동적계획법

동적

: 어떤 부분 → 2개 이상의 문제를 푸는 데 사용 가능!
→ 1번만 계산해서 재활용 → 속도↑
→ 각 문제의 답을 **메모리**에 저장!
- overlapping subproblem : 두 번 이상 계산되는 부분 문제
- cache : 이미 계산된 값을 저장해두는 메모리의 장소

계획법

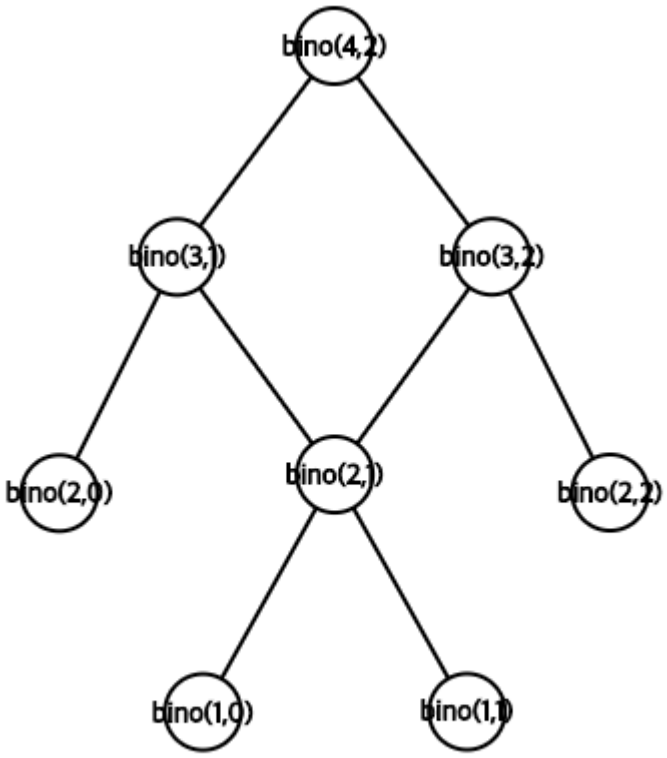
: 더 작은 문제들로 나눈 후 각 조각의 답 계산 → 원래 대답 해결

- 나눠진 문제들이 같은 부분 문제에 의존하면?
    - 그냥 재귀를 사용한다면 중복계산 ↑ (depth ↑ → 지수로 증가 : combinational explosion)

ex. 이항계수 계산 (binomial coefficient)
반복문 X → 재귀 호출 몇 번 일어나나 확인하면 됨! (같은 값을 두 번 이상 계산할 일 빈번하게 일어남!)

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

```
def bino(n,r):
  // n=r or r=0
  if r == 0 || n==r:
    return 1
  return bino(n-1, r-1) + bino(n-1, r)
```



위의 경우 중복호출 수가 n, r이 커질수록 증가한다. (n +1 → 호출수 *2)

계산량을 줄이려면?

A. bino(n,r)의 값은 일정 → n, r의 조합에 대해 답을 저장하는 캐시 배열을 만듦 → 각 입력에 대한 반환값 저장

→ 함수 : 매 호출마다 배열에 접근 → 저장되어 있음? return / 저장되어 있지 않음? 계산 → 저장 → return

메모이재이션(memoization) : 함수 결과 저장할 장소 마련 & 한 번 계산한 값 저장 → 재활용 최적화 기법

```
def bino2(n,r):
  cache = [[-1 for _ in range(r+1)] for _ in range(n+1)]
  if (r==0 || n==r):
    return 1
  if (cache[n][r] != -1:
    return cache[n][r]
  cache[n][r] = bino2(n-1, r-1) + bino2(n-1, r)
  return cache[n][r]
```

2번 이상 반복 계산되는 부분 문제들의 답을 미리 지정 → 속도 향상! 알고리즘 설계기법 : 동적계획법

- 메모이제이션을 적용할 수 있는 경우
  - 함수의 반환값이 input 만으로 결정되는 지 여부 : 참조적 투명성(referential transparency)
  - input 고정되면 output이 항상 같은 함수 : 참조적 투명 함수(referential transparent function)

  → 참조적 투명 함수에만 메모이제이션 적용 가능!

- 메모이제이션 구현 패턴
  - 기저 사례 제일 먼저 처리
  - 함수의 반환값은 항상 0 이상 → cache의 초기값 -1로 설정 (음수가 반환되면 사용X)
  - ret 사용! : cache[a][b]에 대한 reference

    → ret 값 바뀌면 : cache[a][b]도!

```
def someObscureFunction(a,b):
  memo = [[0 for _ in range(a+1)] for _ in range(b+1)]
  // or memo= {:...} 와 같이 dict도 사용 가능
  if (): //기저사례 작성
    return ()
  ret = memo[a][b]
  if ret != -1:
    return ret
```

- 메모이제이션의 시간복잡도 분석

: 존재하는 부분 문제 수 * 한 부분 문제를 풀 때 필요한 반복문의 수행 횟수