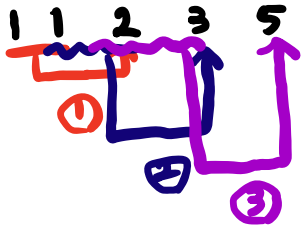
# DP ( Dynamic Programming )

⇒ DP는 다이나믹 프로그래밍으로, 하나의 큰 문제를 여러개의 작은 문제로 나누어서 그 결과를 저장하며 큰 문제를 해결할 때, 다시 사용하는 것으로 알고리즘 보다는 하나의 문제해결의 패러다임으로 볼 수 있다.

## Why DP?

⇒ 재귀함수에서 몇몇 값들이 여러번 반복되어 계산되는 비효율이 발생할 수 있다!

예를 들어 '피보나치 수열'을 예로 들면, $f(n) = f(n-1) + f(n-2)$




⇒ 이미 계산한 값을 사용함! (이를 위해 일반 재귀함수는 구했던 값을 또 계산)

DP는 이미 계산한 값을 사용하므로, 더 효율적이다! $(O(f(n)))$ 정도!

↳ 다항식 수준으로, 상황에 따라 다름

* DP를 사용하기 위해서는 다음 두 가지의 조건을 만족해야한다!

### 1) Overlapping Subproblems (중복 문제)

⇒ DP는 문제 결과값을 재활용하므로, 동일한 문제들이 반복적으로 나타난다.
이 때문에, 부분 문제 (계산)가 반복적으로 나타나지 않으면 DP를 사용할 수 없다.

ex) fibonacci ⇒ $f(n) = f(n-1) + f(n-2)$

$f(4) = f(3) + f(2)$

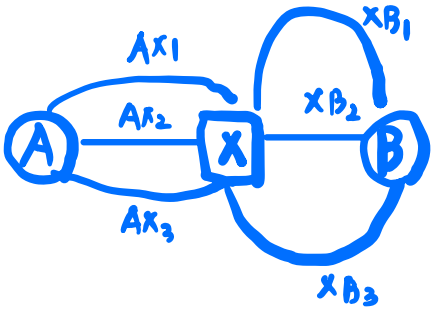
$f(2)+f(1)$ $\quad f(1)+f(0)$ → 중복되는 값들이 보인다!

f(1)+f(0)

## 2) Optimal Substructure (최적 부분 구조)

⇒ 부분문제의 최적결과 값을 사용해 전체문제의 최적 결과를 낼 수 있는 경우이다.

ex)




⇒ 여기서 최단거리는 무조건 1가지다.
다른 경로를 선택한다면 전체 최단 거리는 변하지 않는다.

## DP 사용하기

1) DP로 풀수있는 문제인지 확인! ⇒ 데이터의 최대화/최소화 문제 (위 조건들 충족)

2) 문제의 변수 파악 ⇒ 피보나치 수열처럼, 몇개의 변수를 쓰게 될지 알아야 재사용에 편하다.

3) 변수간 관계식 만들기 ⇒ 변수들에 의해 값이 계속 달라지지만, 동일한 변수값(위 피보나치 예시 참조)이면, 동일한 값을 가진다. 이 값을 사용해야 하므로 '점화식'을 만들어 관계식을 구성한다.

4) 메모하기 ⇒ 변수간 관계식까지 생성했다면, 관계식에서 변수에 대한 결과값을 저장한다. 메모한다고 해서 "Memoziation"이라 불렀다. 위 메모 값들을 재사용하는 방식으로 문제를 해결한다.

5) 기저상태 파악하기 ⇒ 가장 작은 문제의 상태를 말해야 한다.
(보통 하드코딩으로 직접 나타내는 경우도 있다.)
피보나치에서 $f(0)=f(1)=1$ 처럼 가장 작은 문제!

6) 구현하기 ⇨ 두 가지 방식으로 구현한다.

① Bottom-up (Tabulation) ⇨ 반복문 사용

② Top-down (Memoziation) ⇨ 재귀 사용

## 6-1) Bottom-up

⇒ 아래에서 부터 계산을 수행하고 누적시켜서 전체 큰 문제를 해결한다

dp라는 배열을 만들어서 (1차원으로 가정 ⇨ dp[0]이 가저상해, dp[n]이 목표값)

dp[0]부터 반복문을 통해 점화식으로 결과를 내서 dp[n]까지 값을 전이시키는 방식

Why Tabulation? (역할은 memoziation 이랑 비슷함 ⇒ 재활용 하므로)

↳ dp[0]부터 하나씩 채우는 과정을 table-filling 이라고 함 (기원)

## 6-2) Top-down

⇒ dp[0]부터 출발하는 것대신, dp[n]의 값을 찾기 위해 위에서부터 바로 호출을 해서 dp[0] 상태까지 내려간다음, 해당 결과 값을 재귀를 통해 전이시켜 재활용한다. (이미 이전에 완료한 계산이면, 단순히 메모리에서 저장된 값을 꺼내서 사용하면 된다.)

최근 상태값을 메모해 두었다고 해서 'Memoziation' 이라 불린다.