ChatGPT

알고리즘 스터디: 7주차 – 동적 계획법의 핵심과 실전 문제 해결

목차

0. 지난주 학습 내용 되짚기

지난 주 6주차에서는 효율적인 **탐색 기법**들을 학습하였습니다. 배열이나 리스트 상에서 부분 해답을 빠르게 찾는 투 포인터(two pointers) 및 슬라이딩 윈도우(sliding window) 기법을 익혀서, 이중 루프를 단일 루프로 줄이는 방법을 배웠습니다. 또한 정렬된 데이터에 대해 **이분 탐색(binary search)**과 **파라메트릭 서치(parametric search)**를 활용하여 탐색 범위를 로그 수준으로 줄이는 기법도 다루었습니다. 이와 함께 Meet-in-the-Middle(중간에서 만나기), 삼분 탐색(ternary search) 등 추가 개념들을 간단히 소개하며 다양한 문제 상황에서 알고리즘을 적용하는 전략을 넓혀보았습니다. 이러한 기법들을 통해 시간 복잡도를 획기적으로 개선하는 방법과 서로 다른 알고리즘들을 조합하는 중요성을 체감할 수 있었습니다.

이번 주 7주차에서는 **동적 계획법(Dynamic Programming, DP)**을 학습합니다. 그동안 정렬, 탐색, 그리디 등으로는 풀기 어려웠던 복잡한 문제들을 해결하는 핵심 기법으로서 DP를 다룰 예정입니다. 작은 부분 문제로 쪼개고 이전 결과를 재사용함으로써 **모든 가능성을 체계적으로 고려**하면서도 효율성을 놓치지 않는 방법을 배워볼 것입니다. 메모이제이션(memoization)을 통한 **중복 계산 회피**와 문제의 **상태 정의** 및 **최적 부분 구조(optimal substructure)** 활용 등이 이번 주 학습의 중요한 포인트입니다. 예제 코드와 단계별 설명을 따라가며 **동적 계획법의 기본 개념과 문제 해결 패턴**을 확실히 익혀봅시다.

1. DP의 탄생 배경

동적 계획법(이하 DP)은 1950년대에 미국 수학자 리처드 벨만(Richard Bellman)에 의해 제안된 기법으로, 복잡한 최적화 문제를 효율적으로 풀기 위해 고안되었습니다. DP의 기본 아이디어는 **복잡한 문제를 작은 부분 문제로 분할**하고, 그 부분 문제들의 해를 조합하여 원래 문제의 해답을 얻는 것입니다. 이때 동일한 부분 문제가 여러 번 반복되면 매번 새로 푸는 대신 **한 번만 풀고 저장해두었다가 재사용**하여 불필요한 계산을 없앱니다. 이렇게 하면 지수적으로 폭발하던 계산량을 **다항 시간**으로 줄일 수 있게 됩니다.

DP라는 용어에서 "동적(dynamic)"은 문제를 단계적으로 **변화**(진행)시켜 나간다는 뉘앙스를, "계획법(programming)"은 수학적 계획 혹은 표 채우기 방식을 의미합니다. 이름은 복잡하지만, 실상 **재귀적 관계를 활용하여 해를 구하는 체계적인 방법**이라고 볼 수 있습니다. DP의 근간에는 두 가지 중요한 특징이 있습니다:

- **Optimal Substructure (최적 부분 구조)**: 문제의 최적 해결 방법이 그 하위 부분 문제들의 최적 해결 방법으로 구성되는 성질을 말합니다. 쉽게 말해, 큰 문제의 최적해에 작은 문제들의 최적해가 포함되어 있다는 것입니다. 이 특성이 있어야 DP로 문제를 풀 수 있는 토대가 마련됩니다. (예시: 최단 거리 문제에서 특정 경로의 최단 경로에도 부분 경로들의 최단경로가 포함됨)

- **Overlapping Subproblems (중복 부분 문제)**: 동일한 작은 부분 문제가 여러 번 반복해서 나타나는지를 뜻합니다. 이런 중복이 존재하면 **한 번만 계산해서 결과를 재사용**하는 방식으로 효율을 크게 높일 수 있습니다. (예시: 피보나치 수열에서 F(10)을 구할 때 F(5) 등의 계산이 여러 경로로 중복되어 등장)

이러한 성질들을 만족하는 문제라면 **동적 계획법**을 적용하여 효과적으로 해결할 수 있습니다. DP는 본래 수학, 공학 분야의 최적화 문제(예: 경로 최적화, 운영 계획 등)에서 출발했지만, 오늘날 프로그래밍 대회나 코딩 테스트에서는 **배열, 문자열, 그래프 등 다양한 분야의 문제 해결에 두루 활용**되는 중요한 알고리즘 기법입니다. 특히 **피보나치 수 계산**, **최단 경로 탐색**, **배낭 채우기 문제** 등에서 DP의 개념이 자연스럽게 등장합니다. 예를 들어, 단순 재귀로 계산하면 지수 시간이 걸리는 피보나치 수열도 DP를 쓰면 중복 계산을 막아 선형 시간에 구할 수 있습니다. 이러한 맥락에서 DP는 **브루트 포스나 백트래킹의 지수적 탐색을 다항 시간으로 최적화**하는 강력한 도구로 탄생하게 되었습니다.

2. 두 가지 접근법 (Top-Down, Bottom-Up)

DP를 구현하는 방식에는 크게 **Top-Down(탑다운)**과 **Bottom-Up(바텀업)** 두 가지 접근법이 있습니다. 두 방식은 문제를 해결하는 흐름이 다르지만 결국 동일한 DP 원리가 적용되며, 결과도 같아야 합니다.

- **Top-Down** 방식은 큰 문제를 해결하기 위해 재귀적으로 **하위 문제를 풀어가는 과정**입니다. 처음에 풀고자 하는 큰 문제에서 시작하여, 필요한 작은 문제들을 재귀 호출로 구하고, 그 결과를 이용해 답을 얻습니다. 이때 동일한 부분 문제가 여러 번 호출되지 않도록 **메모이제이션(메모리 저장)**을 활용합니다. 한 번 계산한 작은 문제의 결과는 배열이나 딕셔너리 등에 저장해 두고, 이후 다시 필요할 때 계산 없이 바로 가져다 씁니다. 이런 방식 때문에 Top-Down을 흔히 **재귀 + 메모이제이션** 기법이라고 부릅니다. 필요한 부분 문제들만 계산하므로, 문제에 따라서는 전체 공간을 탐색하지 않고도 답을 얻는 이점도 있습니다. 다만 재귀 호출 스택을 사용하므로 **언어의 재귀 한계**나 **함수 호출 오버헤드**에 주의해야 합니다.

- **Bottom-Up** 방식은 가장 작은 부분 문제들(기본/base case)부터 차근차근 **모든 경우를 계산하여 답을 쌓아올리는 과정**입니다. 일반적으로 배열이나 표(table)에 작은 문제들의 해답을 저장해나가면서 결국 원하는 큰 문제의 해답에 도달합니다. 이를 **타뷸레이션(tabulation)** 방식이라고도 합니다. Bottom-Up은 명시적인 재귀 없이 **반복문(iteration)**으로 구현되므로, 재귀 호출 부담이 없고 구조가 단순한 경우가 많습니다. 단, 문제의 모든 부분 경우를 다 계산하므로 Top-Down에 비해 불필요한 계산이 약간 들어갈 수 있지만(필요 없던 부분까지도 채워나가므로), 현대 컴퓨터 환경에서는 대부분 무시할 수 있는 수준이고 예측 가능한 실행 시간이 나온다는 장점이 있습니다.

두 접근법은 각각 장단점이 있지만, **정답을 얻는 원리나 효율은 결국 동일한 DP**입니다. 구현상의 편의로 선택하면 되고, 어떤 문제들은 재귀로 표현하면 더 직관적이어서 Top-Down이 코딩하기 쉬운 반면, 어떤 문제들은 차라리 반복문으로 푸는 Bottom-Up이 깔끔하기도 합니다. 중요한 것은 **문제의 상태(state)와 점화식(recurrence)을 올바르게 수립**하는 것입니다. 아래에서 소개할 예제 문제를 통해 두 방식 중 하나를 택해 구현해 보겠습니다 (필요하면 다른 방식으로도 구현이 가능함을 언급하겠습니다).

예제 문제 – 1, 2, 3 더하기 (BOJ 9095)

이 문제는 정수 N을 1, 2, 3의 합으로 나타내는 방법의 개수를 구하는 유명한 DP 문제입니다. 예를 들어 N=4일 때, 합으로 표현하는 방법은 `1+1+1+1`, `1+1+2`, `1+2+1`, `2+1+1`, `2+2`, `1+3`, `3+1`로 총 7가지가 있습니다. **중복되는 부분 문제** 구조가 뚜렷하여 DP로 효율적으로 해결 가능합니다. 작은 수부터 답을 계산하여 큰 수의 답을 만들 수 있는데, N에 대한 해를 구하려면 N-1, N-2, N-3에 대한 해를 알아야 함을 쉽게 추론할 수 있습니다.

우리는 **Bottom-Up** 방식으로 이 문제를 풀어보겠습니다. DP상태 `dp[x]`를 "정수 x를 1,2,3의 합으로 나타내는 방법의 수"로 정의합니다. 그러면 **점화식**은 다음과 같습니다:
- `dp[x] = dp[x-1] + dp[x-2] + dp[x-3]`

이는 마지막에 사용하는 수가 1인 경우, 2인 경우, 3인 경우로 나누어 생각한 합산이며, 자연스럽게 중복 없는 모든 경우를 세어줍니다. **기본 값(초기 조건)**으로는 `dp[0] = 1` (아무 숫자도 사용하지 않는 경우를 한 가지 방법으로 센다)이며, `dp[1]=1`, `dp[2]=2` 등이 됩니다. 이 점화식에 따라 1부터 N까지 차례로 값을 채워나가면 최종적으로 dp[N]이 답이 됩니다. 아래 코드는 이 로직을 구현한 파이썬 코드입니다. (Top-Down으로 구현할 경우 재귀적으로 dp(n) = dp(n-1)+dp(n-2)+dp(n-3)을 계산하고 메모이제이션하면 됩니다.)

```
# BOJ 9095: 1, 2, 3 더하기
# 문제: 정수 N을 1, 2, 3의 합으로 나타내는 방법의 수를 구하기 (중복 순서 다른 경우 별도로 계산).
# 접근: DP Bottom-Up 방식. dp[x] = dp[x-1] + dp[x-2] + dp[x-3].
# 입력: 첫 줄에 테스트 케이스 T, 이후 T개의 줄에 정수 N (1 <= N <= 11).
# 출력: 각 N마다 가능한 표현 방법의 수 출력.

import sys

input = sys.stdin.readline
T = int(input().strip())
# 테스트 케이스별 결과를 계산해야 하므로 최대 필요한 N까지 DP 준비
nums = [int(input().strip()) for _ in range(T)]
max_n = max(nums)

# DP 테이블 초기화 (0부터 max_n까지)
dp = [0] * (max_n + 1)
dp[0] = 1  # dp[0]: 합을 만들지 않는 방법은 한 가지 (아무 것도 선택하지 않음)
if max_n >= 1:
    dp[1] = 1  # 1을 만드는 방법: "1" (1가지)
if max_n >= 2:
    dp[2] = 2  # 2를 만드는 방법: "1+1", "2" (2가지)

# Bottom-Up DP 계산
for i in range(3, max_n + 1):
    dp[i] = dp[i-1] + dp[i-2] + dp[i-3]

# 각 테스트 케이스 결과 출력
for n in nums:
    print(dp[n])
```

위 코드에서는 미리 입력으로 주어질 N값들 중 최대값까지 DP 테이블을 계산해 두고, 각 질의에 답하도록 구현했습니다. 점화식 `dp[i] = dp[i-1] + dp[i-2] + dp[i-3]`에 따라 작은 수의 결과를 활용해 큰 수의 결과를 구하

는 Bottom-Up 패턴을 볼 수 있습니다. (참고로 Top-Down 방식으로 풀었다면 재귀함수 `solve(n)`을 정의하고, `solve(n) = solve(n-1)+solve(n-2)+solve(n-3)` 구조로 작성하며, 한 번 계산된 n에 대한 결과는 메모에 저장해 두는 식으로 구현했을 것입니다.)

3. 점화식과 DP 구조

DP 문제를 풀 때 **가장 중요한 단계는 문제를 DP로 모델링**하는 것입니다. 이는 곧 **상태(State)**를 정의하고 그 사이의 관계, 즉 **점화식(Recurrence Relation)**을 수립하는 작업입니다. DP 상태는 문제의 부분 해를 나타내는 변수(또는 변숫 집합)이며, 점화식은 이 상태들 간의 **관계와 전이 방식**을 설명합니다. 올바른 점화식을 세우기 위해서는 문제의 구조를 깊이 이해하고, **이전 단계와 다음 단계 해답 간의 연결고리**를 찾아야 합니다.

점화식을 세울 때 고려해야 할 요소들: - **상태 정의**: DP 배열이나 테이블에서 인덱스(또는 키)가 의미하는 바를 명확히 해야 합니다. 1차원일지 다차원일지, 각 차원이 뜻하는 것은 무엇인지 등을 결정합니다. 예를 들어, "`dp[i]` = i길이의 수열에 대한 최적값" 또는 "`dp[i][j]` = i와 j까지 고려했을 때의 값" 같은 식으로 정의합니다. - **전이 및 점화식**: 한 상태에서 다른 상태로 어떻게 이동(계산)할 수 있는지를 규칙으로 정리합니다. 일반적으로 "`dp[current]` = combine(`dp[past]` ...)" 형태로, 현재 상태의 값은 이전 상태들의 값으로부터 결정됩니다. 이때 **어떤 이전 상태들이 필요한지**와 **어떻게 결합하는지**를 도출합니다. - **초기값(기저 사례)**: 재귀적 정의를 끝낼 수 있는 가장 작은 부분 문제들의 해답을 지정합니다. 보통 `dp[0]`, `dp[1]` 등의 값이나, 2차원 DP의 `dp[0][j]`, `dp[i][0]` 등 경계 조건을 채워넣습니다.

점화식을 수립했다면, 이는 일종의 **수학적 공식**이나 **재귀 관계**로 볼 수 있습니다. 이후 구현은 Top-Down이면 재귀+메모이제이션, Bottom-Up이면 반복문+테이블 채우기로 하면 됩니다. 중요한 것은 DP 테이블을 채울 때 **모든 필요한 상태를 빠짐없이, 그리고 올바른 순서로 계산**하는 것입니다. 특히 Bottom-Up에서는 어떤 순서로 채워나가야 각 상태를 계산할 때 필요한 이전 상태들이 이미 구해져 있는지 신경 써야 합니다 (예: `dp[i]` 계산에 `dp[i-1]` 등이 필요하면 dp는 작은 수에서 큰 수 순으로 채워야 함).

예제 문제 – 1로 만들기 (BOJ 1463)

다음은 DP에서 **최적화 문제**의 고전적인 예시입니다. 정수 N이 주어졌을 때, N을 1로 만들기 위해 사용할 수 있는 연산은 **1) 3으로 나누기, 2) 2로 나누기, 3) 1을 빼기** 세 가지입니다. 이 연산들을 최소 횟수로 사용해서 N을 1로 만들 때의 **최소 연산 횟수**를 구하는 문제입니다. 예를 들어 N=10일 경우, 10 → 9 → 3 → 1로 3번 만에 1로 만들 수 있습니다 (최소 횟수). Greedy하게 접근하면 항상 2나 3으로 나눌 수 있을 때 나눈다 식으로 생각할 수 있지만, 경우에 따라 최적해가 그런 단순 규칙만으로 나오지 않을 수 있어 **DP로 모든 가능성을 고려**하는 것이 안전한 문제입니다.

이 문제의 DP 풀이 구조를 생각해 봅시다. 상태 `dp[x]`를 "정수 x를 1로 만드는 최소 연산 횟수"로 정의할 수 있습니다. 그러면 x에서 1로 가는 방법은 다음 한 가지 연산을 먼저 적용한 뒤, 남은 부분을 해결하는 형태로 나타낼 수 있습니다. x에서 취할 수 있는 연산별로 나누어 보면:

- 만약 x가 3으로 나누어 떨어지면, `x -> x/3` 연산을 사용할 수 있고 이 경우 필요한 총 횟수는 `dp[x/3] + 1`입니다.
- 만약 x가 2로 나누어 떨어지면, `x -> x/2` 연산을 사용할 수 있고 이 경우 횟수는 `dp[x/2] + 1`입니다.
- 언제든지 `x -> x-1` 연산을 사용할 수 있고 이 경우 횟수는 `dp[x-1] + 1`입니다.

이 중 **최소의 연산 횟수**를 선택하면 되므로, 점화식은:
- `dp[x] = 1 + min(dp[x-1], dp[x/2] (가능하면), dp[x/3] (가능하면))`

물론 `x`가 해당 연산의 조건을 만족할 때만 고려합니다. 예를 들어 10의 경우 `dp[10] = 1 + min(dp[9], dp[5], dp[...?])` (10은 3으로 안 나눠지니 dp[10/3]은 제외). 기본적으로 `dp[1] = 0` (1은 1로 만드는 데 연산 0번)으로 두고, 2부터 N까지 위 규칙으로 최소 횟수를 채워나가면 됩니다. 이 문제는 작은 수의 답이 큰 수의 답을

결정하므로 **Bottom-Up**이 자연스럽고, Greedy로 섣불리 판단하면 최적해를 놓칠 수 있으므로 DP가 정확한 해답을 보장합니다.

```
# BOJ 1463: 1로 만들기
# 문제: 정수 N을 1로 만들 때 사용하는 연산(÷3, ÷2, -1)의 최소 횟수 구하기.
# 접근: DP로 최소 연산 횟수 계산. dp[x] = min(dp[x-1], dp[x/2]?, dp[x/3]?) + 1.
# 입력: 첫 줄에 정수 N (1 <= N <= 1,000,000).
# 출력: N을 1로 만들기 위한 최소 연산 횟수.

N = int(input().strip())
dp = [0] * (N + 1)
dp[1] = 0   # 1은 시작점, 연산 0번

for i in range(2, N + 1):
    # 기본적으로 -1 연산을 한 경우로 초기화
    dp[i] = dp[i-1] + 1
    # 2로 나눠지는 경우 비교
    if i % 2 == 0:
        if dp[i//2] + 1 < dp[i]:
            dp[i] = dp[i//2] + 1
    # 3으로 나눠지는 경우 비교
    if i % 3 == 0:
        if dp[i//3] + 1 < dp[i]:
            dp[i] = dp[i//3] + 1

print(dp[N])
```

위 코드는 1부터 N까지 순차적으로 dp값을 채우는 Bottom-Up 방식입니다. 각 단계에서 `-1`을 하는 경우를 기본으로 하고, 나눗셈이 가능한 경우를 체크하여 최소값을 취하도록 구현했습니다. 이렇게 하면 `dp[2] = dp[1]+1 = 1`, `dp[3] = 1` (3 -> 1 한 번), `dp[4] = dp[3]+1 = 2` (4 -> 3 -> 1), ..., `dp[10] = min(dp[9], dp[5]) + 1 = min(2,3)+1 = 3`과 같이 채워집니다. 이 점화식의 수립으로 모든 경우를 다 따져보지 않고도 최적해를 효율적으로 구할 수 있었습니다.

이렇듯 **점화식 수립**은 DP 문제 해결의 알파이자 오메가입니다. 문제를 읽고 DP로 풀기로 결정했다면, 먼저 상태와 점화식을 종이에라도 써 보는 습관을 들이면 좋습니다. 한 번 점화식이 완성되면 코딩 단계는 비교적 수월해집니다.

4. 최적화 테크닉

DP를 적용하다 보면 경우에 따라 **메모리나 시간 면에서의 최적화 기법**이 필요해질 수 있습니다. 기본적인 DP 솔루션이 정답의 아이디어를 제공하지만, 그대로 구현하면 **메모리 부족**이 발생하거나 **시간 초과**가 나는 상황이 있을 수 있습니다. 이럴 때는 점화식의 특성을 살펴 **불필요한 부분을 줄이거나 자료구조를 개선**하는 등의 최적화 테크닉을 동원해야 합니다.

대표적인 DP 최적화 아이디어 몇 가지를 소개하면:

- **메모리 최적화**: 많은 DP 솔루션들이 큰 테이블(배열)을 필요로 하지만, 모든 값을 항상 저장해둘 필요는 없습니다. **이전 단계의 값 몇 개만 있으면 다음 값을 구할 수 있는** 경우, 배열을 작게 유지하거나 **1차원으로 압축**할 수 있습니다. 예를 들어 피보나치 수열이나 위의 1,2,3 더하기 문제에서는 `dp[i]` 계산에 `dp[i-1],`

`dp[i-2], dp[i-3]`만 참조되므로, 길이 N의 배열 전체를 유지하지 않고 직전 3개의 값만 저장해도 됩니다. 비슷하게 2차원 DP에서 이전 행만 참조한다면 배열을 2행으로 줄이거나 아예 1행으로 만들 수 있습니다.

- **시간 최적화 (점화식 개선)**: 어떤 DP는 점화식상 **중첩 루프**가 필요한 경우가 있습니다. 예를 들어 기본적인 최장 증가 부분 수열(LIS) DP를 구현하면 `O(N^2)` 시간이 걸리고, 100만개의 입력이 주어지면 수행이 불가능합니다. 이런 경우, 점화식을 수학적 통찰이나 자료구조로 최적화할 수 있는지 살펴봐야 합니다. LIS 문제는 이분 탐색을 활용하여 `O(N log N)`에 풀 수 있는 알고리즘이 존재합니다. 또 다른 예로 **0-1 배낭 문제**의 기본 DP는 아이템 N개, 무게 한도 W에 대해 `O(N*W)` 시간이 걸리지만, 특정 조건에서 **무게에 대한 이분 탐색 최적화**나 **가치에 대한 DP**로 바꾸는 등 개선이 가능하고, 때로는 **Meet-in-the-Middle 기법**으로 절반씩 나눠 푸는 최적화도 고려됩니다.

- **DP + 자료구조 활용**: DP 점화식 중에 최솟값이나 최댓값을 구하기 위해 많은 상태를 훑는다면, **세그먼트 트리**나 **큐** 등을 사용해 빠르게 구할 수 있습니다. 예를 들어 어떤 DP에서 `dp[i] = 1 + min(dp[j])` (조건을 만족하는 j 범위 내) 같은 식이라면, 단순 계산은 느리지만 슬라이딩 윈도우 최적화나 세그먼트 트리로 최소값 쿼리를 해소할 수 있습니다. 이러한 **모노톤 큐 최적화**나 **Convex Hull Trick** 같은 고급 기법들은 특수한 DP에서 시간 복잡도를 줄여주는 역할을 합니다.

초심자 단계에서는 우선 **메모리 최적화** 정도에 집중하면 좋습니다. 아래 예제에서는 0-1 Knapsack(배낭 문제)에서의 메모리 최적화를 보여드립니다.

예제 문제 – 평범한 배낭 (BOJ 12865)

이 문제는 전형적인 **0-1 배낭 문제**로, 무게와 가치가 주어진 N개의 물건 중 일부를 골라 최대 무게 K 이하로 배낭에 넣을 때 얻을 수 있는 **최대 가치**를 구하는 문제입니다. 기본적인 DP 풀이는 `dp[i][w]`를 "앞에서부터 i번째 물건까지 고려했을 때 배낭 용량 w로 얻을 수 있는 최대 가치"로 정의하여 2차원 DP 테이블을 채우는 것입니다. 점화식은 다음과 같습니다 (물건 i의 무게를 `weight[i]`, 가치를 `value[i]`로 가정):

- 물건 i를 배낭에 **담지 않는 경우**: `dp[i][w] = dp[i-1][w]` (이전까지 고려한 상황과 가치 동일)
- 물건 i를 배낭에 **담는 경우** (가능한 경우만): `dp[i][w] = dp[i-1][w - weight[i]] + value[i]`
- 위 두 경우 중 큰 값을 취함: `dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i]] + value[i])` (단, weight[i] <= w)

2차원 배열 `dp[N][K]`를 채워서 `dp[N][K]`가 답이 됩니다. 하지만 제한 조건에 따라 `N`이 100, `K`가 100,000까지 갈 수 있어서, dp 테이블의 크기는 약 100 * 100,000 = 10^7 수준이 됩니다. 파이썬에서 2차원 리스트 100만 단위는 메모리 사용과 초기화 시간 면에서 부담이 될 수 있습니다. **메모리 최적화 기법**을 적용하면, 위 점화식에서 `dp[i]` 행을 계산할 때 참조하는 것은 오직 `dp[i-1]` 행뿐이라는 점에 착안할 수 있습니다. 즉, **과거 한 행만 있으면 현재 행은 계산하고, 이전 행은 더 이상 쓰지 않는** 형태입니다. 따라서 굳이 2차원 전체를 유지할 필요 없이 1차원 배열 하나로 갱신해도 됩니다. 다만 **값을 덮어쓸 때 주의**가 필요합니다. 현재 1차원 배열을 갱신하면서 이전 값들이 망가진 상태로 참조되지 않도록, **배낭 문제에서는 용량 w를 큰 값부터 작은 값 순으로 순회**해야 합니다. 이렇게 하면 dp[w] 값 갱신에 dp[w - weight]의 **이전 단계 값**이 보존되어 사용됩니다.

아래 코드는 이러한 1차원 DP 최적화를 적용한 풀이입니다.

```
# BOJ 12865: 평범한 배낭
# 문제: N개 물건 (각 무게 W, 가치 V). 최대 무게 K 배낭에 넣어 얻을 수 있는 최대 가치 출력.
# 접근: 0-1 Knapsack DP. 2차원 dp 대신 1차원 dp[w] 사용 (이전 아이템 고려 값 누적).
# 입력: 첫 줄에 N, K. 다음 N개 줄에 각 물건의 W, V.
# 출력: 배낭에 넣을 수 있는 최대 가치.
```

```
import sys
input = sys.stdin.readline

N, K = map(int, input().split())
items = [tuple(map(int, input().split())) for _ in range(N)]

# 1차원 DP 테이블: dp[w] = 최대 가치
dp = [0] * (K + 1)

for weight, value in items:
    # 현재 아이템을 고려하여 역순으로 DP 갱신
    # 역순을 도는 이유: 같은 아이템을 두 번 사용하는 것을 방지하고 이전 단계 값을 유지하기 위함
    for w in range(K, weight - 1, -1):
        # 현재 용량 w에서 이 아이템을 담을 수 있다면, 담는 경우 vs 담지 않는 경우 비교
        candidate = dp[w - weight] + value  # 이 아이템 담았을 때 가치
        if candidate > dp[w]:
            dp[w] = candidate

print(dp[K])
```

위 코드에서는 2차원 배열 없이 `dp` 한 행으로만 결과를 누적합니다. 각 아이템에 대해 용량 K부터 weight까지 거꾸로 내려오면서 `dp[w]` 값을 갱신하는데, 이렇게 함으로써 `dp[w - weight]`는 아직 현재 아이템을 고려하기 **전의 값**(즉 `dp[i-1]` 단계)을 유지하고 있으므로 올바른 계산이 이뤄집니다. 만약 순방향으로 w=0부터 올리면서 갱신하면 한 아이템을 여러 번 넣는 잘못된 계산이 되니 주의해야 합니다. 이러한 테크닉으로 메모리 사용을 크게 줄이고, 실제 파이썬 구현에서 약간의 속도 향상도 얻습니다.

이 밖에도, 문제에 따라서는 **점화식 자체를 개선**해서 시간복잡도를 낮춰야 하는 경우도 있습니다. 예를 들어, 앞서 언급한 LIS(가장 긴 증가 부분 수열)는 기본 DP로 `O(N^2)`이지만, 이분 탐색을 활용해 `O(N log N)` 알고리즘을 사용하면 N이 매우 큰 경우에도 풀 수 있습니다. 이러한 최적화는 해당 알고리즘의 아이디어를 별도로 알아야 하지만, **DP적 사고**를 기반으로 성능까지 끌어올린다는 점에서 도전 가치가 있습니다.

요약하자면, **DP 최적화 테크닉**은 상황에 따라 다양하게 존재합니다. 우선은 **불필요한 메모리 낭비를 줄이는 1차원 DP, 메모이제이션 활용** 등에 익숙해지고, 더 나아가 **문제 특화 알고리즘**이나 **자료구조**와의 결합을 통해 DP를 빠르게 만드는 방법도 차츰 접해보면 좋겠습니다.

5. DP vs 다른 패러다임

동적 계획법은 다른 알고리즘 패러다임들(분할 정복, 그리디, 완전 탐색 등)과 구별되는 고유한 강점을 지닙니다. 여기서는 DP를 **그리디(Greedy)** 및 **완전 탐색(브루트포스/backtracking)**과 비교해보고, 적절한 적용 시점을 생각해보겠습니다.

- **DP vs Greedy:** 그리디 알고리즘은 매 단계에서 **국소 최적 선택**을 함으로써 전체 최적을 이루려는 접근입니다. 그리디는 구현이 간단하고 속도가 빠르지만, **문제에 따라 오답을 내기 쉽습니다**. 국소 최적이 항상 전역 최적을 보장하려면 문제 자체에 특정 조건(예: Greedy-choice property, matroid 구조 등)이 있어야 합니다. 반면 DP는 **모든 가능성을 다 고려**하여 그 중 최적해를 찾아냅니다. 따라서 특정 조건이 없는 일반 상황에서도 올바른 답을 구할 수 있지만, 계산량이 커질 수 있기에 **중복 부분 문제**가 있을 때만 실용적으로 동작합니다. 예를 들어, **동전 거스름돈 문제**에서 동전 단위가 특정 조건 (예: 배수 관계)에 있으면 그리디로 최적해를 구할 수 있지만, 일반적인 동전 단위 세트에서는 그리디가 실패하고 DP로 모든 경우를 고려해야 합니다.

- **DP vs 완전 탐색(Bruteforce/Backtracking):** 완전 탐색은 가능한 해를 **전부 생성하여 검사**하는 방식으로, 최적해는 찾을 수 있지만 **시간 복잡도 폭발** 문제가 있습니다. DP는 완전 탐색으로 풀 수 있는 많은 문제들을 **중복 부분 문제 제거**를 통해 **효율화한 방법**으로 볼 수 있습니다. 백트래킹에 가지치기(pruning)를 극한으로 적용한 것이 DP라는 말도 있습니다. 예를 들어, 백트래킹으로 모든 경우의 수를 탐색하는 재귀 코드는 비슷한 계산을 여러 번 반복할 수 있는데, DP는 한 번 계산한 부분 결과를 메모하여 백트래킹이 다시 같은 계산을 하지 않게 합니다. 따라서 완전 탐색으로 접근할 때 입력 크기가 조금만 커져도 불가능한 문제들이 DP 덕분에 가능해지는 경우가 많습니다. 하지만 **모든 완전 탐색 문제가 DP로 바뀌는 것은 아닙니다** – 문제 구조가 DP 조건(중복 부분 문제, 최적 부분 구조)을 만족해야만 합니다.

간단히 말해, **Greedy**와 **DP**는 모두 최적해를 찾지만 접근법이 상반됩니다. Greedy는 일부 문제에서 빠르고 간단하지만 적용 조건을 만족해야 하고, DP는 더 일반적으로 적용되나 그만큼 계산 비용이 듭니다. **완전 탐색**과 **DP**는 해를 찾는 범용성 면에서 비슷하지만, DP는 구조를 활용해 효율을 높인다는 점이 다릅니다.

예제 문제 – 동전 2 (BOJ 2294)

이 문제는 **동전 거스름돈** 문제의 한 변형으로 볼 수 있습니다. 다양한 단위의 동전들이 주어졌을 때, 특정 금액을 만들기 위한 **최소 동전 개수**를 구하는 문제입니다. Greedy로 풀어도 될 것처럼 보이지만, 동전 단위에 따라 Greedy 해법이 최적을 보장하지 못하는 경우가 있습니다. 예를 들어, 동전 단위가 4, 6이고 목표금액이 8일 때, Greedy로는 6+?으로 시작하여 8을 만들 수 없지만 사실은 4+4로 2개 동전으로 만들 수 있습니다. 이런 경우 Greedy는 실패합니다. DP로 풀면 모든 조합을 다 고려하여 최적해(여기서는 2개)를 찾아낼 수 있습니다.

우리는 이 문제를 **DP**로 풀어서 Greedy와의 차이를 확인해봅니다. DP상태 `dp[x]`를 "금액 x를 만들 때 필요한 최소 동전 개수"로 정의합니다. 가능한 전이는 주어진 동전 하나를 사용했을 때로 생각할 수 있습니다: 즉, 금액 x에서 어떤 동전 c를 하나 쓰면 남는 금액은 x-c이고, 따라서 `dp[x-c]`를 알고 있다면 `dp[x] = dp[x-c] + 1`이 됩니다. 모든 동전에 대해 가능한 경우를 따져 최소를 취하면 됩니다. 점화식은:
- `dp[x] = min( dp[x - coin] + 1 )` for all coin values (사용 가능한 경우만)

초기값으로 `dp[0] = 0` (0원을 만드는 데 동전 0개)이고, 만들 수 없는 금액은 무한대에 준하는 큰 값으로 초기 설정합니다. 이 점화식은 동전의 순서와 관계없이 조합으로 보기 때문에, **모든 동전 종류를 다 고려**하며 풀어야 합니다. 구현은 1차원 DP로도 가능하며, 동전의 사용 순서는 상관없으므로 한 번 계산한 dp[x]는 이후에도 확정값으로 사용됩니다.

```
# BOJ 2294: 동전 2
# 문제: 주어진 동전들로 금액 K를 만들 때 필요한 최소 동전 개수 (불가능하면 -1).
# 접근: DP로 모든 조합 탐색. dp[x] = min(dp[x - coin] + 1 for each coin).
# 입력: 첫 줄에 N, K. 다음 N개 줄에 동전 가치.
# 출력: 만들 수 있으면 최소 동전 개수, 없으면 -1.

import sys
input = sys.stdin.readline

N, K = map(int, input().split())
coins = [int(input().strip()) for _ in range(N)]

MAX = 10001  # 동전 개수 최대치를 위한 큰 값 (임의로 설정)
dp = [MAX] * (K + 1)
dp[0] = 0  # 0원은 동전 0개 필요

for x in range(1, K+1):
    for coin in coins:
        if x - coin >= 0 and dp[x-coin] != MAX:
```

```
            # coin 사용 가능하고, (x-coin)을 만들 수 있는 경우
            candidate = dp[x-coin] + 1
            if candidate < dp[x]:
                dp[x] = candidate

# 결과 출력
if dp[K] == MAX:
    print(-1)  # 만들 수 없는 경우
else:
    print(dp[K])
```

위 코드는 모든 금액 x에 대해 모든 동전 coin을 시도하며 최소 개수를 찾는 전형적인 DP입니다. 시간 복잡도는 `O(N*K)`로, N(동전 종류 수)과 K(목표 금액)의 곱에 비례합니다. N과 K가 크면 비효율적일 수 있지만, 입력 제약 내에서는 충분히 돌아갑니다. Greedy와 달리 동전 단위 사이의 관계를 가리지 않고 항상 정답을 구할 수 있다는 점이 핵심입니다. (만약 동전 단위가 **정렬되어 있고 배수 관계**라면 Greedy로도 풀 수 있지만, 일반적인 케이스를 대비해 DP를 쓰는 것이 안전합니다.)

이 예제에서 볼 수 있듯, 그리디는 인간이 직관적으로 접근하기 쉽지만 문제에 따라 오답을 낼 수 있고, DP는 느릴 수 있으나 **문제 조건만 만족하면 반드시 정답**이라는 신뢰성이 있습니다. 실제 문제 해결 상황에서는 **문제가 작은 입력에서는 Greedy나 완전 탐색으로 풀리지만, 큰 입력에서는 DP를 도입해야 해결 가능한지**를 판단하는 것이 중요합니다. 그 판단은 연습을 통해 경험을 쌓아야 하며, 본 스터디의 다양한 문제풀이가 그런 감을 익히는데 도움이 될 것입니다.

6. 클래식 DP 아키타입

동적 계획법으로 풀 수 있는 문제들은 매우 다양하지만, 그 중에서도 **전형적이고 자주 등장하는 문제 유형**들이 있습니다. 이러한 클래식 문제들을 알아두면 새로운 DP 문제를 만났을 때 해결 실마리를 더 빨리 찾을 수 있습니다. 몇 가지 대표적인 DP 아키타입과 예시를 소개합니다:

- **피보나치 수열 및 간단한 수열 DP**: DP의 가장 기초적인 형태로, 수열이나 계단 오르기 문제 등이 여기에 속합니다. 앞서 다룬 `1, 2, 3 더하기` (BOJ 9095) 문제나, `피보나치 수 (BOJ 2748)` 계산 등이 대표적입니다. 점화식이 직접적으로 주어지는 형태로, DP 개념을 처음 연습하기 좋습니다.

- **배낭 문제(Knapsack)**: 조합 최적화의 전형으로, **한정된 자원 안에서 최대 가치를 찾는 문제**입니다. 0-1 배낭뿐만 아니라 배낭 문제의 변형들이 많이 출제됩니다. 예를 들어 **동전 거스름돈** (앞의 동전 문제들), **부분합/부분곱 최적화** 등이 여기에 포함됩니다. 배낭 문제 유형은 보통 2차원 DP 또는 1차원 최적화로 풀게 되며, 입력 크기에 따라 `O(N*K)` 시간복잡도를 갖습니다.

- **문자열 편집 및 비교**: 문자열을 대상으로 하는 DP도 빈출합니다. 두 문자열의 유사도를 측정하는 **최소 편집 거리(Edit Distance)** 문제나, 두 문자열의 공통 부분을 찾는 **최장 공통 부분 수열(LCS)** 문제가 유명합니다. 이런 문제들은 2차원 DP 테이블을 채워서 풀게 됩니다. 시간복잡도는 일반적으로 `O(N*M)` (두 문자열 길이가 N, M일 때)입니다. LCS는 예시로 다뤄볼 예정입니다.

- **수열의 최적 부분**: 수열에서 특정 성질을 가진 부분을 찾는 문제들이 있습니다. 가장 대표적인 것이 **가장 긴 증가 부분 수열(LIS)** 문제입니다. DP로 풀면 `O(N^2)`이지만, 이분 탐색 최적화로 `O(N log N)`에 풀 수 있는 유명한 문제입니다. 이 외에도 **연속 부분합 최대 (Kadane 알고리즘)** 문제(예: BOJ 1912 연속합) 등이 있습니다. 이러한 문제들은 1차원 DP로 해결됩니다.

- **그래프 최단 경로**: 그래프 문제도 DP로 접근할 수 있습니다. 특히 **플로이드-워셜 알고리즘**은 DP로 모든 쌍 최단 경로를 구하는 알고리즘입니다. 정점의 개수가 N일 때 `dp[k][i][j]`를 "1..k번 정점까지만 중간 경유지

로 사용해서 i->j 최단거리"로 정의하여 점화식을 세우면, 단계별로 최단거리를 업데이트하여 최종 답을 얻습니다. 이 알고리즘은 `O(N^3)`이라 크기가 크면 힘들지만, **DP를 그래프 문제에 적용한 사례**로 이해해두면 좋습니다. 또한 **벨만-포드 알고리즘**도 DP 시각에서 간선(relaxation)을 반복 적용하는 것으로 볼 수 있습니다.

- **기타**: 이 외에도 **구간 DP(Interval DP)**라고 불리는 유형은 문자열 괄호 제거나 행렬 곱셈 최적화(MCM)처럼 구간을 확장해나가며 푸는 문제들이 있고, **비트마스크 DP**는 뒤에 소개할 것처럼 부분집합을 상태로 가지는 문제들(예: 외판원 순회)이 있습니다. **트리 DP**는 트리 구조에서 부모-자식 간 관계로 DP를 수행하는 문제들이며, **Digit DP**는 숫자의 각 자리별로 상태를 정의해 범위 내 카운팅을 하는 유형입니다 (이들도 곧 간단히 소개하겠습니다).

정리하면, **DP 아키타입**들은 문제의 구조에 따라 비슷한 DP 풀이라는 것입니다. 새로운 문제를 만났을 때, "이건 예전에 풀었던 ○○ 문제와 구조가 비슷하군" 하고 떠올릴 수 있다면 점화식을 세우기가 한결 수월해집니다. 알고리즘 스터디를 통해 다양한 유형의 DP 문제를 경험해 보는 것이 그래서 중요합니다.

예제 문제 – LCS (BOJ 9251)

최장 공통 부분 수열 (Longest Common Subsequence) 문제는 두 문자열이 주어질 때, 두 문자열에 **모두 나타나는 가장 긴 부분 수열**의 길이를 구하는 고전적인 DP 문제입니다. 여기서 "부분 수열"이란 문자의 연속 여부와 상관없이 순서만 유지하면 인정되는 것을 뜻합니다 (연속된 부분은 부분 문자열이라 구별됨). 예를 들어 문자열 "ACAYKP"와 "CAPCAK"이 주어지면, 둘의 LCS는 "ACAK" 등이며 길이는 4입니다.

LCS 문제는 완전 탐색으로 풀기에는 조합 가짓수가 많지만, **DP로 중복 탐색을 피하며** 해결할 수 있습니다. 2차원 DP를 정의하여, `dp[i][j]`를 "첫 번째 문자열의 i번째 문자까지와 두 번째 문자열의 j번째 문자까지 고려했을 때의 LCS 길이"로 잡습니다. 그러면 다음과 같은 점화식이 자연스럽게 나옵니다:

- 만약 두 문자열의 i번째 문자와 j번째 문자가 같다면, 그 문자를 LCS에 추가할 수 있으므로 `dp[i][j] = dp[i-1][j-1] + 1`. (두 문자열에서 그 문자 하나를 나란히 매칭시킨 것)
- 문자가 다르다면, 하나는 버리거나 혹은 다른 하나를 버리는 경우로 나눠 생각하며 최댓값을 취합니다. 즉, `dp[i][j] = max(dp[i-1][j], dp[i][j-1])`. (한쪽 문자를 버리고 이전 상태에서의 최장 부분수열을 유지)

초기 조건으로 `dp[0][j] = 0` (첫 문자열이 0글자일 때는 공통 부분 수열 길이 0), `dp[i][0] = 0` (둘째 문자열 0글자일 때도 0)으로 두고, 1부터 순차적으로 채워나가면 됩니다. 최종 답은 `dp[len(A)][len(B)]`에 해당합니다.

```
# BOJ 9251: LCS (최장 공통 부분 수열)
# 문제: 두 문자열 입력으로 주어질 때, 둘의 LCS(Longest Common Subsequence) 길이를 출력.
# 접근: 2차원 DP. dp[i][j] = 문자열A의 i번째까지와 문자열B의 j번째까지의 LCS 길이.
#       점화식: A[i]==B[j]면 dp[i][j] = dp[i-1][j-1] + 1, 다르면 dp[i][j] = max(dp[i-1][j], dp[i][j-1]).
# 입력: 두 줄에 걸쳐 문자열 A, B.
# 출력: LCS의 길이 (정수).

A = input().strip()
B = input().strip()
n = len(A)
m = len(B)

# DP 테이블 초기화 (0으로 기본 채움)
dp = [[0] * (m + 1) for _ in range(n + 1)]
```

```
for i in range(1, n + 1):
    for j in range(1, m + 1):
        if A[i-1] == B[j-1]:
            dp[i][j] = dp[i-1][j-1] + 1
        else:
            # 윗칸과 왼칸 중 큰 값 가져오기
            if dp[i-1][j] >= dp[i][j-1]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = dp[i][j-1]

print(dp[n][m])
```

위 알고리즘은 입력 문자열 길이를 각각 n, m이라 하면 `O(n*m)` 시간에 실행됩니다. 이중 루프를 돌며, 각 위치에 대해 문자를 비교하고 이전 결과를 참고하여 값을 채우는 **전형적인 2차원 DP**입니다. Greedy로는 접근하기 어렵고, 완전 탐색으로 풀면 지수시간이 걸리는 문제를 DP로 효율화한 사례입니다. LCS 개념은 문자열 처리에서 기본이 되는 만큼 꼭 알아두어야 합니다. (심화로는 LCS 길이뿐만 아니라 **실제 LCS 문자열을 복원**하는 기법도 있는데, 이는 DP 테이블을 추적해서 구현합니다.)

이처럼, DP의 클래식 문제들은 **정형화된 상태와 점화식을 갖고 있으므로 해법이 비교적 잘 알려져 있습니다**. 그렇지만 처음 보는 사람에게는 점화식을 떠올리는 것이 쉽지 않기 때문에, 한 번 직접 구현해 보면서 그 원리를 이해하는 것이 중요합니다. 다양한 문제를 풀다 보면 "아, 이건 전에 푼 ○○ 문제랑 비슷한 형태네"라고 느끼는 순간이 오는데, 그게 바로 실력이 향상된 증거라고 할 수 있습니다.

7. 마무리 및 다음 주 예고

이번 주에는 **동적 계획법(DP)**의 핵심 개념과 여러 가지 응용 예제를 학습했습니다. DP의 등장 배경부터 시작하여, Top-Down과 Bottom-Up 구현 방식의 차이, 점화식을 세우는 방법, 그리고 메모이제이션이나 1차원 배열 활용 등의 최적화 테크닉까지 폭넓게 다루었는데요. 실제 문제 풀이를 통해 확인한 바와 같이, DP는 **복잡한 문제도 작은 부분으로 나눠 풀 수 있게 해주는 강력한 도구**입니다. 덕분에 완전 탐색이 불가능한 큰 문제도 DP를 사용하면 효율적으로 해결할 수 있었고, 그리디로 풀리지 않는 경우에도 전수 조사를 통해 정확한 해답을 얻을 수 있음을 보았습니다.

이번 주 교재에서 풀어본 예제들은 DP의 다양한 면모를 보여주었습니다. 경우의 수를 구하는 조합 문제, 최단/최소를 구하는 최적화 문제, 1차원 수열 문제, 2차원 문자열 문제 등 **여러 가지 상황에서 DP를 적용하는 법**을 연습했습니다. 각 문제마다 DP 상태를 정의하고 점화식을 찾아내는 과정이 조금씩 달랐지만, 공통적으로 "작은 문제 해답이 큰 문제 해답을 만든다"는 DP의 철학이 깔려있습니다. 이것을 항상 염두에 두고, 앞으로 새로운 문제를 만나더라도 **재귀적으로 사고하고, 풀었던 문제 유형들과 비교**하면서 접근해 보세요.

다음 주에는 **그래프 알고리즘**으로 넘어갈 예정입니다. 그래프 알고리즘은 자료구조와 알고리즘의 또 다른 큰 축으로, **BFS/DFS 탐색, 최단 경로 알고리즘(다익스트라, 플로이드-워셜 등), 최소 신장 트리** 등의 주제를 다루게 됩니다. 특히 최단 경로 문제는 이번 주에 언급한 DP 개념과도 연결되며, 그래프만의 특수한 알고리즘들도 등장할 것입니다. 동적 계획법에서 익힌 **체계적 사고와 최적화 개념**은 그래프 문제를 이해하는 데에도 큰 도움이 될 것입니다. 다음 강의에서도 새로운 개념들을 차근차근 배우고, 실전 문제로 연습하며 실력을 키워나갈 테니 기대해주시기 바랍니다!

마지막으로, 이번 주 학습한 내용은 꼭 복습하시고, 제공된 예제 코드들을 직접 실행하거나 변형해 보면서 내 것으로 만들어보세요. 알고리즘은 **직접 구현하고 다양한 입력으로 테스트해볼 때 비로소 이해가 깊어집니다.** 어려운 점이 있었다면 스터디 동료들과 질의응답을 통해 해결하고 넘어가시길 권장합니다. 그럼 다음 주 그래프 알고리즘 강의에서 다시 만나겠습니다. 고생하셨습니다!

8. 확장 유형 소개 (문제 풀이 없이 유형 설명 위주)

동적 계획법은 기본 유형들 외에도 알고리즘 대회나 고급 문제에서 등장하는 **확장된 형태의 DP 기법**들이 있습니다. 이번 주에 모두 다루지는 않았지만, 알아두면 좋을 몇 가지 DP 유형들을 소개하며 마무리하겠습니다. 각 유형의 핵심 아이디어와 예시 문제 번호를 함께 제시하니, 관심 있는 분들은 도전해 보세요.

8.1 비트마스크 DP

비트마스크 DP는 DP의 상태를 이진수로 표현된 **부분집합(bitmask)**으로 나타내는 기법입니다. 주로 N개 요소의 부분집합을 선택하거나 순서를 결정하는 문제에서 사용되며, 상태공간 크기는 2^N에 비례합니다. 대표적인 예가 **외판원 순회 문제(TSP)**입니다. TSP에서는 각 도시의 방문 여부를 비트마스크로 표현하여, `dp[mask][i]`를 "방문집합 mask 상태에서 현재 도시 i에 있을 때의 최소 비용"으로 정의하고 풀 수 있습니다. 일반적으로 비트마스크 DP는 N이 20~25 이하인 경우에 적용하며, 그 이상이면 2^N이 너무 커져서 힘듭니다. 그래도 NP-완전 문제들에 DP로 접근하는 유용한 사례가 많습니다.
- **예시 문제**: 백준 2098 **외판원 순회** (N개 도시를 모두 방문하는 최소 경로 찾기)

8.2 트리 DP

트리 DP는 그래프 중에서도 **트리 구조**에서 활용되는 DP입니다. 트리는 사이클이 없기 때문에, 한 루트에서 시작하여 **DFS(깊이우선탐색)**로 내려가며 서브트리의 DP 값을 계산하고, 다시 올라오면서 값을 결합하는 방식으로 많이 구현됩니다. 트리 노드마다 DP값을 정하는데, 부모-자식 관계로부터 점화식을 세우는 것이 특징입니다. 예를 들어 **트리의 독립집합** 문제에서는 `dp[node][0/1]` 등으로 노드를 선택하거나 안 했을 때 최대 값 등을 계산합니다. 또 **사회망 서비스(SNS)** 문제(백준 2533)처럼, 트리에서 얼리 어답터를 최소로 선택하는 문제도 트리 DP로 풀립니다. 트리 DP의 핵심은 **자식들의 값들을 모아 부모의 값을 결정**하는 구조를 잡는 것입니다.
- **예시 문제**: 백준 2533 **사회망 서비스(SNS)** (트리에서 최소 얼리 어답터 찾기), 백준 1949 **우수 마을** (트리에서 최대 인구 합 선택 문제)

8.3 Digit DP (자릿수 DP)

Digit DP는 숫자를 자리별로 탐색하는 DP 기법으로, 0부터 어떤 큰 숫자 N까지의 범위에서 특정 조건을 만족하는 수의 개수 등을 구하는 데 자주 쓰입니다. 예를 들어, **0~N 사이의 숫자 중 특정 조건(특정 숫자의 개수가 k개 이하 등)을 만족하는 개수를 세는 문제**들이 있습니다. Digit DP에서는 일반적으로 자리 위치, 현재까지의 속성(예: 합이나 특정 숫자 등장 횟수), 그리고 아직 제한 N과 같냐/작냐를 나타내는 플래그 등을 상태로 사용합니다. `dp[pos][state][tight]` 식으로 정의하고, 가장 높은 자리부터 한 자리씩 내려오며 DP를 수행합니다. 이 방법을 쓰면 자릿수가 최대 100자리인 큰 수 범위도 다룰 수 있습니다. 구현이 까다롭지만 알고리즘 문제에서 종종 출제되는 테크닉입니다.
- **예시 문제**: 백준 1562 **계단 수** (자릿수 DP + 비트마스크: 0~9 모든 숫자 사용되는 계단 수 개수 구하기), 기타 숫자 개수 세기 관련 문제들

8.4 고급 DP 최적화 기법

마지막으로, DP 점화식의 구조를 활용한 고급 최적화 기법들이 있습니다. 예를 들어 **Convex Hull Trick**은 DP 점화식이 `dp[i] = \min_j (dp[j] + m*j + b)` 꼴로 떨어질 때 여러 후보 직선들의 최소값을 빠르게 구하기 위한 방법이고, **Divide and Conquer Optimization**은 특정 DP (`dp[i][j] = \min_{k<j} (dp[i-1][k] + cost(k,j))`)의 최적 분할점이 앞 단계의 최적 분할점과 정렬되는 경우 적용하여 `O(N*M*log M)` 등을 `O(N*M)`으로 줄이는 테크닉입니다. 또 **Monotonic Queue Optimization**은 DP 계산 중 슬라이딩 윈도우 내 최솟값/최댓값을 빠르게 얻는 경우에 쓰입니다. 이러한 기법들은 매우 특수한 상황에만 적용되지만, 알고리즘 대회 고득점 문제에서 시간을 줄이는 열쇠가 되기도 합니다. - (이 부분은 개념 소개만 하고, 문제 번호는 생략합니다. 필요하다면 추후 심화 시간에 다룰 예정입니다.)

이런 확장 유형들은 **DP의 응용 범위가 얼마나 넓은지**를 보여줍니다. 처음에는 생소할 수 있지만, 이제 기본적인 DP에 익숙해졌다면 차츰 이런 고급 주제들도 접해보길 권합니다. 물론 한 번에 다 이해하려 하기보다는, 우선은 이번 주 학습한 **기본 원리와 패턴**을 충분히 자기 것으로 만드는 것이 우선입니다. 그러고 나서 여유가 된다면 위에 소개한 유형 별 예시 문제에 도전해보세요. 분명 또 다른 알고리즘적 통찰을 얻을 수 있을 것입니다. Happy Coding!
