[Q] 예습자료에서 (3%7)×(3%7) %7 가 3×P(1) % 7로 바뀌는 중간 과정이 이해가 가지 않습니다.

[A]
먼저 3을 7로 나눈 나머지 3%7 = 3 입니다.
그리고 P(s) = 3^s % 7 의 정의에 의해, P(1) = 3^1 % 7 = 3 % 7 = 3 입니다.
따라서 {(3%7)×(3%7)} %7 = {3 × 3} %7 = {3 × P(1)} %7 로 쓸 수 있습니다.

그리고 일반적으로 임의의 정수 n에 대해 P(n) = {3×P(n-1)} %7 이 되는 이유는 아래 등식에 기반합니다.

(a×b)%c = {(a%c)×(b%c)} %c

이 등식의 의미는
“(좌변) a×b를 먼저 수행한 후에 얻은 큰 수에 %c를 취한 것은
(우변) a와 b에 먼저 %c를 취해 작은 수로 만든 후에 이들을 곱한 것과 같다” 입니다.

이 등식을 사용하면 큰 수에 대한 %c 계산을 더 쉽게 할 수 있습니다.
예를 들어 210%13을 손으로 계산해 보면 시간이 좀 걸립니다.
하지만 210=14×15 임을 활용하면 위 등식에 의해
210%13 = (14×15)%13 = {(14%13)×(15%13)} %13 = {1×2} % 13 = 2 로 간단하게 구할 수 있습니다.

이 등식에 의해 임의의 정수 n에 대해
P(n) = 3^n %7 = {3 × 3^(n-1)} %7 = {(3%7) × (3^(n-1)%7)} %7 = {3 × P(n-1)} %7이 됩니다.

[Q] Dynamic Programming을 사용한 알고리즘은 코드로 어떻게 구현하나요?

[A]

구현 방법은 대표적으로 2가지인데, ① 함수의 재귀호출을 사용하거나 ② 룹을 사용해 작은 문제부터 차례로 모두 풀어나가는 방법이 있습니다. 예를 들어 작은 문제와 큰 문제의 해 간에 아래와 같은 관계가 있다고 하겠습니다.

f(n) = f(n-2) + f(n-5), if n>=2
f(1) = 1
f(n) = 0, if n<=0

이를 방법 ①과 ②로 구현한 수도코드는 아래와 같습니다. f(0)~f(k)까지를 모두 계산한다고 가정했습니다.

<table>
<tr><th>① 함수의 재귀호출 사용</th><th>② 룹을 사용해 작은 문제부터 차례로 모두 풀기</th></tr>
<tr><td><pre>
int fmemo[] // 한 번 계산한 값 저장하는 배열
            // 모든 값 -1로 초기화
int f(int n) {
    if (n<=0) return 0
    else if (n==1) return 1
    else if (fmemo[n]>=0) return fmemo[n]
    else {
        fmemo[n] = f(n-2) + f(n-5)
        return fmemo[n]
    }
}

int main() {
    for(int i=0;i<=k;i++) print(f(i) + ‘\n’);
}
</pre></td><td><pre>
int main() {
  int fmemo[]={0,1,0,1,0} //f(0)~f(4) 미리 저장
  for(int i=5;i<=k;i++)
    fmemo[i] = fmemo[i-2] + fmemo[i-5]

  for(int i=0;i<=k;i++)
    printf(fmemo[i] + ‘\n’)
}
</pre></td></tr>
</table>

위 예제에서는 **기존에 계산했던 작은 문제의 답을 여러 차례 필요**로 합니다. 예를 들어 f(10)은 f(12)를 계산할 때도 필요하고, f(15)을 계산할 때도 필요합니다. 따라서 f(10)을 한 번 계산했다면 이를 저장해 두었다가 이후에 이를 다시 필요로 할 때 저장한 값을 꺼내서 재사용한다면, 매번 f(10)을 다시 계산하는 것에 비해 더 효율적일 것입니다. 이처럼 **한 번 계산한 값을 저장해 두었다가 재사용**하기 위해 배열 fmemo[]를 두었음을 유의해 보세요(코드의 **붉은 색** 부분). 이러한 기법을 ‘**memoization**’이라고 합니다.

①과 같이 재귀호출로 구현할 때 memoization 하는 것을 잊어서 했던 계산을 다시 하는 비효율적인 코드가 되는 경우가 많습니다. 이번 시간 실습 문제에서도 코드 제출 시 ‘시간 초과’ 오류가 난다면 memoization을 제대로 했는지 검증해 보세요.

그리고 f(0)~f(k)를 빠짐없이 모두 계산해야 한다면 ①과 ② 모두 괜찮으며, 특히 함수 호출이 적은 ②가 더 효율적일 것입니다. 하지만 f(0)~f(k) 중 일부만을 계산해야 한다면 ①의 방식이 더 효율적입니다. 예를 들어 f(10)만이 필요하다고 하겠습니다. ①의 방식이라면 f(10)의 계산에 필요한 f(8)과 f(5) 등은 계산하겠지만, f(10)의 계산에 불필요한 f(9)를 계산하지는 않을 것입니다. 하지만 ②의 방식은 f(9)도 계산합니다. 이처럼 ①의 방식은 f(k)의 계산에 필요한 작은 문제만 찾아서 풀어준다는 장점이 있습니다.