1주차: 해시와 두 포인터

센서 로그의 가까운 중복

같은 이벤트가 재전송되어 [4, 7, 4, 4]가 기록되었습니다. 같은 값이 한 칸 이내에 다시 나타나는지 확인하려고 합니다. 첫 번째 4와 세 번째 4만 비교하면 마지막 두 4를 놓칩니다.

매번 모든 과거 기록을 다시 읽으면 로그가 길어질수록 비교가 빠르게 늘어납니다. 이번 차시에서는 질문에 필요한 과거 정보만 저장하고, 정렬된 입력에서는 답이 될 수 없는 후보를 제거합니다.

1주차의 문제와 학습 목표

1차시: 배열·집합·사전의 연산 비용, 값과 이전 위치의 연결. 2차시: 정렬된 값에서 두 포인터 이동의 근거와 세 값의 합으로의 확장

- 입력·출력·제약 정의
- 저장 상태의 의미와 후보 제거 근거
- Python 구현의 시간·보조 공간·출력 비용

실습: LeetCode 1, 219, 167, 15

입력 크기와 한 번의 작업

n은 현재 컨테이너의 원소 수입니다. O(1)은 n이 커져도 작업 횟수가 일정한 정도라는 뜻이고, O(n)은 최악의 작업 횟수가 n에 비례한다는 뜻입니다. O(1)이 항상 실제 실행시간까지 더 짧다는 뜻은 아닙니다.

작업	n=4	n=8	증가 방식
지정한 칸 하나 읽기	1칸	1칸	일정
없는 값을 끝까지 찾기	4번 비교	8번 비교	n에 비례
서로 다른 두 위치 모두 비교	6쌍	28쌍	n(n-1)/2

이 강의의 수치 계산은 입력 범위 안의 정수·참조 연산을 O(1)로 셉니다. 큰 정수, 긴 문자열의 해시·비교는 길이에 따른 비용을 따로 계산합니다. Big-O는 증가율의 상한이며 최악·기대·분할상환 중 무엇을 분석하는지도 함께 밝혀야 합니다.

list: 위치로 읽기와 값으로 찾기

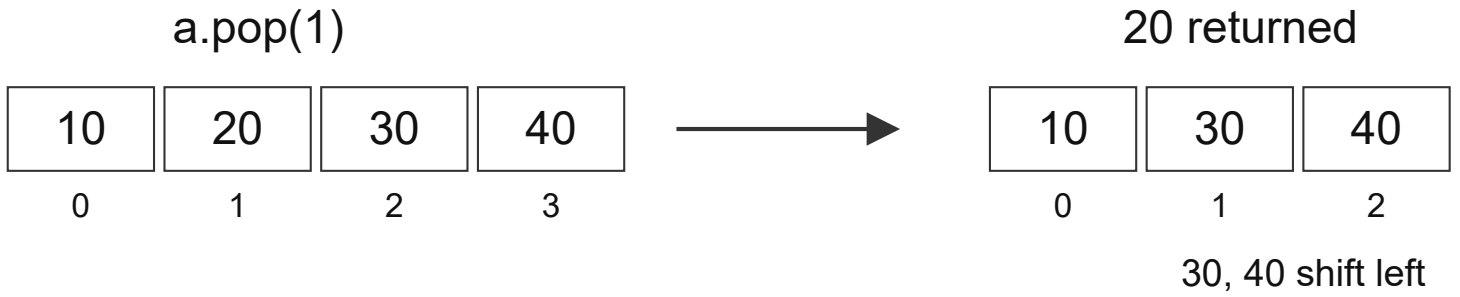
`a=[10,20,30,40]`에서 `a[2]`는 세 번째 칸으로 바로 접근합니다. CPython list는 원소 참조를 배열에 저장하므로 앞의 두 값을 비교하지 않습니다. 반면 `30 in a`는 찾는 값의 위치를 모르므로 앞에서부터 비교합니다.

표현	확인한 값	결과	최악 시간
`a[2]`	지정한 2번 칸	30	O(1)
`a[2] = 99`	지정한 2번 칸	값 교체	O(1)
`30 in [10,20,30,40]`	10,20,30	True	O(n)
`50 in [10,20,30,40]`	10,20,30,40	False	O(n)
`a.index(40)`	앞에서 40까지	인덱스 3	O(n)
`len(a)`	저장된 길이	4	O(1)

인덱스를 이미 아는 것과 값을 찾아 인덱스를 알아내는 것은 다른 작업입니다. `.index`는 값이 없으면 ValueError를 냅니다.

list: 어디에서 제거하는가

`a=[10,20,30,40]`에서 `pop(1)`은 값 20을 반환하고 뒤의 30·40을 한 칸씩 앞으로 옮깁니다. 원소가 이미 어디 있는지 알아도 빈칸을 메우는 작업이 남습니다.



표현	수행 작업	시간
`a.pop()`	맨 뒤 제거, 뒤 원소 이동 없음	분할상환 O(1)
`a.pop(i)`, `del a[i]`	뒤의 n-i-1개 참조 이동	일반 위치 최악 O(n)
`a.remove(x)`	첫 x 검색 후 뒤 참조 이동	최악 O(n)
`a.insert(0,x)`	기존 참조를 모두 한 칸 뒤로	O(n)

`pop(i)`의 i는 위치, `remove(x)`의 x는 값입니다. 예를 들어 `remove(20)`은 두 번째 위치라는 뜻이 아닙니다. 리스트 재할당 비용은 분할상환 비용에 포함합니다.

append와 분할상환 O(1)

list는 현재 길이보다 여유 있는 저장 공간을 확보합니다. 여유 칸이 있으면 append는 끝에 참조 하나를 기록합니다. 공간이 부족하면 더 큰 저장 공간을 확보하고 기존 참조들을 옮길 수 있어 그 한 번은 O(n)입니다.

경우	한 번의 작업	여러 번 누적한 비용
여유 공간이 있는 append	끝에 한 원소 기록	O(1)
저장 공간을 늘리는 append	기존 n개 참조 이동 가능	그 호출은 O(n)
빈 리스트에 N번 append	확장 비용을 전체 호출에 나눔	합 O(N), 분할상환 O(1)

분할상환은 입력이 무작위라는 가정이 아닙니다. 일련의 연산 전체 비용을 연산 수로 나눈 분석입니다. 한 번의 호출이 언제나 같은 시간이라는 보장과 구분합니다.

마지막 원소 pop도 재할당을 포함하면 분할상환 O(1)입니다. 이 강의에서 list를 스택으로 사용하는 코드는 이 비용을 적용합니다.

확장 비용을 여러 호출에 나누기

용량을 1,2,4,8칸으로 두 배씩 늘리는 배열에서 append 8회를 세어봅니다. CPython의 실제 증설 비율을 재현한 표는 아니며, 비율을 정해 확장할 때 총 이동량이 어떻게 제한되는지 계산합니다.

추가 후 길이	필요한 새 용량	옮기는 기존 참조 수
1	1	0
2	2	1
3	4	2
4	유지	0
5	8	4
6~8	유지	0

새 값 기록 8회와 복사 1+2+4=7회를 합하면 15회입니다. N이 2의 거듭제곱이면 복사 합은 N−1, 총 기록은 2N−1입니다. N이 그 사이에 있어도 복사 합은 2N보다 작아 총비용은 O(N)입니다. 반면 용량을 한 칸씩만 늘리면 1+2+…+(N−1)을 복사하므로 $O(N^2)$이 됩니다.

전체 O(N)을 호출 N회로 나누면 append의 분할상환 비용은 O(1)입니다.

복사와 슬라이싱의 시간·메모리

```
a = [10, 20, 30, 40]
b = a
c = a[1:3]
a[1] = 99
print(b)  # [10, 99, 30, 40]
print(c)  # [20, 30]
```

b=a는 같은 리스트를 가리키는 참조를 하나 추가합니다. 리스트 전체를 복사하지 않아 시간·보조 공간 모두 O(1)입니다. a[1:3]은 참조 2개를 새 리스트에 복사하여 이후 슬롯 교체가 분리됩니다.

작업	시간	새로 생기는 공간
q개 원소 슬라이스	O(q)	O(q)
a.copy(), list(a)	O(n)	O(n)
sorted(a)	최악 O(n log n)	복사·정렬 작업 공간 O(n)

예를 들어 rows=[[1],[2]]; copied=rows.copy() 뒤 copied[0].append(9)를 실행하면 두 목록 모두 첫 행이 [1,9]입니다. 바깥 슬롯만 복사했고 그 슬롯이 가리키는 첫 행 객체는 공유하기 때문입니다. 메모리는 처리 횟수가 아니라 동시에 유지되는 저장량입니다.

stack과 queue: 꺼내는 순서

스택은 마지막에 넣은 값을 먼저 꺼내는 LIFO, 큐는 먼저 넣은 값을 먼저 꺼내는 FIFO입니다. 자료의 사용 순서이며, Python에서는 각각 list의 뒤쪽 연산과 deque의 양 끝 연산으로 구현합니다.

```
from collections import deque
stack = [10, 20, 30]
queue = deque([10, 20, 30])
print(stack.pop())     # 30
print(queue.popleft()) # 10
```

목적	연산	비용과 이유
스택에 넣고 꺼내기	list.append / pop	뒤쪽만 수정, 분할상환 O(1)
큐에 넣고 꺼내기	deque.append / popleft	양 끝에서 O(1), 전체 이동 없음
앞쪽에도 넣기	deque.appendleft	O(1)
deque 가운데 접근·값 검색	q[i], x in q	최악 O(n)

deque는 양 끝 작업에 적합하며 임의 위치 접근이 list처럼 O(1)인 것은 아닙니다. 4–6주차 BFS에서 사용하는 큐에도 같은 비용이 적용됩니다.

dict와 set: 키로 찾을 정보

dict는 키에 연결된 값을, set은 값의 존재 여부를 저장합니다. 해시값으로 조사할 위치를 좁혀 보통 모든 원소를 순서대로 비교하지 않습니다. 충돌이 많으면 비교가 늘 수 있습니다.

```
latest = {4: 2, 7: 1}
print(4 in latest)          # True: 키 검사
print(latest[4])            # 2: 연결된 값
print(2 in latest.values()) # True: 값들을 검색
```

연산	기대/평균 시간	공간 변화
`key in d`, `d[key]`, `d.get(key)`	O(1)	새 항목 없음
`d[key]=v`, `d.pop(key)`	O(1), 확장 비용 분할상환	키 추가/제거
`x in s`, `s.add(x)`, `s.discard(x)`	O(1), 확장 비용 분할상환	원소 추가/제거
`v in d.values()`	최악 O(n)	보기 객체만 추가
`d.copy()`, `set(a)`	기대 O(n)	최대 O(n)

상수 비용 해시·동등 비교와 잘 분산된 키를 가정합니다. 심한 충돌 시 조회도 최악 O(n)이며, 긴 문자열의 해시 비용을 무조건 O(1)로 볼 수는 없습니다.

한 연산의 비용을 전체 코드로 합산

같은 데이터를 전부 제거해도 선택한 연산에 따라 총 비용이 달라집니다. n개 리스트의 앞을 계속 제거하면 이동하는 참조 수는 n-1, n-2, …, 0개입니다.

코드의 반복 작업	한 번	횟수 또는 총 이동량	전체
list의 맨 뒤 pop	분할상환 O(1)	n번	O(n)
list의 맨 앞 pop(0)	현재 길이에 비례	n(n-1)/2회 이동	$O(n^2)$
deque의 popleft	O(1)	n번	O(n)
매 값마다 이전 list에서 검색	최대 O(n)	n번	$O(n^2)$
매 값마다 dict 키 조회	기대 O(1)	n번	기대 O(n)

합을 구하며 누적 값 하나만 유지하는 코드는 n번 더하므로 시간 O(n), 보조 공간 O(1)입니다. 반복 횟수와 저장량은 따로 셉니다.

입력·보조 공간·출력 공간

구현	입력 외에 저장하는 것	보조 공간	출력 공간
합만 반환	합·현재 값	O(1)	O(1)
누적 합 리스트 반환	합·현재 값	O(1)	O(n)
본 값들을 set에 저장	서로 다른 d개 값	O(d)	bool O(1)
크기 n의 복사 후 정렬	복사본·정렬 작업 공간	O(n)	반환 형식에 따라 다름
깊이 h의 재귀	아직 끝나지 않은 호출 h개	O(h)	반환 형식에 따라 다름

공간은 실행 중 가장 많이 동시에 사용하는 시점을 기준으로 셉니다. 매번 작은 변수를 새 값으로 덮어쓰는 것과 결과 리스트에 계속 append하는 것은 다릅니다.

출력 공간을 보조 공간과 분리해서 적습니다. 반환 리스트가 n칸이고 작업용 변수만 O(1)이면 “보조 O(1), 출력 O(n), 합계 O(n)”입니다. Python 객체의 실제 바이트 수는 구현과 값의 크기에 따라 달라집니다.

CPython 연산 복잡도 · deque 문서

LeetCode 1: 문제 정의와 단순 풀이

Two Sum: 서로 다른 두 위치의 합이 target인 인덱스를 반환합니다. 원문은 답 하나를 보장하며 같은 위치를 두 번 사용할 수 없습니다.

`nums=[4,8,3], target=7`의 답은 `[0,2]`입니다. 모든 쌍을 검사하면 추가 저장 없이 풀 수 있습니다.

```
def two_sum_brute(nums, target):
    for left in range(len(nums)):
        for right in range(left + 1, len(nums)):
            if nums[left] + nums[right] == target:
                return [left, right]
    return []
```

쌍의 개수와 반복 조회

서로 다른 두 위치의 쌍은 다음만큼 있습니다. n=10,000이면 49,995,000쌍입니다. 실제로는 답을 일찍 찾을 수도 있지만 최악에는 이 수에 비례해 비교합니다.

$$\frac{n(n-1)}{2}$$

현재 값이 3이고 target이 7이라면 필요한 과거 값은 4 하나입니다. 모든 과거 값을 다시 비교하는 대신, 4의 존재 여부와 위치를 바로 조회할 상태를 만듭니다.

Two Sum의 이전 값 상태

`seen[value]`는 이미 지난 위치 하나입니다. 현재 위치를 읽기 전에 저장하지 않으므로 사전에서 찾은 위치와 현재 위치가 다릅니다.

```
def two_sum(nums, target):
    seen = {}
    for index, value in enumerate(nums):
        if target - value in seen:
            return [seen[target - value], index]
        seen[value] = index
    return []  # Defensive extension: official inputs have one answer.
```

각 반복은 현재 값의 짝 `target-value`를 사전에서 조회합니다. 찾지 못했을 때만 현재 값을 저장하므로 같은 위치를 두 번 쓰지 않습니다. `nums=[4,8,3]`에서 3을 읽을 때 사전의 4:0이 인덱스 0을 알려줍니다.

index	value	필요한 값	조회 전 seen	결과
0	4	3	{}	없음 → 4:0 저장
1	8	-1	{4:0}	없음 → 8:1 저장
2	3	4	{4:0,8:1}	발견 → [0,2]

저장된 값의 개수는 지금까지 읽은 서로 다른 값의 수입니다. 사전에는 미래 위치가 들어 있지 않습니다.

Two Sum의 정당성과 갱신 순서

반복 시작 때 seen은 이전 위치만 포함합니다. 답의 두 번째 위치에 도착하면 첫 위치의 값이 저장되어 있으므로 답을 찾습니다.

`nums=[3,3], target=6`에서 현재 값을 먼저 저장하면 조회 결과가 현재 인덱스 자신이 됩니다. 첫 3에서는 조회만 실패하고 저장한 뒤, 둘째 3에서 `[0,1]`을 반환해야 합니다.

각 원소에서 사전 조회·저장 횟수는 고정입니다. 기대 O(1)인 이 작업을 n번 하므로 기대 시간 O(n)입니다. 보조 사전은 최악에 n개의 서로 다른 값과 위치를 담아 O(n), 반환 인덱스 두 개는 O(1)입니다. 심한 해시 충돌까지 포함한 시간 상한은 $O(n^2)$입니다.

LeetCode 219: 거리 조건의 추가

Contains Duplicate II: 같은 값의 서로 다른 두 위치가 거리 k 이내인지 판정합니다.

`nums=[4,7,4,4], k=1`은 마지막 두 위치 때문에 True입니다. k=0이면 서로 다른 위치를 만족할 수 없습니다.

각 위치에서 앞의 최대 k개를 다시 보면 O(n·min(n,k)) 시간. 슬라이스를 만들면 반복마다 복사 비용도 듭니다. 값마다 어떤 과거 위치 하나를 남기면 충분한지 살펴봅니다.

가장 최근 인덱스의 의미

`latest[value]`는 그 값이 마지막으로 나타난 위치입니다. 현재 위치와 가장 가까운 과거 위치이므로 이것도 너무 멀면 더 오래된 위치는 모두 실패합니다.

index	value	이전 latest[value]	거리	처리
0	4	없음	-	4:0 저장
1	7	없음	-	7:1 저장
2	4	0	2 > 1	실패 후 4:2로 갱신
3	4	2	1 <= 1	True 반환

최근 인덱스의 구현

```
def nearby_duplicate(nums, k):
    latest = {}
    for index, value in enumerate(nums):
        if value in latest and index - latest[value] <= k:
            return True
        latest[value] = index
    return False
```

조건을 만족하지 않은 재등장도 최근 위치로 갱신합니다. 최초 위치만 남기면 예시의 마지막 쌍을 찾지 못합니다.

latest의 값은 해당 숫자를 가장 최근에 본 위치입니다. 멀리 떨어져 실패한 경우에도 갱신해야 다음에 더 가까운 쌍을 찾습니다. k가 커져도 값마다 최근 위치 하나만 저장하며, 오래된 키를 지우는 구현은 아닙니다.

최근 상태의 정당성과 저장 비용

현재 위치 j에서 같은 값의 가장 최근 위치 i가 거리 j-i를 최소화합니다. 최근 위치가 k 이내면 답이며, 아니면 이전 위치 어느 것도 답이 아닙니다. 갱신 후 다음 반복에서도 같은 의미가 유지됩니다.

확인 질문: 값 4가 여러 번 나오더라도 인덱스를 모두 저장할 필요가 없는 이유를 설명합니다.

각 값마다 조회·거리 비교·저장을 한 번씩 하므로 기대 O(n) 시간입니다. 서로 다른 값 수 d만큼 사전 항목이 생겨 보조 공간 O(d), d<=n입니다. k=1이어도 입력이 모두 다르면 이 코드는 n개 키를 유지하므로 공간을 O(k)라고 할 수 없습니다.

1차시 이해 확인

- `4 in events`와 `4 in latest`는 각각 무엇을 검사하나요?
- `[3,3]`에서 조회와 저장 순서가 바뀌면 어느 시점에 잘못되나요?
- `[4,7,4,4], k=1`에서 2번 위치의 갱신을 생략하면 어떤 답을 놓치나요?
- 매 반복에서 `latest.copy()`를 만들면 복잡도는 어떻게 달라지나요?

확인: 리스트에서는 원소 값, 사전에서는 키를 검사합니다. `[3,3]`을 먼저 저장하면 첫 반복부터 `[0,0]`을 만들 수 있습니다. 최근 위치 갱신을 생략하면 마지막 두 4를 놓칩니다. 매번 사전을 복사하면 최악에 $0+1+\dots+(n-1)$개 항목을 복사해 기대 시간 $O(n^2)$, 사본 하나만 유지해도 최대 추가 공간 $O(n)$입니다.

정렬된 부품 길이의 합

정렬된 길이 `[1,4,6,9]`에서 합이 7인 두 부품을 찾습니다. 왼쪽 값 1과 오른쪽 값 9의 합은 10입니다. 지금 9를 버려도 되는 이유를 설명할 수 있나요?

2차시에서는 정렬 순서가 한 번에 여러 후보를 지우는 근거임을 보입니다. 포인터 이동을 암기하기 전에 제거되는 모든 쌍을 확인

LeetCode 167의 입력 계약

Two Sum II - Input Array Is Sorted: 입력은 비내림차순이며 답 하나가 보장됩니다. 서로 다른 두 원소를 사용하고 결과 위치는 1부터 셉니다. 추가 공간은 상수여야 합니다.

```
numbers=[1,4,6,9], target=7 → [1,3]
```

이 문제는 보조 공간 O(1)을 요구합니다. 이전 값을 사전에 저장하는 방식은 최대 O(n) 공간을 사용하므로, 정렬된 입력에 두 포인터를 적용합니다.

합이 작은 왼쪽 후보의 제거

현재 후보 범위는 `[left,right]`입니다. 양 끝 합이 target보다 작다면, 현재 left와 짝지을 수 있는 모든 값은 right의 값 이하입니다. 따라서 그 합들도 모두 너무 작습니다.

left를 하나 늘리면 현재 left를 사용하는 실패 쌍만 제거합니다. right를 줄여서는 부족한 합을 키울 수 없습니다.

불변식: 아직 찾지 못한 답이 있다면 현재 범위 안에 두 위치가 모두 남아 있습니다.

합이 큰 오른쪽 후보의 제거

양 끝 합이 target보다 크다면, 현재 right와 짝지을 수 있는 모든 값은 left의 값 이상입니다. 따라서 그 합들도 모두 너무 큽니다.

현재 쌍	합	target=7의 판정	제거되는 후보
1,9	10	너무 큼	9와 1·4·6의 쌍
1,6	7	정답	더 제거하지 않음

같은 원소를 두 번 쓸 수 없으므로 반복 조건은 left<right입니다.

두 포인터의 구현

```
def two_sum_sorted(numbers, target):
    left, right = 0, len(numbers) - 1
    while left < right:
        total = numbers[left] + numbers[right]
        if total == target:
            return [left + 1, right + 1]
        if total < target:
            left += 1
        else:
            right -= 1
    return []
```

루프 안에서는 0-based 인덱스를 쓰고 반환할 때만 1을 더합니다.

작은 합에서 left를 늘리는 것은 현재 왼쪽 값으로 만들 수 있는 모든 쌍이 너무 작기 때문입니다. 큰 합에서는 오른쪽 값을 제외합니다. 이동 뒤 두 포인터 사이 후보만 남으므로 이미 탈락한 쌍을 다시 읽지 않습니다.

포인터의 종료와 전체 이동 수

매 반복에서 답을 반환하거나 포인터 하나가 안쪽으로 한 칸 움직입니다. 두 포인터의 간격이 줄어들므로 무한 반복하지 않습니다.

반례: 정렬 전제 밖 `[3,1,4], target=5`에 같은 규칙을 적용하면 1과 4의 답을 놓칩니다. 이 규칙은 원문이 보장한 정렬 조건에 의존합니다.

간격 right-left는 처음 n-1이며 반복마다 적어도 1 줄어듭니다. 따라서 비교는 최대 O(n)번입니다. 포인터 두 개와 합만 저장하여 보조 공간 O(1), 반환 위치 두 개도 O(1)입니다. 원소들을 복사하거나 정렬하는 코드는 포함되어 있지 않습니다.

LeetCode 15: 세 값과 중복 조합

3Sum: 서로 다른 세 위치의 합이 0인 값 조합을 중복 없이 반환합니다. 값이 같아도 위치가 다르면 사용할 수 있습니다.

`[-2,0,1,1,2] → [[-2,0,2],[-2,1,1]]`

세 위치를 모두 열거하면 $O(n^3)$입니다. 값을 하나 고정하면 나머지 두 값의 목표 합은 고정 값의 음수가 됩니다.

정렬과 고정 값의 역할

초기 설정: 입력의 복사 정렬, 첫 원소의 위치 index 고정. left=index+1, right=n-1로 두면 세 위치가 항상 다릅니다.

현재 세 값의 합이 음수면 left를 늘리고, 양수면 right를 줄입니다. 167과 같은 정렬 기반 제거

3Sum은 값 조합을 반환하므로 정렬한 배열에서 탐색합니다. 원래 인덱스를 반환하는 문제에서는 값과 원래 인덱스를 함께 저장해야 합니다.

3Sum에서 고정 값 -2의 추적

index,left,right	세 값	합	동작
0,1,4	-2,0,2	0	[-2,0,2] 저장; 양쪽 이동
0,2,3	-2,1,1	0	[-2,1,1] 저장; 양쪽 이동
0,3,2	범위 끝	-	고정 값 -2의 탐색 종료

다음 고정 값: 0. 값 1·1·2에서 두 값을 골라도 0의 음수인 0을 만들지 못합니다. 현재 고정 값에서 찾지 못한 조합과 다음 고정 값의 조합을 구분

3Sum 구현: 고정과 후보 이동

```
def three_sum(nums):
    values = sorted(nums)
    result = []
    for index in range(len(values) - 2):
        if index and values[index] == values[index - 1]:
            continue
        left, right = index + 1, len(values) - 1
        while left < right:
            total = values[index] + values[left] + values[right]
            if total < 0:
                left += 1
```

고정 값 하나마다 뒤쪽에서 두 포인터를 새로 시작합니다. 같은 고정 값을 건너뛰고, 정답을 저장한 뒤 양쪽의 같은 값을 건너뛰어 값 조합의 중복을 제거합니다. 서로 다른 위치의 같은 값 두 개를 사용하는 것은 허용됩니다.

3Sum 구현: 저장과 중복 건너뛰기

```
            elif total > 0:
                right -= 1
            else:
                result.append([values[index], values[left], values[right]])
                left += 1
                right -= 1
                while left < right and values[left] == values[left - 1]:
                    left += 1
                while left < right and values[right] == values[right + 1]:
                    right -= 1
    return result
```

정답을 저장한 뒤 양쪽 포인터를 이동하고, 방금 사용한 값과 같은 값을 건너뜁니다.

`[-2,0,0,2,2]`에서 첫 답을 찾은 위치는 (0,1,4)입니다. 양쪽 이동 뒤 (left,right)=(2,3)인데 `values[2]==values[1]==0`이므로 left=3이 되어 종료합니다. 추가 0과 2로 만들 조합도 이미 저장한 `[-2,0,2]`여서 새 답을 잃지 않습니다.

중복 제거의 두 위치

첫째, 이미 고정한 값과 같은 값을 다시 고정하지 않습니다. 둘째, 답을 찾은 뒤 같은 left 값과 right 값이 반복되는 것을 건너뜁니다. 같은 값도 서로 다른 위치에 있으면 함께 사용할 수 있습니다. 중복 제거의 대상은 이미 반환한 값 조합입니다.

`[0,0,0,0]`에서는 위치 조합이 여러 개여도 값 조합 `[0,0,0]` 하나만 반환합니다. `[-2,0,1,1,2]`의 `[-2,1,1]`은 서로 다른 두 1의 위치를 사용하므로 유효합니다.

3Sum의 정당성과 비용

정렬된 조합마다 가장 작은 값이 하나 있습니다. 그 값을 고정한 차례의 두 포인터 탐색이 나머지 쌍을 포함하고, 합의 방향으로 실패 후보만 제거합니다. 중복 건너뛰기는 이미 기록한 같은 값 조합만 없앱니다.

고정 인덱스가 최대 n개이고 각 인덱스에서 두 포인터의 총 이동이 O(n)이므로 $O(n^2)$입니다. 중복을 넘기는 while도 같은 방향 이동의 일부라 별도 n을 다시 곱하지 않습니다. 정렬 O(n log n)은 이보다 작습니다. 복사·정렬 작업 공간 O(n)과 정답 z개의 출력 공간 O(z)를 구분합니다.

문제 조건에 따른 상태 선택

질문	남길 상태	확인할 조건
정렬 안 된 두 값의 합	이전 값→위치	같은 위치 재사용 금지
가까운 같은 값	값→최근 위치	거리 k; 실패 후에도 갱신
정렬된 두 값의 합	left,right	정렬·상수 공간·1-based 반환
중복 없는 세 값의 합	고정 위치와 두 포인터	서로 다른 위치·값 조합 중복 제거

자료구조 이름을 먼저 고르기보다, 다음 판단에 필요한 정보와 버려도 되는 후보를 먼저 적습니다.

독립 실습과 제출물

학생 실습에서 LeetCode 1, 219, 167, 15를 풉니다. `student_week_01.py`에 코드를 작성하고 작은 입력을 먼저 손으로 추적합니다.

각 문제에 상태의 의미, 정당성, 시간·공간 분석과 경계 입력을 함께 제출합니다. 막힐 때는 단계별 힌트·해설을 필요한 단계까지만 확인합니다.

Python 실습 묶음에서 `python week01_walkthrough.py`로 강의의 예시와 추적 결과를 확인

핵심 참고자료

Erik Demaine·Srini Devadas, Introduction to Algorithms, MIT 6.006 (2011년 가을 학기).

문제의 전체 입력 조건은 각 LeetCode 공식 문제 링크에서 확인합니다. 다음 주에는 연속 구간의 상태를 갱신하는 슬라이딩 윈도우와 가장 최근 미해결 정보를 보관하는 스택으로 이어집니다.