ChatGPT

알고리즘 스터디: 8주차 – 그래프 기초 확장 (탐색, 트리 및 위상정렬)

목차

0. 지난주 학습 내용 되짚기

지난 주 7주차에서는 동적 계획법(Dynamic Programming, DP)의 핵심 개념들을 학습했습니다. 작은 부분 문제로 분할하고 이전 결과를 메모이제이션(memoization)을 통해 재사용함으로써, 복잡한 문제의 모든 가능성을 체계적으로 고려하면서도 효율적으로 해답을 찾는 방법을 배웠습니다. 최적 부분 구조(optimal substructure)와 중복 부분 문제(overlapping subproblems)라는 DP의 두 가지 중요한 조건을 이해하고, 이를 다양한 문제에 적용해보면서 DP의 위력을 체감할 수 있었습니다. 예를 들어 **1, 2, 3 더하기**, **1로 만들기**, **평범한 배낭** 등 여러 문제를 풀며 경우의 수 계산, 최적화 문제, 1차원 수열 처리, 2차원 문자열 처리 등 다양한 상황에서 DP를 활용해보았습니다. 매 문제마다 상태를 정의하고 점화식(재귀적 관계)을 찾아내는 연습을 진행하여, **"작은 문제 해답이 모여 큰 문제 해답을 만든다"**는 동적 계획법의 철학을 확실히 익혔습니다.

DP를 통해 완전 탐색에서는 불가능해 보이던 문제들도 다룰 수 있게 되었으며, 코드 구현 시 Top-Down(재귀 + 메모이제이션)과 Bottom-Up(반복문을 통한 타뷸레이션) 두 가지 방식의 장단점도 비교해보았습니다. 이러한 체계적인 사고방식과 최적화 개념은 앞으로 다룰 그래프 알고리즘을 이해하는 데에도 큰 도움이 될 것입니다. 이번 주 8주차에서는 **그래프(Graph)** 알고리즘의 기초로 넘어가서, **깊이/너비 우선 탐색(DFS/BFS)**부터 **그래프의 속성(사이클, 연결성, 이분성)**, 그리고 특별한 그래프인 **트리(Tree)** 구조와 **트리 순회/복원**, **위상정렬** 등의 주제를 학습해보겠습니다.

1. 그래프의 개념과 활용

그래프(Graph)는 현실 세계의 다양한 관계를 표현할 수 있는 강력한 **자료구조**이자 개념입니다. 그래프는 **정점(vertex, 노드)**들의 집합과 이들을 연결하는 **간선(edge)**들의 집합으로 이루어진 구조로, 복잡한 연결 관계를 모델링하기에 적합합니다. 예를 들어 컴퓨터 네트워크의 토폴로지, 소셜 네트워크에서 사람들 간의 친구 관계, 도로 지도에서 도시와 도로의 연결, 문제 해결에서의 상태 공간 탐색 등 수없이 많은 상황을 그래프로 표현할 수 있습니다. 정점은 개별 객체나 상태를 나타내고, 간선은 그 사이의 관계나 이동 가능성을 나타낸다고 볼 때, 그래프를 활용하면 이러한 요소들 간의 **경로 찾기**, **관계 탐색**, **군집 분석** 등의 작업을 알고리즘으로 처리할 수 있습니다.

그래프에는 방향성이 있을 수도 있고 없을 수도 있습니다. **무방향 그래프(undirected graph)**에서는 간선으로 연결된 두 정점이 상호 연결되어 있어 한쪽에서 다른 쪽으로 이동이 항상 가능하지만, **방향 그래프(directed graph)**에서는 간선에 방향이 부여되어 특정 정점에서 다른 정점으로의 단방향 관계를 나타냅니다. 또한 간선에는 가중치(weight)가 부여될 수도 있는데, 이는 두 정점 사이를 이동하거나 연결하는 데 드는 비용, 거리 등을 의미합니다. 이러한 **가중치 그래**

프(weighted graph)에서는 최단 거리나 최소 비용 경로를 찾는 문제가 중요하게 다뤄집니다. 반면 가중치가 없는 **비가중치 그래프(unweighted graph)**에서는 단순 연결 관계만 고려하면 됩니다.

그래프 알고리즘은 문제 해결에 핵심적인 역할을 합니다. **경로 존재 여부**나 **최단 경로 탐색**, **그래프의 연결 여부**와 **구조적 속성** 파악 등은 그래프 이론의 중심 과제들입니다. 이를 위해 그래프를 컴퓨터 내부에서 표현하고 다루는 여러 가지 방법들이 존재하며, 상황에 맞게 적절한 표현 방식을 선택하는 것이 중요합니다. 다음 절에서는 그래프의 대표적인 표현 방식들에 대해 알아보겠습니다.

2. 그래프의 표현 방식 (인접 행렬, 인접 리스트 등)

그래프를 저장하고 처리하기 위해서는 컴퓨터 상에서 효과적으로 표현해야 합니다. 대표적으로 **인접 행렬(adjacency matrix)**, **인접 리스트(adjacency list)**, **간선 리스트(edge list)** 세 가지 방법이 많이 사용됩니다. 각각 메모리 사용량과 접근 방식, 그리고 연산 속도 측면에서 장단점을 지니고 있습니다.

- **인접 행렬**: 인접 행렬은 정점의 개수가 N 일 때 $N \times N$ 크기의 2차원 배열로 그래프 연결 정보를 표현하는 방식입니다. 행과 열은 정점을 나타내고, 특정 행 i 와 열 j 의 교차점에 있는 값이 1(또는 가중치 값)이면 정점 i 에서 정점 j 로 연결되는 간선이 있음을 의미합니다. 없으면 0으로 표시합니다. 무방향 그래프의 경우 대칭 행렬이 되며, 방향 그래프의 경우 대칭이 아닐 수 있습니다. 인접 행렬의 장점은 **임의의 두 정점 연결 여부를 O(1)**에 바로 확인할 수 있다는 것이고, 구현이 단순하며 행렬 연산을 활용한 그래프 알고리즘에 응용하기 쉽다는 점입니다. 반면 단점으로는 **메모리 사용량이** $O(N^2)$ 으로 정점 수가 많을 때 비효율적이며, 연결이 드문 희소 그래프(sparse graph)의 경우에도 큰 행렬 공간을 차지하게 됩니다. 또한 모든 간선을 탐색(iterate)할 때는 행렬의 모든 원소를 확인해야 하므로 $O(N^2)$ 시간이 걸려, 간선 개수가 적은 그래프에서는 비효율적입니다.

- **인접 리스트**: 인접 리스트는 각 정점에 대한 **연결된 이웃 정점들의 목록**을 저장하는 방식입니다. 구현상으로 정점 개수만큼의 리스트(array of lists)를 만들고, 각각의 리스트에 해당 정점과 직접 연결된 인접 정점들을 나열합니다. 예를 들어 1번 정점의 인접 리스트에는 1번과 간선으로 연결된 모든 정점들이 들어있습니다. 무방향 그래프라면 간선을 추가할 때 양쪽 정점의 리스트에 모두 추가하고, 방향 그래프라면 한쪽 방향으로만 추가합니다. 인접 리스트의 장점은 **메모리 효율이 높다**는 것입니다. 간선이 존재하는 연결 관계만 저장하므로, 전체 메모리 사용량이 $O(N + M)$ (정점 수 + 간선 수)에 비례합니다. 특히 간선 수 M 이 정점 수에 비해 훨씬 적은 희소 그래프에서 메모리 낭비가 적습니다. 또한 **모든 간선을 탐색할 때 O(N + M)**의 시간으로 가능하여, 그래프 탐색 알고리즘 구현에 유리합니다. 단점으로는, 임의 두 정점 사이의 연결 여부를 바로 확인하려면 해당 리스트를 일일이 확인해야 하므로 최악의 경우 $O(N)$ 시간이 걸릴 수 있다는 점이 있습니다. 따라서 밀도가 높은 그래프(dense graph)에서는 인접 리스트보다 인접 행렬이 두 정점 연결 확인에는 빠를 수 있습니다.

- **간선 리스트**: 간선 리스트는 말 그대로 그래프의 **간선들을 나열한 목록**입니다. 보통 간선을 나타내는 튜플 (u, v) (및 가중치가 있다면 (u, v, w)) 형태로 모든 간선 정보를 저장합니다. 이 방식은 그래프 구조보다는 개별 간선 중심으로 접근할 때 유용합니다. 예를 들어 간선을 정렬하여 최소 신장 트리(MST)를 구하는 **크루스칼(Kruskal) 알고리즘**에서는 입력을 간선 리스트로 받아 처리하는 편이 자연스럽습니다. 또한 그래프를 입력으로 받을 때 간선 리스트 형태로 주어지는 경우도 많습니다. 간선 리스트의 장점은 구현이 간단하고, 간선 중심의 알고리즘(예: MST, 일부 그래프 DP 등)에 직접 활용하기 좋다는 것입니다. 다만 **정점의 이웃을 효율적으로 찾기 위해서는** 보통 간선 리스트를 인접 리스트나 행렬로 변환하여 사용하는 경우가 많습니다. 순수 간선 리스트만으로 BFS나 DFS를 수행하려면 일일이 원하는 정점의 간선을 검색해야 하기 때문에 비효율적입니다. 따라서 간선 리스트는 주로 **입력 형태**나 **특정 알고리즘 단계**에서 활용되고, 내부 처리에는 인접 리스트/행렬로 변환하여 쓰는 경우가 많습니다.

정리하면, **그래프의 표현 방식 선택은 문제의 크기와 특성에 따라 달라집니다.** 정점 개수가 적고 간선 밀도가 높은 경우 인접 행렬이 단순하고 빠를 수 있으며, 정점이나 간선이 매우 많은 경우 인접 리스트로 메모리를 아끼면서 효율적으로 탐색을 진행하는 것이 일반적입니다. 또한 알고리즘에 따라 간선 목록 형태의 입력을 바로 활용하기도 합니다. 코딩 테스트나 알고리즘 문제 풀이에서는 보통 인접 리스트를 가장 많이 사용하지만, 상황에 맞춰 행렬과 리스트의 장단점을 기억해 두는 것이 좋습니다.

3. 그래프 탐색 (DFS, BFS)

그래프의 기본적인 활용은 **탐색(traversal)**에서 시작합니다. 그래프 탐색이란 하나의 정점에서 시작하여 그래프를 **체계적으로 방문**하는 절차로, 연결된 모든 정점을 찾아내거나 특정 목표를 발견하는 데 사용됩니다. 가장 널리 알려진 그래프 탐색 방법으로 **깊이 우선 탐색(Depth-First Search, DFS)**과 **너비 우선 탐색(Breadth-First Search, BFS)**이 있습니다. 두 알고리즘 모두 그래프의 모든 정점을 방문할 수 있으며, 구현 방법과 방문 순서에 차이가 있을 뿐 **시간 복잡도는** $O(N+M)$ 으로 동일합니다 (N은 정점 수, M은 간선 수). 아래에서는 DFS와 BFS의 동작 원리와 구현 방법을 살펴보겠습니다.

깊이 우선 탐색(DFS): 말 그대로 **깊게 우선으로** 탐색하는 방법입니다. 한 정점에서 시작하여 갈 수 있는 경로가 있으면 계속 **한 방향으로 쭉 내려가며** 탐색을 진행하고, 더 이상 진행할 수 없게 되면 다시 뒤로 돌아와 다른 경로를 탐색합니다. 이 과정은 **재귀 호출**이나 **명시적 스택(stack)**을 이용하여 구현할 수 있습니다. DFS의 특징은 분기(branch)를 최대한 깊숙이 탐색하기 때문에, 특정 경로를 따라 탐색하는 과정에서 **백트래킹(backtracking)**이 자연스럽게 이루어진다는 점입니다. 예를 들어 미로 찾기나 퍼즐 해결에서 DFS는 한 길을 끝까지 따라가 보고 막히면 다시 돌아와 다른 길을 시도하는 식으로 활용됩니다. 구현 시에는 방문한 정점을 표시하는 **방문 배열(visited)**이 필요합니다. 이미 방문한 정점을 다시 방문하지 않도록 체크하지 않으면, **사이클이 있는 그래프의 경우 무한 루프**에 빠질 수 있기 때문입니다. 재귀적 DFS 구현은 간결하지만, **재귀 호출의 깊이**가 너무 깊어질 경우 스택 오버플로우에 주의해야 합니다. 반복문과 명시적 스택을 사용하는 방법으로도 구현할 수 있으며, 이 경우 스택 자료구조에 시작 정점을 넣고, 루프를 돌며 스택에서 하나씩 꺼내 처리하는 방식으로 동작합니다.

```
# DFS (재귀 구현) 예시
graph = {                          # 그래프를 인접 리스트로 표현 (딕셔너리 사용 예시)
    1: [2, 3],
    2: [4, 5],
    3: [5],
    4: [],
    5: [6],
    6: []
}
visited = set()                    # 방문한 정점을 기록할 집합

def dfs(v):
    if v in visited:               # 이미 방문한 정점은 무시
        return
    visited.add(v)
    print(v, end=" ")              # 방문한 정점 출력 (방문 순서 확인용)
    for w in graph[v]:             # 모든 인접 정점에 대해
        dfs(w)                     # 재귀적으로 DFS 수행

dfs(1)   # 1번 정점부터 DFS 시작
# 출력 예시: 1 2 4 5 6 3
```

위의 DFS 재귀 구현은 그래프를 파이썬 딕셔너리로 표현하고, `dfs` 함수를 통해 깊이 우선으로 방문하는 예입니다. 1번 정점에서 출발하여 인접한 2, 3을 차례로 탐색하되, 2를 탐색하는 도중에는 2의 이웃들을 모두 방문하고, 2를 통해 더 이상 방문할 곳이 없을 때 비로소 3을 탐색하는 것을 볼 수 있습니다. 출력된 방문 순서 `1 2 4 5 6 3`를 보면 1->2를 깊이 파고들어 4까지 간 후, 4에서 막혔으니 돌아와 5, 그리고 5의 이웃 6까지 방문한 뒤, 더 이상 2를 통해 갈 곳이 없으므로 다시 처음으로 돌아와 3을 방문하는 순서임을 알 수 있습니다. 이처럼 DFS는 한 경로를 끝까지 탐색한 후에 다음 경로로 넘어가는 **후입선출(LIFO)** 형태의 탐색을 수행합니다.

너비 우선 탐색(BFS): BFS는 **넓게 우선으로** 탐색하는 방법입니다. 시작 정점으로부터 가까운 정점들을 먼저 모두 방문한 뒤에 차츰 멀리 있는 정점들을 탐색하는 방식으로, **선입선출(FIFO)** 구조인 **큐(queue)**를 활용하여 구현합니다. BFS의 작동 원리는, 시작 정점에서 출발하여 먼저 그 이웃들을 모두 큐에 넣고 방문 처리한 뒤, 큐에서 차례로 꺼내면서 해당 정점의 이웃들을 또 방문하는 식으로 진행된다는 것입니다. 이렇게 하면 **거리가 1인 정점들**, 그 다음 **거리 2인 정점들**… 순으로 탐색이 이루어집니다. BFS의 중요한 특징은 **최단 거리 특성**인데, **가중치가 없는 그래프**에서 BFS를 활용하면 어떤 정점으로 가는 **최소 이동 횟수**를 자연스럽게 찾을 수 있습니다. 이는 BFS가 계층적(layer-by-layer)으로 탐색하기 때문에 가능한 일로, 나중에 다룰 최단 경로 문제의 기초가 되기도 합니다.

BFS 구현 또한 방문 배열(또는 집합)을 사용하여 중복 방문을 방지해야 하며, 큐를 이용하기 때문에 파이썬에서는 `collections.deque`를 활용하면 편리합니다. 아래는 BFS의 대표적인 구현 예시입니다.

```
from collections import deque

# 위와 동일한 그래프 사용 (1: [2,3], 2: [4,5], 3: [5], 4: [], 5: [6], 6: [])
visited = set()
def bfs(start):
    q = deque([start])          # 큐에 시작 정점을 넣음
    visited.add(start)
    while q:
        v = q.popleft()         # 큐에서 맨 앞의 정점을 꺼냄 (FIFO)
        print(v, end=" ")       # 방문한 정점 출력
        for w in graph[v]:
            if w not in visited:    # 아직 방문하지 않은 이웃 정점을
                visited.add(w)
                q.append(w)             # 큐에 넣고 방문 처리

bfs(1)   # 1번 정점부터 BFS 시작
# 출력 예시: 1 2 3 4 5  (또는 1 2 3 5 4 등, 그래프 구조에 따라 순서 달라질 수 있음)
```

위 코드에서 1번 정점에서 시작한 BFS 탐색은 먼저 1의 이웃들인 2와 3을 방문 큐에 넣고, 1을 처리합니다. 그 다음 2를 꺼내어 2의 이웃들(4, 5)을 방문 큐에 넣고, 이어 3을 꺼내어 3의 이웃(5)을 넣으려고 시도하지만 5는 이미 2를 통해 방문 큐에 들어갔으므로 중복 삽입하지 않습니다. 이후 큐에 남은 4, 5, 6을 차례로 처리하면 탐색이 완료됩니다. BFS의 방문 순서는 1과 인접한 정점들이 먼저, 그 다음 단계의 정점들로 진행되므로, 출력 결과를 보면 1 -> 2,3 -> 4,5 -> 6 순으로 탐색이 이뤄진 것을 확인할 수 있습니다. 이처럼 BFS는 **층층이 퍼져나가는 방식**으로 그래프를 탐색하며, **최단 거리 문제에서 매우 유용**합니다.

DFS와 BFS는 그래프 탐색의 양대 기본 방법으로, 문제의 요구에 따라 적합한 방식을 선택하거나 두 방법을 조합하여 활용합니다. 예를 들어 **모든 연결 요소를 탐색**하여 개수를 세는 문제라면 DFS나 BFS 아무거나 사용해도 상관없지만, **최단 이동 횟수**를 찾아야 한다면 BFS가 더 적합합니다. 반면 **경로 탐색**이나 **백트래킹을 통한 조합 탐색** 등에서는 DFS가 주로 활용됩니다. 두 알고리즘 모두 그래프 탐색의 기초이므로, 동작 원리를 확실히 이해하고 구현 패턴을 익혀두어야 합니다.

4. 그래프의 주요 속성 (사이클, 연결성, 이분 그래프)

그래프를 탐색하는 기법을 배웠다면, 이를 바탕으로 그래프의 중요한 **속성(properties)**들을 판단할 수 있습니다. 그래프의 구조적 특성을 이해하면 그 그래프가 품고 있는 문제를 파악하거나, 적절한 알고리즘을 선택하는 데 도움이 됩니다. 여기서는 **사이클 존재 여부**, **연결성**, **이분 그래프 여부**라는 세 가지 핵심 속성을 살펴보겠습니다.

사이클 판별: **사이클(cycle)**은 그래프에서 **시작점과 동일한 점으로 되돌아올 수 있는 경로**를 말합니다. 무방향 그래프에서는 DFS나 BFS 중 탐색 도중에 **이미 방문한 정점**을 다시 만났을 때 (부모 노드가 아닌 경우) 사이클이 존재한다고 판단할 수 있습니다. 예를 들어 DFS를 수행하다가 어느 정점의 이웃이 이미 방문된 상태인데, 그 이웃이 현재 탐색 중인 경로 상의 **부모 정점이 아니라면** 이는 앞쪽 어딘가에서 이미 방문한 정점을 다시 만난 경우이므로 사이클을 이룹니다. 이를 활용하여 **DFS 기반 사이클 검출** 알고리즘을 구현할 수 있습니다. 방향 그래프의 경우에는 **DFS 중 현재 탐색 스택에 있는 정점(방문은 했지만 아직 탐색 완료되지 않은 정점)을 다시 만나면** 사이클이 발견되었다고 판단합니다. 한편, **위상정렬 알고리즘**을 활용하여 간접적으로 사이클 존재 여부를 판단하기도 합니다: 방향 그래프에서 위상정렬 결과 얻은 정점의 수가 원래 정점 수보다 적다면 사이클이 있다는 신호입니다 (자세한 내용은 뒤의 위상정렬 절에서 설명). 사이클 여부를 알고 나면, **그래프가 순환 구조를 갖는지** 혹은 **트리와 같이 비순환 구조인지**를 구분할 수 있게 되어 문제 해결 방향이 달라질 수 있습니다.

연결성 확인: **연결 그래프(connectivity)**인지 여부는 그래프의 모든 정점들이 서로 이어져 있는지를 나타내는 성질입니다. 무방향 그래프의 경우 **한 정점에서 다른 임의의 정점까지 경로가 존재하면** 모두 하나의 연결 요소(connected component)에 속해있다고 합니다. 만약 일부 정점들이 떨어져 있어서 서로 도달할 수 없다면 그래프는 둘 이상의 연결 요소로 분리됩니다. 방향 그래프에서는 강한 연결성(strong connectivity) 개념이 따로 있으며, **모든 정점쌍에 대해 서로 도달 가능한 경우**를 강하게 연결되었다고 정의합니다 (방향까지 고려하기 때문에 무방향보다 조건이 까다롭습니다). 일반적인 그래프 탐색으로 연결성을 확인하려면, 임의의 한 정점에서 DFS/BFS를 수행하여 **방문 가능한 모든 정점을 표시**하고, 탐색 종료 후 방문하지 못한 정점이 있다면 그래프가 불완전하게 연결되었다는 뜻입니다. 이런 식으로 반복 탐색하면 각각의 탐색이 하나의 연결 성분(component)을 찾아내게 되고, 연결 성분의 개수를 세는 문제도 풀 수 있습니다. 연결성 판단은 **네트워크 구성**이나 **클러스터링** 등에서 중요하며, 나아가 특정 정점을 제거하거나 간선을 제거했을 때 그래프가 어떻게 분리되는지를 보는 **단절점(articulation point)**, **단절선(bridge)** 등의 개념으로 확장되기도 합니다.

이분 그래프 판별: **이분 그래프(bipartite graph)**란 정점 집합을 두 그룹으로 나누었을 때 **모든 간선이 서로 다른 그룹에 속한 정점들을 연결**하도록 할 수 있는 그래프를 말합니다. 쉽게 말해, 그래프의 정점을 두 색으로 색칠(coloring)했을 때 **어떠한 간선도 같은 색 정점끼리 연결하지 않도록** 칠할 수 있다면 그 그래프는 이분 그래프입니다. 이를 판별하는 알고리즘으로는 BFS나 DFS를 활용한 **그래프 두색 칠하기**가 대표적입니다. 예를 들어 DFS를 수행하면서 인접한 정점에는 현재 정점과 다른 색을 할당하고 내려가는 식으로 탐색합니다. 만약 탐색 중에 **인접한 두 정점이 이미 같은 색으로 색칠되어 있어야 하는 상황이 발생하면** 이분 그래프가 아님을 알 수 있습니다. 또는 BFS를 활용하면 시작 정점을 색 A로 칠하고, 그 이웃들은 색 B, 그 다음 이웃의 이웃들은 다시 색 A… 번갈아 칠하면서 진행합니다. 진행 중에 충돌이 생기면 이분 그래프가 아닙니다. 이분 그래프는 **사이클과 밀접한 관련**이 있는데, 사실 **홀수 길이 사이클(odd-length cycle)**이 존재하면 이분 그래프가 될 수 없고, 이분 그래프인 그래프에는 홀수 사이클이 존재하지 않습니다. 이분 그래프 판별은 **그래프를 둘로 나누는 문제**나 **2색 문제** (예: 팀 나누기, 대립 관계 표현 등)에 자주 응용됩니다. 또한 이분 그래프에서는 **매칭(matching)** 알고리즘 등 특수한 문제들이 잘 정의되기 때문에, 주어진 그래프가 이분 구조인지 확인하는 것은 실전 문제 풀이에서도 중요한 단계가 될 수 있습니다.

요약하면, 그래프 탐색을 응용하여 **사이클 존재 여부**, **그래프의 연결 요소 파악**, **이분 가능 여부** 등을 효율적으로 판별할 수 있습니다. 이러한 속성들은 그래프의 유형과 활용 가능한 알고리즘 범위를 결정짓는 요소이므로, 문제를 접했을 때 해당 그래프의 성질을 파악하는 습관을 들이는 것이 좋습니다.

5. 트리의 기본 개념 및 특성 (지름, 서브트리 등)

트리(Tree)는 그래프의 한 특별한 형태로, **사이클이 없는 연결 그래프(무방향)**를 말합니다. 다시 말해, N개의 정점을 가진 트리는 정확히 N-1개의 간선을 가지며, **어떤 두 정점 사이에도 사이클이 존재하지 않는** 구조입니다. 트리는 그래프의 특수한 경우이지만, 별도로 떼어내어 많이 다루는 이유는 이 구조만의 유용함과 단순함 때문입니다. **계층적 구조(hierarchical structure)**를 나타내기에 좋아서, 예컨대 기업의 조직도나 폴더 디렉토리, 가계도와 같은 데이터를 표현할 때 트리가 활용됩니다. 또한 트리는 그래프 알고리즘에서 다양한 **분할 정복**이나 **DP**의 무대로 쓰이는데, 이는 트리가 갖는 구조적 특성 덕분입니다.

트리의 중요한 특성 중 하나는 **두 정점 사이의 경로가 유일하다**는 것입니다. 사이클이 없고 연결되어 있으므로 임의의 두 노드 간에 경로가 딱 하나만 존재하게 됩니다. 이 특성은 트리에서의 탐색과 자료 저장을 단순하게 만들어 줍니다. 또한 트리에는 **루트(root)** 개념을 도입할 수 있는데, 아무 노드나 하나 선택하여 루트로 삼으면 트리는 자연스럽게 **계층 구조**를 갖게 됩니다. 부모-자식 관계, 형제 노드, 깊이(depth), 레벨(level) 등의 개념이 정의되며, 특히 **이진 트리(binary tree)**처럼 각 노드의 자식 수에 제한을 둔 트리 구조는 알고리즘 이론과 구현에서 매우 자주 등장합니다.

여기서는 트리의 몇 가지 핵심 속성을 살펴보겠습니다.

- **트리의 지름**: 트리의 지름(tree diameter)이란 트리에 존재하는 **가장 긴 경로의 길이**를 말합니다. 두 노드 간의 경로 중 가장 많은 간선을 거치는 경로의 길이가 지름이 되며, 해당 경로에 있는 노드의 개수를 지름으로 정의하기도 합니다 (간선 수 기준인지 노드 수 기준인지 문맥에 따라 다름). 트리의 지름을 구하는 고전적인 방법은 **두 번의 DFS/BFS**를 사용하는 것입니다. 임의의 한 노드에서 시작하여 가장 먼 노드를 찾고, 그 노드로부터 다시 한 번 DFS/BFS를 수행하여 가장 먼 노드를 찾으면 그 두 번째 탐색에서 얻은 거리가 지름이 됩니다. 이는 트리가 가지고 있는 특성을 이용한 것으로, 첫 탐색에서 얻은 말단 노드는 지름 경로 상의 한 극단임을 보장할 수 있기 때문에 나오는 결과입니다. 트리의 지름은 **트리 구조에서 가장 멀리 떨어진 두 지점**을 찾는 것이므로, 통신망에서 지연 시간이 가장 긴 두 지점이라든가, 조직도에서 가장 거리가 먼 두 직급 등으로 비유할 수 있습니다. 문제 풀이에서는 트리의 지름을 활용하여 응용 문제들을 해결하는 경우가 많습니다 (예: 트리 위의 최장 경로에 어떤 제약을 두는 문제 등).

- **서브트리**: 트리에서 **서브트리(subtree)**란 특정 노드를 루트로 하는 하위 트리를 말합니다. 어떤 노드와 그 노드의 모든 후손(descendants)을 모으면 하나의 서브트리가 되는데, 이는 원 트리의 부분 트리에 해당합니다. 트리 구조에서는 한 노드를 제거하면 그 아래에 속했던 노드들이 통째로 분리되어 하나의 서브트리를 이루기 때문에, **트리 DP**나 **서브트리 크기 계산** 등의 작업이 용이합니다. 예를 들어, 루트가 있는 트리에서 각 노드의 서브트리 크기(해당 노드를 루트로 하는 서브트리의 노드 개수)를 계산하고 싶다면, DFS를 이용하여 후손 노드들을 모두 탐색한 뒤 돌아오면서 개수를 세어주면 됩니다. 사이클이 없는 구조 덕분에 **한 번 방문한 노드의 부모로 돌아갈 때 그 서브트리의 모든 처리가 완료된 것**을 보장할 수 있기 때문입니다. 이런 방식으로 트리에서는 다양한 속성(노드 수, 합계 값, 최댓값 등)을 서브트리 단위로 손쉽게 계산할 수 있습니다.

- **기타 특성**: 트리는 사이클이 없으므로 **한 노드를 제거하면 그래프가 분리**됩니다. 이러한 점에서 트리의 간선은 모두 **단절선(bridge)**의 역할을 합니다 (간선을 하나 제거하면 연결 요소가 둘 이상으로 늘어남). 또한 트리에서는 **레벨 순회(level order traversal)**라 하여 BFS를 이용한 계층별 순회도 자주 사용됩니다 (예: 완전 이진 트리의 노드를 레벨별로 출력하기 등). 트리 구조에서는 그래프의 일반적인 속성들이 특수하게 단순화되는데, 예를 들어 **이분 그래프** 여부를 따질 필요도 없습니다 (트리는 항상 이분 그래프로 간주될 수 있음, 왜냐하면 사이클이 없으므로 홀수 사이클이 존재하지 않기 때문입니다). 이러한 단순함 덕분에, 그래프 문제를 풀 때 그래프가 트리 구조라고 주어지면 문제 난도가 크게 낮아지는 경우도 많습니다. 대신 노드 개수가 N 일 때 간선도 $N-1$ 개로 정해져 있어서, 그래프 알고리즘의 복잡도 분석 시 M 을 $N-1$ 로 치환해 생각하면 되는 장점도 있습니다.

6. 트리 순회 (전위, 중위, 후위) 및 트리 복원

트리가 특히 **이진 트리(binary tree)** 구조로 주어졌을 때, **트리 순회(tree traversal)**라는 개념이 등장합니다. 이는 트리의 각 노드를 어떤 **순서**로 방문할 것인지를 나타내는 것으로, 방문 순서에 따라 **전위(preorder)**, **중위(inorder)**, **후위(postorder)** 순회 세 가지로 크게 분류됩니다. (여기서 전/중/후위는 루트 노드를 언제 방문하느냐에 따라 달라지는 명칭입니다.)

- **전위 순회**: Root -> Left -> Right 순서로 방문합니다. 즉, 어떤 노드를 방문했을 때 바로 그 노드를 처리(출력 등)한 후에 왼쪽 서브트리, 오른쪽 서브트리 순으로 내려갑니다. 전위 순회 결과의 첫 번째 원소는 항상 **루트 노드**가 된다는 특징이 있습니다. 예를 들어 전위 순회로 트리를 탐색하면 루트부터 시작해서 최상위 노드들을 먼저 쭉 방문하게 되므로, 트리의 구조를 개략적으로 파악하기 좋습니다.

- **중위 순회**: Left -> Root -> Right 순서로 방문합니다. 왼쪽 자식을 모두 방문한 후에 자신을 처리하고, 그 다음 오른쪽으로 넘어가는 방식입니다. **이진 탐색 트리(BST)**의 경우 중위 순회하면 값이 정렬된 순서로 출력된다는 중요한 성질이 있습니다. 일반 이진 트리에서는 반드시 그렇지는 않지만, 중위 순회를 하면 루트 노드는 결과의 중앙쯤에 나오게 됩니다.

- **후위 순회**: Left -> Right -> Root 순서입니다. 자식들을 모두 방문한 뒤에 부모를 나중에 방문하는 방식으로, **폴더를 삭제하거나 포스트오더 방식으로 연산을 처리**할 때 주로 사용됩니다. 예컨대 이진 트리의 **후위 표기법(postfix)** 계산이나 디렉토리의 크기를 계산하는 문제에서 후위 순회를 활용합니다. 후위 순회 결과의 마지막 원소는 항상 루트 노드가 됩니다.

위의 세 가지 순회 방식은 각각 트리의 노드들을 방문하는 **서로 다른 순서**일 뿐이며, 모든 노드를 한 번씩 방문한다는 점에서는 동일합니다. 특히 전위, 중위, 후위 순회의 결과를 알면 그에 따라 **트리 구조를 복원**할 수 있다는 것이 중요한데, 일반적으로 **중위 순회 결과**와 다른 한 가지 순회 결과(전위 또는 후위)를 이용하면 **이진 트리를 유일하게 복원**할 수 있습니다. 이는 알고리즘 문제에서도 자주 등장하는 유형입니다.

트리 복원: 전위+중위 순회 또는 후위+중위 순회 결과가 주어졌을 때 원래의 트리를 구성하는 방법은 **재귀적 분할 정복**으로 이해할 수 있습니다. 전위 순회 결과에서는 첫 번째 값이 루트라는 것을 이용하고, 해당 값이 중위 순회 결과의 어디에 위치하는지를 찾아보면 그 값 기준으로 중위 결과가 왼쪽 서브트리와 오른쪽 서브트리로 나뉩니다. 중위 결과에서 루트 값의 왼쪽 부분이 왼쪽 서브트리의 중위 순회 결과, 오른쪽 부분이 오른쪽 서브트리의 중위 결과가 됩니다. 전위 결과에서는 루트 다음에 나오는 값들이 왼쪽 서브트리의 전위 결과, 그 다음이 오른쪽 서브트리의 전위 결과라는 것을 유추할 수 있습니다. 이렇게 **분할된 부분들에 대해 재귀적으로 같은 복원 작업**을 반복하면 트리가 완성됩니다. 후위+중위의 경우에도 유사한 논리로, 후위 순회 결과의 마지막 값이 루트라는 점을 활용하여 분할하면 됩니다.

코드로 구현하면 아래와 같은 형태가 됩니다. (편의를 위해 전위+중위 결과로 트리를 복원하는 예시를 보여줍니다.)

```
# 전위(preorder)와 중위(inorder) 결과를 이용한 이진 트리 복원
preorder = [3, 9, 20, 15, 7]      # 예시 전위 순회 결과
inorder  = [9, 3, 15, 20, 7]      # 예시 중위 순회 결과

def rebuild(pre, ino):
    if not pre or not ino:
        return None
    root = pre[0]                          # 전위 순회 결과의 첫 값이 루트
    root_index = ino.index(root)           # 중위 순회 결과에서 루트 위치 찾기
    left_ino = ino[:root_index]            # 중위 기준 왼쪽 부분 -> 왼쪽 서브트리
    right_ino = ino[root_index+1:]         # 중위 기준 오른쪽 부분 -> 오른쪽 서브트리
    # 왼쪽 부분의 크기에 따라 전위 결과를 분할
    left_pre = pre[1:1+len(left_ino)]
    right_pre = pre[1+len(left_ino):]
    # 재귀적으로 트리 구축
    return {
        "value": root,
        "left": rebuild(left_pre, left_ino),
        "right": rebuild(right_pre, right_ino)
    }

tree = rebuild(preorder, inorder)
print(tree)
# 출력 예시 (중첩 딕셔너리로 트리 표현):
# {'value': 3, 'left': {'value': 9, 'left': None, 'right': None},
```

```
# 'right': {'value': 20, 'left': {'value': 15, 'left': None, 'right': None},
#         'right': {'value': 7, 'left': None, 'right': None}}}
```

위 코드는 전위/중위 순회 결과를 가지고 재귀적으로 이진 트리를 복원하는 파이썬 구현입니다. 출력은 파이썬 딕셔너리로 만들어진 트리 구조를 보여주는데, `value` 키에 노드 값, `left`와 `right` 키에 각각 왼쪽, 오른쪽 자식 노드가 재귀적으로 담겨있습니다. 예시에서 복원된 트리 구조는 루트 값 3, 왼쪽 자식 9, 오른쪽 자식 20 (그 아래 왼쪽 자식 15, 오른쪽 자식 7)인 트리로서, 주어진 전위/중위 순회 결과와 일치하는 트리입니다. 이처럼 순회 결과를 다루는 문제에서는 **재귀적 사고**가 중요하며, 작은 트리 조각을 복원하여 큰 트리를 완성해가는 방법을 익혀두어야 합니다.

트리 순회와 복원은 면접 질문이나 알고리즘 문제에서도 자주 등장하는 주제입니다. 특히 순회 결과를 보고 트리를 구성하거나, 반대로 트리를 순회하여 특정 순서로 출력하는 코드를 작성하는 것은 트리 개념 이해의 기본이므로 확실히 짚고 넘어가야 합니다.

7. 위상정렬 (Kahn 알고리즘 및 DFS 기반)

위상정렬(Topological Sort)은 **방향 그래프(directed graph)**의 특별한 정렬로, **싸이클이 없는 방향 그래프(DAG, Directed Acyclic Graph)**에서 정의됩니다. 위상정렬 결과는 그래프의 모든 정점을 나열한 리스트로서, **모든 간선 (u -> v)에 대해 u가 v보다 앞에 나오는** 순서를 뜻합니다. 쉽게 말해 **선행 조건을 위반하지 않도록 정렬한 순서**라고 이해할 수 있습니다. 예를 들어 선수 과목이 있는 학습 과정에서 과목들을 수강 순서대로 나열한다거나, 프로젝트의 작업들을 의존성에 따라 일정 순으로 배치하는 것이 위상정렬에 해당합니다. DAG에 대해서는 항상 하나 이상의 위상정렬 결과가 존재하며, 만약 그래프에 사이클이 있다면 **위상정렬은 불가능**합니다 (아무리 순서를 나열해도 사이클을 깨트릴 수 없으므로).

위상정렬을 수행하는 알고리즘으로는 크게 두 가지가 많이 알려져 있습니다:

- **Kahn 알고리즘**: 큐를 사용하는 방법으로, **차수(indegree)** 개념을 활용합니다. 각 정점에 대해 **진입 차수** (indegree), 즉 다른 정점에서 들어오는 간선의 개수를 계산하고 시작합니다. 그런 다음 **진입 차수가 0인 정점들**을 모두 큐에 넣습니다. 진입 차수가 0이라는 것은 선행 조건을 갖지 않는 작업이라는 뜻이므로, 이것들이 먼저 수행될 수 있습니다. 큐에서 하나씩 정점을 꺼내면서 그것을 위상정렬 결과 리스트에 추가하고, 그 정점에서 나가는 간선들을 제거(다른 정점들의 진입 차수를 1씩 감소)합니다. 그리고 새롭게 진입 차수가 0이 된 정점이 있으면 큐에 넣습니다. 이 과정을 더 이상 꺼낼 정점이 없을 때까지 반복하면, 방금 꺼낸 순서가 바로 위상정렬 결과가 됩니다. 만약 도중에 큐가 비었는데 아직 모든 정점을 처리하지 못했다면, 그래프에 사이클이 있어서 위상정렬을 완료할 수 없다는 의미입니다. Kahn 알고리즘의 시간 복잡도는 $O(N + M)$ 이며, 간단한 자료구조 연산으로 이루어져 있어서 구현이 비교적 쉬운 편입니다. 아래는 Kahn 알고리즘의 흐름을 간략히 보여주는 의사코드 형태의 파이썬 예시입니다.

```
from collections import deque

N = 6  # 정점 개수 (예시)
graph = {                    # 방향 그래프의 인접 리스트 표현 (예시)
    1: [2, 5],
    2: [3],
    3: [4],
    4: [],
    5: [4],
    6: [2, 5]
}
indegree = {i: 0 for i in range(1, N+1)}
for u in graph:
```

```
    for v in graph[u]:
        indegree[v] += 1

q = deque([v for v in indegree if indegree[v] == 0])
topo_order = []
while q:
    u = q.popleft()
    topo_order.append(u)
    for w in graph[u]:
        indegree[w] -= 1
        if indegree[w] == 0:
            q.append(w)

if len(topo_order) < N:
    print("사이클 존재로 위상정렬 불가")
else:
    print("위상정렬 순서:", topo_order)
# 출력 예시: 위상정렬 순서: [1, 6, 2, 5, 3, 4]
```

위 예시는 방향 그래프에서 1,6 두 정점이 처음에 진입 차수가 0이라 큐에 들어가고, 그 다음 1을 꺼내면서 1 -> 2, 1 -> 5 간선을 제거하여 2와 5의 차수를 감소시킵니다. 이어 6을 꺼내면서 6 -> 2, 6 -> 5 간선을 제거합니다. 이때 2의 차수가 0이 되면 큐에 추가, 마찬가지로 5도 차수가 0이 되면 추가됩니다. 이렇게 진행하면 얻어진 `topo_order`가 위상정렬 결과가 됩니다. 한 가지 그래프에 대해 여러 가지 위상정렬 결과가 있을 수 있는데, 이는 동시에 진입 차수 0인 정점들 사이의 순서에는 자유도가 있기 때문입니다. 예를 들어 위 예시에서 1과 6의 순서, 또는 2와 5의 순서는 여러 가지가 될 수 있습니다.

- **DFS 기반 알고리즘**: 깊이 우선 탐색을 활용하여 위상정렬을 얻는 방법도 있습니다. 이 방법은 재귀적인 DFS를 수행하면서 **모든 인접 노드들을 방문 완료한 후에 해당 노드를 스택에 쌓는** 방식으로 구현됩니다. 구체적으로, 각 정점에 대해 DFS를 수행하되, **이미 방문한 노드는 건너뛰고**, DFS 재귀 호출이 끝난 후에 자기 자신을 결과 목록 (또는 스택)에 추가하는 것입니다. 이렇게 하면 **후위 순회**의 일종으로 모든 선행 노드들이 결과에 자신보다 먼저 들어가게 되며, 결국 스택에 쌓인 역순이 위상정렬 순서를 이룹니다. 구현 시 주의할 점은, 사이클을 감지하기 위해 **방문 상태**를 체크해야 한다는 것입니다. 예를 들어, DFS 도중 **현재 탐색 중인 경로에 있는 노드를 다시 만나면** 사이클이 존재함을 알 수 있습니다. 이를 위해 각 노드에 대해 **방문 안 함(unvisited)**, **방문 진행 중(visiting)**, **방문 완료(visited)**의 세 가지 상태를 관리하고, 방문 진행 중인 노드를 다시 만나면 사이클이라고 판단할 수 있습니다. 사이클이 없다면 DFS 종료 후 스택에 쌓인 노드들을 순서대로 꺼내면 위상정렬 결과가 얻어집니다. 이 알고리즘도 기본적으로 DFS의 간선 탐색을 그대로 수행하므로 **시간 복잡도는** $O(N+M)$ 이며, 재귀 호출을 사용한다는 점을 제외하면 Kahn 알고리즘과 성능상 큰 차이는 없습니다.

위상정렬은 **선행 조건이 있는 작업 나열 문제**에 광범위하게 활용됩니다. 예를 들어 **과목 수강 순서 결정**, **작업 스케줄링**, **컴파일 순서 결정(파일 간 의존성 처리)**, **선수 그래프를 이용한 순위 계산** 등에서 쓰입니다. 또한 알고리즘 문제에서도 **DAG 최장 경로**나 **게임 진행 순서** 등을 물어볼 때 위상정렬을 응용하게 됩니다. 기본적인 위상정렬 알고리즘은 꼭 이해하고 구현할 수 있어야 하며, 특히 큐를 이용한 Kahn 알고리즘은 직관적이고 사이클 검출에도 응용되니 확실히 짚고 넘어가세요.

8. 마무리 및 다음 주 예고

이번 주는 그래프 알고리즘의 기초를 확장하여 **그래프 표현, 탐색, 속성 분석, 트리와 위상정렬**까지 폭넓은 내용을 다루었습니다. 새로운 개념이 많이 등장하였지만, 큰 그림에서는 그래프라는 통일된 주제를 중심으로 다양한 문제 상황을 해결하는 방법들을 배운 것입니다. **DFS/BFS 탐색**을 통해 그래프를 다루는 기본기를 익혔고, 이를 응용하여 **사이클 존재 여부나 연결 요소 파악, 이분 그래프 판별** 등의 구조적 분석을 할 수 있음을 보았습니다. 또한 **트리**라는 특별한 그래프에

대해 알아보고, 트리에서만 가능한 효율적인 계산 방법들과 **트리 순회/복원** 같은 흥미로운 주제도 실습해보았습니다. 마지막으로 **위상정렬** 개념을 통해 방향 그래프의 선후 관계를 유지하는 정렬 방법을 학습하였습니다. 이 모든 내용은 그래프 이론의 토대가 되는 중요한 부분들이므로, 시간을 두고 복습하면서 **직접 코드로 구현**해 보길 권장합니다. 예를 들어 간단한 그래프를 손으로 그려보고 BFS와 DFS를 수행해 방문 순서를 추적해보거나, 임의의 트리를 만들어 전위/중위/후위 순회를 구해본 뒤 코드로 검증해보는 것도 좋은 연습이 됩니다.

그래프 알고리즘을 다루기 시작하면, 응용 범위가 매우 넓다는 것을 깨닫게 될 것입니다. 예를 들어 **트리 구조**는 컴퓨터 공학 전반에서 중요하게 사용되며, 파일 시스템, 데이터베이스 인덱스, 트라이(Trie) 등 다양한 곳에 응용됩니다. **위상정렬**은 앞서 언급한 작업 스케줄링 외에도, 게임 개발에서 행동 순서를 결정하거나, 복잡한 규칙이 엮인 시스템에서 **Deadlock**이 발생하지 않는 순서를 찾는 등에 활용됩니다. DFS와 BFS는 더욱 다양한 문제 해결의 기반이 됩니다. **미로 탐색**이나 **퍼즐 해결**에서 상태 공간 그래프를 탐색할 때, BFS는 최단 해결책을 찾는 데 쓰이고 DFS는 모든 해결책을 찾는 데 쓰이는 식입니다. 또한 BFS는 **최단 경로**를 찾기 위한 기본 알고리즘으로, 가중치가 없는 경우에는 최단 거리 문제를 정확하고 빠르게 풀어냅니다.

다음 주 9주차에서는 이번에 배운 그래프의 기초를 바탕으로, **그래프의 심화 알고리즘**들을 다룰 예정입니다. 특히 **최단 경로 알고리즘**으로 유명한 **다익스트라(Dijkstra)**와 **플로이드-워셜(Floyd-Warshall)** 알고리즘을 통해 가중치 그래프에서 최소 비용 경로를 찾는 기법들을 배우고, **최소 신장 트리(MST)** 알고리즘인 **크루스칼(Kruskal)**과 **프림(Prim)**을 통해 그래프에서 최소 연결 구조를 찾는 방법도 익힐 것입니다. 최단 경로 문제는 사실 DP의 한 예로도 볼 수 있기 때문에, 지난 주에 익힌 DP 원리가 그래프 알고리즘에도 어떻게 적용되는지 살펴보게 될 것입니다. 뿐만 아니라, **벨만-포드(Bellman-Ford)** 알고리즘이나 **위상정렬을 응용한 DAG 최단 경로** 등 그래프만의 특수한 알고리즘들도 등장할 예정입니다.

이번 주에 학습한 그래프 탐색과 트리, 위상정렬 등의 개념은 이러한 심화 주제들을 이해하기 위한 **밑바탕**입니다. 예를 들어, **다익스트라 알고리즘**은 기본적으로 BFS와 유사한 방식으로 동작하되 우선순위 큐를 활용하여 가장 가까운 정점을 선택하는 전략을 취합니다. **최소 신장 트리** 알고리즘들은 결국 트리를 만들어가는 과정이므로, 트리 개념에 대한 이해가 선행되어야 합니다. 위상정렬 역시 향후 **사이클이 없는 그래프의 DP**나 **일정 계획 문제** 등에 다양하게 쓰일 것이니 잘 익혀두세요.

마지막으로, 이번 주 배운 내용들을 반드시 복습하시고 직접 구현해보시기 바랍니다. 그래프 개념은 초반에 헷갈릴 수 있지만, **손으로 그려가며** 노드와 간선의 관계를 이해하면 점차 익숙해질 것입니다. 특히 DFS/BFS 구현이나 위상정렬 구현 같은 코드는 몇 번씩 작성해 보는 것을 추천드립니다. 트리 순회도 작은 예시를 만들어 출력 결과를 예측해보고 확인해 보세요. 궁금하거나 어려운 부분이 있다면 스터디 동료들과 토의하면서 개념을 공고히 하시기 바랍니다. 그럼 다음 주 알고리즘 스터디에서 **최단 경로 및 MST** 등 더욱 흥미로운 그래프 알고리즘들로 만나겠습니다. 수고하셨습니다!
