재귀와 반복 - 동적계획법

Recursion & Iteration - Dynamic Programming

실습 #1 : Minimum steps to one

동적계획법으로 프로그램을 설계하고 구현하는 실습이다.

문제 :

양수 인수 n을 받아서 (가) 1을 빼거나, (나) 2로 나누어지면 2로 나누거나, (다) 3으로 나누어지면 3으로 나누는 과정을 계속하여, n이 1이 되기까지 이 과정의 최소 반복 횟수를 구하는 함수 `minsteps`를 작성하자.

이 문제를 푸는 식은 다음과 같은 재귀로 표현할 수 있다.

```
minsteps(n) = 1 + minimum {minsteps(n-1), minsteps(n/2), minsteps(n/3)}
              when n > 1
minsteps(1) = 0
```

이 식을 푸는 방법으로 가장 쉬운 방법은 가장 빨리 수가 줄어드는 과정을 우선 적용하는 것이다. 즉, 우선순위를 (다), (나), (가) 순으로 두고 우선순위가 높은 것을 먼저 적용시켜보면 직관적으로 최소 반복 횟수를 구할 수 있을 것 같다. 이와 같이 해답을 구하는 방법을 **탐욕적 풀이법**greedy approach라고 한다. 사례를 가지고 직접 해답을 구해보자.

호출	답	계산 절차
`minsteps(1)`	0	1
`minsteps(2)`	1	2 => 1
`minsteps(3)`	1	3 => 1 3 => 2 => 1
`minsteps(4)`	2	4 => 2 => 1 4 => 3 => 1
`minsteps(7)`	3	7 => 6 => 2 => 1 7 => 6 => 3 => 1
`minsteps(10)`	3	10 => 5 => 4 => 2 => 1 10 => 9 => 3 => 1
`minsteps(23)`	6	23 => 22 => 11 => 10 => 5 => 4 => 2 => 1 23 => 22 => 21 => 7 => 6 => 2 => 1
`minsteps(237)`	8	237 => 79 => 78 => 26 => 13 => 12 => 4 => 2 => 1
`minsteps(317)`	10	317 => 316 => 158 => 79 => 78 => 39 => 13 => 12 => 4 => 2 => 1

호출	답	계산 절차
minsteps(514)	8	514 => 257 => 256 => 128 => 64 => 32 => 16 => 8 => 4 => 1 => 1 514 => 513 => 171 => 57 => 19 => 18 => 6 => 2 => 1

위의 사례를 보면 탐욕적 풀이법으로 구한 결과가 모두 답은 아님을 알 수 있다. 10, 23, 514가 그런 경우이다. 사실 탐욕적 풀이법으로 답을 빨리 구할 수는 있지만 항상 정답을 얻는다는 보장이 없다. 따라서 이 경우 정답을 확실히 얻는 유일한 방법은 모든 경우를 다 따져보는 수 밖에 없다.

모든 경우를 다 따져보도록 Python 함수를 작성하면 다음과 같다.

```
1  def minsteps0(n):
2      if n > 1:
3          r = 1 + minsteps0(n - 1)
4          if n % 2 == 0:
5              r = min(r, 1 + minsteps0(n // 2))
6          if n % 3 == 0:
7              r = min(r, 1 + minsteps0(n // 3))
8          return r
9      else:
10         return 0
```

이 함수를 실행해보면, n이 커질수록 답을 구하는데 걸리는 시간이 증가한다. 100 미만의 인수는 바로 답이 나오지만, 100을 넘어서면서 시간이 좀 걸리기 시작한다. 514의 경우 좀 오래 기다려야 답이 나온다. 시간이 오래 걸리는 이유는 엄청나게 많은 양의 중복계산을 하기 때문이다. 중복계산을 피하기 위한 방법은 무엇일까? 한 번 계산한 값은 기록해두고 다음에 필요할 때 가져다 쓰면 된다. 이를 **메모해두기** memoization라고 한다. minsteps(n)을 계산하려면 minsteps(1)부터 minsteps(n-1)까지의 계산결과가 잠재적으로 필요하므로 이를 저장해둘 n개의 공간을 메모 리스트로 준비해두면 된다. 메모 리스트의 이름을 memo라고 한다면, minsteps(1)의 계산 결과는 memo[1], minsteps(2)의 계산 결과는 memo[2], ..., minsteps(n)의 계산 결과는 memo[n]에 저장해둔다. (사실 n개의 공간만 필요하지만, memo의 위치번호와 그 장소에 저장되는 minsteps(n)의 n이 일치하면 프로그램이 간단해져 가독성이 좋아지므로 memo 리스트의 길이를 n+1하고 위치번호 0은 쓰지 않는 걸로 한다.) 메모행렬을 이용하도록 수정한 프로그램은 다음과 같다.

```
def minsteps1(n):
    memo = [0] * (n + 1)
    def loop(n):
        if n > 1:
            if memo[n] != 0:
                return memo[n]
            else:
                memo[n] = 1 + loop(n - 1)
                if n % 2 == 0:
                    memo[n] = min(memo[n], 1 + loop(n // 2))
                if n % 3 == 0:
                    memo[n] = min(memo[n], 1 + loop(n // 3))
                return memo[n]
        else:
            return 0
    return loop(n)
```

이제 이 프로그램을 실행하면 인수가 증가해도 실행시간에 큰 변화가 없이 모두 빠르게 계산한다. 그런데 998 이상의 인수에 대해서는 오류가 발생한다. Python은 재귀함수의 호출을 연속 1,000번까지 밖에 할 수 없기 때문이다. Python에서 재귀함수를 자제해야 하는 이유이다.

이 프로그램을 이해하고 아무리 큰 인수에 대해서도 답을 내줄 수 있도록 `for` 문을 사용하여 `minsteps` 함수를 다름 틀에 맞추어 완성하자.

```
def minsteps(n):
    memo = [0] * (n + 1)
    for
    
    
    
    
    
    return memo[n]
```

실습 #2 : 구구단 출력

중첩된 for 문을 이용하여 프로그램을 작성하는 훈련을 하기 위한 실습 문제이다. 주어진 실행사례와 정확히 똑같이 실행창에 프린트하도록 중첩 for 문으로 프로그램 해야 한다.

(1) 가로전개 구구단

구구단을 가로로 다음과 같이 실행창에 프린트하는 함수 gugudan1()을 만들어보자.

```
2 x 2 =  4  2 x 3 =  6  2 x 4 =  8  2 x 5 = 10
2 x 6 = 12  2 x 7 = 14  2 x 8 = 16  2 x 9 = 18

3 x 2 =  6  3 x 3 =  9  3 x 4 = 12  3 x 5 = 15
3 x 6 = 18  3 x 7 = 21  3 x 8 = 24  3 x 9 = 27

4 x 2 =  8  4 x 3 = 12  4 x 4 = 16  4 x 5 = 20
4 x 6 = 24  4 x 7 = 28  4 x 8 = 32  4 x 9 = 36

5 x 2 = 10  5 x 3 = 15  5 x 4 = 20  5 x 5 = 25
5 x 6 = 30  5 x 7 = 35  5 x 8 = 40  5 x 9 = 45

6 x 2 = 12  6 x 3 = 18  6 x 4 = 24  6 x 5 = 30
6 x 6 = 36  6 x 7 = 42  6 x 8 = 48  6 x 9 = 54

7 x 2 = 14  7 x 3 = 21  7 x 4 = 28  7 x 5 = 35
7 x 6 = 42  7 x 7 = 49  7 x 8 = 56  7 x 9 = 63

8 x 2 = 16  8 x 3 = 24  8 x 4 = 32  8 x 5 = 40
8 x 6 = 48  8 x 7 = 56  8 x 8 = 64  8 x 9 = 72

9 x 2 = 18  9 x 3 = 27  9 x 4 = 36  9 x 5 = 45
9 x 6 = 54  9 x 7 = 63  9 x 8 = 72  9 x 9 = 81
```

(2) 세로전개 구구단

구구단을 세로로 다음과 같이 실행창에 프린트하는 함수 gugudan2()을 만들어보자.

```
2 x 2 =  4  3 x 2 =  6  4 x 2 =  8  5 x 2 = 10
2 x 3 =  6  3 x 3 =  9  4 x 3 = 12  5 x 3 = 15
2 x 4 =  8  3 x 4 = 12  4 x 4 = 16  5 x 4 = 20
2 x 5 = 10  3 x 5 = 15  4 x 5 = 20  5 x 5 = 25
2 x 6 = 12  3 x 6 = 18  4 x 6 = 24  5 x 6 = 30
2 x 7 = 14  3 x 7 = 21  4 x 7 = 28  5 x 7 = 35
2 x 8 = 16  3 x 8 = 24  4 x 8 = 32  5 x 8 = 40
2 x 9 = 18  3 x 9 = 27  4 x 9 = 36  5 x 9 = 45

6 x 2 = 12  7 x 2 = 14  8 x 2 = 16  9 x 2 = 18
6 x 3 = 18  7 x 3 = 21  8 x 3 = 24  9 x 3 = 27
6 x 4 = 24  7 x 4 = 28  8 x 4 = 32  9 x 4 = 36
6 x 5 = 30  7 x 5 = 35  8 x 5 = 40  9 x 5 = 45
6 x 6 = 36  7 x 6 = 42  8 x 6 = 48  9 x 6 = 54
6 x 7 = 42  7 x 7 = 49  8 x 7 = 56  9 x 7 = 63
6 x 8 = 48  7 x 8 = 56  8 x 8 = 64  9 x 8 = 72
6 x 9 = 54  7 x 9 = 63  8 x 9 = 72  9 x 9 = 81
```

힌트

문자열 연산	의미	사례
`s.rjust(n)`	n 자리를 확보한 다음 문자열을 오른쪽에 맞추어 조정하는 메소드	`'h'.rjust(2) => ' h'`

print 속성	의미	사례

print(…, end=<문자열>)	end 속성의 기본값은 '\n'이다.
	기본값이란 지정하지 않아도 저절로 갖고 있는 값을 말한다.
	그래서 프린트를 한 후 언급이 없으면 저절로 줄을 넘긴다.
	이를 바꾸고 싶은 경우 end 속성을 지정하면 된다.
		`>>> print(3); print(4)`
		`3`
		`4`
		`>>> print(3, end=' '); print(4)`
		`3 4`