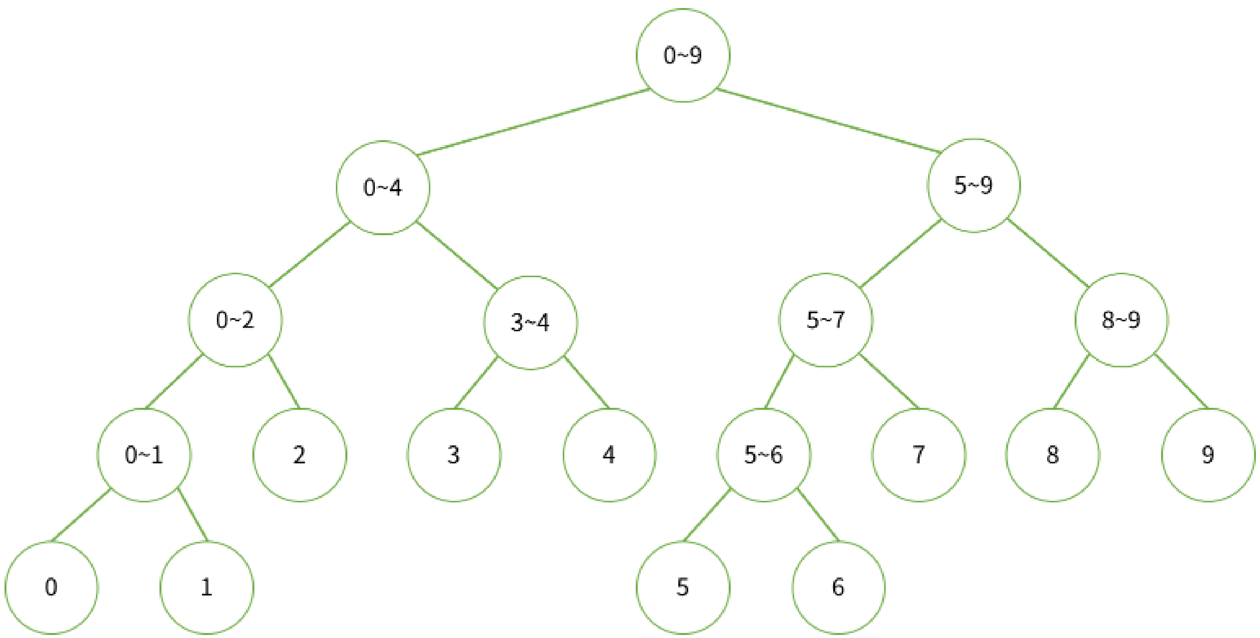
Segment Tree for Array

- 배열 A가 있고, 다음과 같은 두 연산을 각각 최대 M번 수행해야 하는 문제를 생각해 보자.
  1) 구간 l, r (l ≤ r)이 주어졌을 때,
     A[l] + A[l+1] + … + A[r-1] + A[r] 구하기
  2) i 번째 수를 v로 바꾸기. A[i] = v
- 배열에서만 풀면
  1번 연산) O(N) → O(NM)
  2번 연산) O(1) → O(M)
  – 총 시간 복잡도: O(NM)
- 세그먼트 트리를 이용하면
  – 1번, 2번 연산을 각각 O(log N)
    → 총 시간복잡도: O(M log N)

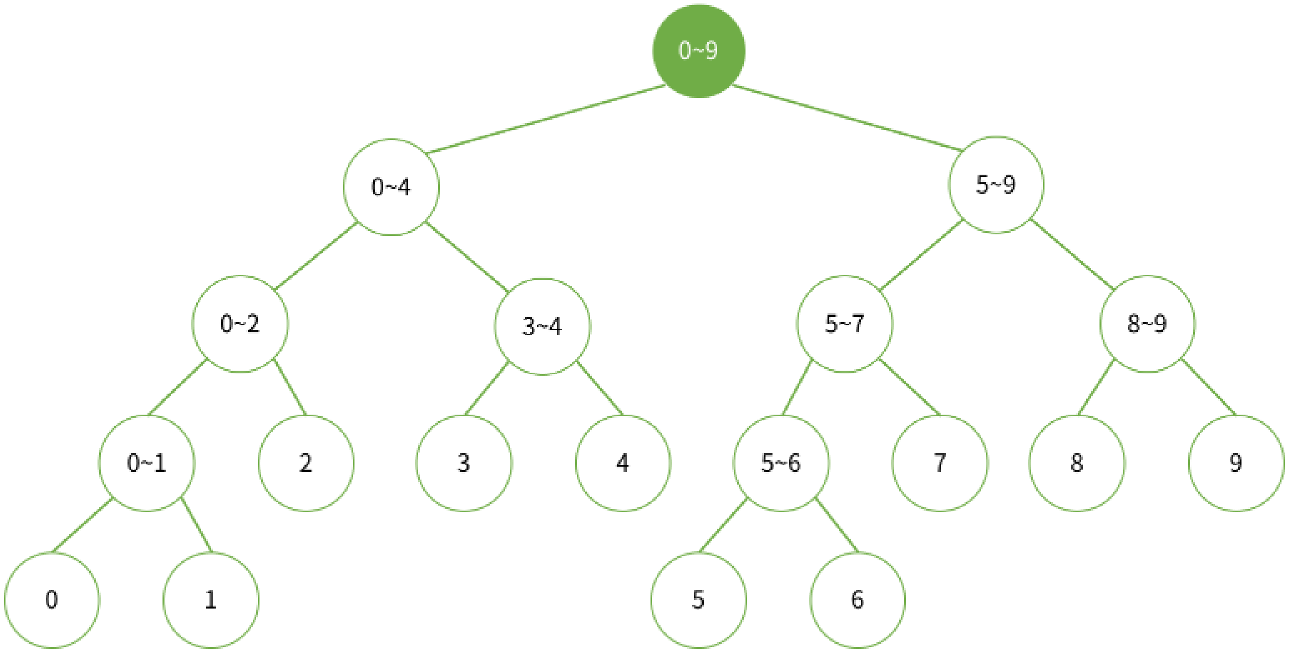
Segment Tree for Array

- 세그먼트 트리에서 단말노드와 리프노드의 의미
  - 리프 노드: 배열의 그 수 자체
  - 내부 노드: 왼쪽 자식과 오른쪽 자식의 합을 저장함
- N=10인 세그먼트 트리의 예



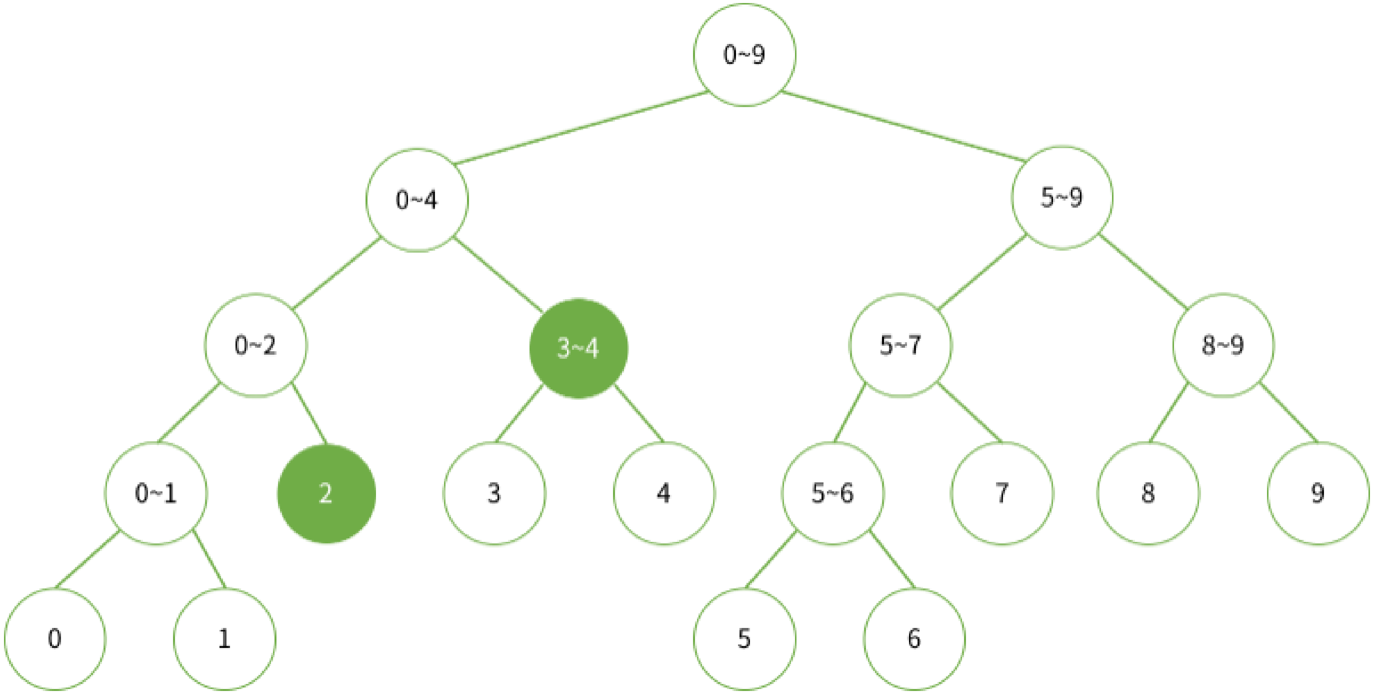
Segment Tree에서 배열의 합 찾기

- 구간 left, right가 주어졌을 때 합 구하기
- 0~9까지 합을 구하는 경우는 루트 노드 하나만으로 합을 알 수 있다.



Segment Tree에서 배열의 합 찾기

- 2~4까지 합을 구하는 예

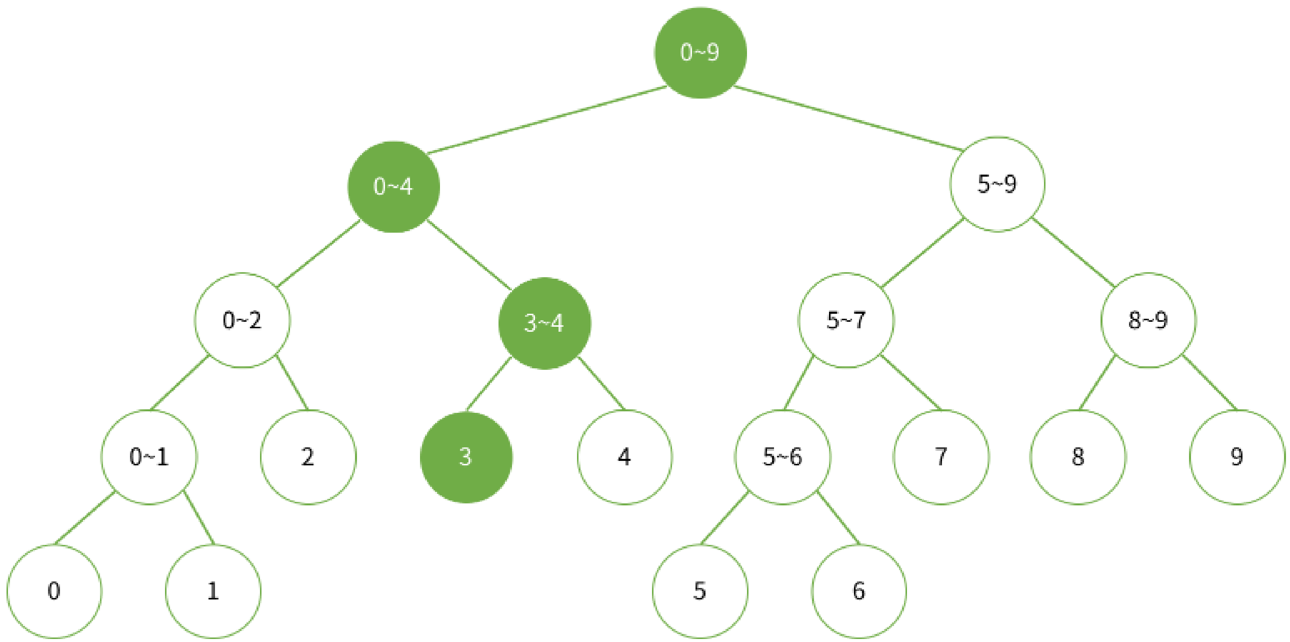


Segment Tree에서 배열의 합 찾기

- node가 담당하고 있는 구간이 [start,end]
- 합을 구해야 하는 구간이 [left,right]
- 다음과 같이 4가지 경우로 나누어질 수 있다
  1. [left,right]와 [start,end]가 겹치지 않는 경우
     - 더 이상 탐색할 필요 없음. 0을 리턴
  2. [left,right]가 [start,end]를 완전히 포함하는 경우
     - 더 이상 탐색할 필요 없음. 그 노드의 값을 리턴
  3. [start,end]가 [left,right]를 완전히 포함하는 경우
     - 자식 트리에서 탐색을 계속해 함
  4. [left,right]와 [start,end]가 겹쳐져 있는 경우 (1,2,3 제외한 나머지 경우)
     - 자식 트리에서 탐색을 계속해 함

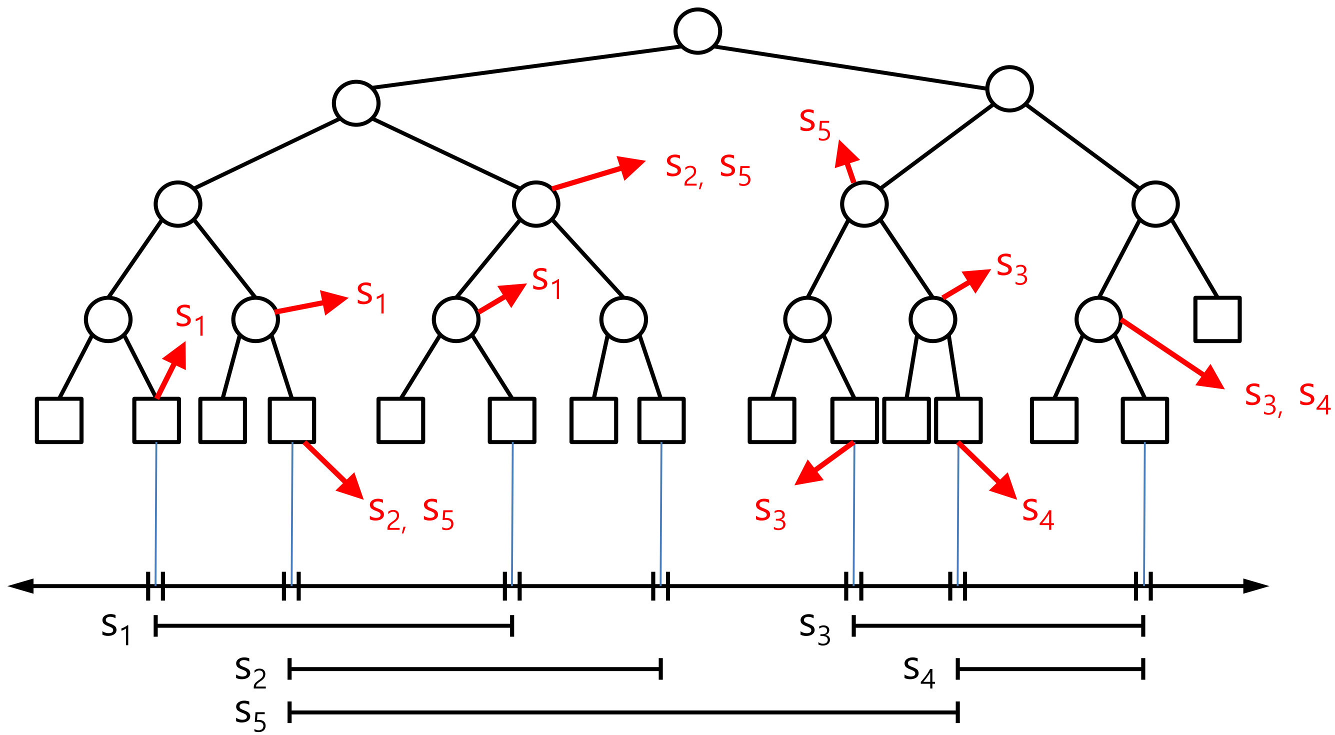
Segment Tree에서 수 변경하기

- 배열의 한 값을 변경하면, 그 숫자가 포함된 구간을 담당하는 노드를 모두 변경해줘야 한다.
- 3번째 수를 변경할 때, 변경해야 하는 구간을 나타내는 예



Segment Tree for Intervals

- 5개의 interval에 대한 segment tree의 예



- O(n log n) 공간, O(n log n) 시간에 구성 가능

Segment Tree for Intervals

- Segment tree는 Interval Tree와 같이 다음과 같은 연산을 지원하는 자료구조
  - IntervalInsert(T, x) : Interval 트리 T에 폐구간을 저장하는 노드 x를 추가한다.
  - IntervalDelete(T, x) : Interval 트리 T에서 노드 x를 제거한다.
  - IntervalSearch(T, i) : 폐구간 i와 서로 겹치는 모든 구간을 계산한다.
  - IntervalPointQuery(T, q) : 점 q를 포함하는 모든 구간을 계산한다.

Segment Tree for Intervals

- IntervalPointQuery를 효율적으로 처리하는 Segment Tree의 구성에 대하여 설명.
- IntervalPointQuery는 n개의 폐구간 I = {[a_1, b_1], [a_2, b_2], ..., [a_n, b_n]} 주어 졌을 때, query point q를 포함하는 모든 구간을 계산함
- Segment tree는 일반적으로 반-동적(semi-dynamic) 연산이 필요한 응용분야에 많이 활용된다.
- 반동적 연산이라 함은 segment tree에 insert되고, delete되는 interval 이 I (이미 정의된)의 폐구간의 끝점으로만 이루어진 interval 이라는 점이다.

Segment Tree for Intervals

- 점 $p_1, p_2, \ldots, p_m$ 을 폐구간 I 에 속하는 구간들의 끝점을 오름차순으로 나열한 점이라고 하자.
- 그러면, 1차원 직선을 아래와 같이 점 $p_1, p_2, \ldots, p_m$ 를 이용하여 여러 개의 구간으로 나눌 수 있다.

$(-\infty,p_1),[p_1,p_1],(p_1,p_2),[p_2,p_2],(p_2,p_2), \ldots,(p_{m-1}, p_m),[p_m,p_m], (p_m,+\infty)$

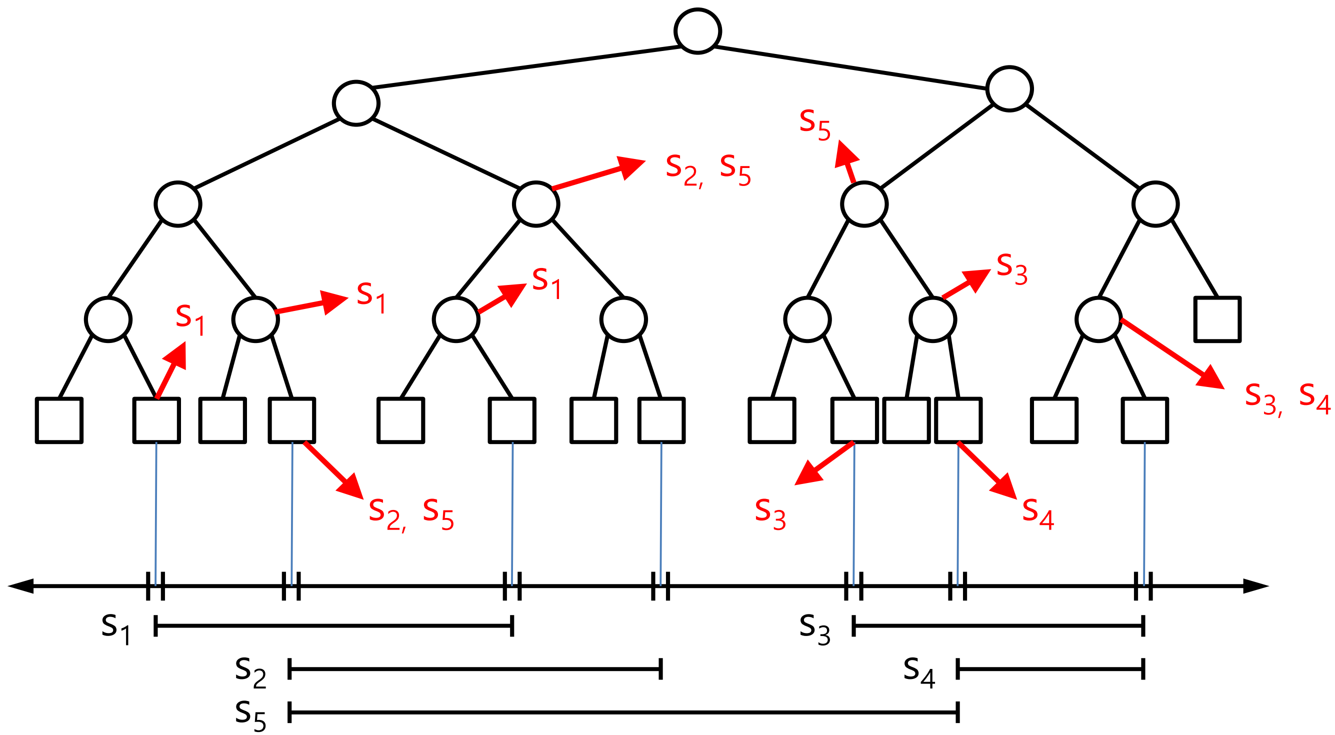
- 위 구간을 기본구간(elementary interval)라고 부르자.

Segment Tree (ST) for Intervals

- 기본구간을 이용하여 다음과 같이 트리를 만듬
  - ST는 균형트리를 근간으로 만들며, 하나의 기본구간마다 대응되는 tree의 단말노드를 만든다. 이 때, 단말노드의 inoder 순서는 기본구간 순서와 일치하게 만든다.
  - ST의 각 내부노드는 그 노드를 루트노드로 하는 모든 단말트리에 대응하는 기본구간의 합에 해당하는 구간을 가진다. 즉, 어떤 내부노드에 대응하는 구간은 그 노드의 두 자식노드에 대응하는 구간의 합이다. 따라서, 루트노드에 대응하는 구간은 1차원 전체구간이 된다.
  - ST의 각 단말노드, 내부노드 v는 다음 두 정보를 저장한다.
    - Int(v) : 노드 v에 대응되는 interval
    - I(v): I 에 속하는 interval 중에서 Int(v) $\subseteq$ [a,b] 이며 Int(parent(v)) $\not\subseteq$ [a,b] 를 만족하는 모든 interval [a, b]들을 리스트로 저장

Segment Tree for Intervals

- 5개의 interval에 대한 segment tree의 예



- O(n log n) 공간, O(n log n) 시간에 구성 가능

Segment Tree for Intervals

- 주어진 점 qx를 포함하는 모든 interval을 계산하는 함수

```
QuerySegmentTree(v, qx)
        I(v)에 속하는 interval 을 report 한다.
        if (v is not a leaf)
            if (qx ∈ Int(v->left))
                QuerySegmentTree(v->left, qx);
            else
                QuerySegmentTree(v->right, qx);
```

- 수행시간: O(log n + k)

Binary Indexed Tree (BIT)

- Fenwick Tree라고도 불림
- BIT (Binary Indexed Tree)는 다음과 같은 1차원, 2차원 혹은 다차원의 Query 를 처리하는데 사용될 수 있는 매우 효율적인 (다른 자료구조로도 처리할 수 있으나, 코드가 상대적으로 매우 간략하여 사용하기 쉬움) 자료구조
  - 1차원 Query (배열 a[MAX]에서)
    - Update a[i]
    - Query sum of a[i] to a[k] ($0 \leq i \leq k < MAX$)
  - 2차원 Query (배열 a[MAX][MAX])
    - Update a[i][j]
    - Query sum of elements in rectangle bound by rows r1 and r2 and by columns c1 and c2

2진수에서 마지막 bit-1 계산

- 어떤 자연수 N 를 이진수로 표현하였을 때, 가장 마지막 bit 1의 위치를 $\ell(N)$ 이라고 하자.
- N으로부터 마지막 bit-1을 추출하는 방법

  N & (-N) 여기서 &는 bitwise-AND

  N & (N^(N-1)) 여기서 ^는 Exclusive OR
- 정수 N에서 마지막 bit-1 을 제거하는 방법

  N – (N & -N)

  N & (N-1)

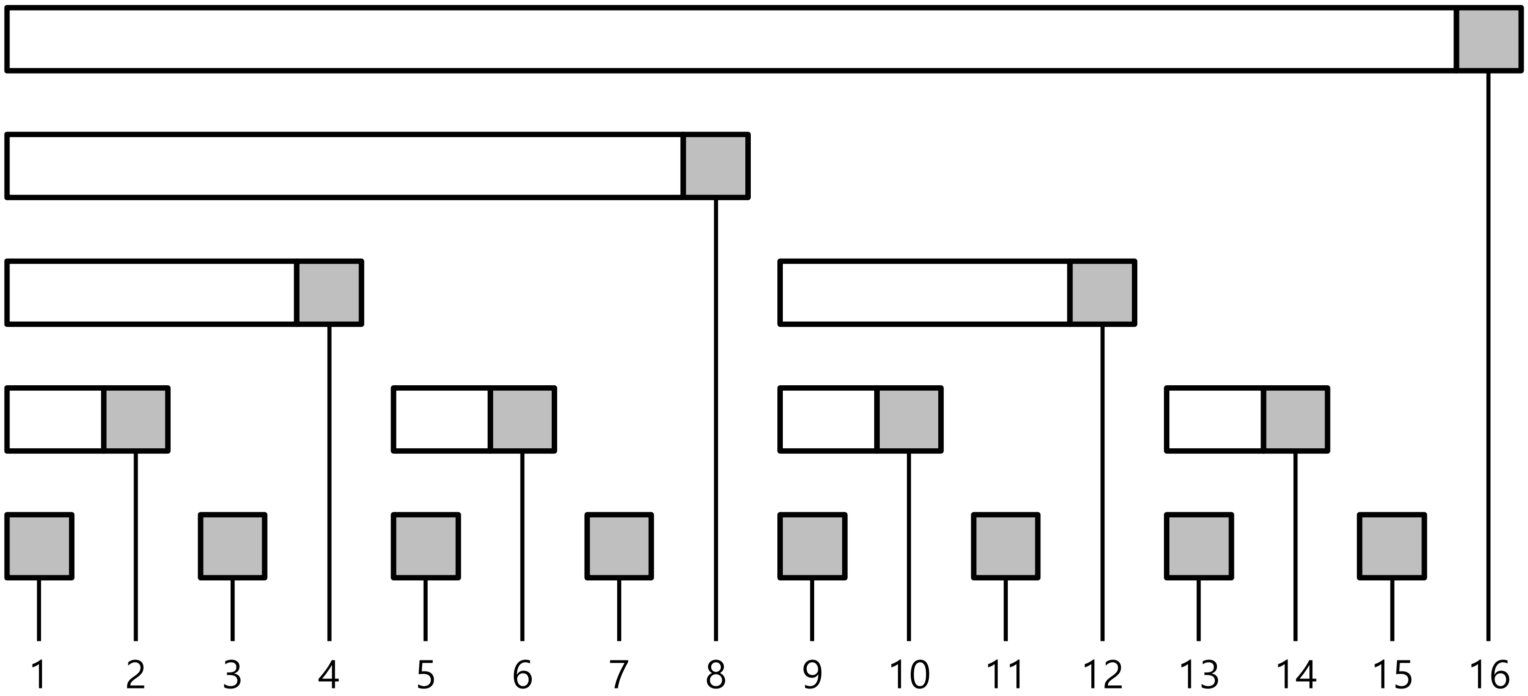
예 N: 00110100

-N: 11001100 N&(-N): 00000100

N – (N & -N) : 00110000

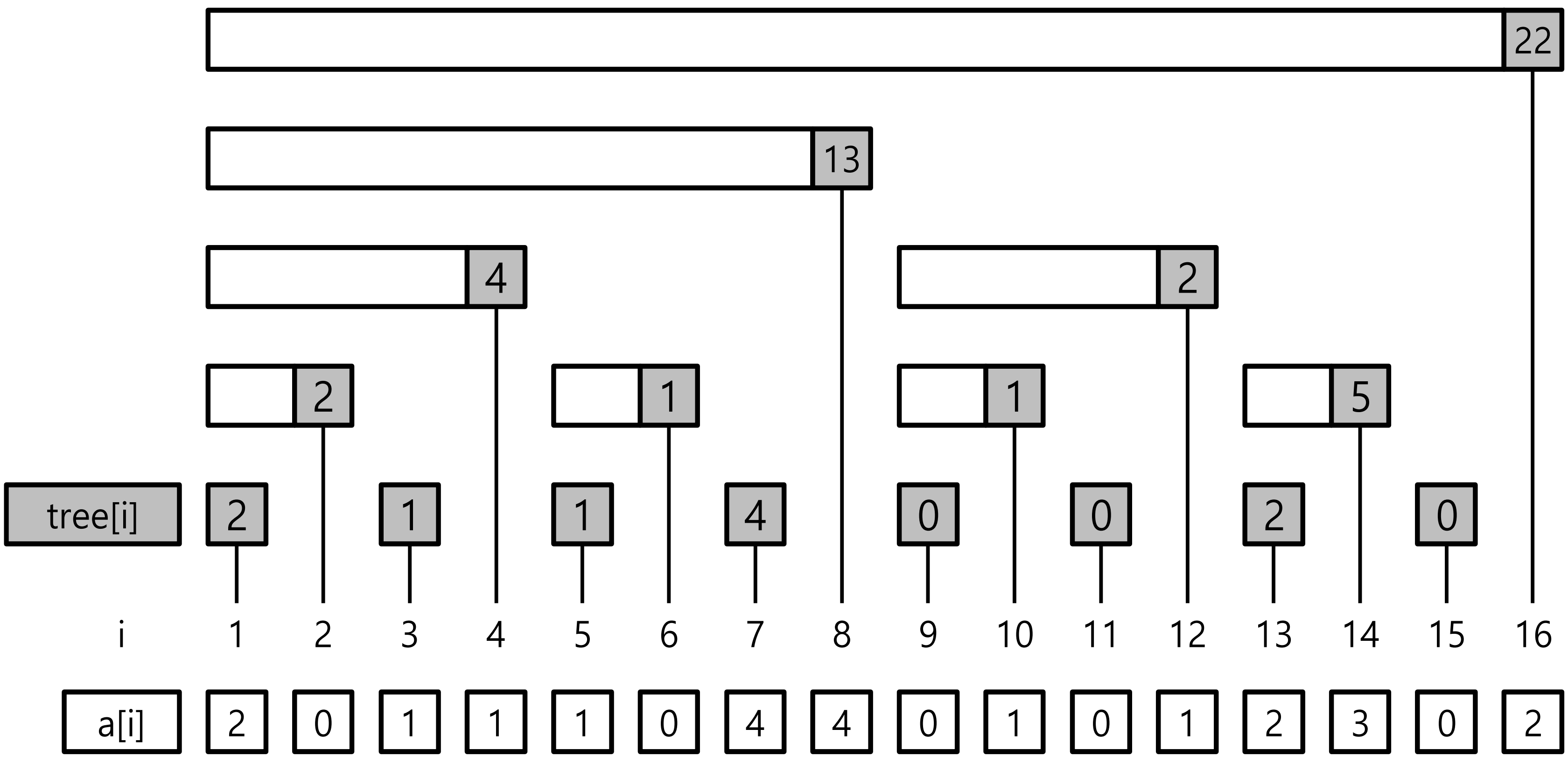
Binary Indexed Tree

- BIT의 예



BIT 초기화

- 자연수 배열 a[MAX] (a[0] 는 0으로 가정) 에 대하여 다음과 같이 BIT에서 각 정수가 대표하는 그룹에 속하는 정수를 index로 하는 배열값의 부분 누적합 (Cumulative Sum)이 저장된 배열 tree[] 을 정의



BIT 초기화

- BIT를 사용하여 부분누적합 계산 예

  tree[N] = tree[N] + tree[N-1] + tree[마지막 1제거] + ...

  예 : N = a100000₂ 이라 하자

  tree[N] = tree[a100000] + tree[a011111] + tree[a011110]

  + tree[a011100] + tree[a011000] + tree[a010000]

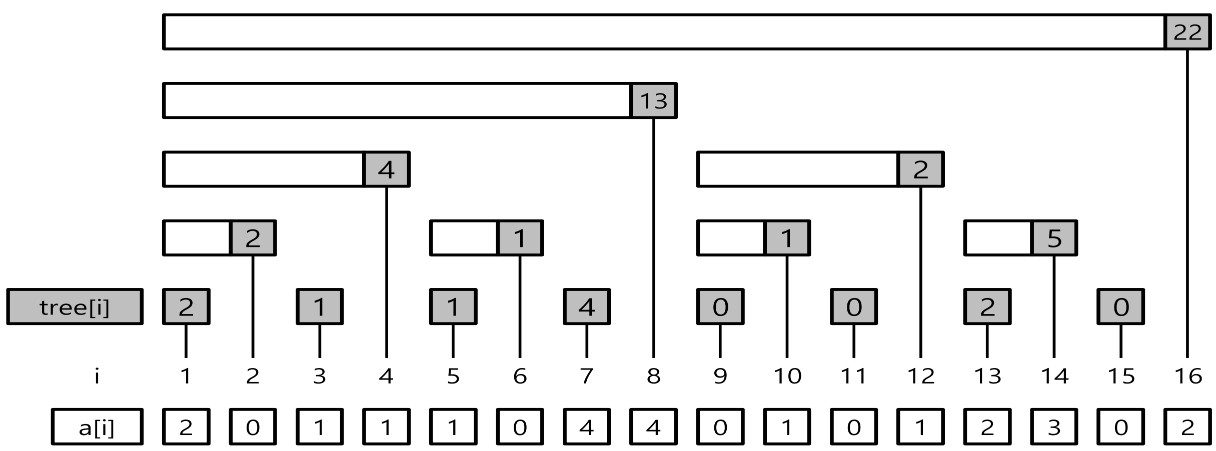
BIT 초기화

- BIT를 사용하여 부분누적합 계산

  tree[12] = tree[1100]+tree[1011]+tree[1010] =
  tree[12]+tree[11]+tree[10]

  tree[16] = tree[10000] + tree[01111] + tree[01110] +
  tree[01100] + tree[01000]

  = tree[16] + tree[15] + tree[14] + tree[12] + tree[8]



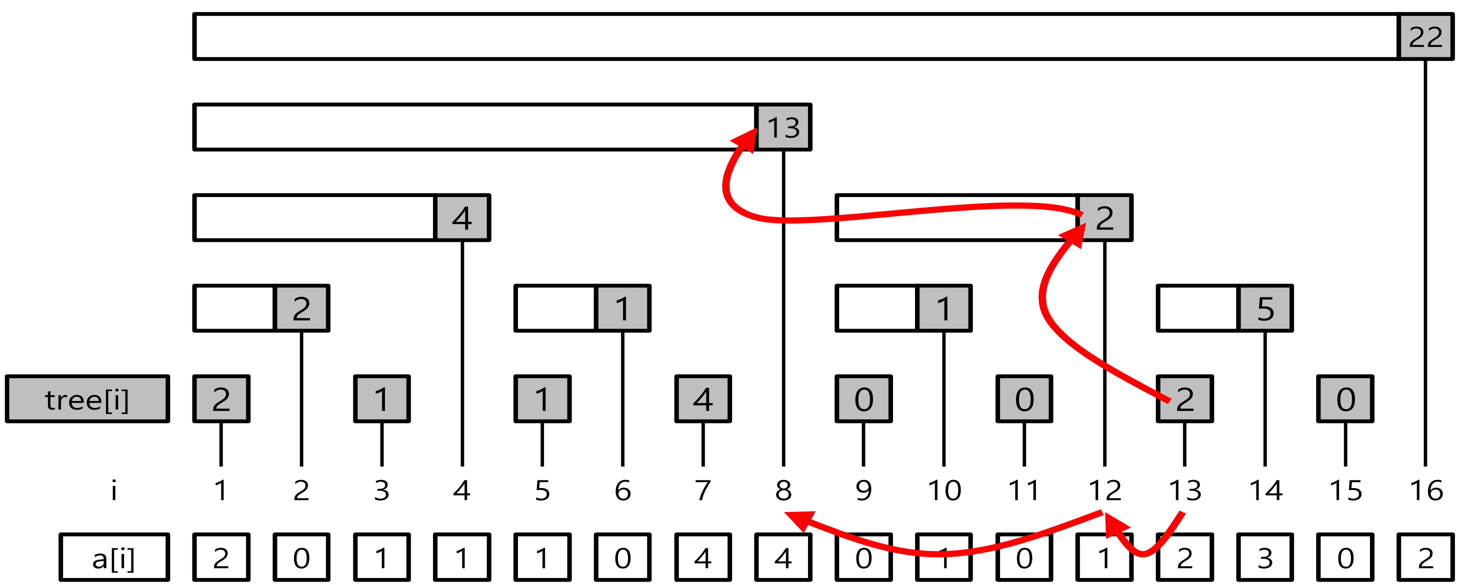
BIT 초기화

```
#define GET_LAST_ONE(N) ((N)^(-(N)))
void initCumulativeSum(int a[], int tree[], int size)
{   int i, sum, index;
    int lastOne, removeLastOne;
    for(i=1; i<=size; i++)
    {
        lastOne = GET_LAST_ONE(i);
        removeLastOne = i - lastOne;
        sum = a[i];
        index = lastOne-1;

        while (index > 0)
        {
            sum += tree[removeLastOne + index];
            index -= GET_LAST_ONE(index);

        }
        tree[i] = sum;
    }
}
```

BIT - 누적 합

- 예를 들어, index = k = 13인 경우

Iteration	index	Position of the last bit 1	index & - index	sum
1	13 = 1101	0	0001 (2^0)	2
2	12 = 1100	2	0100 (2^2)	4
3	8 = 1000	3	1000 (2^3)	17
4	0	-	-	-



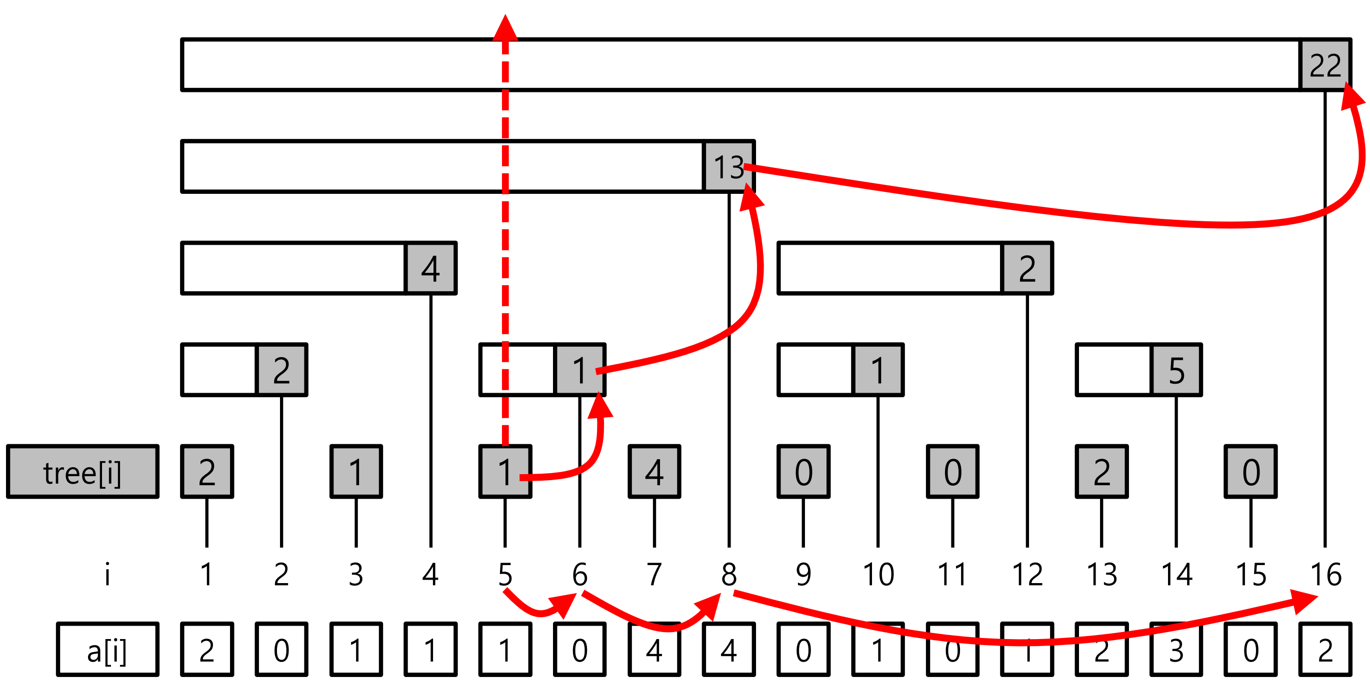
BIT - 누적 합

```
int tree[];

int getCumulativeSum(int k)
{
    int sum, index;

    sum = 0;
    index = k;

    while (index > 0)
    {
        sum += tree[index];
        index -= GET_LAST_ONE(index);
    }

    return sum;
}
```

BIT - Update

- 예를 들어, index = k = 5인 경우

Iteration	index	Position of the last bit 1	Idx + idx & -idx	sum
1	5 = 0101	0	0101 + 0001	
2	6 = 0110	1	0110 + 0010	
3	8 = 1000	3	1000 + 1000	
4	16 = 10000	4	10000	



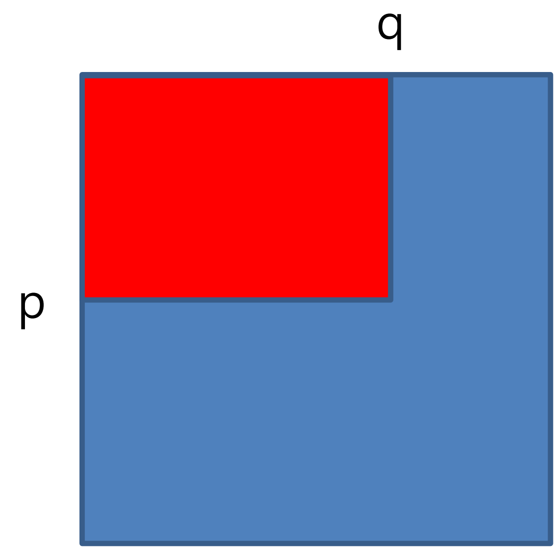
BIT - Update

```
int tree[];
int size;     // size of tree

void update(int k, int value)
{
    int index;

    index = k;

    while (index <= size)
    {
        tree[index] += value;
        index += GET_LAST_ONE(index);//add last significant bit
    }
}
```

2차원 BIT

- 2차원 배열 a[MAX][MAX] 에서 다음과 같은 query는 2차원 BIT를 통하여 연산
  - Update a[i][j]
  - 2차원 배열에서 원소의 위치가 (0, 0)와 (p, q) 에 의해서 정의되는 사각형에 위치한 모든 원소의 합 C(p, q)을 구한다.

$$C = \sum_{i=1}^{p} \sum_{j=1}^{q} a[i, j]$$

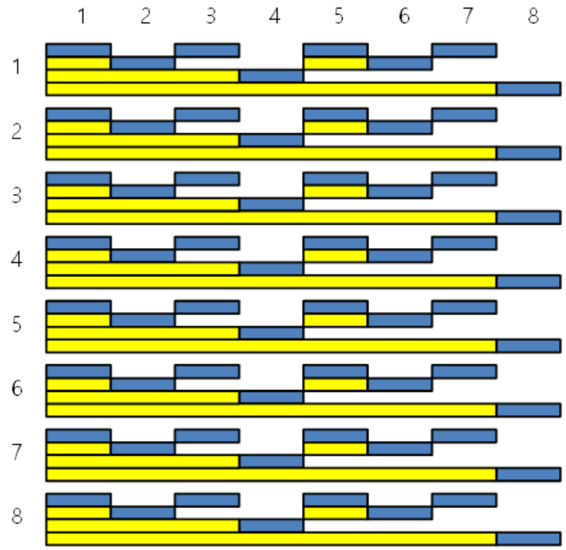


2차원 BIT

- 2차원 배열 a[MAX][MAX]에 대해 2차원 BIT 초기화

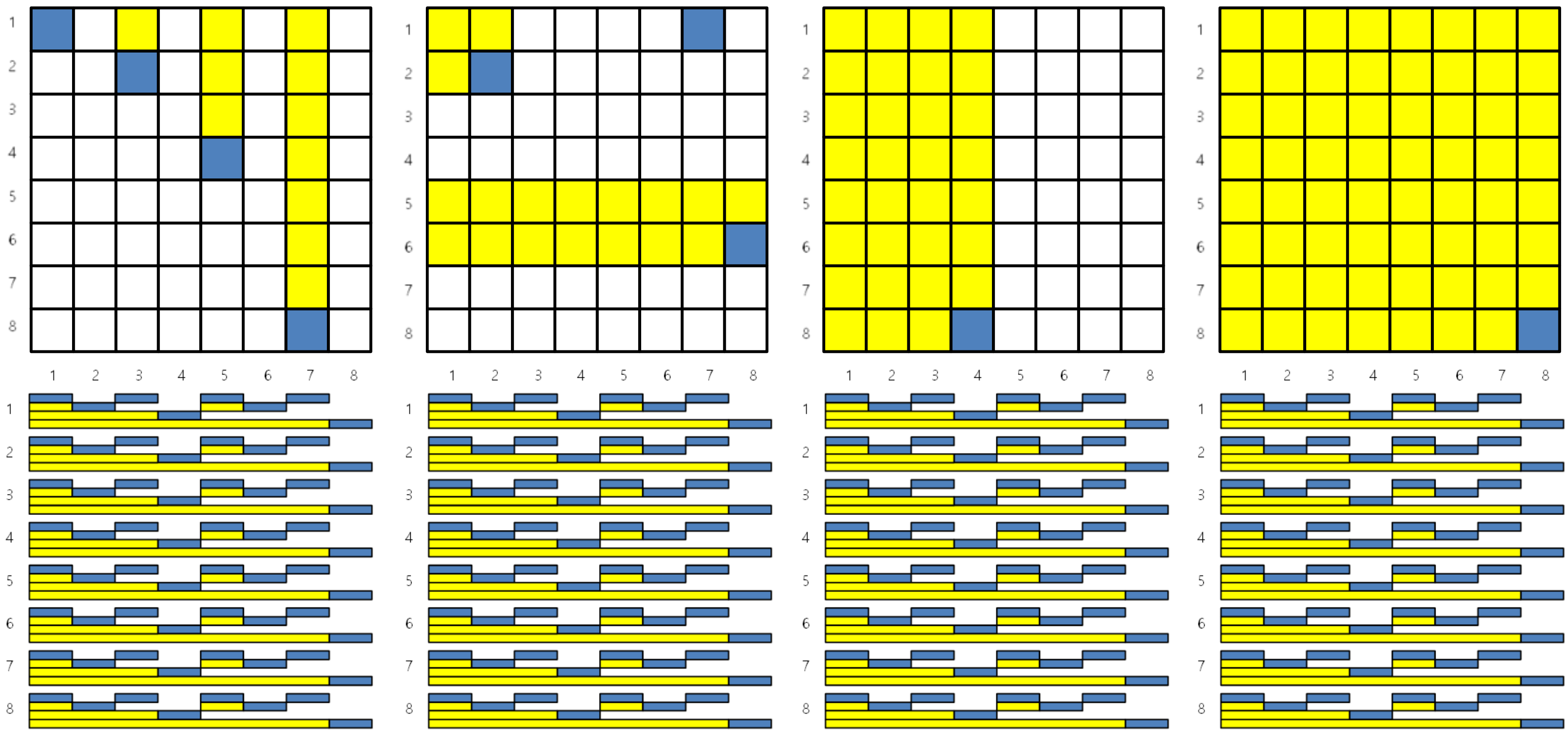
  (Step 1)배열 a[][] 모든 행에 대하여 1차원 BIT tree[][]를 만든다(주어진 1차원 배열에 대하여 그 배열 자체에 1차원 BIT를 만들 수 있음에 유의한다).

  (Step 2) 모든 행에 대하여 1차원 BIT가 만들어진 배열 tree[][]의 모든 열에 대하여 1차원 BIT를 만든다.



2차원 BIT

- 8×8인 배열을 2차원 BIT 를 저장하는 배열 a[][] (혹은 tree[][]) 의 각 원소가 배열의 합을 만드는 예시



0	1	2	3	4	5	6	7	8
1	1	0	3	1	-1	-3	1	2
2	-1	1	5	0	3	2	1	-2
3	2	0	-2	1	3	4	-2	0
4	3	4	3	1	3	0	-3	-2
5	0	3	2	-3	2	4	0	2
6	4	0	-2	4	2	1	2	4
7	0	1	2	3	1	-2	0	2
8	-2	-1	3	1	-3	4	5	2

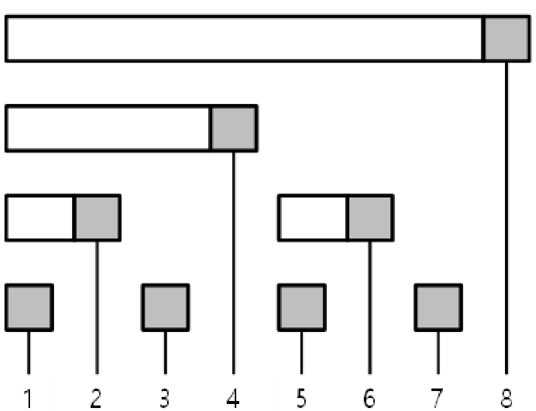
원 data

0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	-1	0	5	5	3	5	1	9
3	2	2	-2	1	3	7	-2	6
4	3	7	3	11	3	3	-3	9
5	0	3	2	2	2	6	0	10
6	4	4	-2	6	2	3	2	15
7	0	1	2	6	1	-1	0	7
8	-2	-3	3	1	-3	1	5	9

각 행에 대해 1차원 BIT 구한 결과

Sum a[1][1]~a[8][8] ?

Sum a[1][1]~a[4][4] ?



최종 2차원 BIT

0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	0	1	8	10	2	1	2	13
3	2	2	-2	1	3	7	-2	6
4	5	10	9	22	8	11	-3	28
5	0	3	2	2	2	6	0	10
6	4	7	0	8	4	9	2	25
7	0	1	2	6	1	-1	0	7
8	7	15	14	37	10	20	4	69

0	1	2	3	4	5	6	7	8
1	1	0	3	1	-1	-3	1	2
2	-1	1	5	0	3	2	1	-2
3	2	0	-2	1	3	4	-2	0
4	3	4	3	1	3	0	-3	-2
5	0	3	2	-3	2	4	0	2
6	4	0	-2	4	2	1	2	4
7	0	1	2	3	1	-2	0	2
8	-2	-1	3	1	-3	4	5	2

원 data

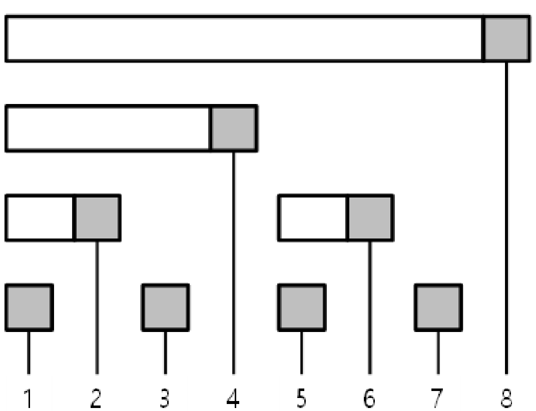
0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	-1	0	5	5	3	5	1	9
3	2	2	-2	1	3	7	-2	6
4	3	7	3	11	3	3	-3	9
5	0	3	2	2	2	6	0	10
6	4	4	-2	6	2	3	2	15
7	0	1	2	6	1	-1	0	7
8	-2	-3	3	1	-3	1	5	9

각 행에 대해 1차원 BIT 구한 결과

최종 2차원 BIT

0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	0	1	8	10	2	1	2	13
3	2	2	-2	1	3	7	-2	6
4	5	10	9	22	8	11	-3	28
5	0	3	2	2	2	6	0	10
6	4	7	0	8	4	9	2	25
7	0	1	2	6	1	-1	0	7
8	7	15	14	37	10	20	4	69

Sum a[1][1]~a[6][5] ?



0	1	2	3	4	5	6	7	8
1	1	0	3	1	-1	-3	1	2
2	-1	1	5	0	3	2	1	-2
3	2	0	-2	1	3	4	-2	0
4	3	4	3	1	3	0	-3	-2
5	0	3	2	-3	2	4	0	2
6	4	0	-2	4	2	1	2	4
7	0	1	2	3	1	-2	0	2
8	-2	-1	3	1	-3	4	5	2

원 data

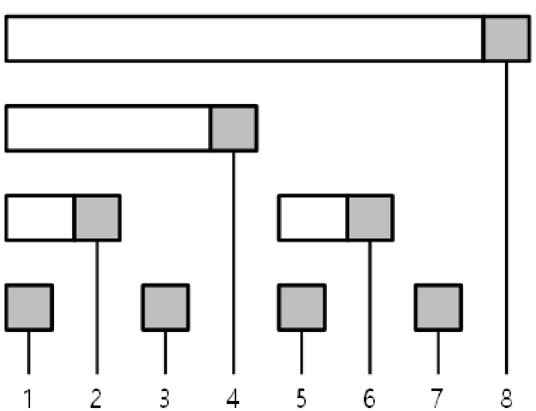
0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	-1	0	5	5	3	5	1	9
3	2	2	-2	1	3	7	-2	6
4	3	7	3	11	3	3	-3	9
5	0	3	2	2	2	6	0	10
6	4	4	-2	6	2	3	2	15
7	0	1	2	6	1	-1	0	7
8	-2	-3	3	1	-3	1	5	9

각 행에 대해 1차원 BIT 구한 결과

최종 2차원 BIT

0	1	2	3	4	5	6	7	8
1	1	1	3	5	-1	-4	1	4
2	0	1	8	10	2	1	2	13
3	2	2	-2	1	3	7	-2	6
4	5	10	9	22	8	11	-3	28
5	0	3	2	2	2	6	0	10
6	4	7	0	8	4	9	2	25
7	0	1	2	6	1	-1	0	7
8	7	15	14	37	10	20	4	69

Sum a[1][1]~a[3][7] ?



2차원 BIT

- 누적합을 이용하면, 임의의 index p1, p2, q1, q2 의하여 정의된 직사각형 내에 포함된 모든 원소의 값은 다음과 같이 구할 수 있다

$$\sum_{i=p1}^{p2}\sum_{j=q1}^{q2} a[i][j] = C(p2, q2) - C(p2, q1-1) - C(p1-1, q2) + C(p1-1, q1-1)$$