6주차 풀이 읽기자료

풀이와 실행 과정

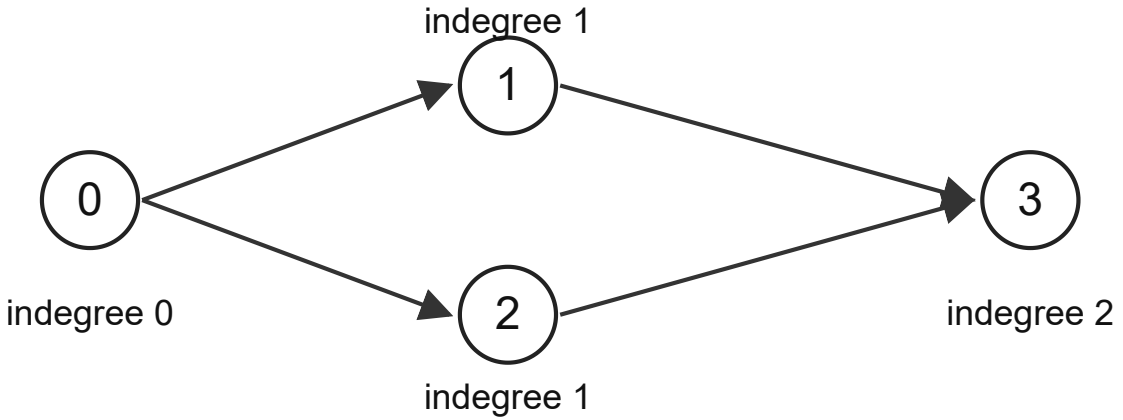
207	210	78	39

분석 파이프라인의 작업이 서로를 기다리면 실행이 멈춥니다. 실행 가능한 순서를 찾은 뒤에는 주어진 항목으로 만들 수 있는 선택을 빠짐없이 생성해야 합니다.

남은 선행 조건으로 실행 순서를 정하고, 선택·재귀·복원으로 분기 간 상태를 분리합니다.

의존관계의 방향과 인접 리스트

`[after,before]`는 before가 끝나야 after를 시작할 수 있다는 뜻입니다. before → after 방향으로 간선을 저장합니다. 진입차수 indegree는 아직 끝나지 않은 선행 작업의 수입니다.



저장 항목	이 그림의 값	의미
graph	[[1,2],[3],[3],[]]	각 작업 완료 시 영향을 받는 작업
indegree	[0,1,1,2]	남은 선행 작업 수
초기 queue	[0]	지금 바로 할 수 있는 작업

graph는 간선 E개를 저장합니다. 정점마다 전체 V개를 넣는 행렬과 달리 인접 리스트의 전체 항목 수는 V+E에 비례합니다.

207 · 풀이의 출발점과 실행 추적

LeetCode 207: Course Schedule · Medium

선수 과목 조건을 모두 만족하며 전체 과목을 끝낼 수 있는지 반환합니다.

어떤 과목도 시작할 수 없다면 서로 기다리는 고리가 있는지 의심해야 합니다. 선행 과목 수가 0인 과목을 반복해서 제거한 뒤 모든 과목을 제거했는지가 판정 기준입니다. 남은 과목이 있다면 그 안에 유향 사이클이 있습니다.

`n=3, prerequisites=[[1,0],[2,1],[1,2]]`: 0 → 1, 1 → 2, 2 → 1

단계	indegree	queue	처리한 과목
시작	[0,2,1]	[0]	[]
0 완료	[0,1,1]	[]	[0]
종료	1과 2가 서로 대기	[]	1개 < 3개

`course_order`는 미완료 순서 대신 []를 반환하고, `can_finish`는 False입니다.

207 · 아이디어를 코드로 옮기기

```
def can_finish(n, prerequisites):
    return len(course_order(n, prerequisites)) == n
```

`can_finish`의 한 줄은 뒤의 `course_order` 전체 탐색을 실행합니다. 함수 호출을 한 줄로 썼다고 계산이 한 번으로 끝나는 것은 아닙니다.

사이클이 있으면 그 위의 모든 과목이 적어도 하나의 미완료 선행 과목을 유지합니다. 사이클이 없는 유한 유향 그래프에는 진입차수 0인 정점이 하나 이상 있으므로 계속 제거할 수 있습니다.

호출한 `course_order`가 모든 정점 V개와 간선 E개를 최대 한 번씩 처리하므로 O(V+E) 시간입니다. 반환된 리스트의 len 조회와 n 비교만 O(1)이며, 그 전에 수행한 탐색 비용을 빼면 안 됩니다.

인접 리스트 O(V+E), 진입차수·큐·순서 O(V)를 생성합니다. 반환값 False나 True는 O(1)이어도 실행 중 최대 보조 공간은 O(V+E)입니다. 공간은 최종 출력 크기만으로 정하지 않습니다.

210 · 풀이의 출발점과 실행 추적

LeetCode 210: Course Schedule II · Medium

선수 조건을 만족하는 순서 하나를 반환합니다. 불가능하면 빈 목록입니다.

의존관계를 매번 전부 확인하는 대신 아직 끝나지 않은 선행 작업 수를 각 작업에 저장합니다. 이 수가 0이면 지금 실행할 수 있습니다. 하나를 끝낼 때 그 작업을 기다리던 이웃들의 수만 1씩 줄입니다.

`n=4, prerequisites=[[1,0],[2,0],[3,1],[3,2]]`. 간선은 before → after입니다.

완료한 node	indegree (0,1,2,3)	queue	order
시작	[0,1,1,2]	[0]	[]
0	[0,0,0,2]	[1,2]	[0]
1	[0,0,0,1]	[2]	[0,1]
2	[0,0,0,0]	[3]	[0,1,2]
3	[0,0,0,0]	[]	[0,1,2,3]

210 · 아이디어를 코드로 옮기기

```
from collections import deque

def course_order(n, prerequisites):
    graph, indegree = [[] for _ in range(n)], [0] * n
    for after, before in prerequisites:
        graph[before].append(after)
        indegree[after] += 1
    queue = deque(v for v in range(n) if indegree[v] == 0)
    order = []
    while queue:
        node = queue.popleft()
        order.append(node)
        for neighbor in graph[node]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
    return order if len(order) == n else []
```

`[after,before]`를 읽어 `graph[before]`에 after를 추가합니다. node를 꺼내면 그 이웃의 indegree를 줄이고, 정확히 0이 되는 순간에만 넣습니다.

207번은 이 순서에 모든 작업이 포함되는지만 반환합니다. 210번은 순서 자체를 반환합니다. 큐에서 가능한 작업 중 어느 것을 먼저 택해도 선행 조건을 지킨 순서가 됩니다.

210 · 정당성·시간·메모리의 근거

각 과목은 모든 선수가 제거된 뒤에만 큐에 들어갑니다. 따라서 반환 순서에서 모든 before의 위치가 after보다 앞섭니다. 수가 부족하면 전체 순서가 아닙니다.

정점 V개와 간선 E개를 저장하는 초기 작업이 O(V+E)입니다. 각 정점은 한 번 큐에 들어가고 각 간선은 출발 정점 처리 때 한 번만 읽습니다. 이웃 수가 정점마다 달라도 그 합은 E이므로 전체 시간은 O(V+E)입니다.

인접 리스트의 빈 목록 V개와 간선 항목 E개, indegree·queue·order의 최대 V개 항목이 필요합니다. 총 추가 공간은 O(V+E), 그중 반환 순서는 O(V)입니다. 207번의 반환값은 bool이지만 이 구현은 같은 순서 함수를 호출하므로 공간도 O(V+E)입니다.

[[1,0]]을 1→0으로 읽으면 역순을 만듭니다. 정답 하나와 목록을 직접 비교하지 말고 모든 간선을 검증합니다.

```
n=4, prerequisites=[[2,0],[2,1],[3,2]] → [0,1,2,3]
```

`[[] for _ in range(n)]`는 정점마다 새 이웃 목록을 만듭니다. `[[]]*n`은 같은 목록을 공유하므로 `graph[0].append(1)`이 모든 정점의 이웃에 1을 넣는 오류가 됩니다. 진입차수가 0이 되는 순간은 한 번뿐이라 정상 입력에서는 한 정점을 큐에 중복 삽입하지 않습니다.

78 · 풀이의 출발점과 실행 추적

LeetCode 78: Subsets · Medium

서로 다른 값으로 이루어진 배열의 모든 부분집합을 반환합니다.

부분집합을 만들 때는 다음에 고를 위치를 현재 위치보다 뒤로 제한합니다. 그러면 [2,5]와 [5,2]를 따로 만들지 않습니다. 현재 경로 자체도 하나의 부분집합이므로 빈 경로부터 결과에 넣습니다.

`nums=[2,5]`. 결과 저장은 visit에 들어간 직후입니다.

실행	path	저장되는 복사본
visit(0)	[]	[]
append(2), visit(1)	[2]	[2]
append(5), visit(2)	[2,5]	[2,5]
pop(), pop()	[]	없음
append(5), visit(2)	[5]	[5]

결과 순서는 `[[],[2],[2,5],[5]]`입니다.

78 · 아이디어를 코드로 옮기기

```
def subsets(nums):
    result, path = [], []

    def visit(start):
        result.append(path.copy())
        for index in range(start, len(nums)):
            path.append(nums[index])
            visit(index + 1)
            path.pop()

    visit(0)
    return result
```

append로 한 선택을 한 뒤 `visit(index+1)`로 다음 위치부터 탐색합니다. 재귀가 돌아오면 pop으로 그 선택만 취소해야 같은 부모에서 다른 선택을 시험할 수 있습니다.

`path.copy()`는 현재 내용을 따로 저장합니다. path 자체를 넣으면 모든 결과가 같은 리스트를 가리켜 나중의 pop에 함께 바뀝니다. 입력 원소는 서로 다릅니다.

78 · 정당성·시간·메모리의 근거

부분집합마다 오름차순 인덱스 열이 하나 있습니다. start가 그 열만 허용하므로 모든 부분집합을 정확히 한 번 만듭니다. 복원 후 부모의 path는 호출 전과 같습니다.

원소마다 포함·제외 두 가능성이 있어 부분집합은 2^n개입니다. n>=1일 때 각 원소는 정확히 절반의 부분집합에 들어가므로 복사하는 원소 수의 합은 $n \cdot 2^{n-1}$입니다. 결과를 실제 리스트로 만드는 시간과 출력 공간은 $O(n \cdot 2^n)$입니다.

현재 path 길이와 재귀 깊이는 각각 최대 n이므로 출력을 제외한 보조 공간은 O(n)입니다. 시간은 전체 탐색량, 공간은 동시에 살아 있는 경로와 호출을 셉니다. 호출 총수를 전부 동시에 저장한다고 계산하지 않습니다.

result.append(path)는 같은 객체를 보관합니다. 마지막 pop 뒤 여러 결과가 빈 목록으로 바뀔 수 있습니다.

```
nums=[2,5] → [[],[2],[2,5],[5]]
```

39 · 풀이의 출발점과 실행 추적

LeetCode 39: Combination Sum · Medium

후보를 반복 사용해 target 합을 만드는 중복 없는 조합을 반환합니다.

양수 후보를 골라 남은 합을 줄입니다. 같은 후보를 여러 번 사용할 수 있으므로 고른 위치부터 다시 탐색합니다. 대신 앞 위치로 돌아가지 않아 [2,3]과 [3,2] 같은 순열 중복을 만들지 않습니다.

`candidates=[2,3], target=5`. remaining은 아직 채워야 할 합입니다.

path	remaining	다음 동작
[]	5	2 선택
[2]	3	2 재사용
[2,2]	1	2 > 1 → 복귀
[2,3]	0	복사 저장 후 복귀
[3]	2	3 > 2 → 종료

결과는 `[[2,3]]`입니다.

재귀 복귀와 부모의 다음 선택

후보 [2,3], 목표 5에서 각 호출의 start와 remaining은 독립된 지역 변수이고 path만 공유합니다. 자식이 return해도 부모의 for는 그 다음 줄부터 계속됩니다.

실행 위치	공유 path	해당 호출의 remaining
부모 visit(0,3), 2 추가	[2,2]	3
자식 visit(0,1), 2>1로 종료	[2,2]	1
부모로 복귀, path.pop()	[2]	여전히 3
부모의 다음 index=1, 3 추가	[2,3]	3
자식 visit(1,0), 복사 저장 후 return	[2,3]	0
부모로 복귀, path.pop()	[2]	3

첫 행의 부모는 이미 첫 2를 고른 호출입니다. pop은 공유 경로의 마지막 선택을 취소합니다. remaining은 부모 값이 바뀐 적이 없어 별도로 더해 복원하지 않습니다. 반대로 전역 remaining을 직접 줄이는 구현이라면 그 값도 복원해야 합니다.

39 · 아이디어를 코드로 옮기기

```
def combinations(candidates, target):
    values, result, path = sorted(candidates), [], []

    def visit(start, remaining):
        if remaining == 0:
            result.append(path.copy())
            return
        for index in range(start, len(values)):
            value = values[index]
            if value > remaining:
                break
            path.append(value)
            visit(index, remaining - value)
            path.pop()

    visit(0, target)
    return result
```

정렬했기 때문에 `value > remaining`이면 뒤 후보도 전부 너무 큽니다. 그래서 continue가 아니라 break할 수 있습니다. remaining이 0일 때만 복사본을 저장합니다.

`visit(index, ...)`는 재사용, 부분집합의 `visit(index+1)`은 해당 원소 재사용 금지입니다. 양수 조건이 있어야 매 호출에서 remaining이 줄고 탐색이 종료합니다.

39 · 정당성·시간·메모리의 근거

양수 선택마다 remaining이 감소해 종료합니다. 비내림차순 인덱스만 선택하므로 순서만 다른 중복을 피하고 모든 가능한 조합을 표현합니다.

후보 수 m, 목표 T, 최소 후보 c라 하면 경로 길이는 L=floor(T/c) 이하입니다. 방문 호출 수 B, 출력 원소 수의 합 S를 쓰면 정렬 O(m log m), 호출당 최대 m개 검사, 복사 O(S)이므로 O(m log m+mB+S) 시간입니다. B는 입력에 따라 크게 달라져 이를 단순 O(n)이라 할 수 없습니다.

보수적으로 B=O($(m+1)^{L+1}$)입니다. 정렬한 후보 m개와 길이 L의 path·호출 스택이 있어 보조 공간 O(m+L), 출력은 O(S)입니다. 전체 탐색 트리를 한 번에 저장하지는 않습니다.

index+1로 재귀하면 [2], target=4의 [2,2]를 놓칩니다. 후보 0은 공식 범위 밖이며 종료 논리를 깨뜨립니다.

```
candidates=[2,5], target=7 → [[2,5]]
```

예를 들어 m=2,T=5,c=2이면 깊이는 L=2 이하입니다. 모든 호출이 최대 m개 자식을 가진다는 느슨한 계산으로 호출 수는 1+m+m^2=7 이하입니다. 실제로는 순서 제한과 합 초과 가지치기로 더 적으며, 저장된 답을 복사하는 S의 비용은 별도로 더합니다.

중간 점검의 답과 근거

1. 없습니다. 두 과목이 서로를 기다리므로 빈 목록을 반환합니다.
2. 정답입니다. 0과 1이 모두 2보다 먼저 나오면 됩니다.
3. 4개이며 `[]`도 하나입니다. 경로를 복사해야 이후 pop이 저장된 답을 바꾸지 않습니다.
4. 같은 인덱스 0. 다음 인덱스로 이동하면 필요한 `[2,2]`를 놓칩니다.

개념 질문 해설

DFS의 진행 중 상태 1은 현재 경로에 있으므로 그곳으로 돌아가는 간선은 사이클입니다. 완료 상태 2는 경로에서 빠진 정점입니다. `0→1`, `0→2`, `2→1`에서 완료된 1의 재방문은 사이클이 아닙니다.

재귀의 return은 그 호출만 끝냅니다. 부모는 호출 다음 줄의 pop을 수행하고 for의 다음 선택을 처리합니다. 긴 그래프는 재귀 제한을 피하는 Kahn 구현으로 풉니다.

실습과 참고자료

LeetCode · 207, 210, 78, 39 · 풀이 · Python ZIP

```
python demo_11.py
python demo_12.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).