재귀와 반복 - 셈

Recursion and Iteration - Programming on Numbers

1. 자연수

자연수natural number[1]는 일상 생활에서 셈하거나 순서를 매기는데 쓰는 수로 무한히 많이 있으며, 집합으로 다음과 같이 표현할 수 있다.

$$N = \{ 0, 1, 2, 3, \cdots \}$$

그런데 이렇게 정의하면 자연수가 무엇인지 얼추 추측할 수는 있겠지만 엄밀하게 정의했다고 할 수는 없다. 자연수의 무한 집합은 **귀납법**induction으로 다음과 같이 단 세줄로 엄밀하게 정의할 수 있다

(1)	**기초** basis	0은 자연수이다.
(2)	**귀납** induction	n이 자연수이면 n+1도 자연수이다.
(3)	그 외에 다른 자연수는 없다.	

(1)의 **기초**에 의하여 0은 자연수이다. 그런데 0이 자연수이니 (2)의 **귀납**에 의해서 1도 자연수임을 확인할 수 있다. 이어서 1이 자연수이니 마찬가지로 (2)의 **귀납**에 의해서 2도 자연수임을 확인할 수 있다. 그런데 2가 자연수이니 마찬가지로 (2)의 **귀납**에 의해서 3도 자연수이다. 이런 식으로 계속하면, 아무리 큰 자연수라도 시간만 충분히 주면 자연수임을 확인할 수 있다. 그리고 (1)과(2)로 확인하지 못하는 자연수는 없다고 (3)에서 못을 박아 불순물의 포함을 원천적으로 차단한다.

자연수 n을 귀납(n>0)과 기초(n=0) 부분으로 나누어 사고하여, 자연수를 다루는 함수를 **재귀 함수** recursive function로 간단히 표현할 수 있는 경우가 많다. 이 장에서는 자연수를 셈하는 문제를 푸는 프로그램을 재귀구조recursion와 반복구조iteration를 사용하여 작성하는 방법을 공부하고, 이 두 대조적인 구조의 차이점을 학습하여 프로그램 반복구조의 원리를 익힌다. 아울러 시간/공간 효율을 높여 프로그램의 성능을 향상시킬 수 있는 방법을 다양한 사례를 통하여 경험해본다.

2. 첫째 사례 : 초 읽기 출력

문제

정수 n을 인수로 받아 n부터 1씩 줄여가면서 화면에 1초에 하나씩 프린트하고, 0이 되면 "발사!"를 프린트하는 프로시저 `countdown`을 만들어보자. 음수 인수는 모두 0으로 취급하여 "발사!"만 프린트 한다.

- 실행 사례

[1] 수학에서는 1보다 크거나 같은 정수를 자연수natural number라고 한다. 자연에 0이라는 수는 없기 때문이다. 그러나 컴퓨터과학에서는 자연수에 0을 포함시킨다. 컴퓨터 메모리에는 0을 저장할 수 있으므로 존재한다고 본다.

```
>>> countdown(3)
3
2
1
발사!
```

재귀

자연수의 귀납구조를 사용하여 countdown(n) 실행을 다음과 같이 **재귀**recursion로 정의할 수 있다.

countdown(n)		명령
귀납 – **반복조건**	n > 0	인수 n을 프린트하고 1초 쉬고 countdown(n-1) 실행
기초 – **종료조건**	n <= 0	print("발사!")

이 재귀 정의를 Python 재귀 함수로 구현하면 다음과 같다.

```
1 import time
2 def countdown(n):
3     if n > 0:
4         print(n)
5         time.sleep(1)
6         countdown(n-1)
7     else:
8         print("발사!")
```

time.sleep(1)은 1초동안 실행을 멈추고 쉬라는 명령이다. 이 명령은 time 모듈에 붙박이로 정의되어 있는데, 이 명령을 사용하려면 소속 모듈을 가져다 쓰겠다고 미리 명시해두어야 한다. 줄 1의 import time 이 바로 그것이다.

임의의 k에 대해서, countdown(k)를 호출하면, n > 0인 경우 자신을 다시 호출하는데 이를 **재귀호출** recursive call이라고 한다. 자신을 다시 호출하기는 하지만 countdown(k-1)과 같이 1만큼 감소한 인수로 호출한다. 따라서 재귀호출을 계속 반복하다보면 언젠가는 인수가 0이 되어 더 이상 재귀호출을 하지 않고 프로그램이 결국은 작업을 마치고 멈추게 된다.

재귀 호출이 어떻게 작동하는지 다음과 같은 요령으로 실행추적을 해보면 쉽게 이해할 수 있다. 여기서 화살표 =>는 계산 한 단계 진행됨을 나타낸다. 즉, countdown(3)을 호출하면, 다음 계산 단계는 countdown의 형식파라미터인 n의 값이 3이므로 countdown 함수의 몸체 3~8 전체에서 변수 n을 모두 3으로 바꾼 명령을 수행하는 것과 같다고 할 수 있다. 같은 방식으로 화살표를 한 단계씩 차례로 따라가면서 계산이 어떻게 수행되는지 이해해보자.

```
countdown(3)
=> if 3 > 0 : print(3); time.sleep(1); countdown(3-1) else : print("발사!")
=> print(3); time.sleep(1); countdown(2)
=> countdown(2)
=> if 2 > 0 : print(2); time.sleep(1); countdown(2-1) else : print("발사!")
=> print(2); time.sleep(1); countdown(1)
=> countdown(1)
```

```
=> if 1 > 0 : print(1); time.sleep(1); countdown(1-1) else : print("발사!")
=> print(1); time.sleep(1); countdown(0)
=> countdown(0)
=> if 0 > 0 : print(0); time.sleep(1); countdown(1-0) else : print("발사!")
=> print("발사!")
```

재귀 호출을 할 때마다 인수가 1씩 감소하고 궁극적으로 0에 도달하여 호출이 종료된다.

반복

같은 문제를 종료조건이 만족할 때까지 작업을 반복iteration하는 방식으로 사고하여 풀 수도 있다. 알고리즘은 다음과 같다.

- 인수가 양수인 동안 그 수를 프린트하고 1씩 줄이는 과정을 반복하다, 수가 0이되면 멈추고 발사!를 프린트한다.

while 문을 사용하여 코딩하면 다음과 같다.

```
import time
def countdown(n):
    while n > 0:
        print(n)
        time.sleep(1)
        n = n - 1
    print("발사!")
```

이 함수 countdown(k)를 호출하면, k를 프린트하고 1초 쉬고 k의 값을 1씩 감소하는 과정을 k가 양수인 한 계속 반복한다. k가 감소하므로 언젠가 0에 도달하여 발사!를 프린트하고 함수 실행이 종료한다.

비교

재귀로 작성한 countdown과 반복으로 작성한 countdown의 유사성을 관찰해보자. 어떤 함수가 더 이해하기 쉽고 자연스러운가? 재귀인가? 반복인가? 재귀호출로 해결하는 풀이법을 하향식top-down 풀이법이라고 하고, while 문으로 반복하여 해결하는 풀이법을 상향식bottom-up 풀이법이라고 한다. 반복 계산을 본질적으로 이해하기 위해서 아래의 재귀 버전과 반복 버전을 잘 살펴보고 이 두 사고법을 이용한 상이한 반복구조를 모두 이해하도록 하자.

<table>
<tr><th>재귀 / 하향식</th><th>반복 / 상향식</th></tr>
<tr><td><pre>import time
def countdown(n) :
    if n > 0 :
        print(n)
        time.sleep(1)
        countdown(n-1)
    else :
        print("발사!")</pre></td><td><pre>import time
def countdown(n):
    while n > 0 :
        print(n)
        time.sleep(1)
        n = n - 1
    print("발사!")</pre></td></tr>
</table>

countdown(n)을 실행하면 n을 프린트하고 1초 쉬고 countdown(n-1)을 실행한다. countdown(0)이면 "발사!"를 프린트 한다.	n을 프린트 하고 1초 쉬고 n을 1만큼 감소하는 작업을 n이 양수인 동안 계속 실행하다가, n이 0이 되면 "발사!"를 프린트 한다.

연습문제

자연수 n을 인수로 받아 n부터 1씩 줄여가면서 화면에 1초에 하나씩 프린트하고, 0이 되면 "발사!"를 프린트하되, n이 10 미만이 되면서 부터 n이 짝수일 때 n 대신 "발사임박!"을 프린트하는 함수 countdown2를 만드시오. 음수 인수는 모두 0으로 취급합니다. 먼저 재귀함수 버전을 만들고, while 반복문을 사용한 버전도 만들어보자.

3. 둘째 사례 : 1부터 n까지 자연수 합 계산

문제

1부터 n까지 자연수 합은 다음 공식으로 덧셈, 곱셈, 나눗셈을 한 번씩만 사용하여 쉽게 구할 수 있다.

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

그렇지만 재귀와 반복 구조를 이해할 목적으로, 덧셈만 가지고 1부터 n까지 합을 구하여 내주는 함수 sigma(n)을 만들어보자. 음수 인수는 모두 0으로 취급하여 0을 내주는 걸로 한다.

- 실행 사례

```
>>> sigma(5)
15
>>> sigma(0)
0
>>> sigma(-3)
0
```

재귀

자연수의 귀납구조를 사용하여 sigma(n)의 답을 다음과 같이 재귀recursion로 정의한다.

sigma(n)		결과
귀납 – 반복조건	n > 0	n + sigma(n-1)
기초 – 종료조건	n <= 0	0

이 함수를 재귀함수로 작성하면 다음과 같다.

```
def sigma(n):
    if n > 0:
        return n + sigma(n-1)
    else:
        return 0
```

sigma(5) 호출을 실행추적 해보자.

```
sigma(5)
=> if 5 > 0 : return 5 + sigma(5-1) else : return 0
=> 5 + sigma(4)
=> 5 + if 4 > 0 : return 4 + sigma(4-1) else : return 0
=> 5 + 4 + sigma(3)
=> 5 + 4 + if 3 > 0 : return 3 + sigma(3-1) else : return 0
=> 5 + 4 + 3 + sigma(2)
=> 5 + 4 + 3 + if 2 > 0 : return 2 + sigma(2-1) else : return 0
=> 5 + 4 + 3 + 2 + sigma(1)
=> 5 + 4 + 3 + 2 + if 1 > 0 : return 1 + sigma(1-1) else : return 0
=> 5 + 4 + 3 + 2 + 1 + sigma(0)
=> 5 + 4 + 3 + 2 + 1 + if 0 > 0 : return 0 + sigma(0-1) else : return 0
=> 5 + 4 + 3 + 2 + 1 + 0
=> 5 + 4 + 3 + 2 + 1
=> 5 + 4 + 3 + 3
=> 5 + 4 + 6
=> 5 + 10
=> 15
```

sigma(4)를 호출하면 바로 이어서 sigma(3)을 호출하고 또 바로 이어서 sigma(2)를 호출하고, 이와 같은 재귀 호출이 sigma(0)을 호출할 때 까지 계속된다. 덧셈을 하려면 양쪽 인수가 모두 있어야 하는데 왼쪽 인수는 바로 알지만 오른쪽 인수를 구하려고 계속 재귀 호출을 할 수 밖에 없다. 오른쪽 인수는 인수가 0이 되어야 비로소 알게 되어, 그동안 알지 못하여 계산할 수 없어서 남겨두었던 덧셈 연산을 역순으로 하여 최종 결과값을 얻게 된다.

계산비용 분석

- **시간** : 답을 구하는데 걸리는 계산시간은 함수를 재귀호출하는 횟수(= 덧셈의 횟수)에 비례한다. 인수가 n이면, 재귀호출을 총 n번 (=덧셈을 총 n번)하므로, 계산시간은 인수 n에 비례한다.
- **공간** : 답을 구하는데 필요한 공간은 재귀 함수를 호출하는 횟수에 비례한다. 재귀호출 할 때마다 답을 구해온 뒤에 더해야 할 수를 기억해두어야 하기 때문이다. 인수가 n이면 재귀 호출을 총 n번 하므로 필요 공간은 인수 n에 비례한다.

꼬리 재귀

앞에서 공부한 합 계산 함수는 재귀호출로 계산한 결과에 더할 수를 기억해두어야 하므로 재귀호출의 횟수만큼 추가 저장 공간이 필요하다. 만약 재귀호출 계산이 끝나고 돌아와서 더 이상 할 계산이 없도록 하면 기억해둘 것이 없으므로 이러한 추가 공간이 필요 없을 것이다. 재귀호출을 할 때 더 이상 기억해둘 것이 없도록 하는 (즉, 재귀함수 호출 결과를 가지고 계산할 것이 남아있지 않은) 재귀 함수를 꼬리재귀 tail recursion 함수라고 한다.

합계 계산 함수도 꼬리재귀 함수로 만들 수 있다. 계산을 남겨두지 않기 위해서 덧셈을 미리 해버리고 그 결과 값을 추가 인수로 가지고 다니도록 다음과 같이 재귀함수 loop을 만든다.

```
1 def sigma1(n):
2     return loop(n,0)
3
4 def loop(n,sum):
5     if n > 0:
6         return loop(n-1,n+sum)
7     else:
8         return sum
```

여기서 중간 계산 결과를 전달하기 위해 인수가 하나 더 필요하므로 보조 함수 loop을 사용하였다. 이 보조함수 loop의 첫째 파라미터 n은 재귀호출의 종료를 제어하기 위한 카운터counter 역할을 하고 (재귀호출할 때마다 1씩 감소) 둘째 파라미터 sum은 중간 계산 결과를 누적하는 누적기accumulator 역할을 한다.

2번에서 최초 loop 함수 호출에서 둘째 인수의 시작 값은 0으로 한다. (어떤 수에다 0을 더해도 자신이 되므로 덧셈의 기본값은 0이다.) loop을 재귀호출 할 때마다 (더할 값을 기억하는 대신) 둘째 인수에 더해서 누적해 나가며, 첫째 인수는 1씩 감소해 나간다. n이 0이 되어 종료조건을 만족하는 시점이 되면, 그동안의 합은 둘째 변수에 누적되어 있으므로 둘째 인수 값 sum을 결과로 바로 내주면 된다.

sigma1(5) 호출을 실행추적 해보자.

```
sigma1(5)
=> loop(5,0)
=> if 5 > 0 : return loop(5-1,5+0) else : return 0
=> loop(4,5)
=> if 4 > 0 : return loop(4-1,4+5) else : return 5
=> loop(3,9)
=> if 3 > 0 : return loop(3-1,3+9) else : return 9
=> loop(2,12)
=> if 2 > 0 : return loop(2-1,2+12) else : return 12
=> loop(1,14)
=> if 1 > 0 : return loop(1-1,1+14) else : return 14
=> loop(0,15)
=> if 0 > 0 : return loop(0-1,0+15) else : return 15
=> 15
```

이렇게 꼬리재귀 형태로 재귀호출을 하면 더 이상 더할 인수를 저장해 놓을 필요가 없어서 공간을 절약할 수 있다.

꼬리재귀의 계산비용 분석

- 시간 : 답을 구하는데 걸리는 계산시간은 함수를 호출하는 횟수에 비례한다. 인수가 n이면, 호출을 총 n+1번 하므로, 계산시간은 인수 n에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀함수를 호출하는 횟수에 상관없이 일정하다.

일반재귀와 꼬리재귀의 비교

일반재귀

```
def sigma(n):
    if n > 0:
        return n + sigma(n-1)
    else:
        return 0
```

꼬리재귀

```
def sigma1(n):
    return loop(n,0)

def loop(n,sum):
    if n > 0:
        return loop(n-1,n+sum)
    else:
        return sum
```

하향식으로 답을 계산하는 재귀 함수는 간단하고 코딩하기 쉬운 반면, 재귀호출이 꼬리재귀형이 아니면 공간을 많이 써서 공간효율이 떨어질 수 있다. 답을 상향식으로 모아가는 꼬리재귀 함수로 변환하면 공간비효율이 사라진다. 일반 재귀 함수는 대부분 기계적으로 꼬리재귀형으로 변환할 수 있다.

재귀 함수를 하향식으로 작성한 뒤 꼬리재귀로 변환하는 연습을 많이해보고 익숙해지도록 하자.

보조 함수의 지역화

위의 loop 함수는 sigma1에서만 호출하는 함수이므로 다음과 같이 내부로 넣어 외부로부터 감추면 좋다.

```
def sigma1(n):
    def loop(n,sum):
        if n > 0:
            return loop(n-1,n+sum)
        else:
            return sum
    return loop(n,0)
```

이와 같이 내부로 감추면 loop 함수는 sigma1의 내부에서만 호출 가능하며 외부에서는 보이지 않으므로 호출 불가능하다. 이렇게 내부용으로만 정의한 함수를 지역함수local function라고 한다. 즉, sigma1 함수는 loop을 지역함수로 만들어 외부에서 보이지 않게 감추었다. 이를 캡슐화encapsulation라고 한다. 지역함수를 만들어 캡슐화를 하면 어떤 장점이 있을까? 생각해보고 학우와 토론해보자.

반복

합 계산 함수를 while 반복문으로 작성하려면 어떻게 할까? 합을 누적해놓을 변수 sum을 하나 만들어 0으로 초기값은 0으로 놓는다. 그리고 n부터 시작하여 sum 변수에 더하여 누적하고 n을 1 감소하는 작업을 반복하다가 n이 0이되면 반복을 종료하도록 하면 된다. 프로그램은 다음과 같다.

```
def sigma2(n):
    sum = 0
    while n > 0:
        sum = n + sum
        n = n - 1
    return sum
```

~~이 프로그램의 실행의미는 강의 비디오를 참고하자.~~

비교 관찰

꼬리재귀 함수와 반복 함수를 비교하여 유사 패턴을 발견할 수 있는지 관찰해보자.

꼬리재귀	while 반복

def sigma1(n):
    def loop(n,sum):
        if n > 0:
            return loop(n-1,n+sum)
        else:
            return sum
    return loop(n,0)
	def sigma2(n):
	    sum = 0
	    while n > 0:
	        sum = n + sum
	        n = n - 1
	    return sum

정리

하향식으로 작동하는 재귀 함수는 사고하여 코딩하기 쉬운 반면, 재귀호출이 꼬리재귀형이 아니면 공간효율이 떨어질 수 있다. 상향식으로 작동하는 꼬리재귀형이나 반복문 버전은 공간비효율이 없어졌다. 일반 재귀 함수를 대부분 꼬리재귀형으로 기계적으로 변환할 수 있다. 꼬리재귀형 함수를 반복문으로 변환하는 것도 기계적이다. 따라서 사고하기 상대적으로 쉬운 하향식으로 프로그램을 작성하고, 꼬리재귀 또는 반복문 버전으로 변환하여 효율성을 향상시키는 건 권장할만한 코딩 습관이다.

연습문제 A

자연수 m과 n을 인수로 받아 m부터 n까지의 합을 계산하여 대주는 함수 sumrange를 만들어보자. m > n인 경우에는 결과는 0으로 한다.

- 실행 사례

```
>>> sumrange(3,2)
0
>>> sumrange(3,3)
3
>>> summation(3,4)
7
```

```
>>> summation(3,6)
18
>>> summation(1,10)
55
>>> summation(-5,10)
40
>>> summation(-5,-2)
-14
```

먼저 재귀함수로 만들어보면 다음과 같다.

```
def sumrange(m,n):
    if m <= n:
        return m + sumrange(m+1,n)
    else:
        return 0
```

위의 실행사례를 가지고 실행추적을 하면서 이 재귀함수를 이해해보자.

이제 이 재귀함수를 꼬리재귀로 변환해보자. 아래 코드 틀에서 밑줄 친 부분을 메꾸어보자.

```
def sumrange(m,n):
    def loop(m,n,sum):
        if m <= n:
            return loop(m+1,n,_____)
        else:
            return _____
    return loop(m,n,___)
```

완성한 꼬리재귀 함수를 참고하여, 다음의 while 반복문을 완성해보자.

```
def sumrange(m,n):
    sum = __
    while m <= n:
        sum = ________
        m = _______
    return sum
```

연습문제 B

n의 계승factorial은 1부터 n까지 자연수 곱으로 n!로 다음과 같이 표현한다.

$$n! = \prod_{k=1}^{n} k$$

이를 재귀로 정의하면 다음과 같다.

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

이 재귀정의에 따르면 0! = 1 이다. 곱할 수가 없는 경우 1로 보자는 뜻으로 이해하면 된다. 이 재귀정의를 그대로 활용하여 일반 재귀함수 fac(n)을 작성하면 다음과 같다.

```
def fac(n):
    if n > 0:
        return fac(n-1) * n
    else:
        return 1
```

이 함수를 꼬리 재귀형으로 변환한 fac1 함수를 먼저 작성하고, 이를 참고하여 while 반복을 사용한 fac2 함수를 작성하자.

4. 셋째 사례 : b^n 계산

문제

b와 n이 모두 정수 또는 부동소수점(실수)일 때 Python의 ** 연산자를 b**n과 같이 사용하거나, pow 내장함수를 pow(b,n)과 같이 사용하여 b^n을 계산할 수 있다. 그런데 재귀함수와 반복문도의 작동원리를 이해하기 위한 목적으로 b는 정수, n은 자연수로 제한하고, b의 n승인 b^n을 계산하여 내주는 함수 power를 별도로 만들어보자. 음수인 n은 모두 0으로 취급하기로 한다.

- 실행 사례

```
>>> power(2,5)
32
>>> power(-3,7)
-2187
>>> power(123,0)
1
>>> power(123,-3)
1
```

풀이 알고리즘 1

b^n을 자연수 n의 귀납정의를 이용하여 다음과 같이 재귀로 정의할 수 있다.

$$b^0 = 1$$

$$b^n = b \times b^{n-1} \qquad (n > 0)$$

이 함수를 Python으로 작성하면 다음과 같다.

```
def power(b,n):
    if n > 0:
        return b * power(b,n-1)
    else:
        return 1
```

power(2,5) 호출을 실행추적 해보자.

```
power(2,5)
=> if 5 > 0 : return 2 * power(2,5-1) else : return 1
=> 2 * power(2,4)
=> 2 * if 4 > 0 : return 2 * power(2,4-1) else : return 1
=> 2 * 2 * power(2,3)
=> 2 * 2 * if 3 > 0 : return 2 * power(2,3-1) else : return 1
=> 2 * 2 * 2 * power(2,2)
=> 2 * 2 * 2 * if 2 > 0 : return 2 * power(2,2-1) else : return 1
=> 2 * 2 * 2 * 2 * power(2,1)
=> 2 * 2 * 2 * 2 * if 1 > 0 : return 2 * power(2,1-1) else : return 1
=> 2 * 2 * 2 * 2 * 2 * power(2,0)
=> 2 * 2 * 2 * 2 * 2 * if 0 > 0 : return 2 * power(2,0-1) else : return 1
=> 2 * 2 * 2 * 2 * 2 * 1
=> 2 * 2 * 2 * 2 * 2
=> 2 * 2 * 2 * 4
=> 2 * 2 * 8
=> 2 * 16
=> 32
```

계산비용 분석

- 시간 : 답을 구하는데 걸리는 계산시간은 함수를 호출하는 횟수에 비례한다. 두번째 인수가 n이면, 호출을 총 n+1번 하므로, 계산시간은 인수 n에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀함수를 호출하는 횟수에 비례한다. 재귀호출 할 때마다 답을 구해온 뒤에 곱해야 할 수를 저장해두어야 하기 때문이다. 두번째 인수가 n이면 재귀 호출을 총 n번 하므로 필요 공간은 인수 n에 비례한다.

꼬리 재귀

위의 재귀함수는 꼬리재귀가 아니다. 다음과 같이 꼬리재귀 형태로 변환할 수 있다.

```python
def power(b,n):
    def loop(b,n,prod):
        if n > 0:
            return loop(b,n-1,b*prod)
        else:
            return prod
    return loop(b,n,1)
```

여기서 loop 함수의 둘째 파라미터 n는 재귀호출의 종료를 제어하기 위한 카운터 역할을 하고 셋째 파라미터 prod는 계산 결과를 축적해나가는 누적기 역할을 한다. 그런데 첫째 파라미터는 변하지 않고 항상 참조가 가능하므로 내부 loop 함수가 파라미터로 들고 다닐 필요가 없다. 첫째 파라미터를 제거한 프로그램은 다음과 같다.

```
def power(b,n):
    def loop(n,prod):
        if n > 0:
            return loop(n-1,b*prod)
        else:
            return prod
    return loop(n,1)
```

power(2,5) 호출을 실행추적 해보자.

```
power(2,5)
=> loop(5,1)
=> if 5 > 0 : return loop(5-1,2*1) else : return 1
=> loop(4,2)
=> if 4 > 0 : return loop(4-1,2*2) else : return 2
=> loop(3,4)
=> if 3 > 0 : return loop(3-1,2*4) else : return 4
=> loop(2,8)
=> if 2 > 0 : return loop(2-1,2*8) else : return 8
=> loop(1,16)
=> if 1 > 0 : return loop(1-1,2*16) else : return 16
=> loop(0,32)
=> if 0 > 0 : return loop(0-1,2*32) else : return 32
=> 32
```

계산비용 분석

- 시간 : 답을 구하는데 걸리는 계산시간은 함수를 호출하는 횟수에 비례한다. 인수가 n이면, 호출을 총 n+1번 하므로, 계산시간은 인수 n에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀 함수를 호출하는 횟수에 상관없이 일정하다.

반복

이제 꼬리재귀 함수를 while 반복으로 변환하면 다음과 같다.

```
def power(b,n):
    prod = 1
    while n > 0:
        prod = b * prod
        n = n - 1
    return prod
```

풀이 알고리즘 2 (실행속도 향상)

n이 양의 짝수이면 $b^n = (b^{n/2})^2 = (b^2)^{n/2}$ 과 같은 등식이 성립한다는 수학적 성질을 이용하면 b^n을 다음과 같이 재귀로 정의할 수 있다.

$$
\begin{aligned}
&b^0 = 1 \\
&b^n = (b^2)^{n/2} \quad (n > 0, n \text{ is even}) \\
&b^n = b \times b^{n-1} \quad (n > 0, n \text{ is odd})
\end{aligned}
$$

n이 짝수이면 b^2를 계산한 결과값을 n/2승하고, n이 홀수이면 전과 같은 방법으로 한다. 그러면 n이 짝수일 때 n-1번 곱하는 대신 자승을 n/2-1번 만 곱하므로 곱셈하는 횟수를 거의 절반 절약한다.

위의 정의를 재귀함수로 작성하면 다음과 같다.

```
def fastpower(b,n):
    if n > 0:
        if n % 2 == 0:
            return fastpower(b**2,n//2)
        else:
            return b * fastpower(b,n-1)
    else:
        return 1
```

fastpower(2,7) 호출을 실행추적 해보자[2].

```
fastpower(2,7)
=> 2 * fastpower(2,6)
=> 2 * fastpower(2**2,6//2)
=> 2 * fastpower(4,3)
=> 2 * 4 * fastpower(4,2)
=> 2 * 4 * fastpower(4**2,2//2)
=> 2 * 4 * fastpower(16,1)
=> 2 * 4 * 16 * fastpower(16,0)
=> 2 * 4 * 16 * 1
=> 2 * 4 * 16
=> 2 * 64
=> 128
```

계산비용 분석

- 시간 : 답을 구하는데 걸리는 계산시간은 함수를 호출하는 횟수에 비례한다. 두번째 인수가 n이면, 호출을 약 log n번 하므로, 계산시간은 log n에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀함수를 호출하는 횟수에 비례한다. 두번째 인수가 n이면 재귀 호출을 총 log n번 하므로 필요 공간은 log n에 비례한다.

2 이젠 실행추적의 원리를 어느 정도 이해했을테니 앞으로는 실행추적 과정을 대폭 줄여 표현하기로 한다.

꼬리 재귀

b^n 계산 함수도 꼬리재귀 함수로 다음과 같이 변환 가능하다.

```
def fastpower(b,n):
    def loop(b,n,prod):
        if n > 0:
            if n % 2 == 0:
                return loop(b**2,n//2,prod)
            else:
                return loop(b,n-1,b*prod)
        else:
            return prod
    return loop(b,n,1)
```

fastpower(2,7) 호출을 실행추적 해보자.

```
fastpower(2,7)
=> loop(2,7-1,2*1) == loop(2,6,2)
=> loop(2**2,6//2,2) == loop(4,3,2)
=> loop(4,3-1,4*2) == loop(4,2,8)
=> loop(4**2,2//2,8) == loop(16,1,8)
=> loop(16,1-1,16*8) == loop(16,0,128)
=> 128
```

계산비용 분석

- 시간 : 답을 구하는데 걸리는 계산시간은 함수를 호출하는 횟수에 비례한다. 인수가 n이면, 호출을 약 $\log_2 n$번 하므로, 계산시간은 $\log_2 n$에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀함수를 호출하는 횟수에 상관없이 일정하다.

참고로 n과 $\log_2 n$과의 차이는 다음 비교표를 보면 명확히 알 수 있다. n이 증가하는 속도보다 $\log_2 n$의 증가속도는 훨씬 완만하다.

n	2	4	8	16	32	64	128	256	512	1024	2048	4096	8196	16384	32768	65536
$\log_2 n$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

반복

이 함수를 while 반복으로 변환하면 다음과 같다.

꼬리재귀

```
def fastpower(b,n):
    def loop(b,n,prod):
        if n > 0:
            if n % 2 == 0:
                return loop(b**2,n//2,prod)
            else:
                return loop(b,n-1,b*prod)
        else:
            return prod
    return loop(b,n,1)
```

while 반복

```
def fastpower(b,n):
    prod = 1
    while n > 0:
        if n % 2 == 0:
            b = b**2
            n = n // 2
        else:
            n = n - 1
            prod = b * prod
    return prod
```

계산비용 분석

- 시간 : 반복문으로 답을 구하는데 걸리는 계산시간은 while 문의 반복 횟수에 비례한다. 인수가 n이면, 반복을 약 log n번 하므로, 계산시간은 log n에 비례한다.
- 공간 : 답을 구하는데 필요한 공간은 재귀 함수를 호출하는 횟수에 상관없이 일정하다.

즉, 위의 꼬리재귀 함수와 계산비용이 동일하다.