**DP(다이나믹 프로그래밍)**

**큰 문제를 작은 단위로 푸는 알고리즘**

- 두 가지 속성을 만족해야 다이나믹 프로그래밍을 풀 수 있음

1. Overlapping Subproblem : 겹치는 부분 문제

2. Optimal Substructure : 문제의 답을 작은 문제의 정답에서 구할 수 있음

1. Overlapping Subproblem

- 피보나치 수 : 크다 작다는 상대적인 문제

- F0 = 0

- F1 = 1

- F(n) = F(n-1) + F(n-2) (단, n >= 2)

- ★문제 : N번째 피보나치 수를 구하는 문제

-- 작은 문제 : N-1번째 피보나치 수를 구하는 문제, N-2번째 피보나치 수를 구하는 문제

- ★문제 : N-1번째 피보나치 수를 구하는 문제

-- 작은 문제 : N-2번째 피보나치 수를 구하는 문제, N-3번째 피보나치 수를 구하는 문제

- 문제가 반복될 때 겹치는 문제가 생긴다.

- 즉, 큰 문제와 작은 문제를 같은 방법으로 풀 수 있다.

2. Optimal Substructure : 문제의 답을 작은 문제의 정답에서 구할 수 있음

- 서울에서 부산가는 가장 빠른 길이 대전과 대구를 순서로 거쳐야 한다면,

서울 -> 대전 -> 대구 -> 부산

- 대전에서 부산을 가는 가장 빠른 길은 대구를 거쳐야 한다.

대전 -> 대구 -> 부산

- 어떤 문제의 정답은 구할 때마다 같다

- Optimal Substructure를 만족한다면, 문제의 크기에 상관없이 어떤 한 문제의 정답은 일정하다.

- 10번째 피보나치 수를 구하면서 구한 4번째 피보나치 수

- 9번째 피보나치 수를 구하면서 구한 4번째 피보나치 수

...

- 4번째 피보나치 수는 항상 같다.

- 사람의 이름을 물어볼 때 그 사람의 이름을 어디다 적어두는거

- 따라서, 정답을 한 번 구했으면, 정답을 어딘가에 메모해 놓는것이다.

= Memoization (메모를 한다는 뜻 = 배열에 저장하는 것)

Memo[i] = i번째 피보나치 수

따라서, 메모를 쓰고, 메모를 읽는 코드가 필요하다.

**- 다이나믹을 푸는 두 가지 방법!**

1. Top-down : 위에서 내려오는 것

2. Bottom-up : 아래에서 내려오는 것

1. Top-down (재귀함수를 이용하여 푼다.)

A. 문제를 풀어야 한다.

-- fibonacci(n)

B. 문제를 작은 문제로 나눈다.

--fibonacci(n-1)과 fibonacci(n-2)로 문제를 나눈다.

C. 작은 문제를 푼다.

-- fibonacci(n-1)과 fibonacci(n-2)를 호출해 문제를 푼다.

D. 작은 문제를 풀었으니, 이제 문제를 푼다.

-- fibonacci(n-1)의 값과 fibonacci(n-2)의 값을 더해 문제를 푼다.

n이라는 문제를 n-1과 n-2로 푼다. (일반적 피보나치 수열 문제(TopDown.java)가 그렇다)

1. Bottom-up(for 문을 이용하여 구함) = 시간 복잡도 : n

A. 문제를 크기가 작은 문제부터 차례대로 푼다.

-- for(int i=2; i<=n; i++)

B. 문제의 크기를 조금씩 크게 만들면서 문제를 점점 푼다.

-- for(int i=2; i<=n; i++)

C. 작은 문제를 풀면서 왔기 때문에, 큰 문제를 항상 풀 수 있다.

-- d[i] = d[i-1] + d[i-2]

D. 그러다보면, 언젠간 풀어야 하는 문제를 풀 수 있다.

-- d[n]을 구하게 된다.

DP 시간 복잡도

총 채워야하는 칸의 수 X 1칸을 채우는 복잡도

N개 X O(1) => O(n)

**문제 풀이 전략**

Top-Down과 Bottom-Up 중 둘 중 하나만 잘해도 된다.

재귀 or for 그 순서임. 나는 for가 더 익숙하고 시간적 소요를 생각하면 for로...