4장

분할 정복 기법

DIVIDE AND CONQUER

큰 문제를 잘게 쪼개어 정복하고, 다시 하나로 합친다

이번 장에서 배울 내용

4.1 분할 정복의 기본

4.2 병합 정렬 (Merge Sort)

4.3 퀵 정렬 (Quick Sort)

4.4 최대 구간 합

4.5 쿼드 트리

4.6 마스터 정리

4.7 코딩 테스트 예제

4.1

분할 정복이란?

"나누어라, 그리고 정복하라"
큰 문제를 작은 문제로 쪼개어 해결하는 알고리즘 설계 전략

나누어라, 그리고 정복하라

큰 문제를 같은 유형의 작은 문제 여러 개로 나누어 각각 해결합니다. 작은 문제의 답을 결합해 원래 문제의 답을 만듭니다.

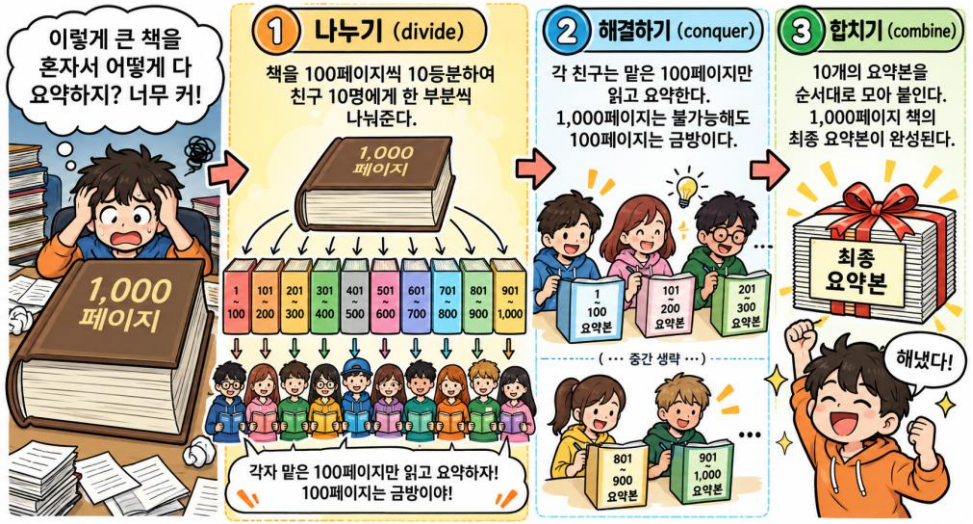
분할, 정복, 결합의 구조를 재귀적으로 적용합니다.

Divide → Conquer → Combine



직관적 이해: 거대한 책 요약하기

1,000페이지짜리 책을 혼자 요약하려면 막막하다. 하지만 친구 10명이 있다면 이야기가 달라진다.



나누기 (divide)

책을 100페이지씩 10등분하여 친구 10명에게 한 부분씩 나눠준다.

해결하기 (conquer)

각 친구는 맡은 100페이지만 읽고 요약한다. 1,000페이지는 불가능해도 100페이지는 금방이다.

합치기 (combine)

10개의 요약본을 순서대로 모아 붙인다. 1,000페이지 책의 최종 요약본이 완성된다.

큰 문제를 그대로 상대하면 어렵지만, 작게 쪼개면 각각은 쉽다. 이것이 분할 정복의 핵심 통찰이다.

분할 정복의 3단계

분할 (Divide)

문제를 같은 유형의 더 작은 하위 문제 여러 개로 나눈다.

정복 (Conquer)

하위 문제를 재귀적으로 해결한다. 충분히 작아지면 직접 푼다.

결합 (Combine)

하위 문제의 해답을 조합해 원래 문제의 해답을 만든다.

분할 정복은 한 번만 나누는 알고리즘이 아니다

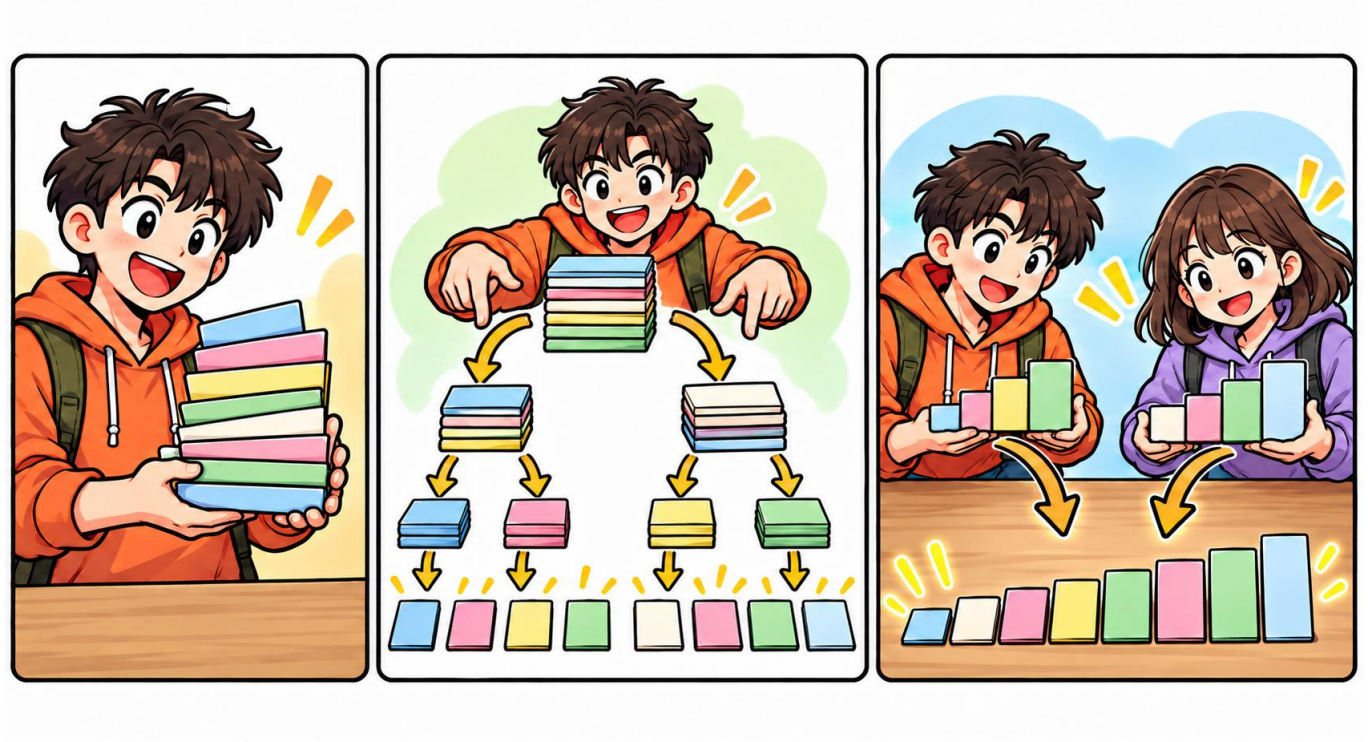
분할된 문제가 아직 커서 즉시 해결할 수 없다면 다시 분할한다. 아주 쉽게 풀릴 때까지 계속 분할한다.

divide_and_conquer.pseudo

```
DIVIDE_AND_CONQUER(P):
    if P의 크기가 충분히 작으면 then return SolveDirectly(P)
    P1, P2, ..., Pk <- Divide(P)                  // 분할
    Si <- DIVIDE_AND_CONQUER(Pi)   for i = 1..k   // 정복
    return Combine(S1, S2, ..., Sk)               // 결합
```

병합 정렬: 한 장까지 나누고 합치기

입력: [8, 3, 6, 2, 7, 1, 5, 4] → 출력: [1, 2, 3, 4, 5, 6, 7, 8]
작은 문제의 정렬 결과를 받아, 두 줄의 맨 앞에서 작은 값을 골라 붙입니다.



한 장은 이미 정렬된 상태입니다. 정렬된 두 결과를 병합해 더 큰 결과를 만듭니다.

병합 정렬 (Merge Sort)

반으로 나누고, 정렬된 두 배열을 합친다
원소가 하나이면 이미 정렬된 상태다.

정렬 전

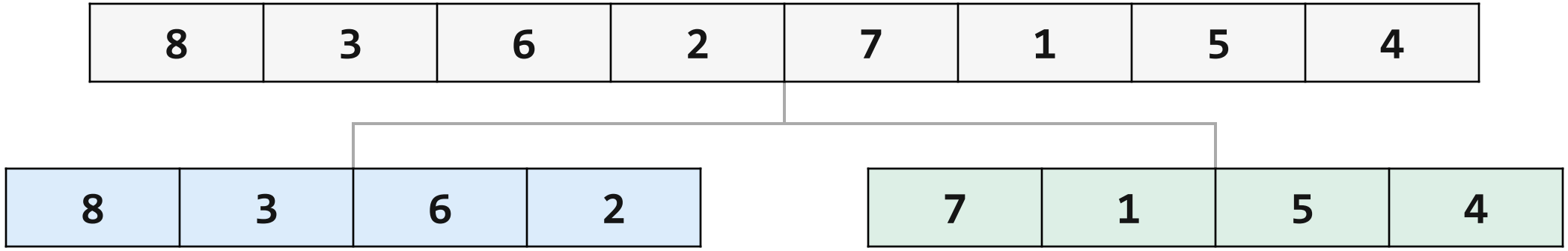
8	3	6	2	7	1	5	4

정렬 후

1	2	3	4	5	6	7	8

나누기 ① · 8개를 4개씩

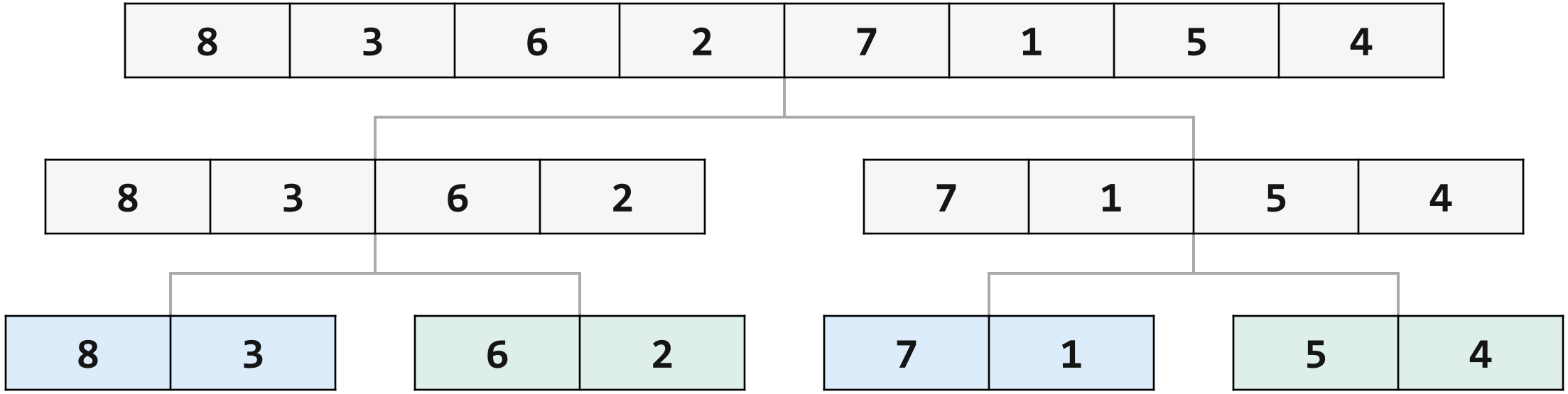
가운데 위치에서 나눕니다. 숫자의 크기는 아직 비교하지 않습니다.
왼쪽 [8,3,6,2], 오른쪽 [7,1,5,4]의 원래 순서를 유지합니다.



다음: 네 개짜리 구간도 더 작은 문제로 나눕니다.

나누기 ② · 4개를 2개씩

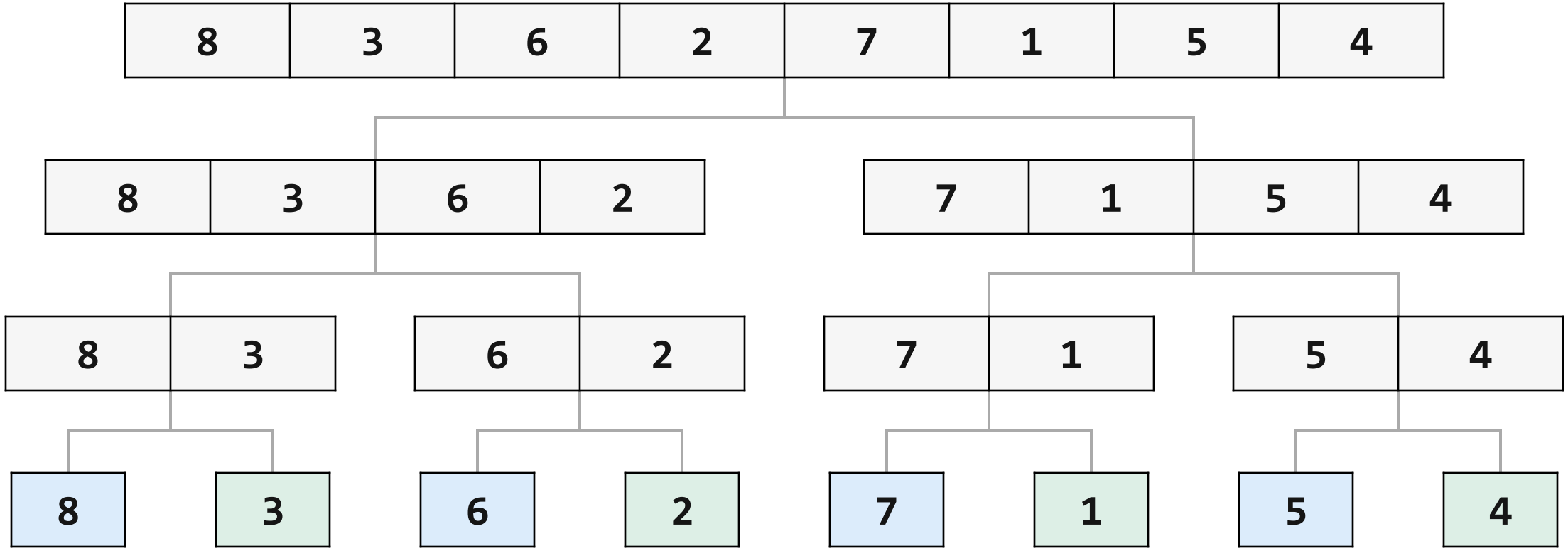
네 개짜리 구간도 각각 가운데에서 나눕니다.
두 개짜리 네 묶음이 생겼지만, 아직 정렬된 결과는 아닙니다.



다음: 두 개짜리 구간을 원소 하나씩 나눕니다.

나누기 ③ · 2개를 1개씩

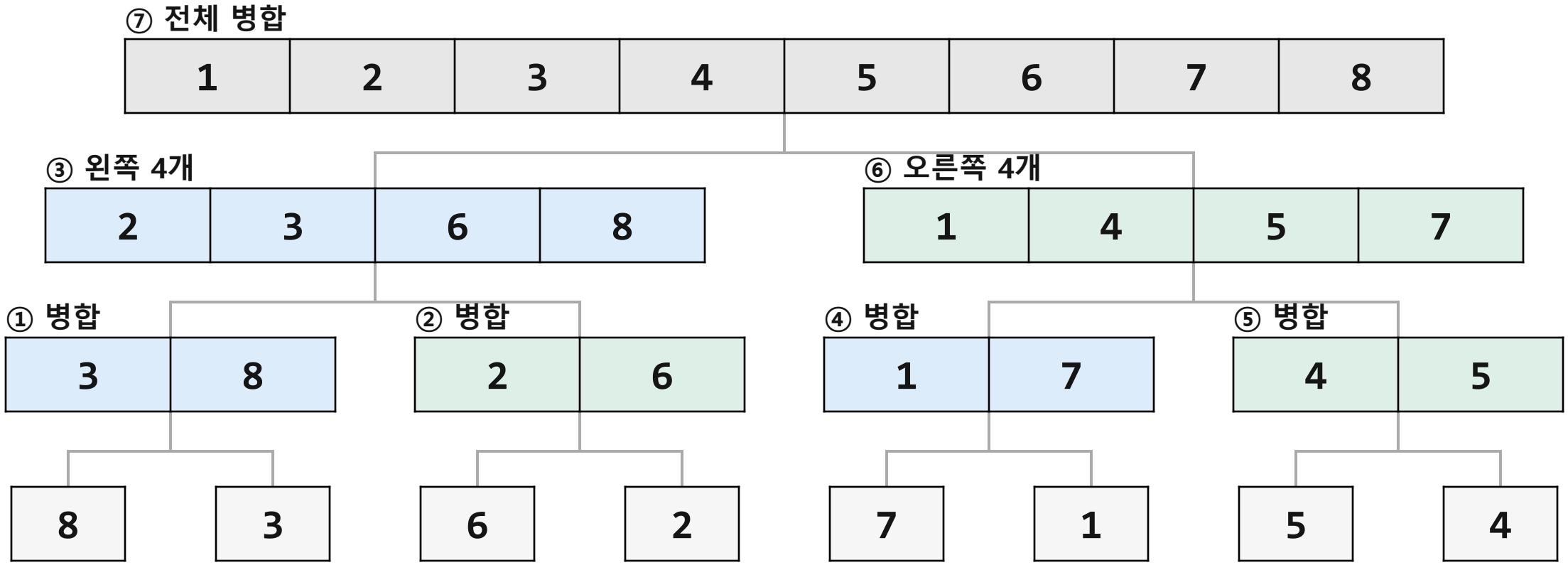
원소가 하나이면 더 정렬할 것이 없습니다. 여기서 재귀 호출을 멈춥니다.
이제 한 원소짜리 결과 두 개를 받아 병합을 시작합니다.



한 원소 = 이미 정렬됨 → 반환 → 두 결과를 병합

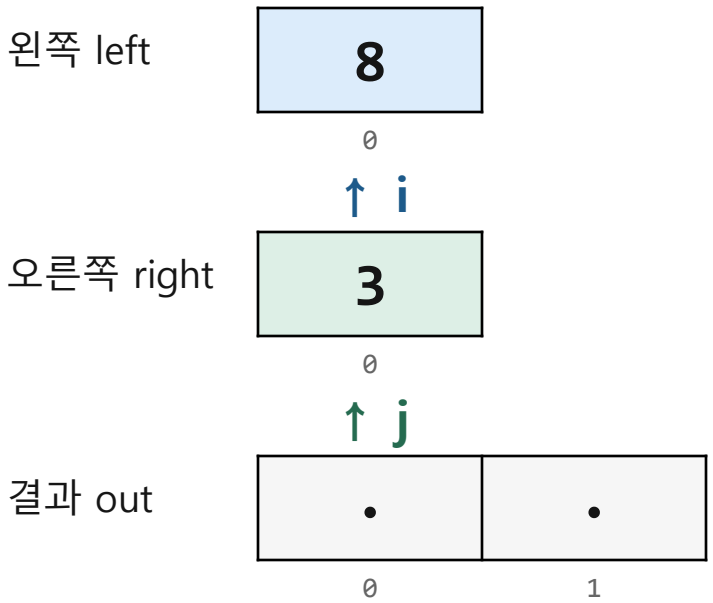
실제 병합 순서 · 왼쪽부터 끝낸다

① → ② → ③: 왼쪽 4개 완료 ④ → ⑤ → ⑥: 오른쪽 4개 완료
⑦: 두 결과를 합쳐 전체 8개 정렬

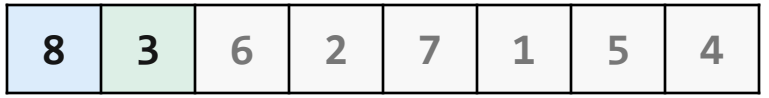


① [8]과 [3] 병합 · 시작

i = 0, j = 0. 왼쪽 8과 오른쪽 3을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 앞쪽 두 개

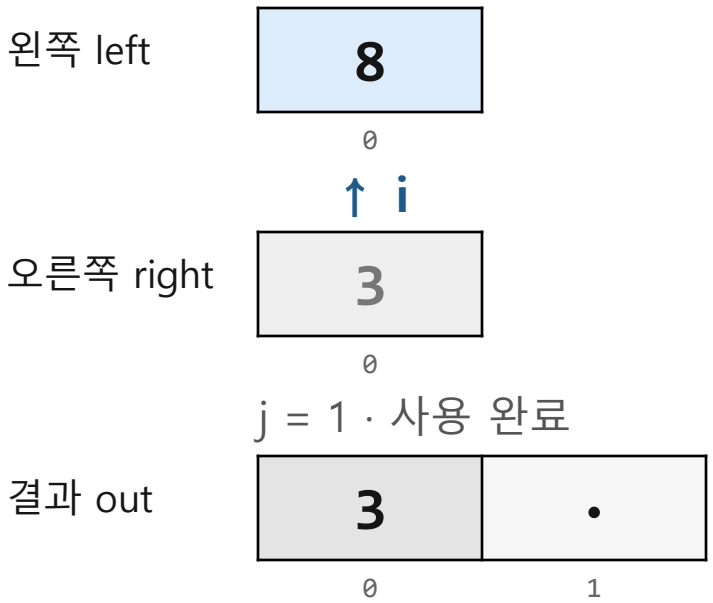
한 원소짜리 두 결과를 받습니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

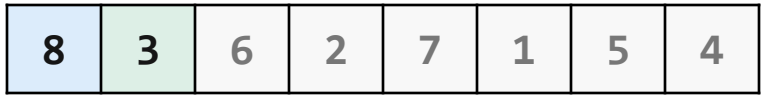
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

① 병합 · 1번째 값 3 추가

왼쪽 8 > 오른쪽 3 → 오른쪽의 3을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 앞쪽 두 개

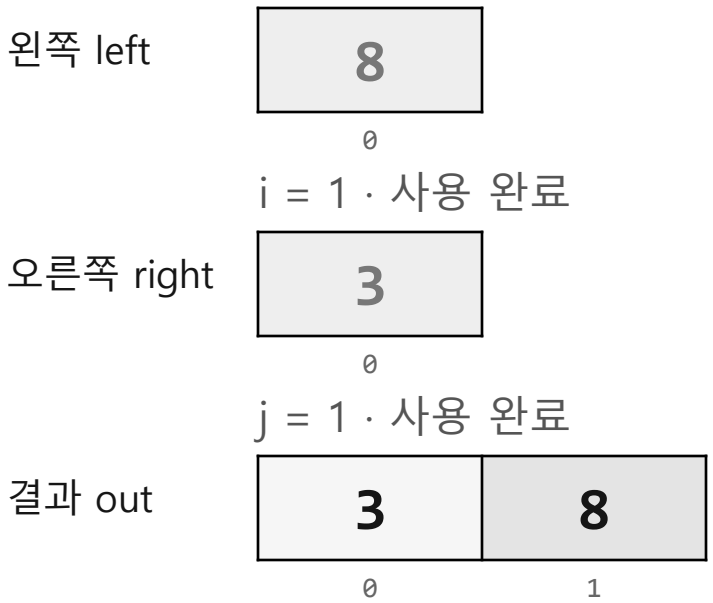
한 원소짜리 두 결과를 받습니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

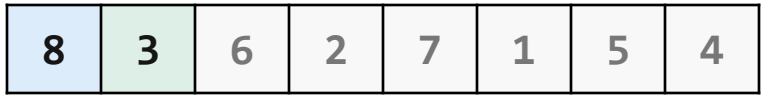
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

① 병합 · 2번째 값 8 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 8을 붙입니다.
i는 0 → 1. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 앞쪽 두 개

이번 병합의 정렬 결과

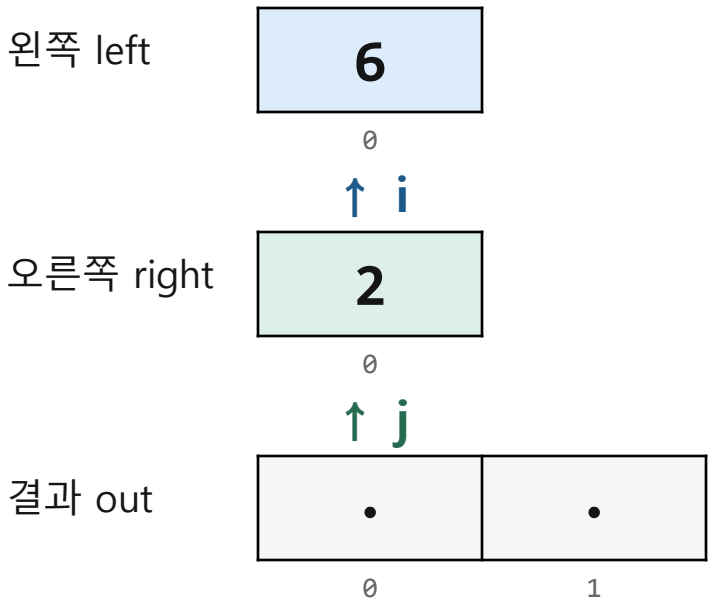
[3, 8]

③에서 왼쪽 입력으로 사용

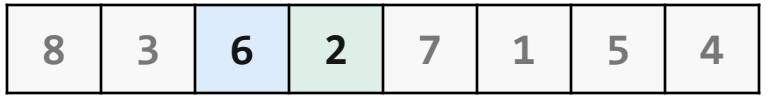
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

② [6]과 [2] 병합 · 시작

i = 0, j = 0. 왼쪽 6과 오른쪽 2을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 뒤쪽 두 개

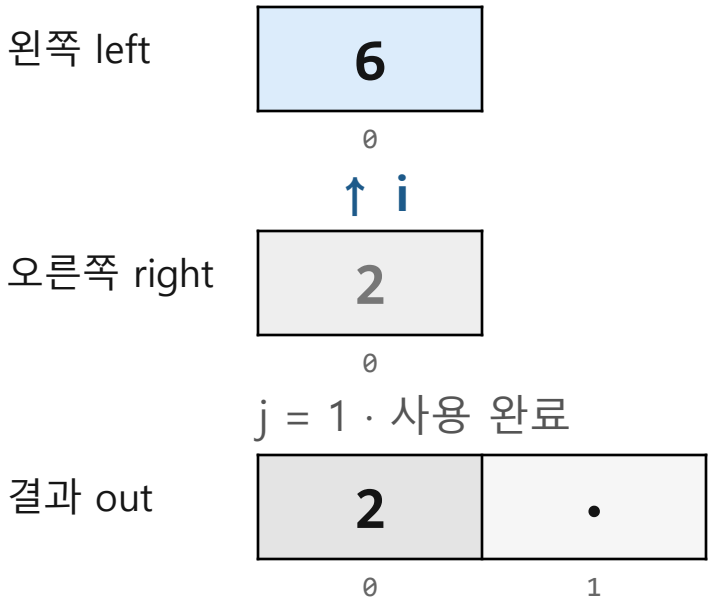
①의 [3,8]은 준비된 채 기다립니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

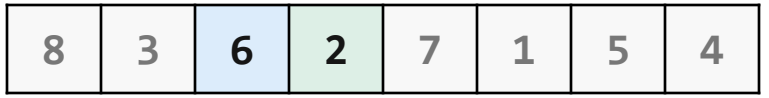
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

② 병합 · 1번째 값 2 추가

왼쪽 6 > 오른쪽 2 → 오른쪽의 2을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 뒤쪽 두 개

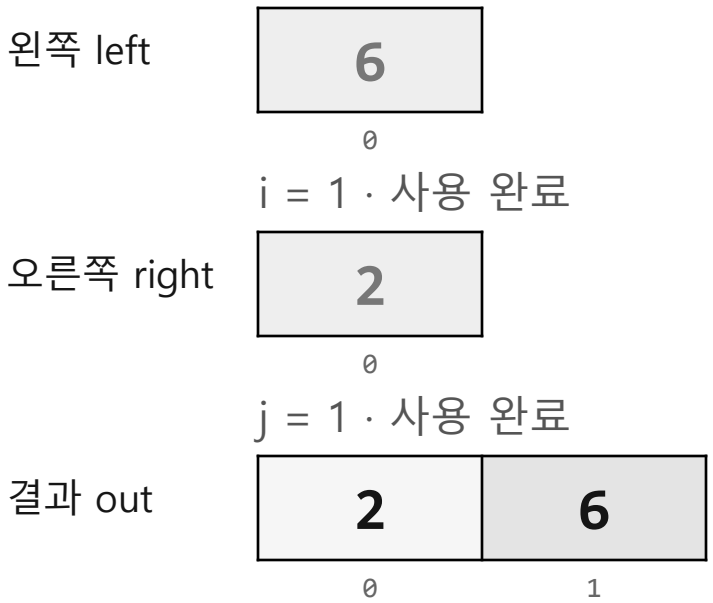
①의 [3,8]은 준비된 채 기다립니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

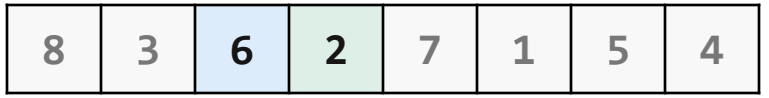
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

② 병합 · 2번째 값 6 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 6을 붙입니다.
i는 0 → 1. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개의 뒤쪽 두 개

이번 병합의 정렬 결과

[2, 6]

③에서 오른쪽 입력으로 사용

파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

병합의 비교: 고른 쪽의 포인터만 이동

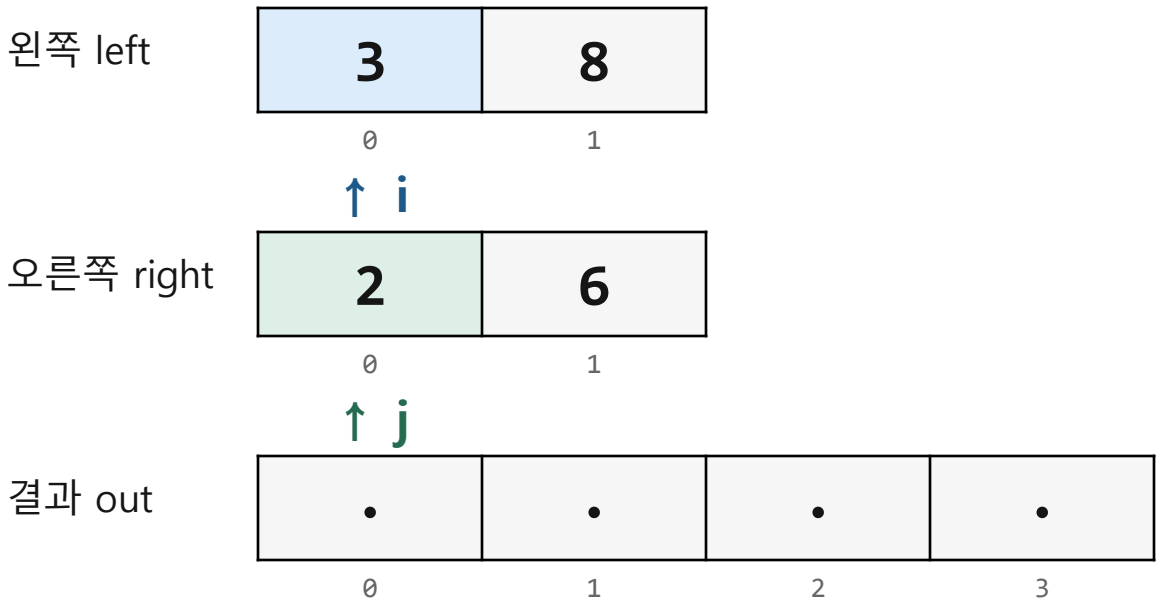
왼쪽 [3, 8]과 오른쪽 [2, 6]은 이미 정렬되어 있습니다. 맨 앞의 3과 2를 비교합니다.



2를 꺼내면 j만 앞으로 이동합니다. i는 아직 사용하지 않은 3을 계속 가리킵니다.

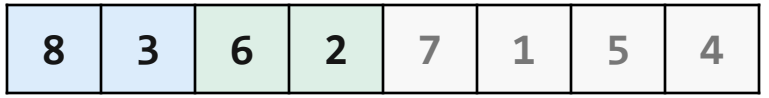
③ [3, 8]과 [2, 6] 병합 · 시작

i = 0, j = 0. 왼쪽 3과 오른쪽 2을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



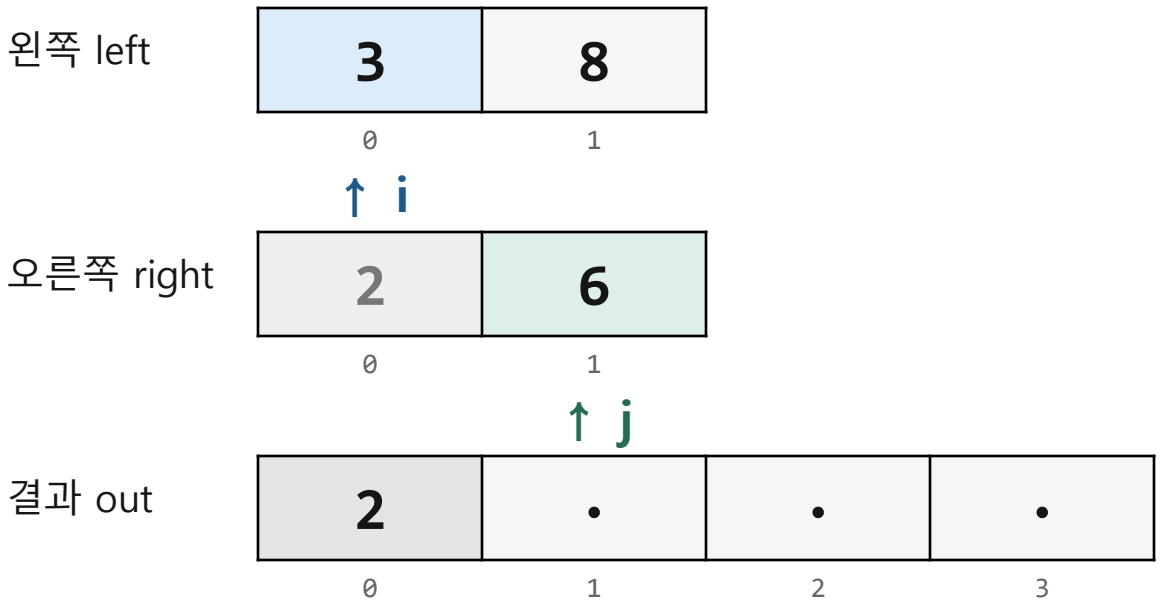
왼쪽 4개 전체

왼쪽: ①의 결과 오른쪽: ②의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

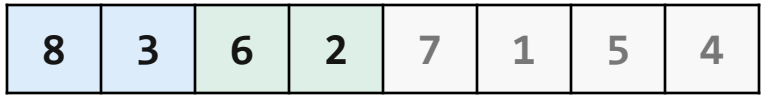
③ 병합 · 1번째 값 2 추가

왼쪽 3 > 오른쪽 2 → 오른쪽의 2을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



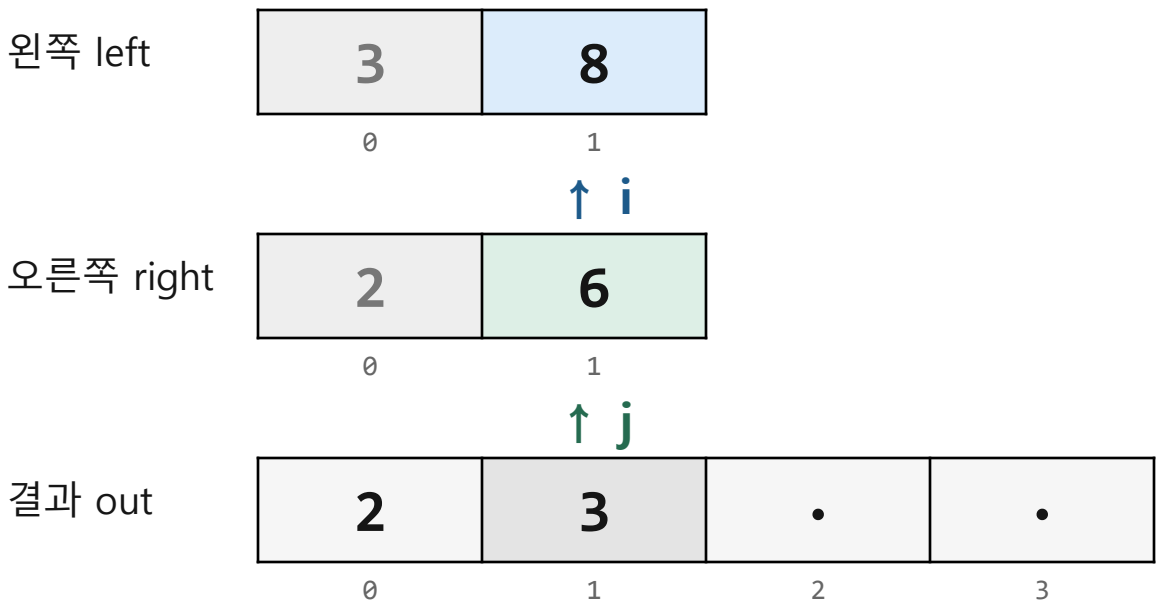
왼쪽 4개 전체

왼쪽: ①의 결과 오른쪽: ②의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

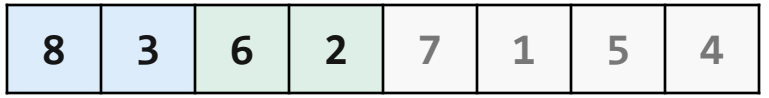
③ 병합 · 2번째 값 3 추가

왼쪽 3 ≤ 오른쪽 6 → 왼쪽의 3을 붙입니다.
i는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



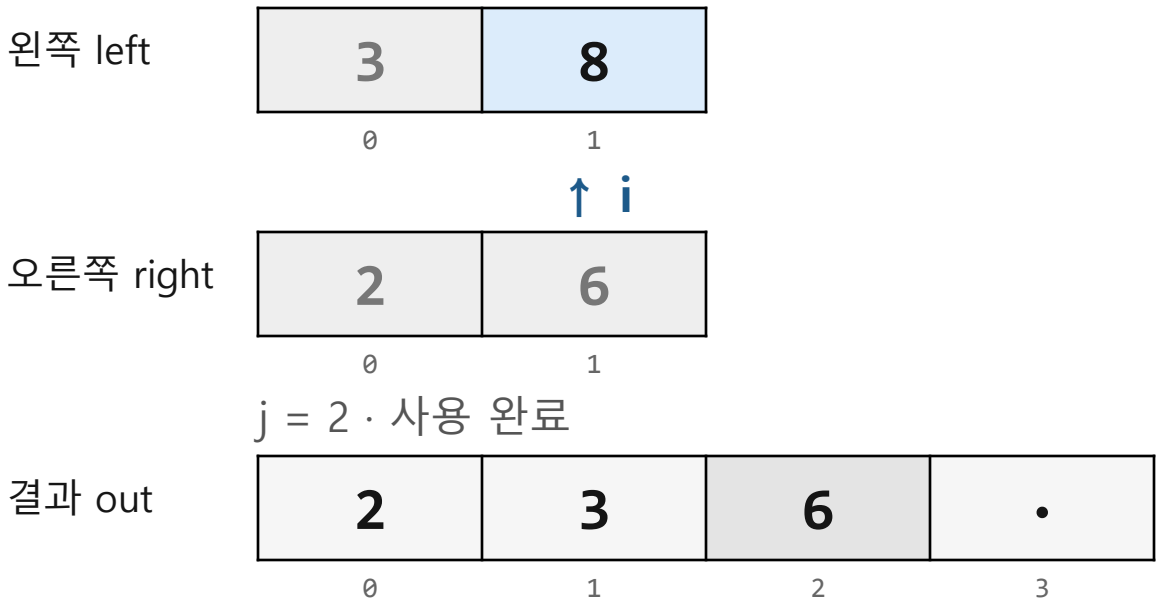
왼쪽 4개 전체

왼쪽: ①의 결과 오른쪽: ②의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

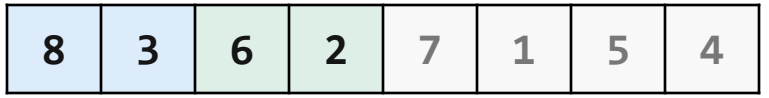
③ 병합 · 3번째 값 6 추가

왼쪽 8 > 오른쪽 6 → 오른쪽의 6을 붙입니다.
j는 1 → 2. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



왼쪽 4개 전체

왼쪽: ①의 결과 오른쪽: ②의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

한쪽을 다 쓰면 남은 값을 붙입니다

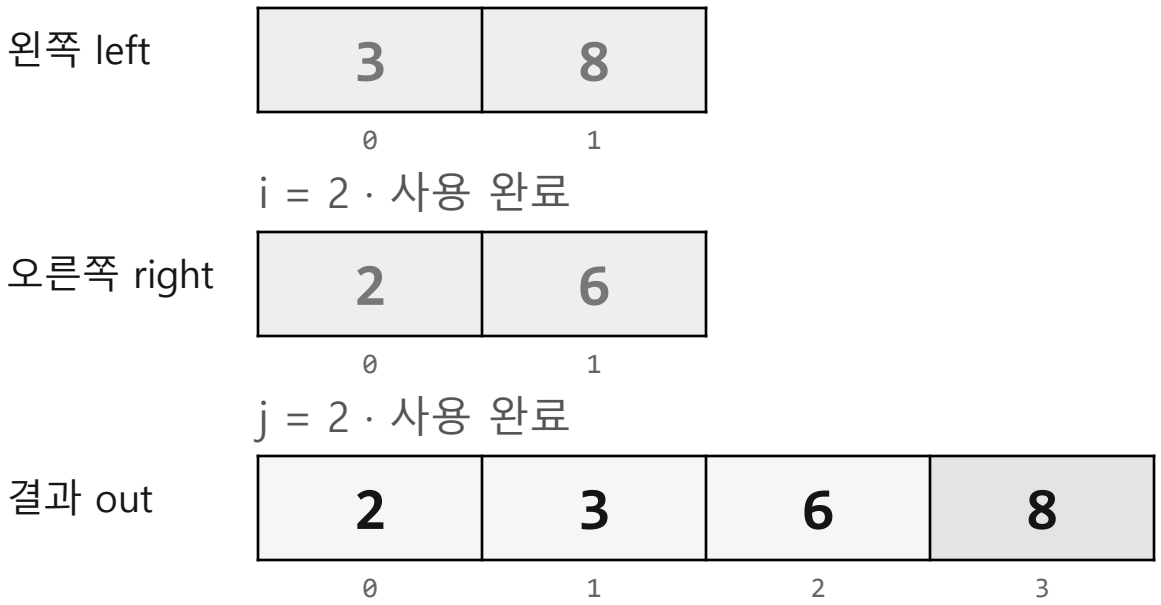
현재 결과는 [2, 3, 6]입니다. 오른쪽은 비었고, 왼쪽에는 8만 남았습니다.



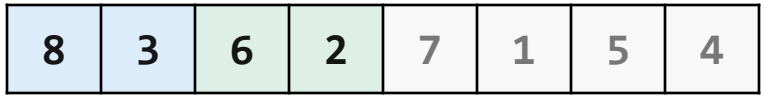
비어 있는 쪽에서는 값을 읽지 않습니다. 남은 쪽의 값을 순서대로 결과에 붙입니다.

③ 병합 · 4번째 값 8 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 8을 붙입니다.
i는 1 → 2. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



왼쪽 4개 전체

이번 병합의 정렬 결과

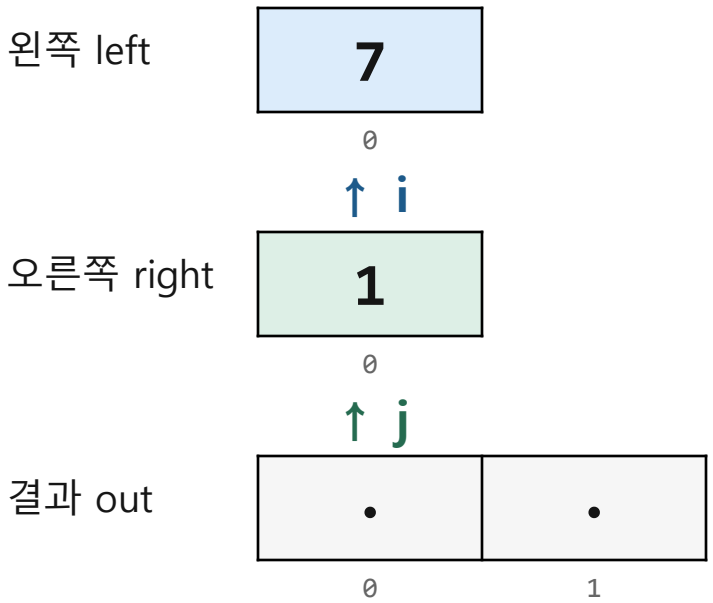
[2, 3, 6, 8]

왼쪽 4개 완료 → 오른쪽으로

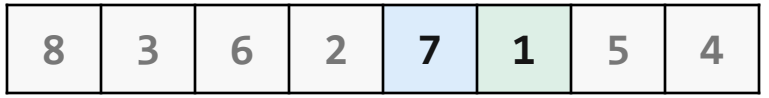
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

④ [7]과 [1] 병합 · 시작

i = 0, j = 0. 왼쪽 7과 오른쪽 1을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 앞쪽 두 개

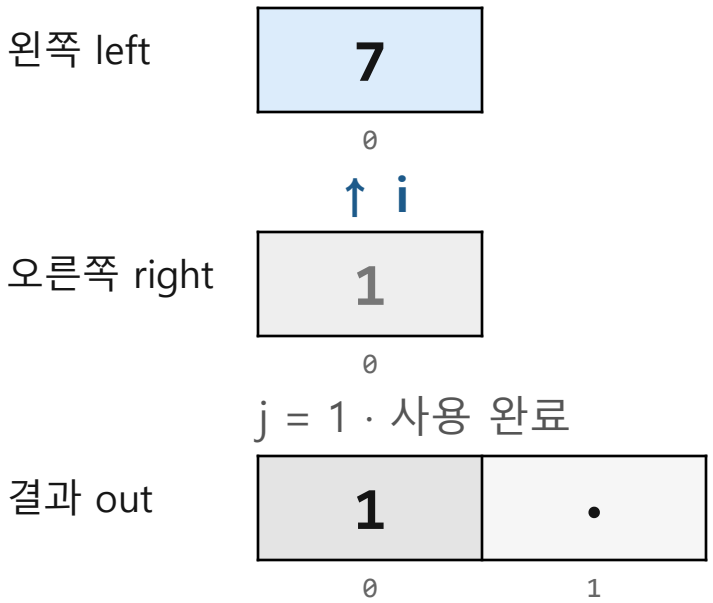
왼쪽 4개 [2,3,6,8]은 정렬 완료

선택한 쪽의 인덱스만
한 칸 움직입니다.

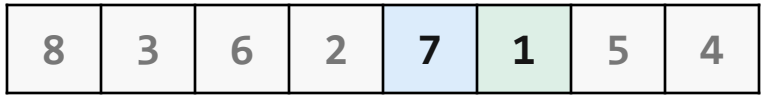
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

④ 병합 · 1번째 값 1 추가

왼쪽 7 > 오른쪽 1 → 오른쪽의 1을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 앞쪽 두 개

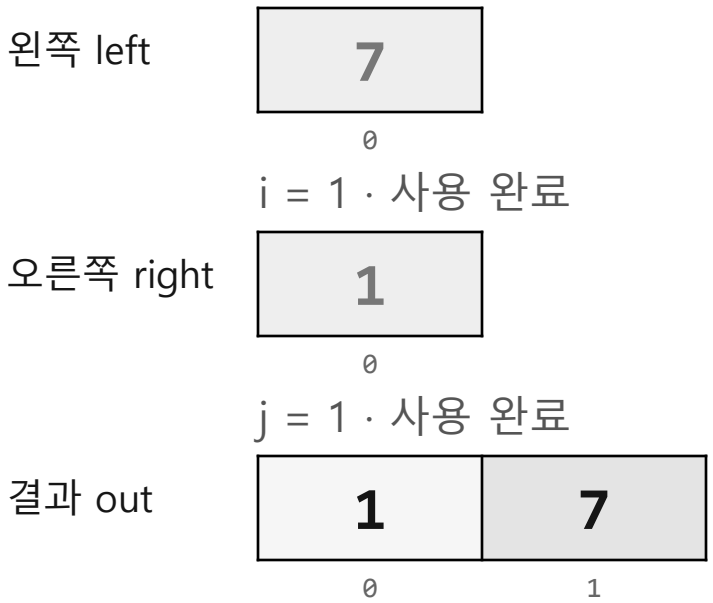
왼쪽 4개 [2,3,6,8]은 정렬 완료

선택한 쪽의 인덱스만
한 칸 움직입니다.

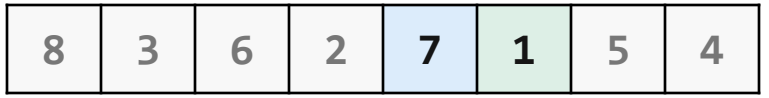
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

④ 병합 · 2번째 값 7 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 7을 붙입니다.
i는 0 → 1. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 앞쪽 두 개

이번 병합의 정렬 결과

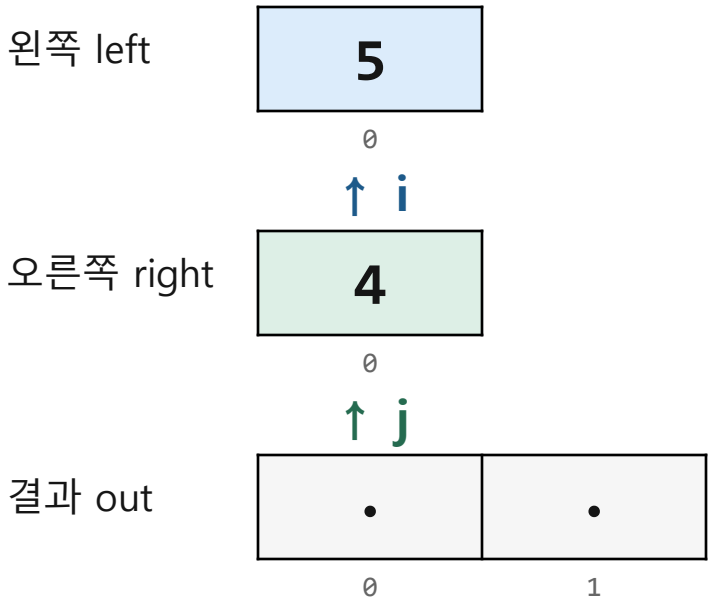
[1, 7]

⑥에서 왼쪽 입력으로 사용

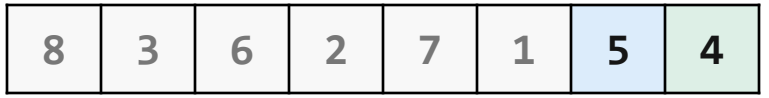
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

⑤ [5]과 [4] 병합 · 시작

i = 0, j = 0. 왼쪽 5과 오른쪽 4을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 뒤쪽 두 개

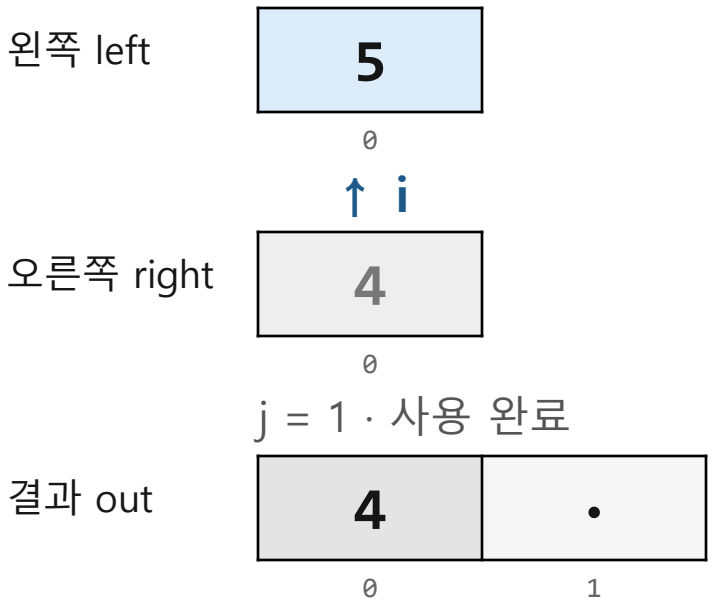
④의 [1,7]은 준비된 채 기다립니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

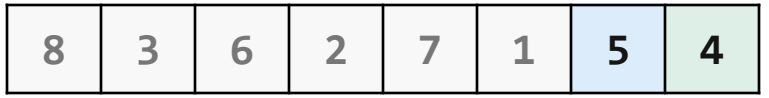
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

⑤ 병합 · 1번째 값 4 추가

왼쪽 5 > 오른쪽 4 → 오른쪽의 4을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 뒤쪽 두 개

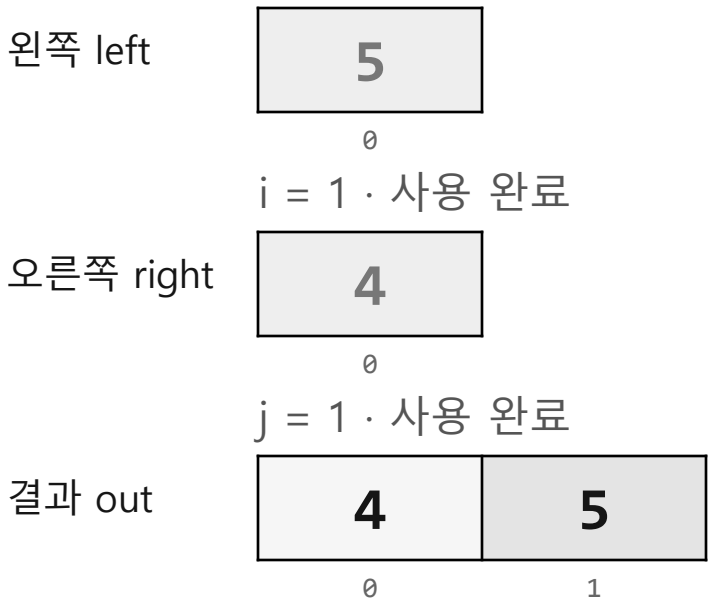
④의 [1,7]은 준비된 채 기다립니다.

선택한 쪽의 인덱스만
한 칸 움직입니다.

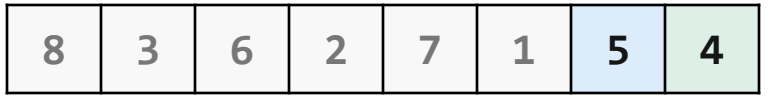
파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

⑤ 병합 · 2번째 값 5 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 5을 붙입니다.
i는 0 → 1. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개의 뒤쪽 두 개

이번 병합의 정렬 결과

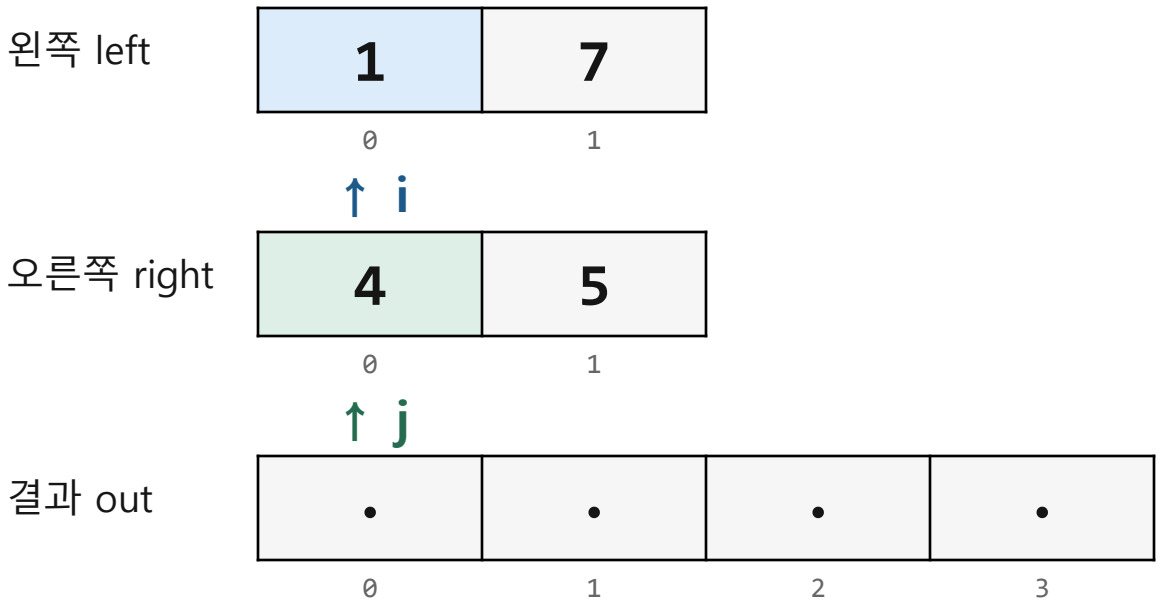
[4, 5]

⑥에서 오른쪽 입력으로 사용

파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

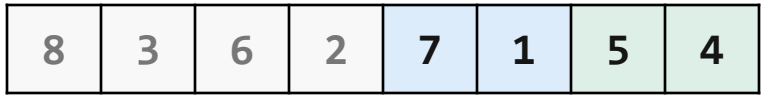
⑥ [1, 7]과 [4, 5] 병합 · 시작

i = 0, j = 0. 왼쪽 1과 오른쪽 4을 비교합니다.
정렬된 두 결과를 받아 작은 값부터 새 결과에 넣습니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



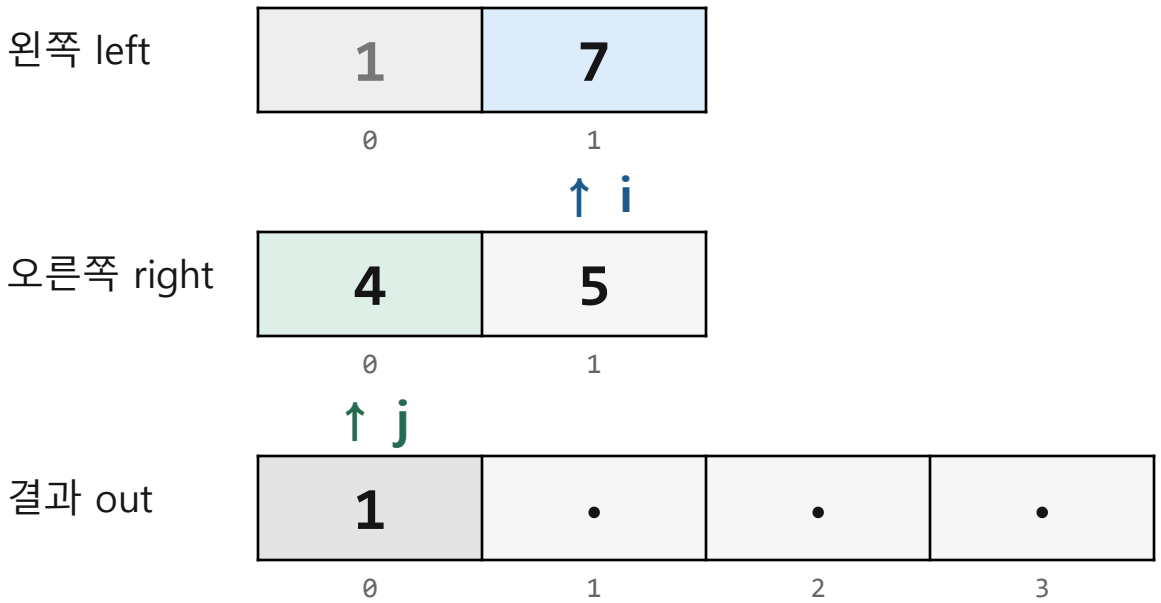
오른쪽 4개 전체

왼쪽: ④의 결과 오른쪽: ⑤의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

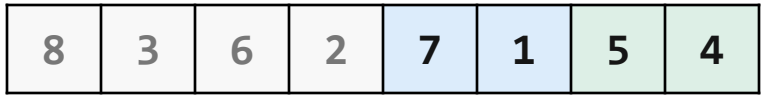
⑥ 병합 · 1번째 값 1 추가

왼쪽 1 ≤ 오른쪽 4 → 왼쪽의 1을 붙입니다.
i는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



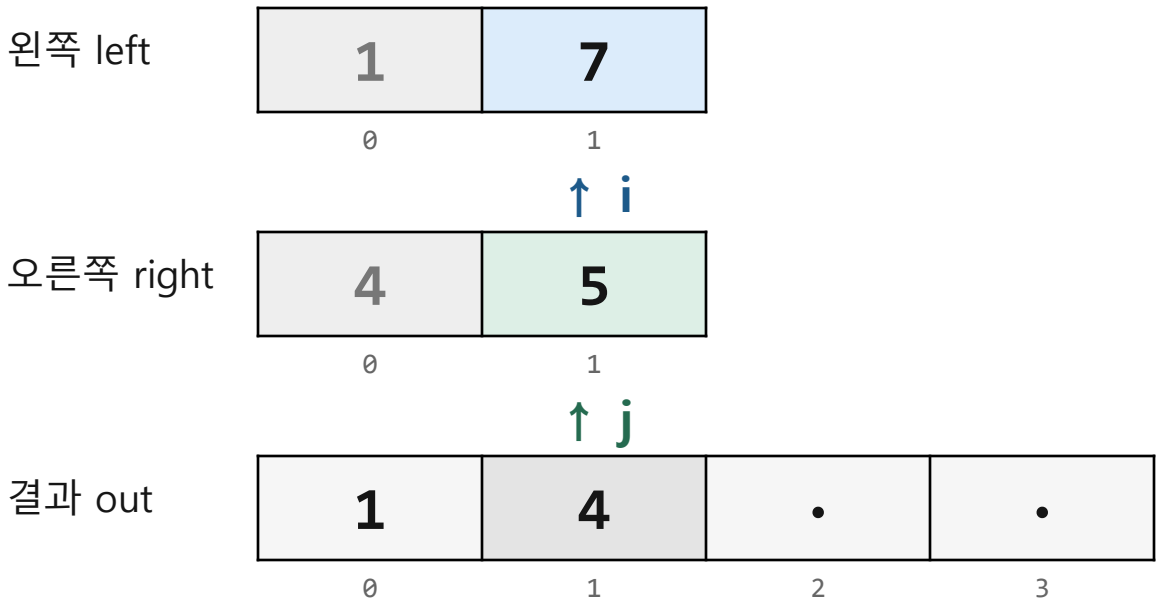
오른쪽 4개 전체

왼쪽: ④의 결과 오른쪽: ⑤의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

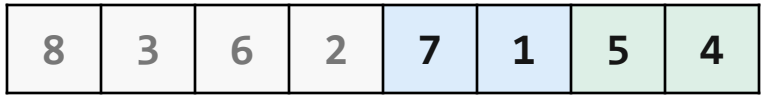
⑥ 병합 · 2번째 값 4 추가

왼쪽 7 > 오른쪽 4 → 오른쪽의 4을 붙입니다.
j는 0 → 1. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



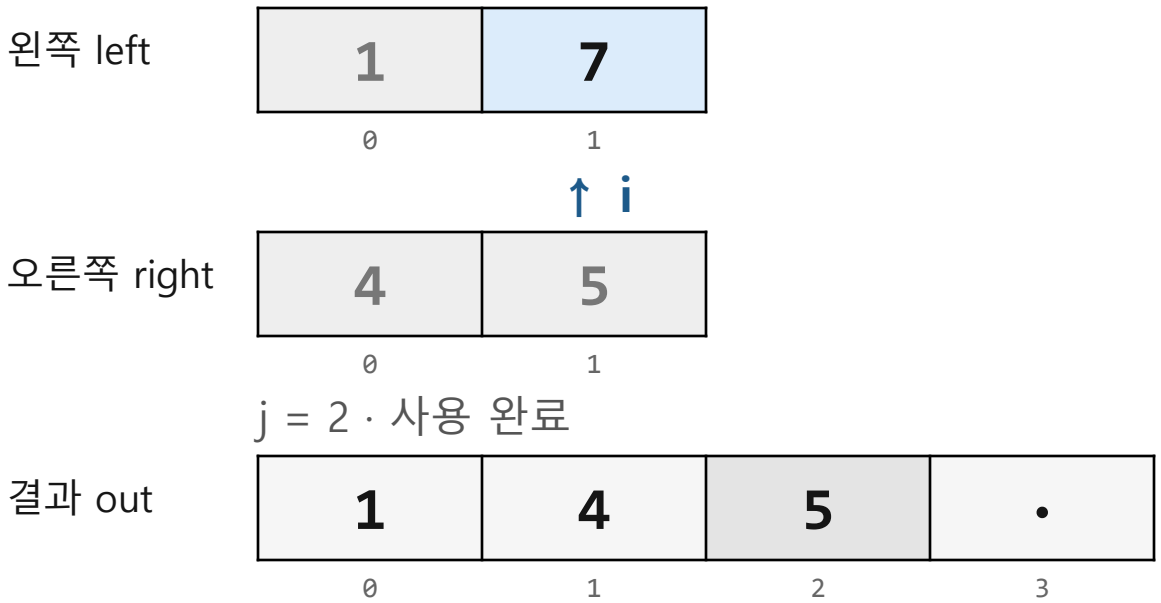
오른쪽 4개 전체

왼쪽: ④의 결과 오른쪽: ⑤의 결과

선택한 쪽의 인덱스만
한 칸 움직입니다.

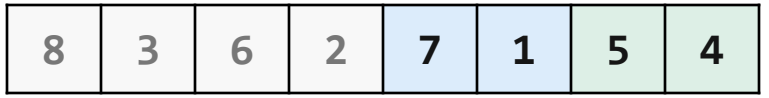
⑥ 병합 · 3번째 값 5 추가

왼쪽 7 > 오른쪽 5 → 오른쪽의 5을 붙입니다.
j는 1 → 2. 사용하지 않은 쪽의 인덱스는 그대로입니다.



파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

처음 배열에서 이번에 합치는 부분



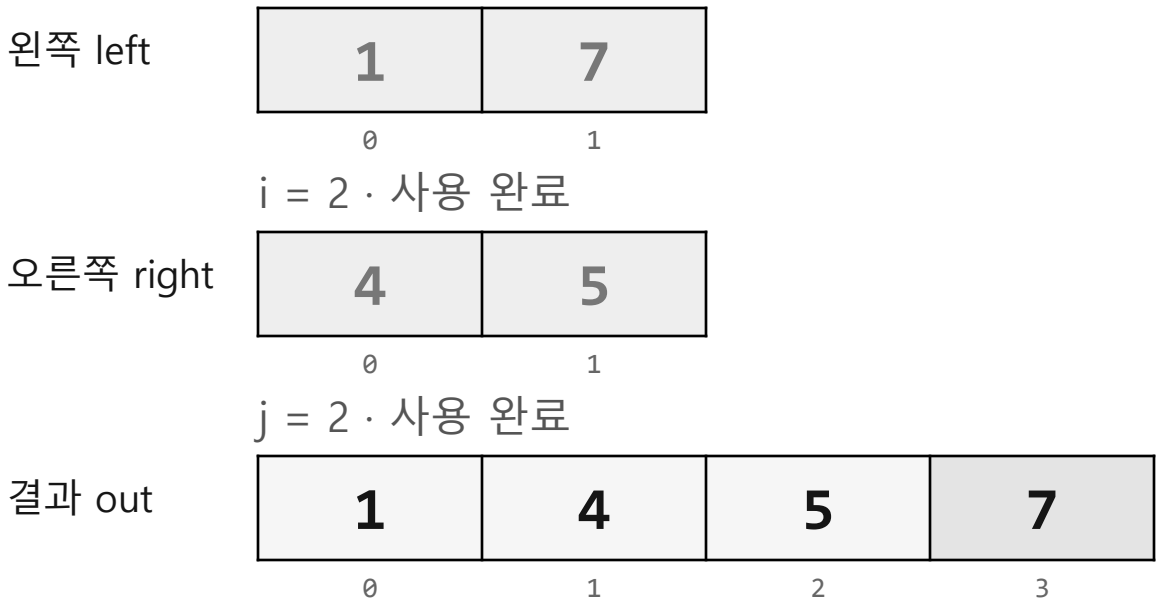
오른쪽 4개 전체

왼쪽: ④의 결과 오른쪽: ⑤의 결과

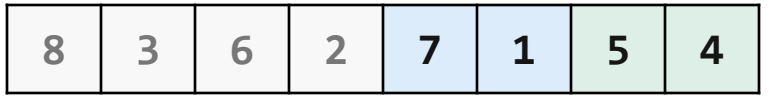
선택한 쪽의 인덱스만
한 칸 움직입니다.

⑥ 병합 · 4번째 값 7 추가

오른쪽을 다 사용했습니다. 남은 왼쪽의 7을 붙입니다.
i는 1 → 2. 이번 병합의 정렬 결과가 완성되었습니다.



처음 배열에서 이번에 합치는 부분



오른쪽 4개 전체

이번 병합의 정렬 결과

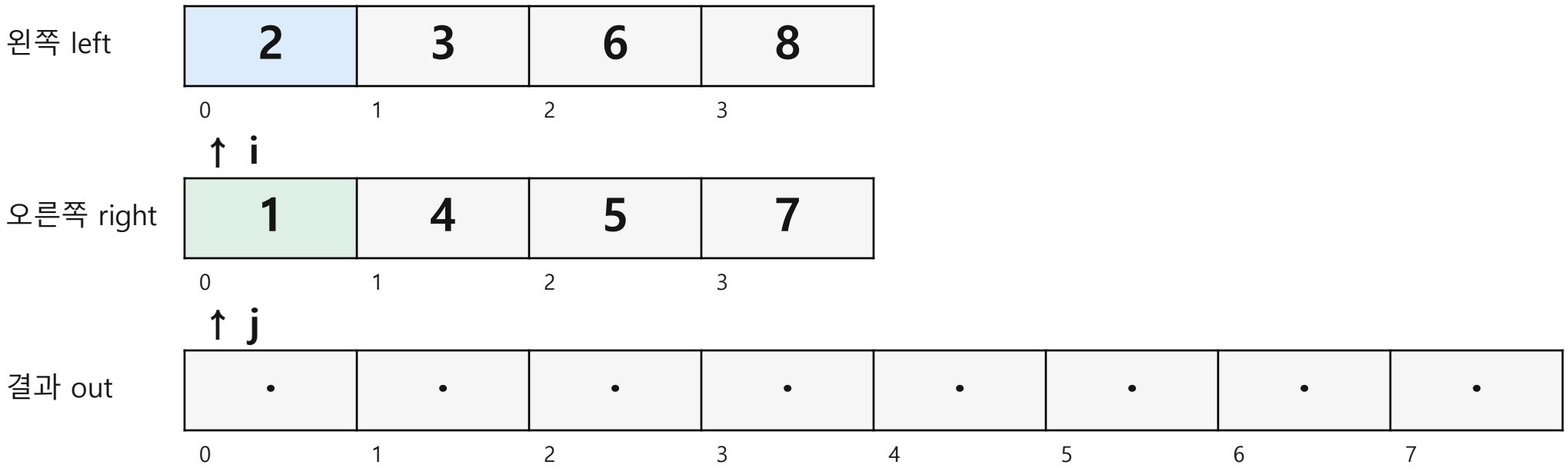
[1, 4, 5, 7]

⑦에서 오른쪽 입력으로 사용

파랑 i: 왼쪽의 다음 값 초록 j: 오른쪽의 다음 값 회색: 사용한 값

병합 시작 · 두 화살표가 첫 칸

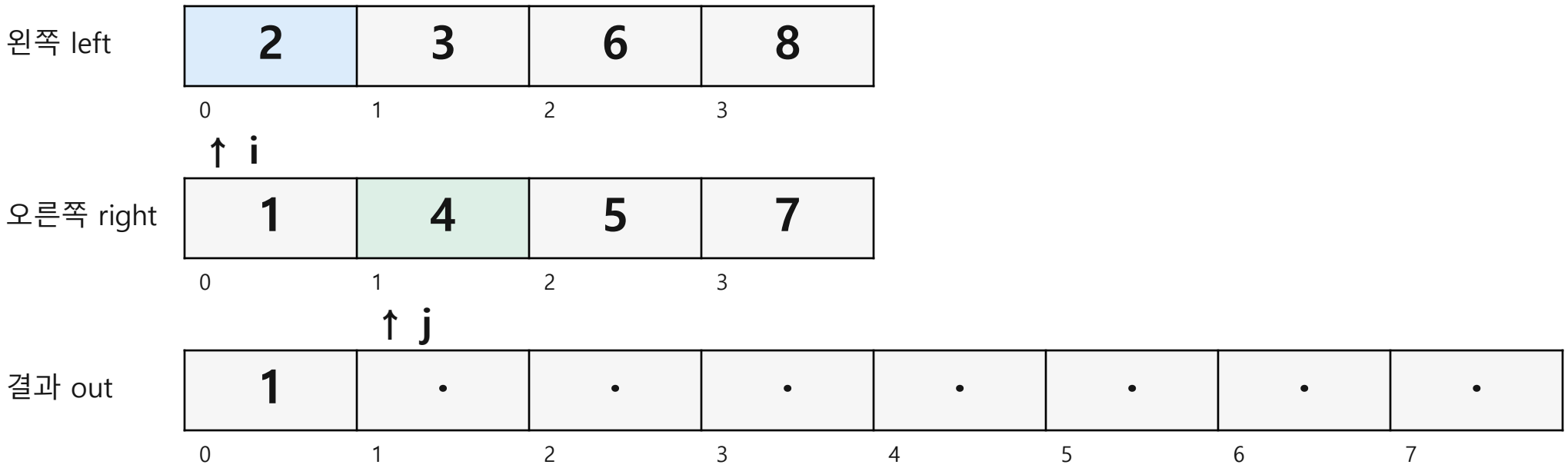
i = 0, j = 0입니다. 왼쪽 2와 오른쪽 1을 비교합니다.
결과 배열은 아직 비어 있습니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 1단계 · 1을 붙인 뒤

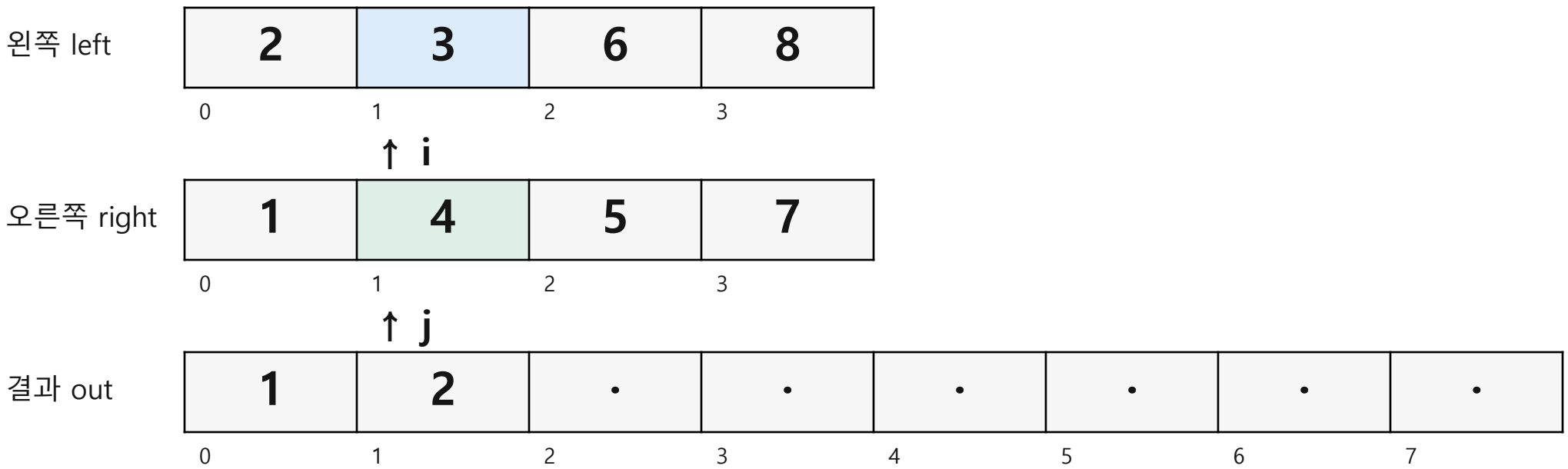
왼쪽 2 > 오른쪽 1. 오른쪽 값을 결과 끝에 붙입니다.
j를 0에서 1로 옮깁니다. i = 0는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 2단계 · 2을 붙인 뒤

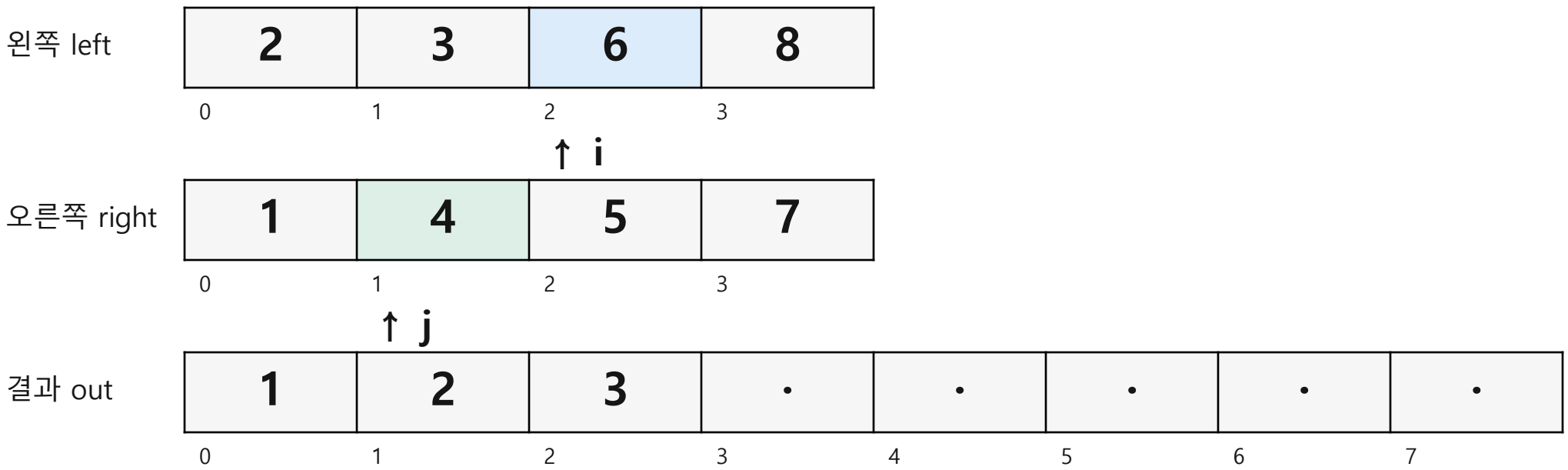
왼쪽 2 ≤ 오른쪽 4. 왼쪽 값을 결과 끝에 붙입니다.
i를 0에서 1로 옮깁니다. j = 1는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 3단계 · 3을 붙인 뒤

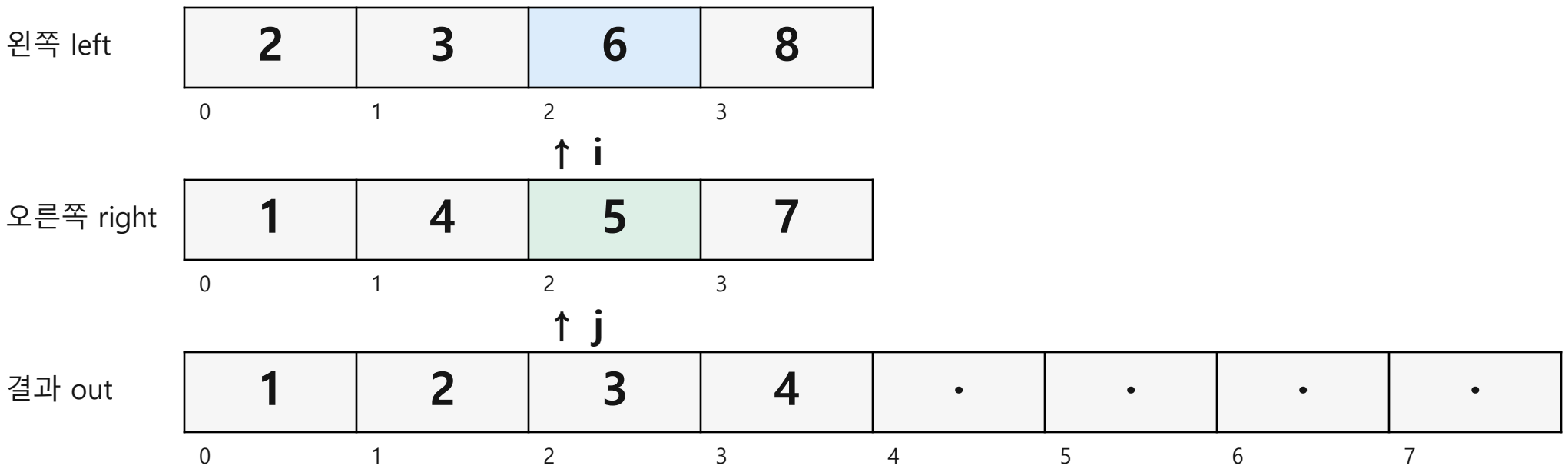
왼쪽 3 ≤ 오른쪽 4. 왼쪽 값을 결과 끝에 붙입니다.
i를 1에서 2로 옮깁니다. j = 1는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 4단계 · 4을 붙인 뒤

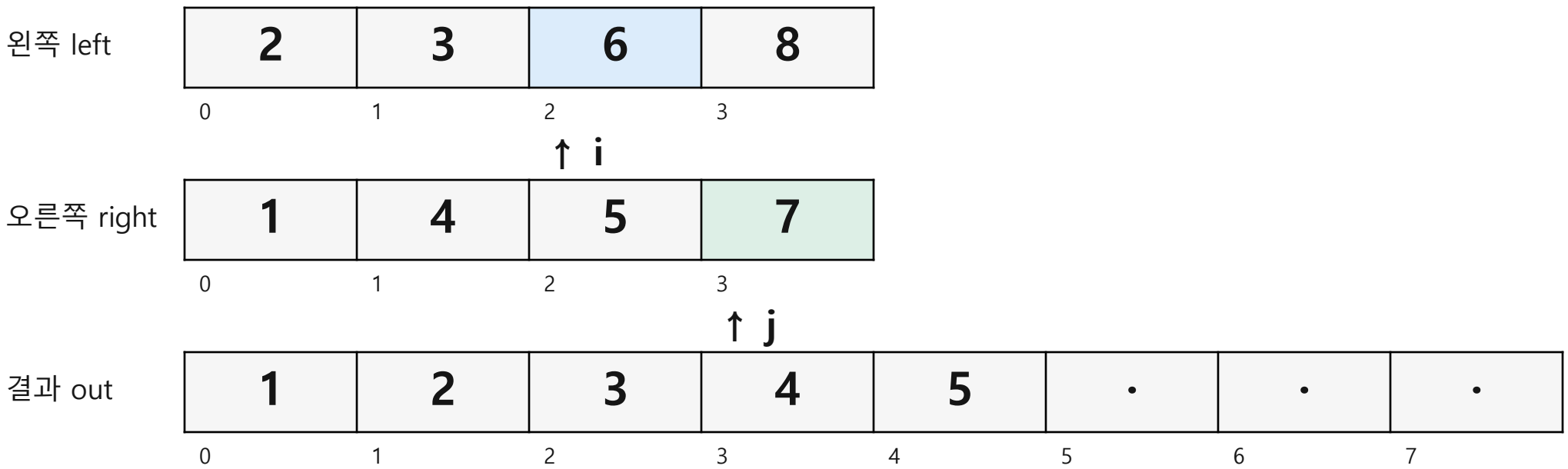
왼쪽 6 > 오른쪽 4. 오른쪽 값을 결과 끝에 붙입니다.
j를 1에서 2로 옮깁니다. i = 2는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 5단계 · 5을 붙인 뒤

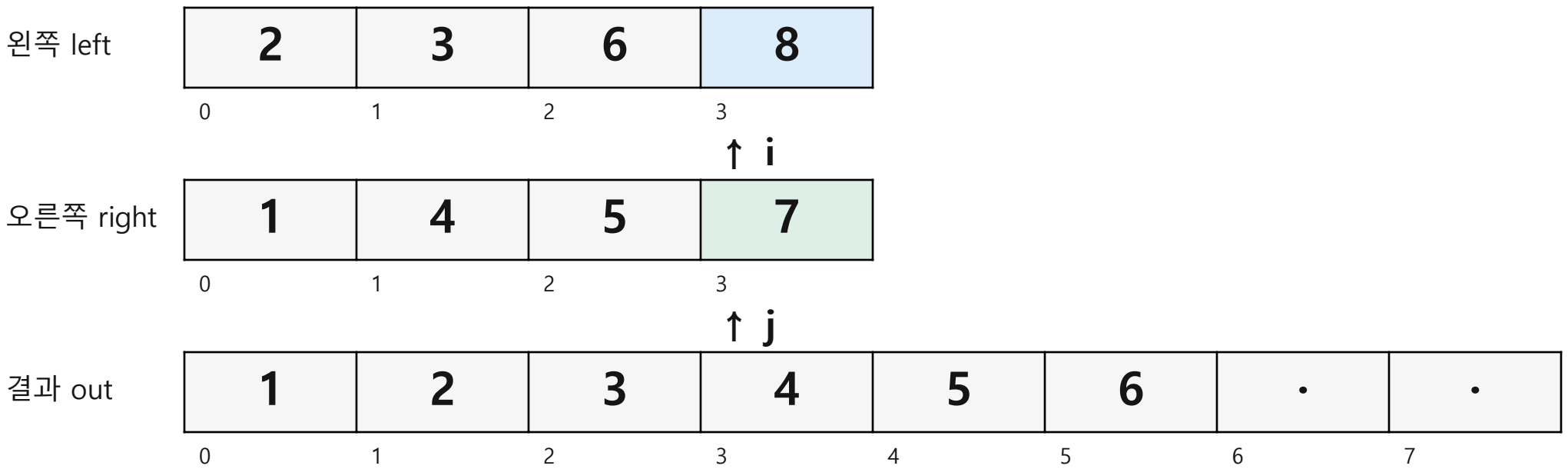
왼쪽 6 > 오른쪽 5. 오른쪽 값을 결과 끝에 붙입니다.
j를 2에서 3로 옮깁니다. i = 2는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 6단계 · 6을 붙인 뒤

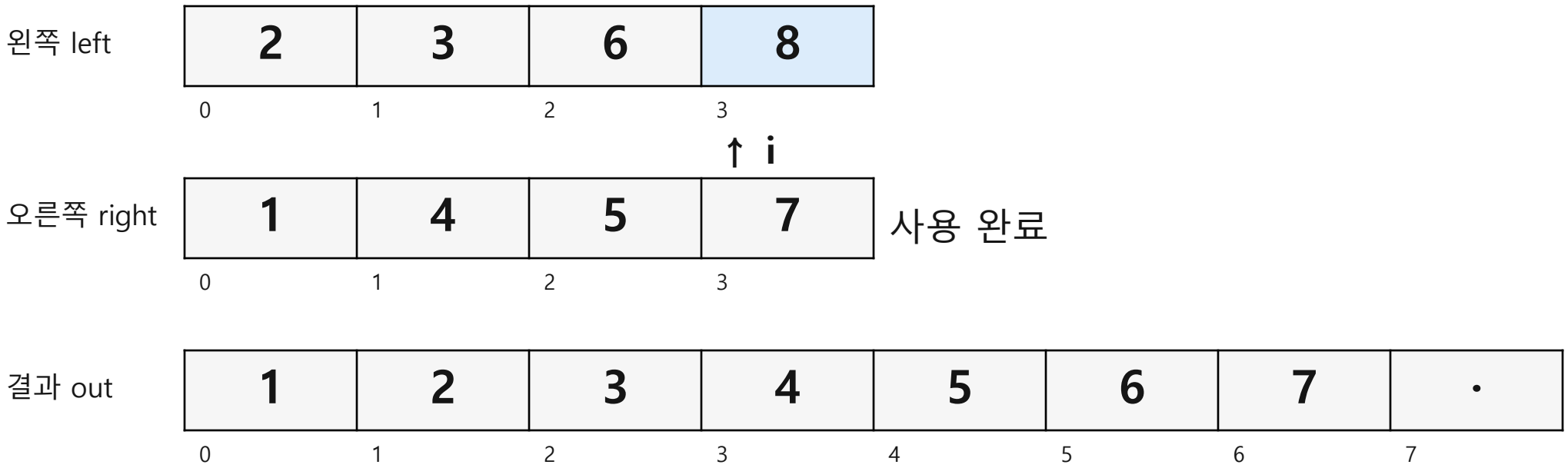
왼쪽 6 ≤ 오른쪽 7. 왼쪽 값을 결과 끝에 붙입니다.
i를 2에서 3로 옮깁니다. j = 3는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 7단계 · 7을 붙인 뒤

왼쪽 8 > 오른쪽 7. 오른쪽 값을 결과 끝에 붙입니다.
j를 3에서 4로 옮깁니다. i = 3는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

병합 8단계 · 8을 붙인 뒤

오른쪽을 다 사용했습니다. 왼쪽의 8을 결과 끝에 붙입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

Python: 병합 함수

```
def merge(left, right):
    out = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            out.append(left[i])
            i += 1
        else:
            out.append(right[j])
            j += 1
    while i < len(left):
        out.append(left[i])
        i += 1
    while j < len(right):
        out.append(right[j])
        j += 1
    return out
```

Python: 재귀 호출 순서

```
def merge_sort(a):
    if len(a) <= 1:
        return a[:]
    mid = len(a) // 2
    left = merge_sort(a[:mid])
    right = merge_sort(a[mid:])
    return merge(left, right)

a = [8, 3, 6, 2, 7, 1, 5, 4]
print(merge_sort(a))
```

C: 정렬된 두 구간의 병합

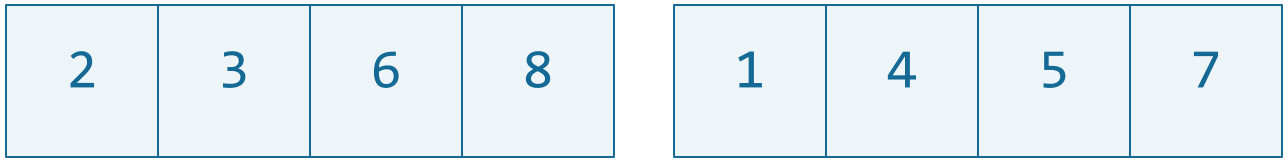
```
void merge(int a[], int tmp[], int lo, int mid, int hi) {
    int i = lo, j = mid + 1, k = lo;
    while (i <= mid && j <= hi) {
        if (a[i] <= a[j]) tmp[k++] = a[i++];
        else tmp[k++] = a[j++];
    }
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];
    for (int t = lo; t <= hi; t++) a[t] = tmp[t];
}
```

C: 재귀 호출과 실행

```
/* 앞 슬라이드의 merge 함수를 함께 넣는다. */
void merge_sort(int a[], int tmp[], int lo, int hi) {
    if (lo >= hi) return;
    int mid = lo + (hi - lo) / 2;
    merge_sort(a, tmp, lo, mid);
    merge_sort(a, tmp, mid + 1, hi);
    merge(a, tmp, lo, mid, hi);
}
/* 실행 예: main 안에서 */
int a[] = {8, 3, 6, 2, 7, 1, 5, 4};
int tmp[8];
merge_sort(a, tmp, 0, 7);
/* a는 {1, 2, 3, 4, 5, 6, 7, 8}이 된다. */
```

병합 한 번: 값 8개를 결과에 8번 넣습니다

이미 정렬된 두 입력



두 줄에서 작은 값부터 하나씩 선택

결과



4개 + 4개 = 8개를 한 번씩 처리

병합 한 번의 작업량은 전체 길이에 비례합니다. 길이가 n이면 O(n)입니다.

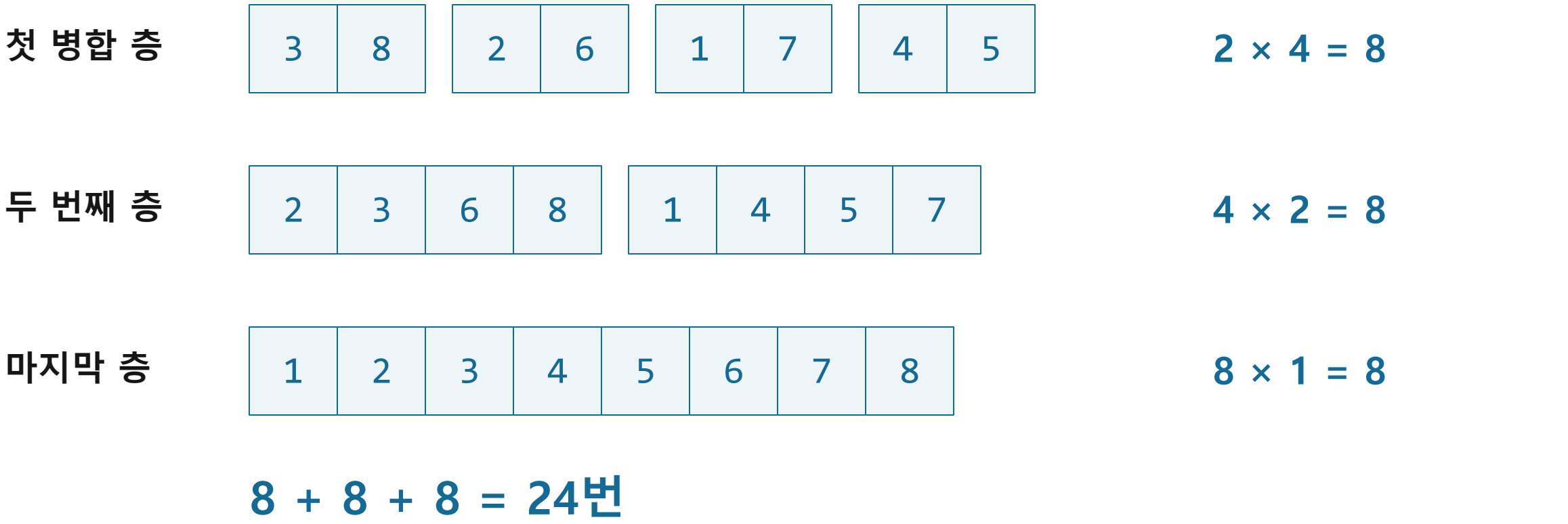
병합 한 번: 세 while문의 합은 선형 시간

반복문	하는 일	진행하는 인덱스
양쪽을 비교하는 while	작은 값 하나를 결과에 넣음	i 또는 j가 1 증가
왼쪽이 남은 while	남은 왼쪽 값을 순서대로 넣음	i가 1 증가
오른쪽이 남은 while	남은 오른쪽 값을 순서대로 넣음	j가 1 증가

세 반복문의 출력 횟수 합 = p + q = n

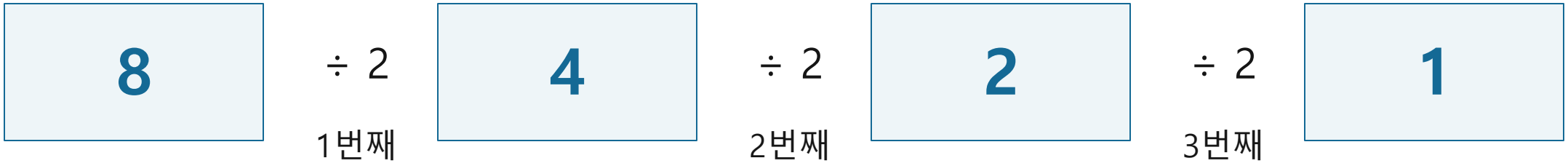
값 하나는 결과에 한 번만 들어갑니다. 병합 한 번의 시간은 O(n)입니다.

전체 병합: 한 층에서 다루는 값은 8개



각 값이 결과에 들어가는 횟수를 셉니다. 한 원소에서 바로 반환하는 비용은 별도로 있습니다.

$\log_2 8 = 3$: 반으로 나누는 횟수가 3번



8에서 1까지 줄이는 횟수: 3번

$$\log_2 8 = 3$$

$\log_2 16 = 4$, $\log_2 32 = 5$. 원소 수를 두 배로 늘리면 병합 층이 하나 늘어납니다.

n log n: 한 층의 작업량에 층 수를 곱합니다

원소 수	병합 층 수	결과에 넣는 횟수
8개	3층	8 × 3 = 24
16개	4층	16 × 4 = 64
32개	5층	32 × 5 = 160

원소 수 n × 층 수 $\log_2 n$

병합 정렬의 전체 시간은 O(n log n). O는 입력이 커질 때 작업량이 늘어나는 정도를 나타냅니다.

T(n): 원소 n개를 정렬하는 데 드는 시간

8개 정렬의 시간 = 왼쪽 4개 + 오른쪽 4개 + 병합

T(8) = **T(4)** + **T(4)** + **병합 비용**

전체 시간 | 왼쪽 절반의 시간 | 오른쪽 절반의 시간 | 현재 구간의 작업

원소 수를 n으로 바꾸면

$$T(n) = 2T(n/2) + O(n)$$

2T(n/2)는 절반 크기의 문제 두 개를 풀 시간입니다. O(n)은 그 두 결과를 합칠 시간입니다.

병합 정렬 확인

왼쪽 [2, 5], 오른쪽 [1, 3, 7]을 병합하자.
① 첫 번째로 선택하는 값과 움직이는 포인터는?
② 왼쪽이 먼저 끝났을 때 어떤 값을 붙이는가?
③ 재귀 코드에서 merge를 두 재귀 호출 뒤에 쓰는 이유는?

왼쪽

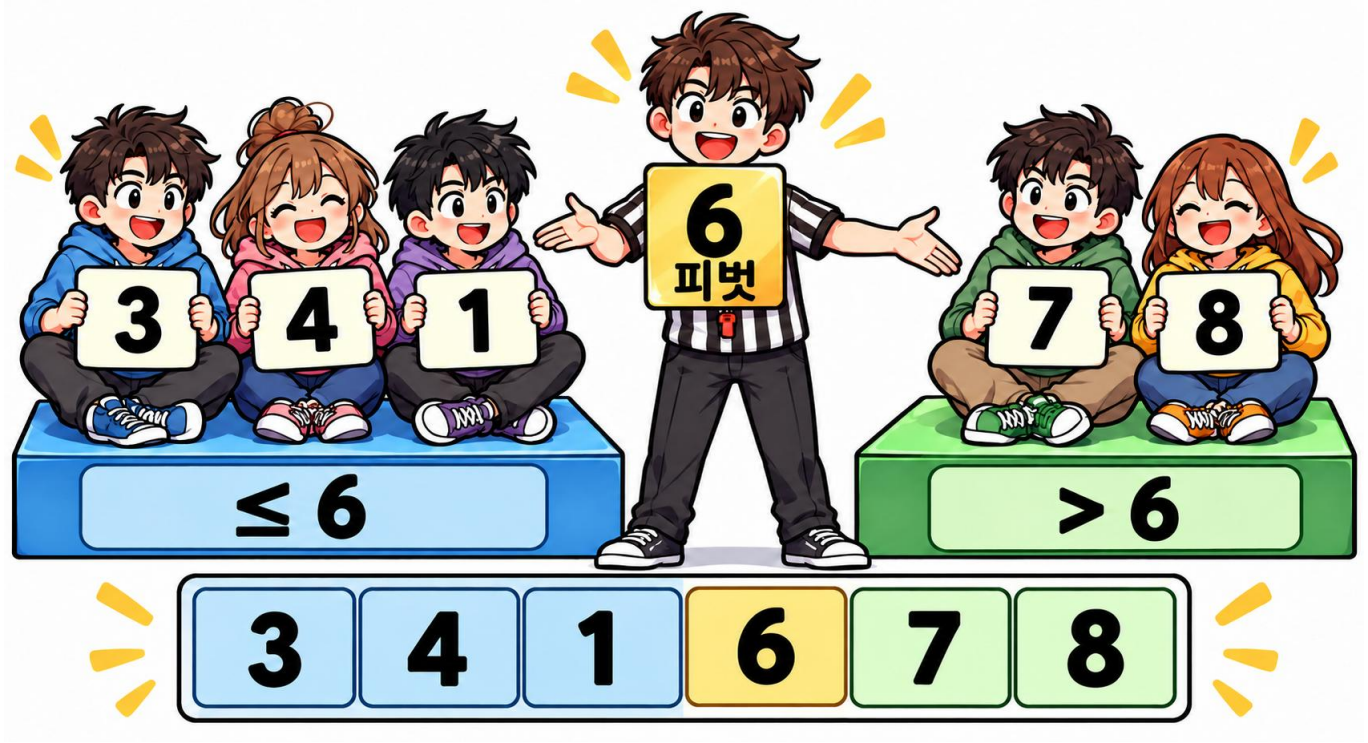
2	5

오른쪽

1	3	7

퀵 정렬: 피벗을 기준으로 구간 나누기

입력은 [8, 4, 6, 3, 7, 1]이고, 이번 비교 기준값인 피벗은 6입니다.
이 구현은 6 이하를 왼쪽에, 6보다 큰 값을 오른쪽에 모읍니다.

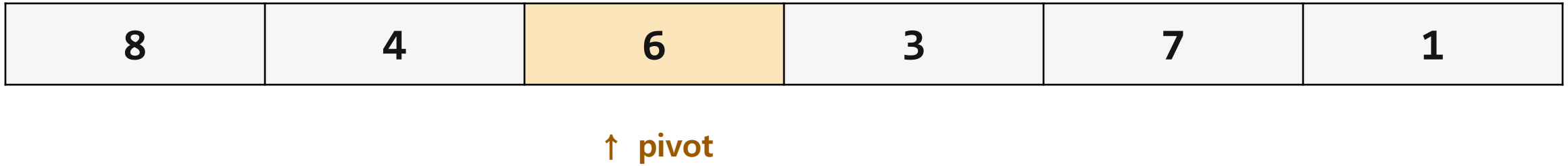


분할 직후에는 양쪽 구간이 아직 정렬되지 않을 수 있습니다. 각 구간을 다시 정렬합니다.

퀵 정렬 (Quick Sort)

피벗은 원소들을 나누는 비교 기준값입니다.
이번 구현은 피벗 이하를 왼쪽, 피벗 초과를 오른쪽에 모읍니다.

이번 입력



분할 뒤 피벗의 자리를 정하고, 피벗 양쪽 구간을 다시 정렬합니다.

피벗 선택: 가운데 칸의 6

현재 구간은 lo = 0부터 hi = 5까지입니다.
가운데 인덱스 (lo + hi) // 2 = 2에서 피벗 a[2] = 6을 고릅니다.



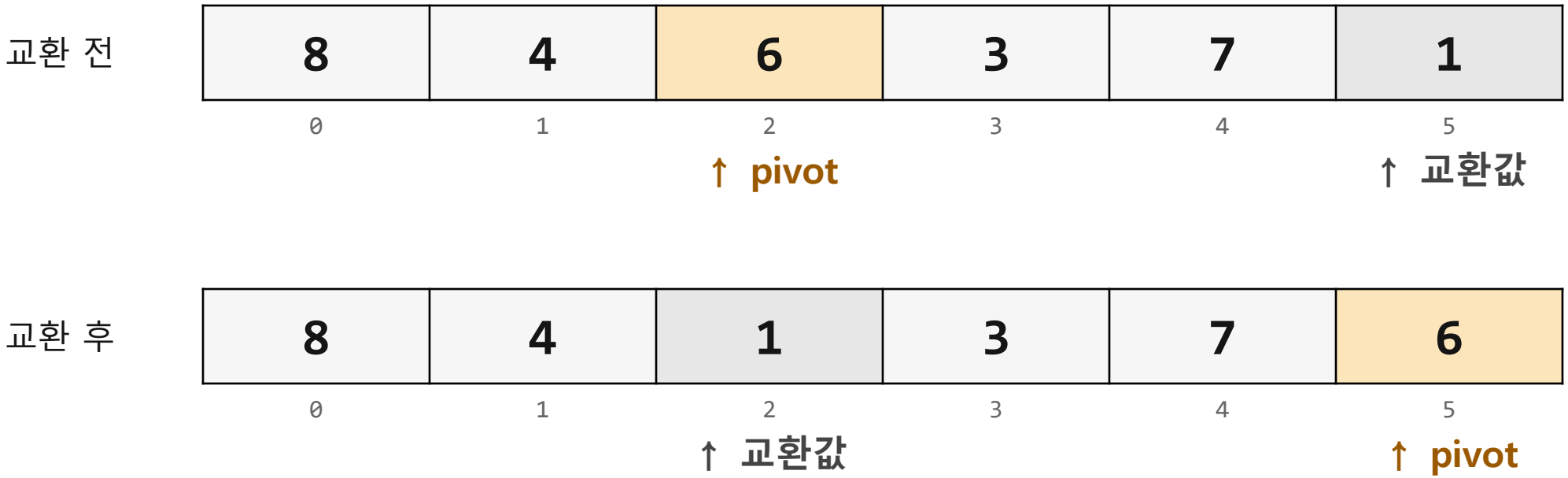

```
pivot_idx = (lo + hi) // 2
```

이번 코드가 사용하는 피벗 선택 규칙입니다.

피벗 6과 마지막 값 1의 교환

피벗 a[2] = 6과 마지막 값 a[5] = 1을 교환합니다.
6은 인덱스 5로, 1은 인덱스 2로 이동합니다.

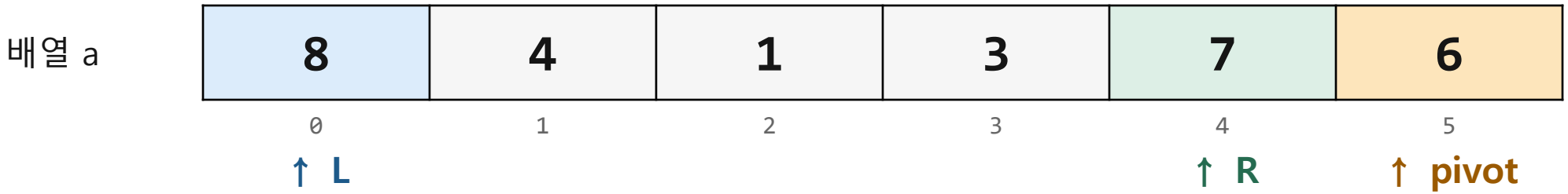
```
a[pivot_idx], a[hi] = a[hi], a[pivot_idx]
```



끝의 6은 검사 동안 임시로 두는 값입니다. 최종 위치는 분할이 끝난 뒤 정합니다.

① 탐색 시작: L = 0, R = 4

L = 0, R = 4에서 시작합니다.
마지막 칸의 피벗 6을 제외하고, 인덱스 0~4를 검사합니다.
첫 비교에서 8 > 6이므로 L은 인덱스 0에서 멈춥니다.

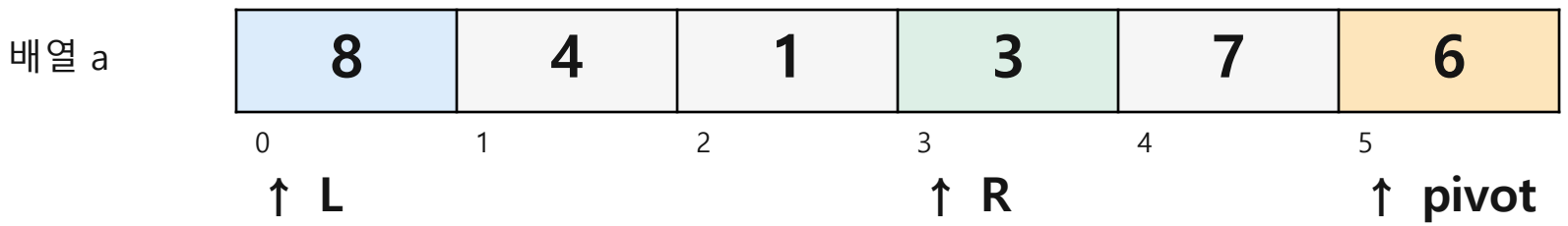



```
pivot = a[hi]
L, R = lo, hi - 1
```

다음: R이 7을 지나, 왼쪽으로 보낼 값 3을 찾습니다.

② R이 4에서 3으로 이동

a[4] = 7 > 6이므로 R을 왼쪽으로 한 칸 옮깁니다.
a[3] = 3 ≤ 6이므로 여기서 멈춥니다. L은 0에 그대로 있습니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

두 포인터가 찾은 8과 3의 교환

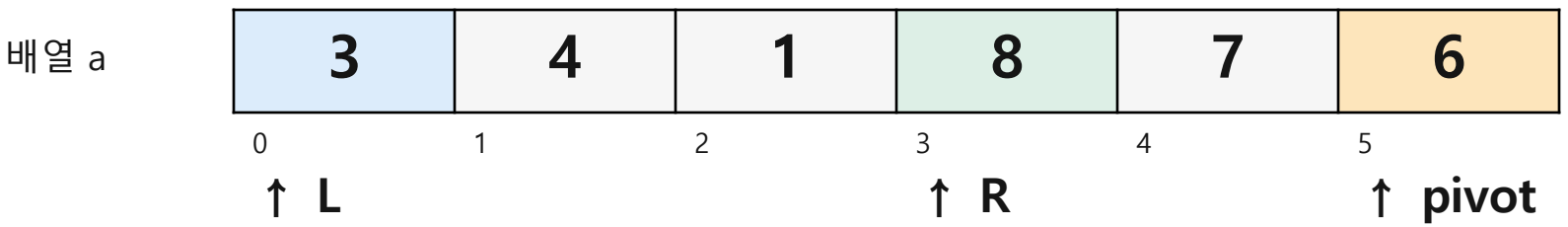
L = 0의 8은 오른쪽으로, R = 3의 3은 왼쪽으로 보내야 합니다. 피벗은 끝의 6입니다.



이번에는 두 값만 맞바꿉니다. L = 0, R = 3은 다음 탐색에서 이동합니다.

③ 두 포인터가 가리킨 8과 3 교환

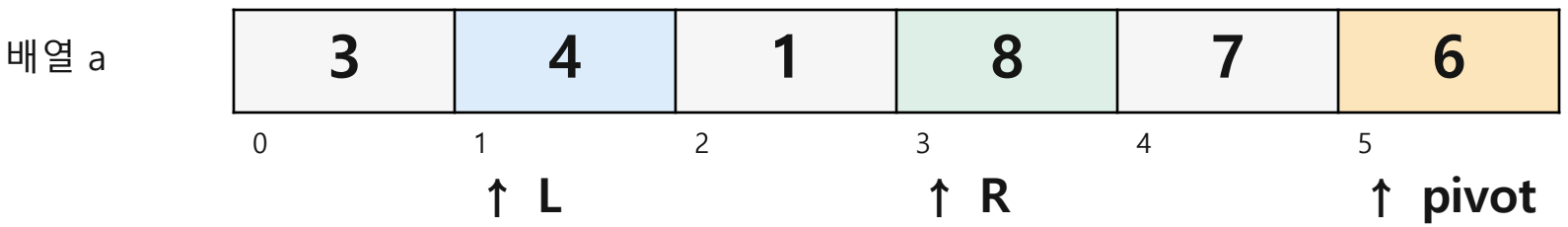
L = 0, R = 3이고 아직 L ≤ R입니다.
a[0]과 a[3]을 교환합니다. 포인터 위치는 그대로입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

④ L이 0에서 1로 이동

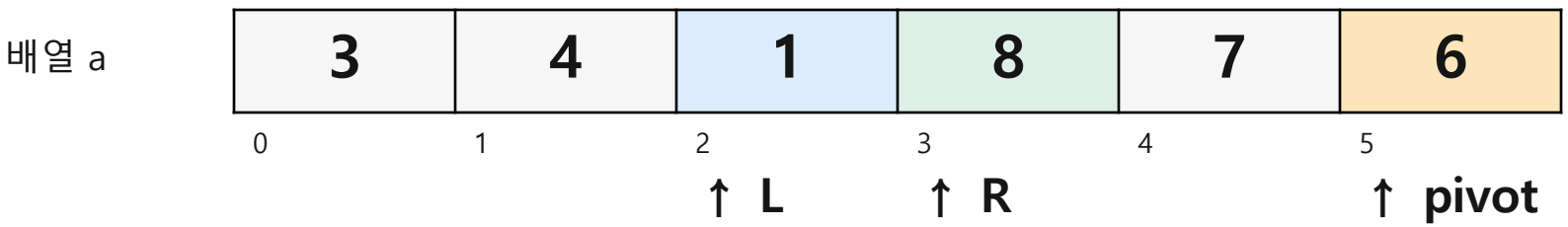
a[0] = 3 ≤ 6이므로 L을 오른쪽으로 한 칸 옮깁니다.
R은 인덱스 3에 그대로 있습니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

⑤ L이 1에서 2로 이동

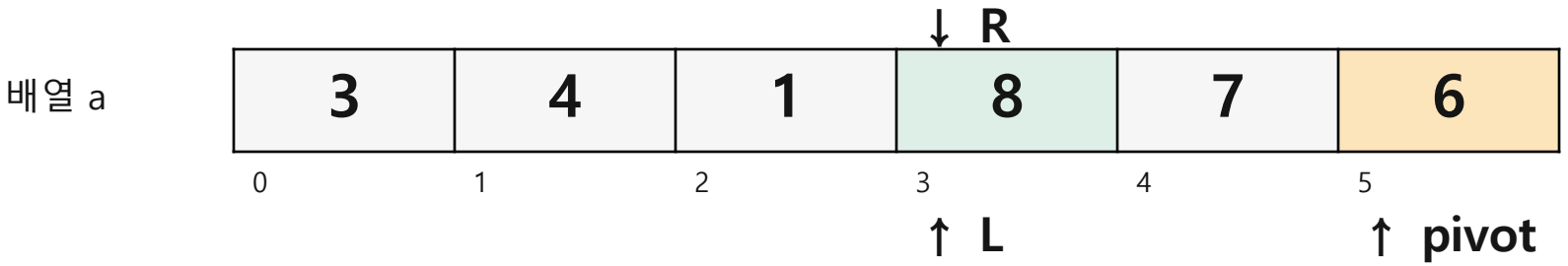
a[1] = 4 ≤ 6이므로 L을 다시 한 칸 옮깁니다.
지금 L이 가리키는 값은 a[2] = 1입니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

⑥ L이 2에서 3으로 이동

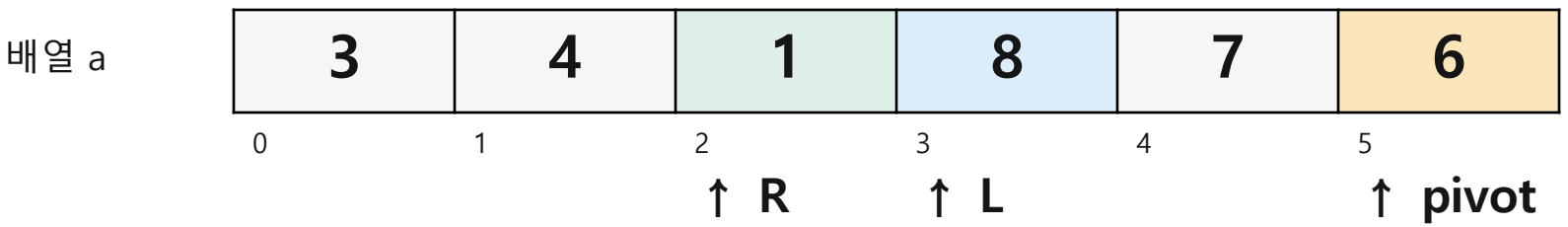
a[2] = 1 ≤ 6이므로 L을 한 칸 옮깁니다.
a[3] = 8 > 6이므로 L의 이동은 여기서 끝납니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

⑦ R이 3에서 2로 이동 · 교차

a[3] = 8 > 6이므로 R을 왼쪽으로 한 칸 옮깁니다.
L = 3 > R = 2가 되었으므로 두 값의 교환을 끝냅니다.



한 페이지씩 넘기며 화살표 위치와 바뀐 값을 비교하세요.

마지막 교환: 피벗 6의 자리 확정

L = 3, R = 2로 교차했습니다. L이 가리키는 8과 맨 끝의 피벗 6을 바꿉니다.

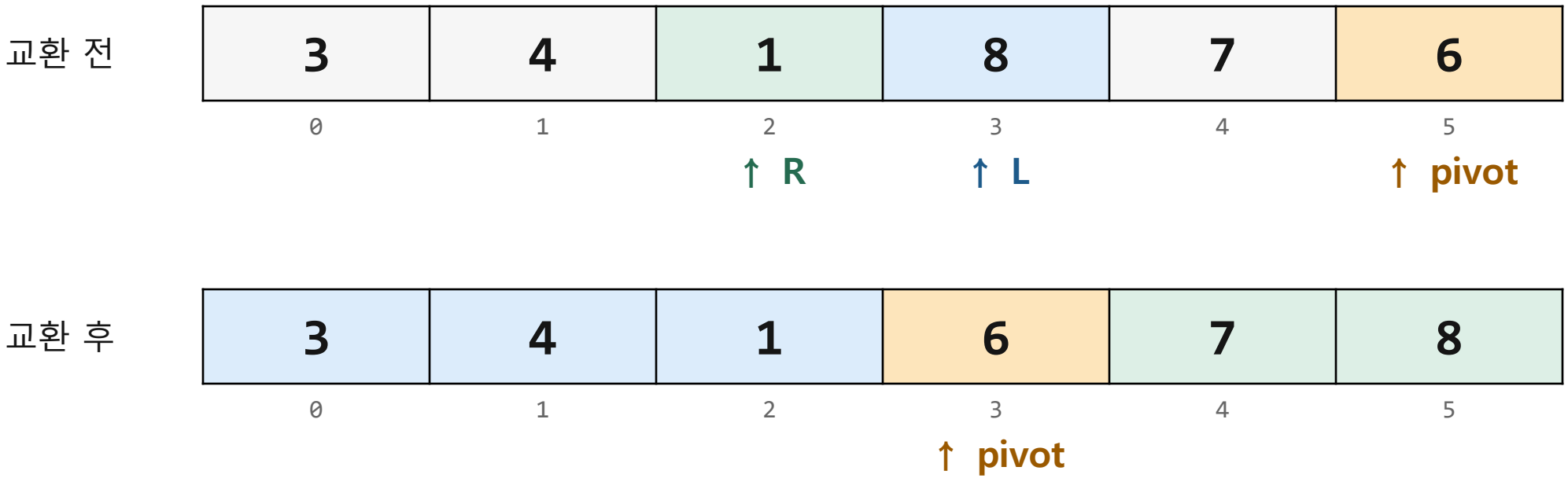


6의 자리는 확정되었습니다. 6을 제외한 왼쪽 세 칸과 오른쪽 두 칸을 다시 정렬합니다.

⑧ 피벗을 두 구간 사이에 놓는 교환

L = 3, R = 2로 교차했습니다. L은 큰 값 구간의 첫 칸입니다.
그 칸의 8과 맨 끝 피벗 6을 교환합니다. 피벗의 최종 위치는 3입니다.

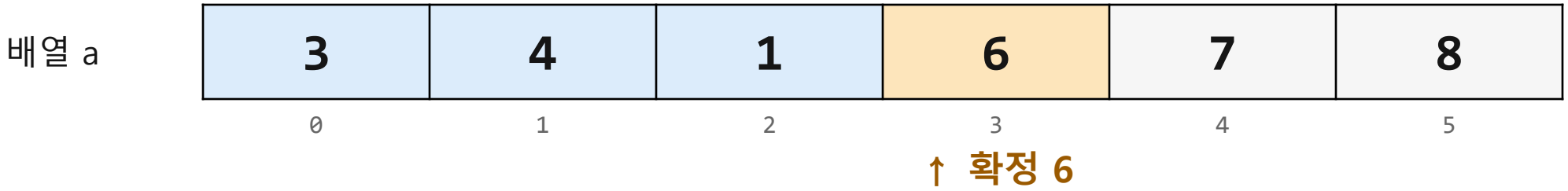
```
a[L], a[hi] = a[hi], a[L]
```



파랑: 6 이하인 왼쪽 구간 주황: 피벗 6 초록: 6보다 큰 오른쪽 구간

다음 호출: 왼쪽 구간 0~2부터

피벗 6의 위치 p = 3은 확정됐습니다. 다음 정렬에서 제외합니다.
왼쪽 [3,4,1]을 먼저 정렬하고, 그 호출이 끝나면 오른쪽 [7,8]을 정렬합니다.




```
quick_sort(a, 0, 2)
quick_sort(a, 4, 5)
```

먼저 왼쪽 호출을 끝냅니다. 오른쪽 호출은 그동안 기다립니다.

왼쪽 [3,4,1]: 피벗 4의 이동

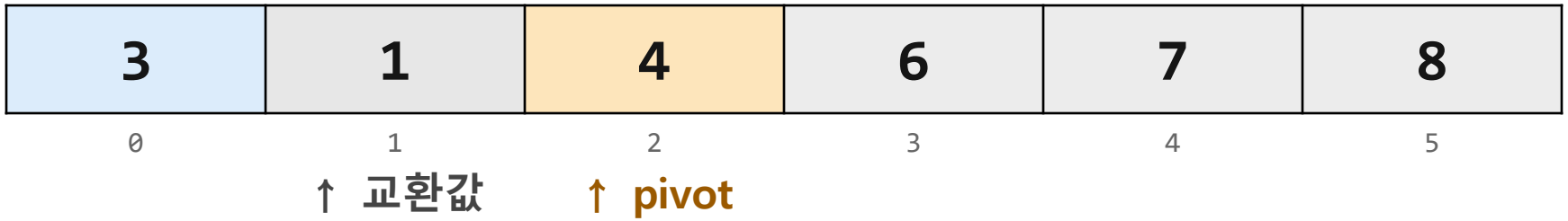
lo = 0, hi = 2이므로 가운데 인덱스는 1입니다. 피벗은 a[1] = 4입니다.
피벗 4와 이 구간의 마지막 값 a[2] = 1을 교환합니다.

```
a[1], a[2] = a[2], a[1]
```

교환 전



교환 후



현재 구간의 마지막 칸은 hi = 2입니다.

피벗 4: L = 0, R = 1에서 시작

검사할 값은 3과 1입니다. 피벗 4는 인덱스 2에 따로 둡니다.
L은 3을, R은 1을 가리킵니다. 먼저 L의 값부터 검사합니다.

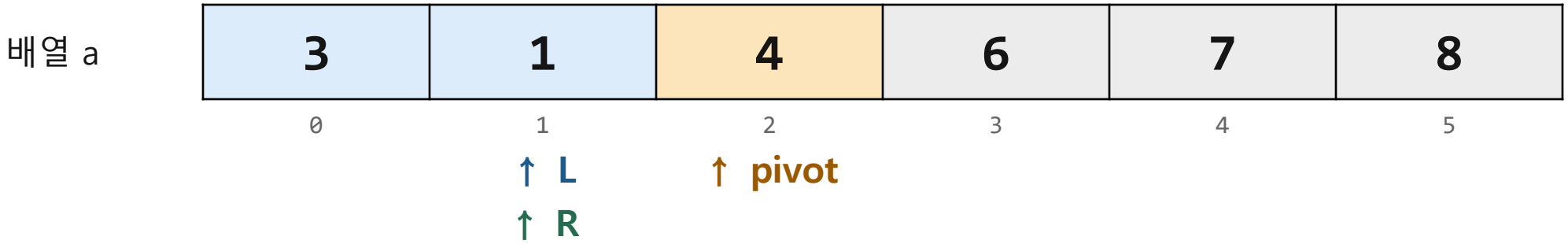



```
pivot = 4
L, R = 0, 1
```

3 ≤ 4이므로 다음에 L을 한 칸 늘립니다.

피벗 4: L이 0에서 1로 이동

a[0] = 3 ≤ 4이므로 L을 1로 늘립니다. 배열 값은 그대로입니다.
이제 L과 R은 모두 인덱스 1의 값 1을 가리킵니다.

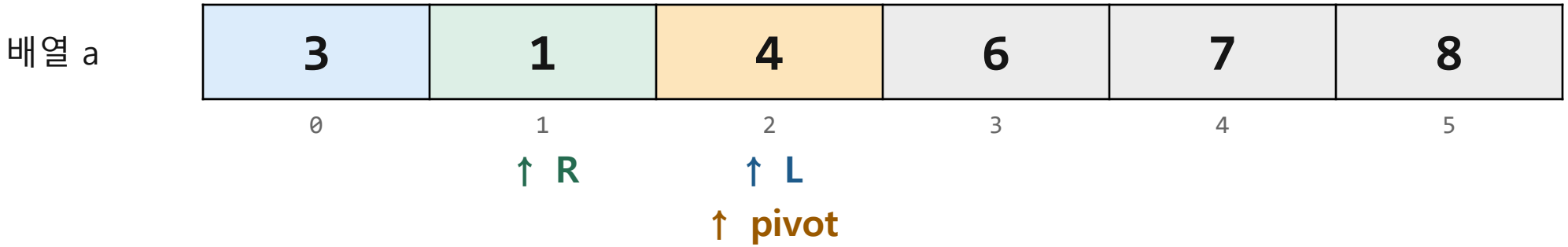


L = 1

L = R인 칸도 검사합니다. 1 ≤ 4이므로 L이 다시 이동합니다.

피벗 4: 교차 후 인덱스 2 확정

a[1] = 1 ≤ 4이므로 L = 2, R = 1로 교차합니다. 탐색을 끝냅니다.
L = hi = 2이므로 마지막 교환은 4와 자기 자신입니다. 배열은 그대로입니다.



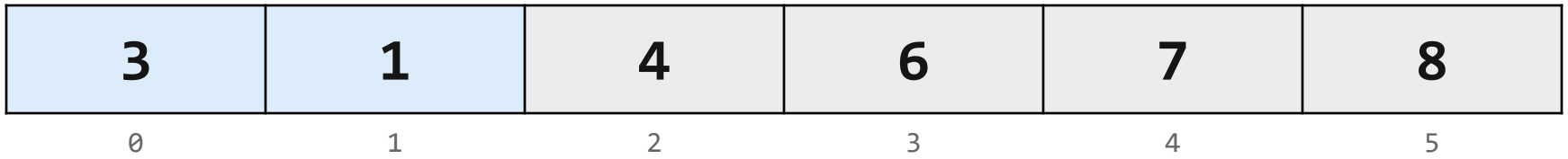

```
a[2], a[2] = a[2], a[2]
```

partition의 반환값 p = 2입니다. 앞의 [3,1]은 아직 정렬해야 합니다.

다음 왼쪽 호출: 구간 0~1

피벗 4의 위치 p = 2를 제외하고, 왼쪽 [3,1]을 다시 정렬합니다.
현재 lo = 0, hi = 1입니다. 4와 6의 칸은 이번 호출에서 움직이지 않습니다.

배열 a




```
quick_sort(a, 0, 1)
```

원소가 두 개이므로 한 번 더 분할합니다.

왼쪽 [3,1]: 피벗 3의 이동

가운데 인덱스 (0 + 1) // 2 = 0이므로 피벗은 a[0] = 3입니다.
피벗 3과 현재 구간의 마지막 값 a[1] = 1을 교환합니다.

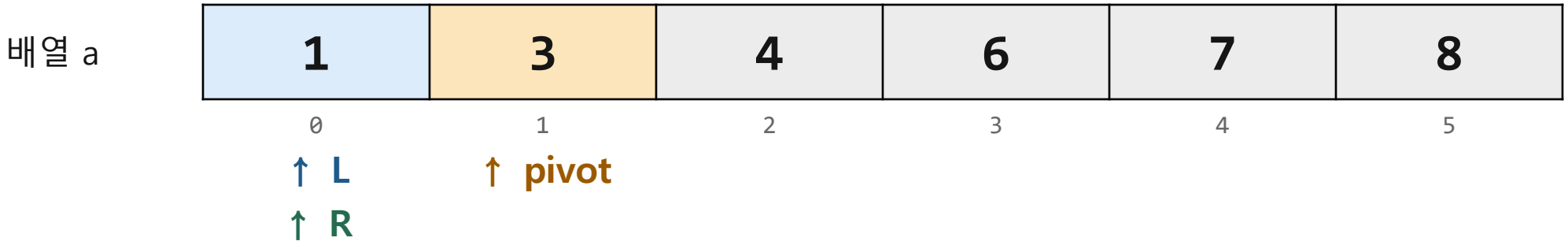
```
a[0], a[1] = a[1], a[0]
```



이번 구간의 hi는 1입니다. 그 뒤의 칸은 바꾸지 않습니다.

피벗 3: L = R = 0에서 시작

피벗 3은 인덱스 1에 두고, 앞의 값 1만 검사합니다.
L과 R은 모두 인덱스 0을 가리킵니다. 1 ≤ 3이므로 L을 늘립니다.

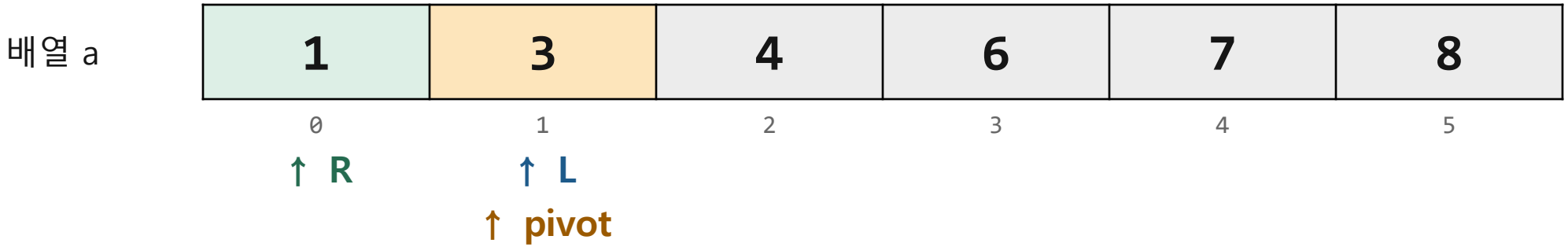



```
pivot = 3
L, R = 0, 0
```

L = R이어도 현재 칸을 한 번 검사합니다.

피벗 3: 교차 후 인덱스 1 확정

1 ≤ 3이므로 L = 1, R = 0이 되어 탐색을 끝냅니다.
L = hi = 1입니다. 마지막 교환은 3과 자기 자신이므로 배열은 그대로입니다.

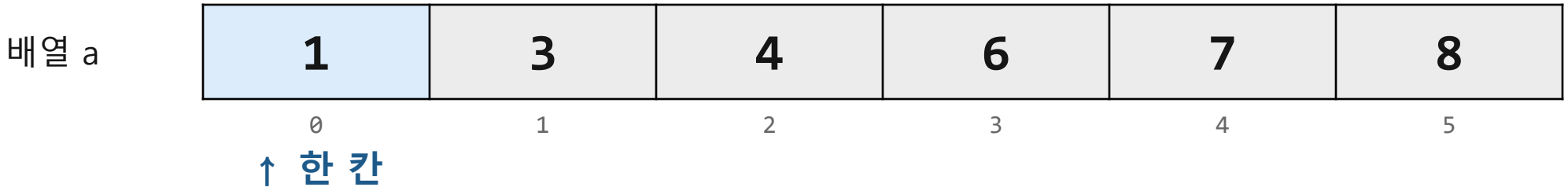



```
a[1], a[1] = a[1], a[1]
```

partition의 반환값 p = 1입니다. 다음 구간은 [0,0]과 [2,1]입니다.

종료 조건: 한 칸과 빈 구간

quick_sort(a, 0, 0)은 원소 하나입니다. quick_sort(a, 2, 1)은 빈 구간입니다.
두 호출 모두 lo ≥ hi이므로 partition을 호출하지 않고 바로 돌아옵니다.

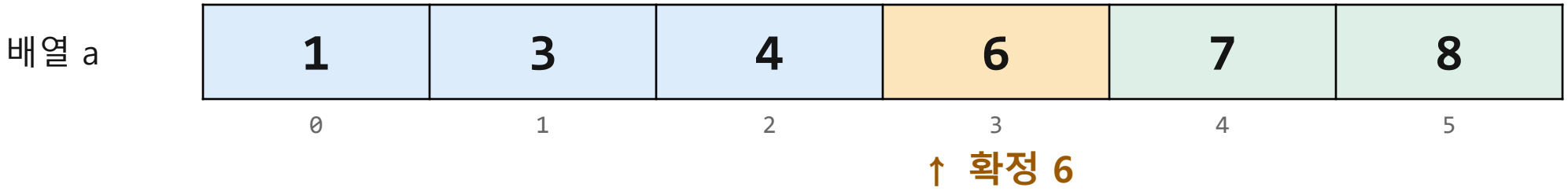



```
quick_sort(a, 0, 0)
quick_sort(a, 2, 1)
```

종료 조건 lo ≥ hi는 원소 하나인 경우와 빈 구간을 함께 처리합니다.

왼쪽 정렬 종료와 원래 호출로 복귀

구간 0~1을 끝내고 0~2 호출로 돌아옵니다. 오른쪽 [3,2]도 빈 구간입니다.
왼쪽 [1,3,4]를 모두 끝냈으므로, 원래 0~5 호출에서 오른쪽 4~5를 시작합니다.



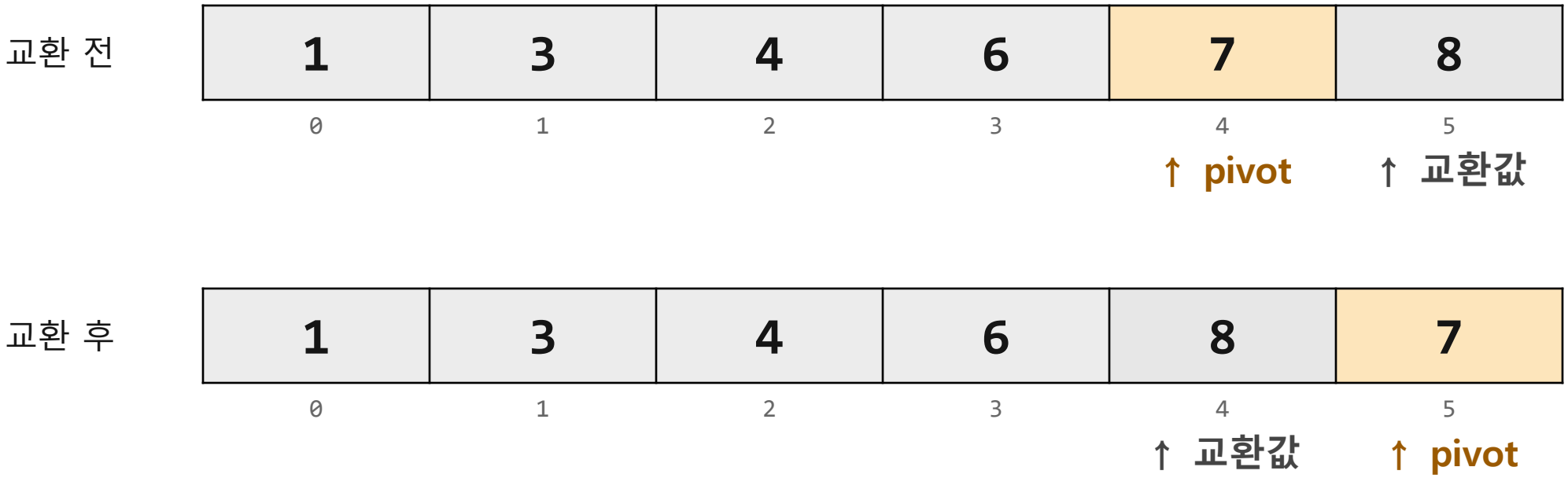

```
quick_sort(a, 3, 2)
quick_sort(a, 4, 5)
```

왼쪽 [1,3,4]는 완료했습니다. 다음 검사 범위는 인덱스 4~5입니다.

오른쪽 [7,8]: 피벗 7의 이동

lo = 4, hi = 5입니다. 가운데 인덱스는 4이고 피벗은 a[4] = 7입니다.
피벗 7과 현재 구간의 마지막 값 a[5] = 8을 교환합니다.

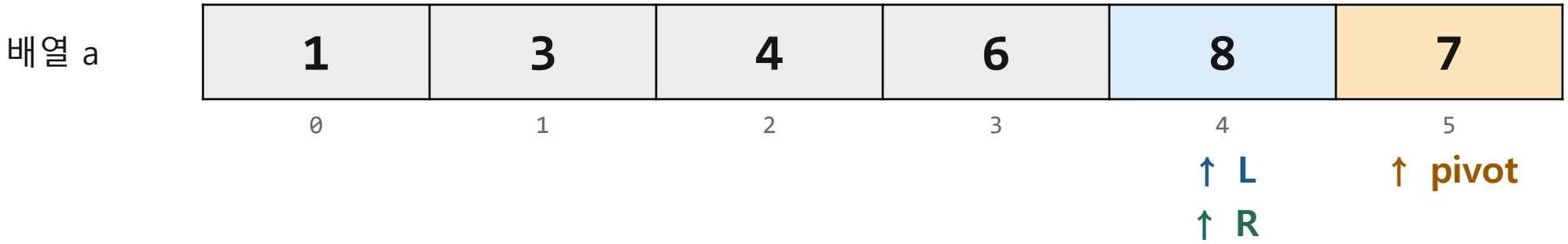
```
a[4], a[5] = a[5], a[4]
```



원래 [7,8]이어도 이 코드의 준비 교환을 그대로 실행합니다.

피벗 7: L = R = 4에서 시작

검사할 값은 a[4] = 8 하나입니다. 피벗 7은 인덱스 5에 둡니다.
8 > 7이므로 L은 멈춥니다. 이어서 R의 조건을 검사합니다.

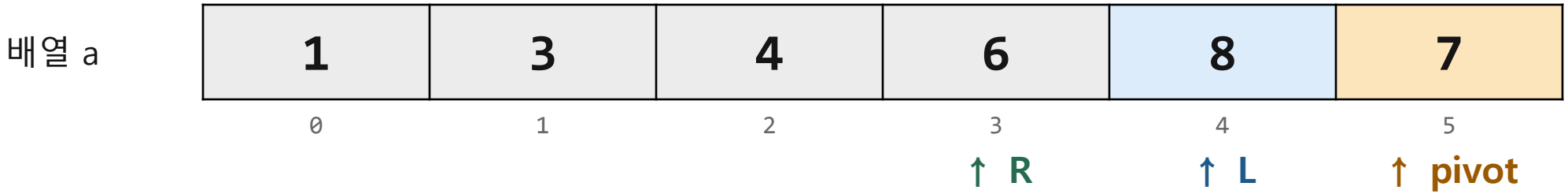



```
pivot = 7
L, R = 4, 4
```

L은 8 > 7에서 멈춥니다. R은 같은 8을 지나 왼쪽으로 이동합니다.

피벗 7: R이 4에서 3으로 이동

a[4] = 8 > 7이므로 R을 3으로 줄입니다. L = 4, R = 3으로 교차합니다.
L ≤ R 조건이 거짓이므로 a[3] = 6은 비교하지 않습니다. 이번 탐색은 끝입니다.




```
L, R = 4, 3
```

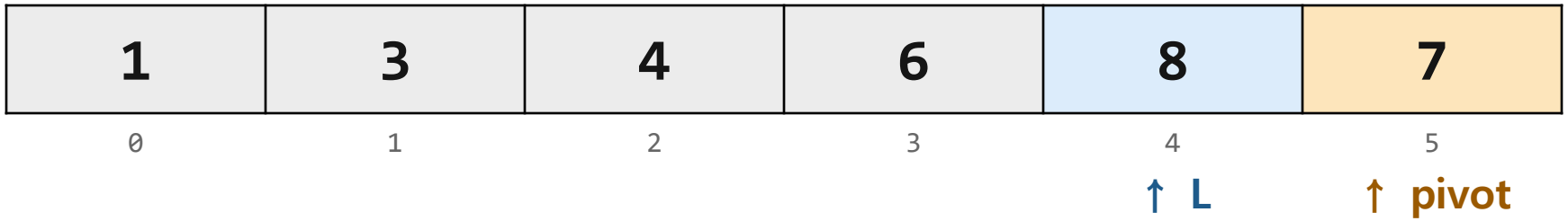
R은 범위 밖으로 이동했지만, 범위 밖의 값 6은 검사하거나 바꾸지 않습니다.

피벗 7과 8 교환: 인덱스 4 확정

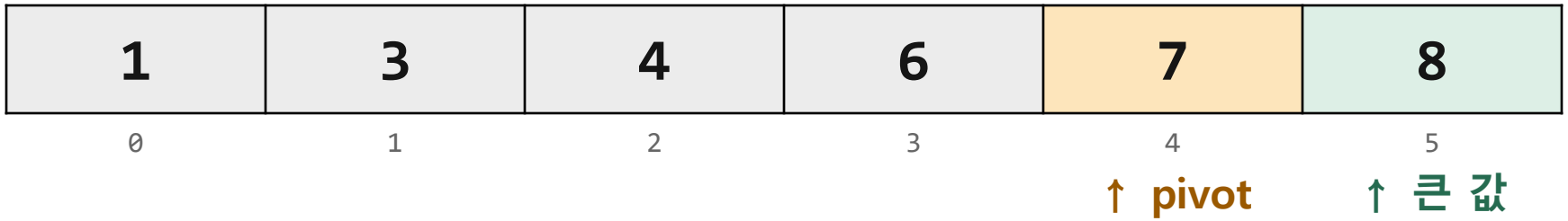
L = 4의 값 8과 hi = 5의 피벗 7을 교환합니다.
피벗 7은 인덱스 4, 큰 값 8은 인덱스 5에 놓입니다.

```
a[4], a[5] = a[5], a[4]
```

교환 전



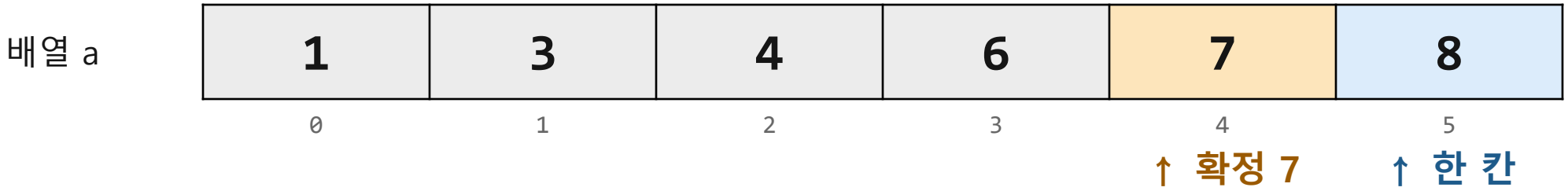
교환 후



partition의 반환값 p = 4입니다. 다음 구간은 [4,3]과 [5,5]입니다.

오른쪽의 남은 호출도 종료

왼쪽 [4,3]은 빈 구간이고, 오른쪽 [5,5]는 값 8 하나입니다.
두 호출 모두 lo ≥ hi에서 끝납니다. 오른쪽 4~5의 정렬도 완료했습니다.

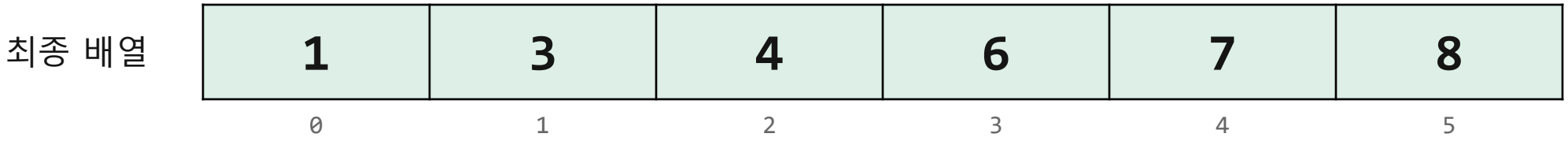



```
quick_sort(a, 4, 3)
quick_sort(a, 5, 5)
```

이 두 호출은 partition 없이 끝납니다. 원래 0~5 호출로 돌아갑니다.

퀵 정렬 완료: 모든 호출 종료

최종 배열은 [1,3,4,6,7,8]입니다. 원래 배열 안에서 정렬했습니다.
왼쪽 호출과 오른쪽 호출이 모두 끝나, 처음 quick_sort(a, 0, 5)도 종료합니다.




```
quick_sort(a, 0, 5)
```

피벗 확정 순서: 6(인덱스 3), 4(인덱스 2), 3(인덱스 1), 7(인덱스 4)

Python: 그림과 같은 partition

```
def partition(a, lo, hi):
    pivot_idx = (lo + hi) // 2
    a[pivot_idx], a[hi] = a[hi], a[pivot_idx]
    pivot = a[hi]
    L, R = lo, hi - 1
    while True:
        while L <= R and a[L] <= pivot:
            L += 1
        while L <= R and a[R] > pivot:
            R -= 1
        if L > R:
            break
        a[L], a[R] = a[R], a[L]
    a[L], a[hi] = a[hi], a[L]
    return L
```

Python: 피벗을 제외한 재귀 호출

```
def quick_sort(a, lo, hi):
    if lo >= hi:
        return
    p = partition(a, lo, hi)
    quick_sort(a, lo, p - 1)
    quick_sort(a, p + 1, hi)

a = [8, 4, 6, 3, 7, 1]
quick_sort(a, 0, len(a) - 1)
print(a)  # [1, 3, 4, 6, 7, 8]
```

C: 양쪽 포인터 partition

```
void swap(int *x, int *y) {
    int t = *x; *x = *y; *y = t;
}
int partition(int a[], int lo, int hi) {
    int m = lo + (hi - lo) / 2;
    swap(&a[m], &a[hi]);
    int pivot = a[hi], L = lo, R = hi - 1;
    while (1) {
        while (L <= R && a[L] <= pivot) L++;
        while (L <= R && a[R] > pivot) R--;
        if (L > R) break;
        swap(&a[L], &a[R]);
    }
    swap(&a[L], &a[hi]);
    return L;
}
```

C: 재귀 호출과 실행

```
/* swap과 partition을 이 함수 앞에 함께 넣는다. */
void quick_sort(int a[], int lo, int hi) {
    if (lo >= hi) return;
    int p = partition(a, lo, hi);
    quick_sort(a, lo, p - 1);
    quick_sort(a, p + 1, hi);
}
/* 실행 예: main 안에서 */
int a[] = {8, 4, 6, 3, 7, 1};
quick_sort(a, 0, 5);
/* a는 {1, 3, 4, 6, 7, 8}이 된다. */
```

같은 값과 빈 구간도 처리한다

이 구현은 피벗 이하를 왼쪽, 초과를 오른쪽에 둔다.
같은 값이 있어도 L이 계속 이동하여 무한 반복을 피한다.
모두 같은 값에서는 한쪽으로 치우쳐 최악 $\Theta(n^2)$가 된다.
빈 배열의 첫 호출은 quick_sort(a, 0, −1), 즉시 종료

모두 같은 값

2	2	2

분할 크기

왼쪽 2개	피벗 1개	오른쪽 0개

파랑 L: 왼쪽 포인터 초록 R: 오른쪽 포인터 주황: 피벗

피벗에 따라 달라지는 시간

분할 한 번은 Θ(n)
균형 분할은 Θ(log n) 단계, 전체 Θ(n log n)
한쪽에 n − 1개씩 남으면 최악 Θ(n²)
서로 다른 값의 무작위 순열에서 평균 Θ(n log n)
가운데 인덱스의 값이 중앙값이라는 보장은 없다.

균형 분할

절반	피벗	절반

치우친 분할

n − 1개	피벗	0개

분할 한 번: 포인터는 뒤로 돌아가지 않습니다

포인터	진행 방향	이 예제의 이동
L	왼쪽 → 오른쪽	0 → 1 → 2 → 3
R	오른쪽 → 왼쪽	4 → 3 → 2

교환 후에도 처음부터 다시 훑지 않습니다.

길이 k인 구간의 partition: O(k)

중첩된 while문의 모양보다, 포인터의 총 이동 횟수를 봅니다.

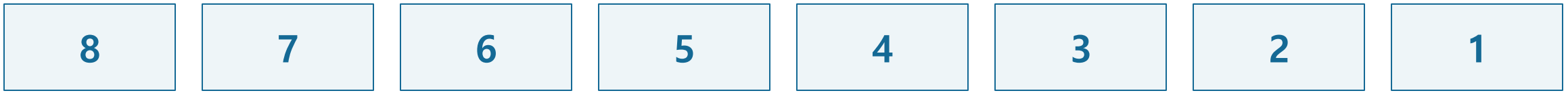
비슷하게 나뉘면: 층 수가 짧아집니다



한 층의 합계 O(n) × 약 O(log n)층

비슷한 크기로 나뉠 때 O(n log n). 가운데 칸의 값이 중앙값이라는 보장은 없습니다.

같은 값이 계속 나오면: 긴 재귀 사슬



입력: `[5, 5, 5, 5, 5, 5, 5, 5]`

피벗 밖의 검사: 7 + 6 + … + 1 = 28

한쪽에 n−1개가 남는 경우: $O(n^2)$

이 코드의 "피벗 이하를 왼쪽" 규칙에서는 같은 값만 있어도 최악의 분할이 가능합니다.

두 정렬의 차이 확인

병합 정렬: 위치로 반씩 나누고, 정렬된 결과를 병합
퀵 정렬: 피벗 값으로 분할하고, 양쪽을 재귀 정렬
병합은 최악에도 $\Theta(n \log n)$, 퀵은 최악 $\Theta(n^2)$
현재 병합 코드는 안정적, 퀵 코드는 안정성 보장 없음
확인: 퀵의 첫 분할만 끝나면 전체가 정렬되어 있을까?

4.4

최대 구간 합

Maximum Subarray Sum
주가 등락폭 데이터에서 가장 큰 이익을 내는 연속 구간을 찾아라

문제: 가장 큰 이익을 내는 연속 구간

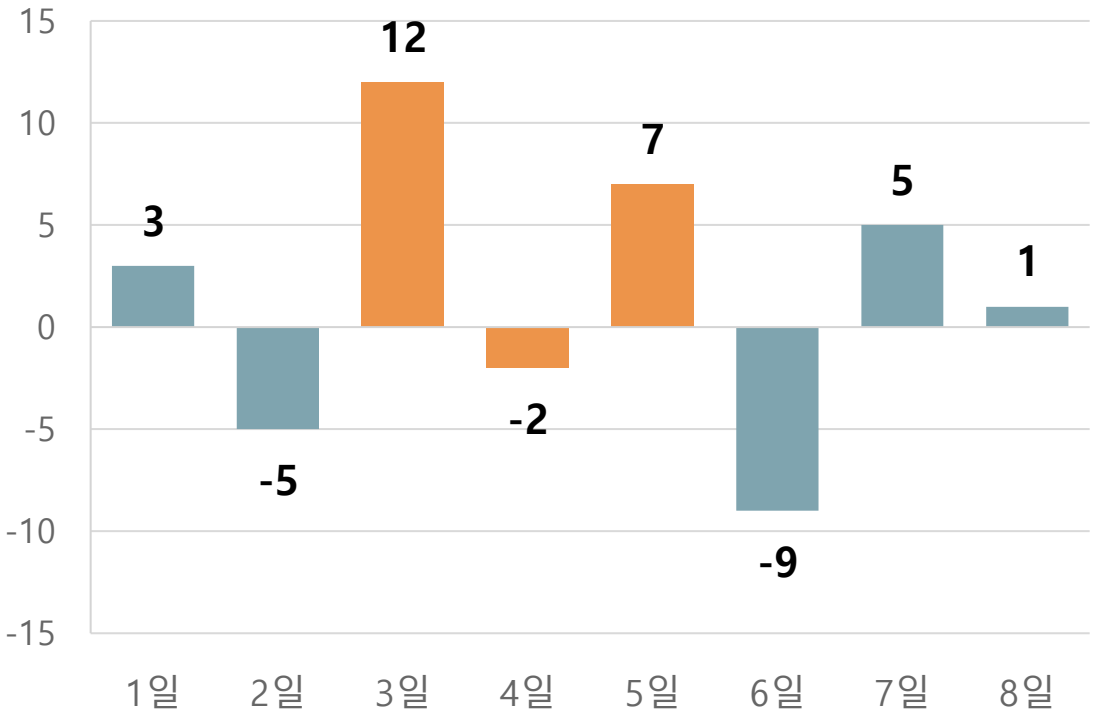
일별 손익 변동이 배열로 주어집니다.
연속한 날짜의 합을 최대로 만드세요.
하루 이상 고르며,
중간 날짜는 건너뛸 수 없습니다.

3	-5	12	-2	7	-9	5	1
1일	2일	3일	4일	5일	6일	7일	8일



전체 8일 합: 12

3~5일 합: 17



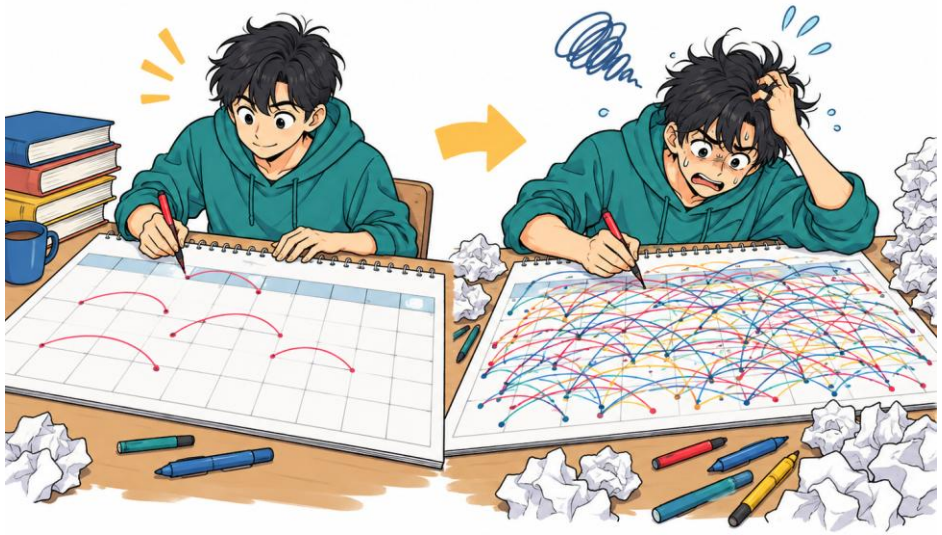
3~5일: 12 − 2 + 7 = 17

모든 구간을 하나씩 확인하는 방법

시작 날짜를 하나 고정하고,
끝 날짜를 한 칸씩 늘립니다.

첫날부터 시작한 구간

```
[3]                 →   3
[3, -5]             →  -2
[3, -5, 12]         →  10
…
```



8일의 후보: 8 + 7 + … + 1 = 36개

입력이 커지면 후보 수가 $O(n^2)$로 늘어납니다.

끝점을 한 칸 늘릴 때 누적합도 갱신합니다. 더 빠른 방법으로 분할 정복을 사용해 봅시다.

왼쪽·오른쪽·중앙을 걸친 구간

왼쪽 절반 안

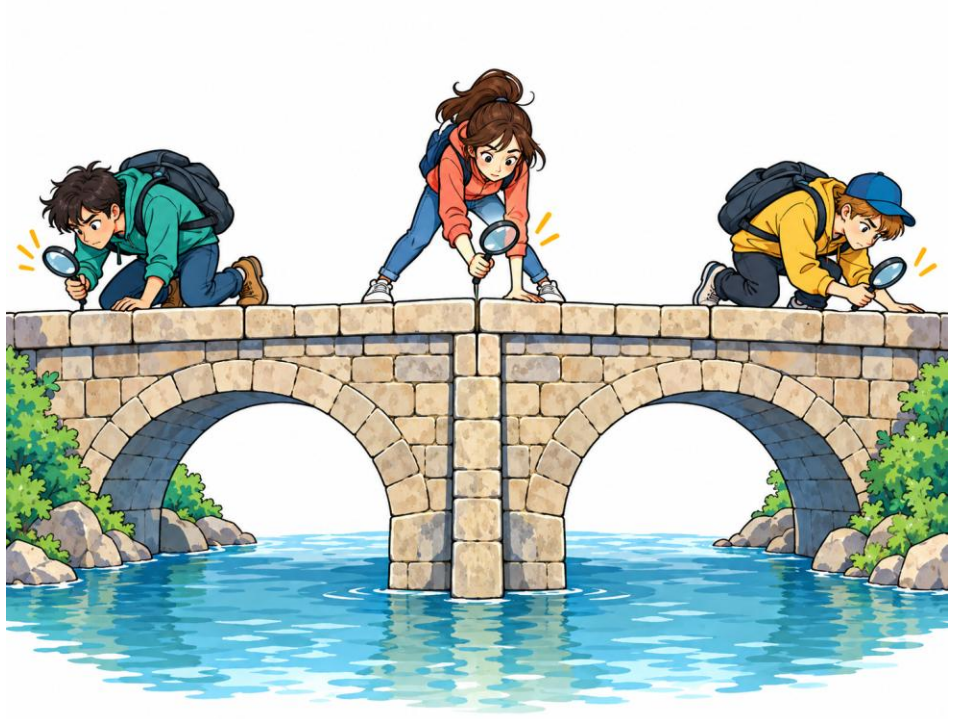
그 절반에서 찾은 최대 합: left_max

오른쪽 절반 안

그 절반에서 찾은 최대 합: right_max

중앙 경계를 걸친 구간

경계 양쪽을 이어 붙인 최대 합: cross_max

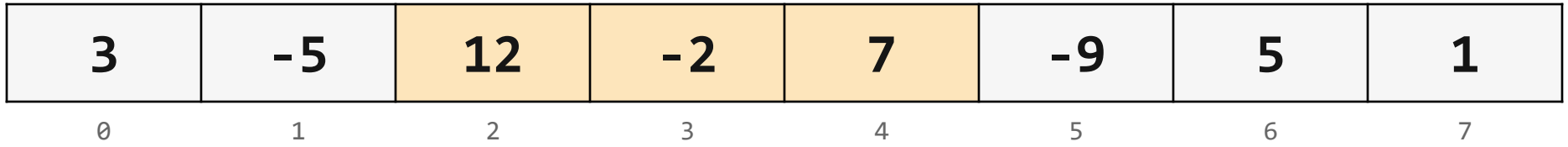


정답 = max(left_max, right_max, cross_max)

중앙을 걸친 구간은 mid와 mid+1을 모두 포함해야 합니다. 중간 칸을 뺄 수 없습니다.

최대 구간 합: 연속한 칸을 고릅니다

a = [3, −5, 12, −2, 7, −9, 5, 1]에서 빈 구간은 고르지 않습니다.
가장 큰 합은 인덱스 2~4의 12 − 2 + 7 = 17입니다.



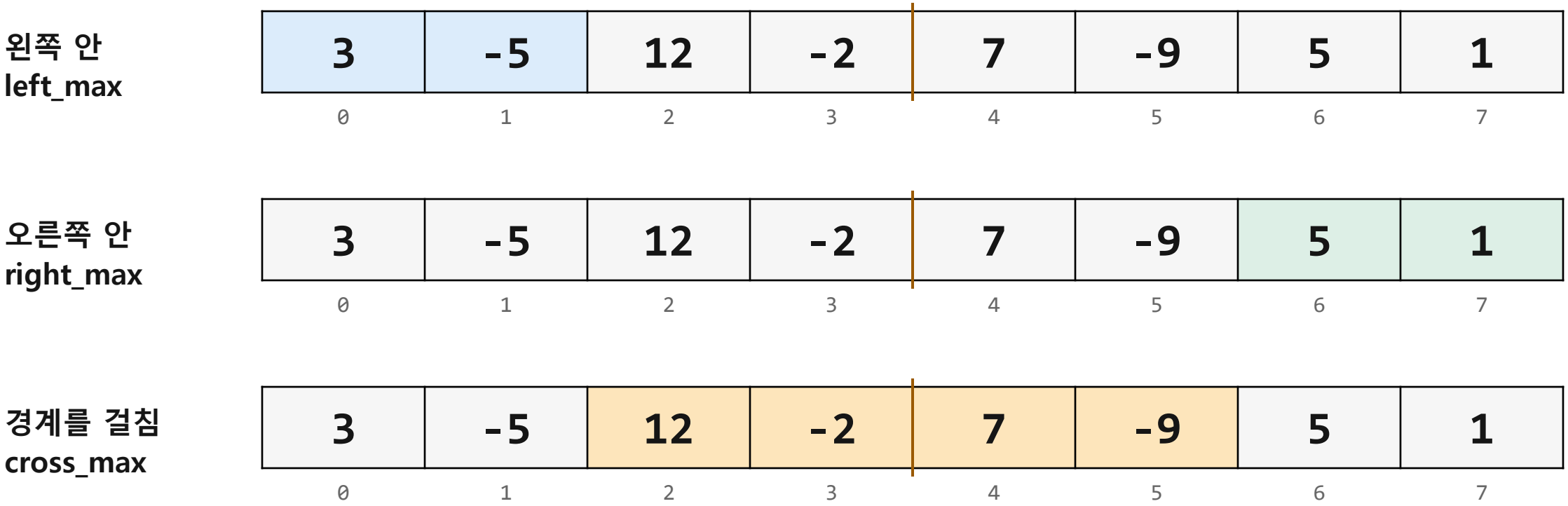
12 − 2 + 7 = 17

배열 전체 합 = 12

중간 칸을 건너뛸 수 없습니다. 음수 한 칸도 답이 될 수 있습니다.

가운데에서 나누면 후보는 세 종류

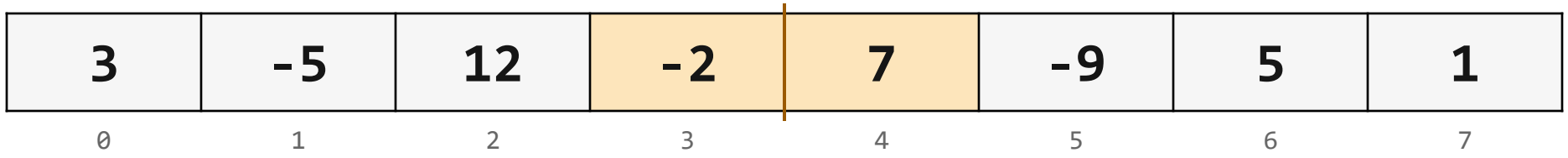
low = 0, high = 7, mid = 3입니다. 경계는 인덱스 3과 4 사이입니다.
왼쪽 안, 오른쪽 안, 경계를 걸치는 구간 중 가장 큰 합을 고릅니다.



세 줄은 후보의 종류를 보여 주는 예입니다. 실제 최댓값은 다음 과정에서 구합니다.

걸친 구간은 경계의 두 칸을 포함

왼쪽 마지막 칸 a[mid]와 오른쪽 첫 칸 a[mid + 1]은 빠질 수 없습니다.
왼쪽은 mid에서 왼쪽으로, 오른쪽은 mid + 1에서 오른쪽으로 늘립니다.



← 왼쪽 시작을 늘림

오른쪽 끝을 늘림 →

왼쪽 끝 = `mid`

오른쪽 시작 = `mid + 1`

left_sum과 right_sum은 경계에 붙은 구간의 합입니다.

left_max + right_max는 답이 아닙니다

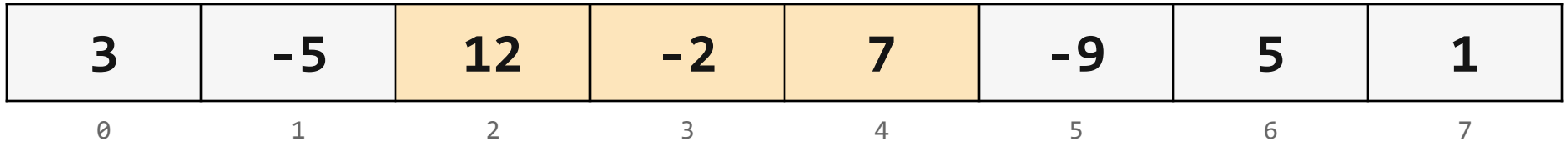
왼쪽에서 12, 오른쪽에서 7을 따로 찾더라도 12 + 7 = 19는 안 됩니다.
경계를 걸치려면 두 값 사이의 −2를 함께 더해야 합니다.

잘못 고른 칸



12 + 7 = 19 → 중간 −2를 빠뜨림

연속한 구간



가능한 합: 12 − 2 + 7 = 17. 아래에서는 경계 칸을 빼놓지 않았습니다.

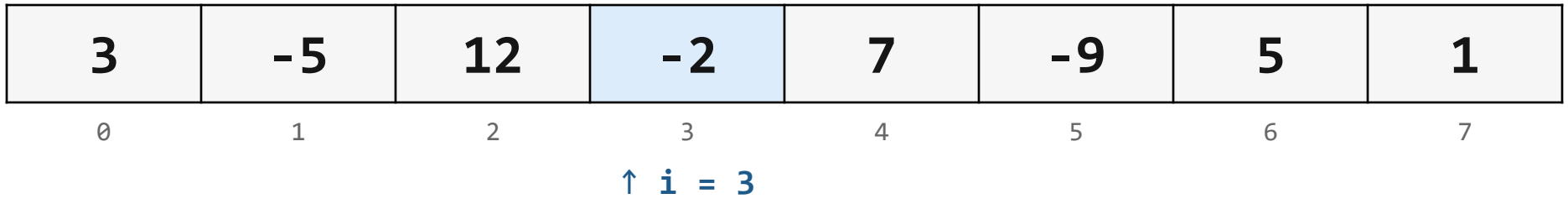
현재 합과 저장한 최댓값을 구별

current_sum은 지금 늘리는 구간의 합입니다. 다음 원소를 여기에 더합니다.
left_sum 또는 right_sum은 지금까지 본 후보 중 가장 큰 합을 저장합니다.

current_sum = 0

left_sum = −∞

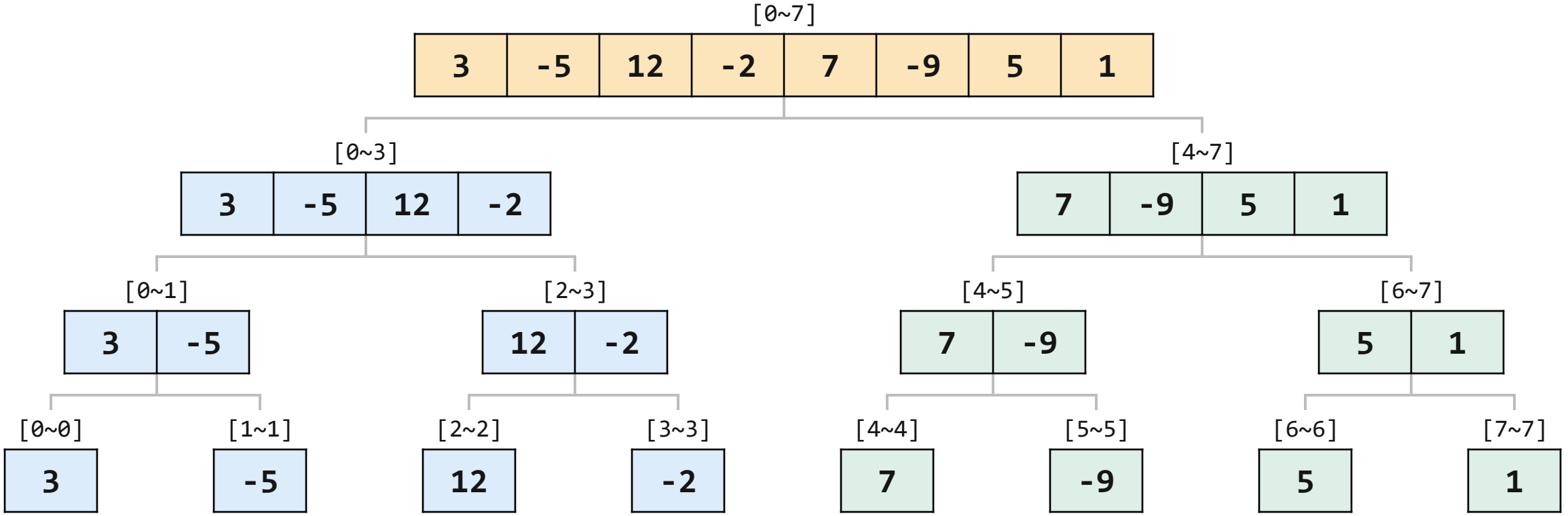
첫 칸을 읽음



첫 검사 뒤: current_sum = −2, left_sum = −2. 초기 0은 후보가 아닙니다.

실행 순서: 왼쪽을 끝내고 오른쪽으로

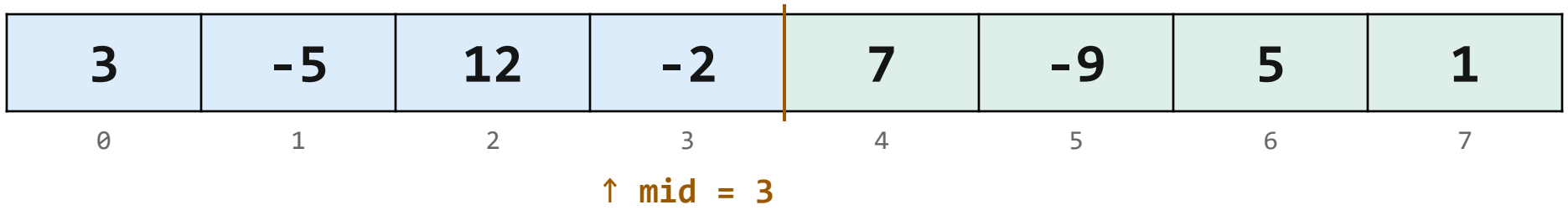
각 호출은 자신의 구간을 다시 반으로 나눕니다. 원소 하나면 그 값을 반환합니다.
왼쪽 호출이 끝나기 전에는 오른쪽 호출을 시작하지 않습니다.



이제 맨 위 호출을 시작해서 실제 코드 순서대로 끝까지 따라갑니다.

[0~7] 호출: mid = 3에서 나눔

현재 범위 low = 0, high = 7, mid = 3입니다.
먼저 왼쪽 [0~3]를 호출합니다. 오른쪽 [4~7]는 기다립니다.

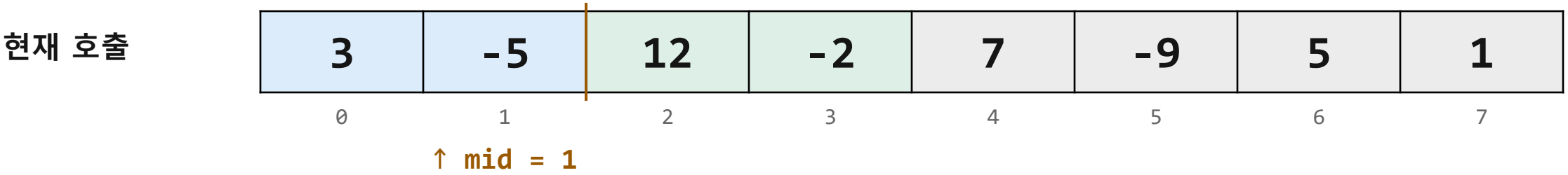



```
left_max  = max_subarray_sum(a, 0, 3)
right_max = max_subarray_sum(a, 4, 7)
```

지금은 전체 여덟 칸의 답을 구하는 호출입니다.

[0~3] 호출: mid = 1에서 나눔

현재 범위 low = 0, high = 3, mid = 1입니다.
먼저 왼쪽 [0~1]를 호출합니다. 오른쪽 [2~3]는 기다립니다.

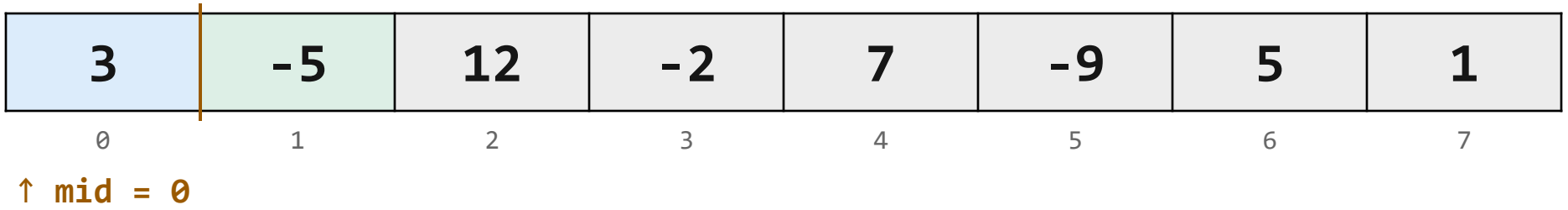



```
left_max  = max_subarray_sum(a, 0, 1)
right_max = max_subarray_sum(a, 2, 3)
```

[0~7] 부모는 left_max의 반환을 기다립니다.

[0~1] 호출: mid = 0에서 나눔

현재 범위 low = 0, high = 1, mid = 0입니다.
먼저 왼쪽 [0~0]를 호출합니다. 오른쪽 [1~1]는 기다립니다.



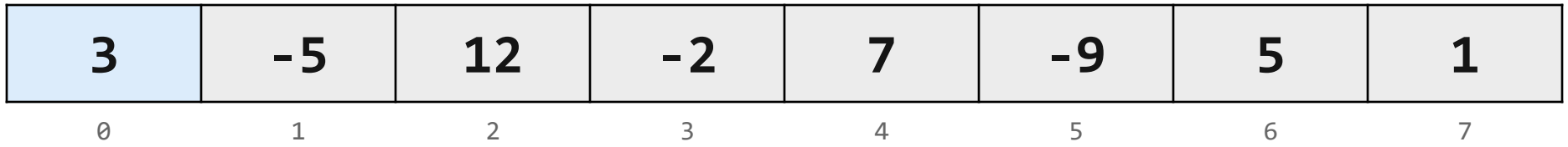

```
left_max  = max_subarray_sum(a, 0, 0)
right_max = max_subarray_sum(a, 1, 1)
```

[0~3] 부모는 left_max의 반환을 기다립니다.

[0~0] 원소 하나: 3 반환

low = high = 0입니다. 남은 원소는 a[0] = 3 하나입니다.
이 값 3을 [0~1] 호출의 left_max에 돌려줍니다.

현재 호출



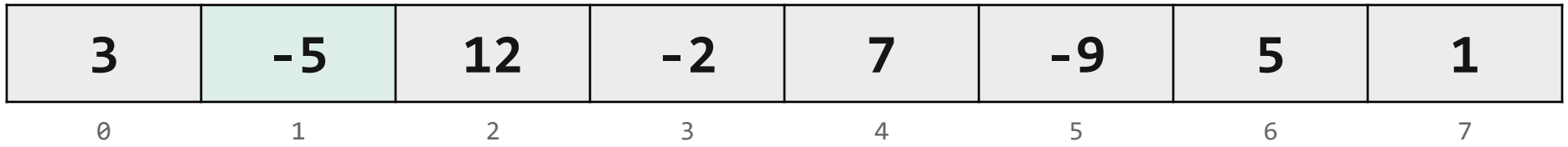

```
if low == high:
    return a[low]     # 3
```

[0~1] 호출의 left_max = 3. 원소 한 개의 호출이 끝났습니다.

[1~1] 원소 하나: −5 반환

low = high = 1입니다. 남은 원소는 a[1] = −5 하나입니다.
이 값 −5을 [0~1] 호출의 right_max에 돌려줍니다.

현재 호출



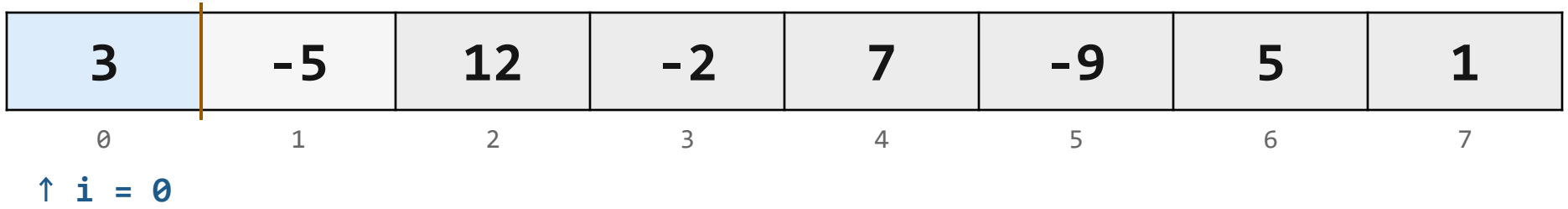

```
if low == high:
    return a[low]     # -5
```

[0~1] 호출의 right_max = −5. 원소 한 개의 호출이 끝났습니다.

[0~1] 왼쪽 스캔 1/1

i = 0, a[0] = 3을 더합니다. 현재 구간은 [0~0]입니다.
현재 합 3 > 이전 기록 −∞ → left_sum 갱신.

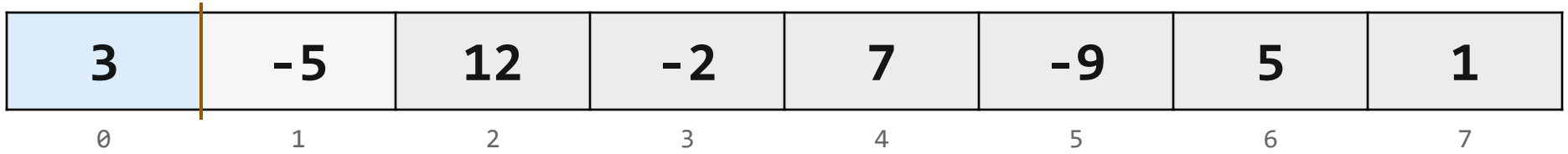
현재 구간



current_sum = 0 + (3) = 3

left_sum: −∞ → 3

저장한 최선

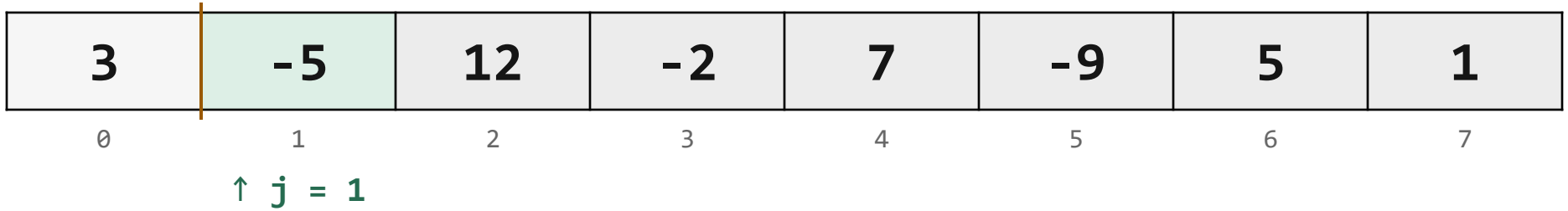


왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

[0~1] 오른쪽 스캔 1/1

j = 1, a[1] = −5을 더합니다. 현재 구간은 [1~1]입니다.
현재 합 −5 > 이전 기록 −∞ → right_sum 갱신.

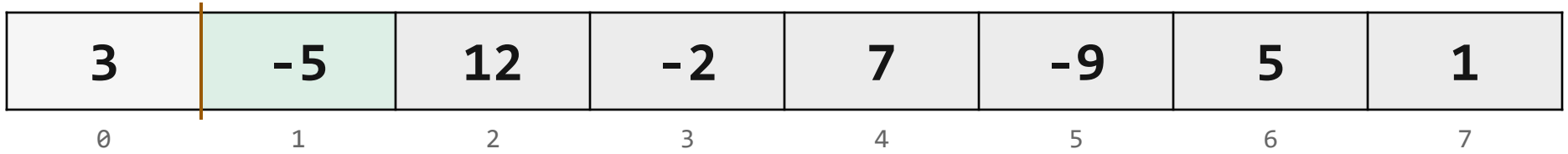
현재 구간




```
current_sum = 0 + (-5) = -5
```

```
right_sum: -∞ → -5
```

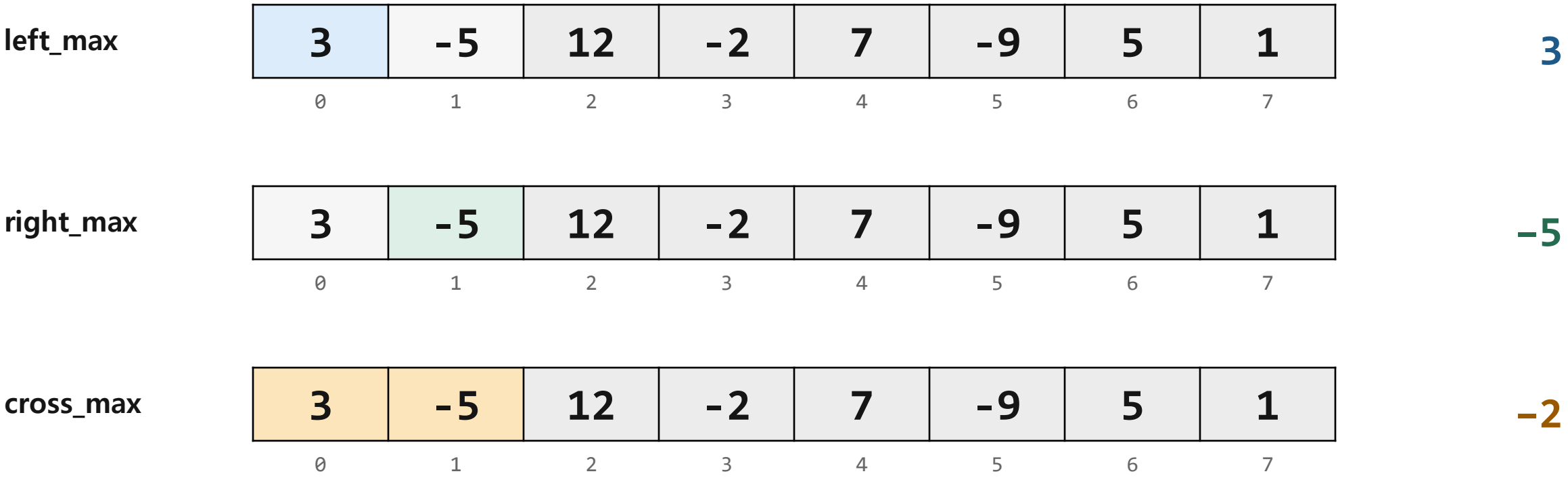
저장한 최선



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[0~1] 세 후보 비교 → 3 반환

왼쪽 3, 오른쪽 −5, 경계를 걸친 합 −2을 비교합니다.
max(3, −5, −2) = 3입니다.

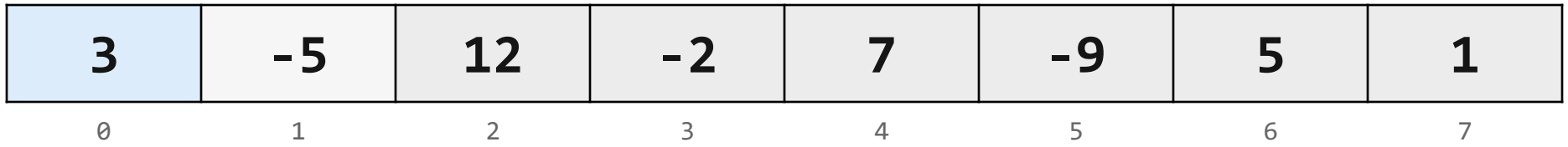


[0~3] 부모의 left_max에 3을 반환합니다.

[0~3] 왼쪽 답 3을 받음

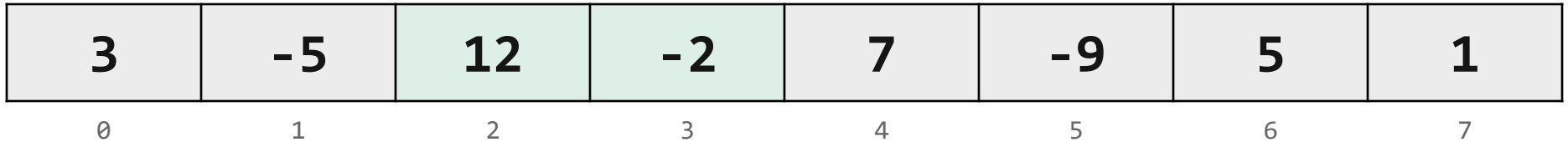
왼쪽 [0~1] 호출이 3을 반환했습니다. left_max = 3입니다.
이제 오른쪽 [2~3] 호출을 시작합니다.

left_max = 3



다음 호출: `max_subarray_sum(a, 2, 3)`

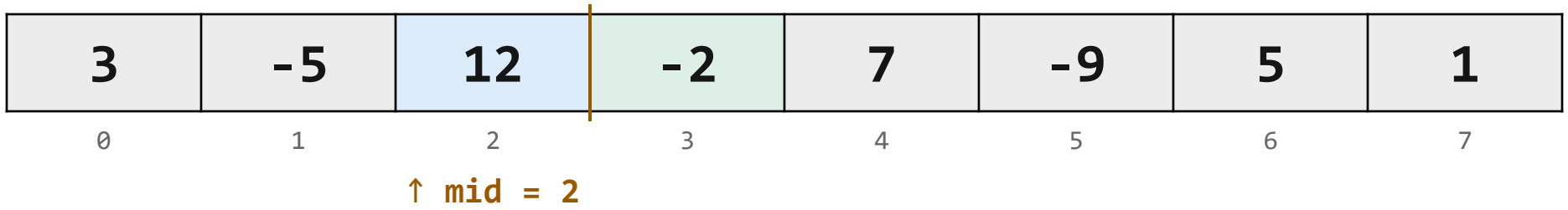
다음 오른쪽



left_max를 저장해 두고 오른쪽 호출이 끝날 때까지 기다립니다.

[2~3] 호출: mid = 2에서 나눔

현재 범위 low = 2, high = 3, mid = 2입니다.
먼저 왼쪽 [2~2]를 호출합니다. 오른쪽 [3~3]는 기다립니다.



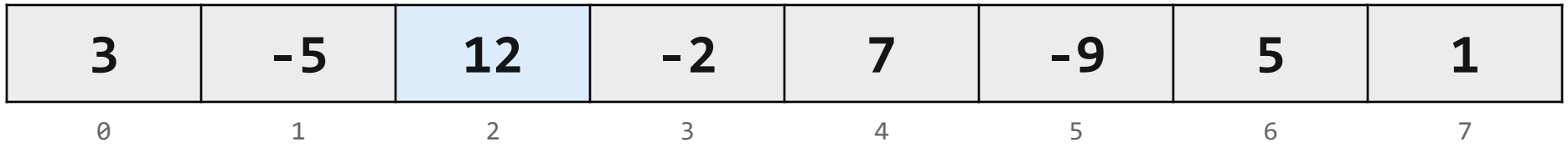

```
left_max  = max_subarray_sum(a, 2, 2)
right_max = max_subarray_sum(a, 3, 3)
```

[0~3] 부모는 right_max의 반환을 기다립니다.

[2~2] 원소 하나: 12 반환

low = high = 2입니다. 남은 원소는 a[2] = 12 하나입니다.
이 값 12을 [2~3] 호출의 left_max에 돌려줍니다.

현재 호출



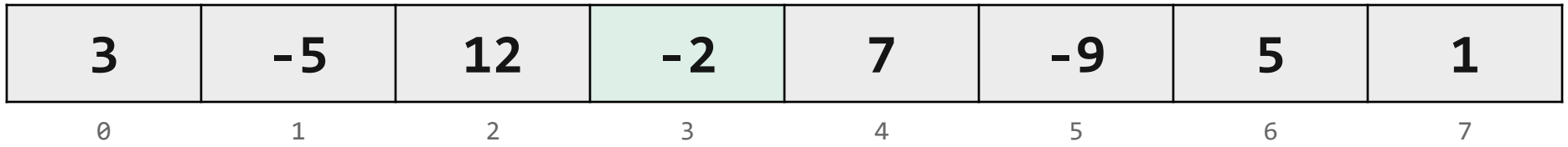

```
if low == high:
    return a[low]     # 12
```

[2~3] 호출의 left_max = 12. 원소 한 개의 호출이 끝났습니다.

[3~3] 원소 하나: −2 반환

low = high = 3입니다. 남은 원소는 a[3] = −2 하나입니다.
이 값 −2을 [2~3] 호출의 right_max에 돌려줍니다.

현재 호출

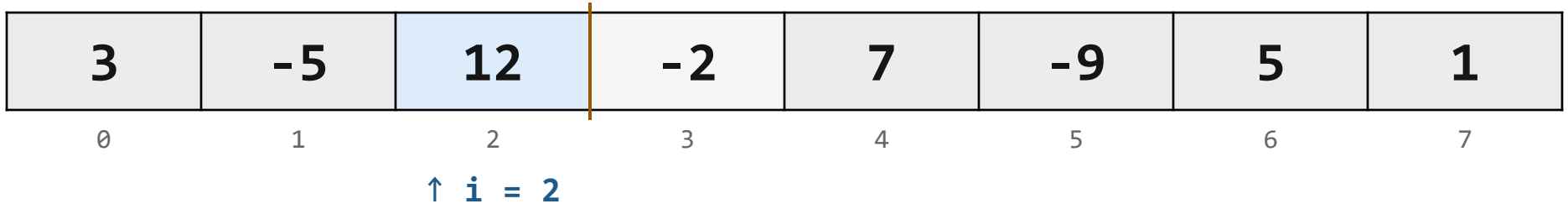



```
if low == high:
    return a[low]     # -2
```

[2~3] 호출의 right_max = −2. 원소 한 개의 호출이 끝났습니다.

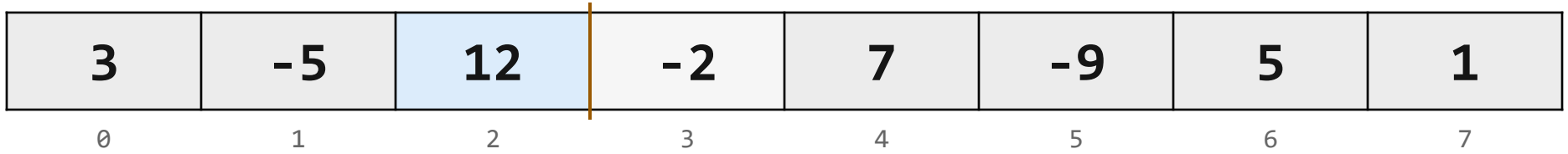
[2~3] 왼쪽 스캔 1/1

i = 2, a[2] = 12을 더합니다. 현재 구간은 [2~2]입니다.
현재 합 12 > 이전 기록 −∞ → left_sum 갱신.




```
current_sum = 0 + (12) = 12
```

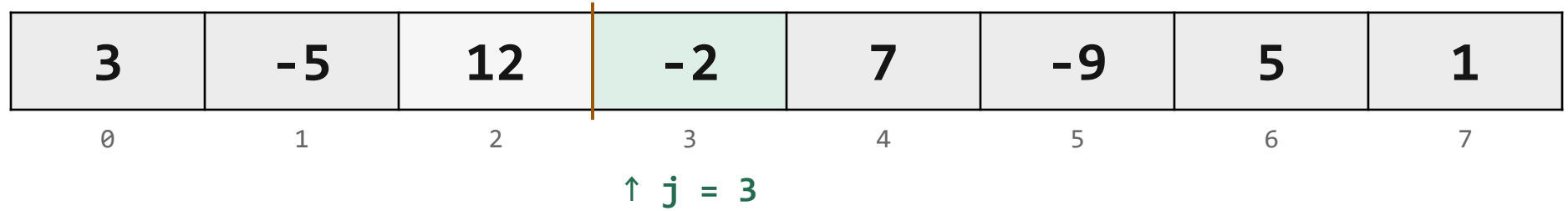
```
left_sum: -∞ → 12
```



왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

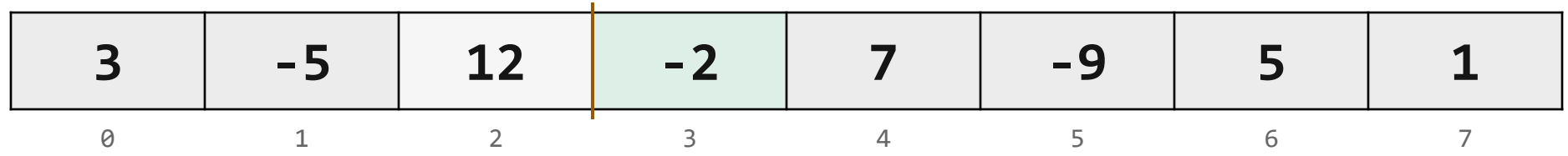
[2~3] 오른쪽 스캔 1/1

j = 3, a[3] = −2을 더합니다. 현재 구간은 [3~3]입니다.
현재 합 −2 > 이전 기록 −∞ → right_sum 갱신.




```
current_sum = 0 + (-2) = -2
```

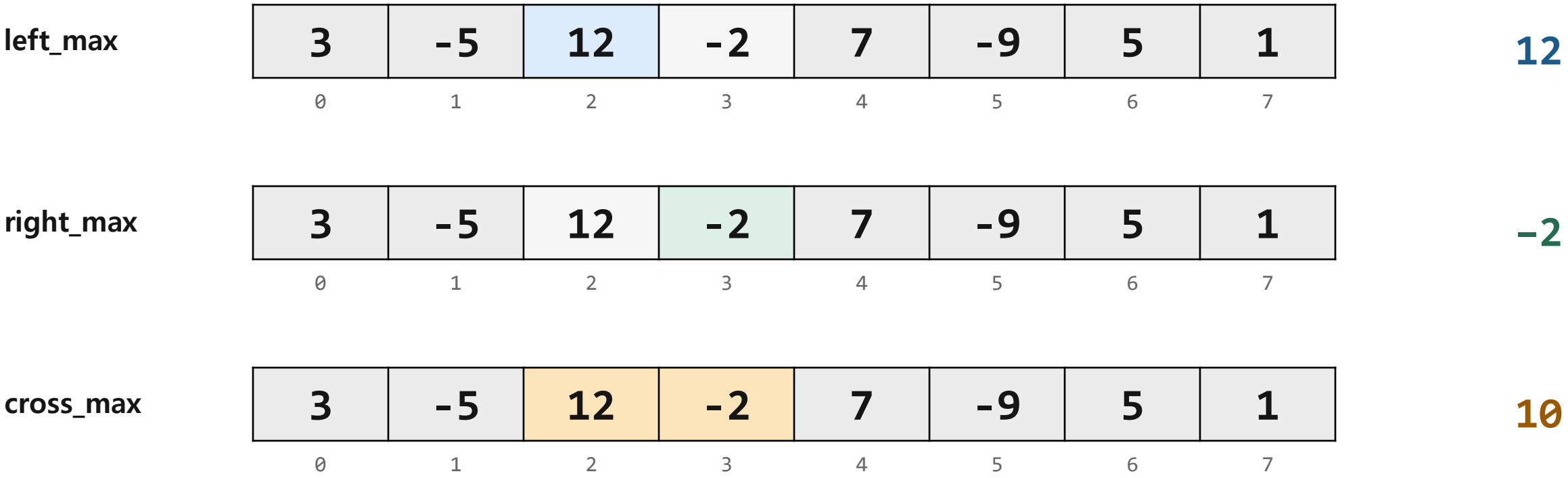
```
right_sum: -∞ → -2
```



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[2~3] 세 후보 비교 → 12 반환

왼쪽 12, 오른쪽 −2, 경계를 걸친 합 10을 비교합니다.
max(12, −2, 10) = 12입니다.

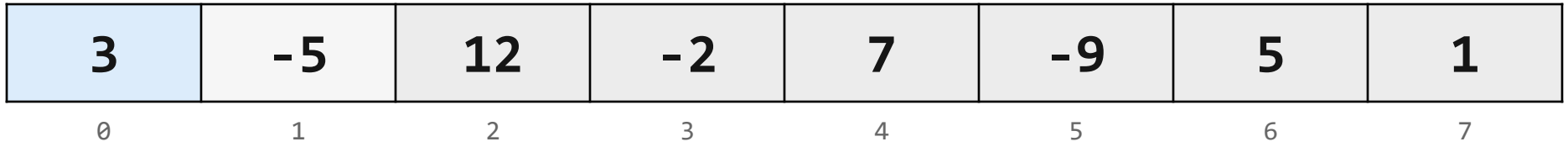


[0~3] 부모의 right_max에 12을 반환합니다.

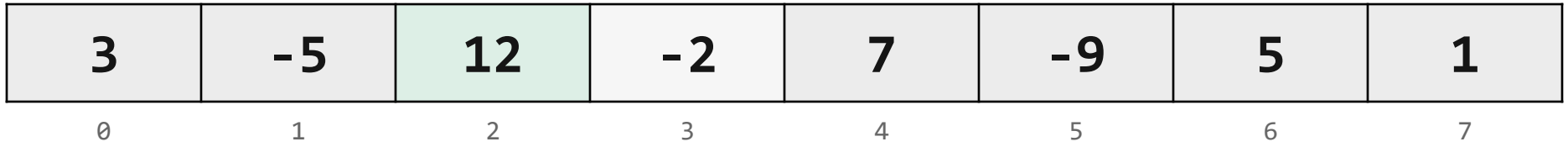
[0~3] 오른쪽 답 12도 받음

left_max = 3, right_max = 12입니다. 두 재귀 호출을 마쳤습니다.
아직 경계를 걸치는 cross_max를 계산하고 비교해야 합니다.

left_max = 3



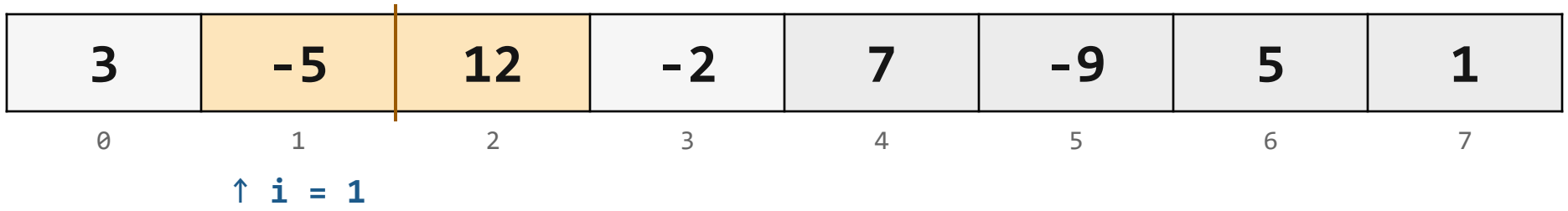
right_max = 12



다음은 경계를 걸치는 후보입니다. left_max + right_max를 계산하지 않습니다.

[0~3] 이제 경계를 걸치는 합 계산

mid = 1입니다. 왼쪽 스캔은 a[1] = −5부터 시작합니다.
current_sum = 0, left_sum = −∞로 두고 i를 왼쪽으로 움직입니다.



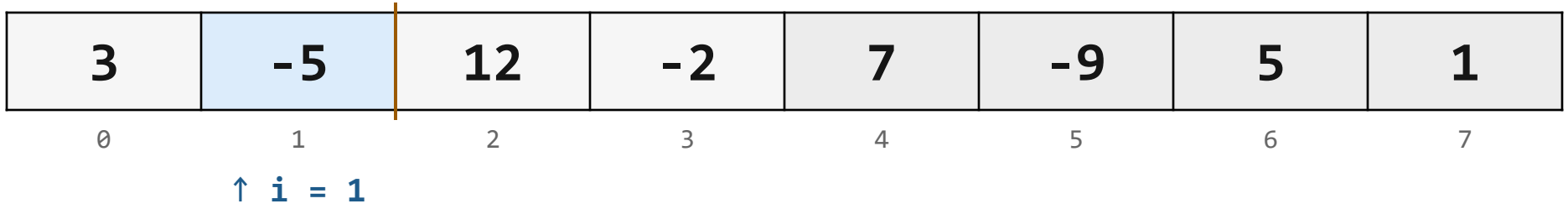

```
current_sum = 0        left_sum = -∞
```

재귀 답은 유지: left_max = 3, right_max = 12.

[0~3] 왼쪽 스캔 1/2

i = 1, a[1] = −5을 더합니다. 현재 구간은 [1~1]입니다.
현재 합 −5 > 이전 기록 −∞ → left_sum 갱신.

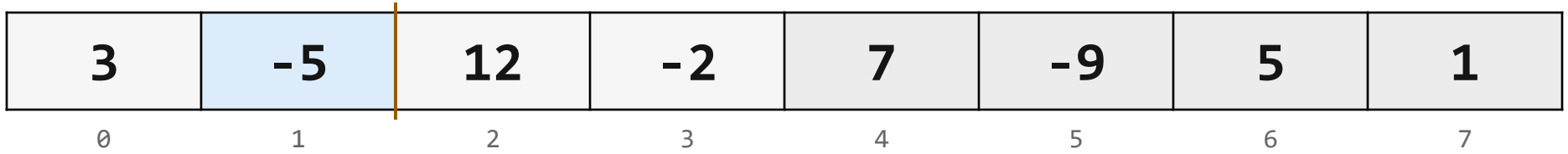
현재 구간




```
current_sum = 0 + (-5) = -5
```

```
left_sum: -∞ → -5
```

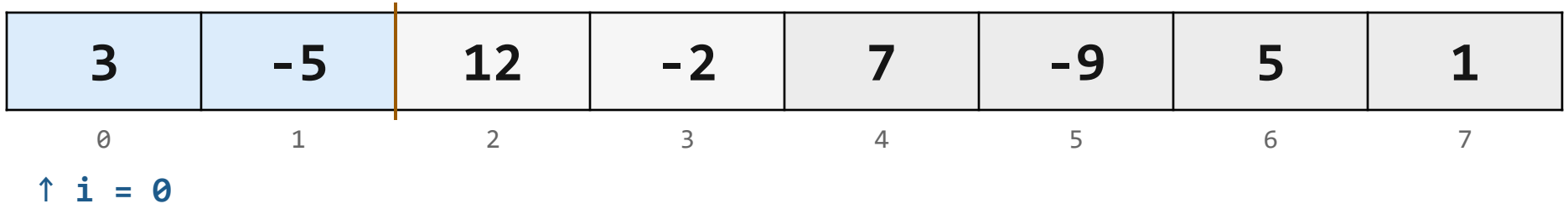
저장한 최선



다음 i = 0: a[0] = 3을 현재 합 −5에 더합니다.

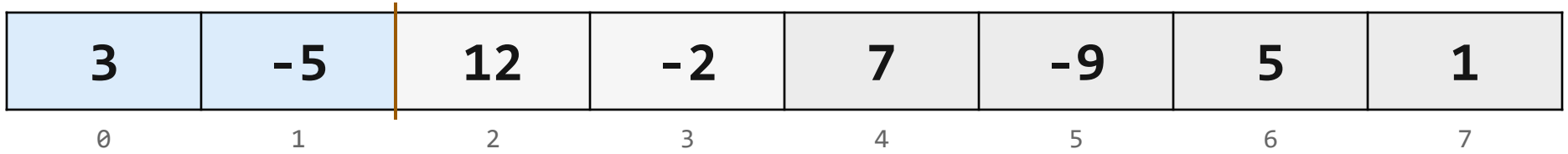
[0~3] 왼쪽 스캔 2/2

i = 0, a[0] = 3을 더합니다. 현재 구간은 [0~1]입니다.
현재 합 −2 > 이전 기록 −5 → left_sum 갱신.




```
current_sum = -5 + (3) = -2
```

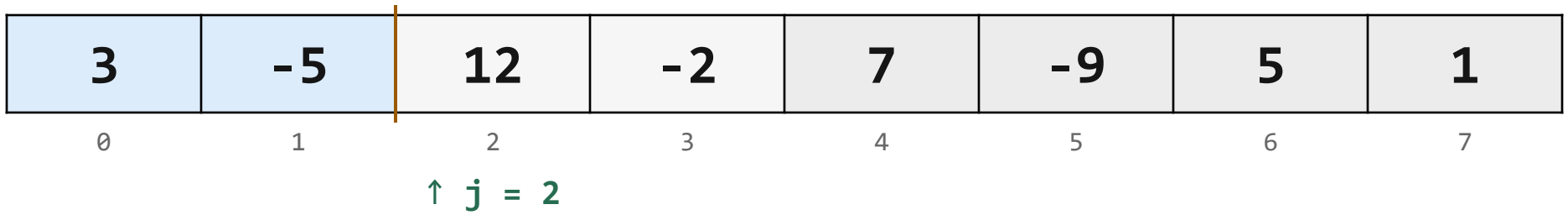
```
left_sum: -5 → -2
```



왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

[0~3] 오른쪽은 합을 0으로 다시 시작

왼쪽 기록 left_sum = −2은 보관합니다. current_sum만 0으로 바꿉니다.
right_sum = −∞로 두고 j = mid + 1 = 2부터 오른쪽을 훑습니다.




```
current_sum = 0

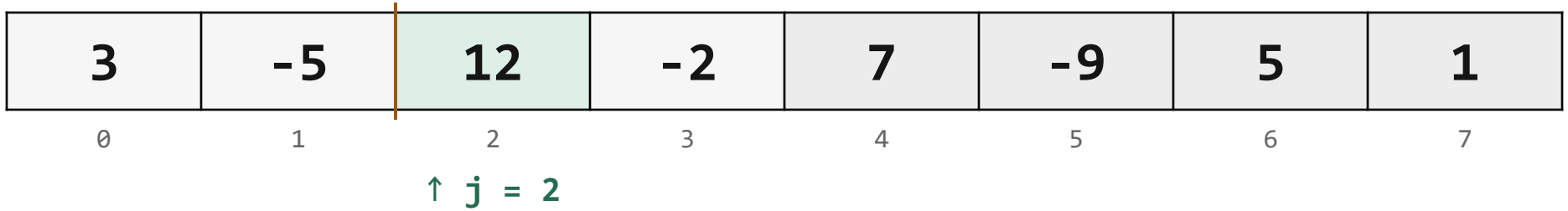
left_sum = -2 (보관)                    right_sum = -∞
```

오른쪽 첫 칸은 a[2] = 12입니다. mid를 다시 읽지 않습니다.

[0~3] 오른쪽 스캔 1/2

j = 2, a[2] = 12을 더합니다. 현재 구간은 [2~2]입니다.
현재 합 12 > 이전 기록 −∞ → right_sum 갱신.

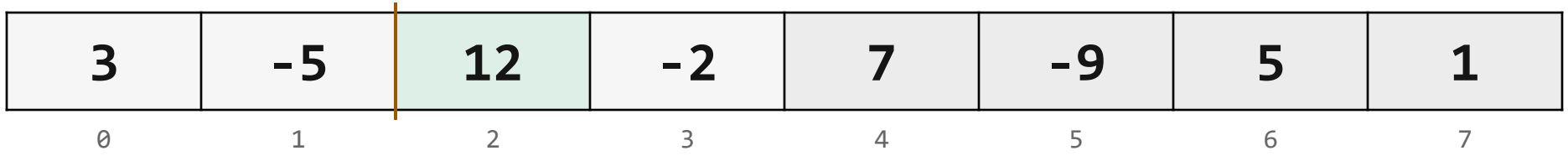
현재 구간




```
current_sum = 0 + (12) = 12
```

```
right_sum: -∞ → 12
```

저장한 최선

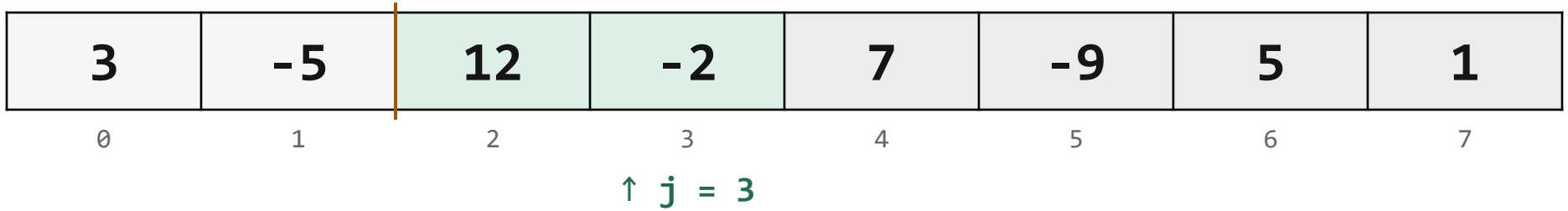


다음 j = 3: a[3] = −2을 현재 합 12에 더합니다.

[0~3] 오른쪽 스캔 2/2

j = 3, a[3] = −2을 더합니다. 현재 구간은 [2~3]입니다.
현재 합 10 ≤ 이전 기록 12 → right_sum 유지.

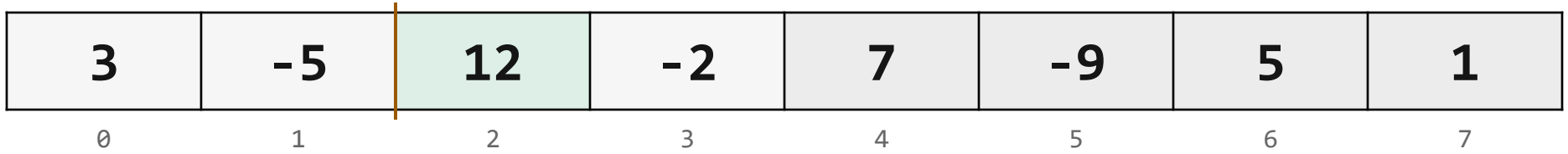
현재 구간



current_sum = 12 + (-2) = 10

right_sum = 12 유지

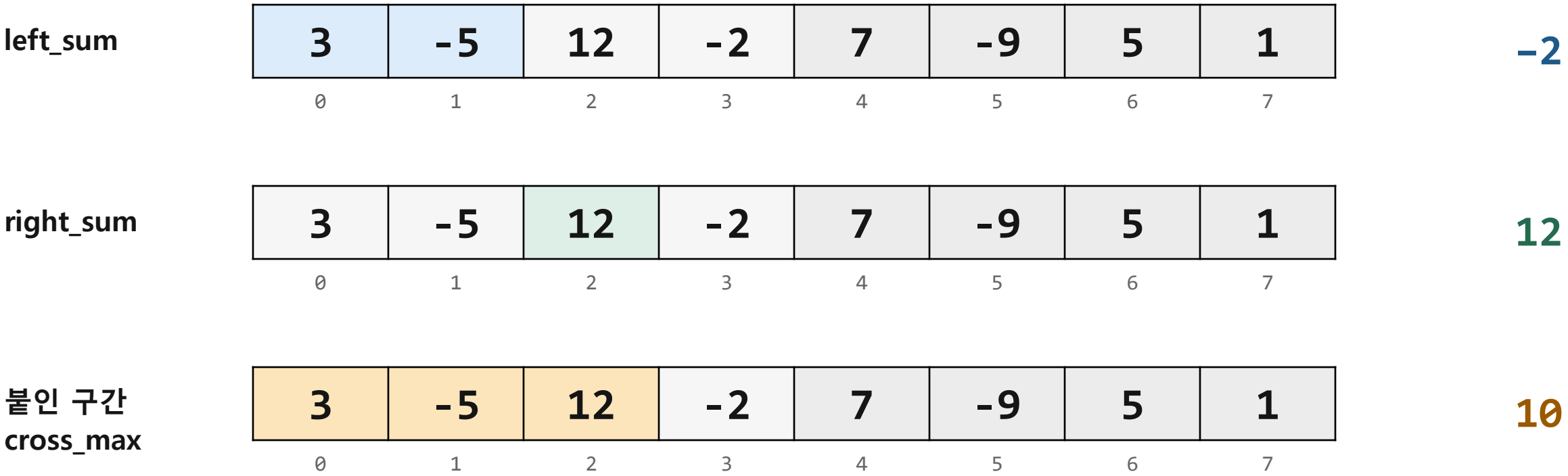
저장한 최선



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[0~3] 경계에서 두 기록을 붙입니다

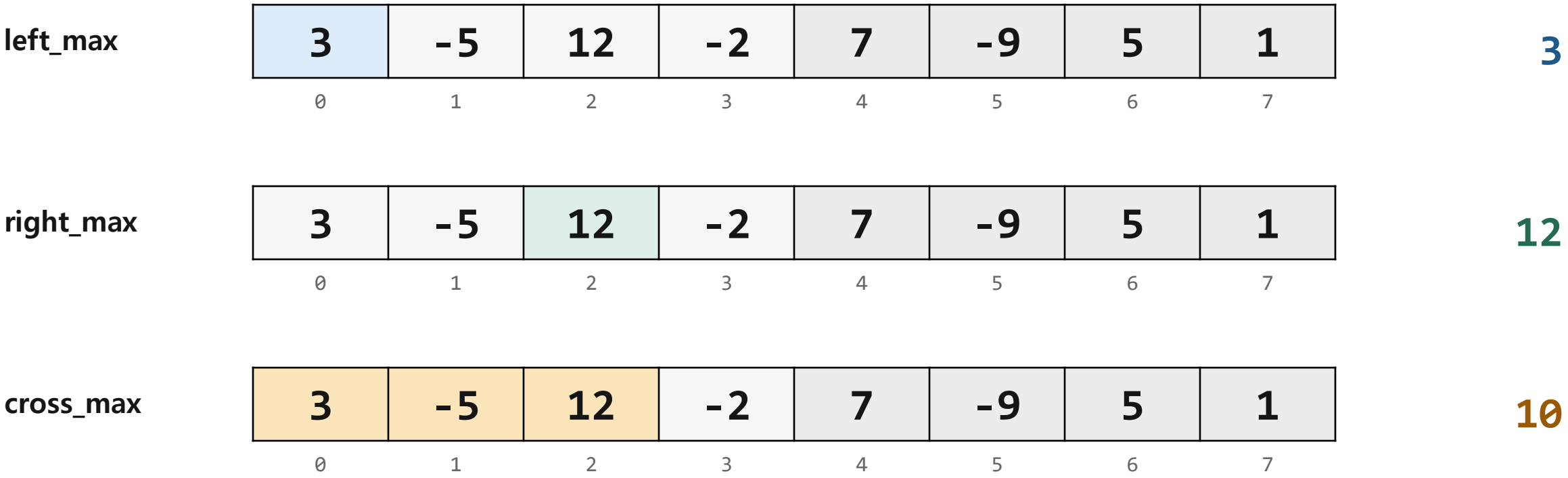
left_sum = −2, right_sum = 12이므로 cross_max = 10입니다.
두 구간은 인덱스 1와 2에서 맞닿아 빈틈 없이 이어집니다.



−2 + (12) = 10. 마지막에 읽은 합이 아니라 최선의 기록을 더합니다.

[0~3] 세 후보 비교 → 12 반환

왼쪽 3, 오른쪽 12, 경계를 걸친 합 10을 비교합니다.
max(3, 12, 10) = 12입니다.

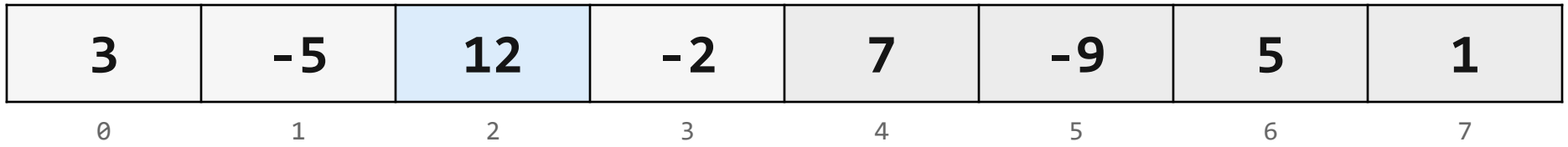


[0~7] 부모의 left_max에 12을 반환합니다.

[0~7] 왼쪽 답 12을 받음

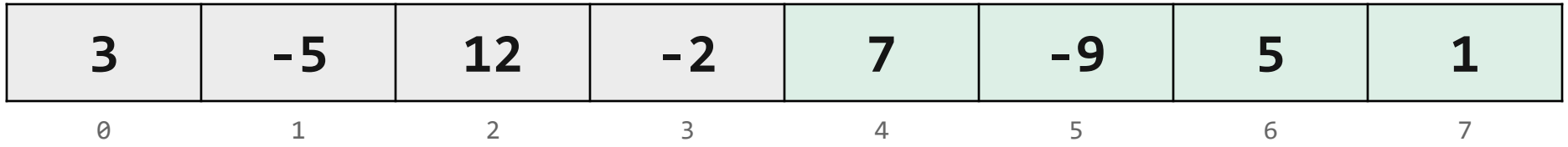
왼쪽 [0~3] 호출이 12을 반환했습니다. left_max = 12입니다.
이제 오른쪽 [4~7] 호출을 시작합니다.

left_max = 12



다음 호출: `max_subarray_sum(a, 4, 7)`

다음 오른쪽

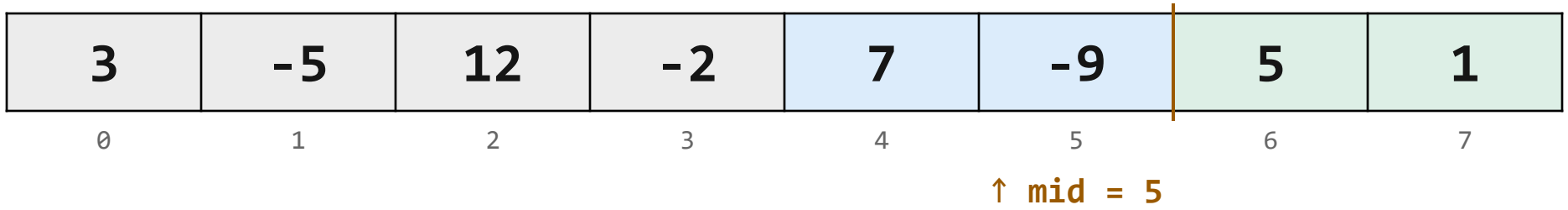


left_max를 저장해 두고 오른쪽 호출이 끝날 때까지 기다립니다.

[4~7] 호출: mid = 5에서 나눔

현재 범위 low = 4, high = 7, mid = 5입니다.
먼저 왼쪽 [4~5]를 호출합니다. 오른쪽 [6~7]는 기다립니다.

현재 호출

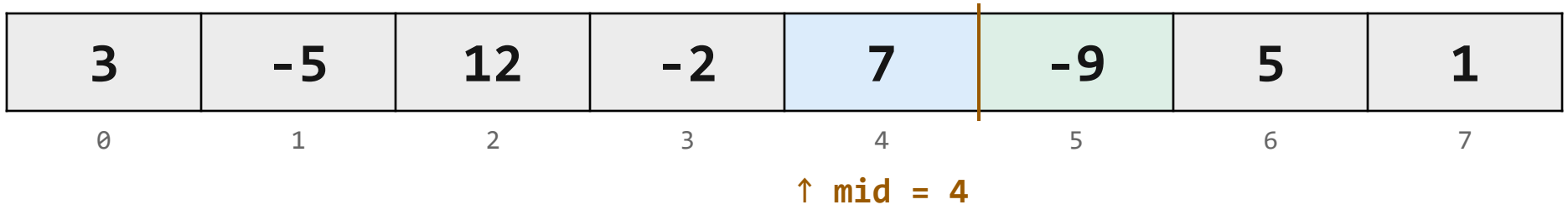



```
left_max  = max_subarray_sum(a, 4, 5)
right_max = max_subarray_sum(a, 6, 7)
```

[0~7] 부모는 right_max의 반환을 기다립니다.

[4~5] 호출: mid = 4에서 나눔

현재 범위 low = 4, high = 5, mid = 4입니다.
먼저 왼쪽 [4~4]를 호출합니다. 오른쪽 [5~5]는 기다립니다.



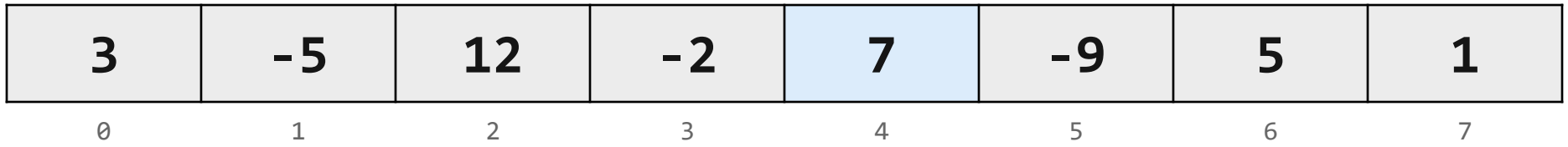

```
left_max  = max_subarray_sum(a, 4, 4)
right_max = max_subarray_sum(a, 5, 5)
```

[4~7] 부모는 left_max의 반환을 기다립니다.

[4~4] 원소 하나: 7 반환

low = high = 4입니다. 남은 원소는 a[4] = 7 하나입니다.
이 값 7을 [4~5] 호출의 left_max에 돌려줍니다.

현재 호출



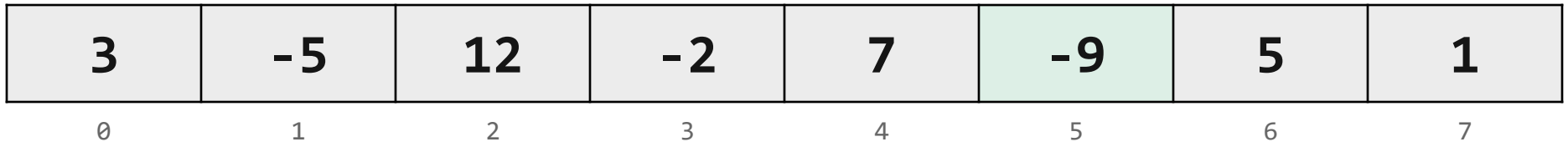

```
if low == high:
    return a[low]     # 7
```

[4~5] 호출의 left_max = 7. 원소 한 개의 호출이 끝났습니다.

[5~5] 원소 하나: −9 반환

low = high = 5입니다. 남은 원소는 a[5] = −9 하나입니다.
이 값 −9을 [4~5] 호출의 right_max에 돌려줍니다.

현재 호출

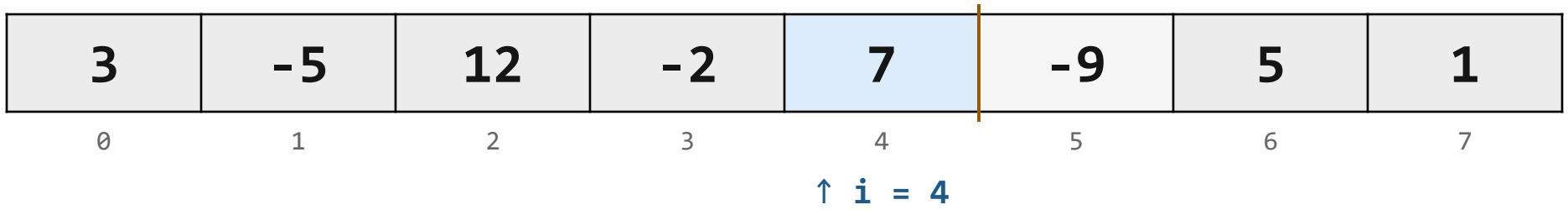



```
if low == high:
    return a[low]     # -9
```

[4~5] 호출의 right_max = −9. 원소 한 개의 호출이 끝났습니다.

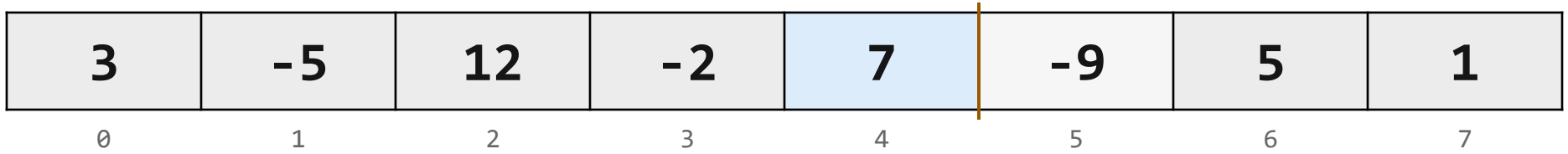
[4~5] 왼쪽 스캔 1/1

i = 4, a[4] = 7을 더합니다. 현재 구간은 [4~4]입니다.
현재 합 7 > 이전 기록 −∞ → left_sum 갱신.




```
current_sum = 0 + (7) = 7
```

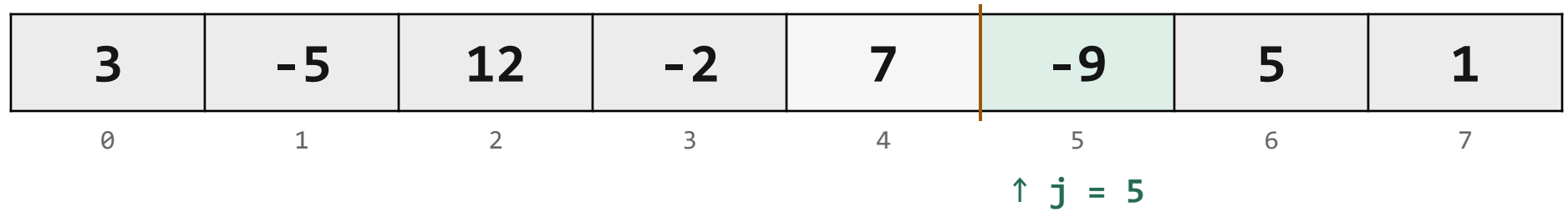
```
left_sum: -∞ → 7
```



왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

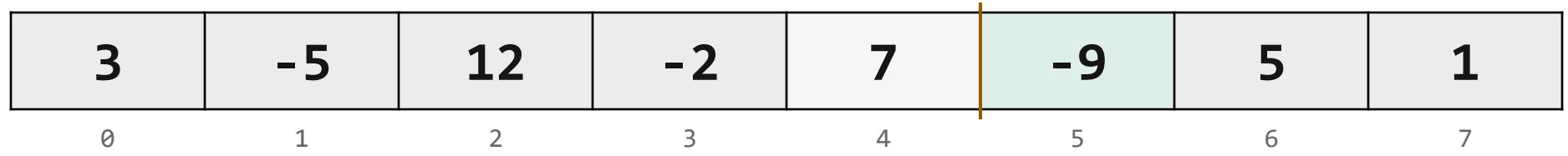
[4~5] 오른쪽 스캔 1/1

j = 5, a[5] = −9을 더합니다. 현재 구간은 [5~5]입니다.
현재 합 −9 > 이전 기록 −∞ → right_sum 갱신.




```
current_sum = 0 + (-9) = -9
```

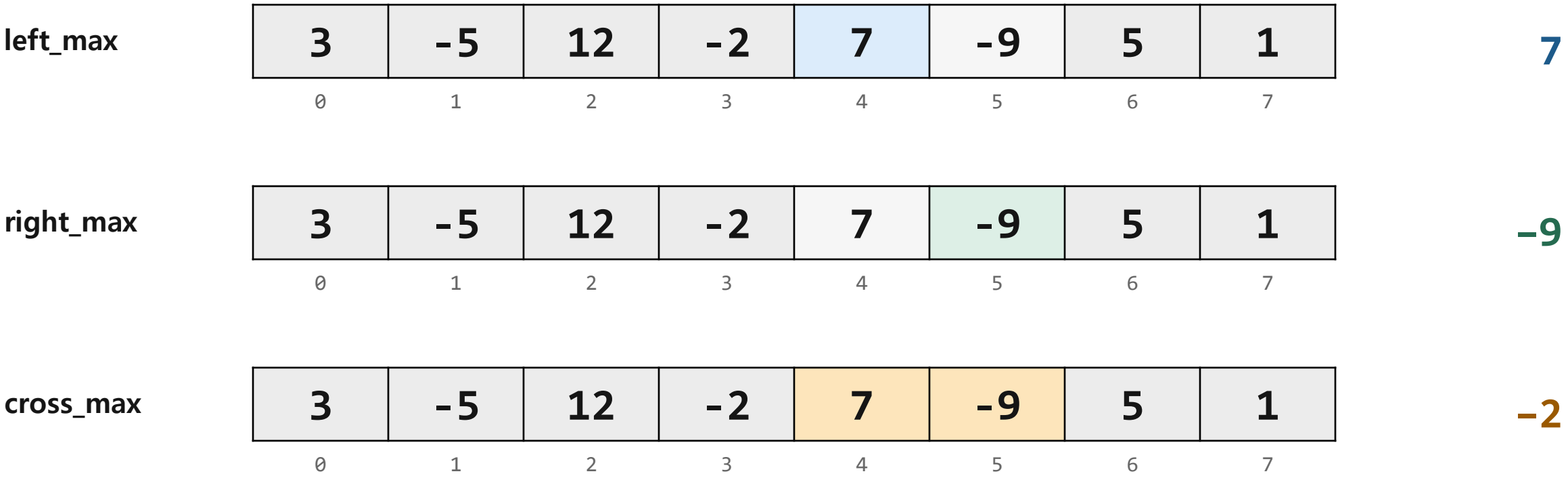
```
right_sum: -∞ → -9
```



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[4~5] 세 후보 비교 → 7 반환

왼쪽 7, 오른쪽 −9, 경계를 걸친 합 −2을 비교합니다.
max(7, −9, −2) = 7입니다.

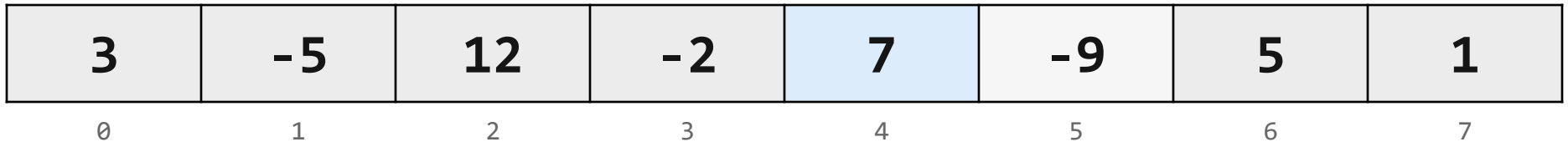


[4~7] 부모의 left_max에 7을 반환합니다.

[4~7] 왼쪽 답 7을 받음

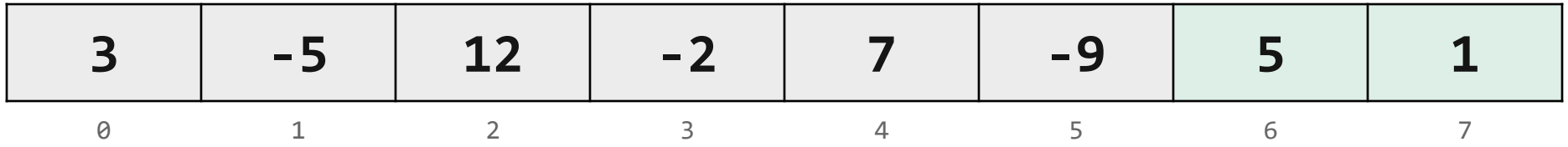
왼쪽 [4~5] 호출이 7을 반환했습니다. left_max = 7입니다.
이제 오른쪽 [6~7] 호출을 시작합니다.

left_max = 7



다음 호출: `max_subarray_sum(a, 6, 7)`

다음 오른쪽

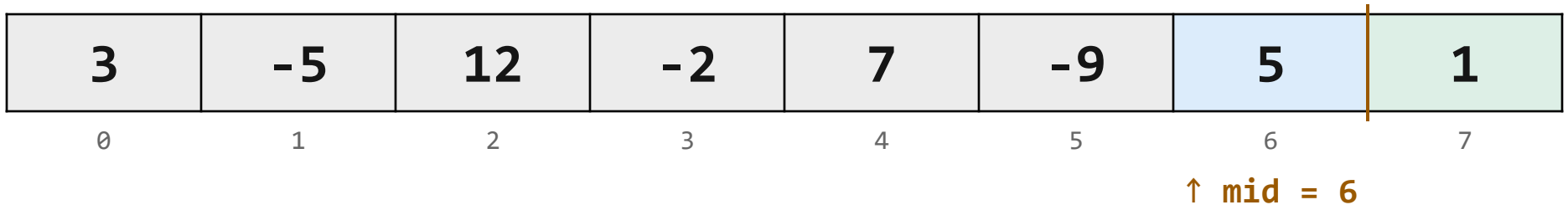


left_max를 저장해 두고 오른쪽 호출이 끝날 때까지 기다립니다.

[6~7] 호출: mid = 6에서 나눔

현재 범위 low = 6, high = 7, mid = 6입니다.
먼저 왼쪽 [6~6]를 호출합니다. 오른쪽 [7~7]는 기다립니다.

현재 호출



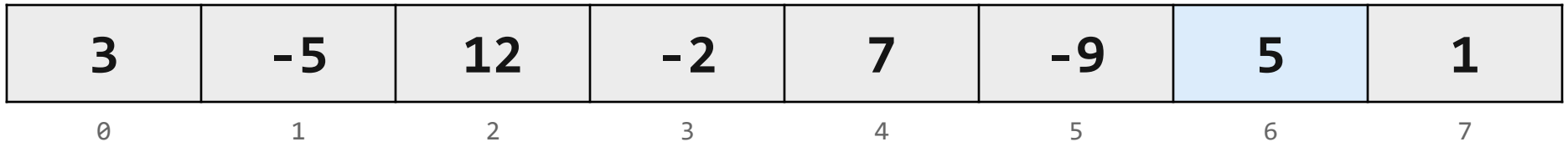

```
left_max  = max_subarray_sum(a, 6, 6)
right_max = max_subarray_sum(a, 7, 7)
```

[4~7] 부모는 right_max의 반환을 기다립니다.

[6~6] 원소 하나: 5 반환

low = high = 6입니다. 남은 원소는 a[6] = 5 하나입니다.
이 값 5을 [6~7] 호출의 left_max에 돌려줍니다.

현재 호출



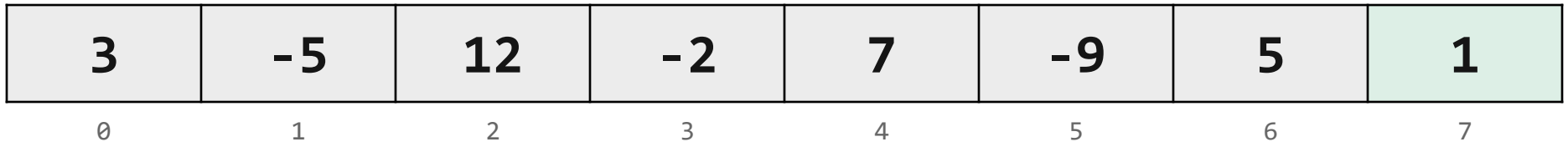

```
if low == high:
    return a[low]     # 5
```

[6~7] 호출의 left_max = 5. 원소 한 개의 호출이 끝났습니다.

[7~7] 원소 하나: 1 반환

low = high = 7입니다. 남은 원소는 a[7] = 1 하나입니다.
이 값 1을 [6~7] 호출의 right_max에 돌려줍니다.

현재 호출



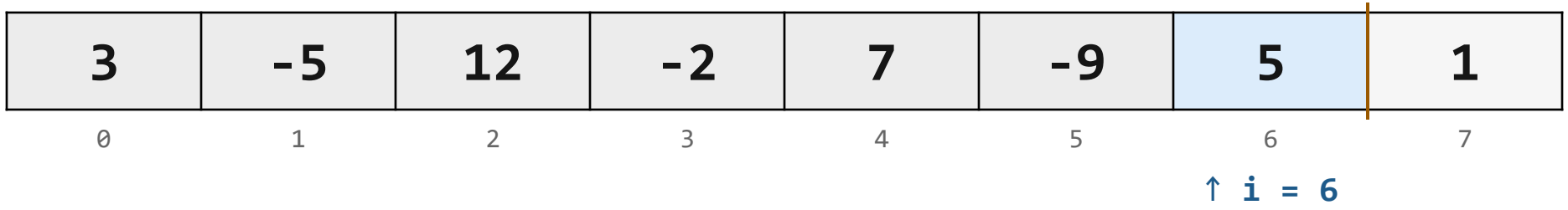

```
if low == high:
    return a[low]     # 1
```

[6~7] 호출의 right_max = 1. 원소 한 개의 호출이 끝났습니다.

[6~7] 왼쪽 스캔 1/1

i = 6, a[6] = 5을 더합니다. 현재 구간은 [6~6]입니다.
현재 합 5 > 이전 기록 −∞ → left_sum 갱신.

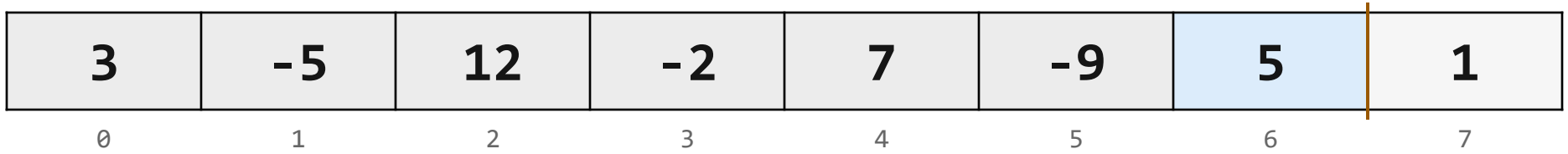
현재 구간



`current_sum = 0 + (5) = 5`

`left_sum: −∞ → 5`

저장한 최선

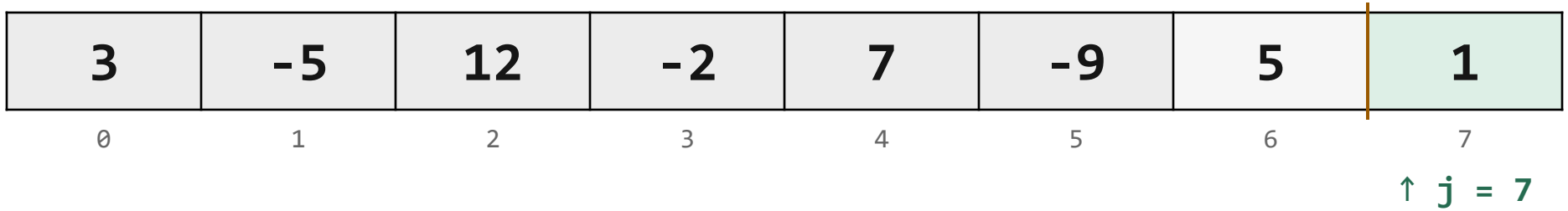


왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

[6~7] 오른쪽 스캔 1/1

j = 7, a[7] = 1을 더합니다. 현재 구간은 [7~7]입니다.
현재 합 1 > 이전 기록 −∞ → right_sum 갱신.

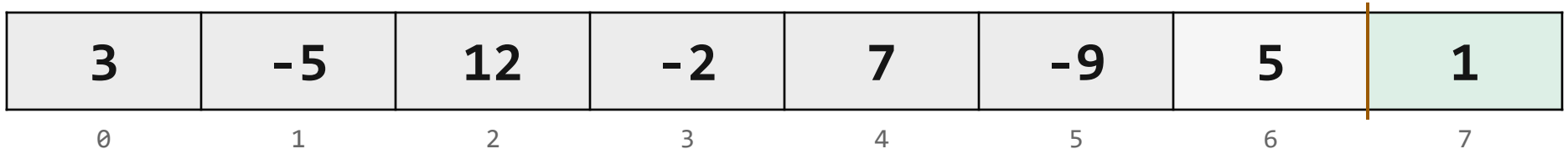
현재 구간



current_sum = 0 + (1) = 1

right_sum: −∞ → 1

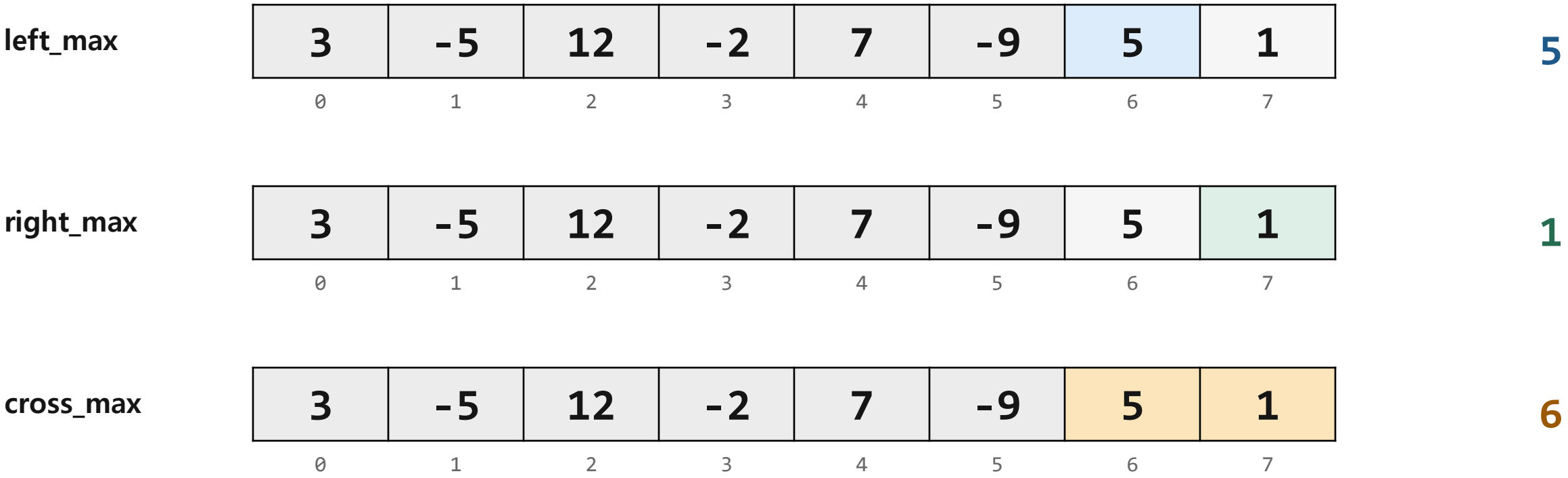
저장한 최선



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[6~7] 세 후보 비교 → 6 반환

왼쪽 5, 오른쪽 1, 경계를 걸친 합 6을 비교합니다.
max(5, 1, 6) = 6입니다.

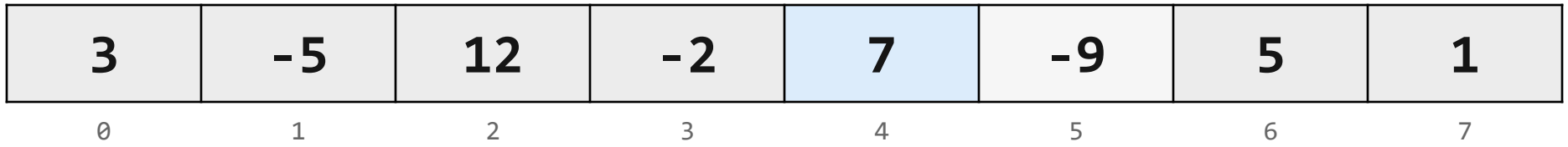


[4~7] 부모의 right_max에 6을 반환합니다.

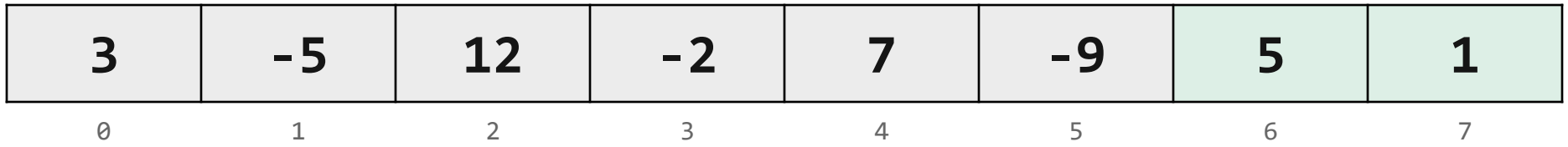
[4~7] 오른쪽 답 6도 받음

left_max = 7, right_max = 6입니다. 두 재귀 호출을 마쳤습니다.
아직 경계를 걸치는 cross_max를 계산하고 비교해야 합니다.

left_max = 7



right_max = 6

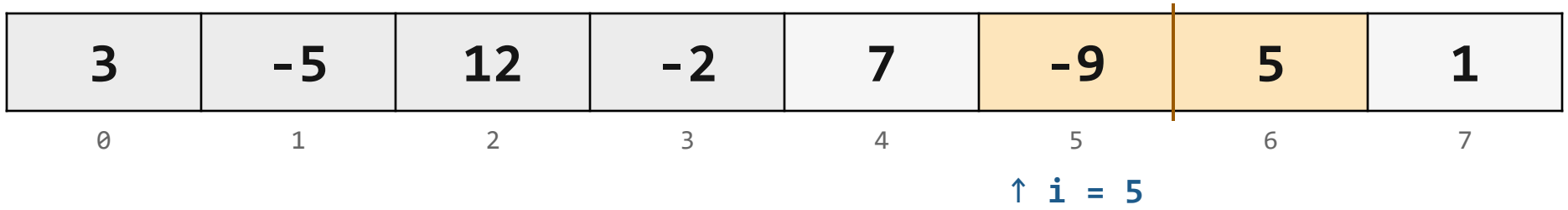


다음은 경계를 걸치는 후보입니다. left_max + right_max를 계산하지 않습니다.

[4~7] 이제 경계를 걸치는 합 계산

mid = 5입니다. 왼쪽 스캔은 a[5] = −9부터 시작합니다.
current_sum = 0, left_sum = −∞로 두고 i를 왼쪽으로 움직입니다.

현재 경계



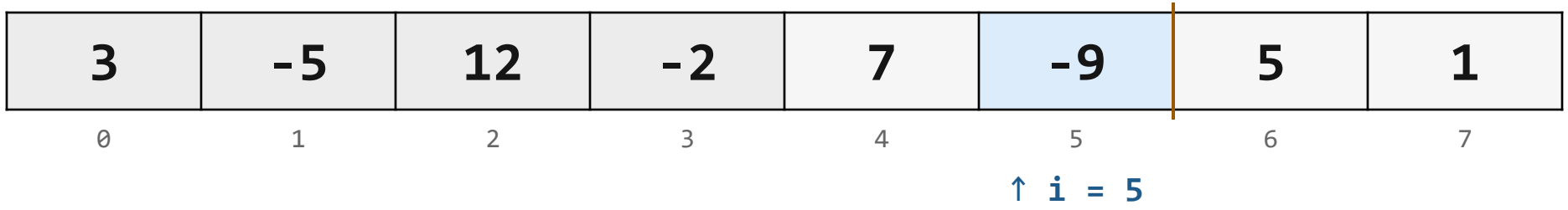

```
current_sum = 0        left_sum = -∞
```

재귀 답은 유지: left_max = 7, right_max = 6.

[4~7] 왼쪽 스캔 1/2

i = 5, a[5] = −9을 더합니다. 현재 구간은 [5~5]입니다.
현재 합 −9 > 이전 기록 −∞ → left_sum 갱신.

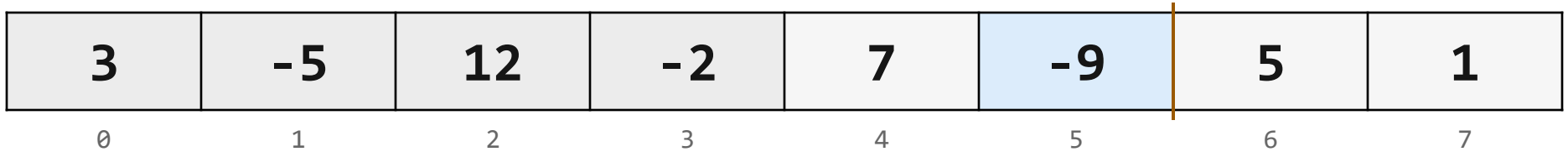
현재 구간




```
current_sum = 0 + (-9) = -9
```

```
left_sum: -∞ → -9
```

저장한 최선

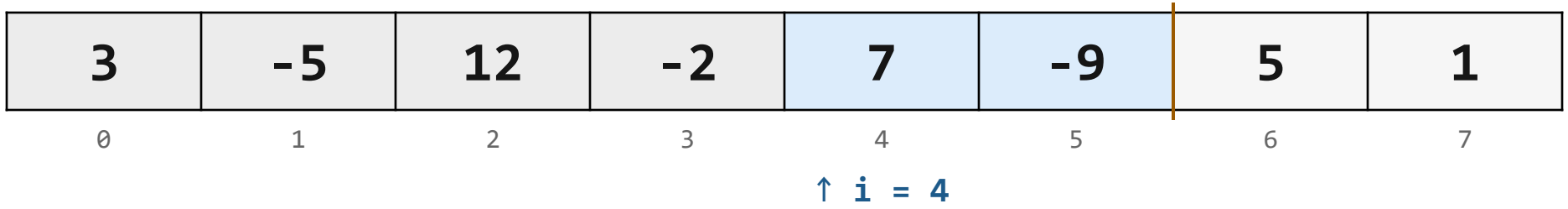


다음 i = 4: a[4] = 7을 현재 합 −9에 더합니다.

[4~7] 왼쪽 스캔 2/2

i = 4, a[4] = 7을 더합니다. 현재 구간은 [4~5]입니다.
현재 합 −2 > 이전 기록 −9 → left_sum 갱신.

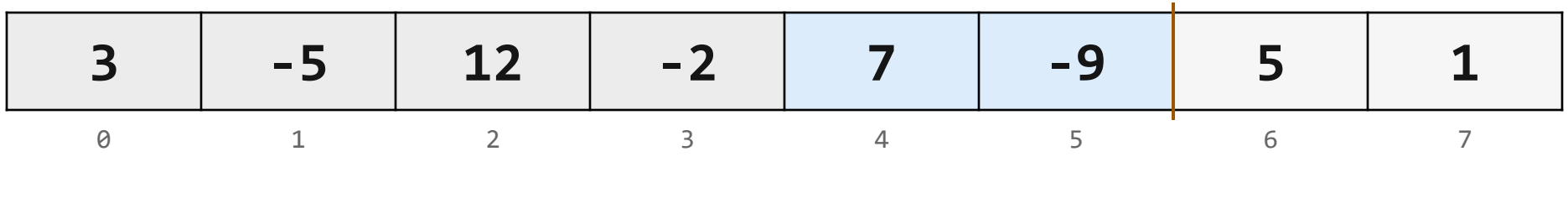
현재 구간



current_sum = -9 + (7) = -2

left_sum: -9 → -2

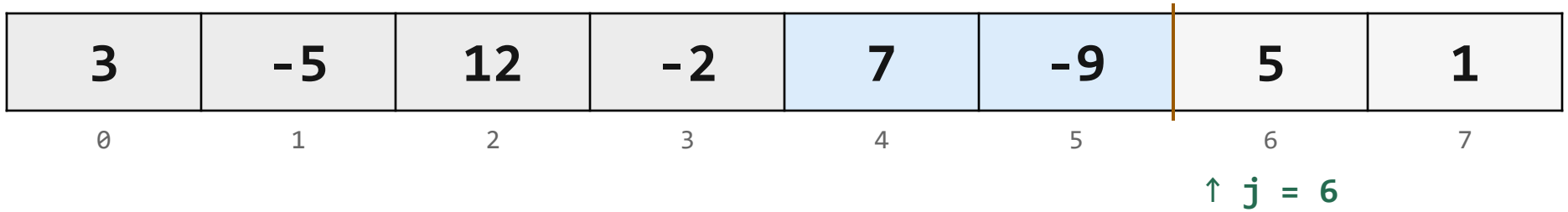
저장한 최선



왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

[4~7] 오른쪽은 합을 0으로 다시 시작

왼쪽 기록 left_sum = −2은 보관합니다. current_sum만 0으로 바꿉니다.
right_sum = −∞로 두고 j = mid + 1 = 6부터 오른쪽을 훑습니다.



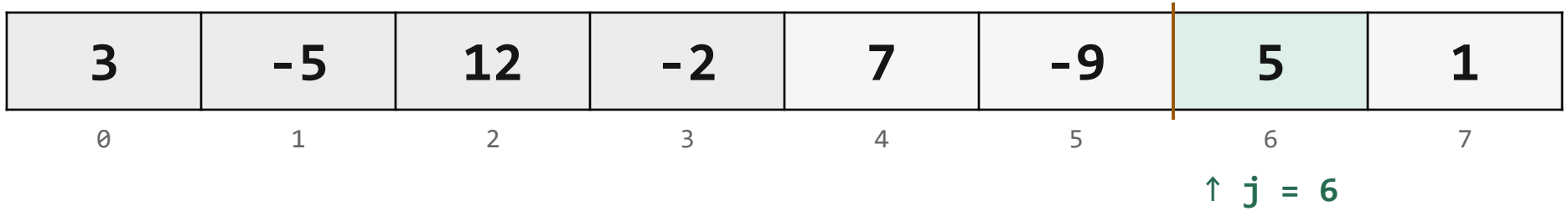

```
current_sum = 0

left_sum = -2 (보관)                    right_sum = -∞
```

오른쪽 첫 칸은 a[6] = 5입니다. mid를 다시 읽지 않습니다.

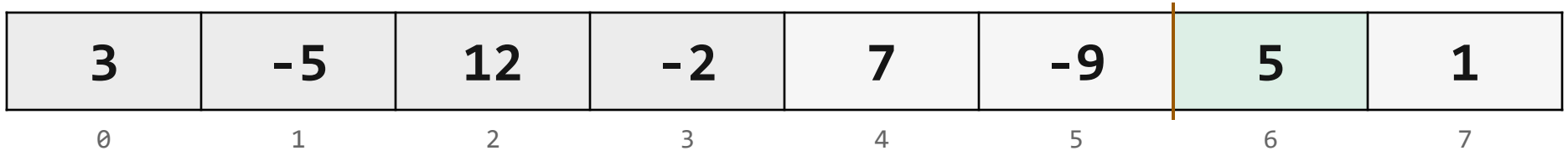
[4~7] 오른쪽 스캔 1/2

j = 6, a[6] = 5을 더합니다. 현재 구간은 [6~6]입니다.
현재 합 5 > 이전 기록 −∞ → right_sum 갱신.




```
current_sum = 0 + (5) = 5
```

```
right_sum: -∞ → 5
```

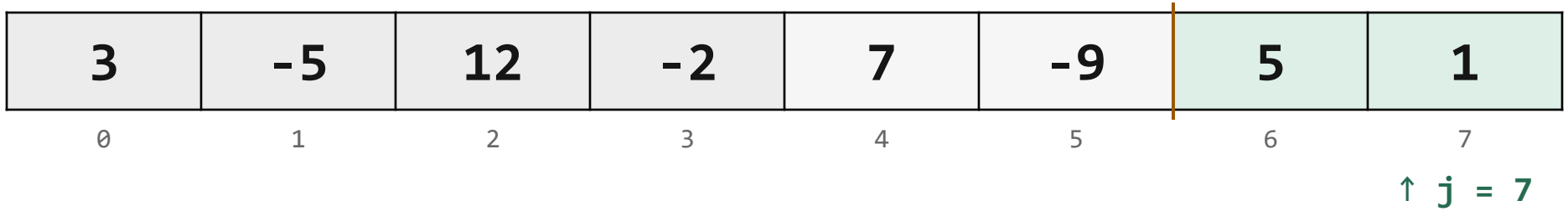


다음 j = 7: a[7] = 1을 현재 합 5에 더합니다.

[4~7] 오른쪽 스캔 2/2

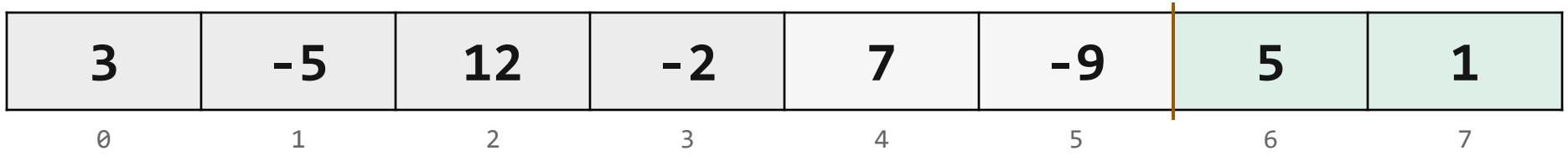
j = 7, a[7] = 1을 더합니다. 현재 구간은 [6~7]입니다.
현재 합 6 > 이전 기록 5 → right_sum 갱신.

현재 구간



current_sum = 5 + (1) = 6 **right_sum: 5 → 6**

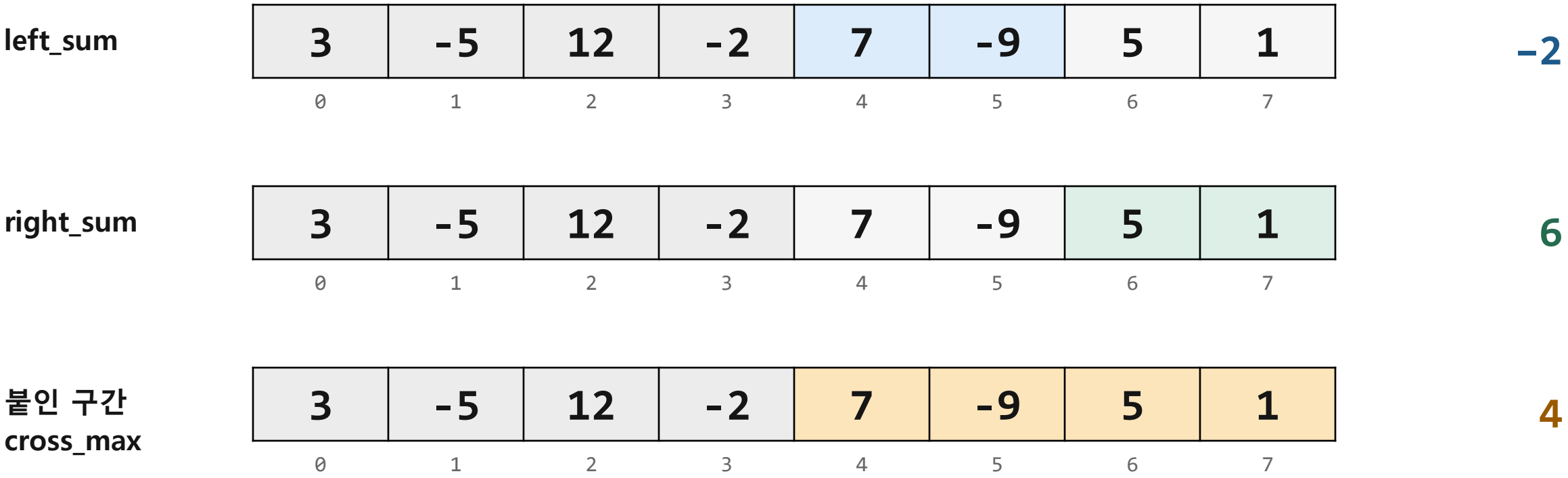
저장한 최선



오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[4~7] 경계에서 두 기록을 붙입니다

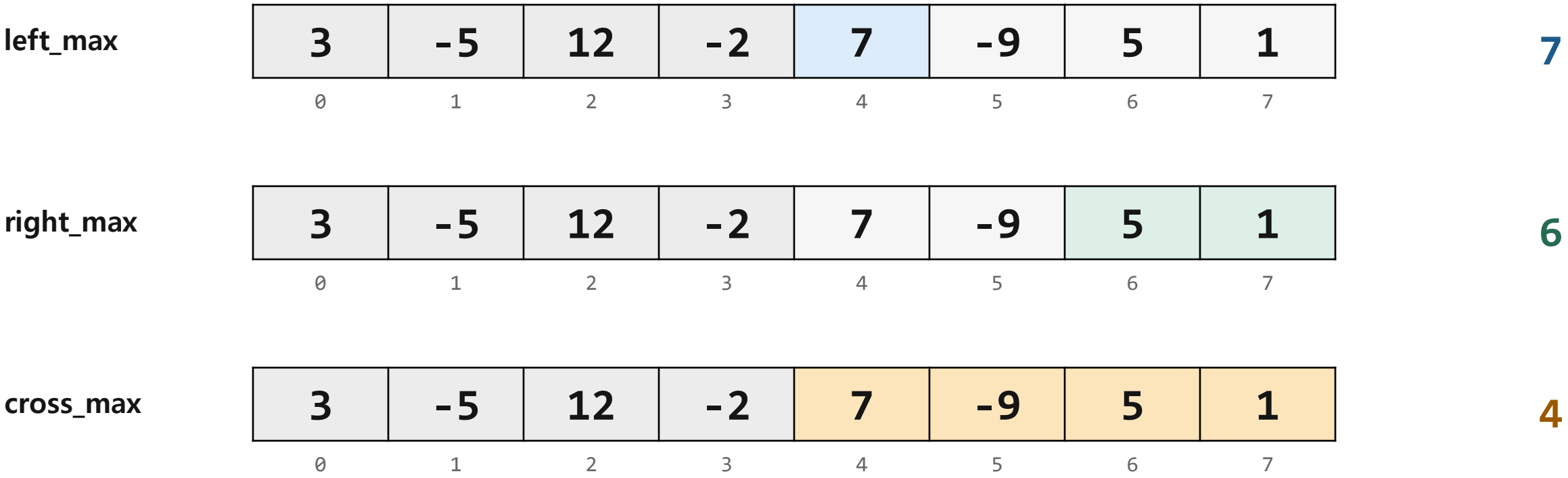
left_sum = −2, right_sum = 6이므로 cross_max = 4입니다.
두 구간은 인덱스 5와 6에서 맞닿아 빈틈 없이 이어집니다.



−2 + (6) = 4. 마지막에 읽은 합이 아니라 최선의 기록을 더합니다.

[4~7] 세 후보 비교 → 7 반환

왼쪽 7, 오른쪽 6, 경계를 걸친 합 4을 비교합니다.
max(7, 6, 4) = 7입니다.

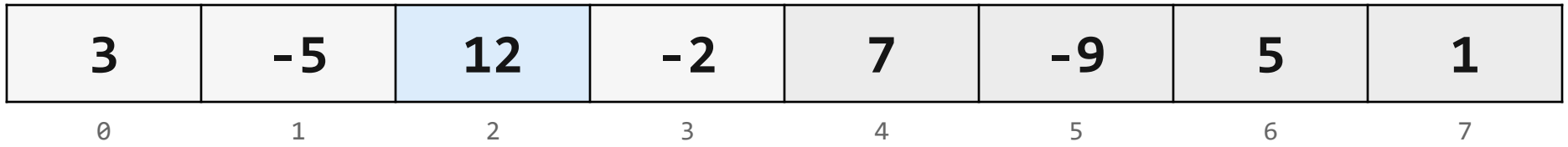


[0~7] 부모의 right_max에 7을 반환합니다.

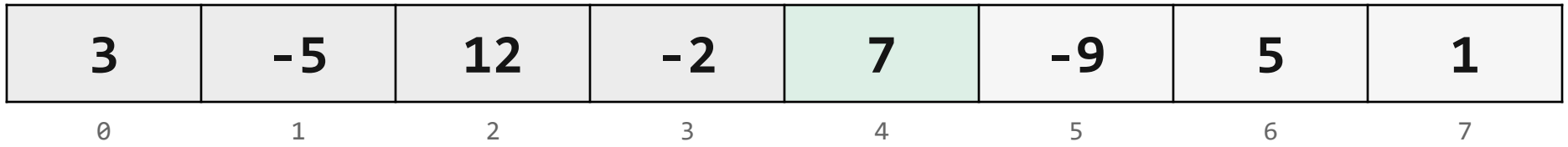
[0~7] 오른쪽 답 7도 받음

left_max = 12, right_max = 7입니다. 두 재귀 호출을 마쳤습니다.
아직 경계를 걸치는 cross_max를 계산하고 비교해야 합니다.

left_max = 12



right_max = 7



다음은 경계를 걸치는 후보입니다. left_max + right_max를 계산하지 않습니다.

중앙을 걸친 구간: 10 + 7 = 17

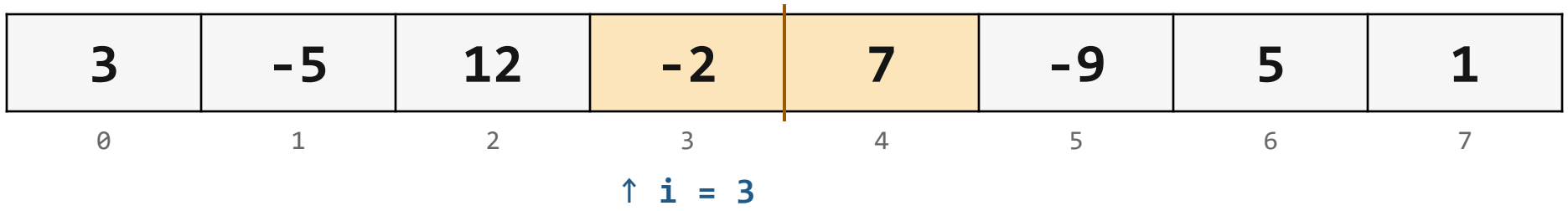
mid = 3의 −2에서 왼쪽으로, mid+1 = 4의 7에서 오른쪽으로 훑습니다.



그림의 최대 합은 중앙에서 훑으며 저장한 합입니다. 절반 전체의 left_max·right_max와 구분하세요.

[0~7] 이제 경계를 걸치는 합 계산

mid = 3입니다. 왼쪽 스캔은 a[3] = −2부터 시작합니다.
current_sum = 0, left_sum = −∞로 두고 i를 왼쪽으로 움직입니다.



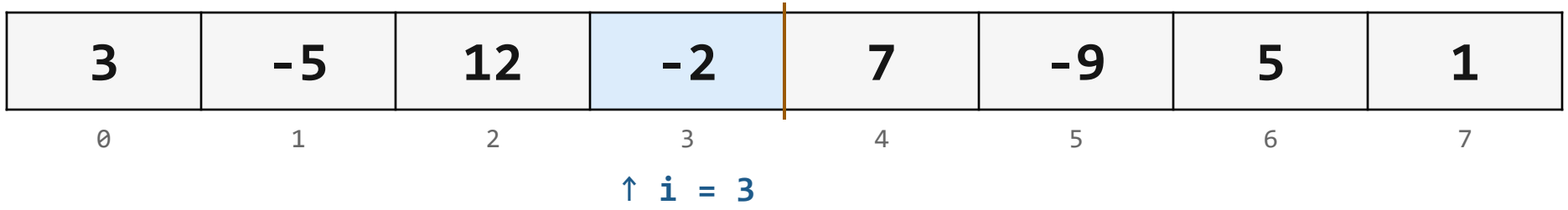

```
current_sum = 0        left_sum = -∞
```

재귀 답은 유지: left_max = 12, right_max = 7.

[0~7] 왼쪽 스캔 1/4

i = 3, a[3] = −2을 더합니다. 현재 구간은 [3~3]입니다.
현재 합 −2 > 이전 기록 −∞ → left_sum 갱신.

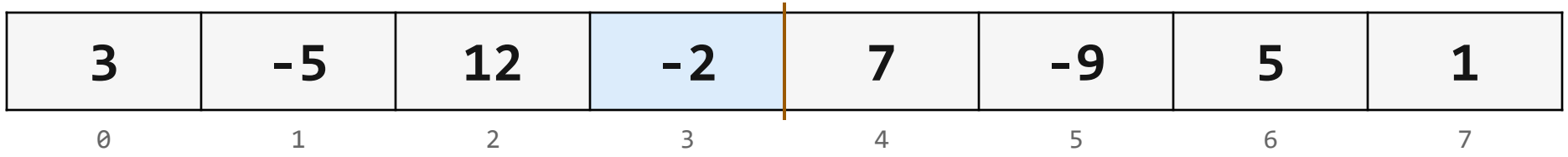
현재 구간




```
current_sum = 0 + (-2) = -2
```

```
left_sum: -∞ → -2
```

저장한 최선

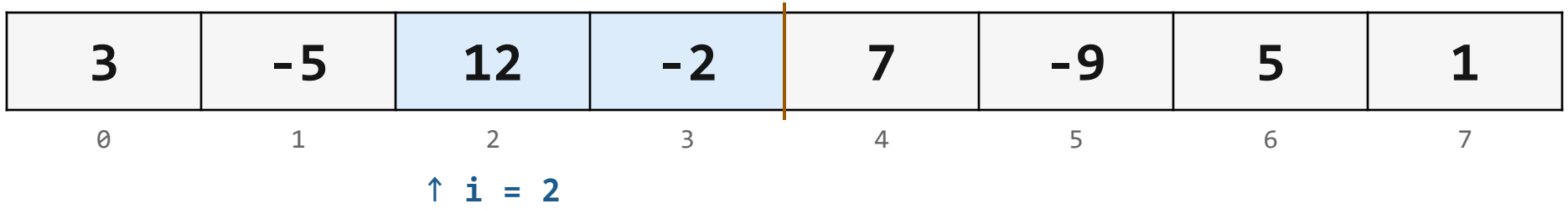


다음 i = 2: a[2] = 12을 현재 합 −2에 더합니다.

[0~7] 왼쪽 스캔 2/4

i = 2, a[2] = 12을 더합니다. 현재 구간은 [2~3]입니다.
현재 합 10 > 이전 기록 −2 → left_sum 갱신.

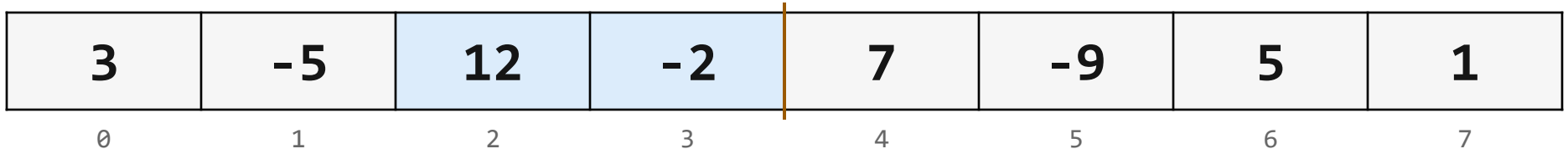
현재 구간



current_sum = -2 + (12) = 10

left_sum: -2 → 10

저장한 최선

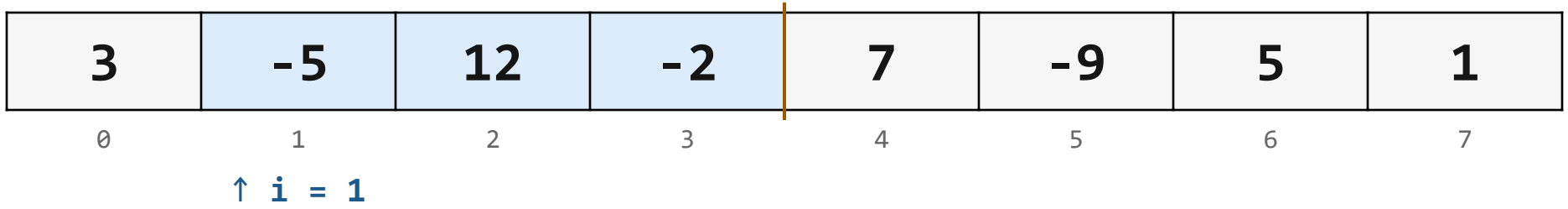


다음 i = 1: a[1] = −5을 현재 합 10에 더합니다.

[0~7] 왼쪽 스캔 3/4

i = 1, a[1] = −5을 더합니다. 현재 구간은 [1~3]입니다.
현재 합 5 ≤ 이전 기록 10 → left_sum 유지.

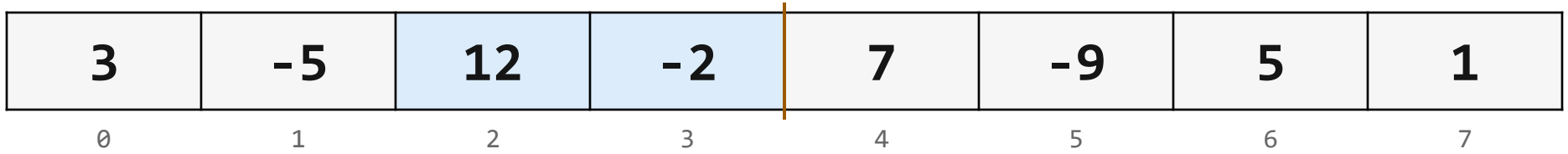
현재 구간



current_sum = 10 + (-5) = 5

left_sum = 10 유지

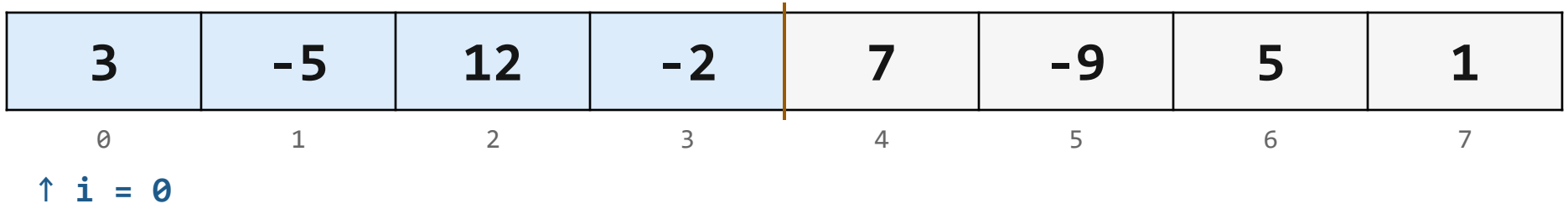
저장한 최선



다음 i = 0: a[0] = 3을 현재 합 5에 더합니다.

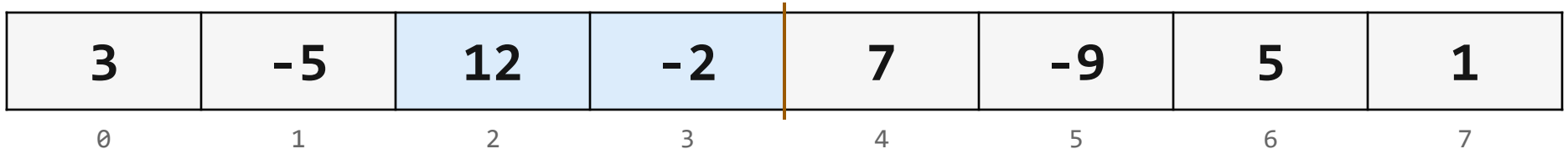
[0~7] 왼쪽 스캔 4/4

i = 0, a[0] = 3을 더합니다. 현재 구간은 [0~3]입니다.
현재 합 8 ≤ 이전 기록 10 → left_sum 유지.



`current_sum = 5 + (3) = 8`

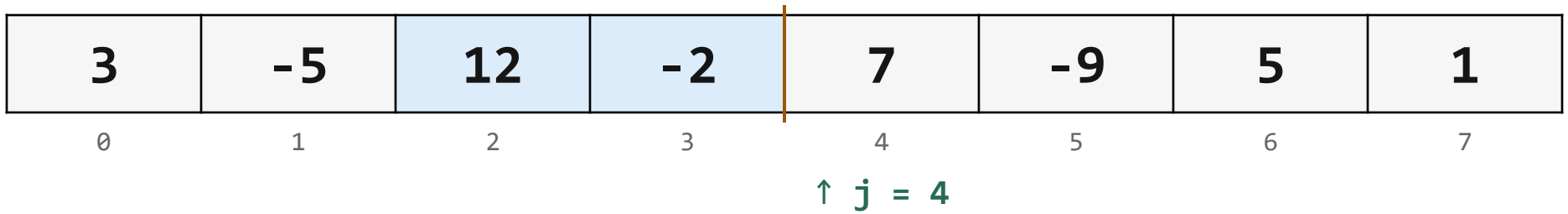
`left_sum = 10 유지`



왼쪽 스캔 끝. 이 기록을 보관하고 오른쪽 현재 합을 0으로 다시 시작합니다.

[0~7] 오른쪽은 합을 0으로 다시 시작

왼쪽 기록 left_sum = 10은 보관합니다. current_sum만 0으로 바꿉니다.
right_sum = −∞로 두고 j = mid + 1 = 4부터 오른쪽을 훑습니다.



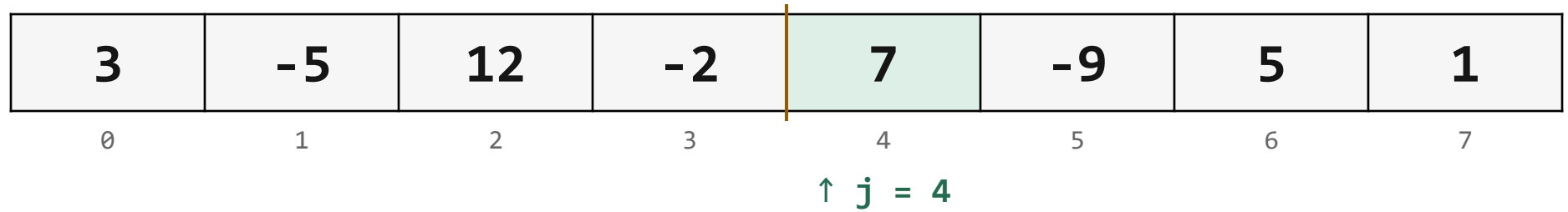

```
current_sum = 0

left_sum = 10 (보관)                    right_sum = −∞
```

오른쪽 첫 칸은 a[4] = 7입니다. mid를 다시 읽지 않습니다.

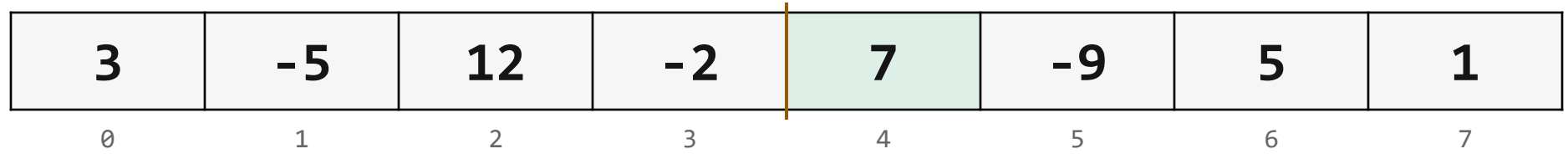
[0~7] 오른쪽 스캔 1/4

j = 4, a[4] = 7을 더합니다. 현재 구간은 [4~4]입니다.
현재 합 7 > 이전 기록 −∞ → right_sum 갱신.




```
current_sum = 0 + (7) = 7
```

```
right_sum: -∞ → 7
```

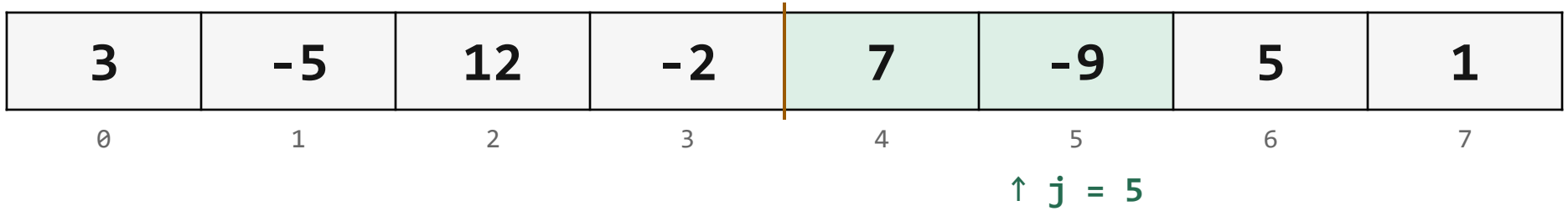


다음 j = 5: a[5] = −9을 현재 합 7에 더합니다.

[0~7] 오른쪽 스캔 2/4

j = 5, a[5] = −9을 더합니다. 현재 구간은 [4~5]입니다.
현재 합 −2 ≤ 이전 기록 7 → right_sum 유지.

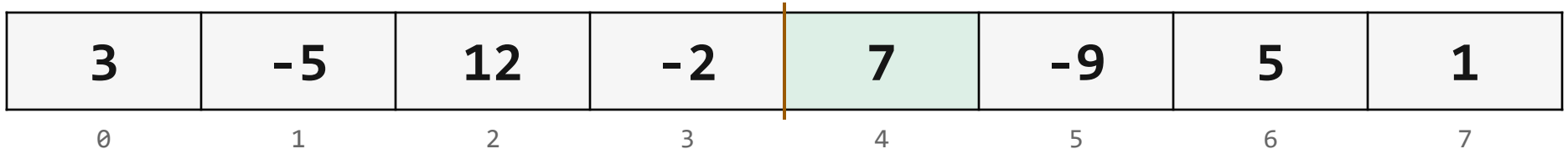
현재 구간



current_sum = 7 + (-9) = -2

right_sum = 7 유지

저장한 최선

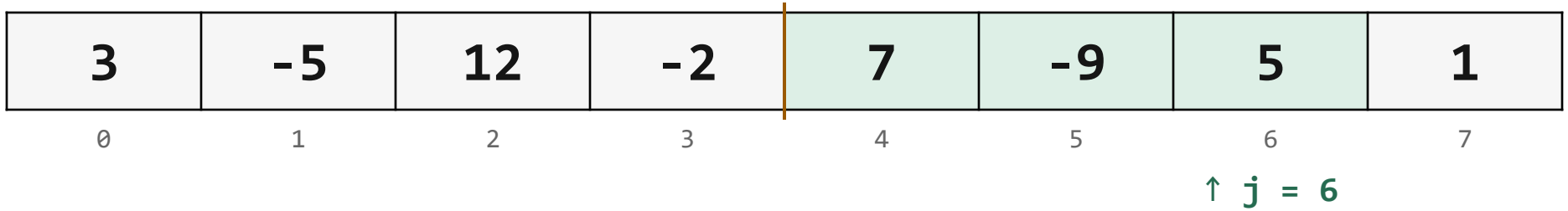


다음 j = 6: a[6] = 5을 현재 합 −2에 더합니다.

[0~7] 오른쪽 스캔 3/4

j = 6, a[6] = 5을 더합니다. 현재 구간은 [4~6]입니다.
현재 합 3 ≤ 이전 기록 7 → right_sum 유지.

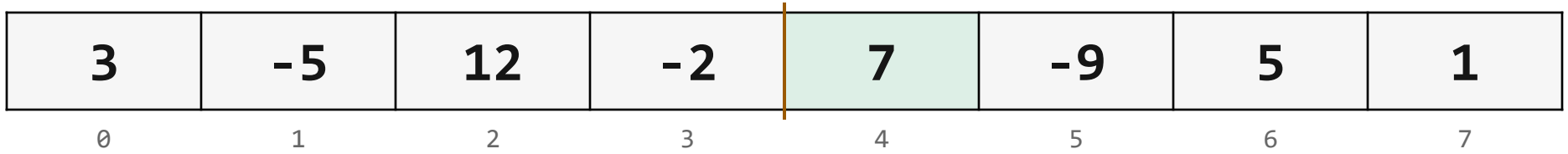
현재 구간



current_sum = -2 + (5) = 3

right_sum = 7 유지

저장한 최선

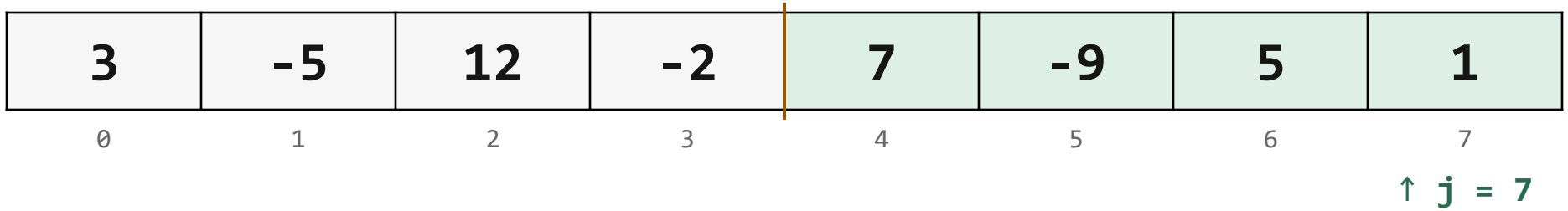


다음 j = 7: a[7] = 1을 현재 합 3에 더합니다.

[0~7] 오른쪽 스캔 4/4

j = 7, a[7] = 1을 더합니다. 현재 구간은 [4~7]입니다.
현재 합 4 ≤ 이전 기록 7 → right_sum 유지.

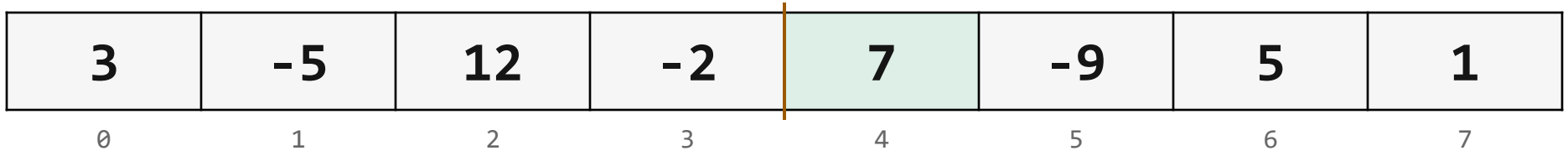
현재 구간



current_sum = 3 + (1) = 4

right_sum = 7 유지

저장한 최선

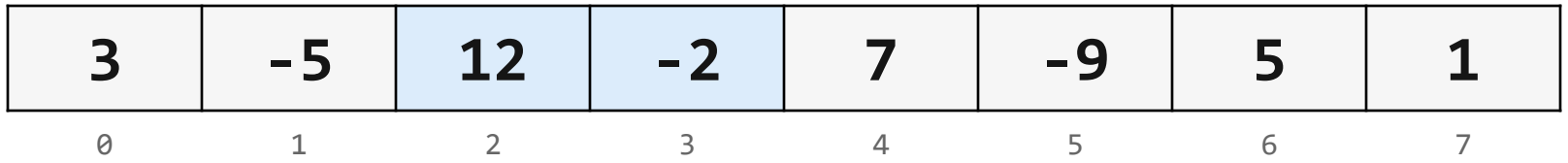


오른쪽 스캔 끝. 왼쪽 기록과 오른쪽 기록을 더해 cross_max를 구합니다.

[0~7] 경계에서 두 기록을 붙입니다

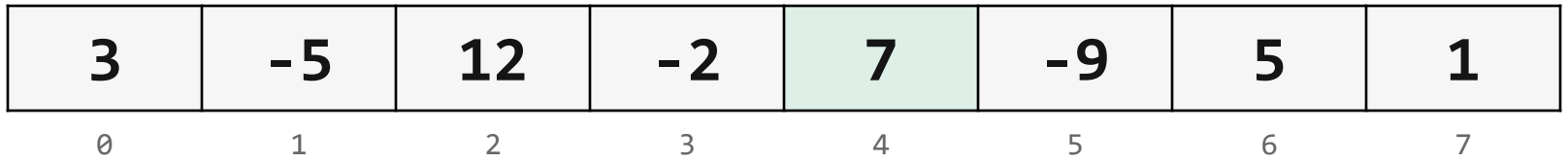
left_sum = 10, right_sum = 7이므로 cross_max = 17입니다.
두 구간은 인덱스 3와 4에서 맞닿아 빈틈 없이 이어집니다.

left_sum



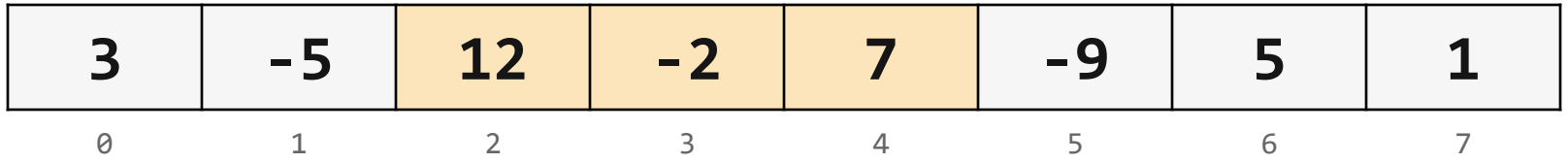
10

right_sum



7

붙인 구간
cross_max

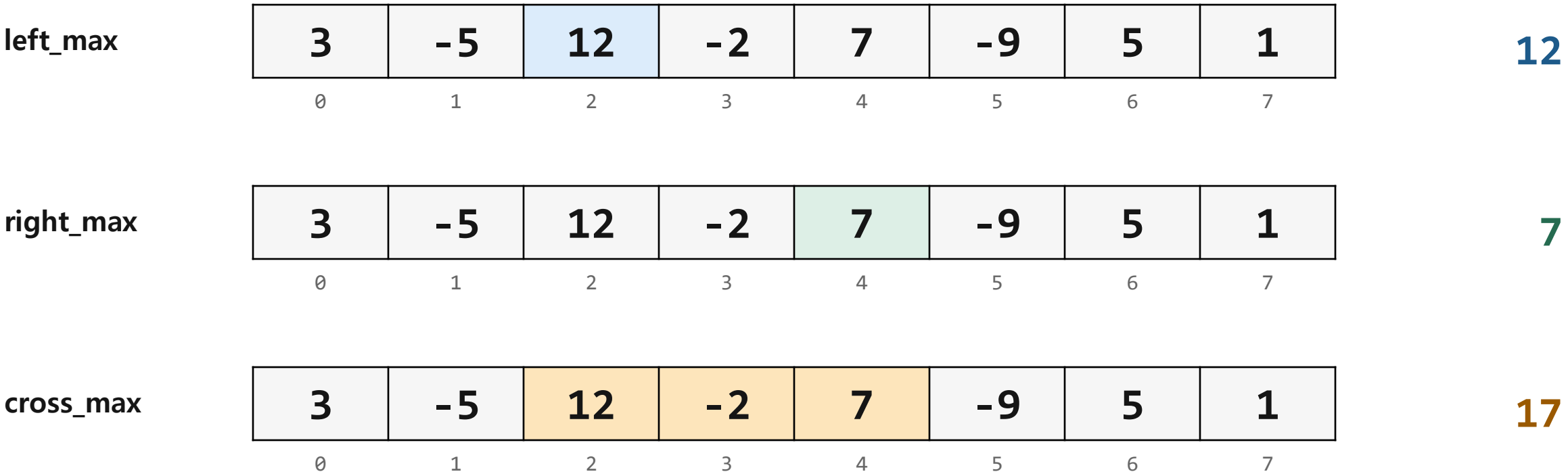


17

10 + (7) = 17. 마지막에 읽은 합이 아니라 최선의 기록을 더합니다.

[0~7] 세 후보 비교 → 17 반환

왼쪽 12, 오른쪽 7, 경계를 걸친 합 17을 비교합니다.
max(12, 7, 17) = 17입니다.



전체 호출의 최종 답 17을 반환합니다.

전체 호출의 코드와 방금 본 값

각 호출은 왼쪽 답, 오른쪽 답, 걸친 답을 차례로 구하고 비교합니다.
원소 하나일 때만 재귀 호출 없이 그 값을 반환합니다.

```
def max_subarray_sum(a, low, high):
    if low == high:
        return a[low]
    mid = (low + high) // 2
    left_max = max_subarray_sum(a, low, mid)
    right_max = max_subarray_sum(a, mid + 1, high)
    cross_max = max_crossing_sum(a, low, mid, high)
    return max(left_max, right_max, cross_max)
```

전체 호출: max_subarray_sum(a, 0, 7) → max(12, 7, 17) → 17

경계 계산 코드 ① 왼쪽 스캔

left_sum은 mid에서 끝나는 구간 중 최댓값입니다.
range(mid, low − 1, −1)은 mid부터 low까지 한 칸씩 왼쪽으로 이동합니다.

```
def max_crossing_sum(a, low, mid, high):
    left_sum = float('-inf')
    current_sum = 0
    for i in range(mid, low - 1, -1):
        current_sum += a[i]
        if current_sum > left_sum:
            left_sum = current_sum
```

i: 3 → 2 → 1 → 0　　기록: -2 → 10 → 10 → 10

새 원소는 current_sum에 더합니다. left_sum은 최댓값을 보관하는 기록입니다.

경계 계산 코드 ② 오른쪽과 반환

current_sum만 0으로 다시 설정합니다. left_sum은 그대로 보관합니다.
오른쪽은 mid + 1부터 high까지 읽고, 두 최댓값 기록을 더해 반환합니다.

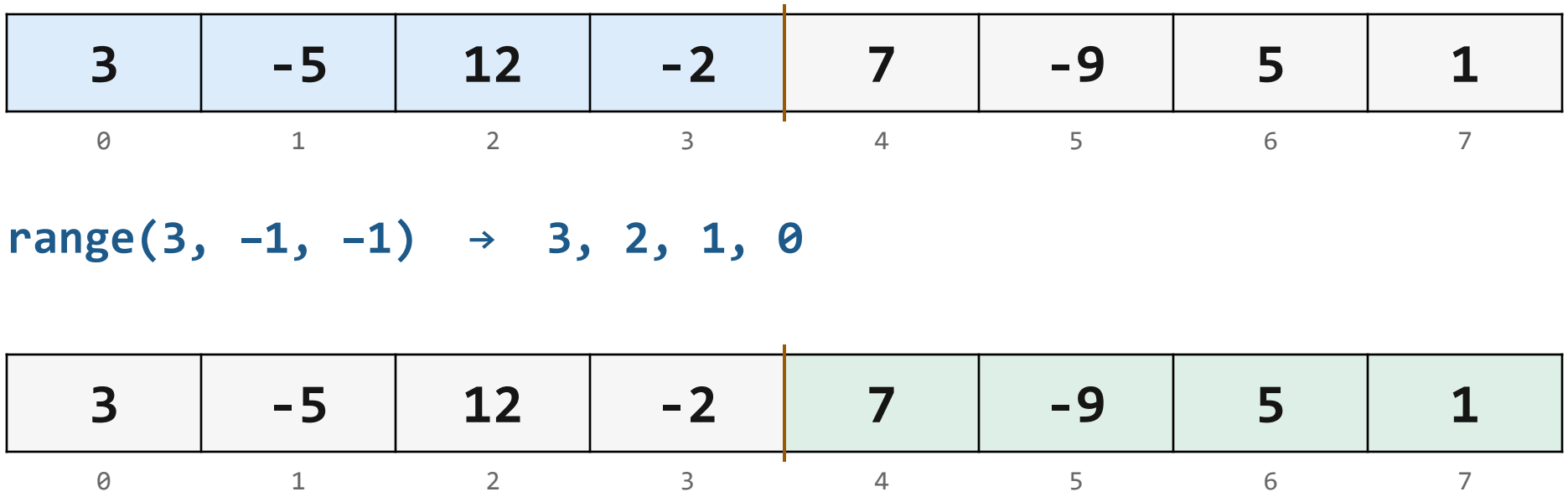
```
right_sum = float('-inf')
current_sum = 0
for j in range(mid + 1, high + 1):
    current_sum += a[j]
    if current_sum > right_sum:
        right_sum = current_sum
return left_sum + right_sum
```

j: 4 → 5 → 6 → 7 **반환: 10 + 7 = 17**

두 반복문은 다른 칸을 읽습니다. 경계 원소를 중복해서 더하지 않습니다.

range의 끝값은 읽지 않습니다

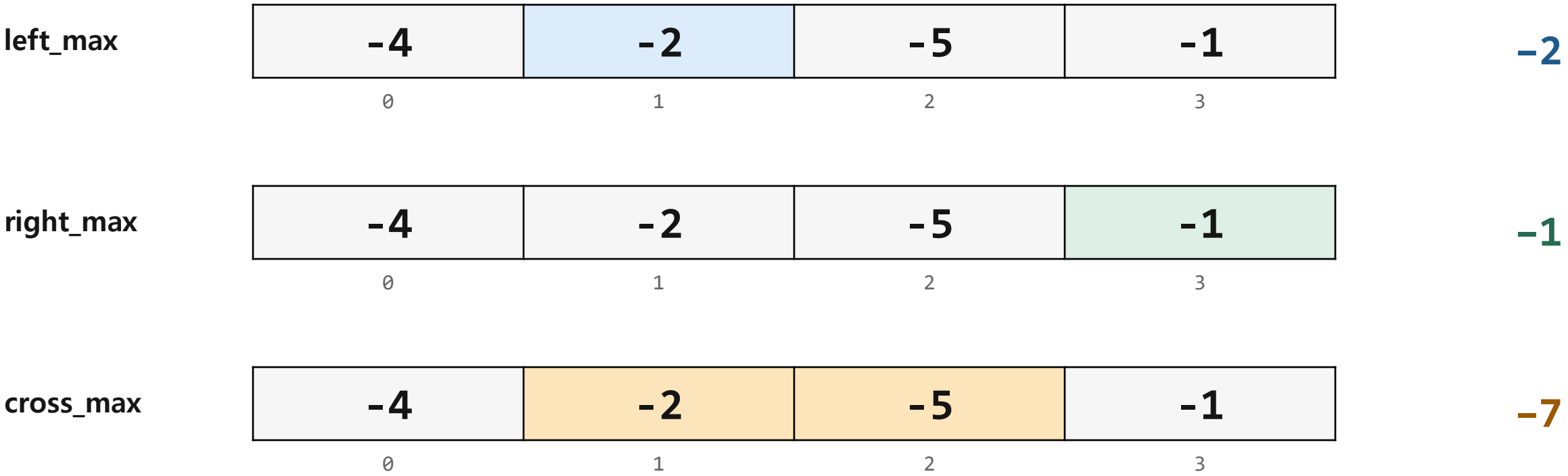
왼쪽은 3, 2, 1, 0을 읽습니다. 오른쪽은 4, 5, 6, 7을 읽습니다.
−1과 8은 range를 멈추기 위한 값이며, 배열에서 접근할 인덱스가 아닙니다.



range(4, 8) → 4, 5, 6, 7. 현재 호출의 low와 high까지만 읽습니다.

모두 음수면 0이 답이 아닙니다

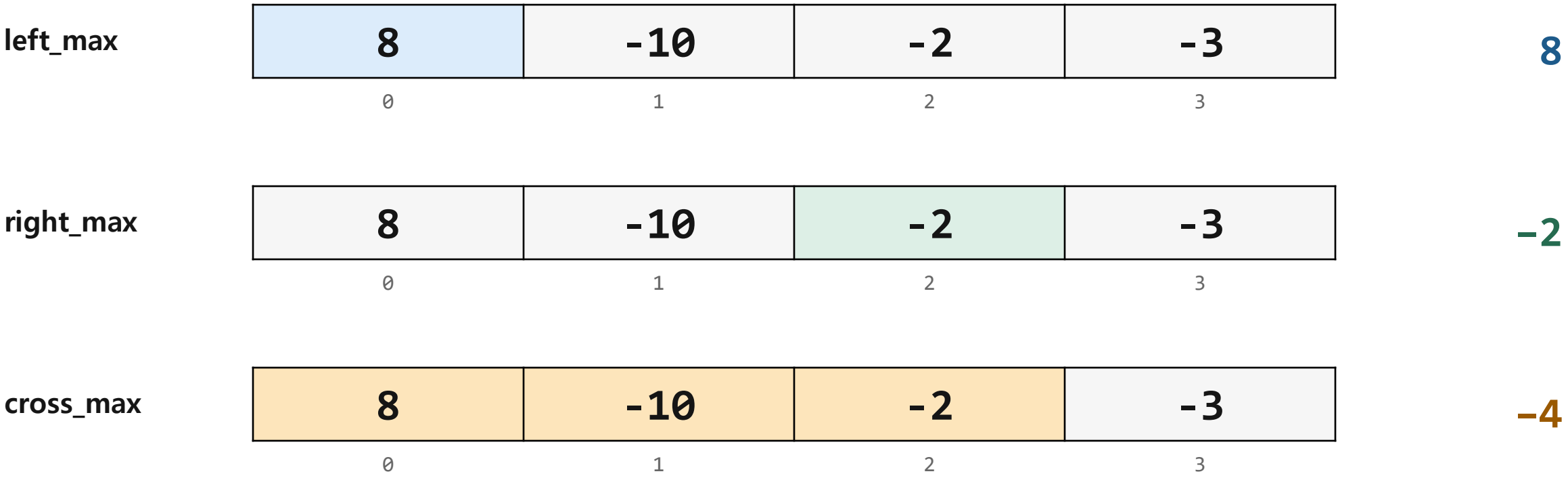
예: a = [−4, −2, −5, −1]. 비어 있지 않은 구간만 고릅니다.
왼쪽 −2, 오른쪽 −1, 걸친 합 −7 중 가장 큰 값은 −1입니다.



왼쪽 스캔: −2 → −6 (기록 −2) / 오른쪽: −5 → −6 (기록 −5)

cross_max가 항상 이기지는 않습니다

예: a = [8, −10, −2, −3]. 경계를 걸치면 −10을 빼놓을 수 없습니다.
왼쪽 8, 오른쪽 −2, 걸친 합 −4 중 최댓값 8을 반환합니다.



경계 합: (8 − 10) + (−2) = −4. 전체 답은 왼쪽 한 칸의 8입니다.

cross_max 한 번: 왼쪽 4칸과 오른쪽 4칸

왼쪽 스캔: 가운데에서 왼쪽으로

3	-5	12	-2

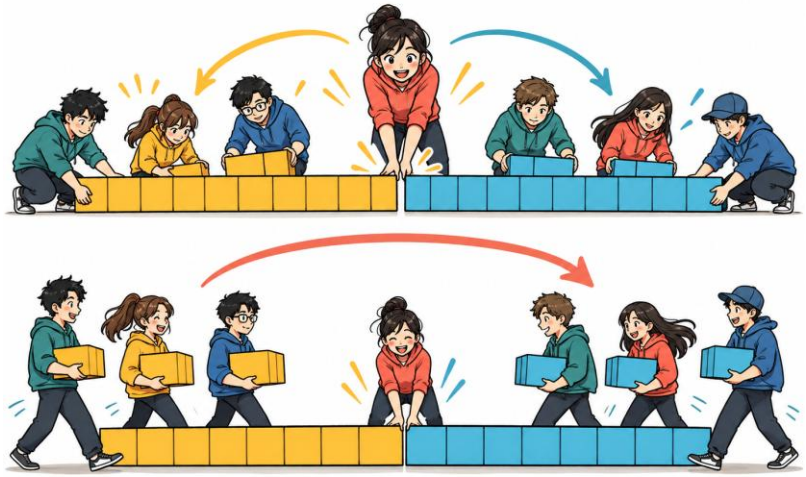
4칸

오른쪽 스캔: 가운데 다음부터 오른쪽으로

7	-9	5	1

4칸

4 + 4 = 8칸



두 반복문은 차례대로 실행합니다. 현재 구간이 n칸이면 cross_max 한 번은 O(n)입니다.

작은 호출을 모아도, 한 층에서 8칸을 읽습니다

2칸짜리 호출



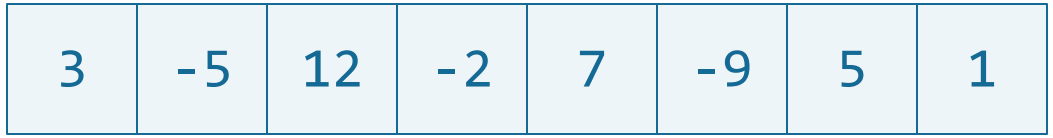
2 × 4 = 8

4칸짜리 호출



4 × 2 = 8

8칸짜리 호출



8 × 1 = 8

스캔 합계: 8 + 8 + 8 = 24칸

한 칸짜리에서는 스캔하지 않고 반환합니다. 호출·반환에도 비용이 있지만 전체 시간 차수는 같습니다.

8칸을 3층, n칸을 약 $\log_2 n$층

입력 길이	구간 크기가 줄어드는 과정	스캔하는 층
8	8, 4, 2, 1	3층
16	16, 8, 4, 2, 1	4층
32	32, 16, 8, 4, 2, 1	5층

한 층의 읽기 n칸 × 약 $\log_2 n$층

최대 구간 합의 전체 시간은 O(n log n). 마지막 한 칸에서는 cross_max를 계산하지 않습니다.

점화식은 코드의 세 부분을 적은 것입니다

$$T(n) = T(n/2) + T(n/2) + O(n)$$

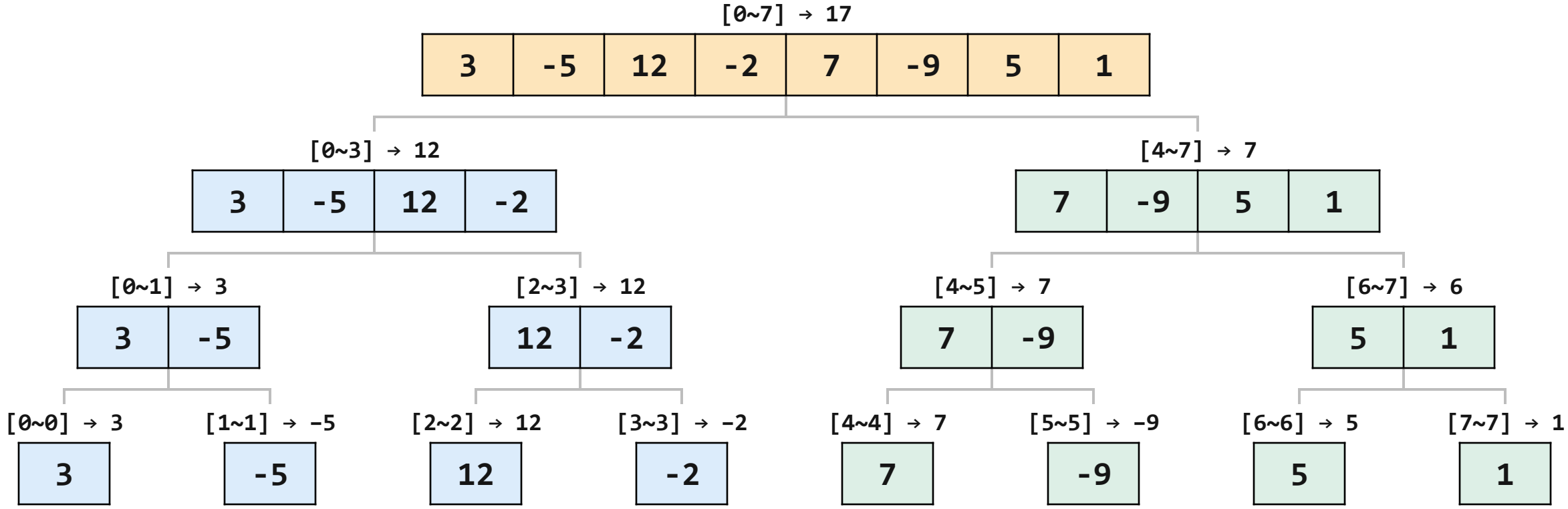
$T(n)$	$T(n/2)$	$T(n/2)$	$O(n)$
전체 시간	왼쪽 절반의 시간	오른쪽 절반의 시간	현재 구간의 작업
	왼쪽 답을 구함	**오른쪽 답을 구함**	**중앙에서 스캔**

$$T(n) = 2T(n/2) + O(n)$$

전체 시간은 앞의 층 그림으로 계산했습니다. n칸씩 약 $\log_2 n$층을 처리하므로 $O(n \log n)$입니다.

작은 반환값이 모여 전체 답 17

원소 하나 → 두 칸 → 네 칸 → 여덟 칸 순서로 답이 돌아옵니다.
전체 호출은 left_max = 12, right_max = 7, cross_max = 17을 비교합니다.



최종 구간: a[2~4] = [12, −2, 7], 합 = 17. 중간 칸을 빠뜨리지 않았습니다.

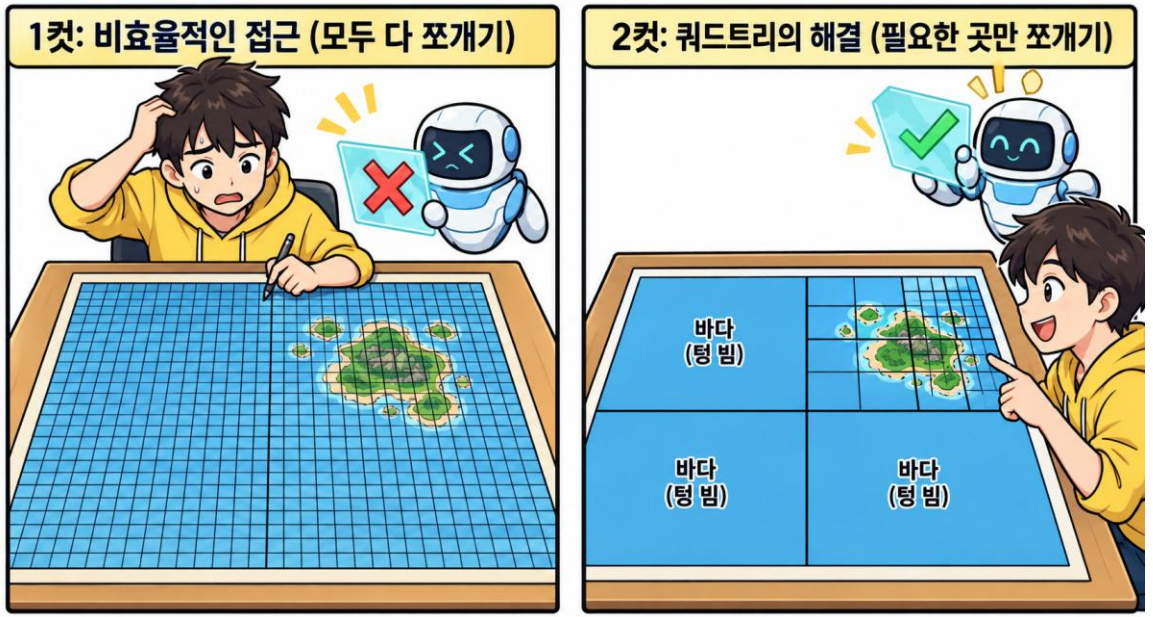
4.5

쿼드 트리

Quadtree
2차원 공간을 재귀적으로 4등분하는 자료 구조
이미지 압축부터 게임 충돌 감지까지

쿼드 트리의 목적: 그림을 영역으로 저장

그림의 같은 색 영역을
한 덩어리로 기록합니다.



전부 같은 색: 색 하나를 기록
색이 섞인 곳: 네 부분으로 나눔

저장한 기록으로
원래 그림을 다시 그릴 수 있습니다.

여기서는 흑백 이미지로 설명합니다. 0은 흰색, 1은 검은색입니다.

4×4 그림: 색 16개를 어떻게 기록할까요?

입력 그림

0	0	1	1
0	0	1	1
1	1	0	0
1	1	0	0

픽셀마다 적으면

```
0 0 1 1
0 0 1 1
1 1 0 0
1 1 0 0
```

전체 16칸의 색을 기록

전체 크기도 알아야 합니다. 한 변은 1, 2, 4, 8, …인 정사각형을 다룹니다.

같은 색 영역: 네 칸을 0 하나로 기록

현재 영역은 2×2

0	0
0	0

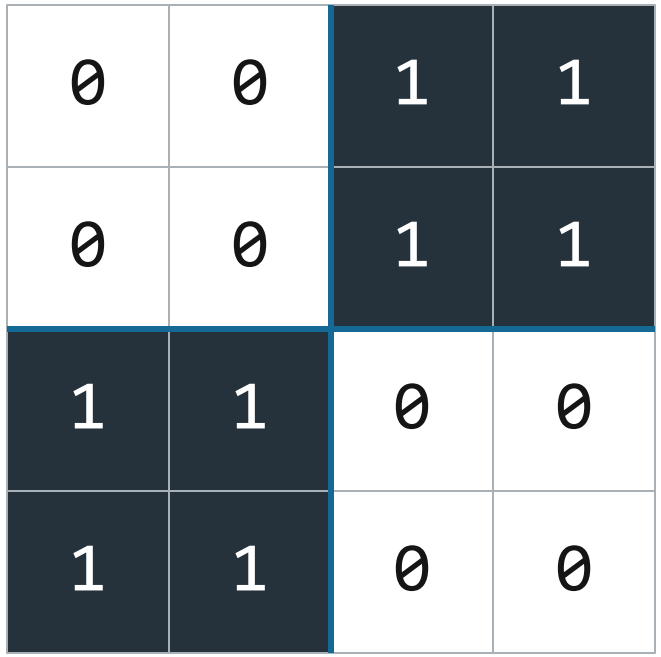
저장한 기록

0

뜻: 이 영역의 네 칸이 모두 흰색

0 또는 1은 현재 영역 전체의 색입니다. 영역의 위치와 크기는 부모가 알려 줍니다.

색이 섞인 영역: 같은 크기의 네 조각



① 왼쪽 위 0

② 오른쪽 위 1

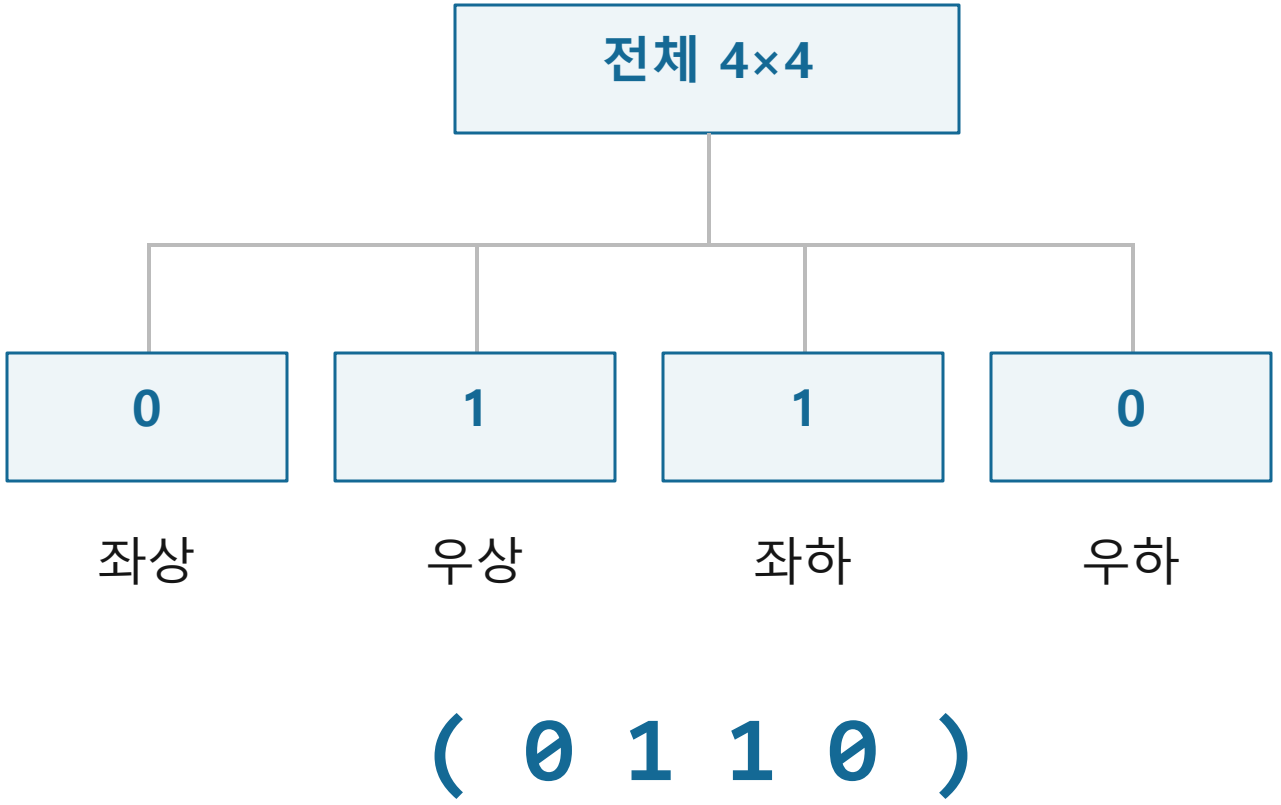
③ 왼쪽 아래 1

④ 오른쪽 아래 0

가로와 세로를 각각 절반으로 나눕니다. 4×4 전체에서 2×2 자식 네 개가 생깁니다.

네 영역의 결과를 (0110)으로 기록

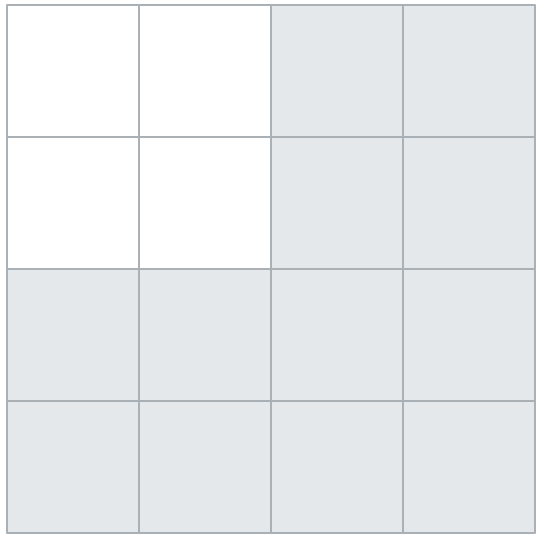
0	0	1	1
0	0	1	1
1	1	0	0
1	1	0	0



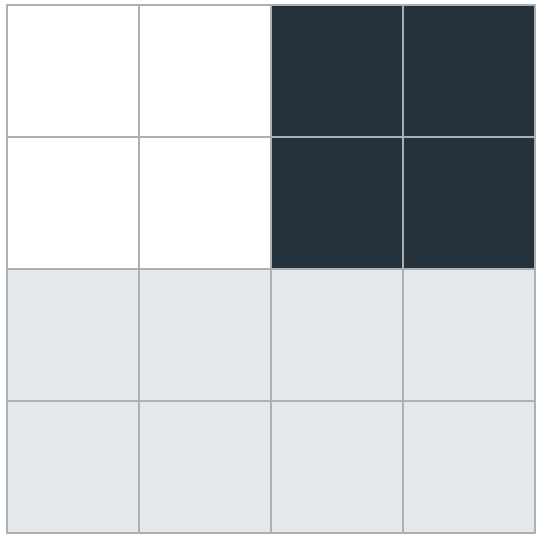
(는 네 부분으로 나눔,) 는 그 부모의 네 결과가 끝남을 뜻합니다.

복원: (0110)을 읽어 원래 그림 그리기

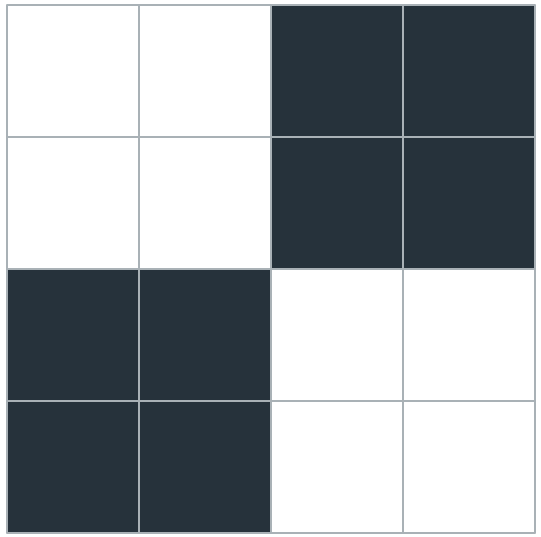
저장한 표현: (0 1 1 0)



① 첫 0: 좌상 채움



② 다음 1: 우상 채움



③ 나머지 1, 0도 채움

회색은 아직 채우지 않은 영역입니다. 마지막 그림은 입력의 16칸과 모두 같습니다.

섞인 자식도 같은 규칙으로 다시 나눕니다

0	0	1	1
0	1	1	1
0	0	0	0
0	0	0	0

왼쪽 위 2×2만 다시 분할

0	0
0	1

첫 자식의 기록: (0001)

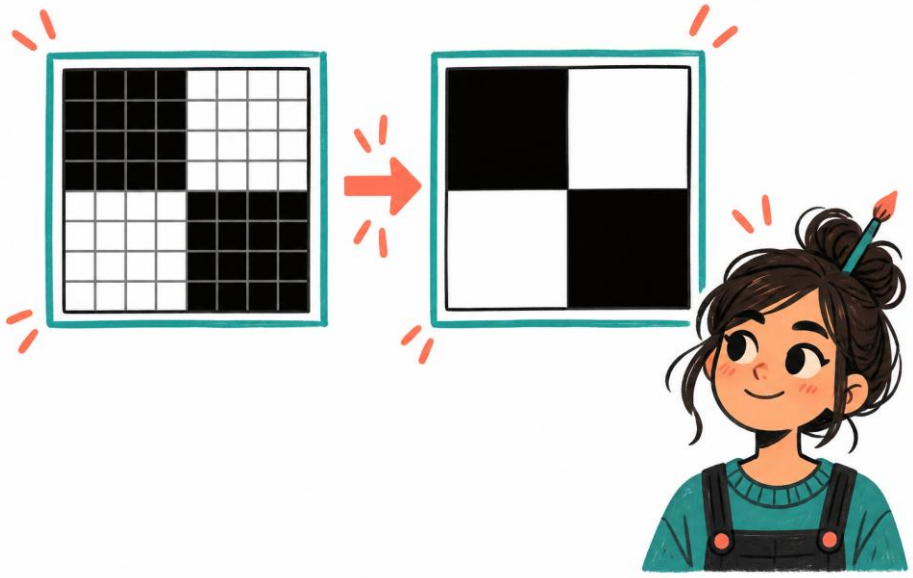
전체: ((0001)100)

안쪽 괄호는 그 자식 영역의 분할입니다. 부모는 네 자식의 결과를 순서대로 묶습니다.

쿼드 트리로 남기는 정보

색 한 글자: 현재 영역 전체의 색
괄호: 네 자식으로 나눈 구조

전체 크기와 자식 순서를 알면
원래 그림의 색과 위치를 복원합니다.



다음 8×8 예제도 같은 규칙입니다. 단색 영역은 끝내고, 섞인 영역만 다시 나눕니다.

같은 색이면 하나로, 섞이면 네 조각

균일한 영역은 0 또는 1. 혼합 영역만 네 부분으로 나눕니다.



좌상 → 우상 → 좌하 → 우하 순서를 저장과 복원에서 똑같이 사용합니다.

입력 영상: 검은 칸의 위치부터 확인합니다

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

행 / 열은 0부터 셉니다.

첫 호출: (행 0, 열 0, 크기 8)

오른쪽 위는 전부 1입니다.

추가 검은 칸: 행 5, 열 1

전체는 0과 1이 섞였습니다.

전체는 혼합 → 네 부분으로 분할

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

첫 분할: 네 호출의 좌표를 정합니다

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

행 / 열은 0부터 셉니다.

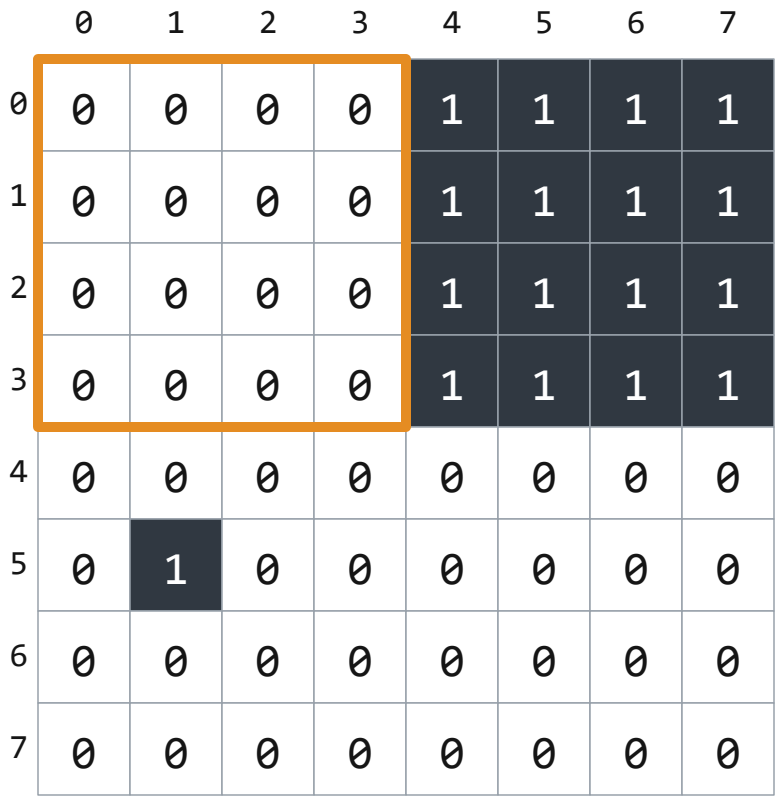
half = 8 // 2 = 4

좌상: (0, 0, 4)
우상: (0, 4, 4)
좌하: (4, 0, 4)
우하: (4, 4, 4)

부모 결과의 틀: (? ? ? ?)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

① 좌상: 16칸을 0 하나로 반환



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

행 / 열은 0부터 셉니다.

현재 호출: (0, 0, 4)

16칸이 모두 0입니다.
더 나누지 않습니다.

반환: "0"

부모가 모은 결과: (0 ? ? ?)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

② 우상: 16칸을 1 하나로 반환

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

행 / 열은 0부터 셉니다.

현재 호출: (0, 4, 4)

16칸이 모두 1입니다.
더 나누지 않습니다.

반환: "1"

부모가 모은 결과: (0 1 ? ?)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

③ 좌하: 검은 칸이 있어 다시 나눕니다

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

행 / 열은 0부터 셉니다.

현재 호출: (4, 0, 4)

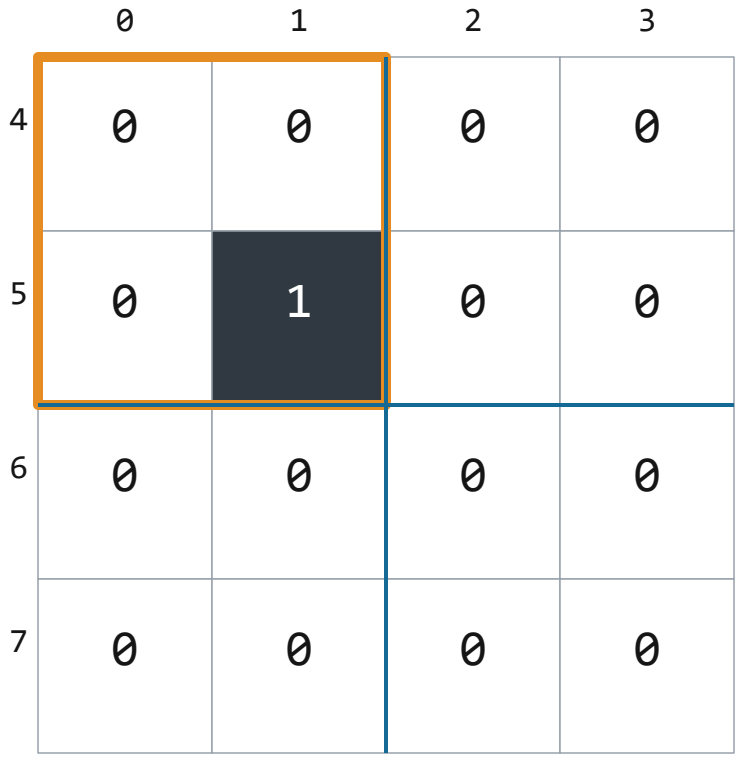
행 5, 열 1의 값이 1입니다.
다른 칸은 0 → 혼합입니다.

다시 크기 2의 네 영역으로 분할

루트는 좌하의 결과를 기다립니다.

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

좌하의 네 부분: 첫 2×2만 혼합입니다



행 / 열은 0부터 셉니다.

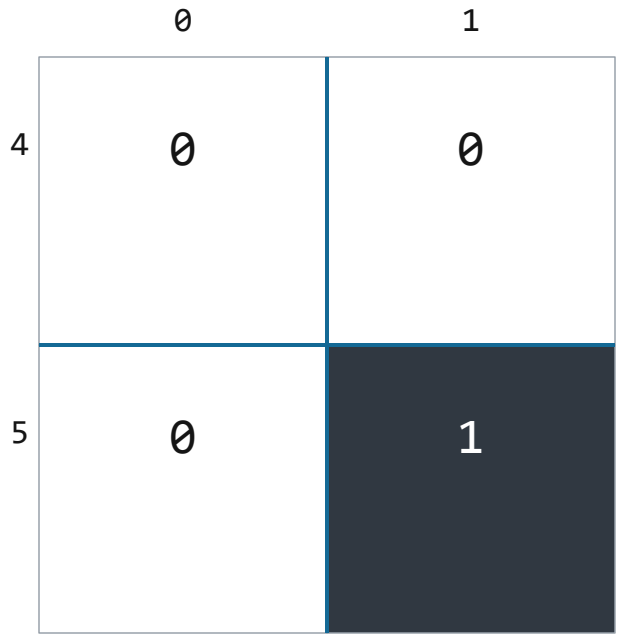
크기 2의 자식들

좌상 (4, 0, 2): 혼합
우상 (4, 2, 2): 전부 0
좌하 (6, 0, 2): 전부 0
우하 (6, 2, 2): 전부 0

먼저 (4, 0, 2)를 처리합니다.

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

혼합된 2×2: 네 픽셀을 순서대로 읽습니다



행 / 열은 0부터 셉니다.

현재 호출: (4, 0, 2)

좌상 (4, 0): 0
우상 (4, 1): 0
좌하 (5, 0): 0
우하 (5, 1): 1

네 자식의 색 순서: 0 → 0 → 0 → 1

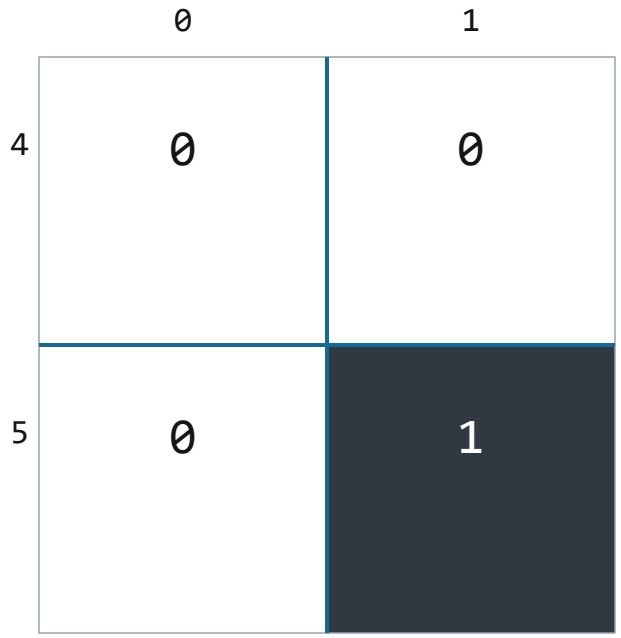
0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

크기 1의 네 호출: 각각 즉시 끝납니다



한 칸은 반드시 균일합니다. 크기 1 아래로는 더 나누지 않습니다.

2×2의 답: (0001)을 반환합니다



행 / 열은 0부터 셉니다.

두 번째 층의 혼합 영역

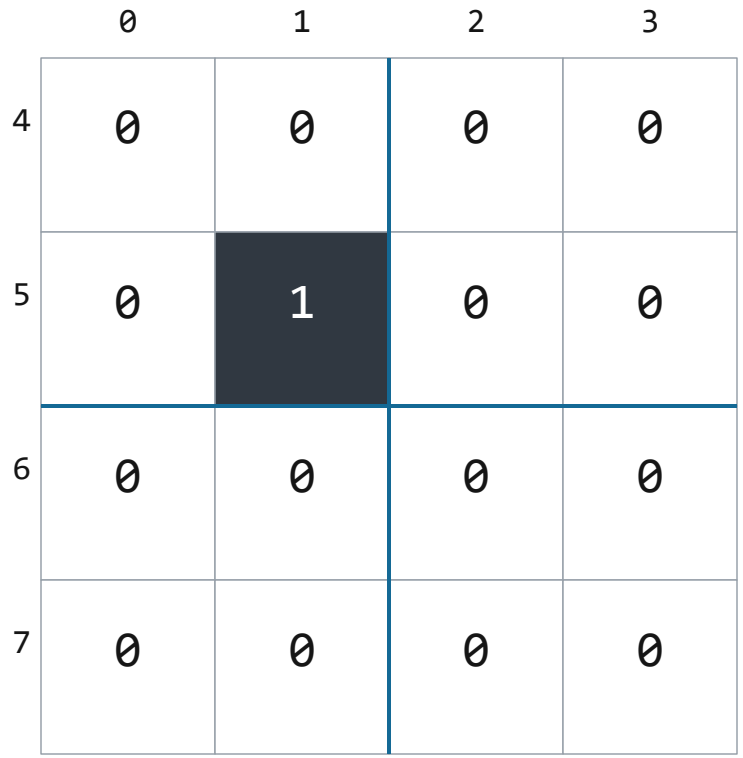
여는 괄호: (
자식 결과: 0 0 0 1
닫는 괄호:)

합치면 "(0001)"

좌하 4×4의 첫 자식 답: (0001)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

좌하의 나머지 세 영역은 모두 0입니다



행 / 열은 0부터 셉니다.

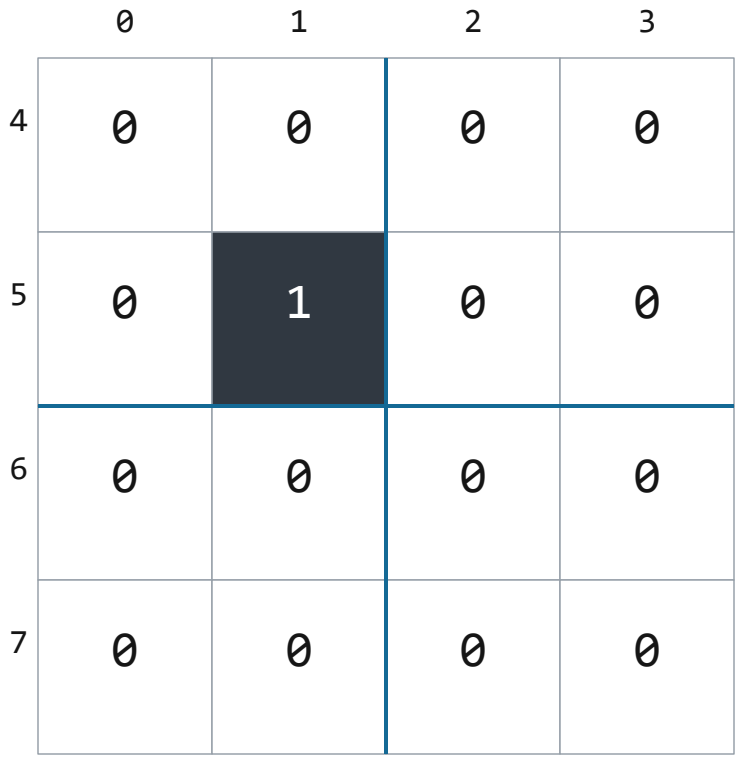
첫 자식: (0001)

우상 (4, 2, 2) → "0"
좌하 (6, 0, 2) → "0"
우하 (6, 2, 2) → "0"

네 답을 모두 받았습니다.

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

좌하 4×4의 답: ((0001)000)



행 / 열은 0부터 셉니다.

좌하 4×4가 받은 네 결과

(0001) / 0 / 0 / 0

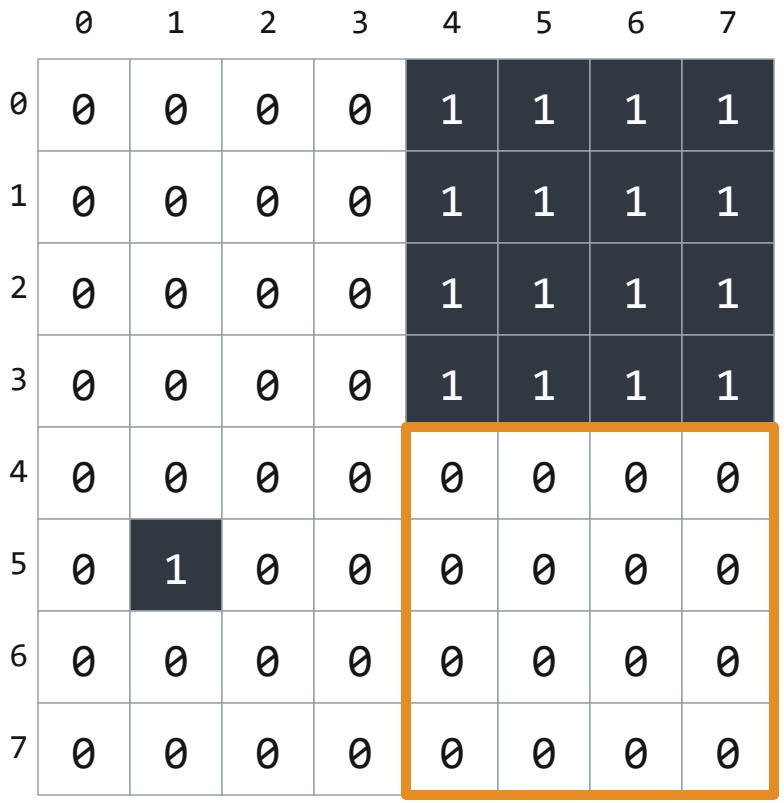
자기 괄호로 한 번 더 묶습니다.

반환: "((0001)000)"

루트: (0 1 ((0001)000) ?)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

④ 우하: 마지막 16칸도 0 하나



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

행 / 열은 0부터 셉니다.

현재 호출: (4, 4, 4)

16칸이 모두 0입니다.
더 나누지 않습니다.

반환: "0"

루트의 네 자식 결과가 완성되었습니다.

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

전체 결과: (01((0001)000)0)

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

행 / 열은 0부터 셉니다.

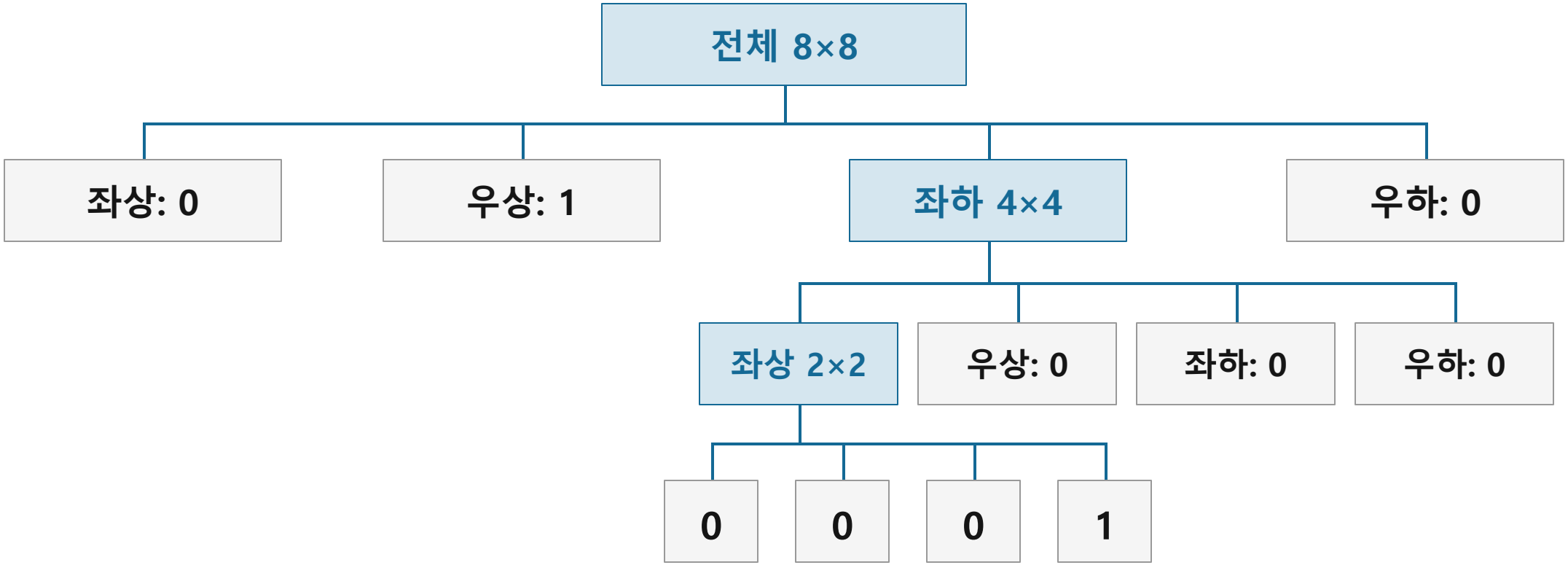
좌상: 0
우상: 1
좌하: ((0001)000)
우하: 0

전체를 괄호로 묶습니다.

완성: (01((0001)000)0)

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

재귀 트리: 나눈 곳만 자식 네 개



혼합 노드 3개 + 단색 리프 10개 = 총 13회 호출

선이 이어진 부모만 네 자식으로 나눕니다. 파란 상자 3개는 각각 괄호 1쌍입니다.

복원 ①: 괄호를 만나면 네 영역으로

(0 1 ((0001)000) 0)

읽은 표현	복원할 위치	하는 일
첫 (	전체 8×8	크기 4의 네 자식으로 내려감
0	좌상 4×4	16칸을 모두 0으로 채움
1	우상 4×4	16칸을 모두 1로 채움
다음 (	좌하 4×4	크기 2의 네 자식으로 내려감

복원은 위치와 크기를 함께 전달합니다. 색 한 글자는 현재 영역 전체를 채웁니다.

복원 ②: 좌하의 겹친 괄호를 읽습니다

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

행 / 열은 0부터 셉니다.

좌하 표현: ((0001)000)

첫 2×2: (0001)
→ 네 픽셀을 0, 0, 0, 1로

나머지 2×2 세 영역: 0, 0, 0
→ 각 영역 전체를 흰색으로

행 5, 열 1의 검은 칸도 복원!

닫는 괄호를 만나면, 그 괄호가 시작한 부모 영역으로 돌아옵니다.

복원 ③: 마지막 0을 채우면 원본과 같습니다

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

행 / 열은 0부터 셉니다.

우하의 "0" → 16칸을 흰색으로

루트의 ")" → 전체 복원 종료

검은 칸 위치가 원본과 같습니다.

저장·복원의 자식 순서는 동일!

64개 픽셀의 색이 모두 복원됩니다.

0 = 흰색, 1 = 검은색. 현재 호출의 영역만 처리한 뒤 결과를 반환합니다.

모든 그림이 더 작아지는 것은 아닙니다

단색 4×4

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

16개 값 → 1글자

체커보드 4×4

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

16개 값 → 26글자

균일 영역은 유리하고, 끝까지 섞인 무늬는 괄호 때문에 표현이 더 길어질 수 있습니다.

함수의 입력: 지금 맡은 영역의 위치와 크기

`quadtree_compress(x, y, n, image)`
현재 영역: (시작 행 x, 시작 열 y, 한 변 n)

강조한 영역: x=0, y=0, n=2

0	0	1	1
0	1	1	1
0	0	0	0
0	0	0	0

행: x부터 x+n-1까지
열: y부터 y+n-1까지

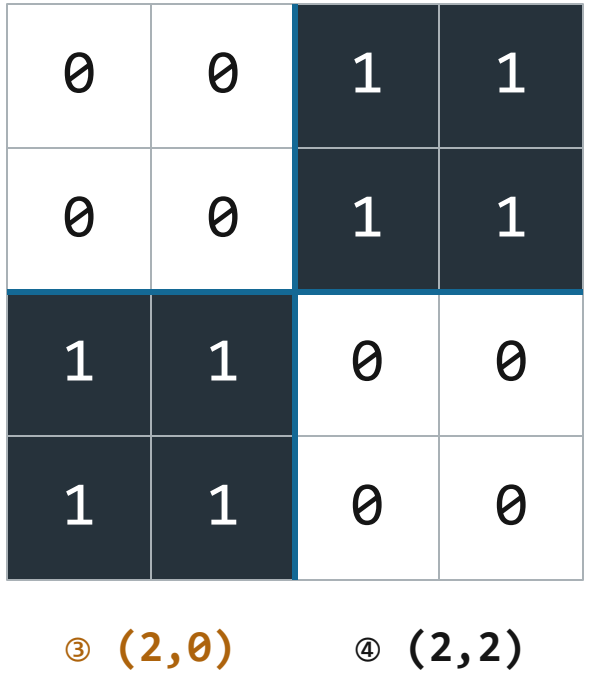
모든 색이 첫 칸과 같으면
return str(color)

n은 한 변의 칸 수이며 1, 2, 4, 8, ...입니다. 이 영역의 전체 칸 수는 n×n입니다.

혼합 영역의 네 호출: 크기는 절반으로

예: x=0, y=0, n=4

① (0,0) ② (0,2)



③ (2,0) ④ (2,2)

```
half = n // 2

① (x,          y,          half)
② (x,          y + half,   half)
③ (x + half,   y,          half)
④ (x + half,   y + half,   half)
```

네 자식이 전체 영역을 빠짐없이 나누어 맡습니다. 자식의 크기는 모두 half입니다.

저장 함수의 흐름: 검사, 반환 또는 분할



한 칸은 반드시 단색입니다. 모든 호출은 자신이 맡은 영역의 기록을 반환합니다.

파이썬 구현 : 8×8 이미지 압축

quadtree_compress.py

```
def quadtree_compress(x, y, n, image):
    color = image[x][y]
    is_uniform = True
    for i in range(x, x + n):
        for j in range(y, y + n):
            if image[i][j] != color:
                is_uniform = False
                break
        if not is_uniform:
            break
    if is_uniform:
        return str(color)
    half = n // 2
    result = "("
    result += quadtree_compress(x, y, half, image)
    result += quadtree_compress(x, y + half, half, image)
    result += quadtree_compress(x + half, y, half, image)
    result += quadtree_compress(x + half, y + half, half, image)
    result += ")"
    return result
```

example.py

```
example_image = [
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 1,1,1,1],
    [0,0,0,0, 0,0,0,0],
    [0,1,0,0, 0,0,0,0],
    [0,0,0,0, 0,0,0,0],
    [0,0,0,0, 0,0,0,0],
]
print(quadtree_compress(0, 0, 8, example_image))
```

OUTPUT

(01((0001)000)0)

픽셀 64개 → 문자 16개

짧게 저장해도, 같은 색인지 검사는 합니다

단색 4×4

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

처음부터 다른 색이 나옴

0	1	0	0
0	0	0	0
0	0	0	0
0	0	0	0

16칸을 확인한 뒤 0 반환

저장 기호의 개수와 픽셀 검사 횟수는 서로 다른 수입니다.

혼합 확인 뒤 네 자식 처리

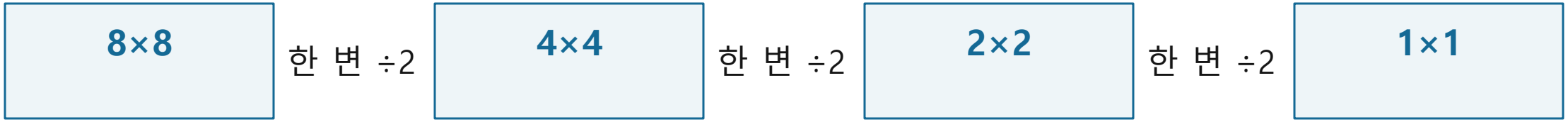
한 층의 영역들을 합치면 전체 그림

한 변 S = 8, 전체 픽셀 N = 8×8 = 64

영역 크기	그 층의 최대 영역 수	검사량의 상한
8×8	1개	1 × 64 = 64칸
4×4	4개	4 × 16 = 64칸
2×2	16개	16 × 4 = 64칸
1×1	64개	64 × 1 = 64칸

현재 영역을 끝까지 검사하는 모델입니다. 같은 색에서 멈추거나 검사에서 일찍 나가면 줄어듭니다.

쿼드 트리의 깊이와 검사 시간의 상한



3번 내려감, 검사하는 층은 최대 4개

색 검사량 ≤ 전체 픽셀 수 × 검사하는 층 수

$$O(N \log S) = O(S^2 \log S)$$

검사 후 분할하는 212 코드의 최악 상한입니다. S는 한 변, $N=S^2$는 픽셀 수입니다.

자식 결과를 합치는 상향식: O(S²)

한 칸을 읽고, 부모는 자식 네 개의 정보만 합칩니다.

비교 기준	212 코드: 검사 후 분할	상향식: 자식 결과 결합
최악 시간	O(S² log S) 상한	Θ(S²)
단색이 많을 때	단색 확인 후 재귀 중단	모든 픽셀을 한 번씩 읽고 결합
코드 난이도	현재 검사·분할 코드 사용	단색 정보와 자식 노드를 반환

픽셀 S²개 + 일정한 비용의 노드 결합 = 전체 Θ(S²)

S는 한 변입니다. 전체가 단색이면 두 방식 모두 Θ(S²). 긴 문자열을 매번 복사하지 않는 결합을 가정합니다.

응용 ② 게임 개발과 충돌 감지

객체 1,000개의 모든 쌍을 비교하면 한 프레임에 499,500번 검사합니다.

무식한 접근: 브루트 포스

$$(n-1) + (n-2) + \dots + 1 = n(n-1)/2 \approx n^2/2$$

① 공간 분할 객체가 많은 구역을 네 부분으로 나누고, 필요한 곳만 더 나눕니다.

② 객체 등록 위치와 크기를 고려해 객체를 구역에 등록합니다.

③ 후보 조회 겹칠 수 있는 이웃 구역과 경계를 걸친 객체까지 포함해 검사합니다.

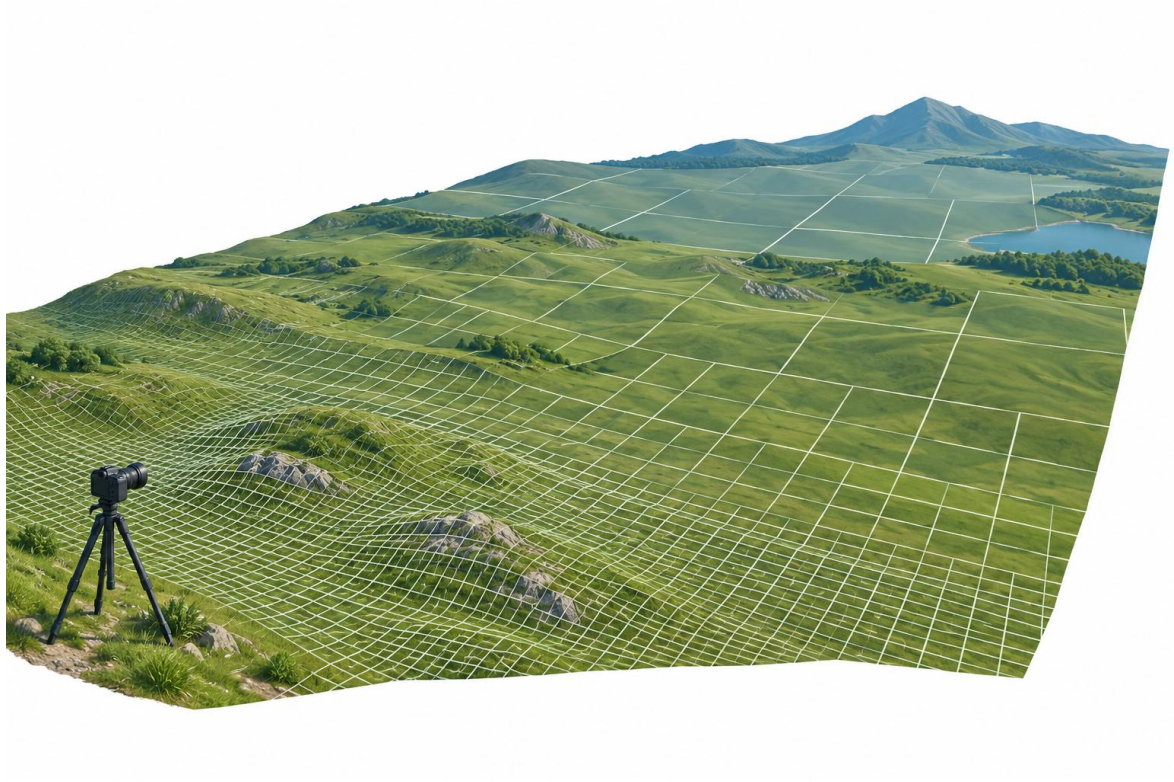
"서로 멀리 떨어져 있는 물체는 충돌 후보에서 제외할 수 있습니다."

응용 ③ 지형 LOD

**카메라에 가까운 지형은 작은 격자로,
먼 지형은 큰 격자로 표현합니다.**

거리로 필요한 세밀함을 정합니다.
화면에서 보이는 오차도 고려합니다.

응용	더 나누는 기준
이미지 압축	색이 섞인 영역
충돌 감지	객체가 많은 영역
지형 LOD	카메라 거리·화면 오차



카메라와 가까울수록 작은 구역으로 분할한 지형의 예

회사 전체의 일: 직급별 작업량을 모두 더하기

사장 1명
부장 4명
직원 16명

한 사람 = 재귀 호출 하나
부하에게 맡긴 일 + 자기의 추가 작업

한 직급의 일 = 인원 수 × 한 사람이 직접 하는 일

회사 전체의 일 = 모든 직급의 합계

위임한 일을 자기의 직접 작업에도 다시 더하면 중복 계산입니다.
여러 사람이 동시에 끝내는 시간을 계산하는 비유는 아닙니다.

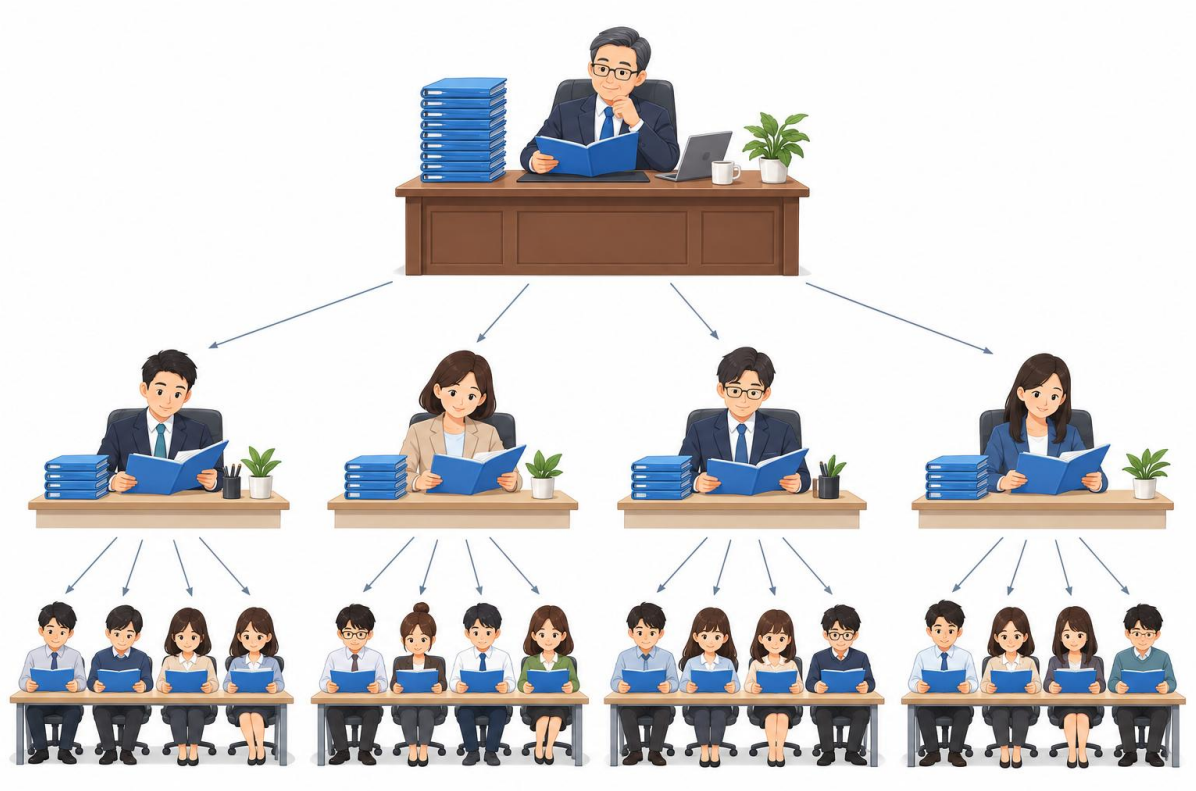
경우 A: 직급마다 전체 작업량이 같음

직급	인원	개인 작업	합계
사장	1	16	16
부장	4	4	16
직원	16	1	16

16 + 16 + 16 = 48

한 직급의 일 × 직급 수

$\Theta(n \log n)$



예시: 입력 n=16을 4등분합니다.
병합 정렬은 2등분하지만 같은 원리입니다.

왜 직급 수가 log인가? 나누는 횟수 세기

입력 16: 16 ÷ 4 = 4, 4 ÷ 4 = 1

나누기 2번, 직급은 3개

입력 64: 64, 16, 4, 1

입력이 4배 커져도 직급은 하나만 늘어납니다.

직급 수 = $\log_4 n + 1$

한 층의 일 $n \times (\log_4 n + 1) = \Theta(n \log n)$

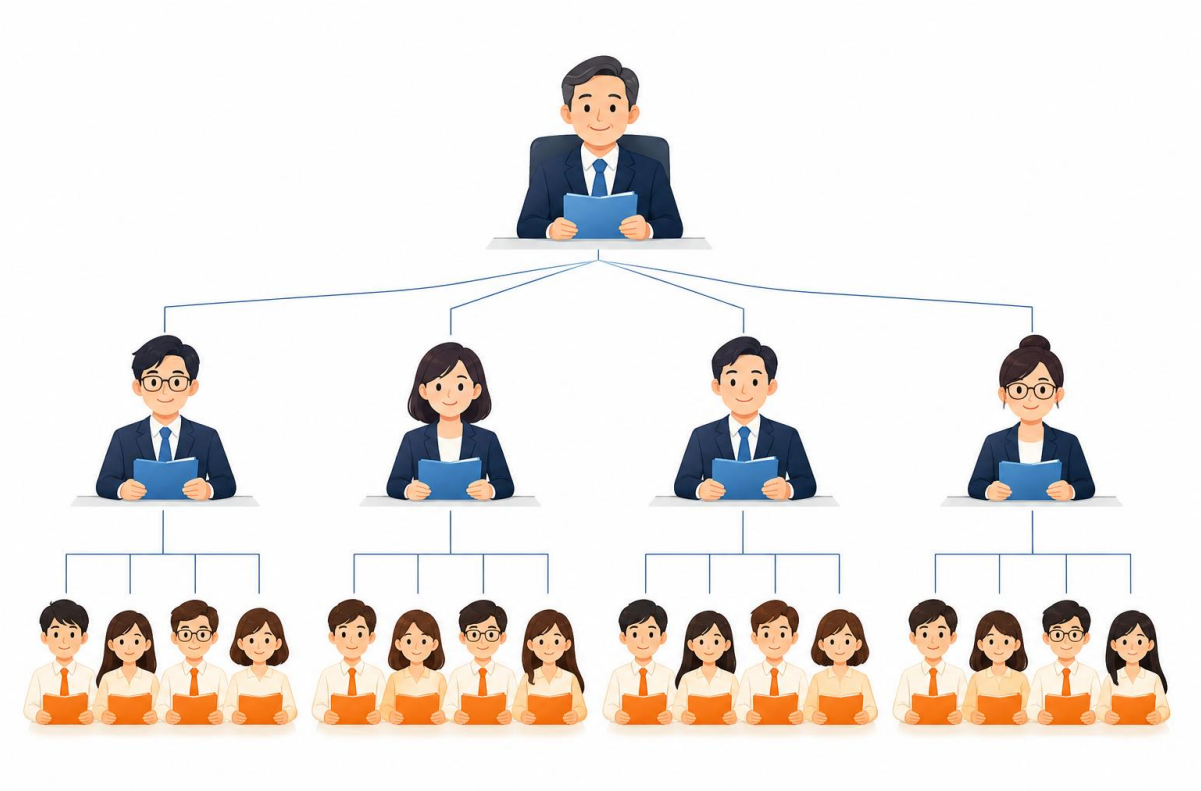
경우 B: 아래로 갈수록 전체 작업량이 커짐

직급	인원	개인 작업	합계
사장	1	1	1
부장	4	1	4
직원	16	1	16

1 + 4 + 16 = 21

21 < 2 × 16

맨 아래 일과 같은 차수: Θ(n)



직급 수를 곱하지 않습니다.
위 직급들이 같은 양을 반복하지 않습니다.

아래층이 많아도 log를 곱하지 않는 이유

입력 n	직급별 합계	전체 작업량
16	1, 4, 16	21 < 2 × 16
64	1, 4, 16, 64	85 < 2 × 64

맨 아래부터: n + n/4 + n/16 + … < 2n

맨 아래에서 이미 n만큼 일합니다. 전체도 n과 같은 차수입니다.

각 직급의 양이 다릅니다. n에 직급 수를 곱하면 실제보다 크게 셉니다.

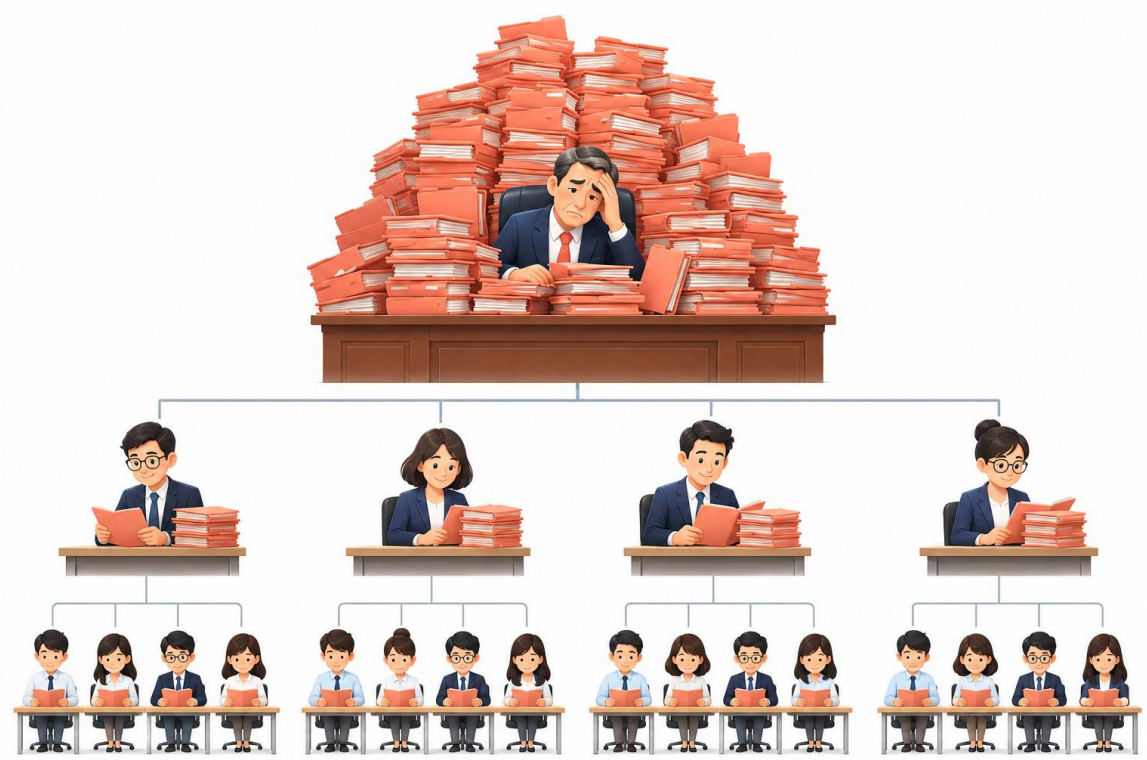
경우 C: 사장에게 전체 작업량이 몰림

직급	인원	개인 작업	합계
사장	1	$16^2 = 256$	256
부장	4	$4^2 = 16$	64
직원	16	1	16

$256 + 64 + 16 = 336$

$336 < 2 \times 256$

사장의 일과 같은 차수: $\Theta(n^2)$



위에서 아래로 직급 합계가 1/4씩 줄어듭니다.
같은 양이 여러 직급에서 반복되지 않습니다.

세 경우: 한 사람이 직접 하는 일이 다릅니다

1명의 추가 작업	직급별 합계 (n=16)	전체 시간
받은 크기만큼	16, 16, 16	$\Theta(n \log n)$
일정한 일	1, 4, 16	$\Theta(n)$
받은 크기의 제곱	256, 64, 16	$\Theta(n^2)$

같은 회사 구조라도 추가 작업이 다르면 전체 시간도 달라집니다.

세 예제 모두 자식은 4개, 각 자식의 입력 크기는 1/4입니다.

쿼드 트리 두 방식: 층별 작업량을 나란히

검사 후 분할: 영역을 다시 읽는 모델

층	호출 수	호출당 검사	층 합계
0	1	64	64
1	4	16	64
2	16	4	64
3	64	1	64

$4T(n/2) + n^2$
모델: $\Theta(n^2 \log n)$

상향식: 자식 네 결과만 합치는 모델

층	호출 수	호출당 일	층 합계
0	1	1	1
1	4	1	4
2	16	1	16
3	64	1	64

$4T(n/2) + 1$
$\Theta(n^2)$

n=S는 한 변, 예시는 S=8입니다. 왼쪽은 전체 검사 모델이며 212 코드에는 $O(S^2 \log S)$ 최악 상한을 적용합니다.

기호는 회사에서 맡긴 일과 직접 한 일

$$T(n) = a \times T(n/b) + f(n)$$

T(n): 자식 호출까지 포함한 전체 시간
a: 끝까지 풀 자식의 개수
b: 자식 입력 크기를 구할 때 나누는 수
f(n): 현재 호출의 직접 작업

예제	a: 자식 수	b: 나누는 수
회사 예제	4	4
병합 정렬	2	2
쿼드 트리 (한 변 S)	4	2

기준 n^p: 맨 아래 칸 개수

맨 아래 호출 하나는 일정한 일을 하고 끝납니다.

맨 아래 칸 개수 = $n^{\log_b a} = n^p$

구조	$b^p = a$	p	맨 아래 칸·직원 수
회사: a=4, b=4	$4^1 = 4$	1	n
병합 정렬: a=2, b=2	$2^1 = 2$	1	n
쿼드 트리: a=4, b=2	$2^2 = 4$	2	S^2

p는 $b^p = a$인 수, 즉 밑이 b인 a의 로그입니다. f(n)을 맨 아래 전체 비용 n^p과 비교합니다.

마스터 정리: 추가 작업과 기준 비용 비교

추가 작업 $f(n) = \Theta(n^d)$, 기준 p는 $b^p = a$

지수 비교	일이 몰리는 곳	전체 시간
$d < p$	맨 아래 직원들	$\Theta(n^p)$
$d = p$	모든 직급이 비슷함	$\Theta(n^p \log n)$
$d > p$	사장	$\Theta(n^d)$

회사 예제의 p=1: d=0이면 n, d=1이면 n log n, d=2이면 n^2

$a \geq 1$, $b > 1$, $d \geq 0$, $T(1) = \Theta(1)$. $f(n) = n^p / \log n$처럼 로그 정도만 다른 식은 이 거듭제곱형 기본표 밖입니다.

직접 대입해 보기: 병합 정렬

$$T(n) = 2T(n/2) + \Theta(n)$$

확인할 값	코드에서 읽은 뜻	값
a	왼쪽과 오른쪽을 둘 다 정렬	2
b	각 구간의 길이가 절반	2
d	현재 병합은 길이 n에 비례	1
p	$2^1 = 2$이므로 기준 지수는 1	1

d = p: 한 층의 n × 로그 차수의 층 수 = Θ(n log n)

이진 탐색은 a=1이라 Θ(log n). 퀵 정렬은 자식 크기가 달라 같은 두 절반 식을 항상 쓸 수 없습니다.

4.7

코딩 테스트

분할 정복으로 실전 문제 두 개를 직접 풀어본다

색종이 만들기

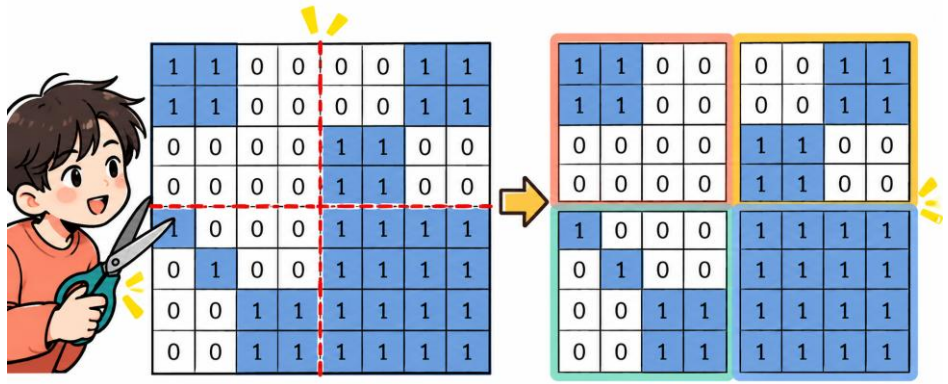
여러 개의 정사각형 칸(N × N)으로 이루어진 종이가 있다. 각 칸은 하얀색 아니면 파란색이다. 전체 종이가 모두 같은 색이 아니면, 가로와 세로의 중앙을 잘라 똑같은 크기의 네 개의 N/2 × N/2 색종이로 나눈다. 이 과정을 잘라진 종이가 모두 같은 색이 되거나 한 칸이 될 때까지 반복한다.

최종적으로 하얀색 색종이와 파란색 색종이가 각각 몇 장인지 구하라.

쿼드 트리와 완전히 동일한 구조

검사(check) → 균일하면 카운트, 아니면 4등분

한 칸(n = 1)은 언제나 균일하므로 재귀는 반드시 끝난다



입력 (N = 8)

`1 1 0 0 0 0 1 1  …  0 0 1 1 1 1 1 1`

출력

하얀색 9장, 파란색 7장

풀이 : 재귀적으로 4등분하며 세기

solve.py

```
import sys
sys.setrecursionlimit(10 ** 6)
def solve(x, y, n):
    global white_cnt, blue_cnt
    color = paper[x][y]
    for i in range(x, x + n):
        for j in range(y, y + n):
            if paper[i][j] != color:
                nn = n // 2
                solve(x,      y,      nn)
                solve(x,      y + nn, nn)
                solve(x + nn, y,      nn)
                solve(x + nn, y + nn, nn)
                return
    if color == 0:
        white_cnt += 1
    else:
        blue_cnt += 1
```

run.py

```
N = 8
white_cnt = blue_cnt = 0
solve(0, 0, N)
print(f"하얀색 색종이: {white_cnt}")
print(f"파란색 색종이: {blue_cnt}")
```

OUTPUT

```
하얀색 색종이: 9
파란색 색종이: 7
```

return 위치가 핵심

다른 색을 찾으면 네 자식을 호출하고
바로 return합니다.
이미 나눈 영역을 다시 세지 않습니다.

색종이 만들기: 넓이와 조각 수를 구분

1	1	1	1
1	1	1	1
1	1	1	1
1	1	1	1

단색 4×4

1인 픽셀: 16개
최종 종이 조각: 1개

단색 확인은 16칸 검사
분할 재귀는 필요 없습니다.

한 변과 픽셀 수를 구분하세요.

검사 비용: 단색 $\Theta(N)$. 반복 전체 검사 모델 $O(N \log S)$ 상한, $N = S^2$.

곱셈: 거듭제곱 나머지를 빠르게 계산하기

A를 B번 곱한 값의 C에 대한 나머지를 구합니다.
절반 지수의 결과를 한 번 구해 저장한 뒤 재사용합니다.

B가 짝수 $A^B = A^{B/2} \times A^{B/2}$

B가 홀수 $A^B = A^{\lfloor B/2 \rfloor} \times A^{\lfloor B/2 \rfloor} \times A$

모듈러 성질 $(a \times b) \bmod c = ((a \bmod c) \times (b \bmod c)) \bmod c$

$T(B) = T(\lfloor B/2 \rfloor) + \Theta(1)$ 전체 $\Theta(\log B)$

예제: $10^{11} \bmod 12 = 4$. 지수 최대 약 21억도 약 31단계입니다. 정해진 크기의 정수 연산을 상수로 셉니다.

풀이 : 분할 정복 거듭제곱 (fast power)

power.py

```
def power(a, b, c):
    if b == 0:
        return 1 % c
    half = power(a, b // 2, c)
    result = (half * half) % c
    if b % 2 == 1:
        result = (result * a) % c
    return result
```

run.py

```
A, B, C = map(int, "10 11 12".split())
print(power(A, B, C))
```

OUTPUT **4**

재귀 호출 추적

power(10, 11, 12)	b = 11 (홀수)
power(10, 5, 12)	b = 5 (홀수)
power(10, 2, 12)	b = 2 (짝수)
power(10, 1, 12)	b = 1 (홀수)
power(10, 0, 12)	return 1 % c

깊이는 $\log_2 B$

지수 B가 약 21억일 때도
절반으로 줄이는 단계는 약 31번입니다.

빠른 거듭제곱: $3^{11} \bmod 7$을 끝까지

11	5	2	1	0

돌아온 지수 b	half	계산 후 나머지
0	기저 사례	1
1	1	$1 \times 1 \times 3 \bmod 7 = 3$
2	3	$3 \times 3 \bmod 7 = 2$
5	2	$2 \times 2 \times 3 \bmod 7 = 5$
11	5	$5 \times 5 \times 3 \bmod 7 = 5$

각 단계는 나머지를 저장합니다. 결과: $3^{11} \bmod 7 = 5$.

지수 절반 호출 한 번: O(log b)

```
half = power(a, b // 2, c)  ← 한 번만 호출
```

```
result = (half * half) % c
```

홀수 b일 때만 a를 한 번 더 곱합니다.

T(b) = T(⌊b/2⌋) + O(1) → O(log b)

정해진 크기의 정수 연산을 상수로 세는 기준입니다. b = 0은 바로 반환합니다.

4장 정리

분할 정복의 3단계

분할(divide) → 정복(conquer) → 결합(combine). 기저 사례에 도달할 때까지 재귀적으로 나눈다.

결합이 알고리즘의 핵심

병합 정렬은 나누기가 쉽고 합치기가 어렵다. 반대로 퀵 정렬은 나누기가 어렵고 합치기가 쉽다.

연속 구간의 경계를 확인합니다

최대 구간 합은 왼쪽·오른쪽 내부와 중앙을 걸친 구간을 모두 비교합니다.

공간도 분할한다

쿼드 트리는 평면을 1/4씩 재귀 분할한다. 이미지 압축, 충돌 감지, LOD의 기반이다.

마스터 정리

T(n) = aT(n/b) + f(n) 형태와 각 경우의 적용 조건을 확인한 뒤 비용의 크기를 비교합니다.

종료 조건과 재귀 구간

빈 구간과 한 원소에서 종료합니다. 퀵 정렬은 확정된 피벗을 제외한 두 구간만 다시 정렬합니다.

Q & A

질문을 나누어 봅시다.
큰 질문도 작게 쪼개면 답할 수 있습니다.