# 3주차: 이진 탐색과 연결 리스트

## 정렬된 측정 기록과 연결된 대기열

장치가 측정값을 정렬된 행에 저장합니다. 새 측정값을 찾을 때 모든 기록을 확인하지 않을 수 있을까요? 이어서 대기열의 노드 객체를 유지한 채 순서를 바꿉니다.

이번 주에는 후보 구간으로 값을 탐색하고 노드의 연결을 바꿔 리스트를 뒤집거나 병합합니다.

### 학습 목표

5차시: 행렬·회전 배열 탐색. 6차시: 리스트 뒤집기·정렬 리스트 병합.

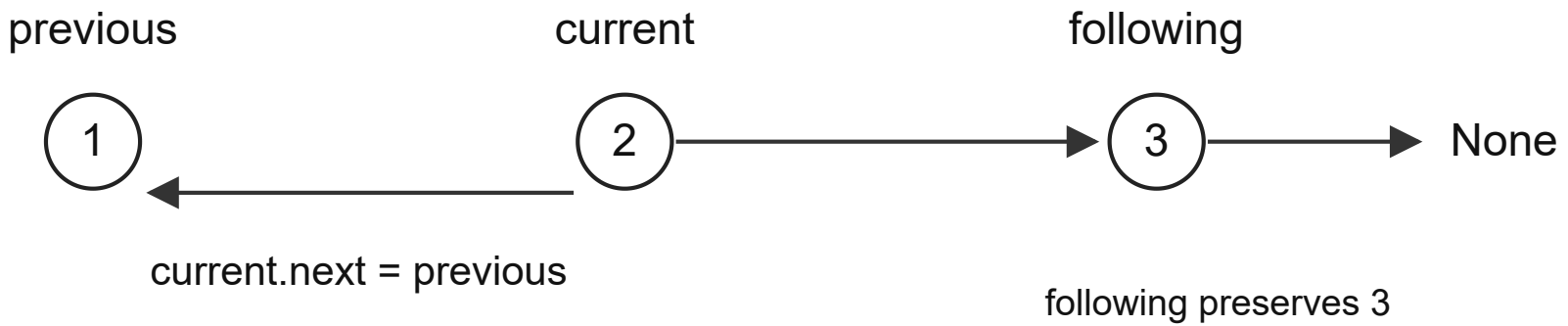
- 후보 구간과 노드 참조 정의
- 제거한 후보와 보존한 노드 설명
- 갱신 추적·종료 근거·비용 분석

실습: LeetCode 74, 33, 206, 21.

# 위치와 노드 참조

배열의 i번째 원소는 인덱스로 직접 읽습니다. 연결 리스트는 노드가 다음 노드를 가리키므로 머리에서 next를 따라 이동합니다. 같은 값 2를 가진 노드 두 개도 서로 다른 객체일 수 있습니다.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val, self.next = val, next
```




그림은 2를 처리할 때입니다. following으로 3을 보존한 뒤 2의 next를 1로 바꿉니다. 노드의 값과 노드 사이 연결은 서로 다른 데이터입니다. `node = node.next`는 지역 참조만 옮기고 `node.next = other`는 연결을 바꿉니다.

# 74 · 풀이의 출발점과 실행 추적

LeetCode 74: Search a 2D Matrix · Medium

행렬에서 target의 존재 여부를 반환합니다.

각 행이 정렬되어 있고 다음 행의 첫 값이 앞 행의 마지막 값보다 큽니다. 따라서 행을 이어 붙였다고 생각하면 하나의 정렬된 배열입니다. 실제로 복사할 필요 없이 가상의 위치를 행·열 번호로 바꿉니다.

`matrix=[[1,3,5],[7,9,11]], target=9`. 비교 직전의 범위와 갱신 결과입니다.

| left,right | mid | 행,열 | 값 | 다음 동작 |
|---|---|---|---|---|
| 0,5 | 2 | 0,2 | 5 | 5 < 9 → left=3 |
| 3,5 | 4 | 1,1 | 9 | True 반환 |

# 74 · 아이디어를 코드로 옮기기

```
def search_matrix(matrix, target):
    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1
    while left <= right:
        mid = (left + right) // 2
        value = matrix[mid // cols][mid % cols]
        if value == target:
            return True
        if value < target:
            left = mid + 1
        else:
            right = mid - 1
    return False
```

`mid // cols`는 앞에서 완전히 지난 행 수, `mid % cols`는 현재 행 안의 위치입니다. 예를 들어 4를 3으로 나누면 행 1, 열 1입니다.

값이 작으면 mid까지, 크면 mid부터 제거합니다. 이미 비교한 mid를 다시 남기지 않으므로 원소 하나가 남아도 탐색이 끝납니다.

# 74 · 정당성·시간·메모리의 근거

행 사이 조건까지 있으므로 평탄 배열은 정렬되어 있습니다. mid가 target보다 작으면 mid 이하를 제거합니다. 매 반복 구간이 줄고, 종료하면 후보가 없습니다.

원소 수를 N=R·C라 하면 비교 후 후보가 대략 N/2, N/4, …로 줄어듭니다. N=8이면 최대 4번 비교하며, 일반적으로 시간은 O(log(N+1))입니다. 각 비교의 행·열 계산과 리스트 접근은 O(1)입니다.

행렬을 평탄화한 새 리스트를 만들지 않습니다. left, right, mid 등 고정 개수의 값만 추가하므로 보조 공간은 O(1)입니다. `sum(matrix, [])` 같은 복사는 이 공간 분석에 포함되지 않는 다른 구현입니다.

# 없는 값을 찾을 때의 종료

같은 행렬 `[[1,3,5],[7,9,11]]`에서 target=8을 찾습니다. `[left,right]`는 양 끝을 포함한 아직 남은 위치입니다.

| 비교 전 범위 | mid와 값 | 제거 근거 | 다음 범위 |
|---|---|---|---|
| [0,5] | 2, 값 5 | 2 이하 값은 모두 8보다 작음 | [3,5] |
| [3,5] | 4, 값 9 | 4 이상 값은 모두 8보다 큼 | [3,3] |
| [3,3] | 3, 값 7 | 마지막 후보도 불일치 | [4,3] |

left>right이면 후보가 0개라 False입니다. left==right는 후보가 한 개 남은 상태이므로 검사해야 합니다. 첫 행의 갱신을 left=mid로 쓰면 `[3,4]`에서 mid=3을 반복할 수 있습니다.

범위 크기를 반씩 줄여 1개가 되려면 약 $\log_2 N$번이 필요하고, 마지막 한 개의 검사까지 더해 최악 비교 횟수는 $\text{floor}(\log_2 N)+1$입니다($N \geq 1$).

# 33 · 풀이의 출발점과 실행 추적

LeetCode 33: Search in Rotated Sorted Array · Medium

회전된 정렬 배열에서 target의 위치를 반환하고 없으면 -1입니다.

회전된 배열 전체는 정렬되어 있지 않아도, 서로 다른 값이라는 조건에서는 mid를 기준으로 적어도 한쪽 절반이 정렬되어 있습니다. 먼저 정렬된 쪽을 찾고, target이 그 값 범위에 들어가는지 확인하면 버릴 절반을 결정할 수 있습니다.

`nums=[4,5,6,7,0,1,2], target=0`

| left,mid,right | 정렬된 쪽 | target 검사 | 갱신 |
|---|---|---|---|
| 0,3,6 | 왼쪽: 4..7 | 0은 범위 밖 | left=4 |
| 4,5,6 | 왼쪽: 0..1 | 0은 범위 안 | right=4 |
| 4,4,4 | nums[4]=0 | 일치 | 4 반환 |

# 33 · 아이디어를 코드로 옮기기

```
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = (left + right) // 2
        if nums[mid] == target:
            return mid
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

`nums[left] <= nums[mid]`이면 왼쪽이 정렬되어 있습니다. `nums[left] <= target < nums[mid]`일 때만 그쪽을 남깁니다. mid는 앞의 일치 검사에서 처리했으므로 target 범위의 mid 쪽 끝은 엄격한 부등식입니다.

오른쪽이 정렬된 경우도 같은 논리입니다. 위치를 반환하므로 정렬을 새로 하면 원래 위치 정보와 회전 구조가 바뀝니다.

# 33 · 정당성·시간·메모리의 근거

서로 다른 값의 한 번 회전 배열에서는 적어도 한 절반이 정렬되어 있습니다. 그 범위 안/밖 검사는 target을 포함할 수 없는 절반만 제거하며 mid±1로 진전합니다.

각 반복에서 mid를 제외한 한쪽 절반만 남깁니다. 비교 횟수는 입력을 몇 번 반으로 줄일 수 있는지에 따라 O(log(n+1))이고, 반복 안의 고정 개수 비교·접근은 O(1)입니다.

새 배열이나 재귀 호출을 만들지 않아 보조 공간은 O(1)입니다. 중복을 허용하면 `[1,1,1,0,1]`처럼 양 끝과 mid가 같아 정렬된 쪽을 판별할 수 없는 경우가 생깁니다. 이 복잡도와 코드는 서로 다른 값 조건에 의존합니다.

# 206 · 풀이의 출발점과 실행 추적

LeetCode 206: Reverse Linked List · Easy

단일 연결 리스트를 뒤집고 새 head를 반환합니다.

리스트의 순서는 값이 저장된 위치가 아니라 next 연결로 정해집니다. 새 노드를 만들지 않고 현재 노드의 next를 앞쪽으로 돌립니다. 다만 연결을 바꾸기 전에 원래 다음 노드를 저장해야 아직 처리하지 않은 부분을 잃지 않습니다.

입력 `1 → 2 → 3 → None`. 각 행은 while 본문을 한 번 끝낸 상태입니다.

| 처리한 노드 | previous가 가리키는 부분 | current가 가리키는 부분 |
|---|---|---|
| 없음 | None | 1 → 2 → 3 |
| 1 | 1 → None | 2 → 3 |
| 2 | 2 → 1 → None | 3 |
| 3 | 3 → 2 → 1 → None | None |

# 206 · 아이디어를 코드로 옮기기

```
def reverse_list(head):
    previous, current = None, head
    while current:
        following = current.next
        current.next = previous
        previous, current = current, following
    return previous
```

`following = current.next`: 원래 다음 노드를 보존합니다. `current.next = previous`: 현재 연결 하나를 뒤집습니다. 마지막 동시 대입은 previous를 방금 처리한 노드로, current를 보존한 다음 노드로 옮깁니다.

current가 None이면 미처리 노드가 없습니다. 새 머리는 previous입니다. 빈 입력에서는 반복을 한 번도 실행하지 않고 None을 반환합니다.

2를 처리하는 순간 previous는 1, current는 2, following은 3을 가리킵니다. next 변경 뒤 동시 대입의 오른쪽 `(current,following)`은 먼저 `(2,3)`으로 평가됩니다. 그러므로 previous는 2, current는 3이 됩니다. `current=current.next`로 전진하면 이미 뒤집힌 연결을 따라 1로 되돌아가므로 저장한 following을 사용합니다.

# 206 · 정당성·시간·메모리의 근거

반복 시작 때 previous는 처리한 접두부의 역순, current는 미처리 접미부입니다. 한 노드를 옮겨도 둘의 합은 원래 노드 전체입니다. current가 None이면 previous가 완성된 역순입니다.

노드 n개를 각각 한 번 방문하고, 방문마다 고정 개수의 참조 대입을 하므로 시간은 O(n)입니다. 첫 노드를 바꾸었다고 뒤 노드의 연결까지 자동으로 바뀌지는 않아 모든 노드를 처리해야 합니다.

previous, current, following의 세 참조만 추가합니다. 입력 노드를 재사용하므로 보조 공간은 O(1), 새 출력 노드 수는 0입니다. 노드의 개수 O(n)과 추가로 사용한 공간 O(1)을 구분합니다. 재귀 구현은 호출 스택 O(n)이 필요합니다.

# 21 · 풀이의 출발점과 실행 추적

LeetCode 21: Merge Two Sorted Lists · Easy

정렬된 두 연결 리스트의 노드를 이어 하나의 정렬 리스트로 반환합니다.

두 정렬된 리스트에서 가장 작은 남은 값은 두 머리 중 하나입니다. 작은 머리를 결과의 꼬리에 연결한 뒤 그 리스트만 한 칸 전진합니다. 뒤쪽 노드를 매번 다시 검색할 필요가 없습니다.

입력 A=1 → 4, B=2 → 3. tail 뒤에 아직 붙어 있는 입력 연결은 다음 대입에서 바뀔 수 있습니다.

| 선택 | 확정된 결과 접두부 | A 머리 | B 머리 |
|---|---|---|---|
| 시작 | 없음 | 1 | 2 |
| A의 1 | 1 | 4 | 2 |
| B의 2 | 1 → 2 | 4 | 3 |
| B의 3 | 1 → 2 → 3 | 4 | None |
| 남은 A 연결 | 1 → 2 → 3 → 4 | 4 | None |

# 21 · 아이디어를 코드로 옮기기

```
from structures import ListNode

def merge_lists(first, second):
    dummy = tail = ListNode()
    while first and second:
        if first.val <= second.val:
            tail.next, first = first, first.next
        else:
            tail.next, second = second, second.next
        tail = tail.next
    tail.next = first or second
    return dummy.next
```

dummy는 첫 삽입도 나머지 삽입과 같게 처리하는 임시 머리입니다. `tail.next, first = first, first.next`는 갱신 전 first를 연결하고 원래 다음 노드로 first를 옮깁니다.

한쪽이 비면 다른 쪽은 이미 정렬되어 있으므로 `tail.next = first or second`로 나머지 사슬 전체를 연결합니다. 반환할 때 dummy는 제외합니다.

# 21 · 정당성·시간·메모리의 근거

tail까지는 정렬된 병합 접두부입니다. 두 head 중 작은 것은 남은 모든 값 중 최소이므로 다음 노드로 안전합니다. 남은 접미부도 정렬되어 있습니다.

길이가 n, m이면 반복 한 번마다 한 노드가 확정됩니다. 두 리스트가 번갈아 선택되는 경우 O(n+m)번의 작업이 필요합니다. 남은 사슬을 연결하는 마지막 대입은 O(1)입니다.

dummy 한 개와 고정 개수의 참조만 추가하므로 보조 공간은 O(1)입니다. 결과에 n+m개 노드가 있어도 기존 입력 노드를 다시 연결한 것이므로 새 노드 n+m개를 저장하지 않습니다. 입력 연결을 보존해야 한다면 복사 구현과 O(n+m) 공간이 필요합니다.

# 실습과 참고자료

LeetCode · 74, 33, 206, 21 · 풀이 · Python ZIP

```
python demo_05.py
python demo_06.py
python verify.py
```

학생 파일을 구현한 뒤 `python verify.py --student`로 검사합니다.

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