그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 06

# 동적 계획법

Dynamic Programming

한 번 푼 문제는 두 번 풀지 않는다

CONTENTS

# 이번 장에서 배울 내용


6.1 동적 계획법이란?
기억하고, 재활용하라

6.2 피보나치 수열
탑다운과 바텀업

6.3 DP 성립 조건
언제 표를 그려야 하는가

6.4 바닥 공사
2×N 타일 채우기

6.5 1로 만들기
그리디가 지고 DP가 이긴다

6.6 0/1 배낭 문제
2차원 표로 되갚는다

6.7 LCS 알고리즘
문자열을 표로 비교한다

6.8 코딩 테스트
개미 전사 · 화성 로봇


5장에서 그리디가 놓쳤던 0/1 배낭 문제를, 이 장에서 표를 그려 정확히 되찾는다.

6.1

# 동적 계획법이란?

"과거를 기억하지 못하는 자는
그 과거를 반복하는 저주에 걸린다." — George Santayana

# 이름과 달리, 정체는 꼼꼼한 기록가

알고리즘에서 가장 있어 보이는 이름을 꼽으라면 단연 동적 계획법일 것이다. 그런데 이 기법의 본질은 "동적"인 것과 아무 상관이 없고, 우리가 아는 코딩과도 거리가 멀다. 오히려 정체는 기억력이 아주 좋은 꼼꼼한 기록가에 가깝다.

> 한 번 계산한 답을 적어 두고, 같은 질문이 오면 다시 계산하지 않고 꺼내 준다.

**나눈다 → 기억한다 → 재활용한다**

# 동적 계획법의 탄생

리처드 벨만은 연구가 실용적으로 보이도록 하려고, 다분히 정치적인 이유에서 "dynamic programming"이라는 이름을 골랐다. 그는 군사 작전과 자원 배분에서 최적의 결정을 찾기 위해 연속적인 의사결정 과정을 체계적으로 분석했고, 그 과정에서 문제를 작은 부분 문제로 나누고 결과를 저장해 중복 계산을 피하는 방법을 제안했다.

### 1950년대 국방부 연구에서 출발

연속적 의사결정을 다루는 수학적 틀로 고안되었다.

### 경로 최적화와 데이터 압축

최단 경로 문제를 비롯한 최적화 전반의 표준 도구가 되었다.

### 오늘날에는 AI와 강화학습까지

유전자 배열 분석, 재무·투자 전략 최적화에도 그대로 쓰인다.

# 동적 계획법 vs 분할 정복

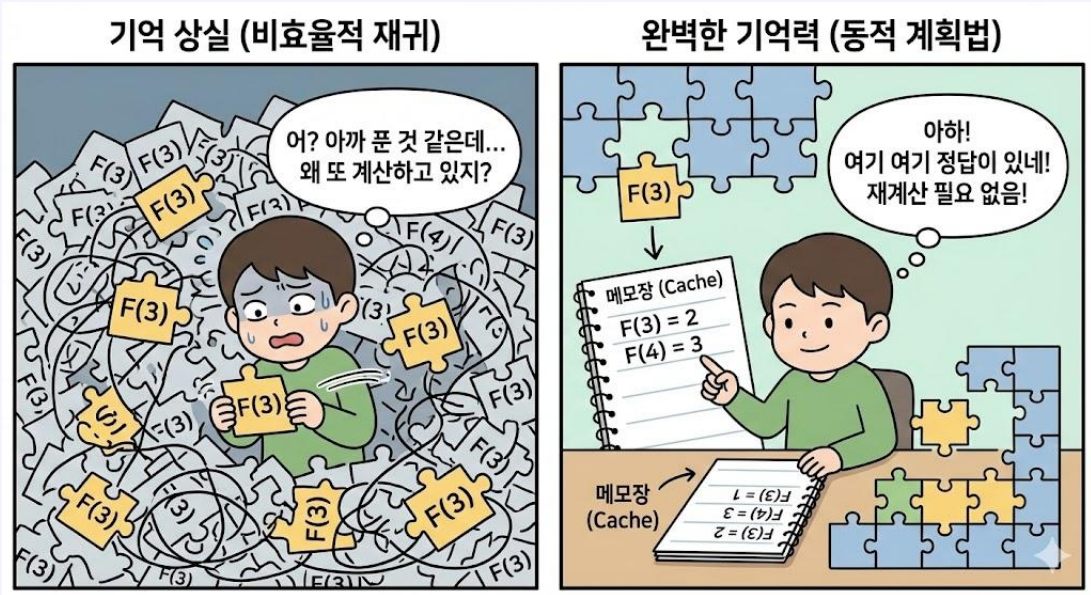
"큰 문제를 작은 문제로 나눈다고요? 그건 합병 정렬 같은 분할 정복이랑 똑같은 것 아닌가요?" 둘은 분명 형제지간이다. 하지만 결정적인 차이가 하나 있다 — 부분 문제가 겹치는가, 그리고 그것을 기억하는가.

### 분할 정복 — 겹치지 않는다

쪼개진 문제들이 서로 독립적이다. 왼쪽 절반 정렬과 오른쪽 절반 정렬은 아무 상관이 없다.

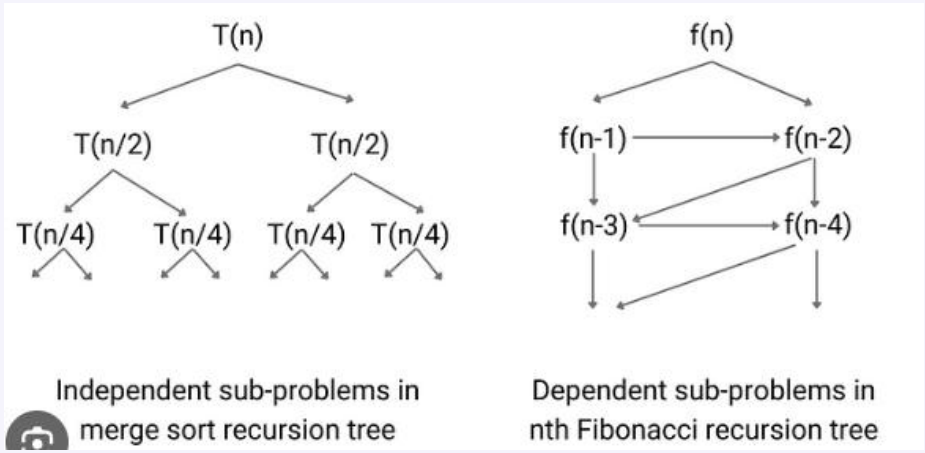
### 동적 계획법 — 계속 겹친다

F(5)를 구하려면 F(4)와 F(3)이 필요한데, F(4)를 구할 때 F(3)이 또 필요하다.



# 부분 문제가 독립인가, 겹치는가

(a) 합병 정렬에서의 부분 문제는 서로 독립적이다. (b) 피보나치 수열에서의 부분 문제는 독립적이지 않다.




**독립적** 겹치는 조각이 없다면 저장할 이유도 없다. 그냥 재귀로 쪼개서 각각 풀면 된다 — 분할 정복.

**중복적** 같은 조각이 반복해서 나온다면 한 번만 풀고 적어 두는 편이 압도적으로 싸다 — 동적 계획법.

# 한눈에 비교하기

| 개념 | 동적 계획법(DP) | 분할 정복(Divide and Conquer) |
|---|---|---|
| 기본 원리 | 작은 문제를 반복적으로 해결하며 중복 계산을 저장하여 최적화 | 문제를 독립적인 작은 문제로 분할하고 각각 해결 후 합침 |
| 부분 문제의 특성 | 중복된 부분 문제(overlapping subproblems)가 존재 | 독립적인 부분 문제(independent subproblems)로 나눌 수 있음 |
| 적용 방식 | 메모이제이션(탑다운) 또는 테이블화(바텀업) 방식 사용 | 재귀적으로 문제를 쪼개서 해결하고 합침 |
| 시간 복잡도 | 중복 계산을 제거하여 효율적으로 계산 가능 | 일반적으로 재귀 깊이에 따라 비효율적일 수 있음 |

| 구분 | 분할 정복 | 동적 계획법 |
|---|---|---|
| **부분 문제** | 서로 독립적 | 서로 중복됨 |
| **결과 저장** | 하지 않는다 | **표에 저장한다** |
| **대표 예** | 합병 정렬, 퀵 정렬 | 피보나치, 배낭 문제 |
| **핵심 무기** | 재귀 | 재귀 + 메모리 |

결국 차이는 "기억하느냐"에 있다. 저장할 표가 있으면 DP, 없으면 분할 정복이다.

# DP의 핵심 철학 — 기억하고, 재활용하라

DP의 핵심은 "두 번 계산하지 않는다"는 효율성에 있다. 이것을 가능하게 하는 장치가 메모이제이션이다.

### 나눈다 (split)

큰 문제를 작은 문제로 쪼갠다. 곧 점화식을 세우는 일이다.

### 기억한다 (memoize)

작은 문제의 정답을 배열이나 테이블에 적어 둔다.

### 재활용한다 (reuse)

같은 문제가 다시 나오면 계산하지 않고 표에서 꺼내 쓴다.

점화식을 세우는 순간 문제의 절반은 풀린다 — 나머지 절반은 표를 채우는 순서를 정하는 일이다.

dp_skeleton.py

```
# DP의 골격 — 어떤 문제든 이 세 줄로 환원된다
dp = [초기값] * (n + 1)          # ② 기억할 표를 만든다
dp[기저] = 이미 아는 답           # ③ 바닥을 깐다
for i in 작은 것부터 큰 것까지:
    dp[i] = 점화식(dp[i-1], dp[i-2], ...)   # ① 나누고 ③ 재활용
```

6.2

# 피보나치 수열

6.2

DP를 이해하는 데 이보다 완벽한 예제는 없다.

$F(n) = F(n-1) + F(n-2)$

# 피보나치 수열 — 순진한 재귀의 함정

토끼 번식 문제에서 유래한 피보나치 수열은 점화식이 그대로 코드가 되는 가장 단순한 예다. 정의를 그대로 옮기면 두 줄이면 끝난다.

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2} \qquad (n \geq 2)
\end{aligned}
$$

fibo_naive.py

```python
"""순진한 재귀 방식으로 피보나치 수 구하기"""
def fibo_naive(n):
    if n <= 1:                    # 기저 조건 (base case)
        return n
    return fibo_naive(n - 1) + fibo_naive(n - 2)

print(fibo_naive(10))   # 55  — 금방 나온다
print(fibo_naive(50))   # ... — 끝나지 않는다
```

**n = 10 이면 즉시**

호출이 백여 번이면 끝난다. 아무 문제가 없어 보인다.

**n = 50 이면 멈추지 않는다**

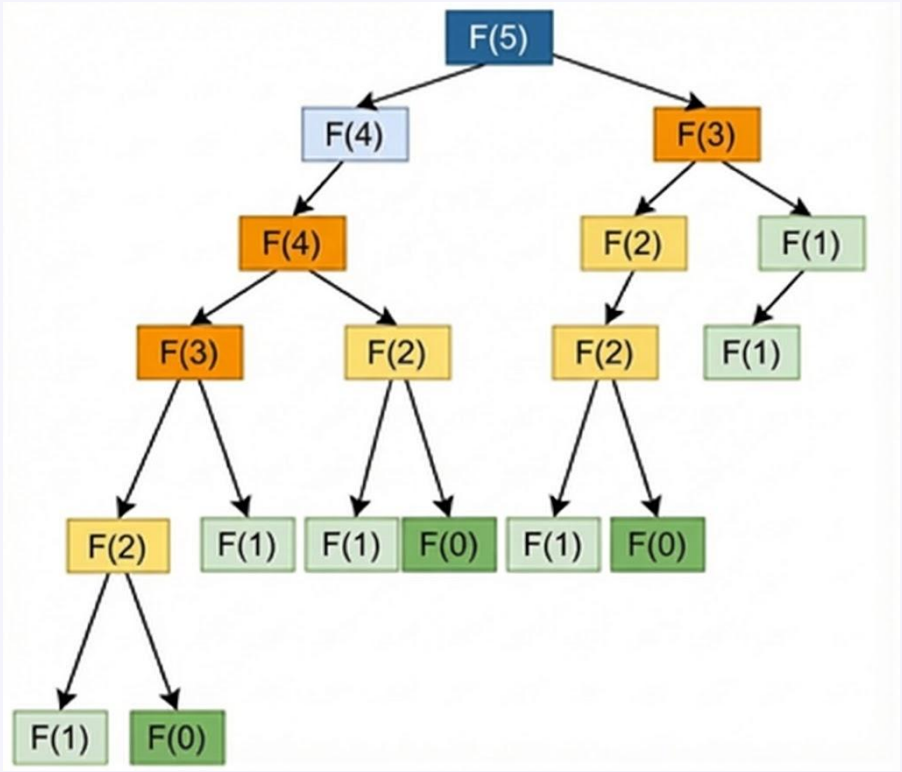
호출 횟수가 폭발한다. 코드는 옳은데 실행이 끝나지 않는다.

**왜 이렇게 느린가**

틀린 코드가 아니다. 같은 값을 몇 번이고 다시 계산하고 있을 뿐이다.

점화식이 옳다고 구현이 실용적인 것은 아니다 — 무엇이 낭비되고 있는지 들여다보자.

# 비극의 원인 — "아까 푼 문제잖아, 왜 또 계산해?"




### 같은 가지가 계속 자란다

F(5)를 구하려고 F(3)을 두 번, F(2)를 세 번, F(1)을 다섯 번 계산한다. n이 커지면 이 낭비가 지수적으로 불어난다.

$$T(n) \approx O(2^n)$$

### 해법은 놀랄 만큼 단순하다

한 번 구한 값을 어딘가에 적어 두기만 하면 된다. 그러면 같은 가지를 두 번 타고 내려갈 이유가 사라지고, 계산해야 할 서로 다른 부분 문제는 n개뿐이 된다.

지수 시간을 선형 시간으로 바꾸는 데 필요한 것은 알고리즘의 교체가 아니라 표 하나다.

# 방법 ① 탑다운 (top-down) + 메모이제이션

큰 문제 F(n)에서 출발해 아래로 내려가며 작은 문제를 호출한다. 기존 재귀와 방향이 같되, 계산한 값을 메모장에 적어 둔다는 점만 다르다.

fibo_topdown.py

```python
# 한 번 계산한 값을 적어 둘 메모장 (딕셔너리)
memo = {0: 0, 1: 1}

"""탑다운(메모이제이션) 방식으로 피보나치 수 구하기"""
def fibo_topdown(n):

    # 1. 메모장에 이미 답이 있는지 확인 (가장 중요!)
    if n in memo:
        return memo[n]        # 계산하지 않고 즉시 반환

    # 2. 없다면 계산하고, 결과를 즉시 기록
    memo[n] = fibo_topdown(n - 1) + fibo_topdown(n - 2)

    # 3. 계산된 값 반환
    return memo[n]

print(fibo_topdown(50))   # 12586269025 (순식간!)
```

**먼저 메모장부터 본다**

이 한 줄이 지수 시간을 선형 시간으로 바꾼다.

**계산 즉시 기록**

반환하기 전에 반드시 적어 둔다. 빠뜨리면 효과가 사라진다.

**재귀 형태를 유지한다**

점화식을 그대로 옮긴 모양이라 읽기 쉽다.

**재귀 깊이에 주의**

n이 크면 호출 스택이 넘칠 수 있다. 파이썬은 특히 그렇다.

# 방법 ② 바텀업 (bottom-up) + 타뷸레이션

가장 작은 기초 문제 F(0), F(1)부터 시작해 표를 바닥부터 순서대로 채워 올라간다. 재귀 호출이 아예 없다.

fibo_bottomup.py

```python
"""바텀업(타뷸레이션) 방식으로 피보나치 수 구하기"""
def fibo_bottomup(n):
    if n <= 1:
        return n

    # 1. DP 테이블 생성 (크기 n+1의 리스트)
    dp = [0] * (n + 1)

    # 2. 기저 상태 초기화 (바닥 공사)
    dp[0] = 0
    dp[1] = 1

    # 3. 작은 문제부터 차례대로 표 채우기
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    # 4. 표의 마지막 칸에 정답이 있다
    return dp[n]

print(fibo_bottomup(50))   # 12586269025 (역시 순식간!)
```

**표를 먼저 만든다**

dp[i]에 F(i)가 들어간다는 정의를 먼저 확정한다.

**바닥을 깐다**

dp[0], dp[1]은 계산이 아니라 이미 아는 값이다.

**반복문으로 채운다**

작은 인덱스부터 채우므로 필요한 값은 항상 이미 준비되어 있다.

**스택 걱정이 없다**

재귀가 없으니 깊이 제한에서 자유롭다.

# 무엇을 선택해야 할까?

| 특징 | 탑다운(top-down) + 메모이제이션 | 바텀업(bottom-up) + 타뷸레이션 |
|---|---|---|
| 구현 방식 | 재귀(recursion) | 반복문(iteration) |
| 방향 | 큰 문제 → 작은 문제(쪼개기) | 작은 문제 → 큰 문제(쌓기) |
| 저장소 이름 | 주로 '메모(memo)'라고 부름 | 주로 '테이블(table)'이라고 부름 |
| 직관성 | 점화식을 그대로 코드로 옮기기 쉽다. (수학적 정의와 유사) | 문제 해결 순서가 명확하고 직관적이다. (1단계, 2단계...) |
| 효율성 | 필요한 부분 문제만 계산한다. (피보나치는 다 필요해서 해당 없음) | 재귀 호출의 오버헤드(함수 호출 비용, 스택 메모리)가 없다. |
| 단점 | 재귀 깊이 문제: n이 너무 크면 스택 오버플로 에러가 발생할 수 있다. | 모든 부분 문제를 다 채워야 한다. (때로는 불필요한 계산을 할 수도 있음) |
| 주 사용처 | 문제가 직관적이지 않고 복잡할 때, 우선 점화식을 세우고 구현해볼 때 유용 | 성능이 중요하거나 n이 매우 클 때, 안정적인 구현을 위해 선호됨 |

| 기준 | 탑다운 | 바텀업 |
|---|---|---|
| **방향** | 큰 문제 → 작은 문제 | 작은 문제 → 큰 문제 |
| **구현** | 재귀 + 메모장 | 반복문 + 테이블 |
| **장점** | 점화식과 모양이 같다 | 빠르고 스택이 안전하다 |
| **단점** | 호출 스택 부담 | 필요 없는 칸도 채운다 |

둘은 같은 점화식을 다른 순서로 계산할 뿐이다. 결과는 언제나 같다.

**실무 기준은 단순하다**

점화식이 복잡하고 필요한 상태만 골라 계산하고 싶다면 탑다운, 표 전체를 어차피 채워야 하고 속도가 중요하다면 바텀업을 고른다.

이 장의 나머지 예제는 모두 바텀업으로 푼다 — 표가 채워지는 과정이 눈에 보이기 때문이다.

# 무엇이 달라졌는가 — 복잡도 비교

| | | |
|---|---|---|
| **순진한 재귀** | 서로 다른 부분 문제가 n개뿐인데도 호출은 지수적으로 늘어난다. n = 50에서 사실상 멈춘다. | $O(2^n)$ |
| **탑다운** | 각 부분 문제를 정확히 한 번만 계산한다. 메모장 조회는 상수 시간이다. | $O(n)$ |
| **바텀업** | 표를 한 번 훑으며 채운다. 호출 비용조차 없어 상수 계수가 가장 작다. | $O(n)$ |

$$O(2^n) \Rightarrow O(n)$$

알고리즘을 바꾼 것이 아니라, 계산한 결과를 버리지 않았을 뿐이다.

6.3

# DP가 성립하는 조건

"이 문제, 정말 DP로 풀 수 있는 문제인가?"
코드를 짜기 전에 반드시 던져야 할 질문이다.

6.3

§6.3 DP 성립 조건

# DP가 성립하기 위한 두 가지 조건

피보나치로 DP의 맛을 봤으니 실전으로 나아갈 차례다. 다만 모든 문제에 DP를 적용할 수는 없다. 이 강력한 도구를 쓰려면 문제가 두 가지 까다로운 조건을 만족해야 한다.

## ① 최적 부분 구조

optimal substructure

작은 문제들의 정답을 합치면 큰 문제의 정답이 되는가? 답을 모듈처럼 조립할 수 있어야 한다.

## ② 중복되는 부분 문제

overlapping subproblems

똑같은 작은 문제가 토씨 하나 안 틀리고 반복해서 등장하는가? 겹치지 않으면 저장할 이유가 없다.

**①은 정확성, ②는 효율성의 조건이다**

①이 없으면 표를 채워도 답이 틀리고, ②가 없으면 답은 맞지만 굳이 표를 만들 이유가 없다. 둘 다 있어야 DP가 의미를 갖는다.

이 두 질문에 모두 "예"라면, 주저 없이 DP 테이블을 그려라.

# ① 최적 부분 구조 (optimal substructure)

"작은 문제들의 최적해를 합치면 큰 문제의 최적해가 되는가?" 이 속성은 문제의 정답을 구하는 방식이 모듈식 조립으로 가능한지를 묻는 것이다. 즉 전체 문제의 최적해가 부분 문제들의 최적해로 구성되어야 한다.

부분의 최선을 그대로 이어 붙였을 때 전체의 최선이 되어야 한다.

**5장에서 이미 만난 성질**

그리디도 같은 조건을 요구했다. DP는 여기에 "중복"이라는 조건 하나를 더 얹은 것이다.


**[1단계: 작은 문제들의 '최적해(가장 좋은 부품)'를 각각 구한다!]**

**[2단계: 그것들을 합치면 자연스럽게 '전체의 최적해(최강 로봇)'가 완성된다!]**

# 성립하는 예 — 최단 경로

서울에서 부산으로 가는 가장 빠른 길을 찾는다고 하자. 그 경로가 반드시 대전을 거친다면, 서울→대전 구간 역시 그 구간의 최단 경로여야 한다. 더 빠른 샛길이 있었다면 전체 경로도 그쪽으로 바뀌었을 테니까.


**Shortest(서울→부산)**

= Shortest(서울→대전) + Shortest(대전→부산)

**작은 구간의 최선이 전체의 최선**

구간을 어디서 끊어도 같은 논리가 성립한다. 그래서 최단 경로 문제는 DP로도, 다익스트라 같은 그리디로도 풀린다.

"부분을 최선으로 만들면 전체도 최선이 된다" — 이것이 성립할 때 표를 채우는 계산이 정당해진다.

# 성립하지 않는 예 — 최장 경로

이번에는 같은 그래프에서 가장 긴 경로를 찾되, 한 번 방문한 도시는 다시 갈 수 없다는 조건을 붙인다. 그러면 부분의 최선이 전체의 최선을 보장하지 않는다.


**Longest(서울→부산)**

≠ Longest(서울→대전) + Longest(대전→부산)

**부분을 늘리면 전체가 막힌다**

서울→대전 구간을 최대한 길게 돌아가면 이미 지나온 도시 때문에 대전→부산 구간이 끊긴다. 조각의 최선이 전체를 망가뜨린다.

최적 부분 구조가 깨지면 DP 테이블을 아무리 정확히 채워도 답이 틀린다. 조건 확인이 먼저다.

# ② 중복되는 부분 문제 (overlapping subproblems)

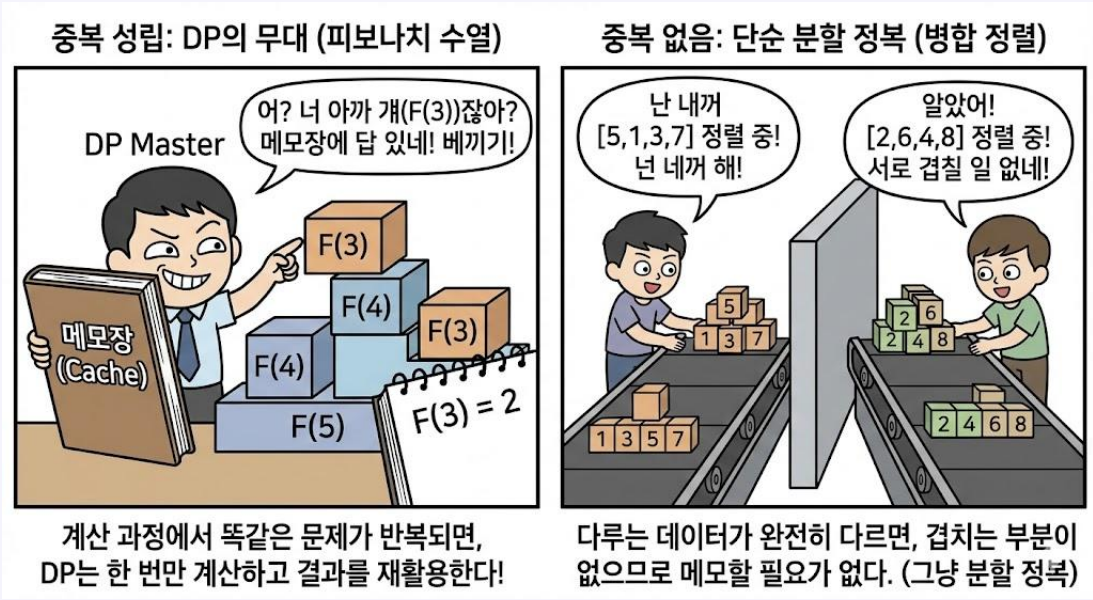
"똑같은 작은 문제가 토씨 하나 안 틀리고 반복해서 등장하는가?" 이 속성은 DP의 효율성, 곧 가성비와 관련이 있다. 문제를 쪼갰는데 겹치는 조각이 하나도 없다면 굳이 답을 저장할 이유가 없다.

### 겹친다 → 저장이 이득

피보나치처럼 같은 값이 수없이 반복되면 저장 한 번이 지수적 절약이 된다.

### 안 겹친다 → 저장은 낭비

합병 정렬처럼 조각이 독립적이면 표를 만들어 봐야 메모리만 쓴다.



# DP 진단 체크리스트

어떤 문제를 만났을 때 다음 두 질문에 모두 "예"라고 답할 수 있다면, 주저 없이 DP 테이블을 그려도 좋다.

| 조건 | 의미 | 확인 방법(자문자답) |
|---|---|---|
| 1. 최적 부분 구조 | 구조적 조건 | "점화식을 세울 수 있는가?"<br>(n번 째 답을 n-1번 째 답을 이용해 구할 수 있는가?) |
| 2. 중복 부분 문제 | 효율성 조건 | "같은 부분 문제가 여러 번 등장하는가?"<br>"메모이제이션이나 테이블이 성능 향상에 도움이 되는가?" |

**질문 ①** 작은 문제의 최적해를 조합해서 큰 문제의 최적해를 만들 수 있는가? **정확성**

**질문 ②** 같은 부분 문제가 여러 번 반복해서 등장하는가? **효율성**

두 질문을 통과했다면 다음 할 일은 하나 — dp[i]가 무엇을 뜻하는지부터 문장으로 적는 것이다.

6.4

# 바닥 공사 (1차원 DP)

6.4

가로 N, 세로 2인 바닥을 타일로 빈틈없이 채우는
방법의 수는 몇 가지일까?

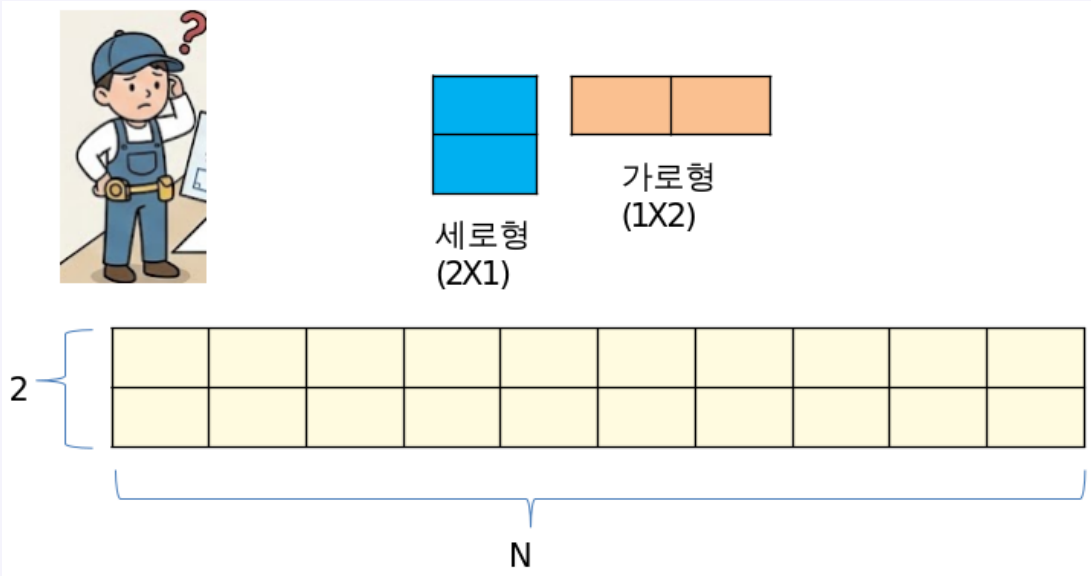
# 바닥 공사 문제

인테리어 업자가 되어 가로 N, 세로 2인 직사각형 바닥을 타일로 빈틈없이 채워야 한다. 쓸 수 있는 타일은 2×1 세로형과 1×2 가로형 두 종류뿐이다.

d[i]를 "2×i 바닥을 채우는 방법의 수"로 정의한다. 우리가 원하는 답은 d[N]이다.

**DP의 첫 단추는 정의 문장**

코드를 짜기 전에 dp 배열이 무엇을 뜻하는지 한 문장으로 적어야 한다. 이 문장이 정확하면 점화식은 대개 저절로 따라 나온다.

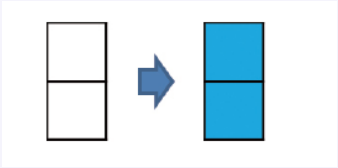


# 작은 문제부터 직접 세어 보기

점화식이 보이지 않을 때는 손으로 몇 개만 세어 보면 된다. 규칙은 대개 세 번째 항에서 드러난다.

**N = 1**

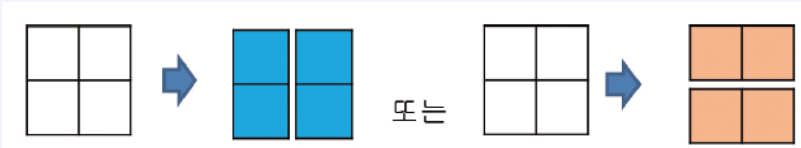
**1가지**



2×1 바닥에는 세로 타일 하나를 놓는 방법밖에 없다.

**N = 2**

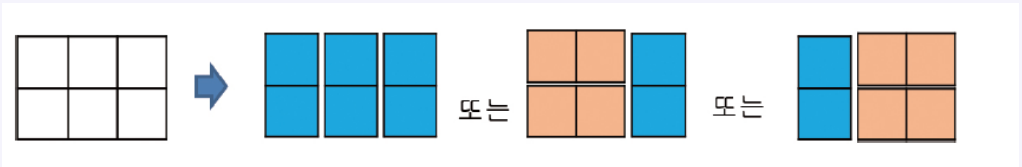
**2가지**




세로 타일 두 개를 나란히 놓거나, 가로 타일 두 개를 위아래로 쌓는다.

**N = 3**

**3가지**




직접 그려 보면 세 가지가 나온다. 1, 2, 3… 규칙이 보일 듯 말 듯하다.

1, 2, 3 … 그다음은 5일까? 우연처럼 보이는 숫자를 점화식으로 확정해야 한다.

§6.4 바닥 공사

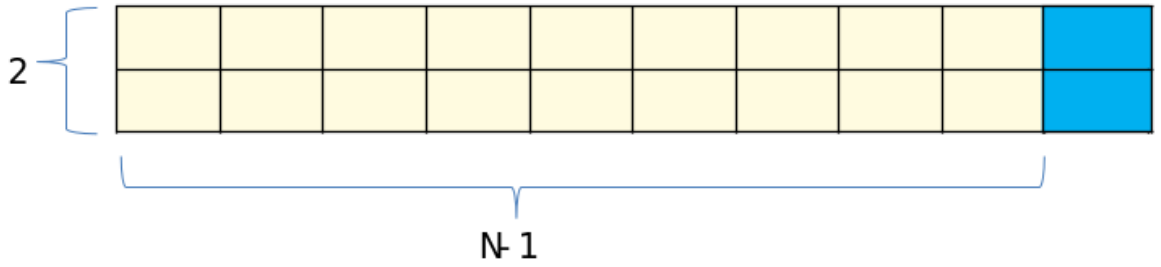
# "마지막 단계"를 기준으로 점화식 유도하기

DP 점화식을 세우는 가장 확실한 방법은 마지막 한 수를 기준으로 경우를 나누는 것이다. 2×N 바닥의 오른쪽 끝에 무엇을 놓았는지만 따지면 된다.

### ① 마지막에 세로 타일 1개

오른쪽 끝에 세로 타일 하나를 세운다. 남은 바닥의 가로 길이는 N-1이 된다.

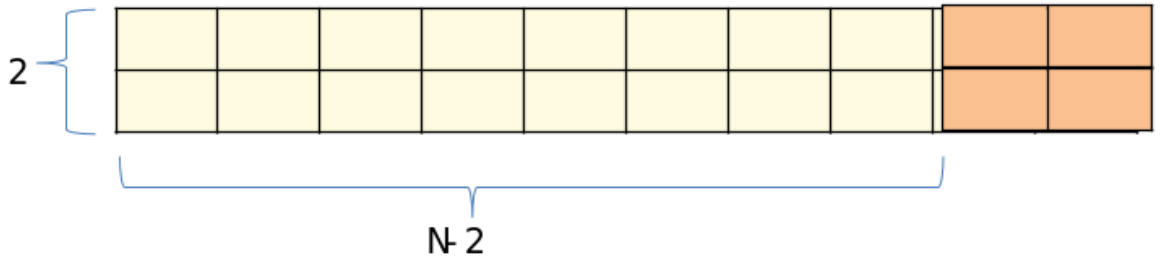
`D[N-1]`




### ② 마지막에 가로 타일 2개

오른쪽 끝에 가로 타일 두 개를 위아래로 쌓는다. 남은 가로 길이는 N-2가 된다.

`D[N-2]`




오른쪽 끝에 놓을 수 있는 경우는 이 둘뿐이고, 두 경우는 서로 겹치지 않는다.

§6.4 바닥 공사

# 점화식 도출

N번째 바닥을 채우는 방법의 수는 앞의 두 경우를 그대로 합친 것과 같다.

$$D[N] = D[N-1] + D[N-2]$$

마지막에 세로를 놓는 경우 + 마지막에 가로를 놓는 경우

### 어디서 많이 본 식

피보나치 수열의 점화식과 완전히 같다. 겉포장만 다를 뿐 알맹이가 같은 문제다.

### 기저는 손으로 센 값

d[1] = 1, d[2] = 2 는 계산이 아니라 직접 세어 확정한 값이다.

### 표를 한 번만 훑는다

작은 i부터 채우면 필요한 값은 항상 이미 표에 들어 있다. O(N)이다.

문제의 외피가 달라도 점화식이 같으면 같은 코드가 된다 — DP를 배우는 진짜 이유다.

# 표가 채워지는 과정


**d[1] = 1, d[2] = 2**

바닥 두 칸을 먼저 확정한다. 여기서부터 쌓아 올린다.

**d[3] = d[2] + d[1] = 3**

앞의 두 칸만 보면 된다. 다시 세어 볼 필요가 없다.

**d[4] = d[3] + d[2] = 5**

이미 채운 값을 재활용하므로 계산은 덧셈 한 번뿐이다.

**마지막 칸이 정답**

표의 오른쪽 끝 d[N]을 읽으면 그것이 구하던 답이다.

# 파이썬 코드

tile.py

```python
def tile_construction(n):
    if n == 1:
        return 1              # 인덱스 오류 방지용 안전장치

    # DP 테이블 초기화 (인덱스는 0부터 n까지)
    d = [0] * (n + 1)

    # 기저 상태: 손으로 세어 확정한 값
    d[1] = 1                  # 2x1 바닥: 세로 1개
    d[2] = 2                  # 2x2 바닥: 세로 2개 또는 가로 2개

    # 점화식으로 작은 것부터 표 채우기
    for i in range(3, n + 1):
        d[i] = d[i-1] + d[i-2]

    return d[n]

# 실행 예시: 가로 길이가 4일 때
print(tile_construction(4))
```

OUTPUT

```
5
```

### d의 뜻을 먼저 적는다

d[i] = 2×i 바닥을 채우는 방법의 수. 이 한 문장이 코드의 설계도다.

### 인덱스는 1부터 쓴다

d[0]을 비워 두면 i와 바닥 길이가 그대로 일치해 헷갈리지 않는다.

### 메모리는 O(N)

사실 직전 두 값만 있으면 되므로 변수 두 개로 줄일 수도 있다.

### 시간 복잡도 O(N)

반복문 한 겹이 전부다. N이 100만이어도 한순간이다.

6.5

# 1로 만들기와 계단 오르기

6.5

그리디가 왜 항상 통하지 않는지,
그리고 왜 DP가 필요한지를 보여 주는 예제.

÷ §6.5 1로 만들기

# 1로 만들기

정수 X가 주어졌을 때 다음 네 가지 연산을 적절히 사용해 값을 1로 만들려고 한다. 연산을 사용하는 횟수의 최솟값을 구하라.

| | | |
|---|---|---|
| ÷ **연산 ①** | X가 5로 나누어떨어지면 5로 나눈다. | `X / 5` |
| ÷ **연산 ②** | X가 3으로 나누어떨어지면 3으로 나눈다. | `X / 3` |
| ÷ **연산 ③** | X가 2로 나누어떨어지면 2로 나눈다. | `X / 2` |
| − **연산 ④** | X에서 1을 뺀다. 언제나 사용할 수 있는 연산이다. | `X - 1` |

? "많이 줄어드는 연산부터 쓰면 되지 않을까?" — 이 직관이 어디서 무너지는지 확인해 보자.

§6.5 1로 만들기

# 그리디의 유혹과 실패 — X = 26

본능은 이렇게 속삭인다. "빨리 줄이려면 나누는 수가 클수록 좋지! 5로, 안 되면 3으로, 그것도 안 되면 2로." 하지만 이 전략은 처참하게 실패한다

### 그리디 전략 — 5회

26 → 13 (5·3으로 안 나눠짐, 2로 나눔)

13 → 12 (소수이므로 1을 뺌)

12 → 4 (3으로 나눔)

4 → 2 (2로 나눔)

2 → 1 (2로 나눔)

### 최적 전략 — 3회

26 → 25 (먼저 1을 뺀다 – 손해처럼 보이지만)

25 → 5 (5로 나눈다 – 대박 찬스!)

5 → 1 (다시 5로 나눈다)

**연산 3회면 충분하다**

**눈앞의 이득을 포기해야 전체가 최선이 된다**

26에서 1을 빼는 것은 당장은 손해다. 그리디는 이 한 수를 절대 두지 못한다. 모든 경우를 표에 적어 두고 비교해야 비로소 보인다.

÷ §6.5 1로 만들기

# DP 전략 — 직전 단계를 모두 따진다

i를 1로 만드는 최소 횟수를 D[i]라 하자. i에 도달하기 직전 단계로 가능한 것은 네 가지뿐이므로, 그 넷 중 가장 작은 값에 연산 한 번을 더하면 된다.

**1을 뺐다면** 직전 단계는 i-1이었다. 총 횟수는 D[i-1]에 뺄셈 1회를 더한 값이다. `D[i-1] + 1`

**2로 나눴다면** i가 2의 배수일 때만 가능하다. 직전 단계는 i/2였다. `D[i//2] + 1`

**3으로 나눴다면** i가 3의 배수일 때만 가능하다. 직전 단계는 i/3이었다. `D[i//3] + 1`

$$D[i] = \min(\, D[i-1],\ D[i//2],\ D[i//3],\ D[i//5]\,) + 1$$

가능한 직전 단계들 중 가장 작은 값에 현재 연산 1회를 더한다

# 표가 채워지는 과정

얼핏 뒤에서부터 역으로 계산하는 것처럼 느껴지지만, 실제로는 작은 수부터 차근차근 올라가는 바텀업 방식이다.


**D[1] = 0**

이미 1이므로 연산이 필요 없다. 여기가 바닥이다.

**먼저 뺄셈으로 채운다**

D[i] = D[i-1] + 1 을 기본값으로 깔아 둔다.

**나눗셈으로 갱신**

2·3·5로 나누어떨어질 때마다 더 작은 값이 있는지 비교한다.

**D[26] = 3**

작은 수부터 올라오면 26에 도달할 즈음 답이 이미 결정되어 있다.

§6.5 1로 만들기

# 파이썬 코드

make_one.py

```python
def make_one(x):
    # d[i]: 정수 i를 1로 만드는 최소 연산 횟수
    d = [0] * (x + 1)

    # 바텀업 진행 (2부터 x까지)
    for i in range(2, x + 1):

        # 1. 일단 1을 빼는 경우를 기본값으로 설정
        d[i] = d[i - 1] + 1

        # 2. 2로 나누어떨어지는 경우와 비교
        if i % 2 == 0:
            d[i] = min(d[i], d[i // 2] + 1)

        # 3. 3으로 나누어떨어지는 경우와 비교
        if i % 3 == 0:
            d[i] = min(d[i], d[i // 3] + 1)

        # 4. 5로 나누어떨어지는 경우와 비교
        if i % 5 == 0:
            d[i] = min(d[i], d[i // 5] + 1)

    return d[x]

print(make_one(26))
```

OUTPUT

```
3
```

### 기본값을 먼저 깐다

뺄셈은 언제나 가능하므로 이것을 출발점으로 두고 나눗셈으로 깎아 내린다.

### min으로 갱신

조건이 맞을 때마다 더 나은 값이 있는지 비교한다. 순서는 상관없다.

### 정답은 3회

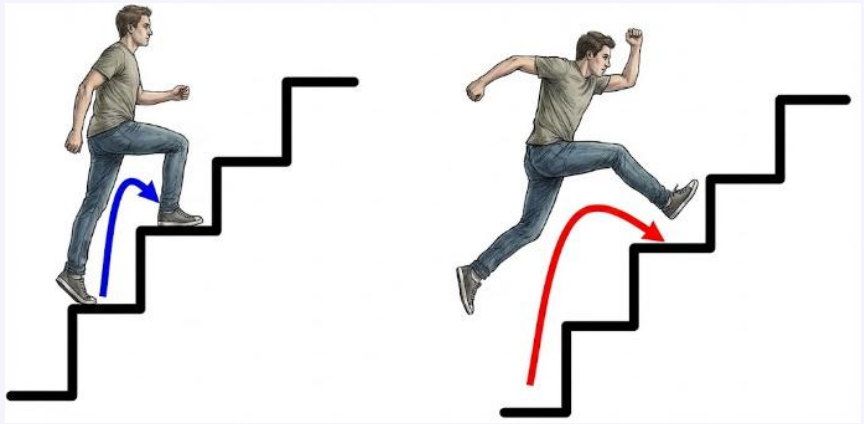
그리디가 5회를 쓴 자리에서 DP는 3회를 찾아낸다.

### 시간 복잡도 O(X)

2부터 X까지 한 번 훑으며 상수 번 비교할 뿐이다.

# 같은 점화식, 다른 문제 — 계단 오르기

n개의 계단을 한 번에 1계단 또는 2계단씩 오를 수 있다. 바닥에서 n번째 계단까지 도달하는 방법의 수는 몇 가지일까? 작은 n부터 직접 세어 보자



| n | 경우 | 가짓수 |
|---|---|---|
| 1 | (1) | 1 |
| 2 | (1,1) (2) | 2 |
| 3 | (1,1,1) (1,2) (2,1) | 3 |
| 4 | (1,1,1,1) (1,1,2) (1,2,1) (2,1,1) (2,2) | 5 |

1, 2, 3, 5 … 익숙한 숫자다. 4번째 계단에 도착하려면 직전이 3번째이거나 2번째뿐이다.

**D[i] = D[i-1] + D[i-2]**

바닥 공사도, 계단 오르기도, 결국 같은 점화식이다

6.6

# 0/1 배낭 문제 정복

5장에서 그리디에게 패배를 안겼던 그 문제를
이번에는 표를 그려 정확히 되찾는다.

§6.6 0/1 배낭 문제

# 그리디의 복수 — 0/1 배낭 문제 정복

드디어 올 것이 왔다. 5장에서 그리디 알고리즘에게 뼈아픈 패배를 안겨 주었던 그 문제, 0/1 배낭 문제를 정복할 시간이다. 이 문제는 DP의 꽃이자 "왜 복잡하게 표를 그려 가며 풀어야 하는가"에 대한 가장 확실한 대답이 된다.

**5장의 결과 — 14,000 G**

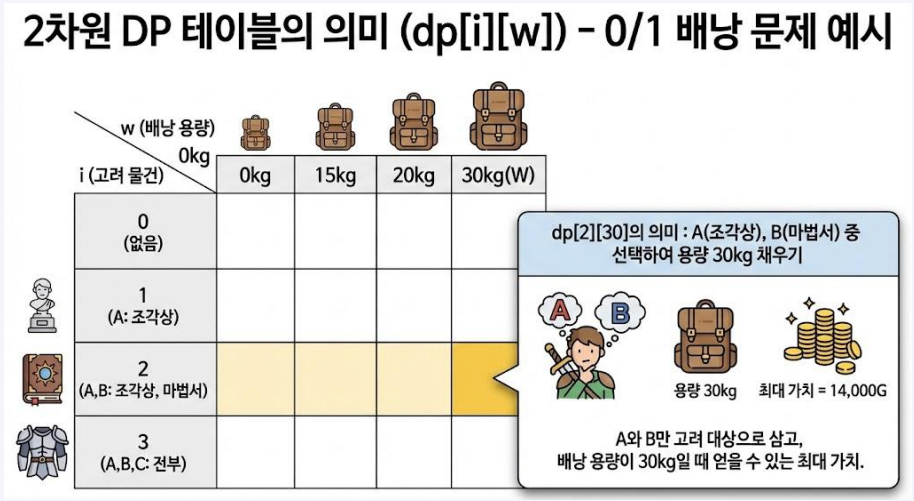
가성비 1등 A만 담고 10kg를 비운 채 나왔다.

**이 장의 목표 — 18,000 G**

B와 C를 함께 담는 조합을 표가 찾아낸다.

# 상황 설정 (복습)

배낭 용량은 30kg이고, 보물은 쪼갤 수 없다. 넣거나(1) 포기하거나(0) 둘 중 하나뿐이다.




| 보물 | 무게 | 가치 | 가성비 |
|---|---|---|---|
| **A (조각상)** | 20kg | 14,000 G | **700 (1등)** |
| **B (마법서)** | 15kg | 9,000 G | 600 (2등) |
| **C (갑옷)** | 15kg | 9,000 G | 600 (2등) |

가성비 순서대로 담으면 A(20kg) 뒤에 남는 10kg에는 아무것도 들어가지 않는다.

**물건과 용량, 두 축을 동시에 봐야 한다**

"몇 번째 물건까지 고려했는가"와 "지금 남은 용량이 얼마인가"라는 두 정보가 필요하므로 DP 테이블은 2차원이 된다.

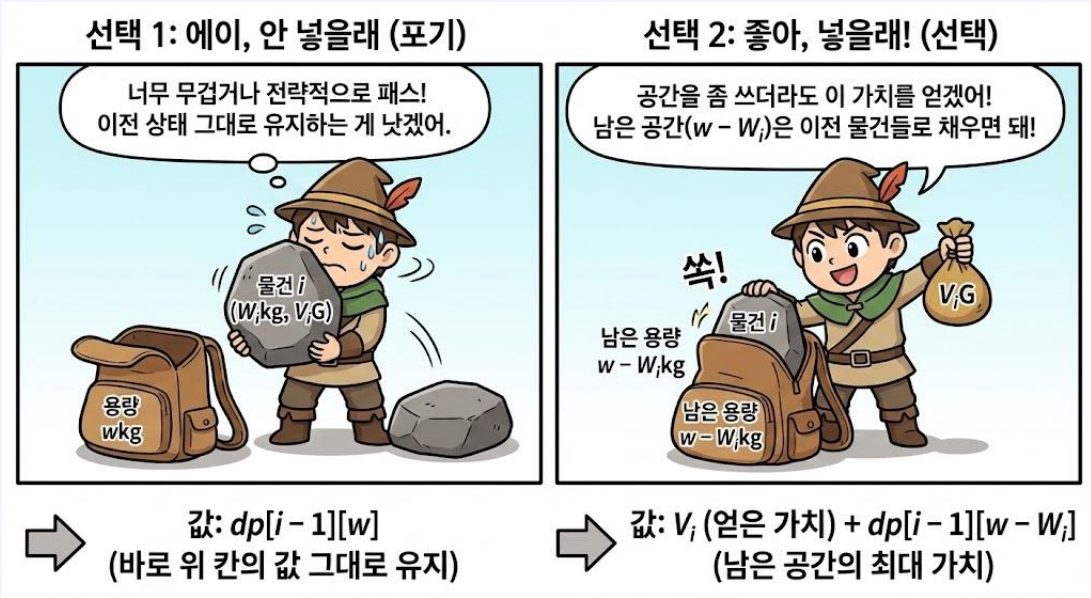
§6.6 0/1 배낭 문제

# 점화식의 마법 — 넣을까, 말까?

표의 한 칸 dp[i][w]를 채우는 주문을 만들어 보자. i번째 물건을 손에 들고 있고 지금 담을 수 있는 무게 한도가 w라고 하자. 이때 할 수 있는 선택은 딱 두 가지뿐이다.

? dp[i][w] = 앞에서 i개만 고려했을 때, 용량 w에서 얻을 수 있는 최대 가치

**포기한다 vs 넣는다**



§6.6 0/1 배낭 문제

# 최종 점화식

### ① "에이, 이 물건은 안 넣을래."

i번째 물건을 포기하면 결과는 i-1개까지만 고려했을 때와 똑같다. 용량도 그대로 남는다.

$$dp[i][w] = dp[i-1][w]$$

### ② "좋아, 이 물건을 넣자!"

용량이 충분할 때만 가능하다. 물건 무게만큼 비운 상태의 최적해에 이 물건의 가치를 더한다.

$$dp[i][w] = \max(dp[i-1][w],\ V_i + dp[i-1][w-W_i])$$

**$dp[i][w] = \max(\ dp[i-1][w],\ dp[i-1][w-w_i] + v_i\ )$**

두 선택지 중 더 큰 값을 고른다 — 단, 두 번째는 $w_i \le w$ 일 때만

# 0/1 배낭 문제 알고리즘 (의사코드)

knapsack.pseudo

```
// dp[i][w] = 앞에서 i개 물건만 고려했을 때,
//            용량 w에서 얻을 수 있는 최대 가치
KNAPSACK(weight[1..n], value[1..n], W):

    dp[0..n][0..W] <- 0

    for i <- 1 to n:
        for w <- 0 to W:

            // 1) i번째 물건을 넣지 않는 경우
            dp[i][w] <- dp[i-1][w]

            // 2) 넣을 수 있다면 (용량이 충분하다면)
            if weight[i] <= w:
                dp[i][w] <- max( dp[i][w],
                                 dp[i-1][w-weight[i]] + value[i] )

    return dp[n][W]
```

**표는 (n+1) × (W+1)**

0번째 행과 0번 열은 "아무것도 없을 때"를 뜻하며 모두 0이다.

**기본값은 "안 넣기"**

먼저 윗칸을 그대로 가져온 뒤, 넣는 쪽이 더 나은지 비교한다.

**w - weight[i] 칸 참조**

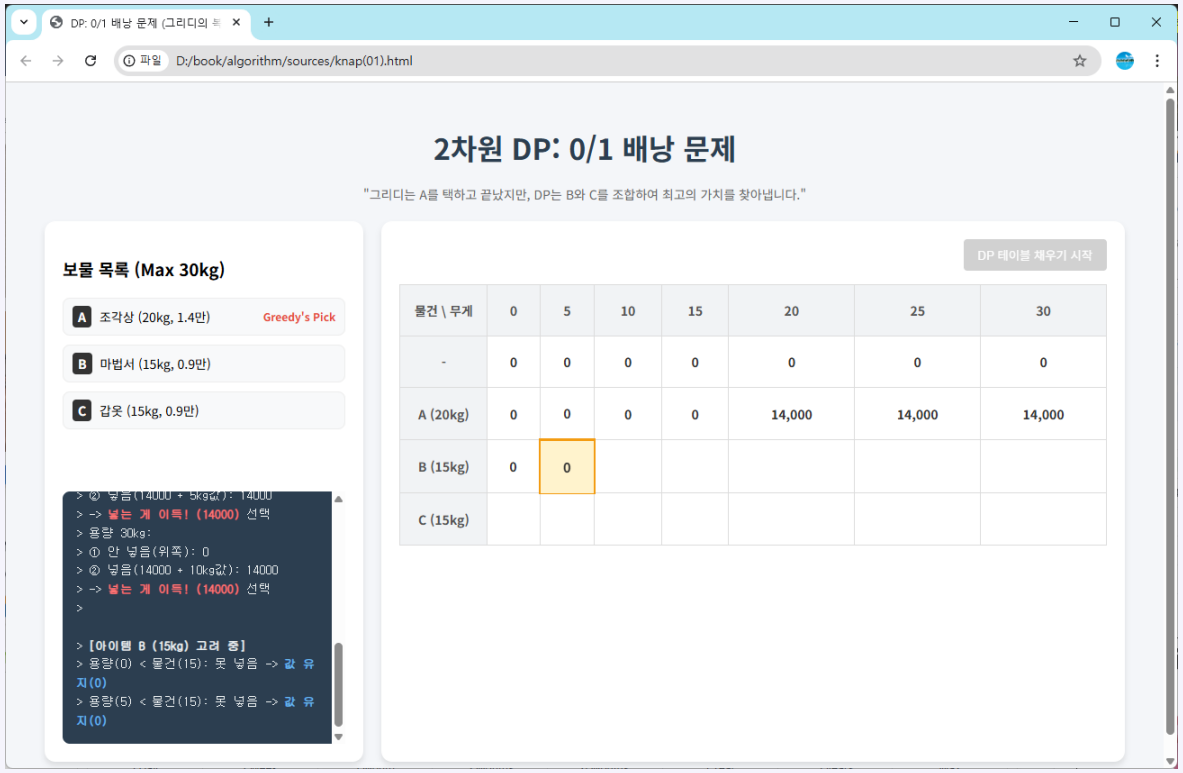
물건을 넣었을 때 남는 용량의 최적해를 가져온다. 항상 이미 계산된 칸이다.

**정답은 오른쪽 아래**

dp[n][W] 한 칸에 모든 물건과 전체 용량을 고려한 최적값이 들어 있다.

이중 반복문이므로 시간·공간 복잡도는 모두 O(n × W)이다.

# 표가 채워지는 과정




### 0행은 전부 0

물건을 하나도 고려하지 않았으니 어떤 용량에서도 가치는 0이다.

### 행 단위로 내려온다

물건을 하나씩 추가하며 위 행을 참조해 아래 행을 채운다.

### 칸마다 두 값을 비교

윗칸 값과 "넣었을 때"의 값 중 큰 쪽이 그 칸의 답이 된다.

### 마지막 칸이 정답

dp[3][30]에 18,000이 적히는 순간 그리디의 패배가 뒤집힌다.

# 파이썬 구현

knapsack.py

```python
def knapsack(weights, values, W):
    n = len(weights)

    # dp[i][w] = 앞에서 i개 물건을 고려했을 때,
    #            배낭 용량이 w일 때의 최대 가치
    dp = [[0] * (W + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(W + 1):

            # i번째 물건을 넣지 않는 경우
            dp[i][w] = dp[i - 1][w]

            # i번째 물건을 넣을 수 있는 경우
            if weights[i - 1] <= w:
                dp[i][w] = max(
                    dp[i][w],
                    dp[i - 1][w - weights[i - 1]] + values[i - 1]
                )

    return dp[n][W]
```

run.py

```python
weights = [20, 15, 15]
values  = [14000, 9000, 9000]
W = 30
print(knapsack(weights, values, W))
```

OUTPUT

```
18000
```

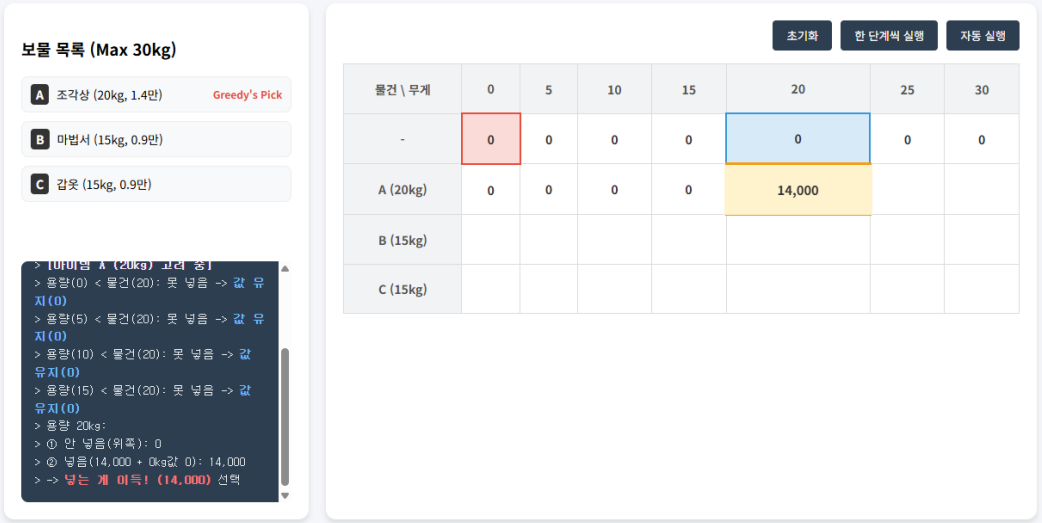
**5장의 14,000을 넘어섰다**

그리디가 고를 수 없었던 B + C 조합을 표가 찾아냈다. 두 미래를 모두 계산해 두었기 때문이다.

weights[i-1]처럼 인덱스가 하나 밀리는 지점이 이 코드의 유일한 함정이다.

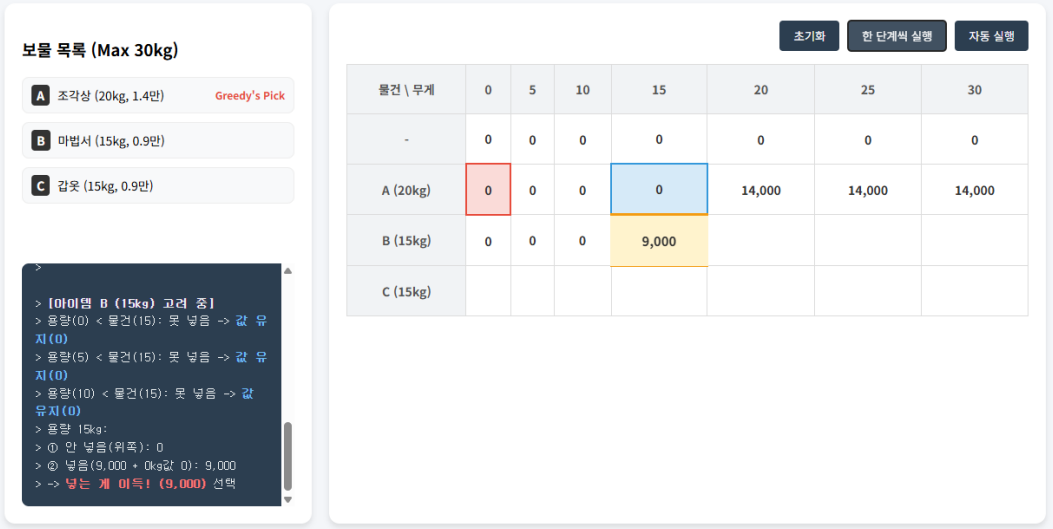
# 테이블 추적 ① — 첫 두 행

## 1행: 물건 A(20kg, 14,000G)만 고려




| 물건 \ 무게 | 0 | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A (20kg) | 0 | 0 | 0 | 0 | 14,000 | | |
| B (15kg) | | | | | | | |
| C (15kg) | | | | | | | |

## 2행: 물건 B(15kg, 9,000G)까지 고려



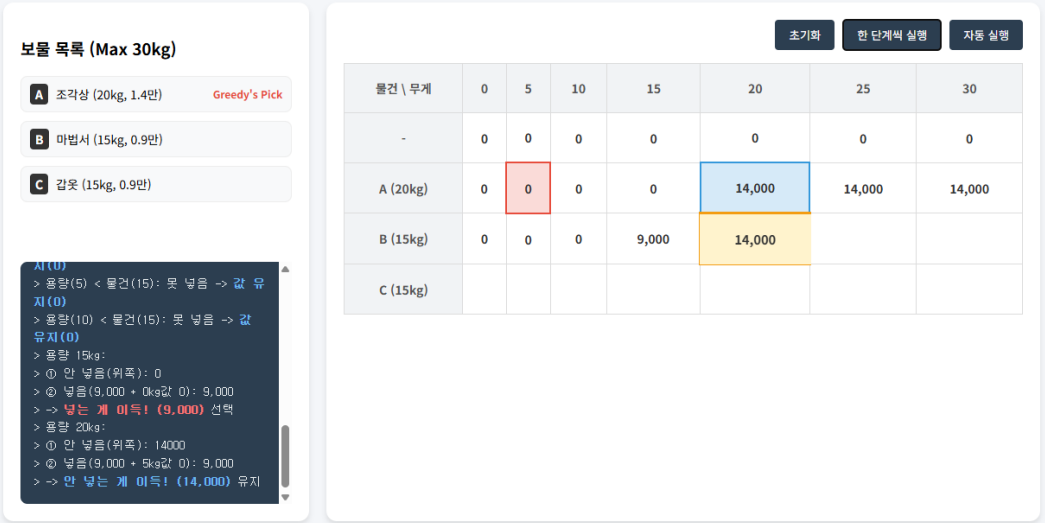

| 물건 \ 무게 | 0 | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A (20kg) | 0 | 0 | 0 | 0 | 14,000 | 14,000 | 14,000 |
| B (15kg) | 0 | 0 | 0 | 9,000 | | | |
| C (15kg) | | | | | | | |

### 20kg 전에는 아무것도 담을 수 없다

1행은 용량이 20 미만인 동안 0이다가 20에서 14,000으로 뛴다. 2행에서는 15kg짜리 B가 들어오면서 15~19 구간이 9,000으로 채워진다.

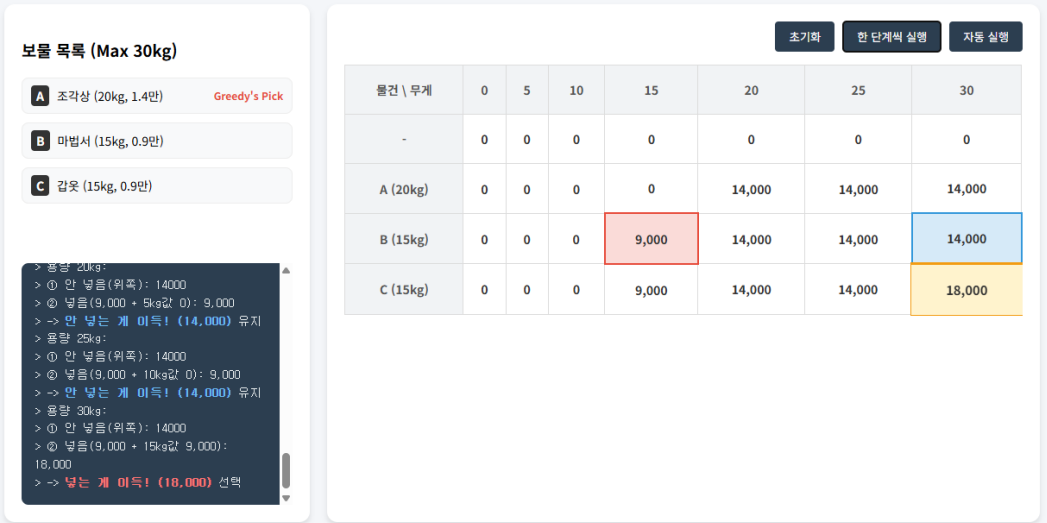
# 테이블 추적 ② — 마지막 행과 정답

### 3행: 물건 C(15kg, 9,000G)까지 고려




| 물건 \ 무게 | 0 | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A (20kg) | 0 | 0 | 0 | 0 | 14,000 | 14,000 | 14,000 |
| B (15kg) | 0 | 0 | 0 | 9,000 | 14,000 | | |
| C (15kg) | | | | | | | |

### 오른쪽 아래 칸 dp[3][30]이 최종 정답




| 물건 \ 무게 | 0 | 5 | 10 | 15 | 20 | 25 | 30 |
|---|---|---|---|---|---|---|---|
| - | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A (20kg) | 0 | 0 | 0 | 0 | 14,000 | 14,000 | 14,000 |
| B (15kg) | 0 | 0 | 0 | 9,000 | 14,000 | 14,000 | 14,000 |
| C (15kg) | 0 | 0 | 0 | 9,000 | 14,000 | 14,000 | 18,000 |

**dp[3][30] = 18,000 G**

B(9,000) + C(9,000) — 그리디가 놓친 조합을 표가 찾아냈다

6.7

# LCS 알고리즘

두 문자열에 공통으로 들어 있는
가장 긴 부분 수열을 찾는다.

6.7

# 부분 문자열과 부분 수열은 다르다

LCS(Longest Common Subsequence)는 두 문자열에서 공통으로 발견되는 부분 수열 중 가장 긴 것의 길이를 찾는 문제다. 먼저 두 용어를 구분해야 한다.

### 부분 문자열 (substring)

문자열에서 연속된 일부분이다. ABCDE의 부분 문자열은 AB, BCD, CDE 등이며 AC는 될 수 없다.

### 부분 수열 (subsequence)

순서만 유지하면 연속이 아니어도 된다. ABCDE의 부분 수열은 AC, ADE, BD 등 — 중간을 건너뛰어도 좋다.

**X = G X A Y C Z** **Y = P G Q A R C**

**공통 수열** G, A, C, GA, GC, AC, GAC — 이 중 가장 긴 것은 GAC 이다.

**LCS = 3**

건너뛰기가 허용되는 순간 경우의 수가 폭발한다. 그래서 표가 필요하다.

# 어디에 쓰이는가

## DNA 염기 서열 분석

A·T·C·G로 이루어진 긴 서열 두 개가 얼마나 닮았는지, 진화 과정에서 어떤 변이가 일어났는지를 LCS 길이로 잰다.

## 표절 검사

과제 표절 검사 프로그램은 두 문서의 공통 부분 수열 길이로 유사도를 수치화한다.

## git diff

두 버전의 파일에서 바뀌지 않은 줄을 찾아내는 작업이 곧 LCS다. diff 도구의 핵심 알고리즘이다.

**"두 서열이 얼마나 닮았는가"라는 질문은 어디에나 있다**

LCS가 길수록 두 대상은 비슷하다고 본다. 생물정보학, 문서 비교, 버전 관리가 모두 같은 표를 채운다.

두 문자열을 동시에 봐야 하므로, 배낭 문제와 마찬가지로 DP 테이블은 2차원이 된다.

# 2차원 배열로 문자열 비교하기

두 문자열 X(길이 N)와 Y(길이 M)가 있다고 하자. dp[i][j]는 X의 앞에서 i번째 글자까지와 Y의 앞에서 j번째 글자까지만 비교했을 때의 LCS 길이로 정의한다.

X=ABC, Y=ADC일 때 예시 테이블 설정은 다음과 같다.

| X \ Y | ""(0) | A(1) | D(2) | C(3) |
|---|---|---|---|---|
| ""(0) | 0 | 0 | 0 | 0 |
| A(1) | 0 | ? | ? | ? |
| B(2) | 0 | ? | ? | ? |
| C(3) | 0 | ? | ? | 최종 정답 |

**0행 0열** 한쪽이 빈 문자열이면 공통 부분 수열도 없다. 따라서 모두 0으로 시작한다. `= 0`

**마지막 칸** 오른쪽 아래 dp[N][M]에 두 문자열 전체의 LCS 길이가 들어간다. `dp[N][M]`

# 점화식 유도 — 글자가 같을 때와 다를 때

### 두 글자가 같을 때 X[i] == Y[j]

이 글자를 공통 부분 수열의 마지막 글자로 붙일 수 있다는 뜻이다. 길이가 1 늘어난다. 이 글자가 붙기 직전의 상황은 X는 i-1까지, Y는 j-1까지 비교하던 때 — 왼쪽 위 대각선 칸이다.

```
dp[i][j] = dp[i-1][j-1] + 1
```

### 두 글자가 다를 때 X[i] ≠ Y[j]

둘을 동시에 마지막 글자로 쓸 수는 없으니 하나는 보류해야 한다. X의 글자를 무시한 결과(위쪽 칸)와 Y의 글자를 무시한 결과(왼쪽 칸) 중 더 긴 쪽을 가져온다.

```
dp[i][j] = max(dp[i-1][j], dp[i][j-1])
```

### 표를 채우는 손이 향하는 방향은 셋뿐이다

글자가 같으면 대각선에서, 다르면 위 또는 왼쪽에서 값을 가져온다. 이 세 방향만 기억하면 LCS 표는 손으로도 채울 수 있다.

역추적할 때도 같은 세 방향을 거꾸로 되짚으면 실제 문자열이 복원된다.

# LCS 길이 구하기 알고리즘 (의사코드)

lcs_length.pseudo

```
// X: 길이 m인 문자열, Y: 길이 n인 문자열
LCS_LENGTH(X, Y):

    m <- LENGTH(X)
    n <- LENGTH(Y)
    create array dp[0..m][0..n]

    // 초기화: 한쪽이 빈 수열이면 LCS 길이는 0
    for i <- 0 to m do  dp[i][0] <- 0
    for j <- 0 to n do  dp[0][j] <- 0

    // 점화식에 따라 테이블 채우기
    for i <- 1 to m do
        for j <- 1 to n do
            if X[i-1] = Y[j-1] then
                dp[i][j] <- dp[i-1][j-1] + 1
            else
                dp[i][j] <- max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]
```

### 0행·0열 초기화

빈 문자열과의 LCS는 0이다. 이 바닥이 있어야 나머지 칸이 참조할 곳이 생긴다.

### 같으면 대각선 + 1

한 글자를 확정하고 두 문자열 모두 한 칸씩 물러난 상태를 본다.

### 다르면 max

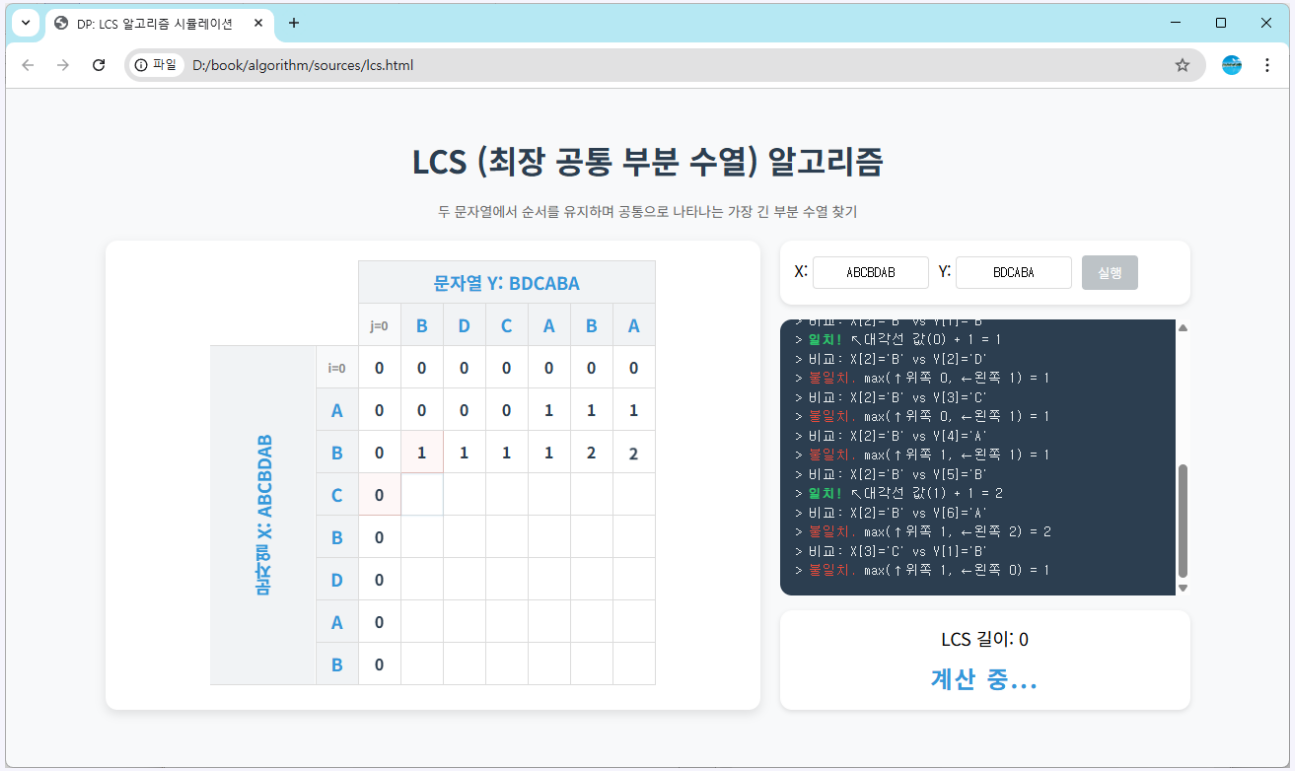
어느 쪽 글자를 포기할지 지금 정하지 않고, 더 나은 쪽을 그대로 물려받는다.

### O(m × n)

표의 칸 수만큼만 일한다. 각 칸은 상수 시간에 채워진다.

점화식 두 줄이 알고리즘의 전부다 — 나머지는 표를 순서대로 훑는 이중 반복문뿐이다.

# 표가 채워지는 과정




### ① 테두리를 0으로

0행과 0열을 0으로 채워 두면 예외 처리가 필요 없다.

### ② 같은 글자를 만나면

왼쪽 위 대각선 값에 1을 더해 적는다. 길이가 자란다.

### ③ 다른 글자를 만나면

위와 왼쪽 중 큰 값을 그대로 옮겨 적는다. 길이는 유지된다.

### ④ 오른쪽 아래가 정답

GCATCG와 GTAGCT의 LCS 길이는 3이다.

# 파이썬 구현

lcs.py

```python
def solve_lcs(text1, text2):
    n = len(text1)
    m = len(text2)

    # 1. DP 테이블 초기화 ((n+1) x (m+1) 2차원 배열)
    #    dp[i][j] = text1의 i번째, text2의 j번째까지의 LCS 길이
    dp = [[0] * (m + 1) for _ in range(n + 1)]

    # 2. 이중 반복문으로 테이블 채우기
    for i in range(1, n + 1):
        for j in range(1, m + 1):

            # CASE 1: 글자가 같으면 (대각선 왼쪽 위 + 1)
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1

            # CASE 2: 글자가 다르면 (위쪽과 왼쪽 중 큰 값)
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # 3. 오른쪽 아래 칸이 최종 정답
    return dp[n][m]
```

run.py

```python
str_a = "GCATCG"
str_b = "GTAGCT"
result = solve_lcs(str_a, str_b)
print("LCS 길이:", result)
```

OUTPUT

```
LCS 길이: 3
```

### 공통 부분 수열은 GAT

GCATCG에서 G·A·T, GTAGCT에서도 G·A·T를 순서대로 뽑을 수 있다. 연속이 아니어도 되므로 길이 3이 나온다.

문자열 인덱스는 0부터, 표 인덱스는 1부터다. text1[i-1]의 -1을 빠뜨리는 실수가 가장 흔하다.

# 길이만이 아니라 실제 문자열까지 찾아내기

표에는 길이만 적혀 있지만, 오른쪽 아래에서 거꾸로 되짚어 오면 실제 공통 수열을 복원할 수 있다.

lcs_trace.py

```python
"""DP 테이블을 역추적하여 실제 LCS 문자열을 찾는 함수"""
def get_lcs_string(text1, text2, dp):
    i = len(text1)
    j = len(text2)
    lcs_str = []

    # 끝에서부터 0이 될 때까지 역으로 추적
    while i > 0 and j > 0:

        # 1. 현재 글자가 같다면 LCS에 포함된다
        if text1[i-1] == text2[j-1]:
            lcs_str.append(text1[i-1])
            i -= 1
            j -= 1              # 대각선 위로 이동

        # 2. 다르다면 값이 더 큰 쪽에서 왔을 것이다
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1              # 위쪽으로 이동
        else:
            j -= 1              # 왼쪽으로 이동

    # 거꾸로 담았으므로 뒤집어서 반환
    return "".join(lcs_str[::-1])
```

### 세 방향을 거꾸로

표를 채울 때 온 길을 그대로 되짚는다. 대각선·위·왼쪽 셋뿐이다.

### 같을 때만 기록

대각선으로 움직이는 순간의 글자가 LCS의 구성원이다.

### 마지막에 뒤집는다

끝에서부터 담았으므로 [::-1]로 순서를 되돌린다.

### 추적은 O(m + n)

한 걸음마다 i나 j가 최소 하나는 줄어든다.

6.8

# 코딩 테스트

1차원 DP와 2차원 DP를 하나씩 직접 풀어 본다.

CT 6.1 개미 전사

# 개미 전사의 식량 창고 습격 (1차원 DP)

개미 전사가 일직선으로 늘어선 식량 창고를 습격하려 한다. 각 창고에는 저장된 식량의 양이 적혀 있다. 들키지 않으려면 인접한 두 창고를 연속으로 털 수 없다. 어떤 창고를 털었다면 바로 옆 창고는 건드릴 수 없다. 이때 얻을 수 있는 식량의 최댓값을 구하라.

input.txt

```
4
1 3 1 5
```

OUTPUT

```
8
```

dp[i] = 0번부터 i번 창고까지 고려했을 때 얻을 수 있는 식량의 최댓값

**선택은 언제나 두 갈래**

i번 창고를 턴다면 i-1번은 포기해야 하므로 dp[i-2] + k[i]가 되고, 털지 않는다면 dp[i-1] 그대로다. 둘 중 큰 값이 답이다.

| i | k[i] | dp[i] |
|---|---|---|
| **0** | 1 | 1 |
| **1** | 3 | 3 |
| **2** | 1 | 3 |
| **3** | 5 | **8** |

1번(3)과 3번(5)을 털어 합계 8 — 인접하지 않으면서 최대가 되는 유일한 조합이다.

CT 6.1 개미 전사

# 풀이 — 털거나, 건너뛰거나

ant_warrior.py

```python
import sys

def solve():
    input = sys.stdin.readline

    n = int(input().strip())            # 창고 개수
    k = list(map(int, input().split())) # 각 창고의 식량

    if n == 1:                          # 예외 처리
        print(k[0])
        return

    # dp[i]: 0번부터 i번 창고까지의 최댓값
    dp = [0] * n
    dp[0] = k[0]
    dp[1] = max(k[0], k[1])

    # 1) i번을 안 턴다      -> dp[i-1]
    # 2) i번을 턴다(i-1 포기) -> dp[i-2] + k[i]
    for i in range(2, n):
        dp[i] = max(dp[i-1], dp[i-2] + k[i])

    print(dp[n-1])

solve()
```

### dp의 뜻을 먼저 적는다

"i번까지 고려했을 때의 최댓값"이라는 정의가 점화식을 그대로 낳는다.

### dp[1]은 max로

0번과 1번 중 큰 쪽이다. 둘 다 털 수는 없다.

### max 한 줄이 전부

털지 않는 경우와 터는 경우를 비교한다. 그리디로는 이 비교가 불가능하다.

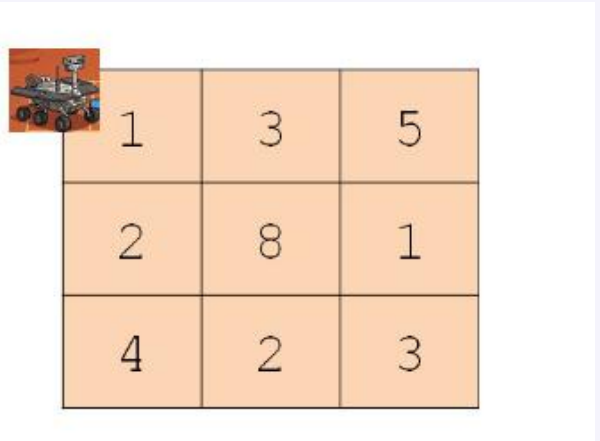
### O(N) · 메모리 O(N)

직전 두 값만 있으면 되므로 변수 두 개로 줄일 수도 있다.

CT 6.2 화성 탐사 로봇

# 화성 탐사 로봇의 최적 경로 (2차원 DP)

N×M 크기의 화성 지도가 있다. 로봇은 좌측 상단 (0,0)에서 출발해 우측 하단 (N-1,M-1)까지 이동한다. 오른쪽·아래·대각선 아래로만 갈 수 있고, 각 칸에는 지형의 험준함(비용)이 적혀 있다.




**오른쪽 / 아래로 이동**
도착 칸의 비용이 그대로 소모된다.

**대각선 아래로 이동**
연료 효율이 좋아 도착 칸 비용의 50%만 든다 (소수점 버림).

map_3x3.txt

```
[1, 3, 5]
[2, 8, 1]
[4, 2, 3]
```

dp[i][j] = (0,0)에서 (i,j)까지 도달하는 데 드는 최소 연료

최소 연료 소모량은 6 — 대각선을 두 번 타는 경로가 정답이다.

CT 6.2 화성 탐사 로봇

# 풀이 ① — 테두리부터 깐다

첫 행과 첫 열은 올 수 있는 방향이 하나뿐이다. 이 테두리를 먼저 채워 두어야 이후 칸들이 참조할 값이 생긴다.

mars_robot.py

```python
def solve_mars_robot(n, m, grid):

    # DP 테이블 초기화
    dp = [[0] * m for _ in range(n)]

    # --- 초기값 (base case) ---
    # 시작점의 비용은 무조건 소모된다
    dp[0][0] = grid[0][0]

    # 첫 번째 행: 위나 대각선에서 올 수 없고 왼쪽에서만 온다
    for j in range(1, m):
        dp[0][j] = dp[0][j-1] + grid[0][j]

    # 첫 번째 열: 왼쪽이나 대각선에서 올 수 없고 위에서만 온다
    for i in range(1, n):
        dp[i][0] = dp[i-1][0] + grid[i][0]
```

### 시작 칸도 비용을 낸다

dp[0][0]은 0이 아니라 grid[0][0]이다. 출발 지점을 밟는 비용이 있다.

### 첫 행은 왼쪽에서만

위쪽도 대각선도 지도 밖이므로 선택의 여지가 없다.

### 첫 열은 위쪽에서만

마찬가지로 누적합을 그대로 쌓아 내려간다.

### 테두리를 빼먹으면

내부 칸이 0을 참조해 비용이 터무니없이 작게 나온다. 가장 흔한 버그다.

2차원 DP는 "테두리 → 내부" 순서가 거의 언제나 정석이다.

CT 6.2 화성 탐사 로봇

# 풀이 ② — 세 방향 중 최소를 고른다

mars_robot.py

```python
    # --- 점화식 적용 (bottom-up) ---
    for i in range(1, n):
        for j in range(1, m):
            cost = grid[i][j]

            # 1. 위에서 오는 경우 (비용 그대로)
            from_top  = dp[i-1][j] + cost

            # 2. 왼쪽에서 오는 경우 (비용 그대로)
            from_left = dp[i][j-1] + cost

            # 3. 대각선에서 오는 경우 (50% 할인, 버림)
            from_diag = dp[i-1][j-1] + (cost // 2)

            # 세 경우 중 최소 비용 선택
            dp[i][j] = min(from_top, from_left, from_diag)

    return dp[n-1][m-1]
```

run.py

```python
N, M = 3, 3
example_map = [
    [1, 3, 5],
    [2, 8, 1],
    [4, 2, 3]
]
print(solve_mars_robot(N, M, example_map))
```

OUTPUT

```
지도 크기: 3 x 3
--------------------
최소 연료 소모량: 6
```

대각선 할인이 있으므로 최단 거리 경로와 최소 비용 경로가 일치하지 않는다.

**cost // 2 는 정수 나눗셈**

파이썬에서 //는 몫이다. 소수점 버림이 조건과 정확히 맞는다.

SUMMARY

# 6장 정리

### 두 번 계산하지 않는다

DP의 본질은 기억이다. 나눈다 → 기억한다 → 재활용한다, 이 세 단계가 전부다.

### 두 조건을 먼저 확인한다

최적 부분 구조가 없으면 답이 틀리고, 중복 부분 문제가 없으면 표를 만들 이유가 없다.

### 탑다운과 바텀업

같은 점화식을 위에서 내려가며 풀면 메모이제이션, 아래에서 올라오며 풀면 타뷸레이션이다.

### 점화식이 같으면 같은 문제

바닥 공사, 계단 오르기는 겉포장만 다를 뿐 피보나치와 똑같은 D[i]=D[i-1]+D[i-2]다.

### 0/1 배낭을 되찾았다

두 축(물건·용량)이 필요하면 표는 2차원이 된다. 그리디의 14,000을 18,000으로 뒤집었다.

### LCS는 문자열용 2차원 DP

같으면 대각선+1, 다르면 위·왼쪽 중 최대. 역추적하면 실제 문자열까지 복원된다.

CHAPTER 06

# Q & A

한 번 푼 문제는 두 번 풀지 않습니다.
질문도 표에 적어 두면 다음에 더 빨라집니다.