1주차 풀이 읽기자료

연산 비용과 풀이

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.

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.

1 · 단계별 힌트

서로 다른 두 인덱스의 값 합이 target인 인덱스 두 개를 반환합니다. 같은 원소를 재사용하지 않으며, 공식 입력에는 답 하나가 있습니다.

LeetCode 1: Two Sum · Easy

H1. 현재 값과 짝이 되는 값은 무엇인가요?

H2. 같은 위치를 두 번 사용하지 않으려면 언제 기록하나요?

H3. 이전 값→인덱스를 저장하고 target-value를 조회한 뒤 현재 값을 기록합니다.

`nums=[4,8,3], target=7`

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]

1 · 정당성과 비용

반복 시작 때 사전에는 앞서 본 위치만 있습니다. 답의 두 번째 위치에 도착하면 첫 위치가 기록되어 있으므로 답을 찾습니다. 모든 쌍을 검사하는 $O(n^2)$ 방법에서 이전 정보를 재사용한 것입니다.

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.

1 · 반례와 검증

[3,3], target=6에서 현재 값을 먼저 저장하면 같은 인덱스를 두 번 반환할 수 있습니다.

```
nums=[4,8,3], target=7 → [0,2]
```

1 · 구현 1/1

```
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.

219 · 단계별 힌트

서로 다른 위치 i,j에서 값이 같고 거리 $|i - j|$가 k 이하이면 True, 아니면 False입니다.

LeetCode 219: Contains Duplicate II · Easy

H1. 같은 값의 어떤 위치를 남길까요?

H2. 가장 가까운 과거 위치보다 오래된 위치가 더 유리할까요?

H3. 값마다 최근 인덱스를 남기고 거리를 비교한 뒤 갱신합니다.

`nums=[4,7,4,4], k=1`

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

219 · 정당성과 비용

최근 위치는 모든 이전 위치 중 현재 위치와 가장 가깝습니다. 이 거리도 k보다 크면 다른 과거 위치도 실패합니다. 갱신으로 다음 반복에도 같은 성질이 유지됩니다.

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.

219 · 반례와 검증

최초 위치만 저장하면 [4,7,4,4], k=1에서 마지막 두 위치를 놓칩니다. k=0에서는 서로 다른 위치의 답이 없습니다.

```
nums=[4,7,4,4], k=1 → True
```

219 · 구현 1/1

```
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
```

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.

167 · 단계별 힌트

비내림차순 배열에서 합이 target인 서로 다른 두 위치를 찾습니다. 답은 1-based 인덱스이며 보조 공간 O(1)이 요구됩니다.

LeetCode 167: Two Sum II - Input Array Is Sorted · Medium

H1. 양 끝 합이 작으면 어느 후보들이 모두 작나요?

H2. 정렬 조건을 지우면 이 제거가 가능한가요?

H3. left<right인 동안 합이 작으면 left, 크면 right를 안쪽으로 이동합니다.

`nums=[1,4,6,9], target=7`

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

167 · 정당성과 비용

합이 작으면 현재 left와 right 이하의 어떤 값도 답을 만들지 못합니다. 합이 크면 현재 right와 left 이상의 어떤 값도 실패합니다. 제거된 쌍에는 답이 없으므로 남은 범위가 답을 보존합니다.

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.

167 · 반례와 검증

정렬 전제 밖 [3,1,4], target=5에는 같은 이동 규칙을 적용할 수 없습니다. left==right는 같은 원소 재사용입니다.

```
numbers=[1,4,6,9], target=10 → [1,4]
```

167 · 구현 1/1

```
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 []
```

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.

15 · 단계별 힌트

합이 0인 서로 다른 세 위치의 값 조합을 모두 반환합니다. 결과는 인덱스가 아닌 값이며, 같은 값 조합을 중복 반환하지 않습니다.

LeetCode 15: 3Sum · Medium

H1. 한 값을 고정하면 남는 문제는 무엇인가요?

H2. 같은 값을 다시 고정하면 새 조합이 생기나요?

H3. 복사 정렬 후 첫 값을 고정하고 뒤쪽에서 두 포인터를 사용합니다. 고정 값과 발견한 쌍의 중복을 건너뜁니다.

`nums=[-2,0,1,1,2]`

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

15 · 정당성과 비용

각 정렬된 세 값은 가장 작은 값의 차례에서 검사됩니다. 두 포인터 제거는 정렬로 정당화됩니다. 같은 값을 건너뛰는 것은 위치가 다른 동일한 값 조합만 제거합니다.

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.

15 · 반례와 검증

[0,0,0,0]은 한 조합만 반환합니다. 값이 같다는 이유로 서로 다른 위치를 금지하면 이 답을 잃습니다.

```
nums=[-2,0,1,1,2] → [[-2,0,2],[-2,1,1]]
```

15 · 구현 1/2

```
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.

15 · 구현 2/2

```
            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
```

합이 0이면 세 값을 저장한 뒤 양 끝을 한 칸씩 옮깁니다. 이후 같은 값을 넘기는 while은 이미 저장한 값 조합을 반복하지 않기 위한 것입니다. 포인터는 계속 안쪽으로만 이동하므로 이 중복 제거도 같은 선형 탐색 비용에 포함됩니다.

`[-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.

실행과 참고자료

```
cd code
python demo_01.py
python demo_02.py
python verify.py
```

완성 코드는 `solutions.py`, 독립 실습은 `student_week_01.py`에 있습니다. LeetCode에서는 공식 메서드 서명에 맞춰 함수를 연결합니다.

Erik Demaine and Srini Devadas, Introduction to Algorithms, MIT 6.006 (Fall 2011).