다이나믹 프로그래밍

메모리 공간을 약간 더 사용하면 연산 속도를 비약적으로 증가시킬 수 있는 방법

보텀업 방식과 탑다운 방식 2가지로 나눌 수 있음!

Top-Down 방식	큰 문제를 해결하기 위해 작은 문제를 호출	재귀함수 이용	메모이제이션
Bottom-Up 방식	작은 문제부터 차근차근 답을 도출	반복문을 이용해 소스코드를 작성	DP 테이블

⭐ 다음 조건을 만족할 때, DP를 사용할 수 있다!

1. 큰 문제를 작은 문제로 나눌 수 있다.
2. 작은 문제에서 구한 정답은 그것을 포함하는 큰 문제에서도 동일하다

문제풀이 단계!

1. 다이나믹 프로그래밍 유형임을 파악
   - 완전 탐색 알고리즘으로 접근했을 때, 시간이 매우 오래 걸리면 다이나믹 프로그래밍을 적용할 수 있는지 확인해보기!
2. 재귀함수로 비효율적인 프로그램 작성한 뒤, 메모이제이션을 적용할 수 있으면(작은 문제에서 구한 답이 큰 문제에서 그대로 사용될 수 있으면) 코드 개선해보기!

⭐ DP를 구현하는 방식으로 **‘메모이제이션’**이 자주 사용됨! (Bottom-up)

메모이제이션이란?

한번 구한 결과를 메모리 공간에 메모해두고, 같은 식을 다시 호출하면 메모한 결과를 그대로 가져오는 기법.

메모이제이션은 값을 저장하는 방법이므로 ‘캐싱’이라고도 불림

HOW?

한번 구한 정보를 ***리스트에 저장***하면 됨! 다이나믹 프로그래밍을 재귀적으로 수행하다가 같은 정보가 필요할 때는 이미 구한 정답을 그대로 리스트에서 가져오면 된다!
→ 사전(dict) 자료형을 이용하기도 함!

ex) 피보나치 수열

- 이전 두 항의 합을 현재의 항으로 설정!
- 점화식을 이용해서 풀이 !

$$a_n = a_{n-1} + a_{n-2},\ a_1 = 1,\ a_2 = 1$$

→인접 3항간 점화식

→ n번째 피보나치 수 = n-1번째 피보나치 수 + n-2번째 피보나치 수

→ 프로그래밍에서는 이러한 수열을 배열이나 리스트로 표현! 수열 자체가 여러개의 수가 규칙에 따라서 배열된 형태를 의미하기 때문!

ex) 재귀함수를 이용해 수학적 점화식 구현

```
def fibo(x):
  if x == 1 or x == 2:
    return 1
  return fibo(x-1) + fibo(x-2)

print(fibo(4))
# 시간복잡도가 O(2^N)으로 동일한 함수가 반복적으로 호출되기 때문에 낭비! → DP 이용!
```

ex) DP를 이용한 피보나치 수열 구현(재귀함수) → 탑다운 방식

```
d = [0] * 100

def fibo(x):
  if x == 1 or x== 2:
    return 1
  if d[x] != 0:
    return d[x]
  d[x] = fibo(x-1) + fibo(x-2)
  return d[x]
```

```
print(fibo(99))
```

‼퀵정렬 vs 다이나믹 프로그래밍

둘 다 큰 문제를 작게 나누는 방식이라는 점에서 공통점!

퀵정렬 - 한번 기준 원소가 자리를 변경해서 자리를 잡게 되면, 그 기준 원소의 위치는 더 이상 바뀌지 않고 그 피벗값을 다시 처리하는 부분 문제는 존재하지 않음

다이나믹 프로그래밍 - 한번 해결했던 문제를 다시금 해결한다는 점이 특징!

ex) DP를 이용한 피보나치 수열 구현(재귀함수) → `보텀업 방식`

```
# 앞서 계산된 결과를 저장하기 위한 DP 테이블 초기화
d = [0] * 100

d[1] = 1
d[2] = 1
n = 99

# 피보나치 함수 - 반복문으로 구현 ( 보텀업 다이나믹 프로그래밍)
for i in range(3, n+1):
  d[i] = d[i-1] + d[i-2]

print(d[n])
```

*전형적인 다이나믹 프로그래밍 형태는 보텀업! → DP 테이블이라고 불림

*일반적으로 반복문을 이용한 다이나믹 프로그래밍이 더 성능이 좋으며, 오버헤드를 줄일 수 있음!