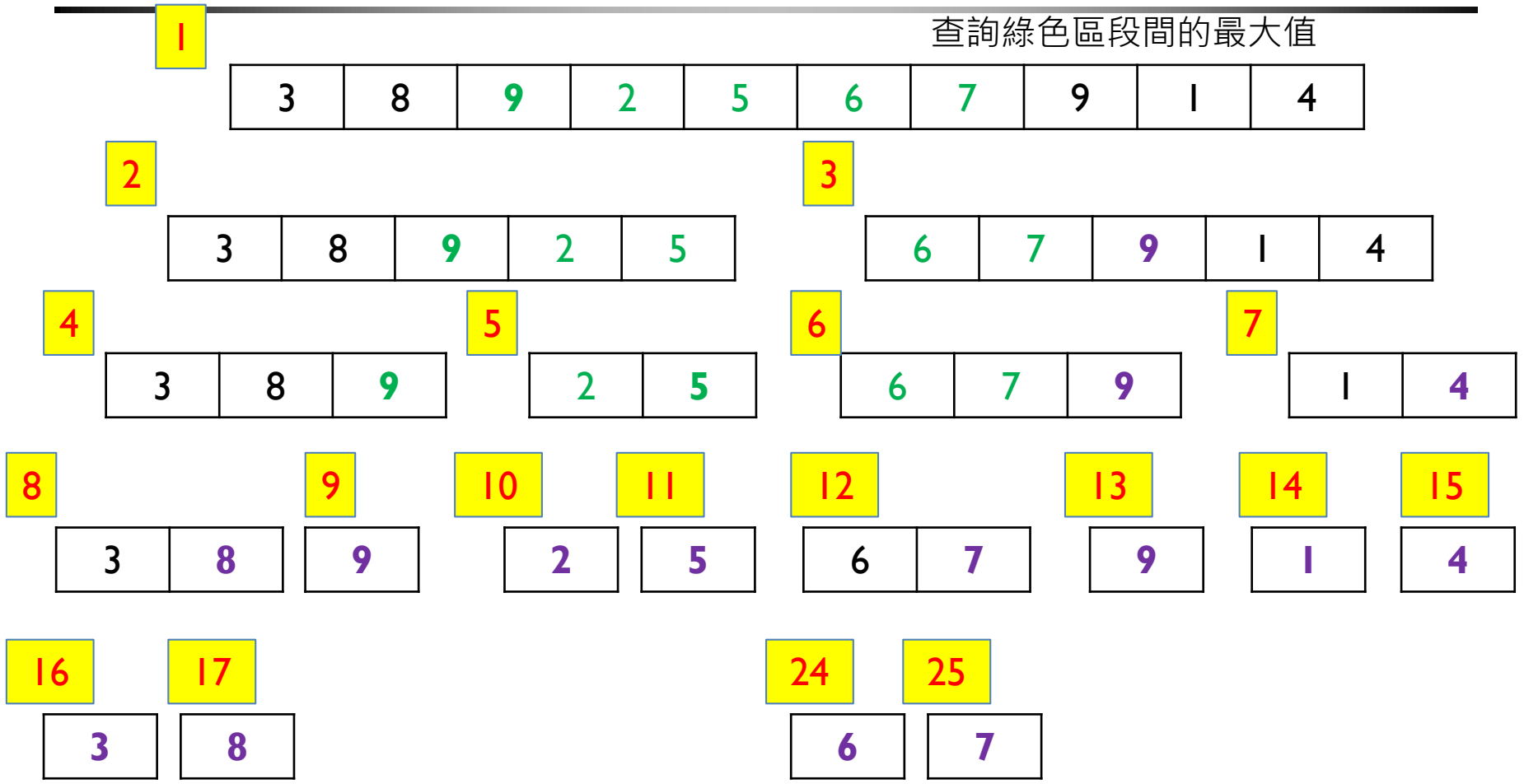
made by electron & free999 & a711186

Segment Tree



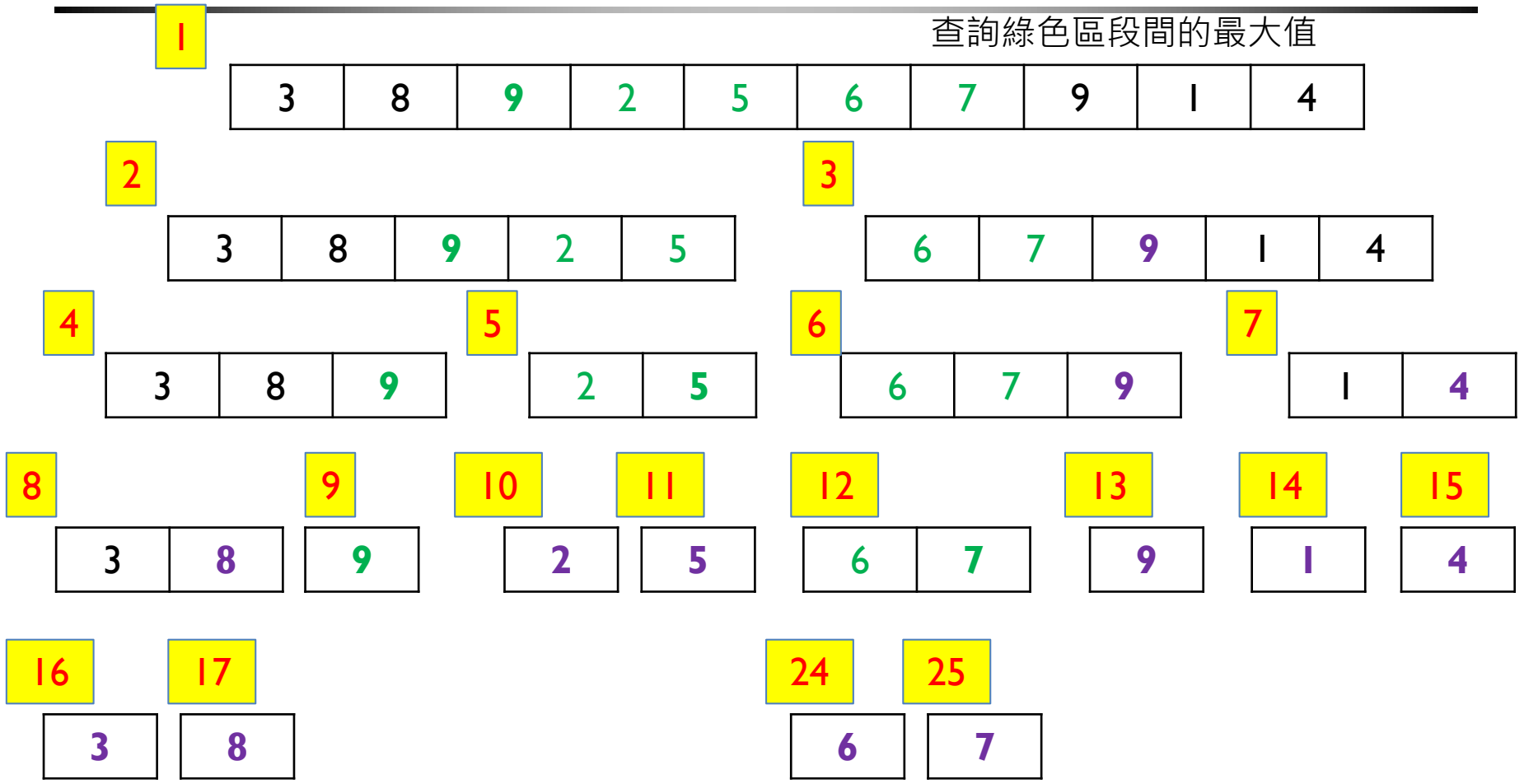
made by electron & free999 & a711186

Segment Tree



made by electron & free999 & a711186

Segment Tree



made by electron & free999 & a711186

Segment Tree

- 資料結構
  - 已找區間最大值為例
- 實作
  - **建立 build**

    bottom up建立線段樹的初始狀態
  - **修改 cover**

    修改線段樹，又可分為單點修改或區間修改
  - **查詢 query**

    對線段樹查詢區間 [L , R]

```
4   struct Node {
5       int val; // maximum in the segment
6       Node *l, *r;
7
8       void update(int update_val) {
9           val = update_val;
10      }
11
12      void pull() { // pull the maximum
13          val = max(l->val, r->val);
14      }
15  };
```

Segment Tree

- 建立 Build
  1) 我是葉子嗎
  2) 建立左子樹
  3) 建立右子樹
  4) pull

```
17  Node *build(int L, int R) {
18      // build this node
19      Node *now = new Node();
20
21      if (L == R) { // this is leaf
22          now->update(s[L]);
23          return now;
24      }
25
26      int mid = (L+R) >> 1;
27      // build left subtree
28      now->l = build(L, mid);
29      // build right subtree
30      now->r = build(mid+1, R);
31      // pull the maximum
32      now->pull();
33      return now;
34  }
```

Segment Tree

- 單點修改 cover
  1) 我是葉子嗎
  2) 修改包含 pos 的子樹
  3) pull

```
36  void cover(Node *now, int L, int R, int pos, int delta) {
37      if (L == R) { // this is leaf
38          now->update(now->val + delta);
39          return;
40      }
41
42      int mid = (L+R) >> 1;
43      // cover the segment containing node at pos
44      if (pos <= mid) {
45          cover(now->l, L, mid, pos, delta);
46      } else {
47          cover(now->r, mid+1, R, pos, delta);
48      }
49      // pull the maximum
50      now->pull();
51  }
```

Segment Tree

- 查詢 query
  1) [L , R] [x , y] 完全不重疊：return 負無限大
  2) [x , y] 包含 [L,R]
  3) Others：向左走向右走

```
53  int query_max(Node *now, int L, int R, int x, int y) {
54      // query boundary over the range of the segment, [x y L R] or [L R x y]
55      if (x>R || y<L)
56          return -INF;
57      // query boundary around the segment, [x L R y]
58      if (x<=L && y>=R)
59          return now->val;
60
61      int mid = (L+R) >> 1;
62      return max(query_max(now->l, L, mid, x, y), query_max(now->r, mid+1, R, x, y));
63  }
```

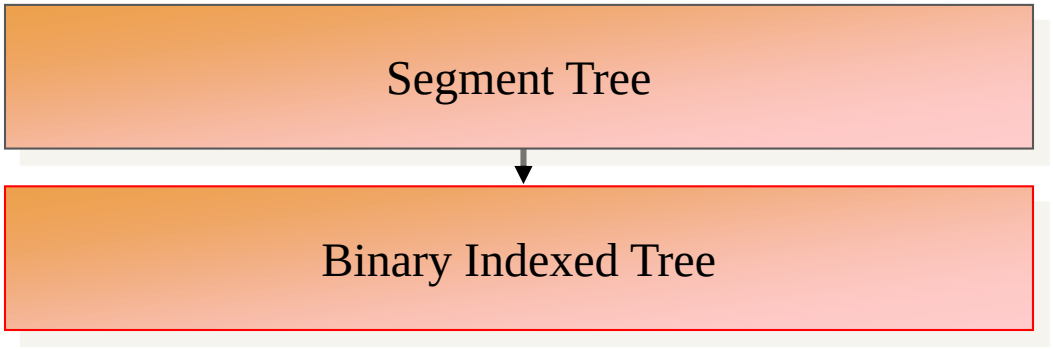
Segment Tree

- 時間複雜度
  - 建構線段樹 O(N)
  - 查詢和修改 O(log N)
  - 共 Q 個操作，總複雜度為 O(N+Q log N)
  - 空間複雜度為 O(N)

Example

POJ-3264 (link)

Outline



made by electron & free999 & a711186

Binary Indexed Tree

- Operation
  - 可以快速算出任意區間的總和、更新數值
  - 不能插入、刪除數值。

- Naïve Solution
  - 陣列長度 = N
  - Query 數 = Q
  - 求區間總和，時間複雜度 **O(QN)**…

Binary Indexed Tree

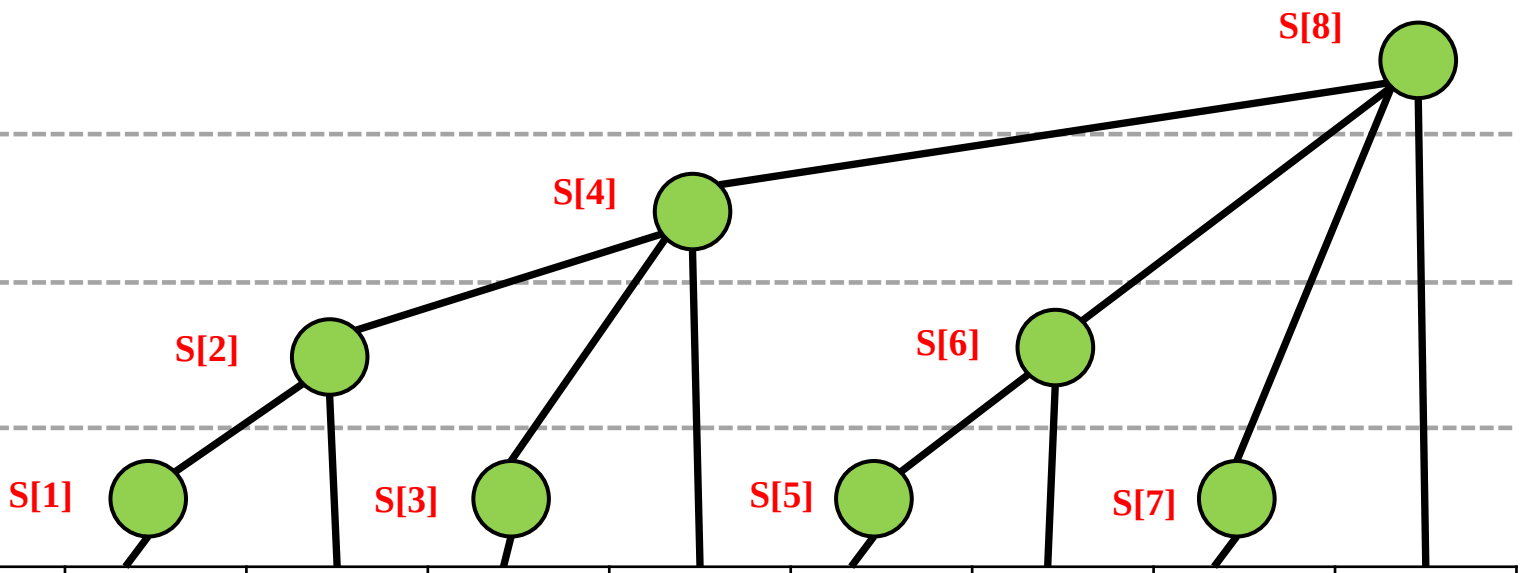
- Fenwick Tree
- Fenwick trees are particularly designed to implement the arithmetic coding algorithm, which maintains counts of each symbol produced and needs to convert those to the cumulative probability of a symbol less than a given symbol.
- Although Fenwick trees are trees in concept, in practice they are implemented as an implicit data structure using a flat array analogous to implementations of a binary heap
- 所有的整數都可以表示成2的冪和，我們也可以把一串序列表示成一系列子序列的和

Binary Indexed Tree

- ori[x]: 原陣列
- c[x]: 從 index 1 到 index x 的元素總和
- s[x]: 建立 BIT 後的新陣列
  - s[0] = 0

Binary Indexed Tree

4
3
2
1
0



ori[i]	5	8	-1	2	4	1	0	9
c[i]	5	13	12	14	18	19	19	28
i	1	2	3	4	5	6	7	8

S[i]	5	13	-1	14	4	5	0	28

Binary Indexed Tree

Define:

```
1    int lowbit (int in) {
2        return in & (-in);
3    }
```

ex:
lowbit(1) = 1 [0001]
lowbit(2) = 2 [0010]
lowbit(3) = 1 [0011]
lowbit(4) = 4 [0100]

	十進位	最高位元表示法	1的補數表示法		2的補數表示法	
正數	128	無	無		無	
	127	01111111	01111111		01111111	
	126	01111110	01111110		01111110	
	:	:	:		:	
	2	00000010	00000010		00000010	
	1	00000001	00000001		00000001	
	0	00000000	00000000		00000000	
負數	0	10000000	11111111		00000000	
	-1	10000001	11111110		11111111	
	-2	10000010	11111101		11111110	
	:	:	:		:	
	-126	11111110	10000001		10000010	
	-127	11111111	10000000		10000001	
	-128	無	無		10000000	

(互為1的補數)　(互為2的補數)

i	1	2	3	4	5	6	7	8
$i_{(2)}$	0001	0010	0011	0100	0101	0110	0111	01000
$-i_{(2)}$	1111	1110	1101	1100	1011	1010	1001	11000
lowbit(i)	1	2	1	4	1	2	1	8

Binary Indexed Tree

s[1] = ori[1]

s[2] = ori[2] + **s[1]**

s[3] = ori[3]

s[4] = ori[4] + **s[3] + s[2]**

s[5] = ori[5]

s[6] = ori[6] + **s[5]**

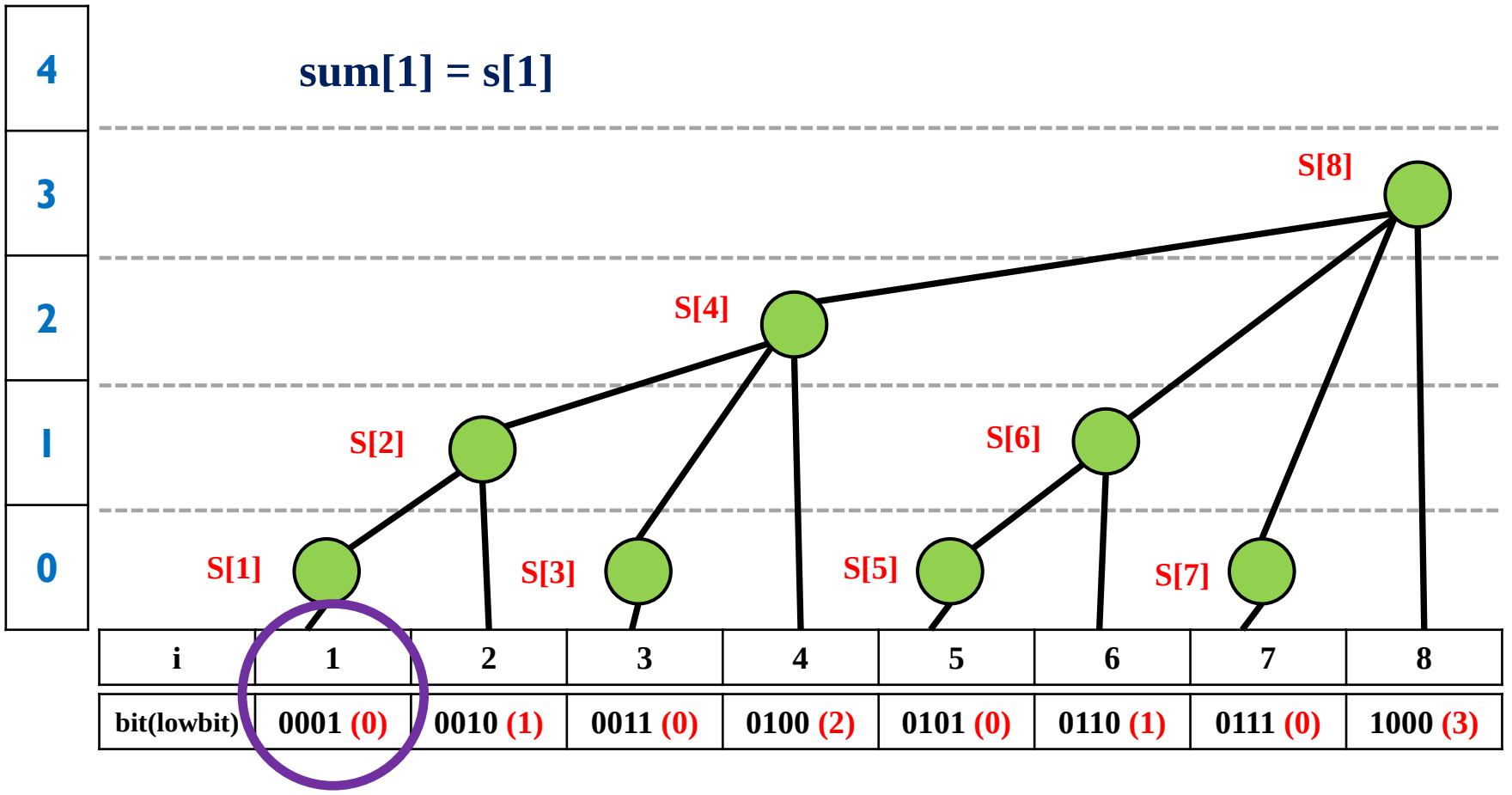
s[7] = ori[7]

s[8] = ori[8] + **s[7] + s[6] + s[4]**

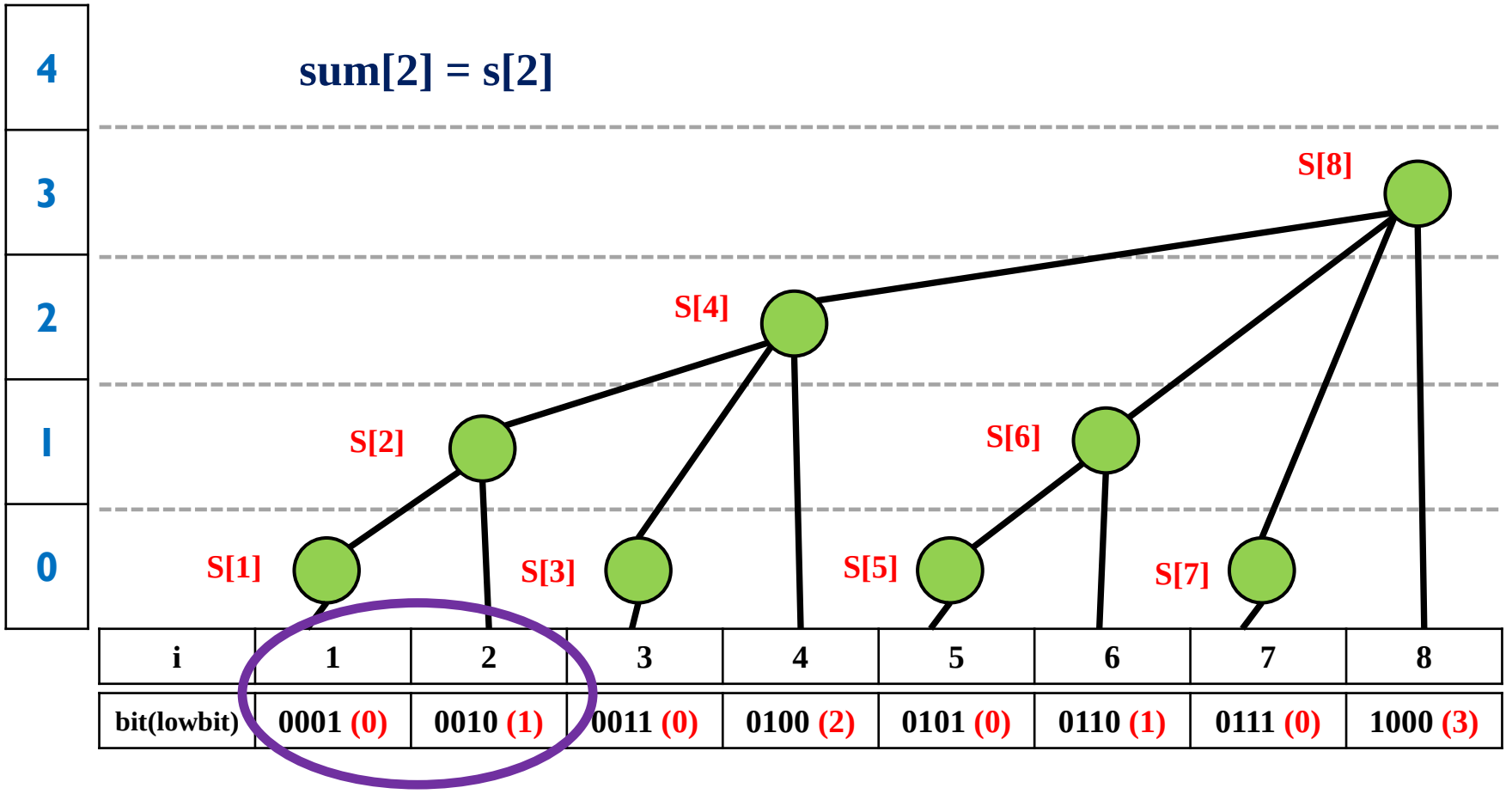
…

What's the regularity?

Binary Indexed Tree

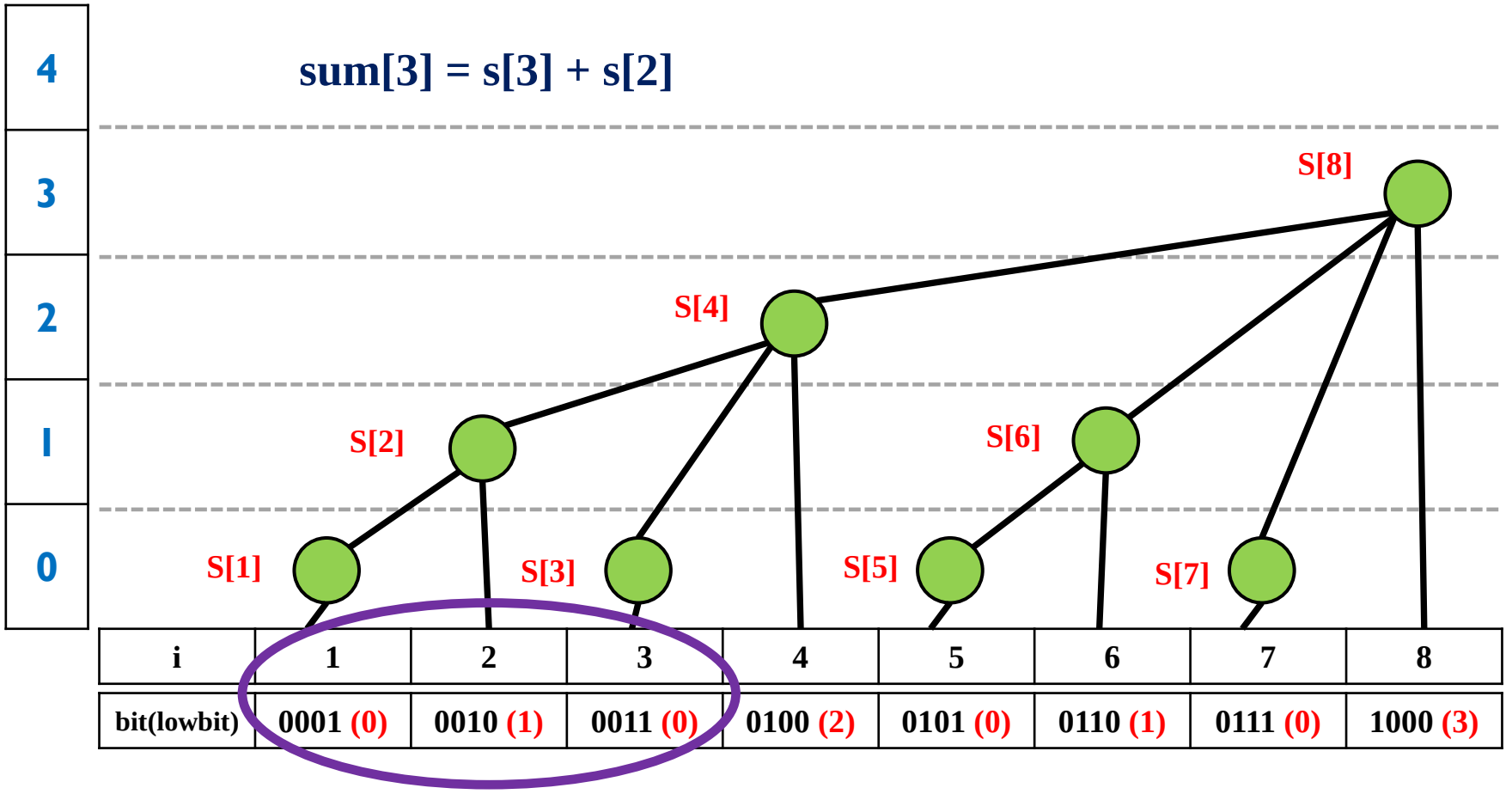


Binary Indexed Tree



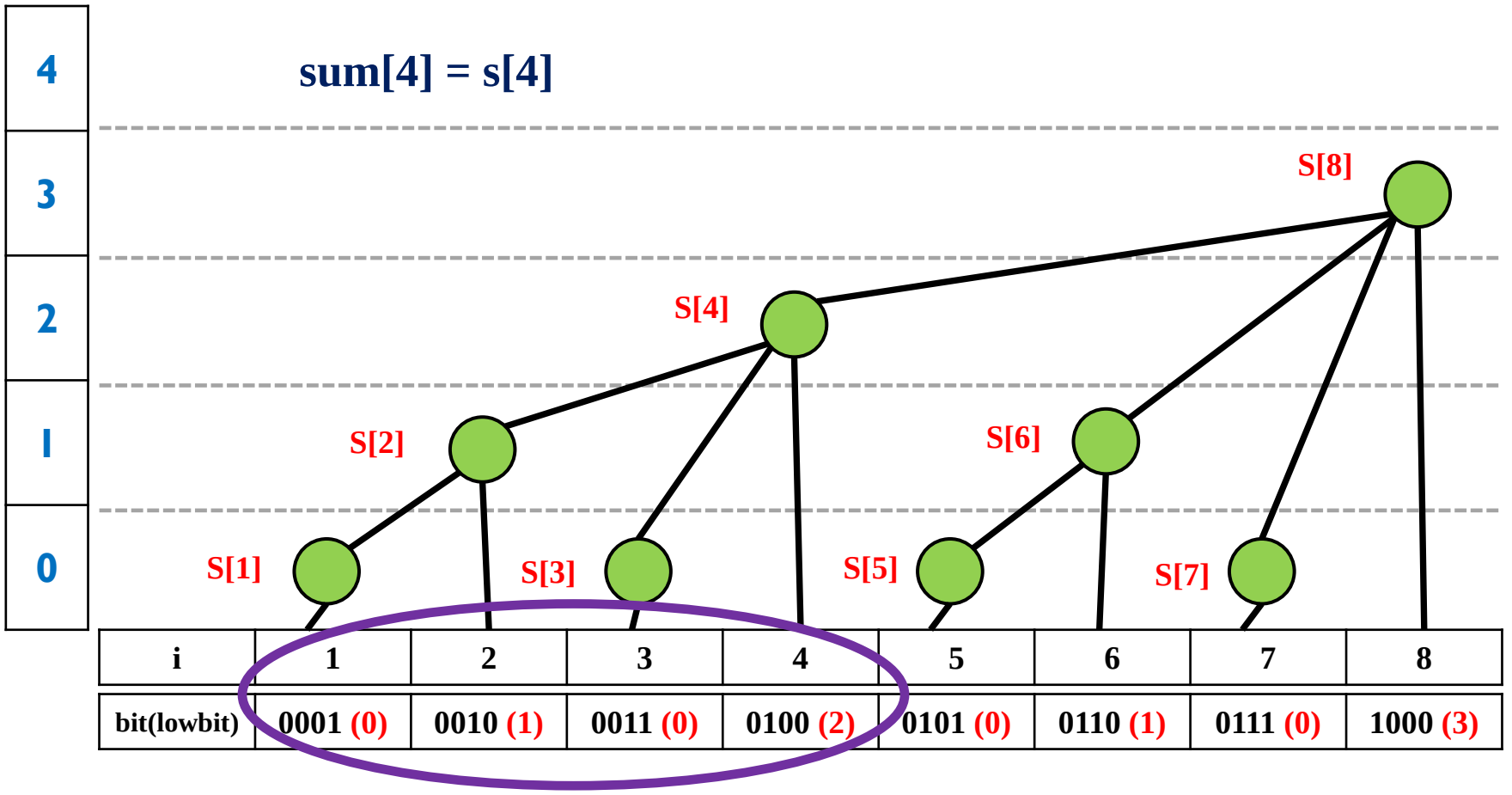
i	1	2	3	4	5	6	7	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

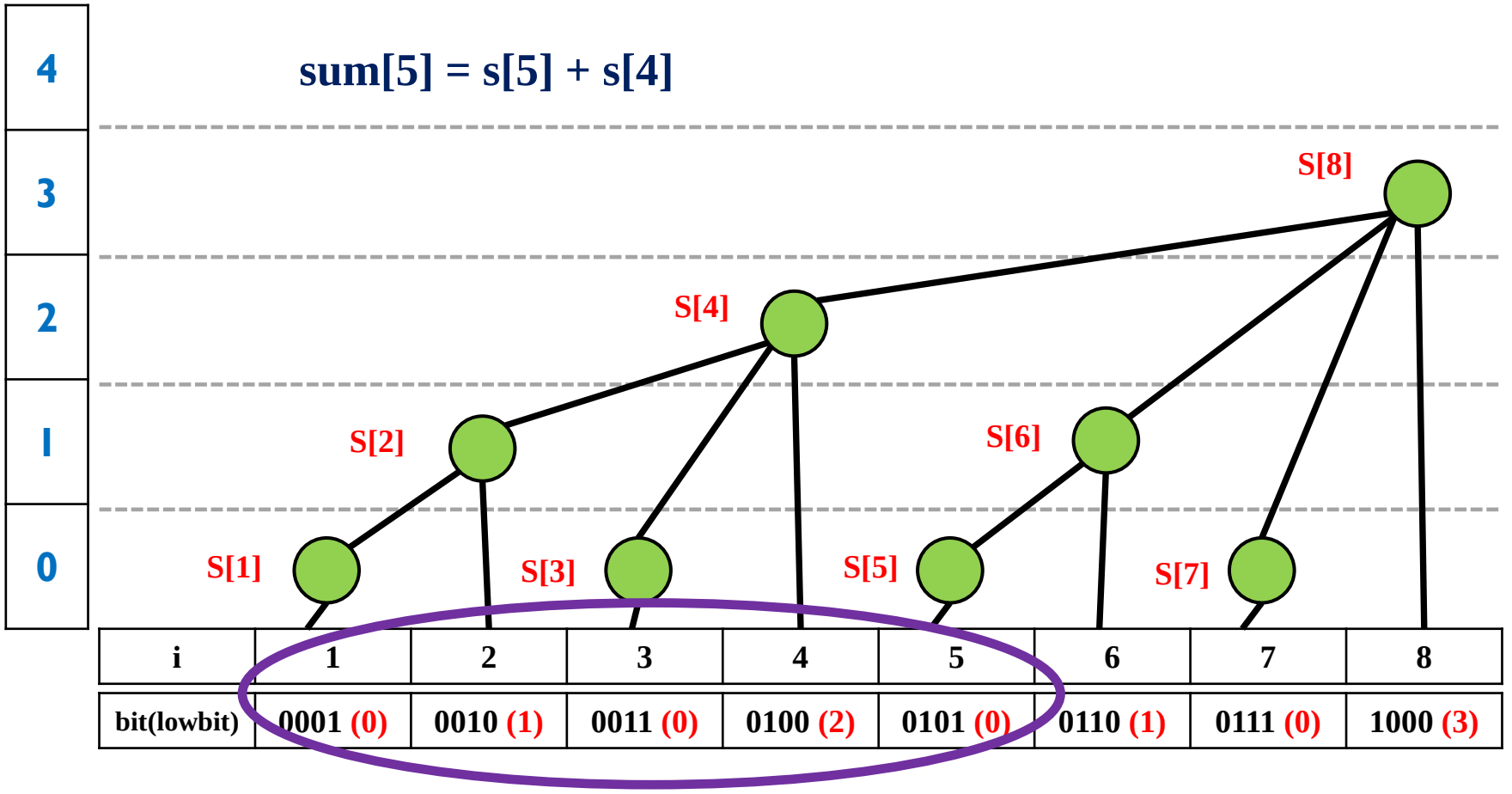


i	1	2	3	4	5	6	7	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

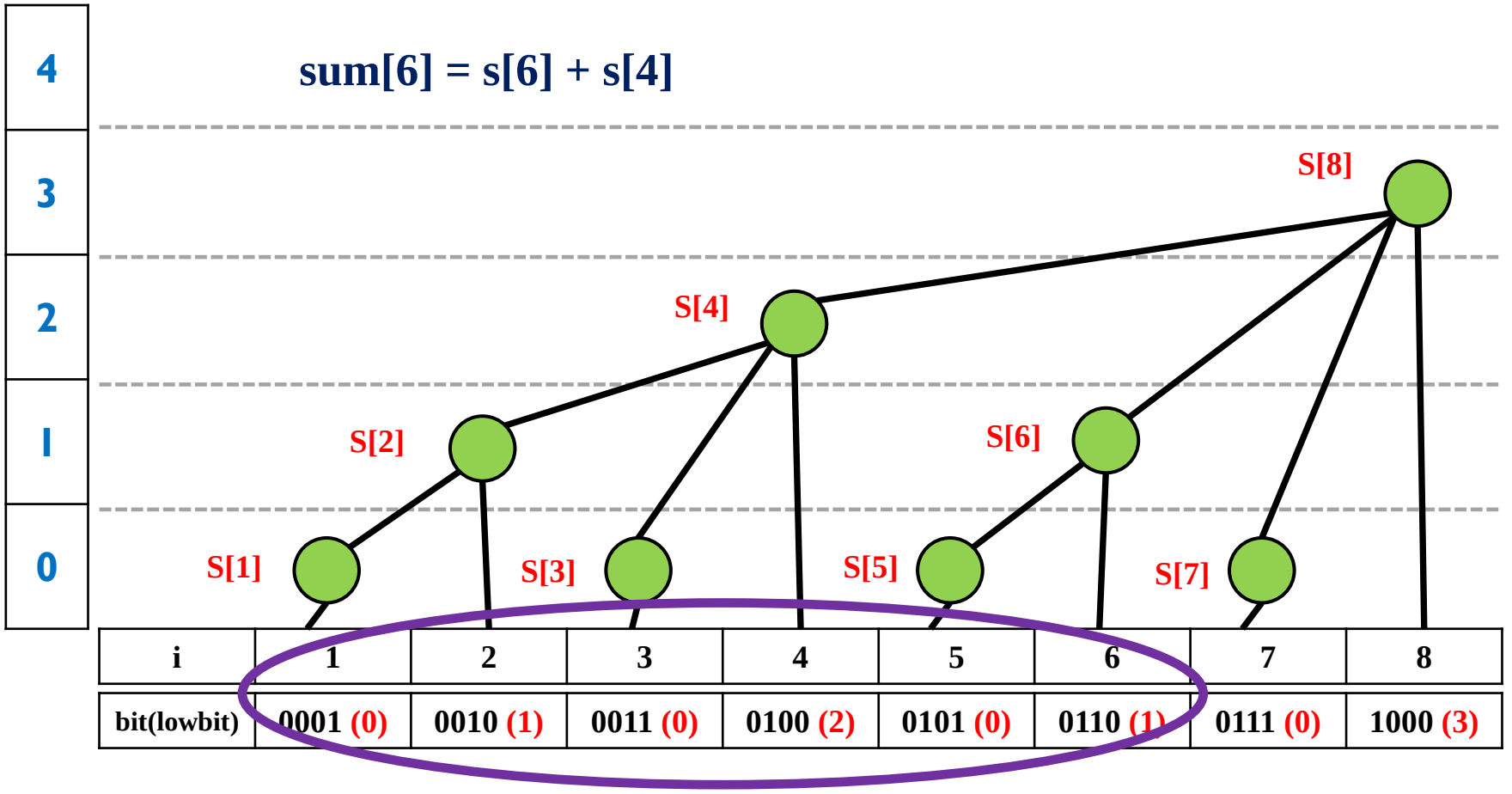


Binary Indexed Tree

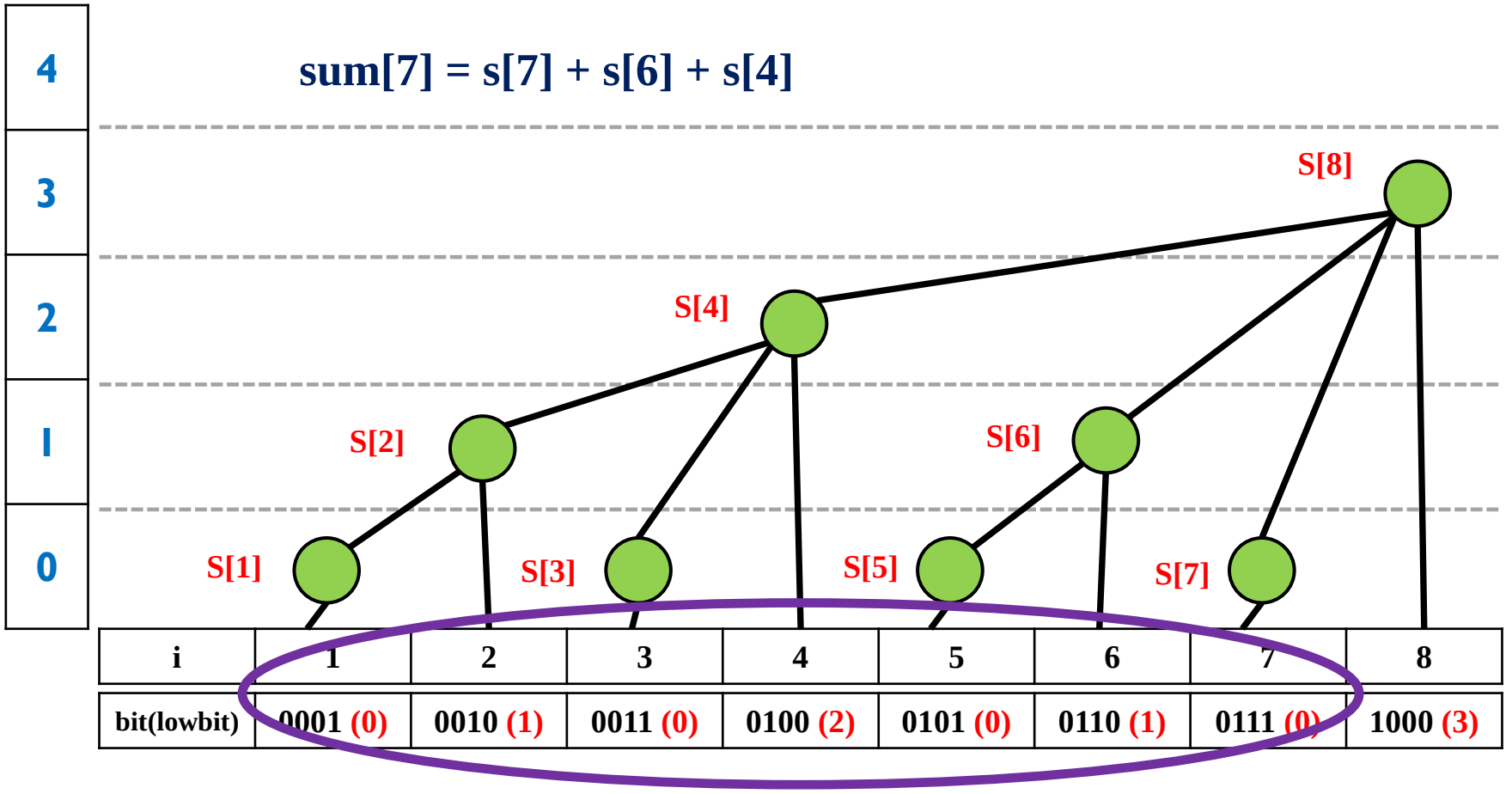


i	1	2	3	4	5	6	7	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

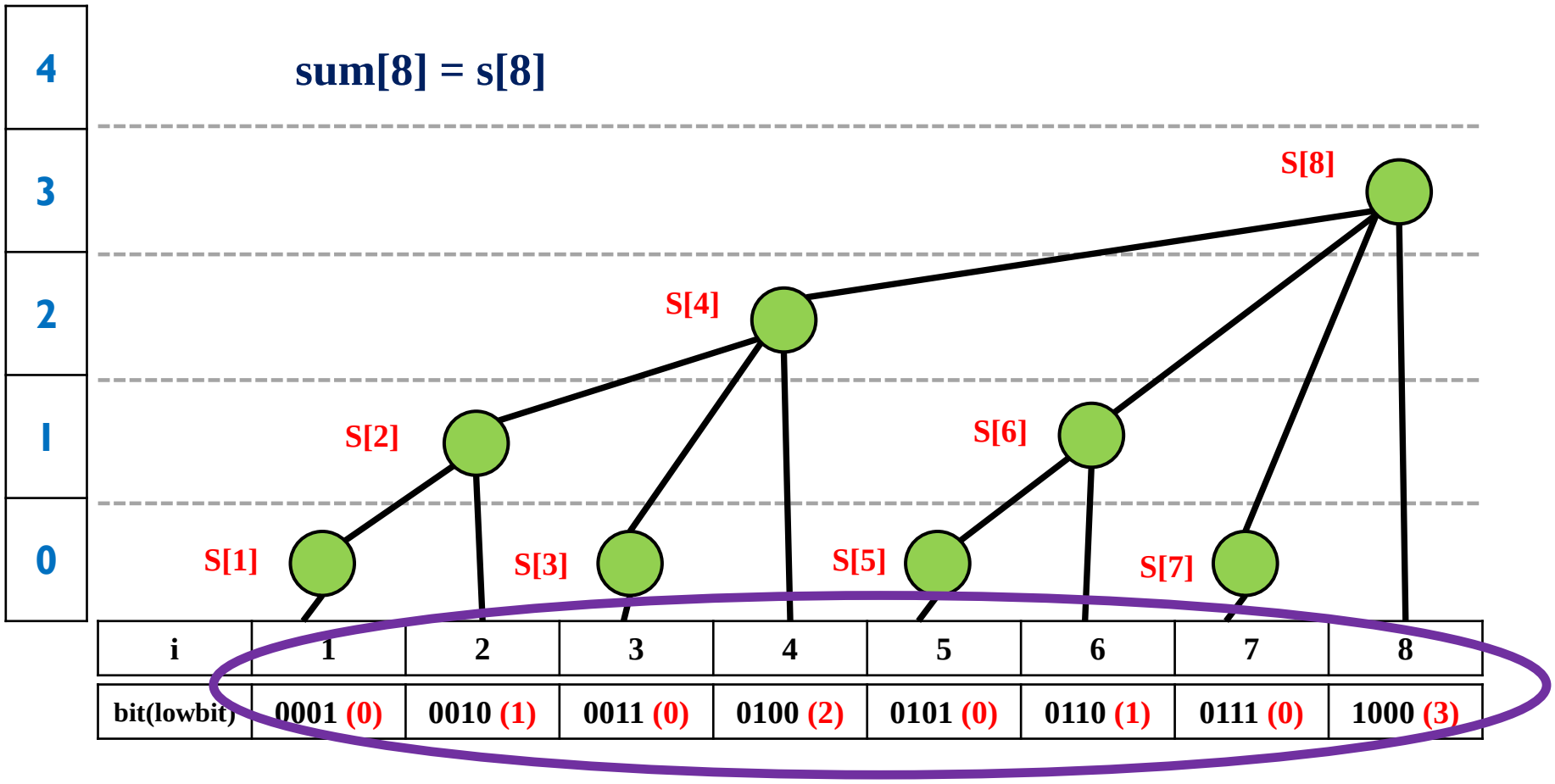
Binary Indexed Tree



Binary Indexed Tree



Binary Indexed Tree

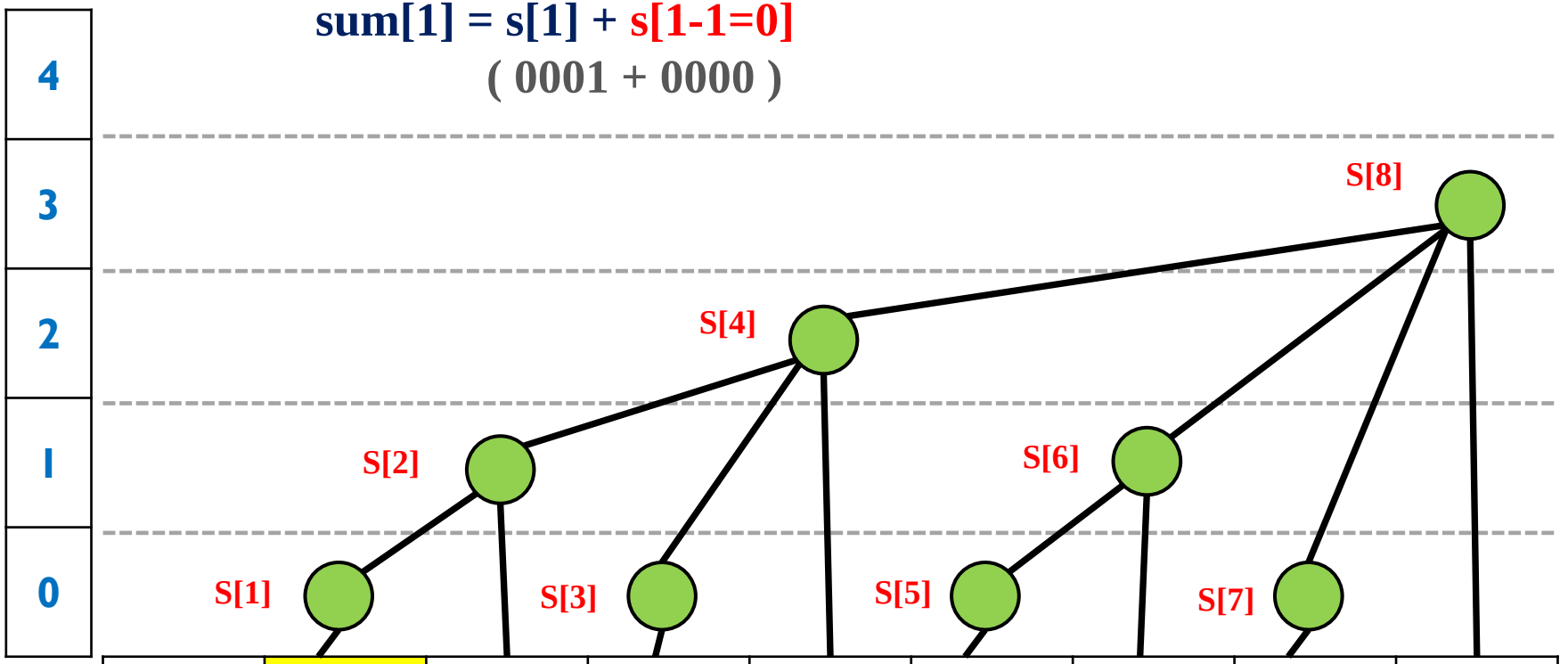


i	1	2	3	4	5	6	7	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[1] = s[1] + s[1-1=0]

(0001 + 0000)

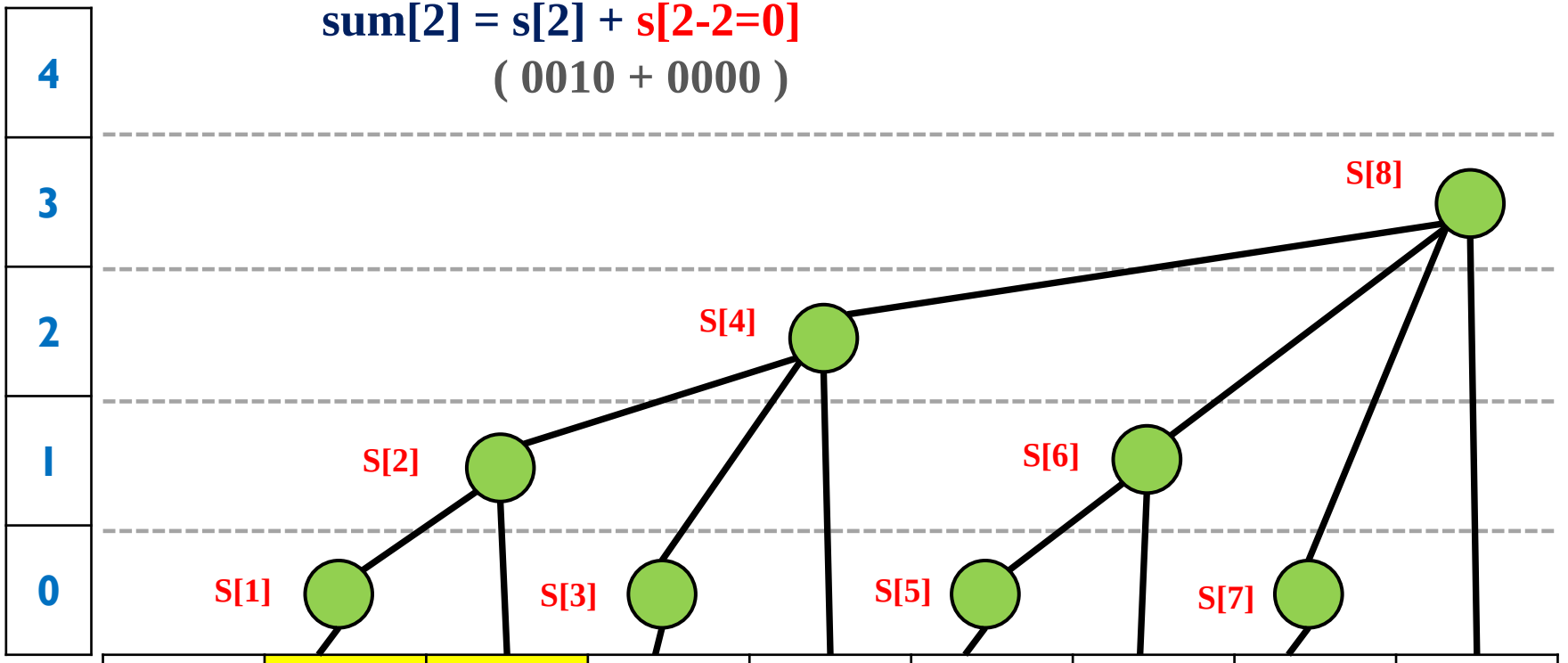


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[2] = s[2] + s[2-2=0]

(0010 + 0000)

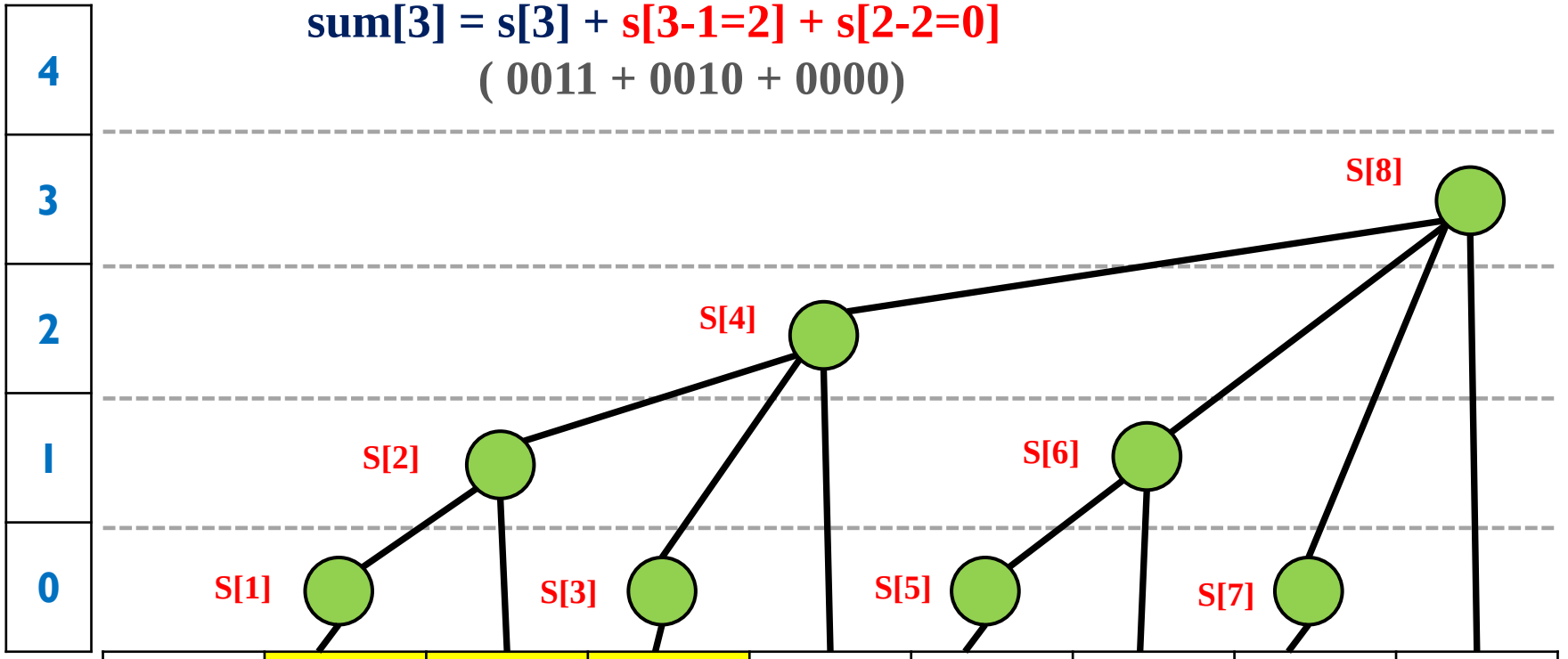


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[3] = s[3] + s[3-1=2] + s[2-2=0]

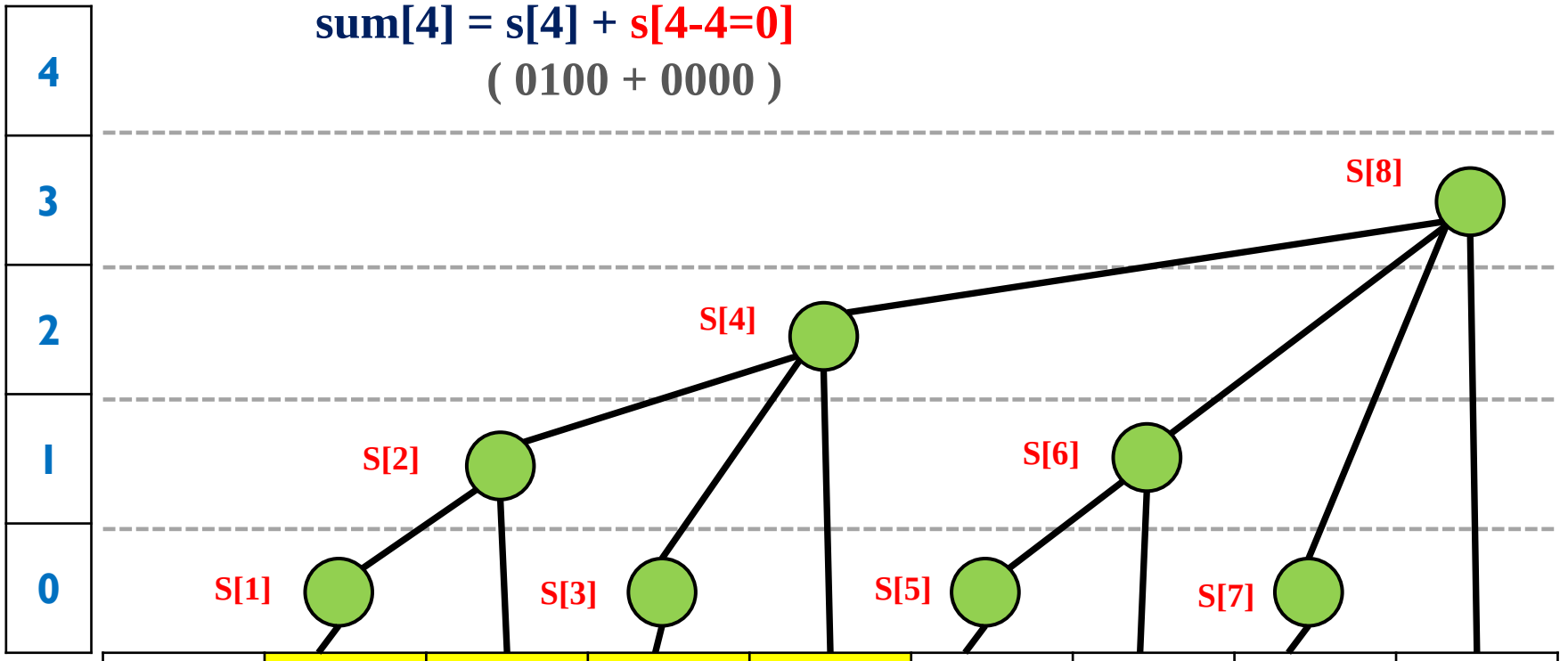
(0011 + 0010 + 0000)



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[4] = s[4] + s[4-4=0]
(0100 + 0000)

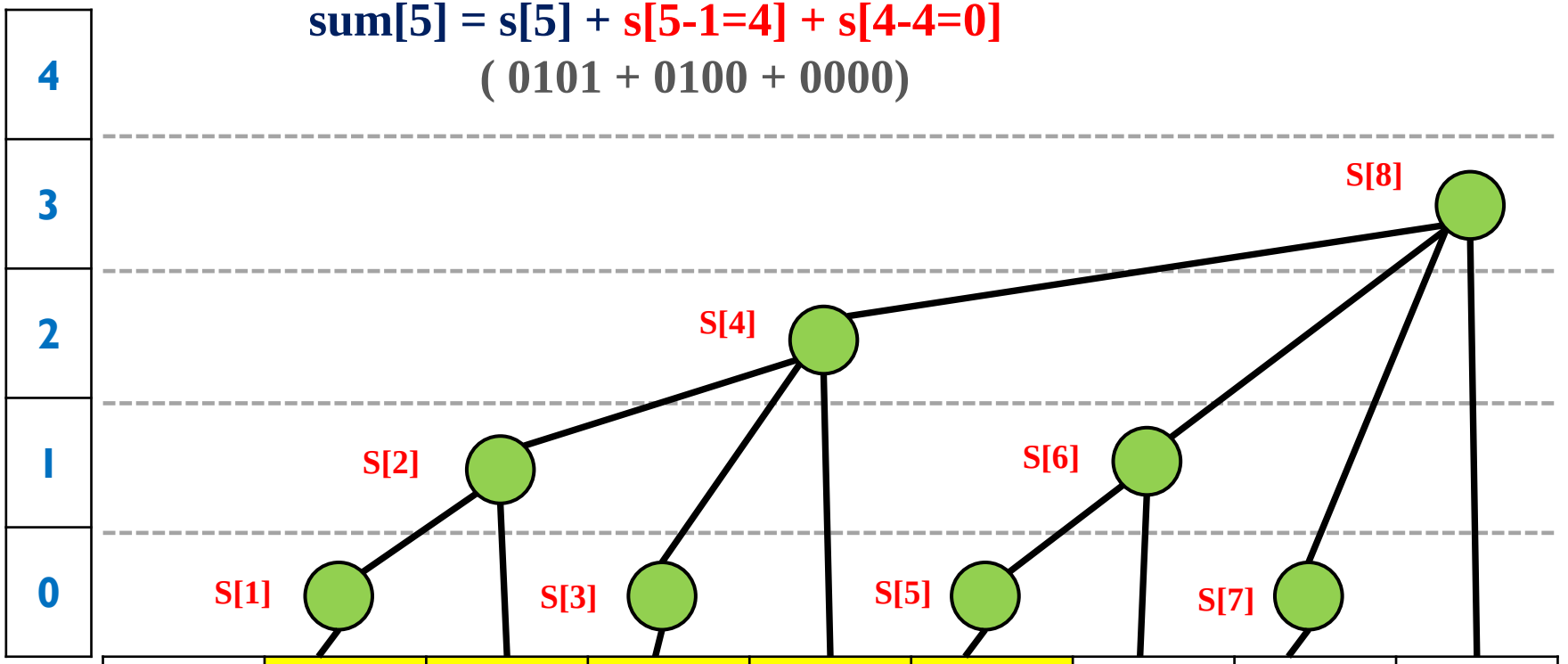


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[5] = s[5] + s[5-1=4] + s[4-4=0]

(0101 + 0100 + 0000)

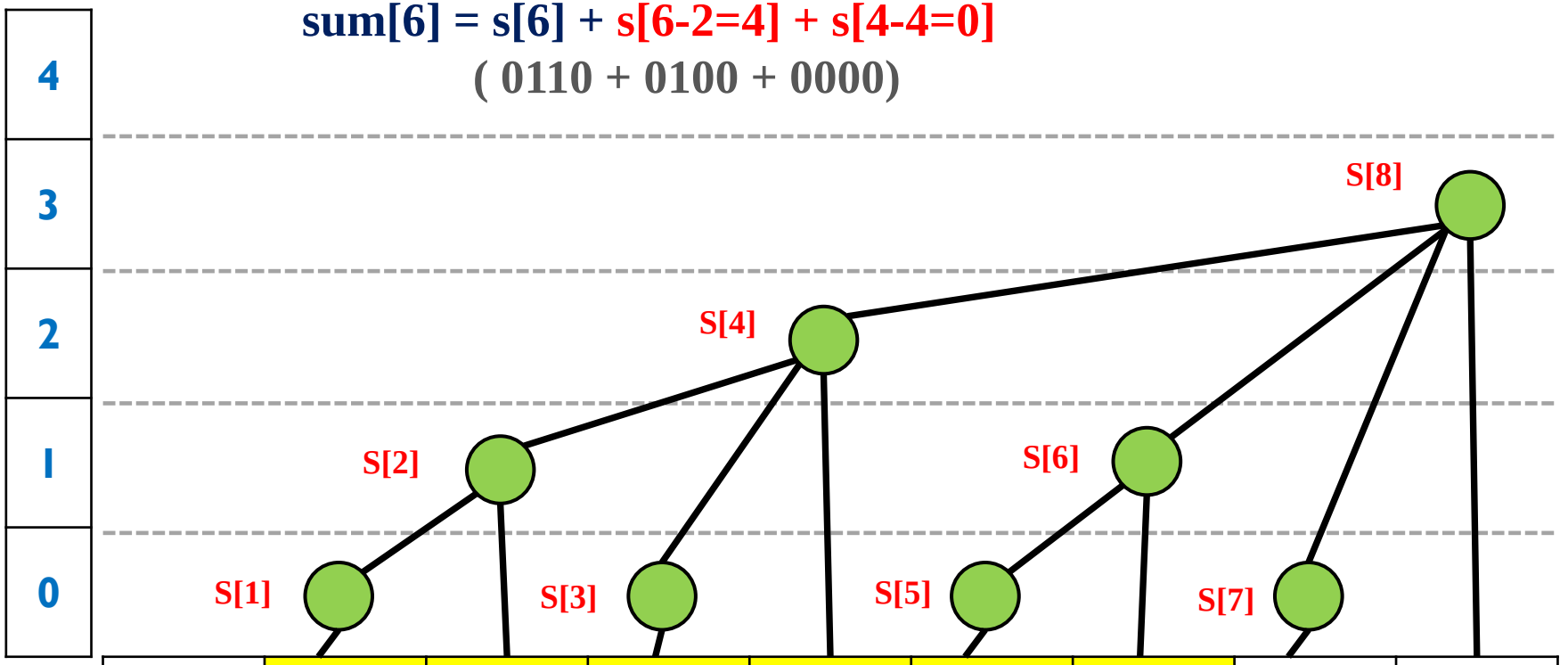


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[6] = s[6] + s[6-2=4] + s[4-4=0]

(0110 + 0100 + 0000)

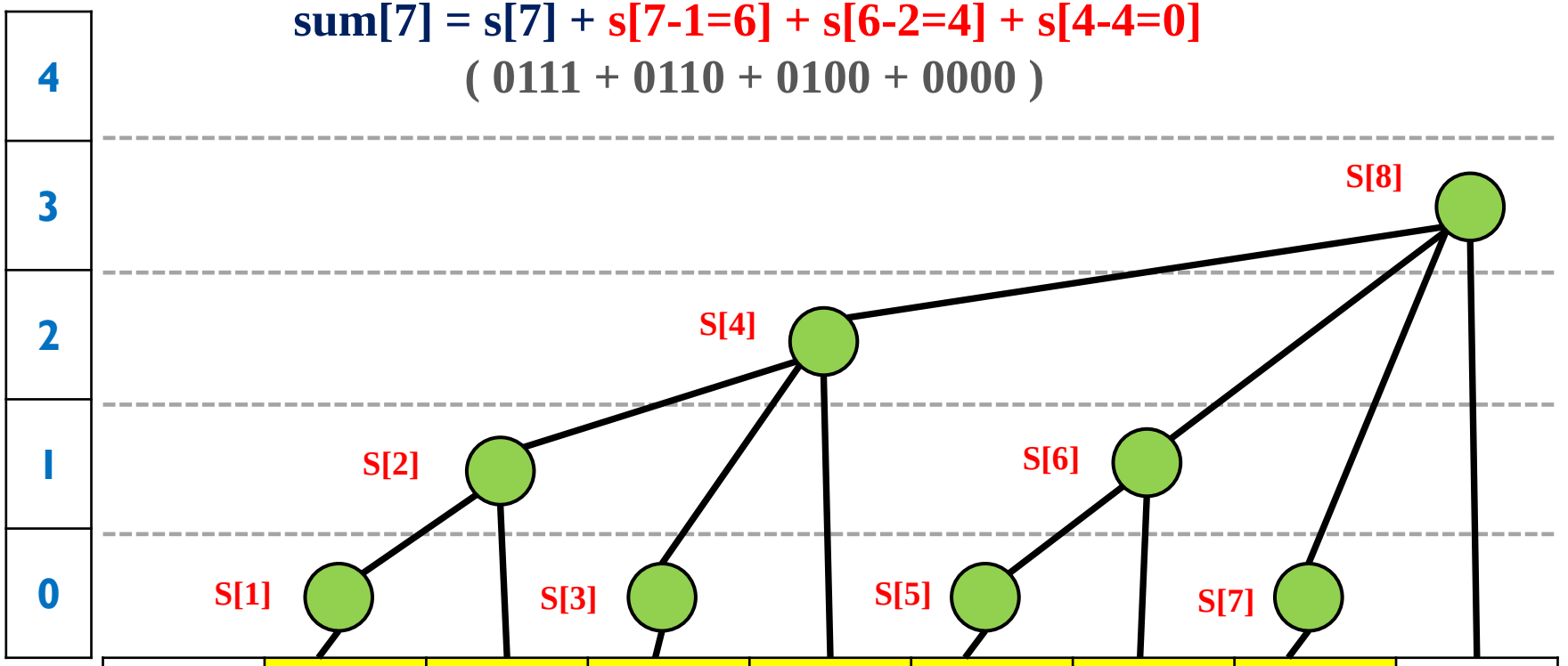


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[7] = s[7] + s[7-1=6] + s[6-2=4] + s[4-4=0]

(0111 + 0110 + 0100 + 0000)

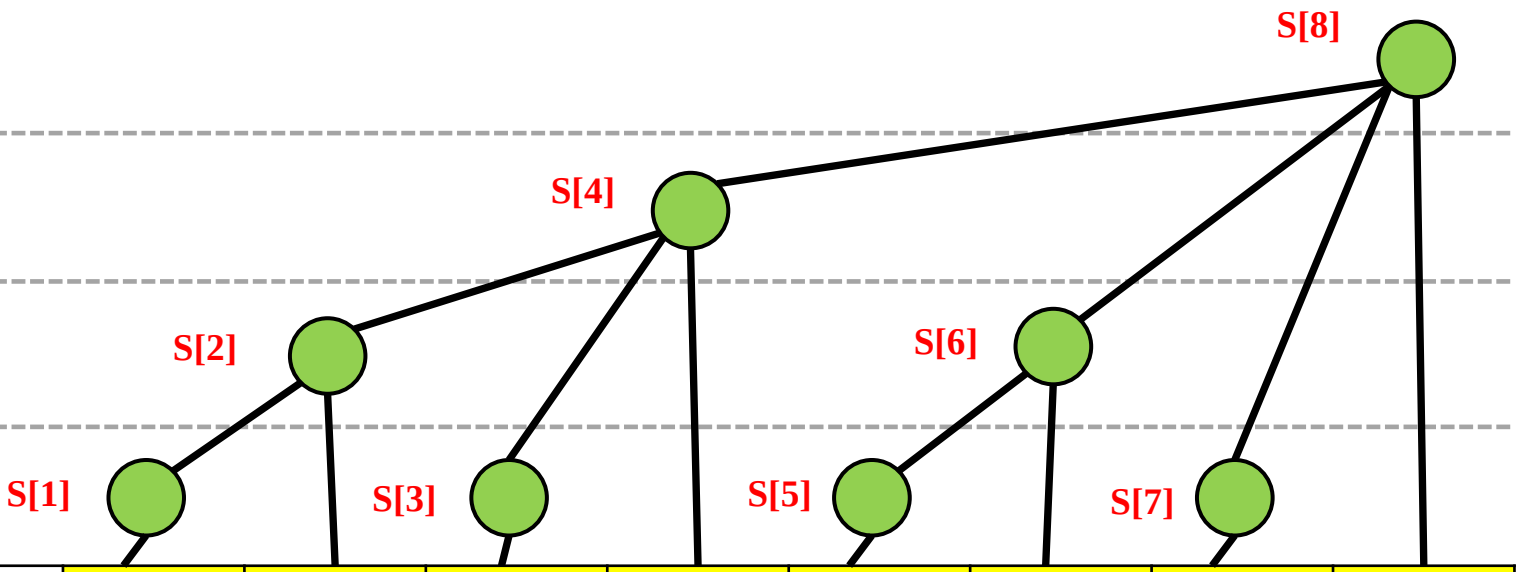


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

sum[8] = s[8] + s[8-8=0]
(1000 + 0000)

4
3
2
1
0

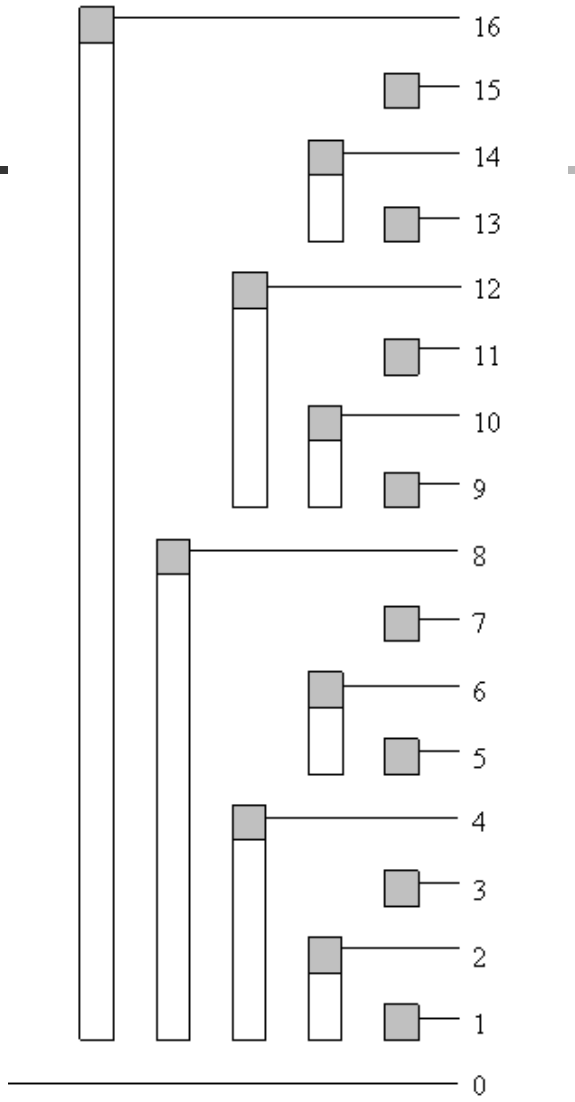


i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8
bit(lowbit)	0001 (0)	0010 (1)	0011 (0)	0100 (2)	0101 (0)	0110 (1)	0111 (0)	1000 (3)

Binary Indexed Tree

```
1   int lowbit (int in) {
2       return in & (-in);
3   }
4
5   int get_sum (int end) {
6       int ans = 0;
7       while(end > 0) {
8           ans += s[end];
9           end -= lowbit(end);
10      }
11  }
```

made by electron & free999 & a711186



ibility for indexes (bar shows range of frequencies accum

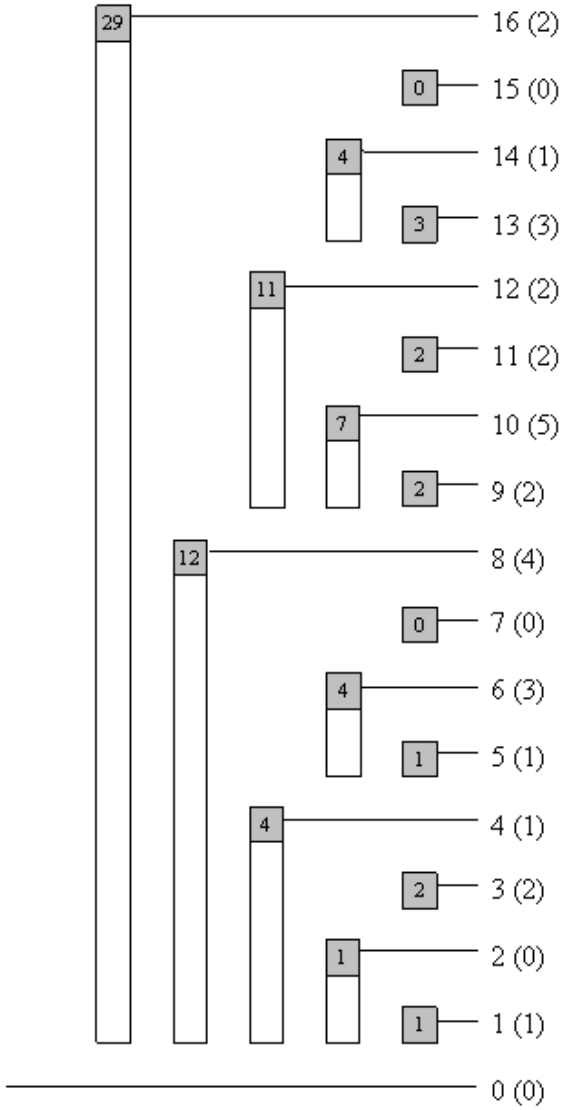
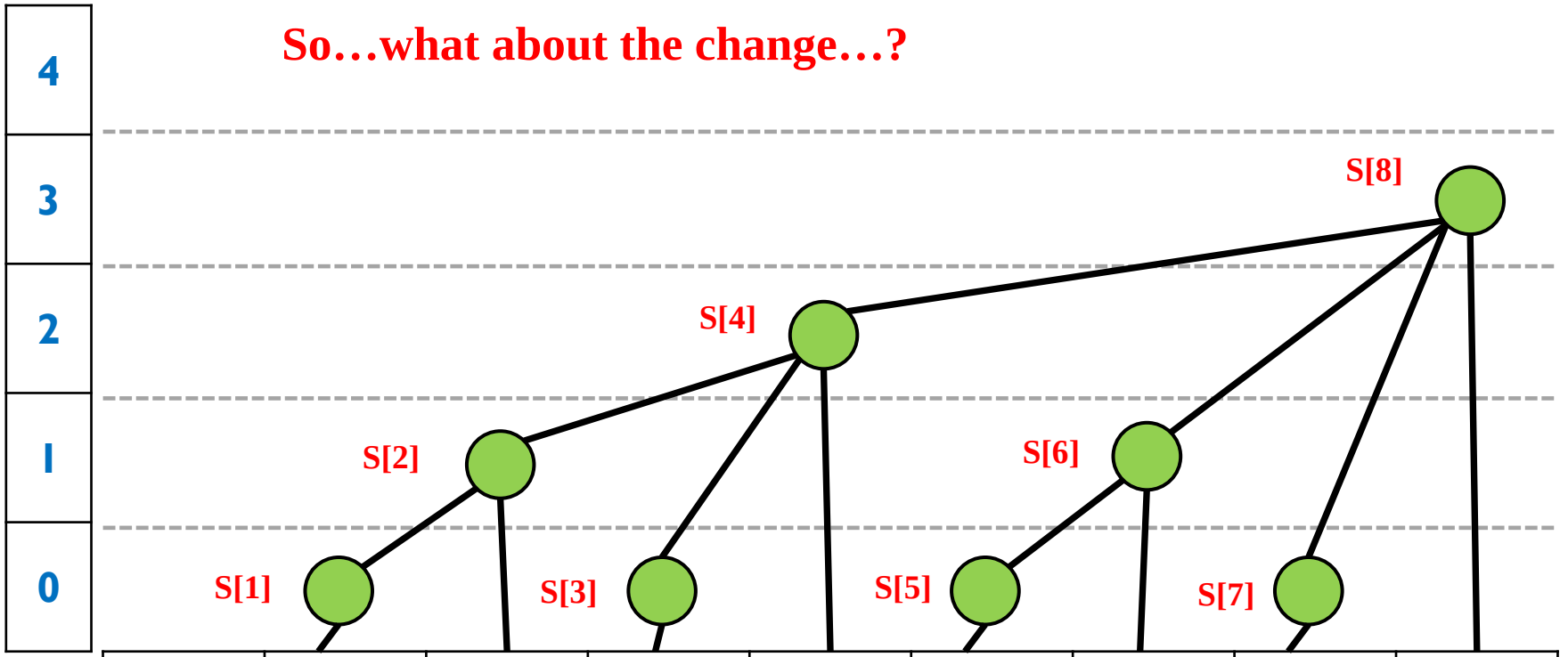


Image 1.4 - tree with tree frequencies

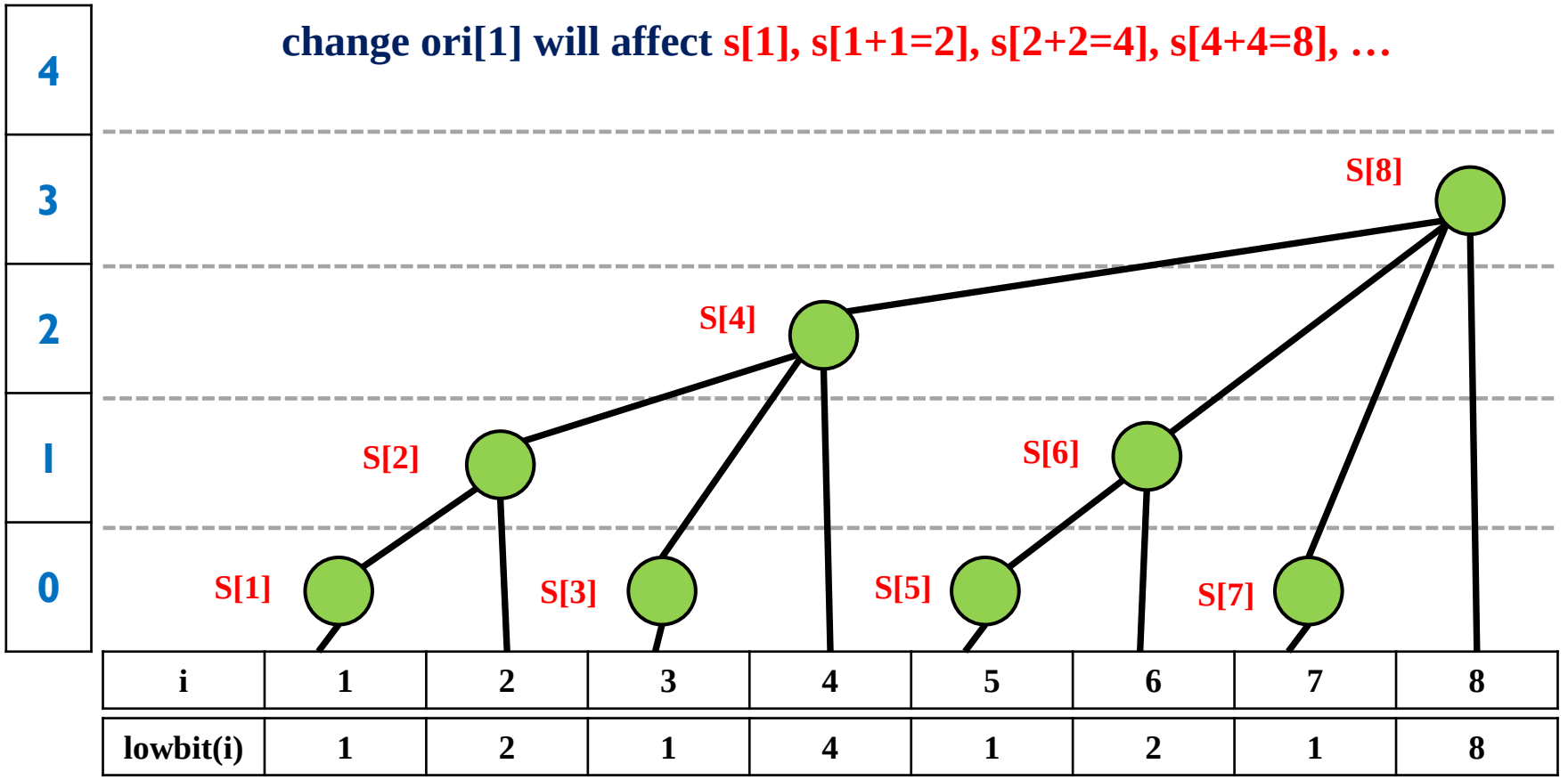
Binary Indexed Tree



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8

made by electron & free999 & a711186

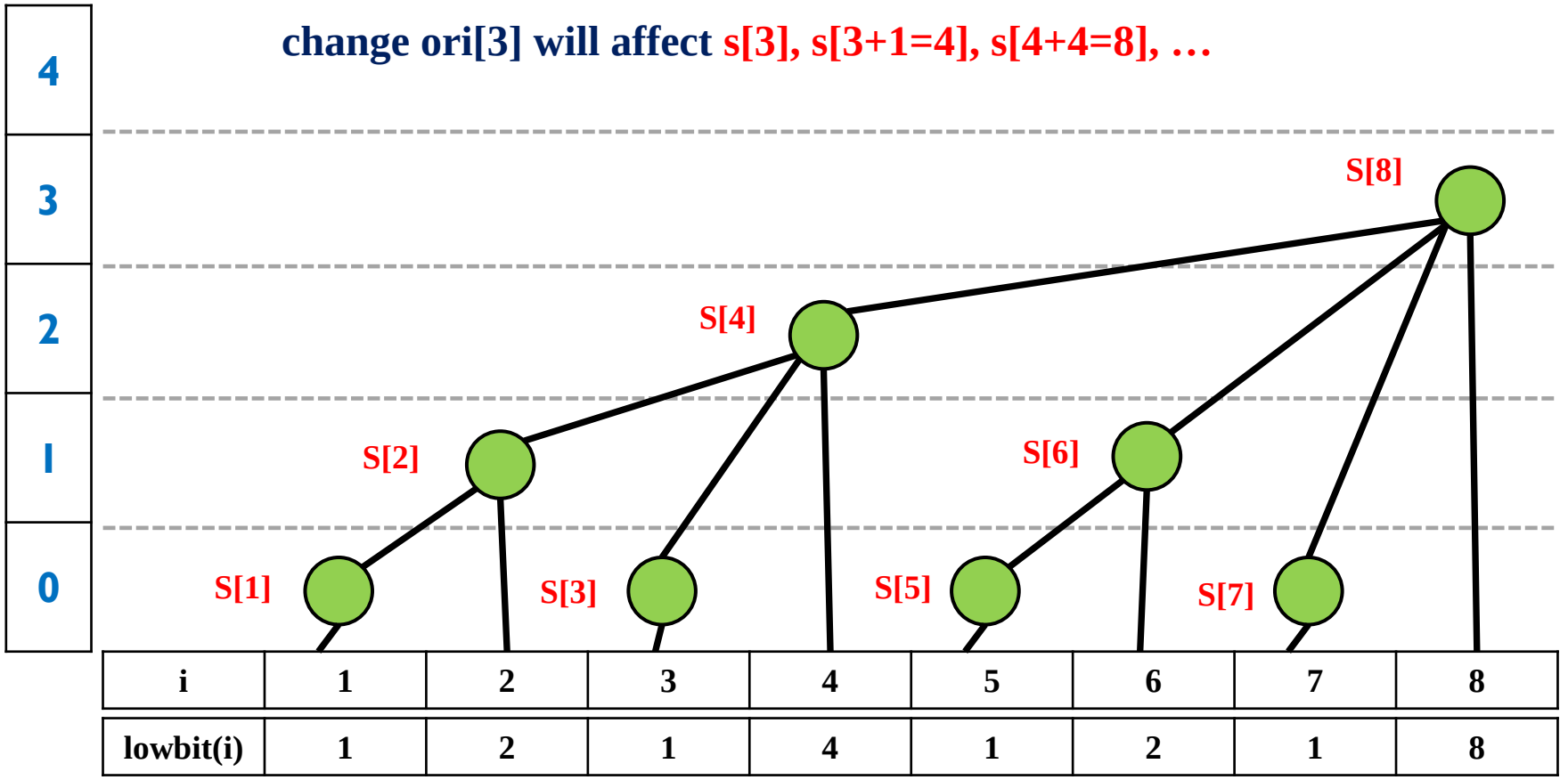
Binary Indexed Tree



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8

made by electron & free999 & a711186

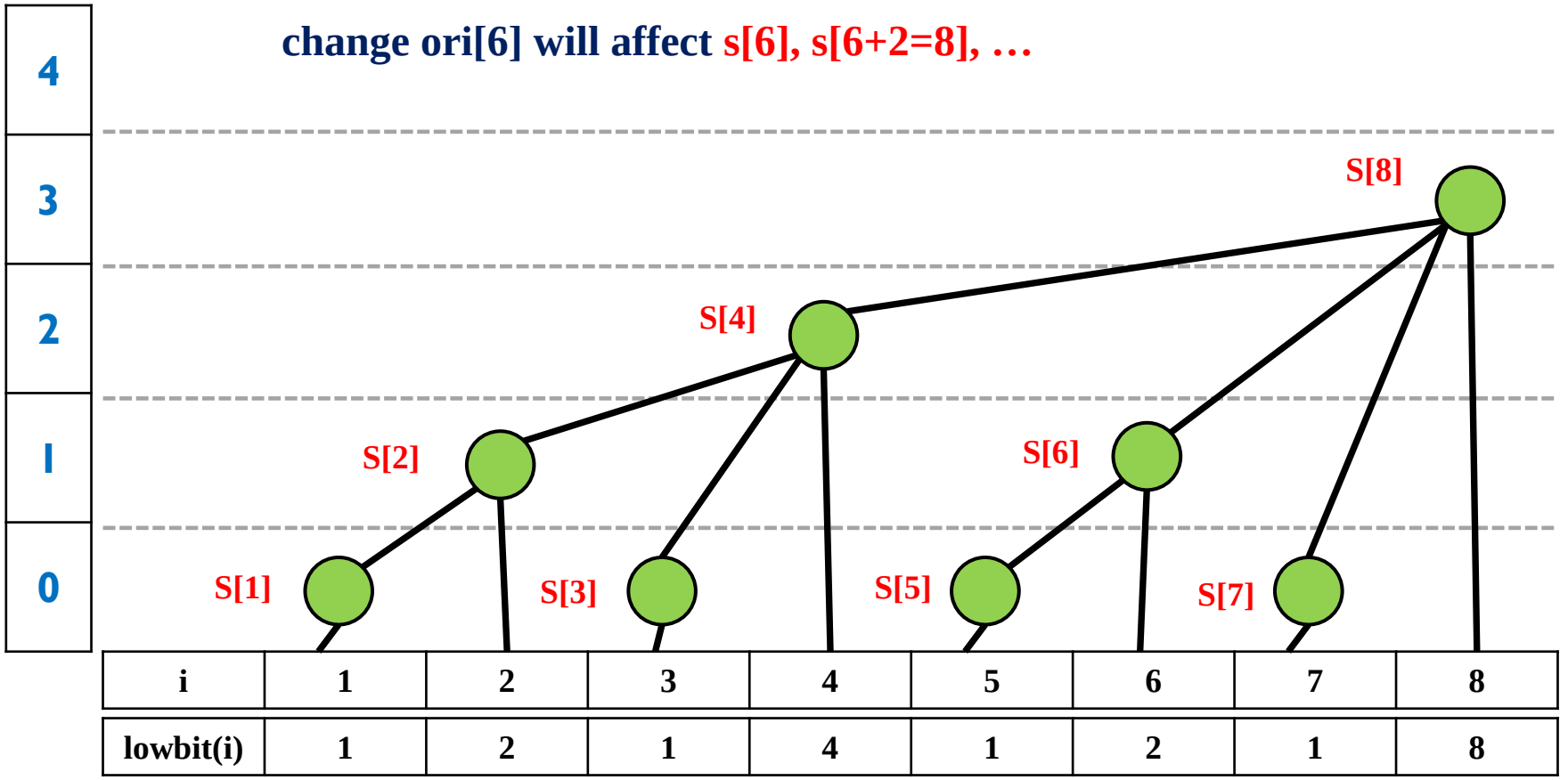
Binary Indexed Tree



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8

made by electron & free999 & a711186

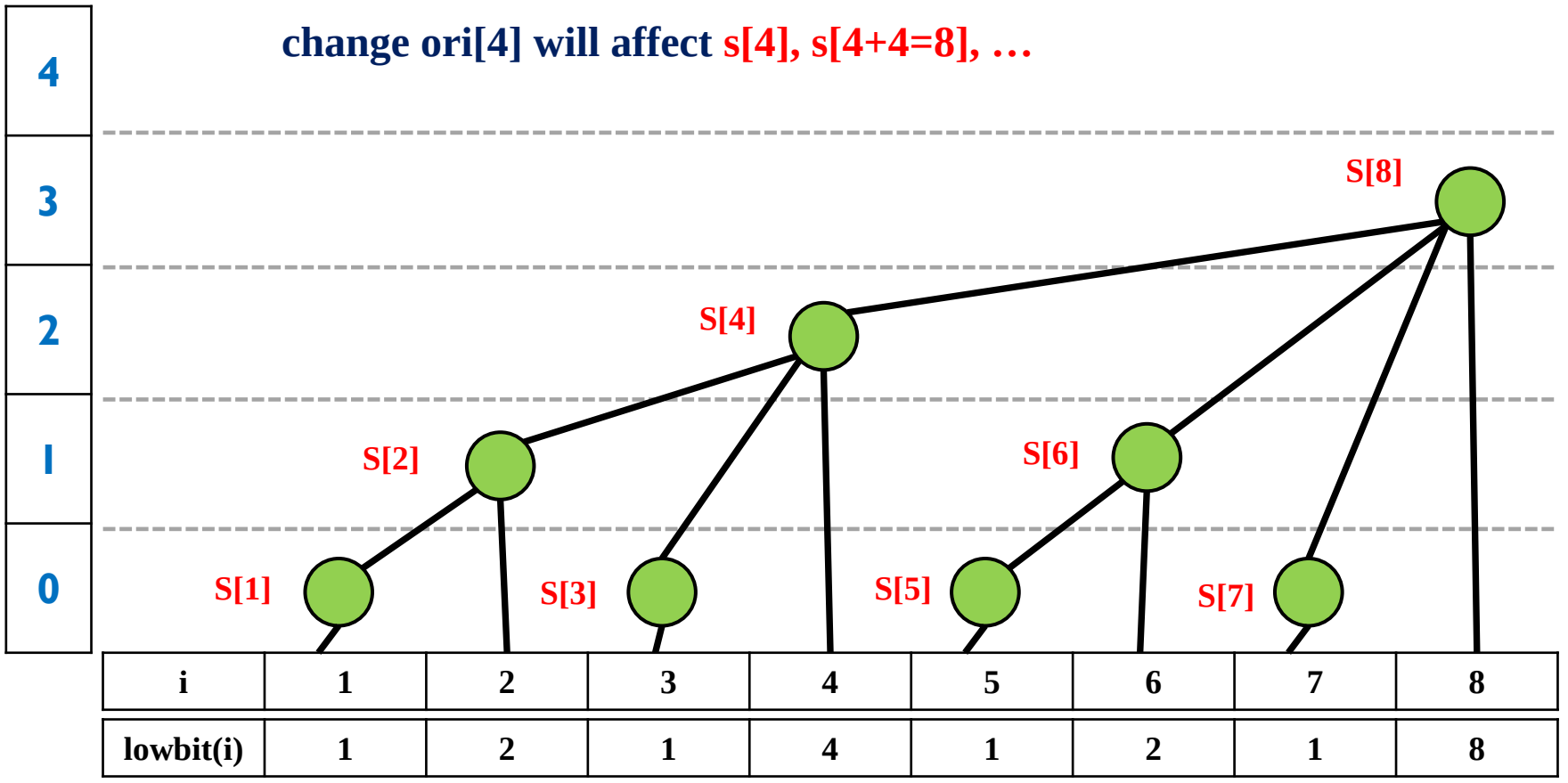
Binary Indexed Tree



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8

made by electron & free999 & a711186

Binary Indexed Tree



i	1	2	3	4	5	6	7	8
lowbit(i)	1	2	1	4	1	2	1	8

Binary Indexed Tree

```
13    int change (int end, int delta) {
14        for(int i = end; i <= maxsize; i += lowbit(i))
15            s[i] += delta;
16    }
```

Notice:

end = 0，0 + lowbit(0) = 0　無限迴圈！！！

Binary Indexed Tree

- How to find the summation between interval [i…j] ?
  - call the subroutine **“getsum[j] – getsum[i-1]”**
- Expand the 1 dimension into 2 dimension by yourself
- Replace such routines with a segment tree by yourself

Binary Indexed Tree

- 時間複雜度
  - 建立時間為 O(NlogN)
  - 計算任意區間總和、修改時間是 O(log N)
  - 共 Q 個操作，總複雜度為 O(N+Q log N)
  - 建立空間為 O(N)

Example

POJ-2352 (link)

Reference

- 演算法筆記-Sequence
  http://www.csie.ntnu.edu.tw/~u91029/Sequence.html#1
- 2015 IOI camp
  http://ioicamp.csie.org/content