Sogang ICPC Team

2021-1 중급 스터디 3회차

Segment Tree

이윤제

yjyj102742@gmail.com

목차

- Segment Tree
  - Range Query
  - What is Segment Tree?
  - 예제
- Lazy Propagation
  - What is Lazy Propagation?
  - 예제
- 마무리
  - 연습문제

Segment Tree

- Range Query
- What is Segment Tree?
- 예제

Range Query

- N개의 수로 이루어진 배열
- 연속한 구간의 합
- Sum of [2, 6] = ?

arr	1	2	-3	4	-5	3	0	1

Range Query

- N개의 수로 이루어진 배열
- 연속한 구간의 합
- Sum of [2, 6] = ?

- 나이브한 접근: arr[2] + arr[3] + arr[4] + arr[5] + arr[6] = 1
- O(n)

arr	1	2	-3	4	-5	3	0	1

Range Query

- N개의 수로 이루어진 배열
- 연속한 구간의 합
- Sum of [2, 6] = ?

- Prefix Sum: prefix[6] – prefix[1] = 1
- O(1)

arr	1	2	-3	4	-5	3	0	1
prefix	1	3	0	4	-1	2	2	3

- N개의 수로 이루어진 배열
- 배열 값을 업데이트
- arr[2] = 4

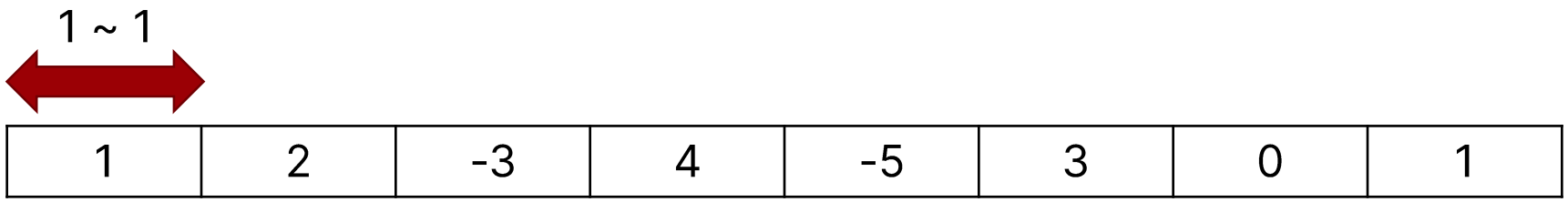
- Prefix[2] = 5, Prefix[3] = 2, ...
- O(n) for update

arr	1	4	-3	4	-5	3	0	1
prefix	1	5	2	6	1	4	4	5

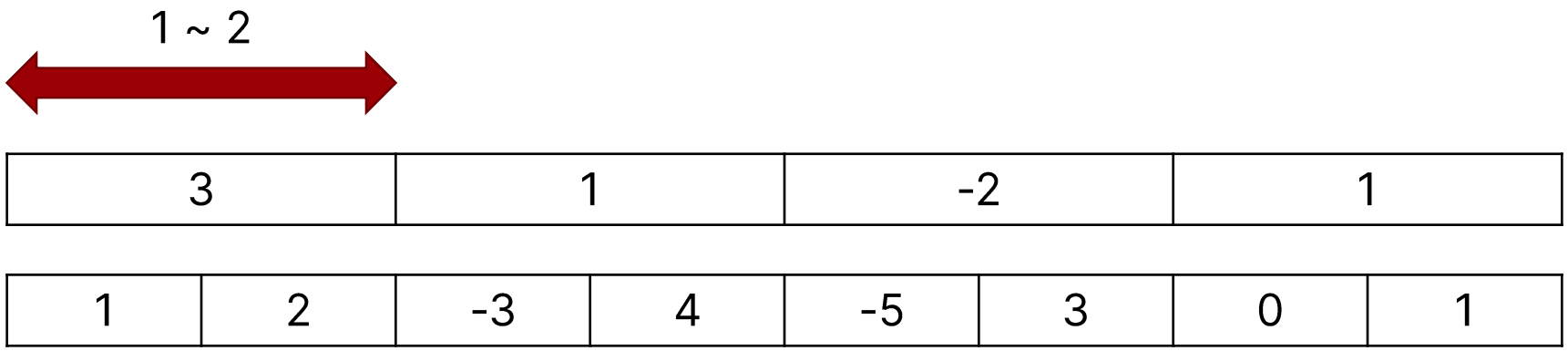
- Range Query
  - 주어진 배열에서 특정 연속구간에 대한 질의를 하는 문제
  - Update를 요구할 때도 있음
- Naive한 방식
  - Query: O(n)
  - Update: O(1)
- Prefix Sum을 사용
  - Query: O(1)
  - Update: O(n)

- Range Query
  - 주어진 배열에서 특정 연속구간에 대한 질의를 하는 문제
  - Update를 요구할 때도 있음
- Segment Tree
  - Query: O(logn)
  - Update: O(logn)

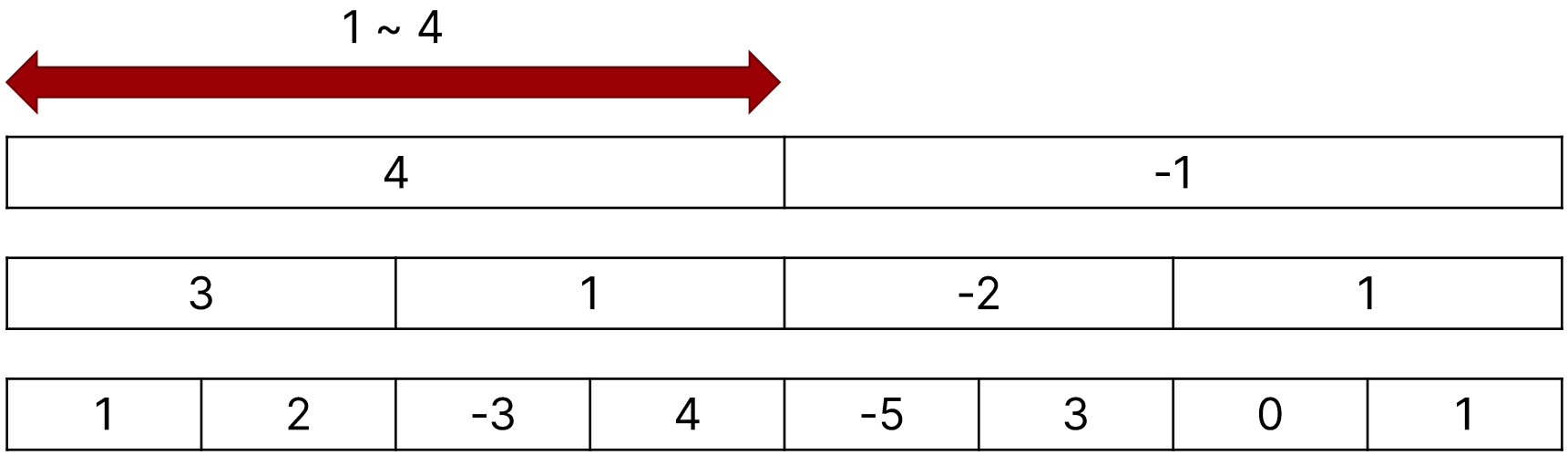
- 아이디어: 분할-정복
  - 일정 단위로 원소들을 묶어두자
  - 구간을 분할해서 각각의 결과값을 합치자



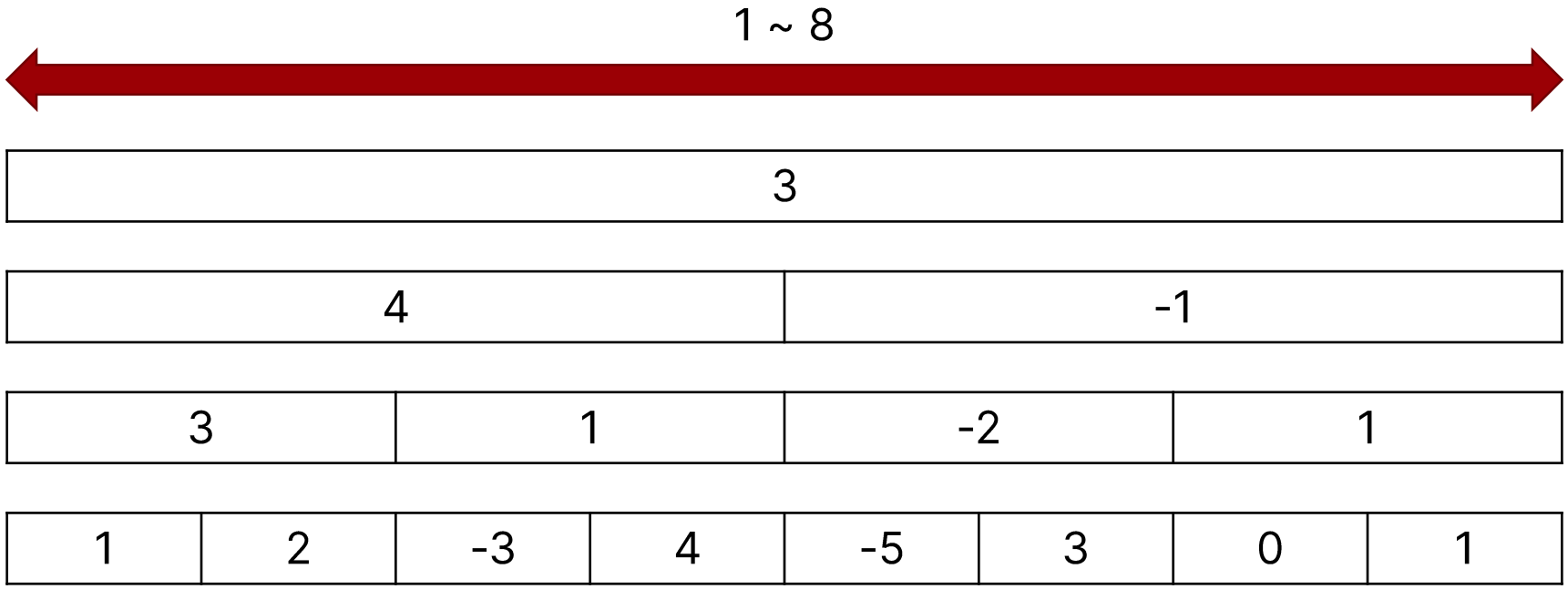
- 아이디어: 분할-정복
  - 일정 단위로 원소들을 묶어두자
  - 구간을 분할해서 각각의 결과값을 합치자



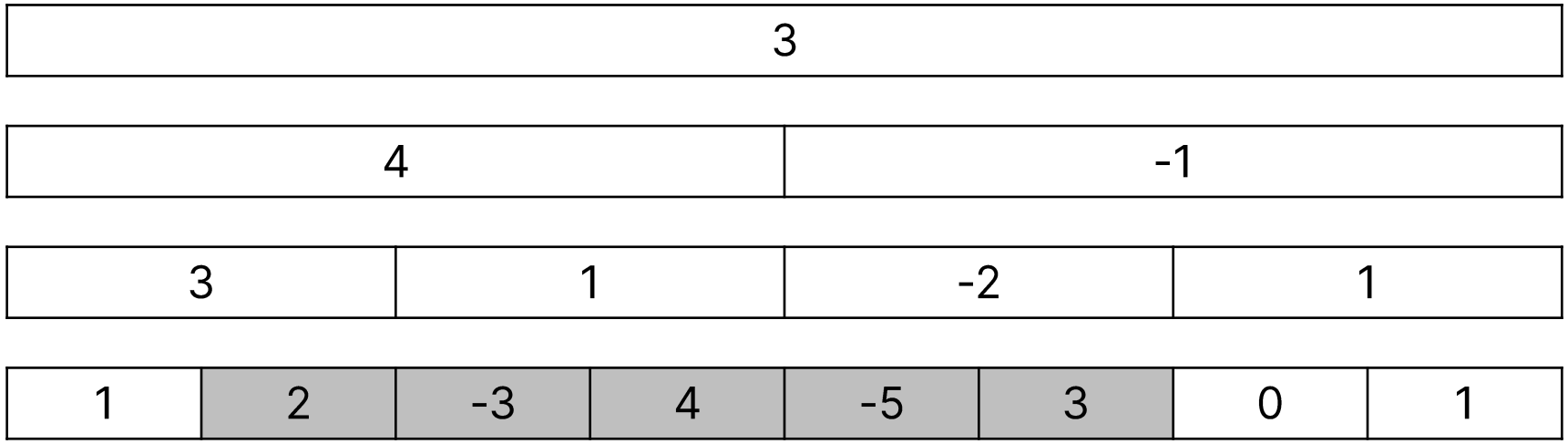
- 아이디어: 분할-정복
  - 일정 단위로 원소들을 묶어두자
  - 구간을 분할해서 각각의 결과값을 합치자



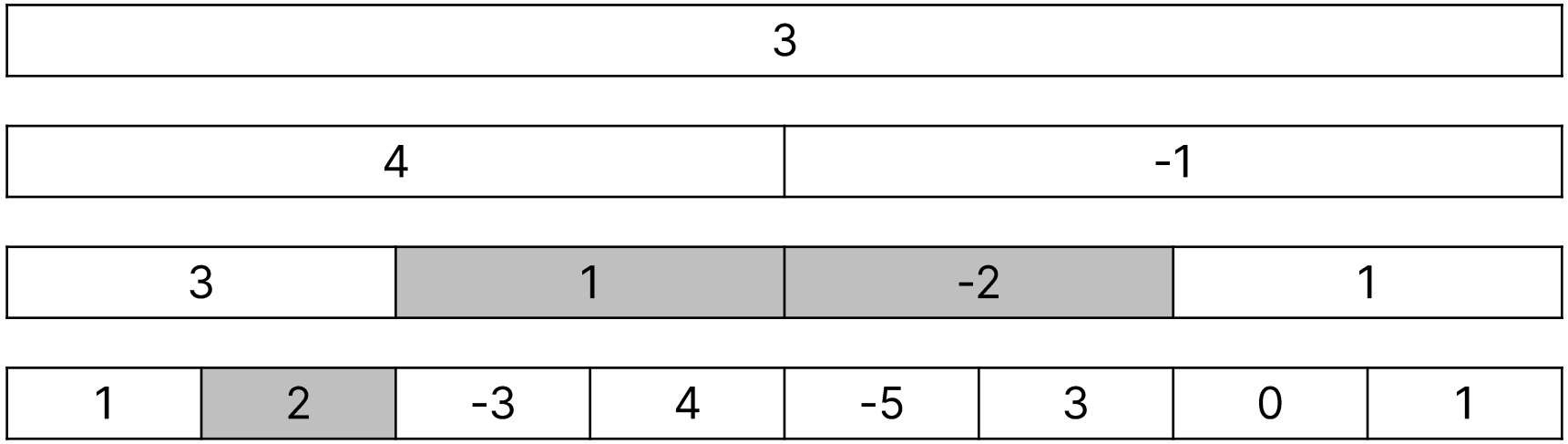
- 아이디어: 분할-정복
  - 일정 단위로 원소들을 묶어두자
  - 구간을 분할해서 각각의 결과값을 합치자



- Sum[2, 6]을 구해보자
  - Naive한 방식: 5개의 값을 더한다



- Sum[2, 6]을 구해보자
  - 묶은 구간을 활용: 3개의 값을 더한다



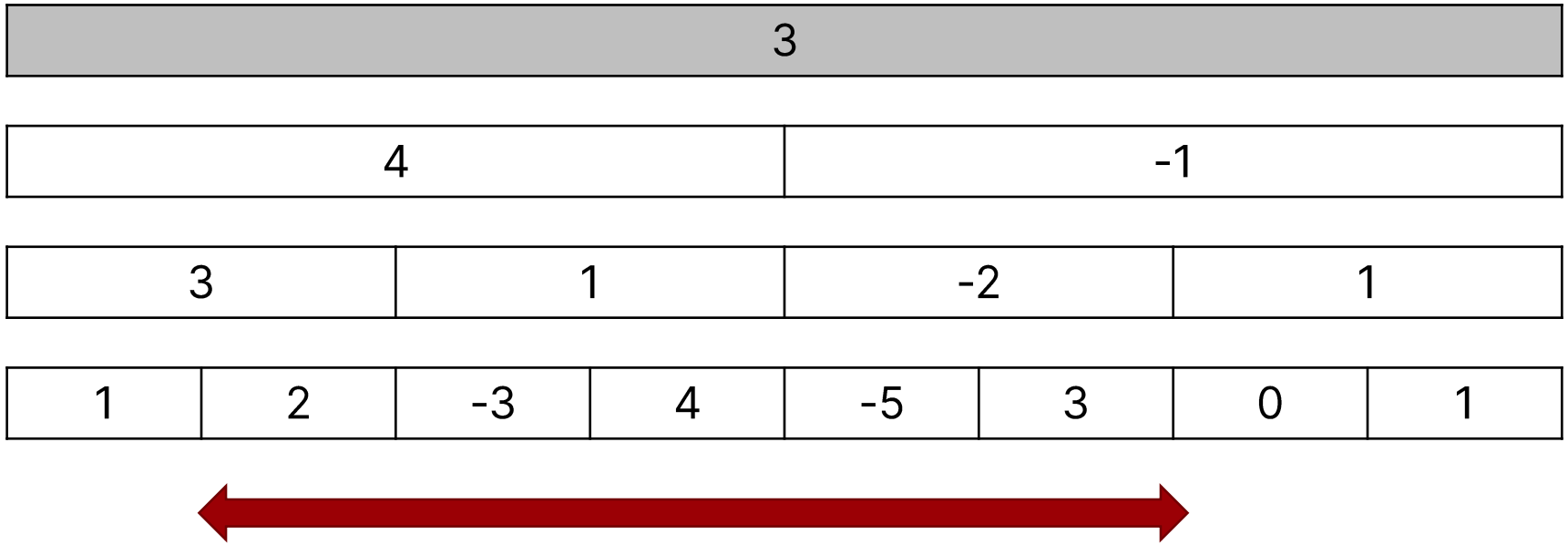
- 노드가 커버하는 구간 ⟷ 우리가 계산하는 구간

1. 아예 겹치지 않거나
   -> 해당 노드를 버린다

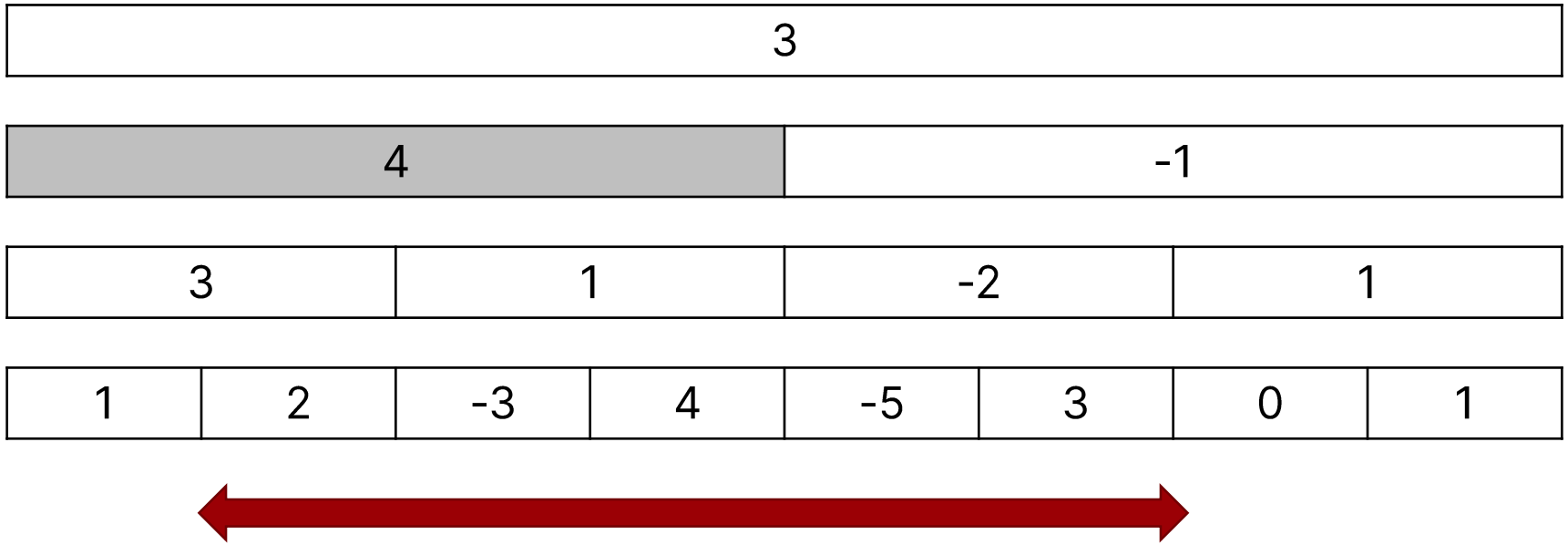
2. 노드의 구간 < 우리가 계산하는 구간에 포함되거나
   -> 해당 노드의 값을 답에 합쳐준다

3. 서로 일부 겹치거나
   -> 구간을 반으로 나눠서 각각 구해진 값을 합쳐준다

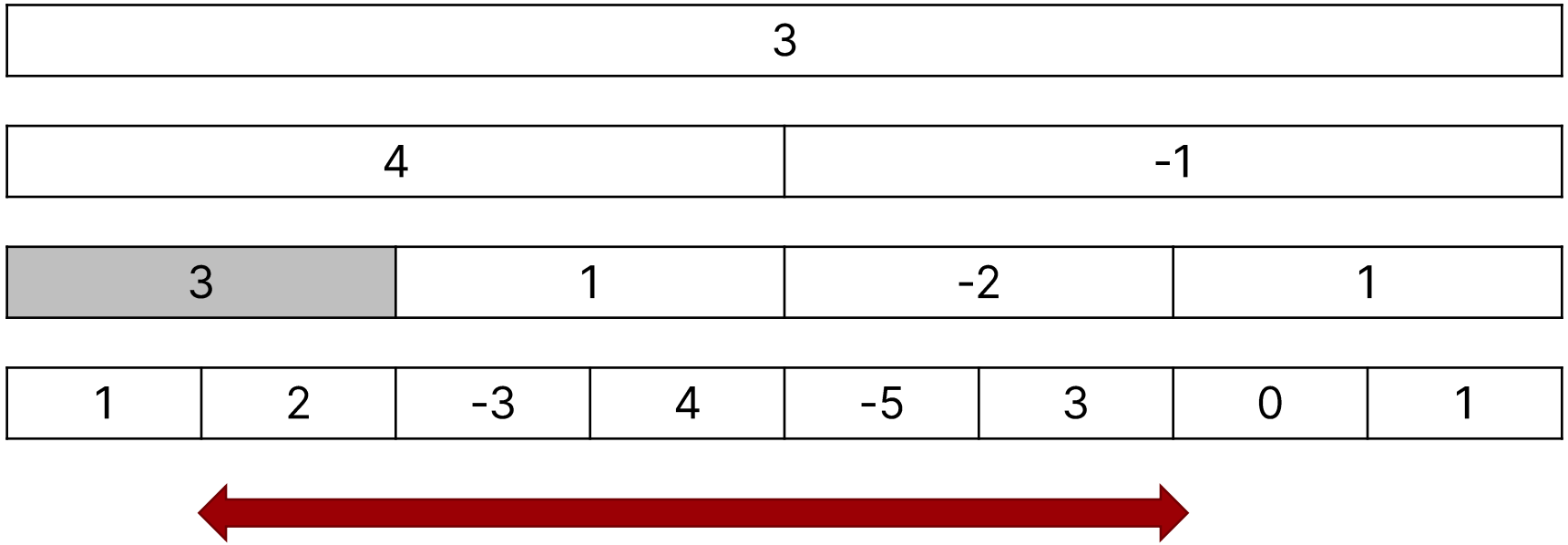
- Sum[2, 6]
- 첫번째 노드에서 시작: [1, 8]
- Case 3: 일부 겹침
  -> 양쪽으로 분할



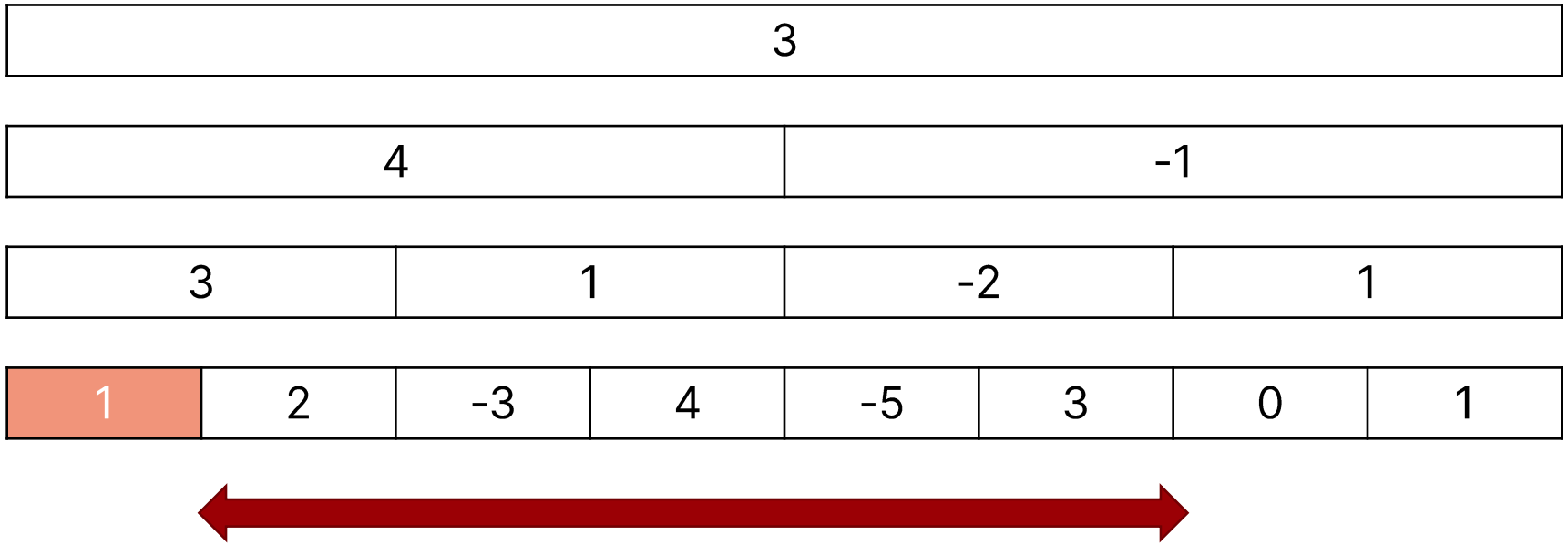
- Sum[2, 6]
- 왼쪽 자식 노드: [1, 4]
- Case 3: 일부 겹침
  -> 양쪽으로 분할



- Sum[2, 6]
- 왼쪽 자식 노드: [1, 2]
- Case 3: 일부 겹침
  -> 양쪽으로 분할

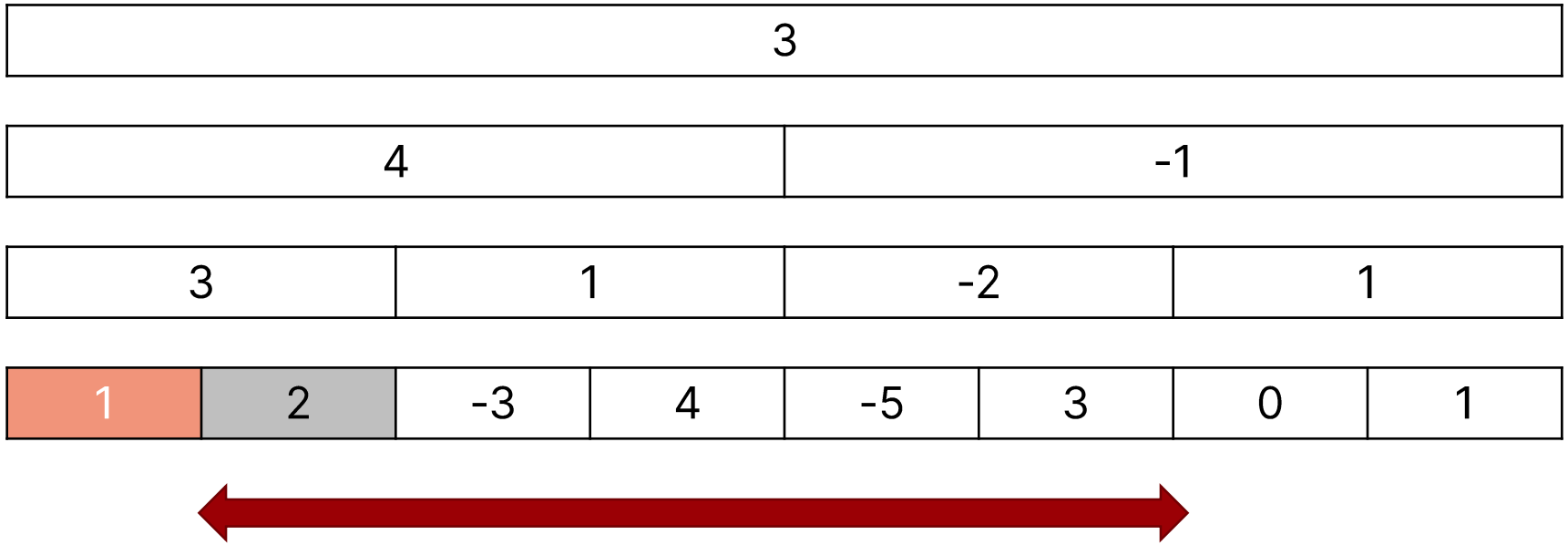


- Sum[2, 6]
- 왼쪽 자식 노드: [1, 1]
- Case 1: 아예 안 겹침
  -> 버림

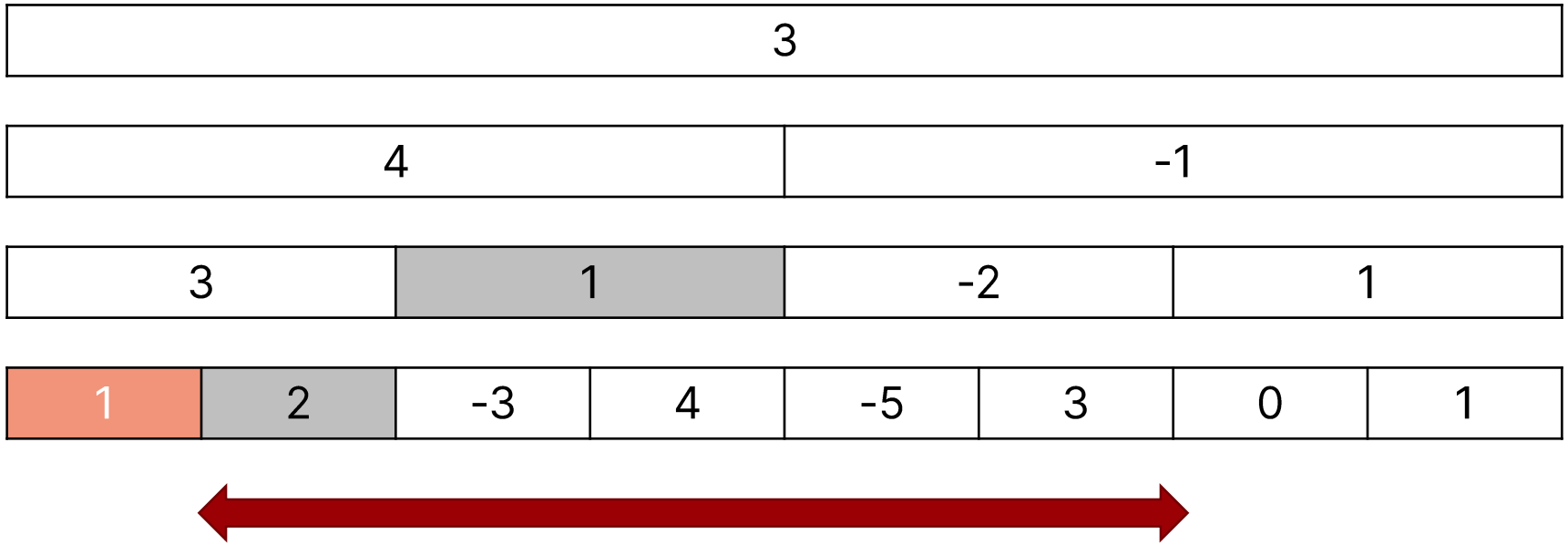


- Sum[2, 6]
- 오른쪽 자식 노드: [2, 2]
- Case 2: 노드구간 < 연산구간

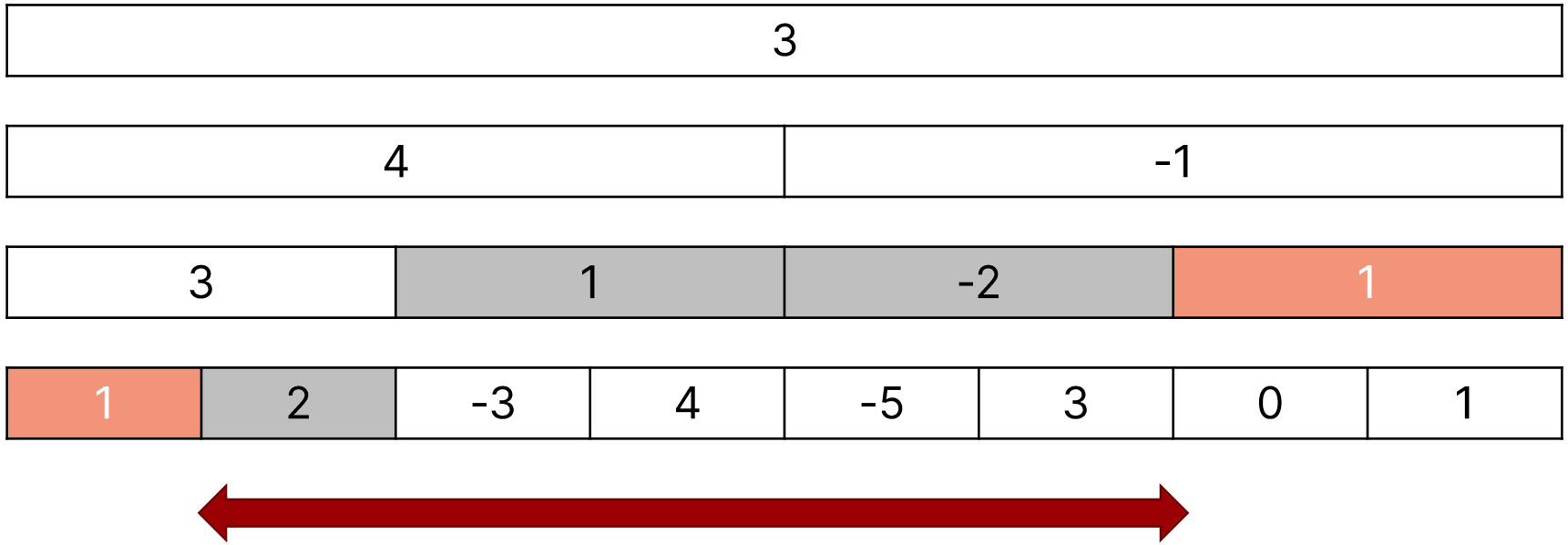
  -> return 노드값



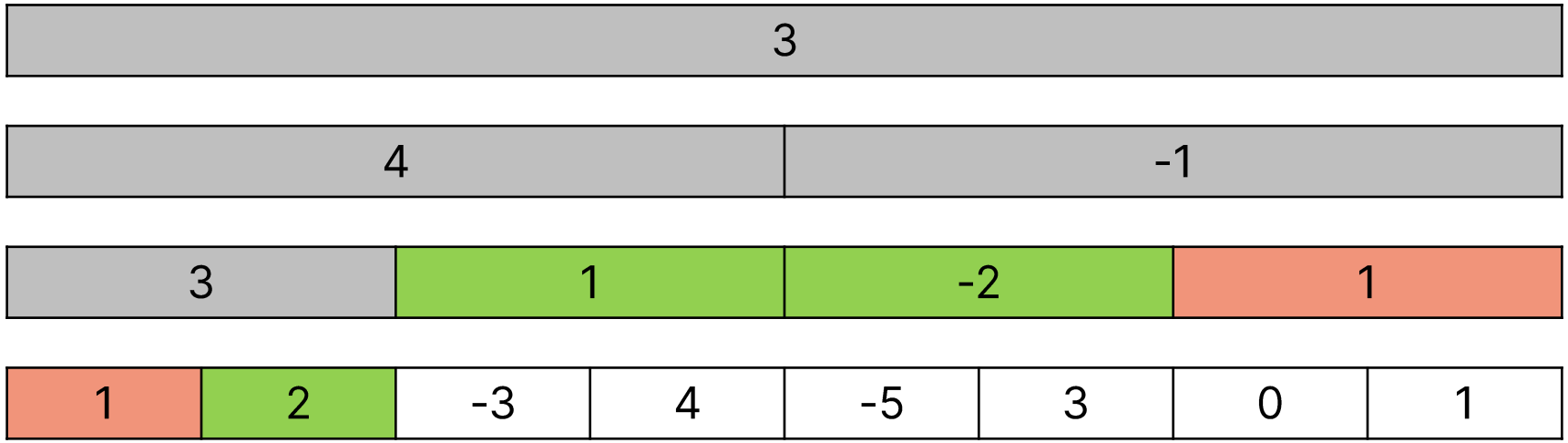
- Sum[2, 6]
- 오른쪽 자식 노드: [3, 4]
- Case 2: 노드구간 < 연산구간
  -> return 노드값



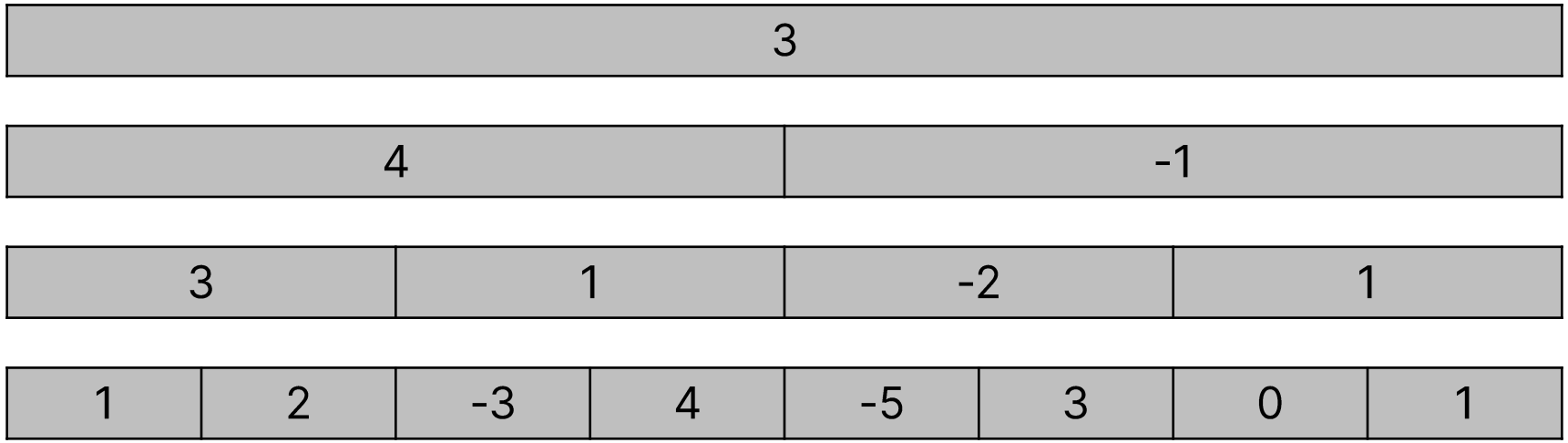
- Sum[2, 6]
- 최종 결과
  - 2 + 1 + -2 = 1



- Query 시간복잡도
  - 각 노드에서는
    - 버리거나(쓰레기값을 return)
    - 값을 return하거나
    - 분할하여 재귀 <- - 병목



- Query 시간복잡도
  - 최악의 경우
    - 모든 노드에서 분할 -> O(n)



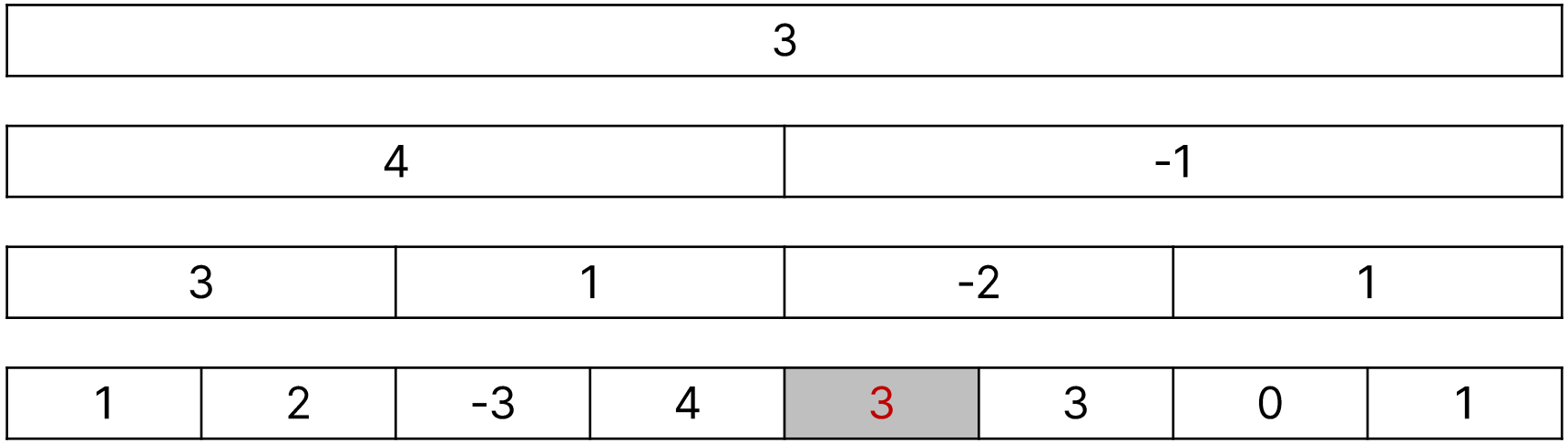
- Query 시간복잡도
  1. 아예 겹치지 않거나
     -> 관심없는 구간에 위치

  2. 노드의 구간 < 우리가 계산하는 구간에 포함되거나
     -> 관심있는 구간에 위치

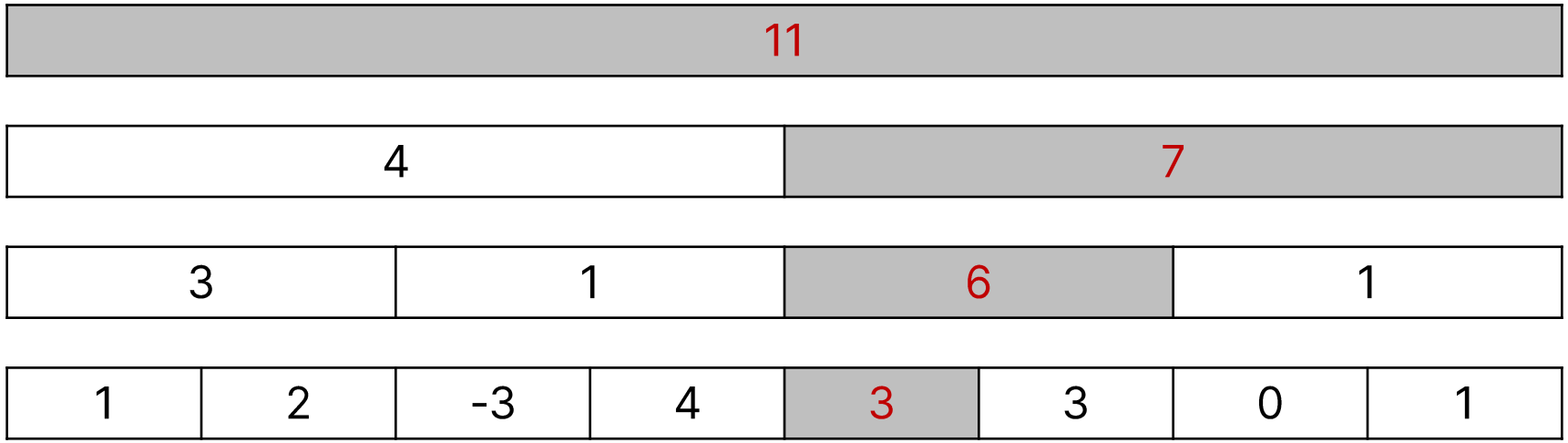
  3. 서로 일부 겹치거나
     -> 노드가 분할할 조건: 관심없는 구간 | 관심있는 구간 에 걸쳐있는 경우

- Query 시간복잡도
  - 관심x | 관심o 사이에 걸쳐있는 노드는 각 레벨에 최대 2개 존재
    - 나머지 노드는 뭔가를 return하고 stop
  - 2개가 분할해봐야 4개
  - 각 레벨에서 만지는 노드의 개수는 최대 4개 -> 전체 = 최대 4*logn개 -> O(logn)

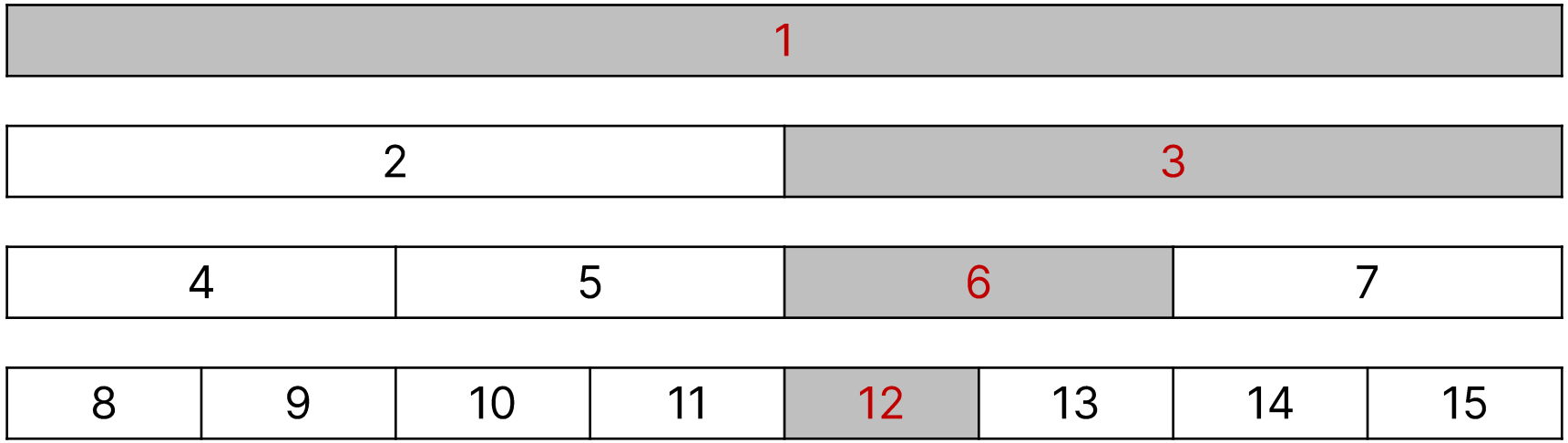
- Update 시간복잡도
  - arr[5] = 3



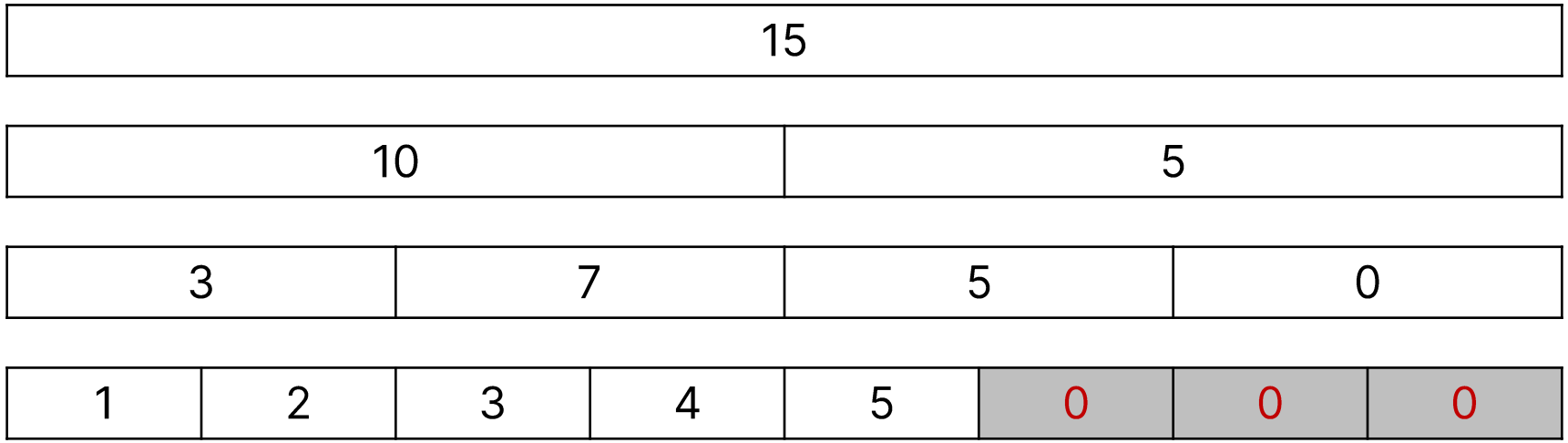
- Update 시간복잡도
  - 어떤 인덱스가 업데이트되면, 함께 업데이트해야 할 구간?
  - 해당 인덱스를 포함하는 구간
  - 루트까지의 경로! -> O(logn)



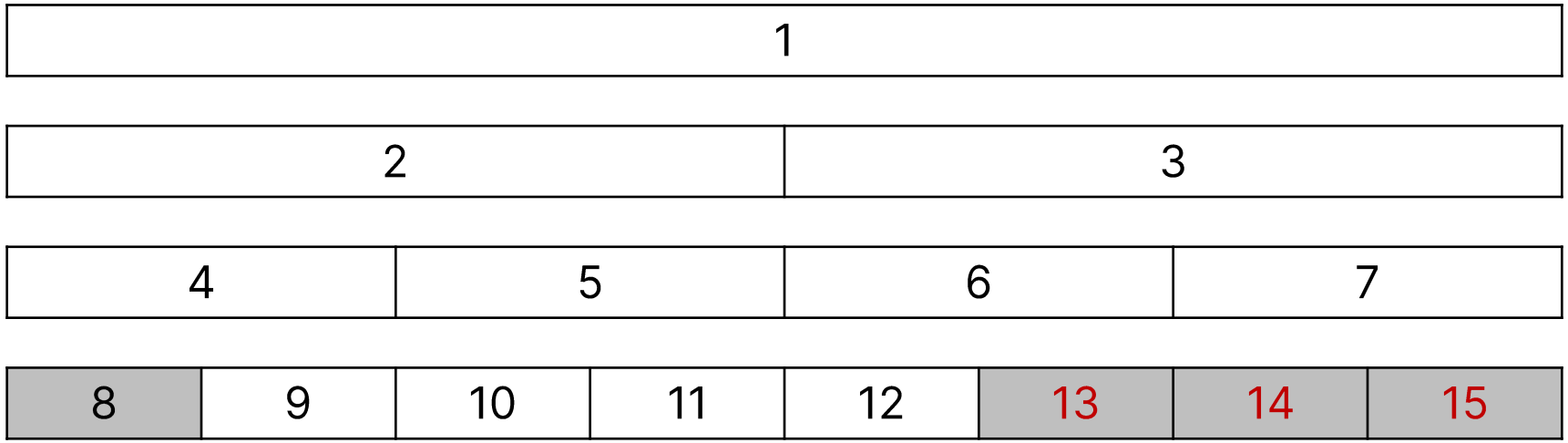
- Update 실제 구현
  - Full Binary Tree의 인덱스 특징 활용
    - 두 child node의 인덱스는 각각 2*parent, 2*parent+1
  - 리프부터 2로 나눠가며 parent로 접근



- 공간복잡도
  - n + n/2 + n/4 + … + 1 ≤ 2*n -> O(n)
  - 구현의 편리성을 위해 주어진 배열의 크기를 2의 거듭제곱꼴로 만들자
    - Ex) 배열이 5개라면 리프가 8개라고 가정. 추가한 리프노드들에는 쓰레기값 투척
    - 리프의 개수가 최대 2*n 가까이 -> seg배열의 크기는 4*n



리프의 인덱스가 리프의 개수로 시작 -> 접근이 편함



- 정리
  - 세그먼트 트리는 구간에 대한 업데이트와 쿼리를 모두 O(logn)에 처리할 수 있는 자료구조이다.
    - 쿼리: 루트 노드부터 시작, 각 노드에 대해 제외/포함/분할의 3가지 경우로 처리
    - 업데이트: 해당 인덱스 -> 루트까지의 경로상의 노드들을 수정

  - 구현 코드 링크(합 세그트리 기준)

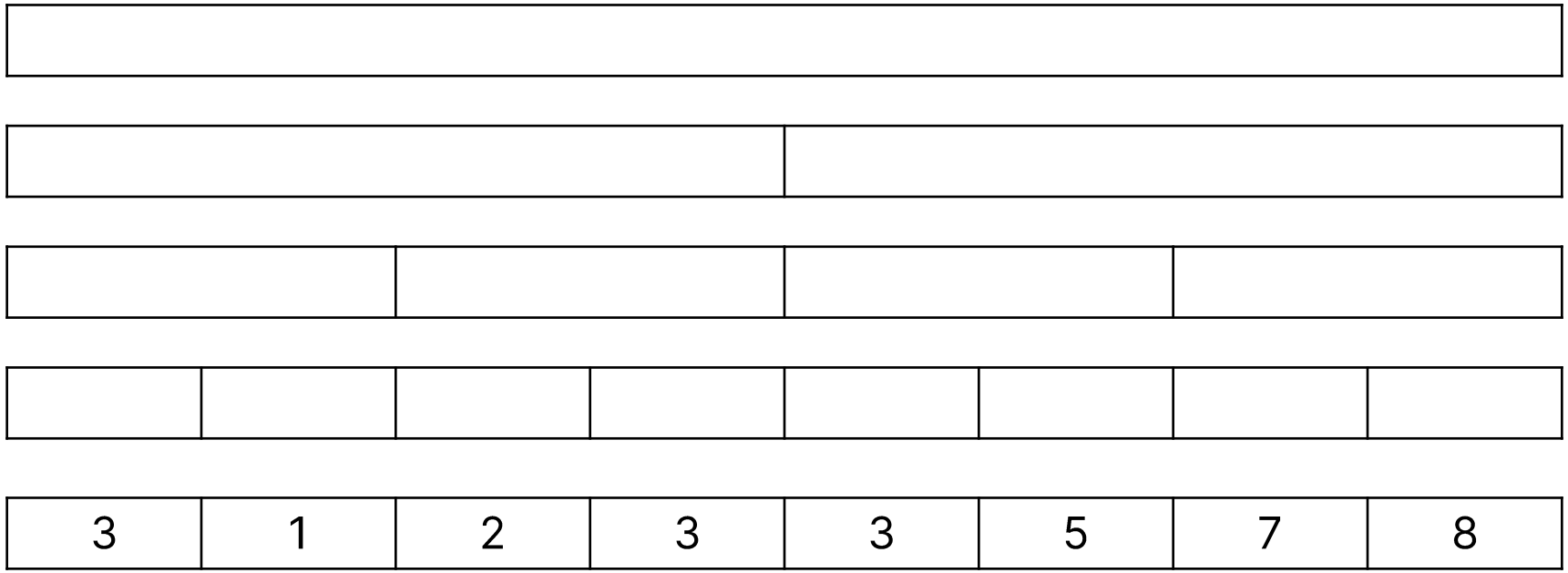
Segment Tree 예제

- 기본형
- 값의 개수 세기
- Plane Sweeping

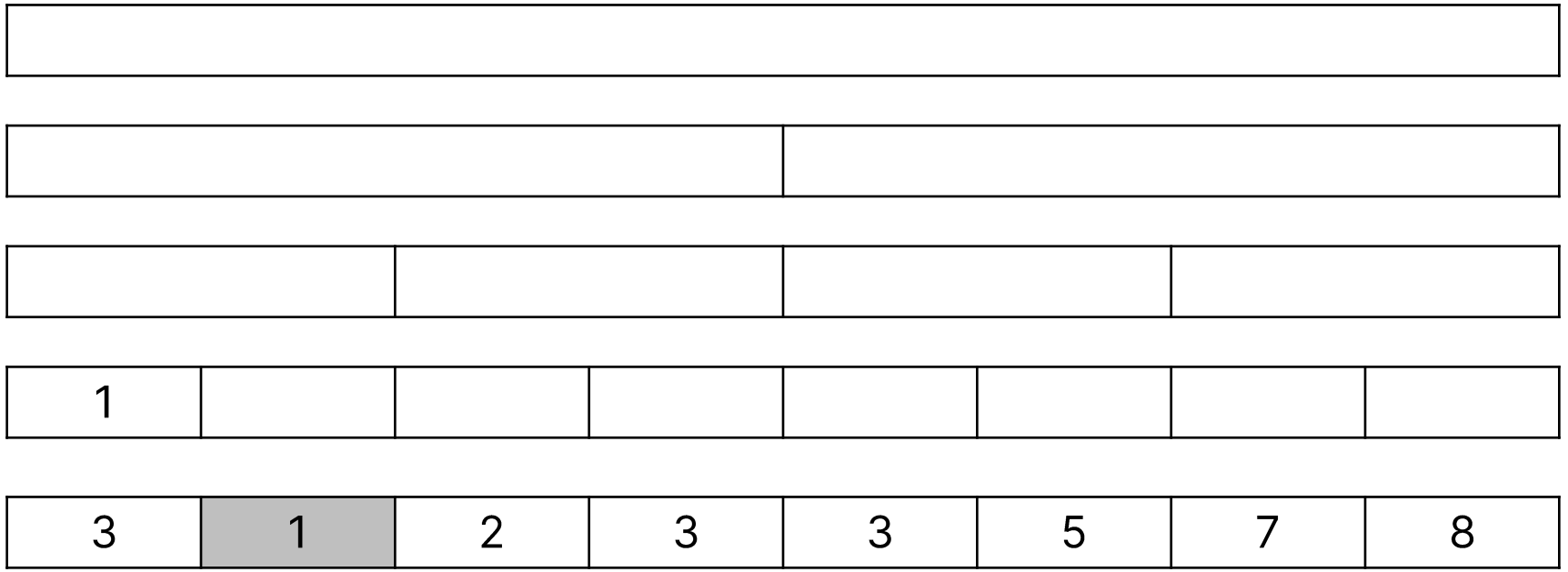
- 주어진 배열에 대해 2가지 쿼리를 처리하라.
  - 주어진 구간 합, 곱, 최대, 최소값을 출력
  - 주어진 인덱스값을 Update

- 주어진 배열에 대해 2가지 쿼리를 처리하라.
  - 주어진 구간 합, 곱, 최대, 최소값을 출력
  - 주어진 인덱스값을 Update
- 앞서 설명한대로 구현
- 쓰레기값에 주의
  - 합: 0
  - 곱: 1
  - 최대: -INF
  - 최소: INF

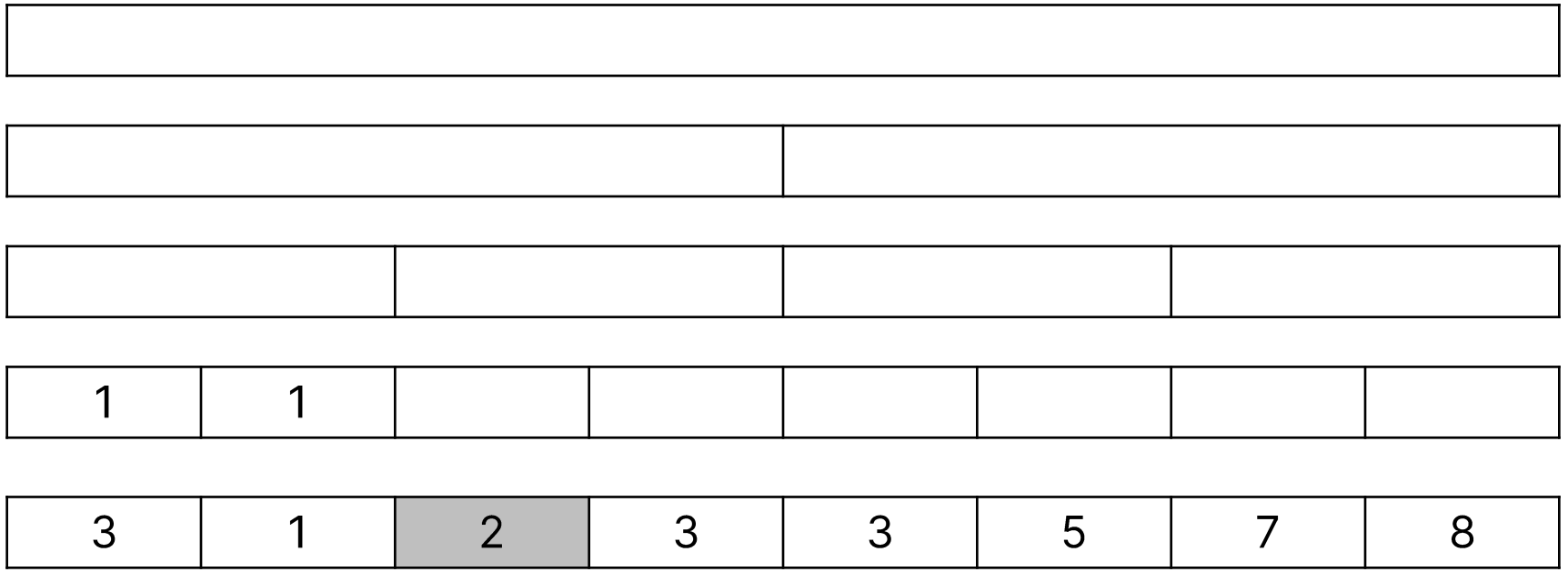
- Segment Tree의 인덱스를 값으로 활용
  - 가장 왼쪽에 있는 리프노드 = 배열 안에 1의 개수
  - 다음 리프노드 = 배열 안에 2의 개수
  - ...
  - Sum[l, r] = 값이 [l, r] 범위에 있는 원소의 개수



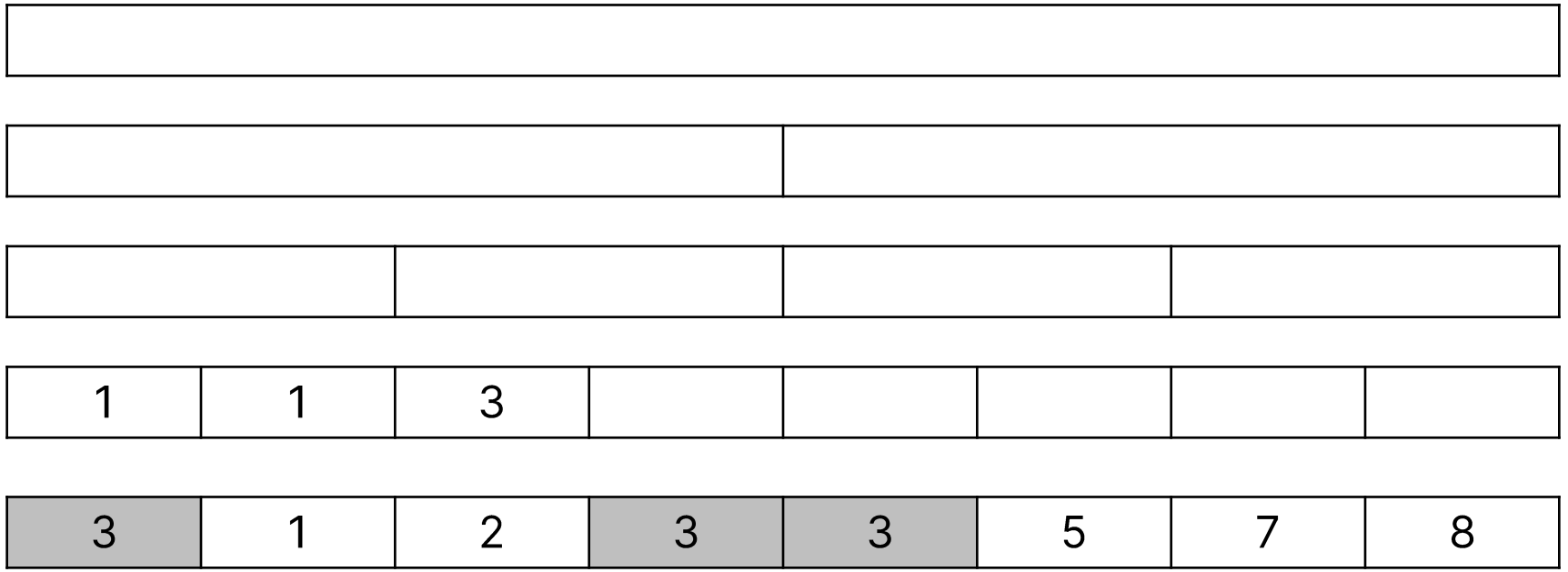
- Segment Tree의 인덱스를 값으로 활용
  - 가장 왼쪽에 있는 리프노드 = 배열 안에 1의 개수
  - 다음 리프노드 = 배열 안에 2의 개수
  - ...
  - Sum[l, r] = 값이 [l, r] 범위에 있는 원소의 개수



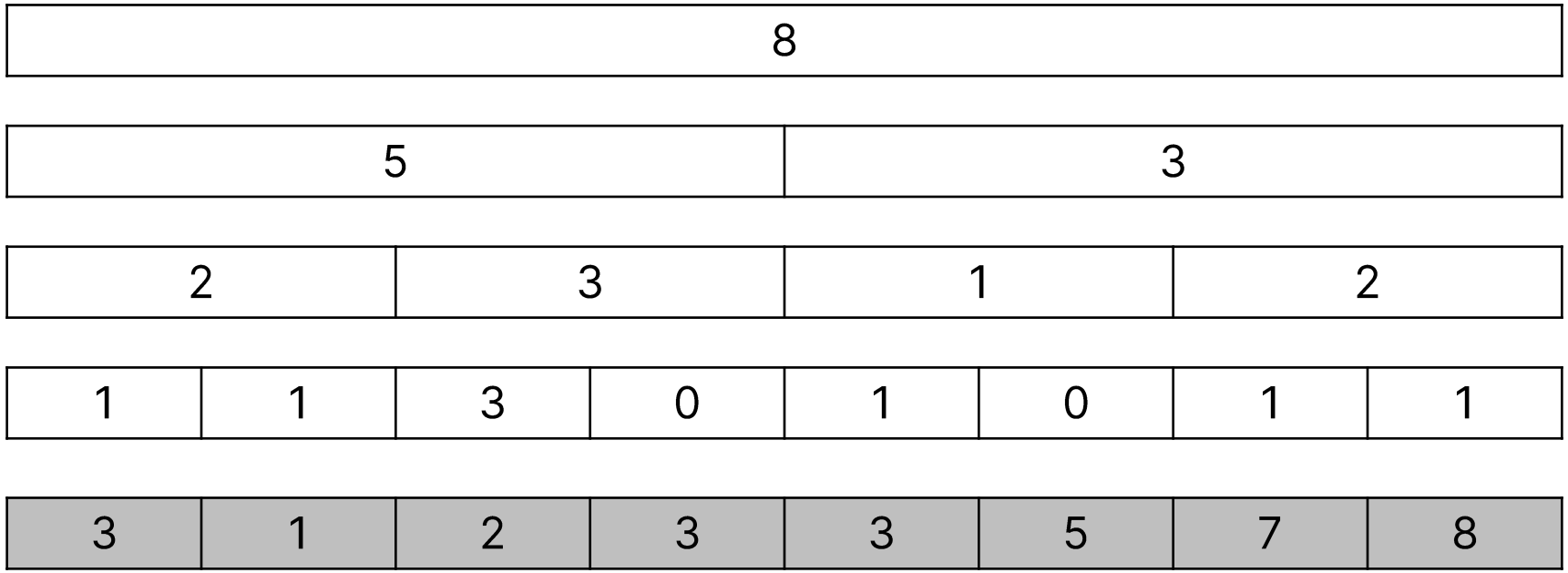
- Segment Tree의 인덱스를 값으로 활용
  - 가장 왼쪽에 있는 리프노드 = 배열 안에 1의 개수
  - 다음 리프노드 = 배열 안에 2의 개수
  - ...
  - Sum[l, r] = 값이 [l, r] 범위에 있는 원소의 개수



- Segment Tree의 인덱스를 값으로 활용
  - 가장 왼쪽에 있는 리프노드 = 배열 안에 1의 개수
  - 다음 리프노드 = 배열 안에 2의 개수
  - ...
  - Sum[l, r] = 값이 [l, r] 범위에 있는 원소의 개수



- Segment Tree의 인덱스를 값으로 활용
  - 가장 왼쪽에 있는 리프노드 = 배열 안에 1의 개수
  - 다음 리프노드 = 배열 안에 2의 개수
  - ...
  - Sum[l, r] = 값이 [l, r] 범위에 있는 원소의 개수



Counting Inversions

- Inversion을 정의
  - 어떤 배열에서 앞에 있는 값이 뒤에 있는 값보다 큰 쌍
- 주어진 배열에서 inversion인 쌍의 개수 세기

4	2	7	1	5	6	3	8

4	2	7	1	5	6	3	8

Counting Inversions

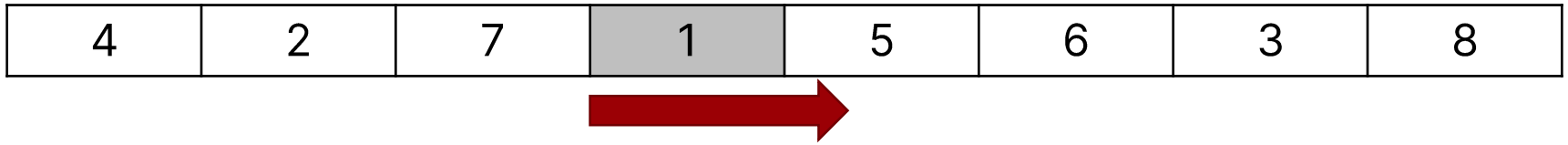
- 각 원소에 대해, 나보다 크면서 내 앞에 있는 수의 개수
  - 값의 개수 세는 seg tree를 활용
  - 내 앞의 원소들까지 반영되어 있다면, 값이 나보다 큰 수의 개수를 빠르게 구할 수 있다.
- Ex) 현재 원소가 1이라면, 현재 세그에서 sum[2, MAX]의 개수 = 3

4	2	7	1	5	6	3	8

0	1	0	1	0	0	1	0

Counting Inversions

- 각 원소에 대해, 나보다 크면서 내 앞에 있는 수의 개수
  - 값의 개수 세는 seg tree를 활용
  - 내 앞의 원소들까지 반영되어 있다면, 값이 나보다 큰 수의 개수를 빠르게 구할 수 있다.
- Ex) 현재 원소인 1을 세그트리에 반영하고 넘어감



Counting Inversions

- 각 원소에 대해, 나보다 크면서 내 앞에 있는 수의 개수
  - 값의 개수 세는 seg tree를 활용
  - 내 앞의 원소들까지 반영되어 있다면, 값이 나보다 큰 수의 개수를 빠르게 구할 수 있다.
- Ex) 현재 원소가 1이라면, 현재 세그에서 sum[2, MAX]의 개수

- 배열을 앞에서부터 훑으며 아래의 동작을 반복
  - 값의 범위가 [a[i]+1, MAX] 안에 있는 원소의 개수를 답에 반영
  - a[i]값을 seg tree에 반영

K번째 작은 수 찾기

- 빈 배열로 시작해 2가지 쿼리 수행
  - 주어진 값을 배열에 추가/삭제
  - 배열의 원소들 중 K번째 작은 수 구하기

  - Ex) K = 3 -> ans = 3
  - Ex) K = 5 -> ans = 5

5	2	7	1	5	6	3	8

K번째 작은 수 찾기

- 현재 배열의 상태

5	2	7	1	5	6	3	8

- 값의 개수 세그

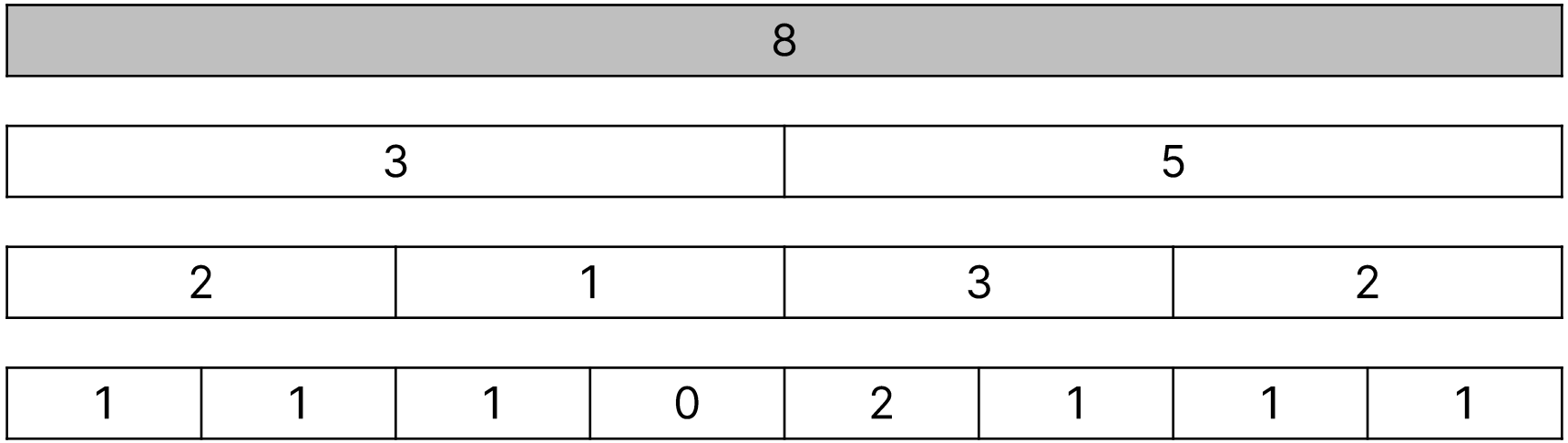
8

3	5

2	1	3	2

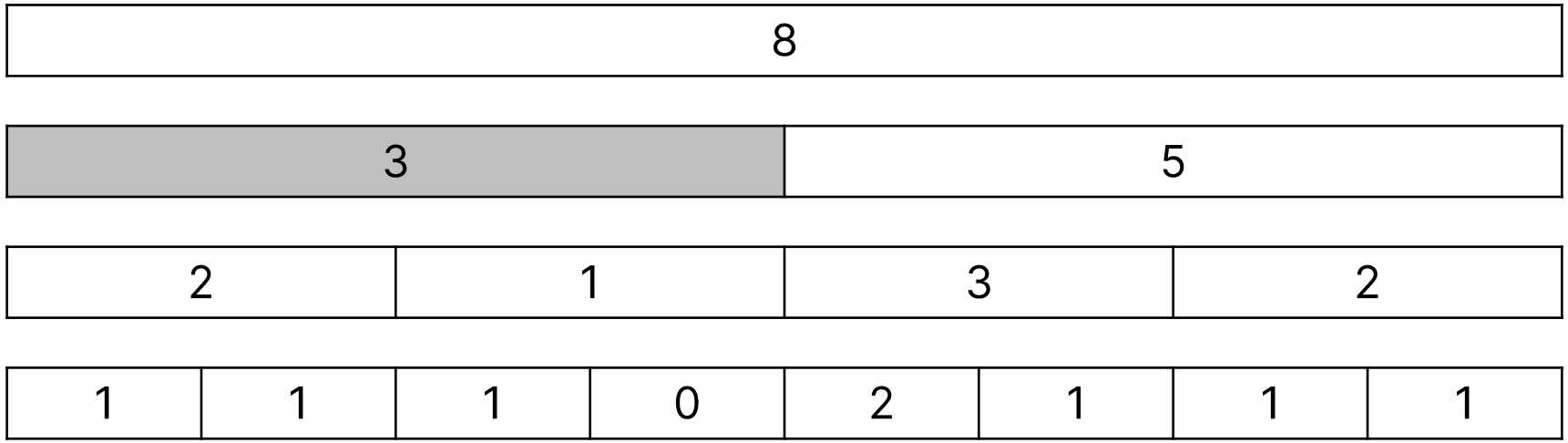
1	1	1	0	2	1	1	1

- K=3
- 왼쪽 노드값이 K 이상이므로, 오른쪽 구간은 고려할 필요가 없음



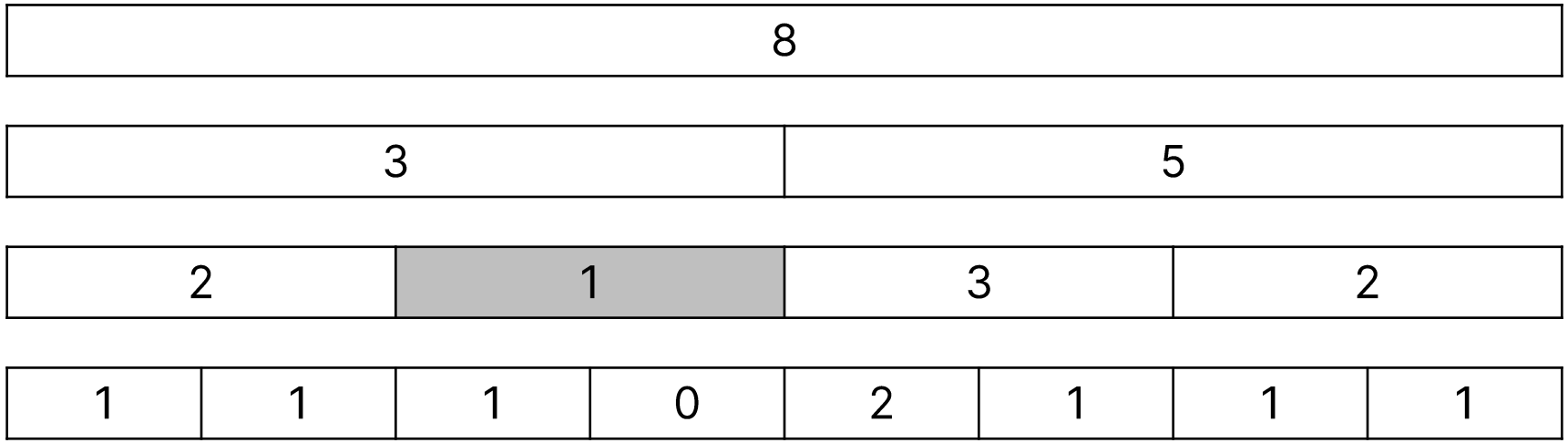
K번째 작은 수 찾기

- K=3
- 왼쪽 구간의 개수가 K보다 작으므로, K번째 작은 수는 오른쪽 구간에 존재
- 전체구간에서 3번째 수 = 오른쪽 구간에서 1번째 수
  - K를 3 – 2 = 1로 조정



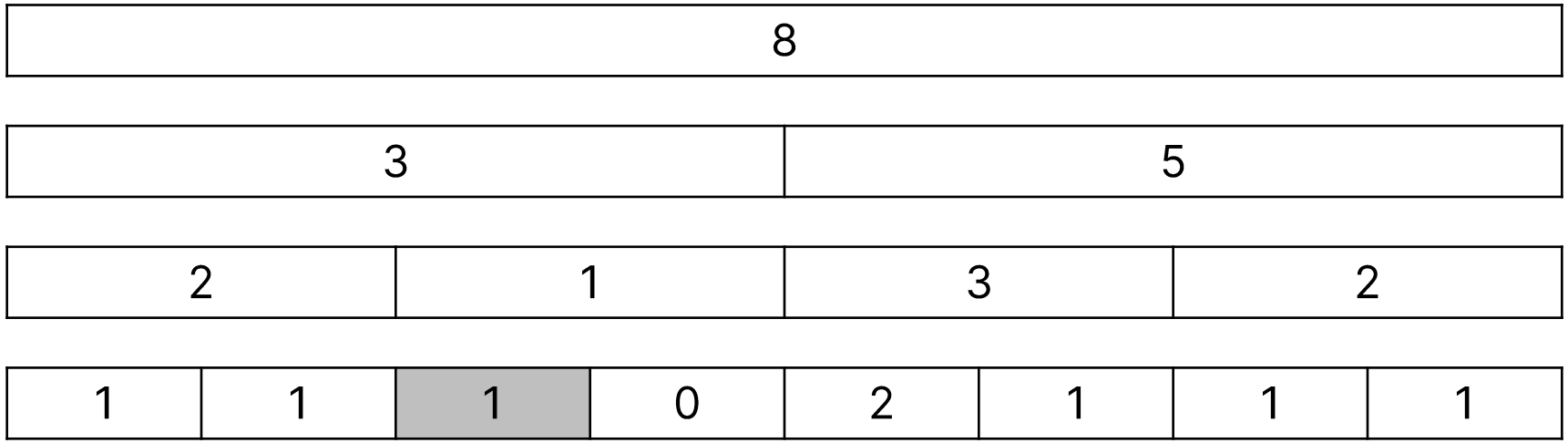
K번째 작은 수 찾기

- K=1
- 왼쪽 구간의 개수가 K이하이므로 왼쪽으로 이동

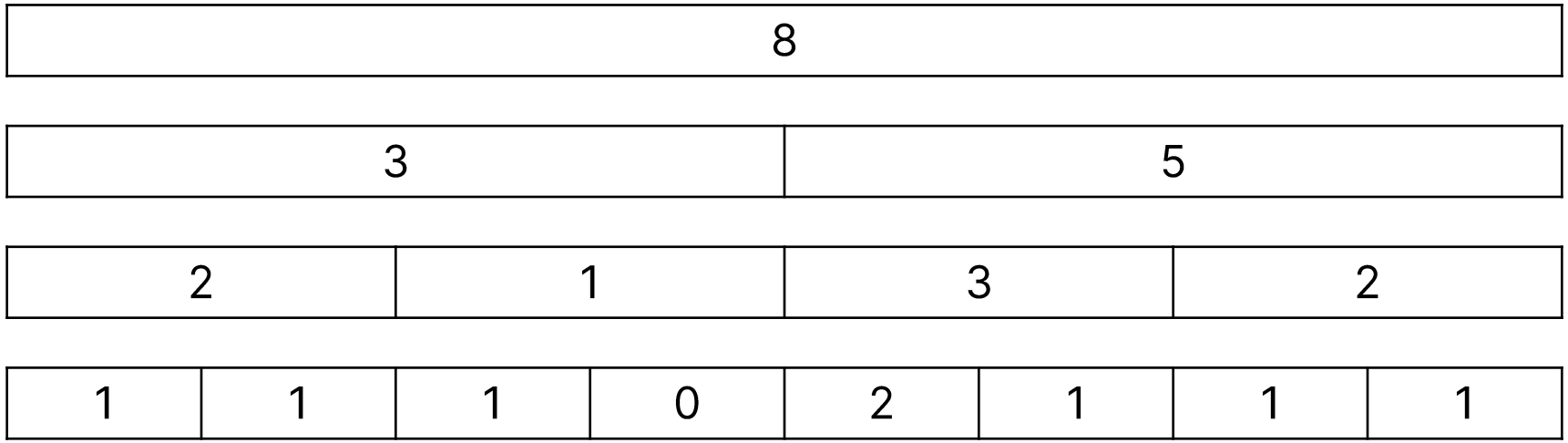


K번째 작은 수 찾기

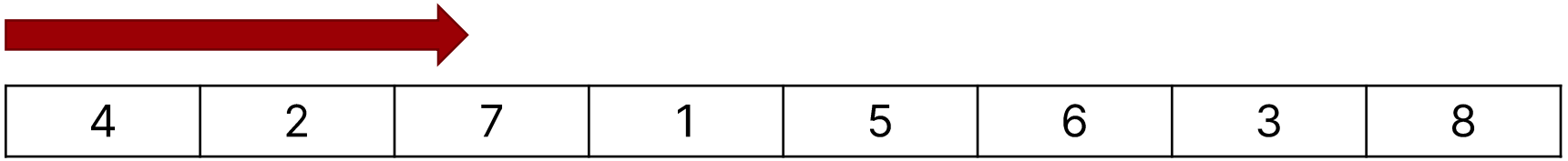
- K=1
- 답: 3



- Update
  - 추가/삭제되는 값에 대응하는 인덱스에 +- 1



- Sweeping
  - 1차원 배열의 앞에서부터 순차적으로 어떤 동작을 수행
  - Ex) Counting Inversions

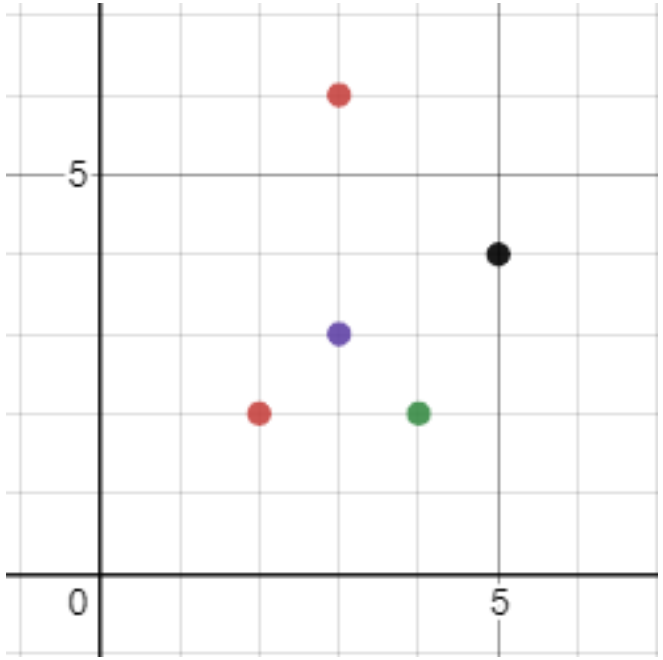


- Plane Sweeping
  - 2차원 평면에서 순서대로 어떤 동작을 수행

문제 설명

- 좌표평면 위에 점들이 뿌려져 있다
- 절대우위 정의
  - 두 점 P1(x1, y1), P2(x2, y2)에 대해
  - $x1 \leq x2, y1 \leq y2$ 를 만족할 때
  - P2가 P1에게 절대우위를 가진다.
- 절대우위 쌍의 개수를 구하여라.

예시



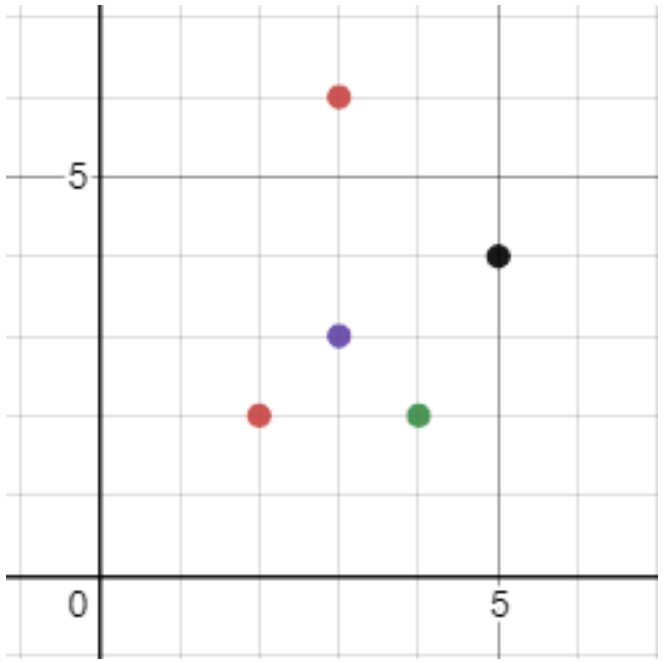
답: 7

문제 설명

- 좌표평면 위에 점들이 뿌려져 있다
- 절대우위 정의
  - 두 점 P1(x1, y1), P2(x2, y2)에 대해
  - $x1 \leq x2, y1 \leq y2$ 를 만족할 때
  - P2가 P1에게 절대우위를 가진다.
- 절대우위 쌍의 개수를 구하여라.

- 나이브 솔루션: O(n^2)

예시

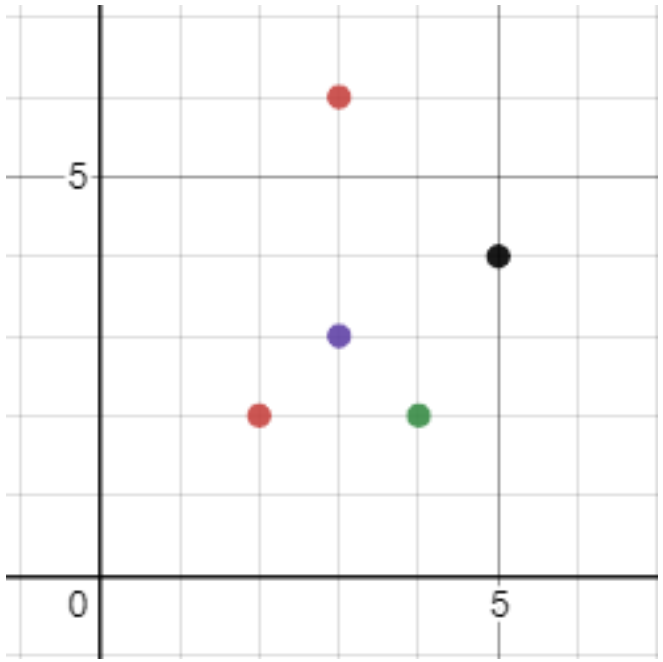


답: 7

풀이

- 어떤 점에 대한 답을 구하기 전에 왼쪽 점들을 모두 세그트리에 반영
  - 먼저 반영된 점 = x좌표는 나 이하 보장
- 반영된 점들 중 y좌표가 자신 이하인 개수만 세어주면 됨

예시



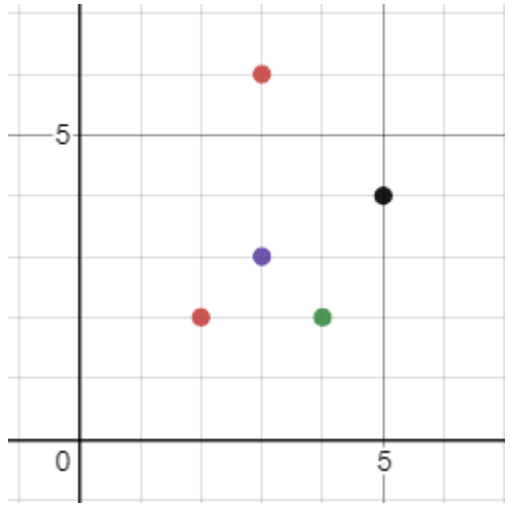
답: 7

풀이

- 어떤 점에 대한 답을 구하기 전에 왼쪽 점들을 모두 세그트리에 반영
  - 먼저 반영된 점 = x좌표는 나 이하 보장
- 반영된 점들 중 y좌표가 자신 이하인 개수만 세어주면 됨

- 모든 점들을 x기준 -> y기준으로 정렬해 두고 순서대로 진행
  - Sum[0, Point[i].y)의 값을 답에 반영
  - Update(Point[i].y)

예시



답: 7

Lazy Propagation

- What is Lazy Propagation?
- 예제

Segment Tree의 한계

- Segment Tree는 단일 원소에 대한 update를 O(logn)에 지원합니다.

- 구간 update는?

Segment Tree의 한계

- Segment Tree는 단일 원소에 대한 update를 O(logn)에 지원합니다.

- 구간 update는?
  - 구간 원소 개수만큼 반복하는 것 외엔 방법이 없습니다...
  - O(klogn): k가 구간의 길이일 때

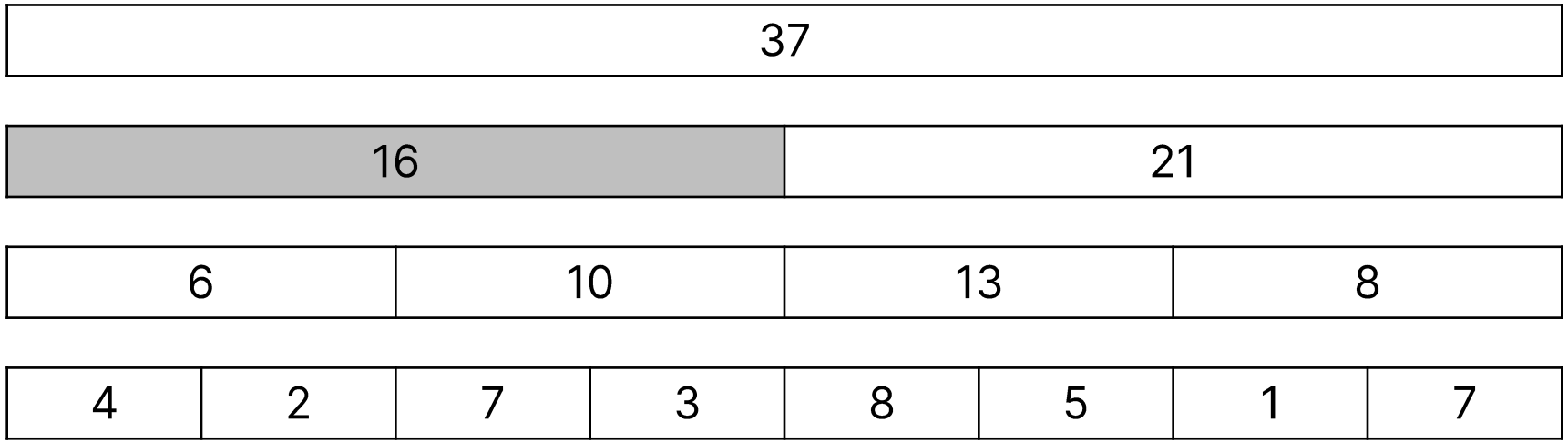
Segment Tree with Lazy Propagation

- Segment Tree with Lazy Propagation은 구간 업데이트를 O(logn)에 제공

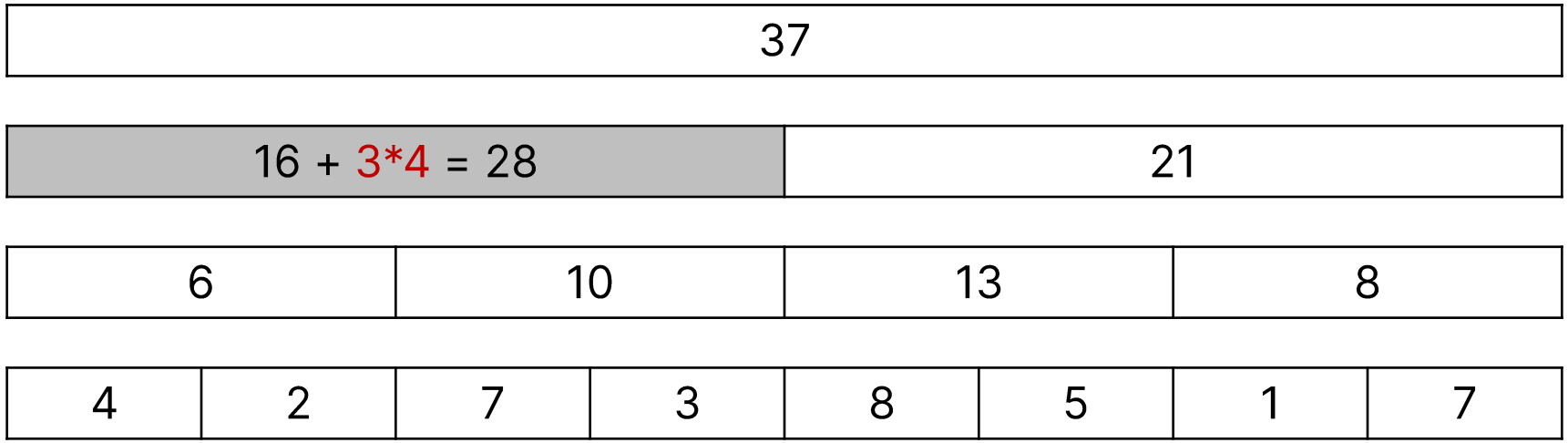
- Lazy Propagation: 게으른 전파
  - Update 요청에 대해 당장 수행하지 않고 뒤로 미룬다!
  - 최대한 미룰 수 있을 때까지
  - ~~마치 저를 보는 것 같네요;;~~

- Lazy 배열을 정의
  - 해당 노드에 이런 업데이트를 수행할 것이다. 언젠가는.
- 어떤 구간을 업데이트해야 할 때, 구간에 해당하는 노드까지만 업데이트
  - 뒷일은 나중에 생각한다
- Propagate 함수 정의
  - 앞에서 미뤄둔 "뒷일"을 수행
  - 내 업데이트를 수행하고, 자식 노드들에게도 전파
- Update함수는 top-down을 사용
  - 밑으로 전파해서 propagate한 값을 다시 반영해야 하므로
  - 예시 동작으로 설명

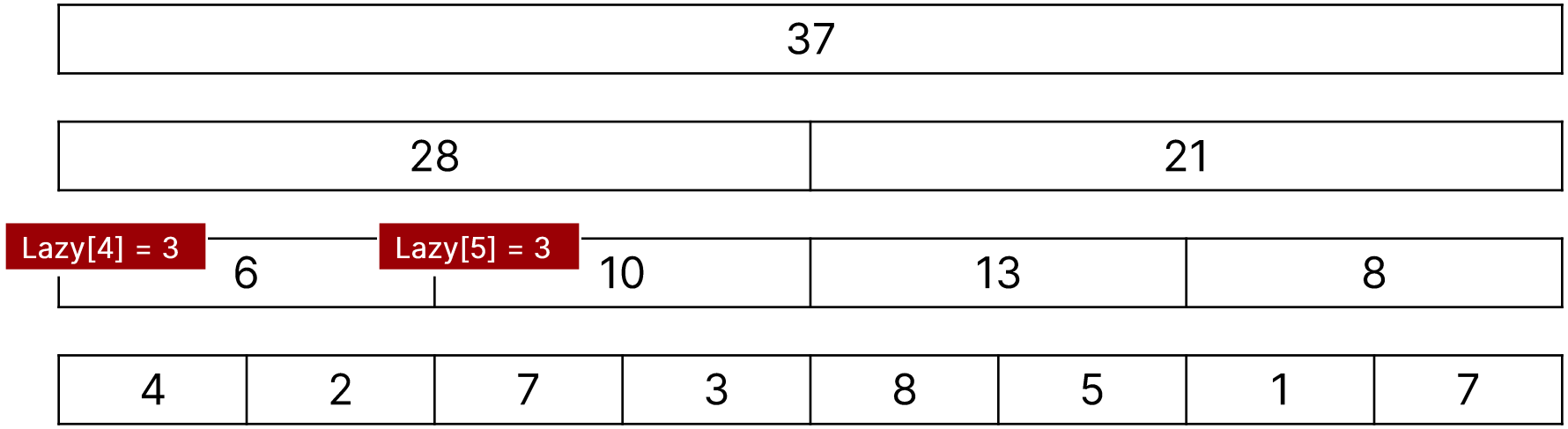
- Sum Segment Tree
- Update[1,4] 내용은 +3



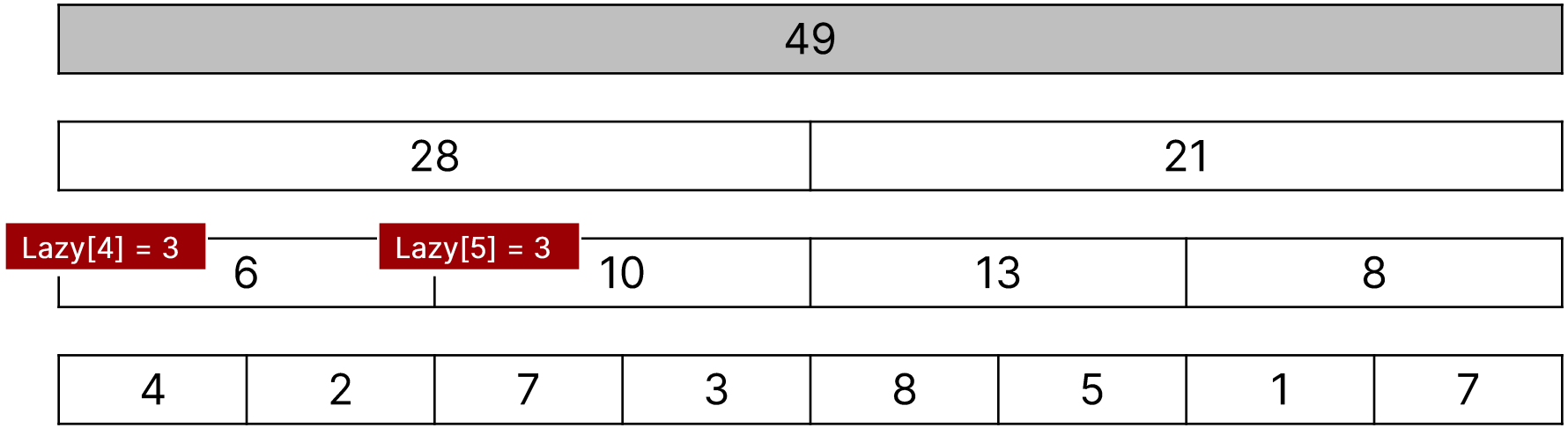
- Sum Segment Tree
- Update[1,4] 내용은 +3
  - 1~4 구간에 해당하는 노드에는 3 * (구간 길이)로 업데이트



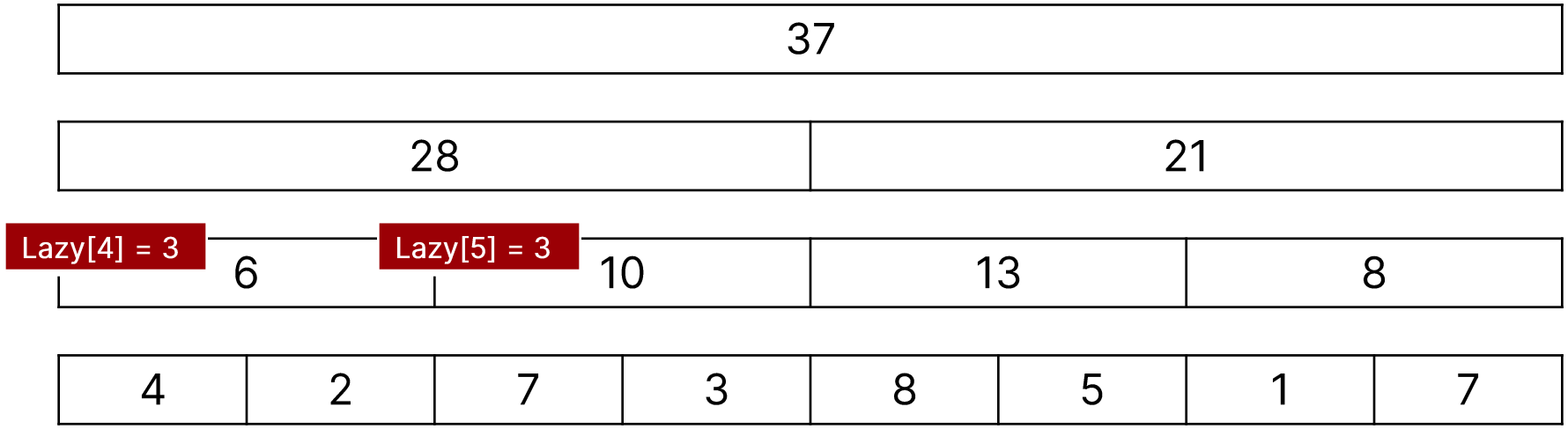
- Update[1,4] 내용은 +3
  - 두 자식 노드에게 lazy배열값 +3
  - seg 배열값은 변화 x
  - 나중에 저 노드들을 만질때 3에 해당하는 update를 해야 한다는 의미.



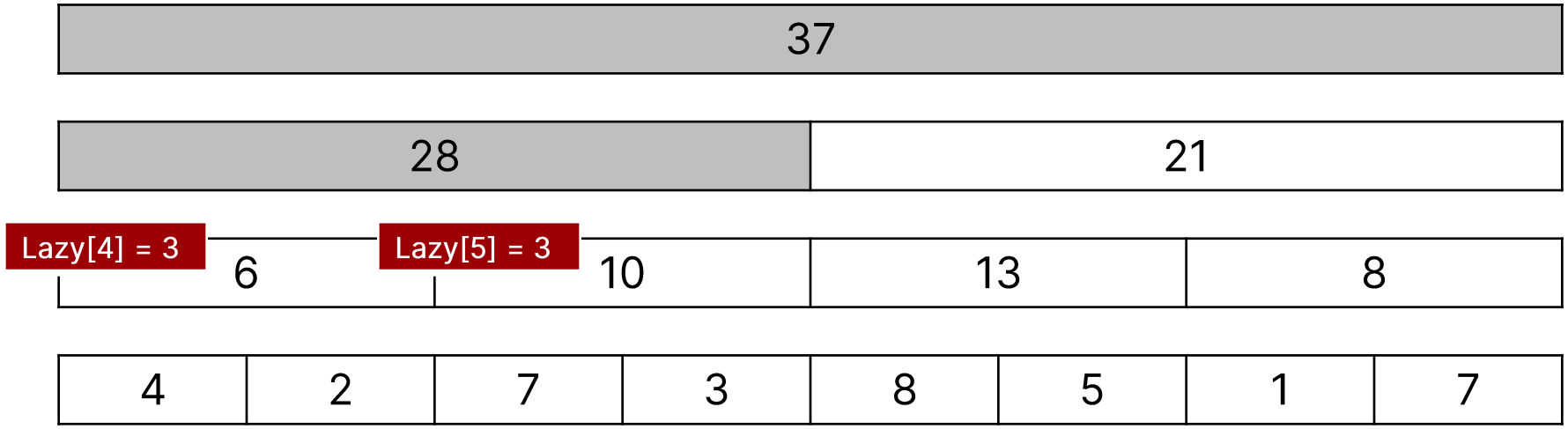
- Sum Segment Tree
- Update[1,4] 내용은 +3
  - 부모값 update



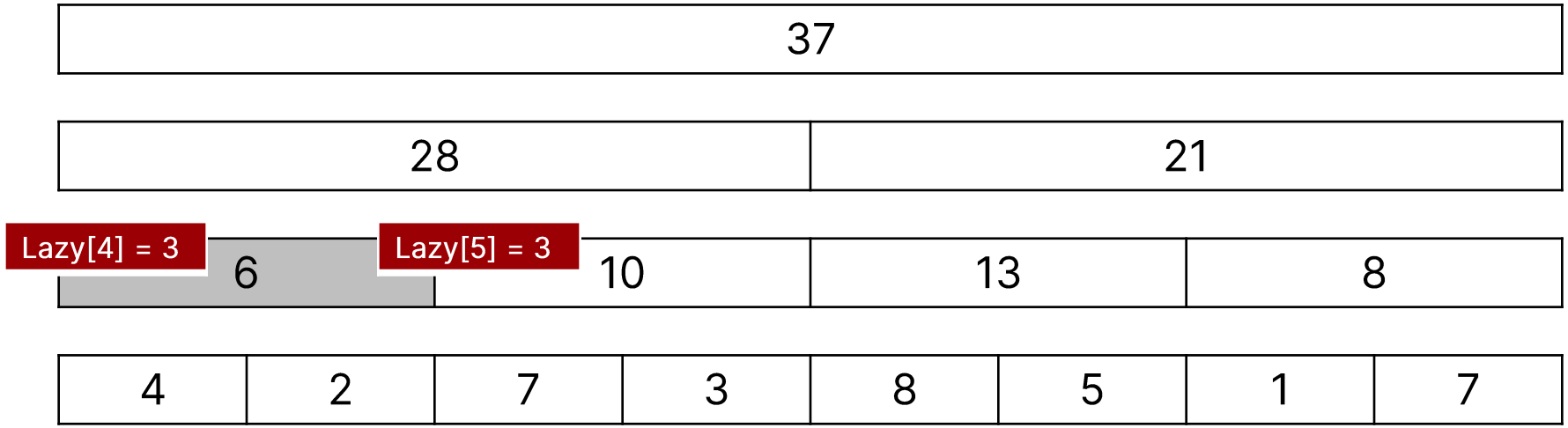
- Sum Segment Tree
- Sum[1,2] 호출



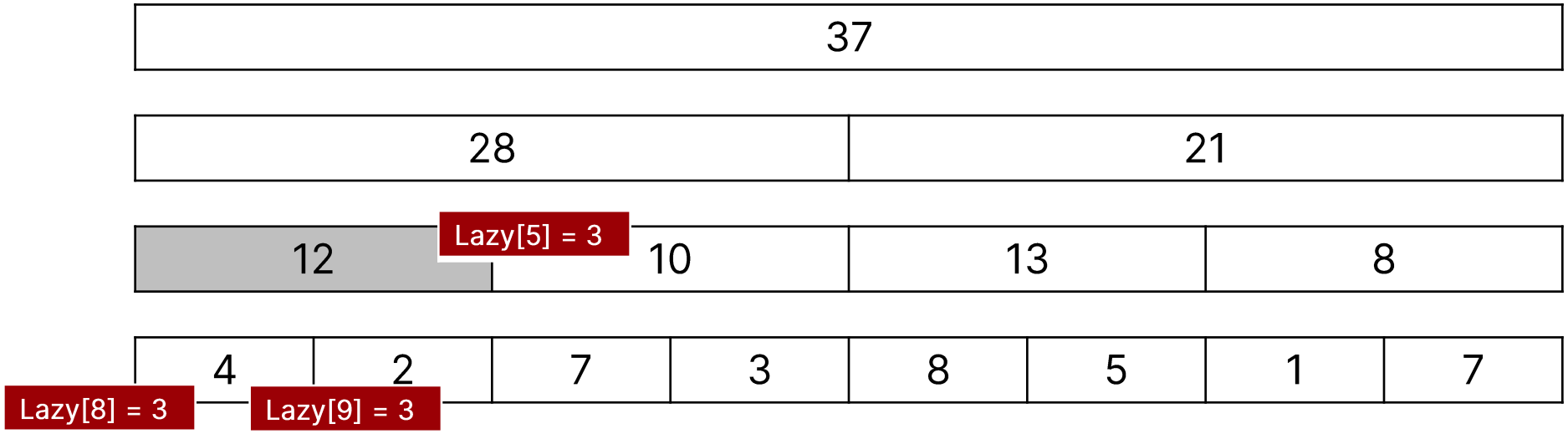
- Sum Segment Tree
- Sum[1,2] 호출



- Sum Segment Tree
- Sum[1,2] 호출
  - 해당 노드를 만질 때 propagate 함수를 호출
  - Lazy배열 값을 반영 -> 6 + 3 * 2 = 12

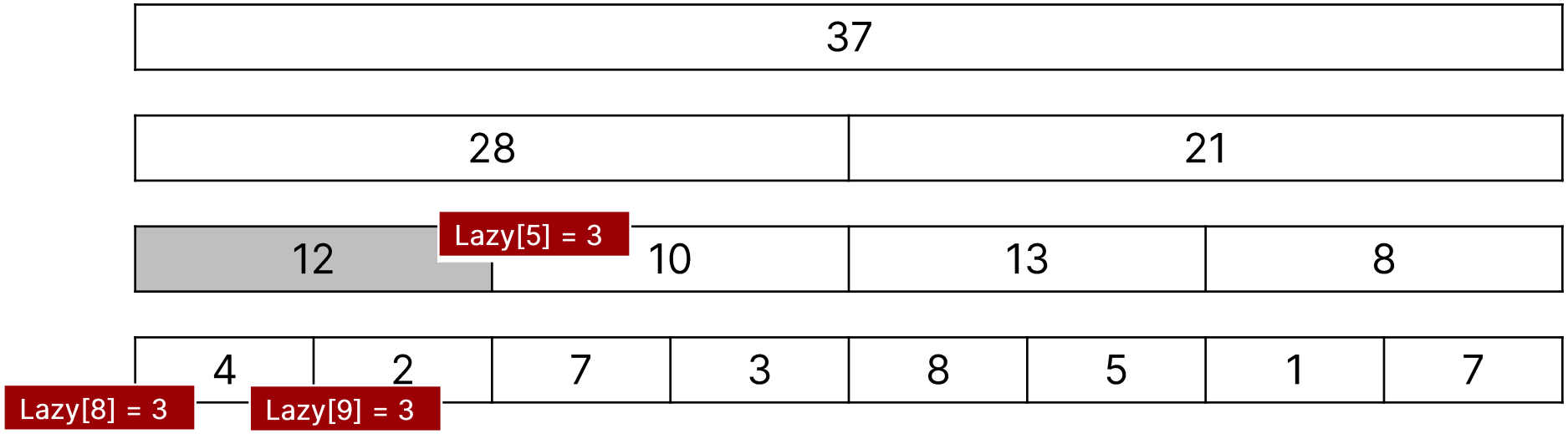


- Sum Segment Tree
- Sum[1,2] 호출
  - 해당 노드를 만질 때 propagate 함수를 호출
  - 자식노드에 Lazy값을 전파



Segment Tree with Lazy Propagation

- Propagate함수는 lazy값을 처리하고 두 자식노드에 뿌려주므로 O(1)
- Update 및 쿼리의 시간복잡도는 그대로 O(logn)
- 구현 코드 링크 (Sum Segment Tree)

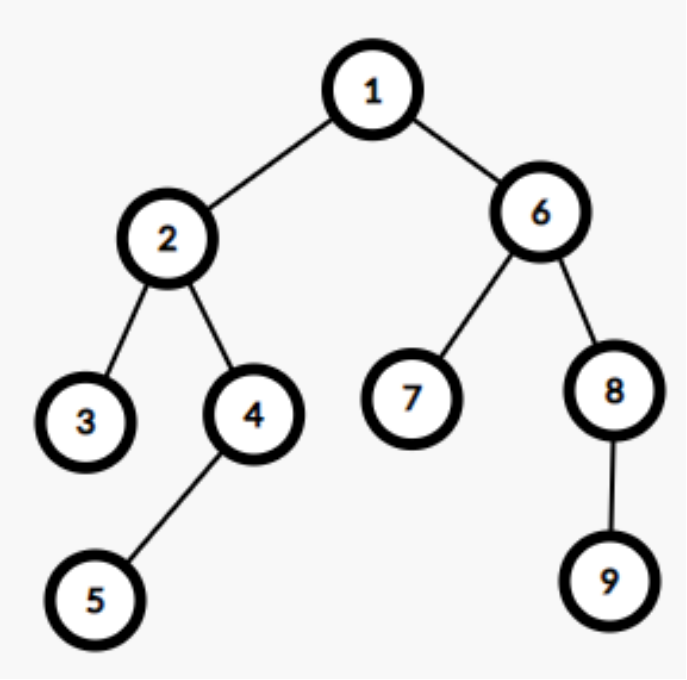


트리상에서의 쿼리

- 주어진 트리구조에서 2가지 쿼리를 수행
  - 어떤 노드의 하위노드에 전부 값을 업데이트
  - 특정 노드의 값을 출력

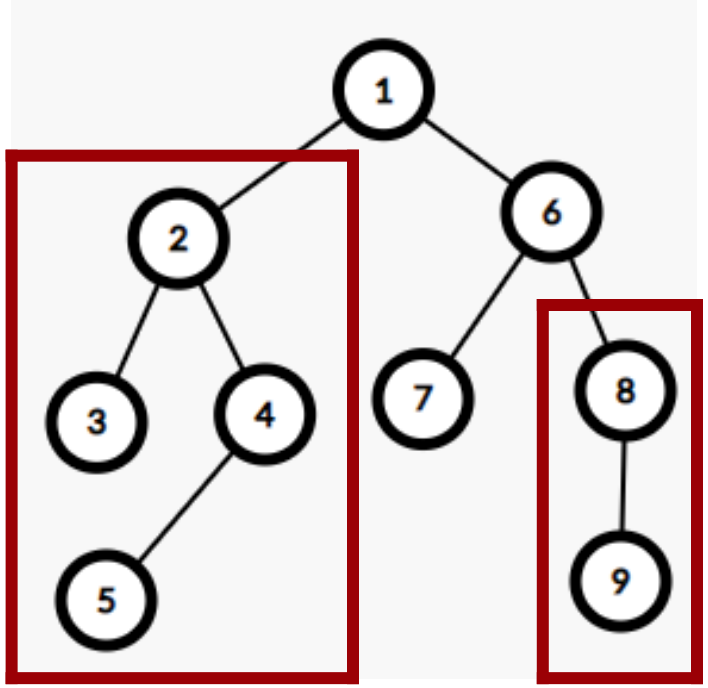
트리상에서의 쿼리

- 주어진 트리구조에서 2가지 쿼리를 수행
  - 어떤 노드의 하위노드에 전부 값을 업데이트
  - 특정 노드의 값을 출력
- 2회차에서 배운 Euler Tour Technique 적용
  - Dfs를 이용한 넘버링



- 주어진 트리구조에서 2가지 쿼리를 수행
  - 어떤 노드의 하위노드에 전부 값을 업데이트
  - 특정 노드의 값을 출력
- 2회차에서 배운 Euler Tour Technique 적용
  - Dfs를 이용한 넘버링

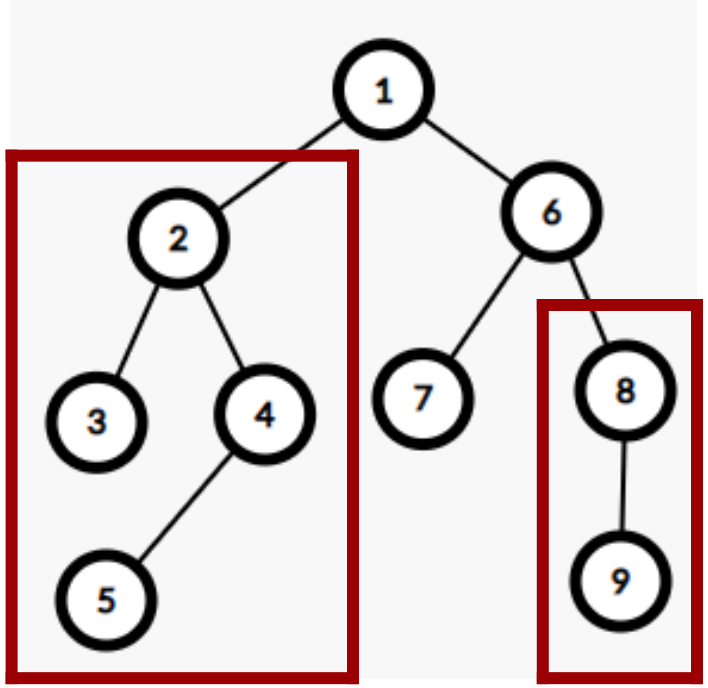
- 2번 노드를 update
  -> [2, 5]번 인덱스노드를 update
- 8번 노드를 update
  -> [8, 9]번 인덱스노드를 update



- 주어진 트리구조에서 2가지 쿼리를 수행
  - 어떤 노드의 하위노드에 전부 값을 업데이트
  - 특정 노드의 값을 출력
- 2회차에서 배운 Euler Tour Technique 적용
  - Dfs를 이용한 넘버링

- 2번 노드를 update
  -> [2, 5]번 인덱스노드를 update
- 8번 노드를 update
  -> [8, 9]번 인덱스노드를 update

  -> 트리를 펼치면 Lazy Propagation 문제가 된다!



마무리

- Segment Tree는 특정 구간에 대한 쿼리를 효율적으로 수행하는 자료구조이다.
  - 구간 대표값 구하기: O(logn)
  - 각 원소에 대한 업데이트: O(logn)
  - 예제
    - 배열의 인덱스 사용하기
    - 원소의 값을 인덱스로 사용하기
    - Plane Sweeping

- Lazy Propagation은 구간에 대한 업데이트도 O(logn)에 수행하는 자료구조이다.
  - 추가 변수
    - 해야 할 업데이트를 저장할 Lazy 배열
    - Lazy값을 뿌려줄 Propagate 함수
  - 각 노드를 만질 때 Propagate 한번씩 해주면 끝
  - 예제
    - On Tree

연습문제

- 2042 구간 합 구하기
- 10999 구간 합 구하기 2
- 11505 구간 곱 구하기
- 10090 Counting Inversions
- 2243 사탕 상자
- 5419 북서풍
- 14267 회사문화 1
- 14268 회사문화 2
- 14287 회사문화 3
- 14288 회사문화 4
- 3133 코끼리

- 스스로 못 풀어내도 괜찮습니다.
- 어떻게든 전부 AC를 받고 이해하는 것이 목표

Q&A