

CES1017 Programming Fundamentals

Week 5: Recursion and Iteration - Programming on Numbers

Instructor: Eunil Park (pa1324@hanyang.ac.kr)



HANYANG UNIVERSITY

Project

- 조편성: 3인 혹은 4인 1팀
- 프로젝트 주제: 자유 주제, Python 이용한 재미있는 소프트웨어 개발
- 주요 일정
 - 4/23: 프로젝트 계획서 제출 Due date (PPT 5장 이내)
 - 4/30: 프로젝트 계획 발표(3분 예정)
 - 6/11: 프로젝트 시연
- 평가기준
 - 주제의 difficulty, 소프트웨어의 completeness, 각 팀원의 역할(발표 시 명시해야 함), 창의성, 발표 등



Last Class

1. 온도변환 프로그램
 1. 안전코딩
 2. 함수 만들기
2. 문자열 처리와 분리



Before We Start

Inch to Meter

```
def inchmeter():
    stand = str(input("Please enter a type : (i/m)"))
    while not ((stand=='i') or (stand=='m')):
        stand = str(input("Please enter a right type : (i/m)"))
    length = float(input("Please enter a length : "))
    if (stand == 'i'):
        print("It is ", length*0.0254," meter")
    else:
        print("It is ", length/0.0254," inch")

inchmeter()
```

구구단

a = 1

while a < 10:

 count = 1

 while count < 10:

 print(a,"*",count,"=",a*count)

 count = count + 1

 a = a + 1

Before We Start



HANYANG
UNIVERSITY

절대값 비교

```
def compa():
    number_a = float(input("Please Enter the first number : "))
    number_b = float(input("Please Enter the second number : "))
    if((number_a == number_b) or (number_a == -(number_b))):  
        print("There are same absoulte values!")
    else:  
        print("There are not same absoulte values!")

compa()
```

Today's Schedule

1. 자연수
2. 첫째 사례: 초 읽기 출력
3. 둘째 사례: 1부터 n까지 자연수 합 계산
4. 셋째 사례: b^n 계산

01. 자연수

자연수(natural number)

- 집합으로 정의
 $N = \{0, 1, 2, 3, \dots\}$
- 귀납법(induction)으로 정의

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

02. 첫째 사례: 초 읽기 출력



HANYANG
UNIVERSITY

문제

- 프로시저 countdown
 - 인수: 정수 n
 - n부터 1씩 줄여가면서 화면에 1초에 하나씩 프린트
 - 0이 되면 “발사!”를 프린트
 - 음수 인수는 “발사!”를 프린트

>>> countdown(3)

3

2

1

발사!

02. 첫째 사례: 초 읽기 출력



반복(iteration)

- 반복(iteration) 방식으로 구현 가능
- 알고리즘
 - 인수가 양수인 동안 그 수를 프린트하고 1씩 줄이는 과정을 반복한다
 - 수가 0이 되면 멈추고 “발사!”를 프린트한다.
- Python while 반복문으로 구현

```
1 import time
2 def countdown(n):
3     while n > 0:
4         print(n)
5         time.sleep(1) ←———— 1초 동안 쉬라는 명령(sleep)
6         n = n - 1
7         print("Go !")
8
9 countdown(int(input("Insert sec.: ")))
```

02. 첫째 사례: 초 읽기 출력



재귀(recursion)

- 프로시저 countdown(n)의 재귀(recursion)로 정의

countdown(n)		명령
귀납 - 반복조건	$n > 0$	인수 n을 프린트하고 1초 쉬고 <u>countdown(n-1)</u> 실행
기초 - 종료조건	$n \leq 0$	print("발사!")

- Python 재귀함수로 구현

```
1 import time
2 def countdown(n):
3     if n > 0:
4         print(n)
5         time.sleep(1)
6         |
7     else:
8         print("Go !")
9
10    |
```

countdown(int(input("Insert sec.: ")))

재귀호출(recursive call): 자신을 다시 호출

종료!

02. 첫째 사례: 초 읽기 출력

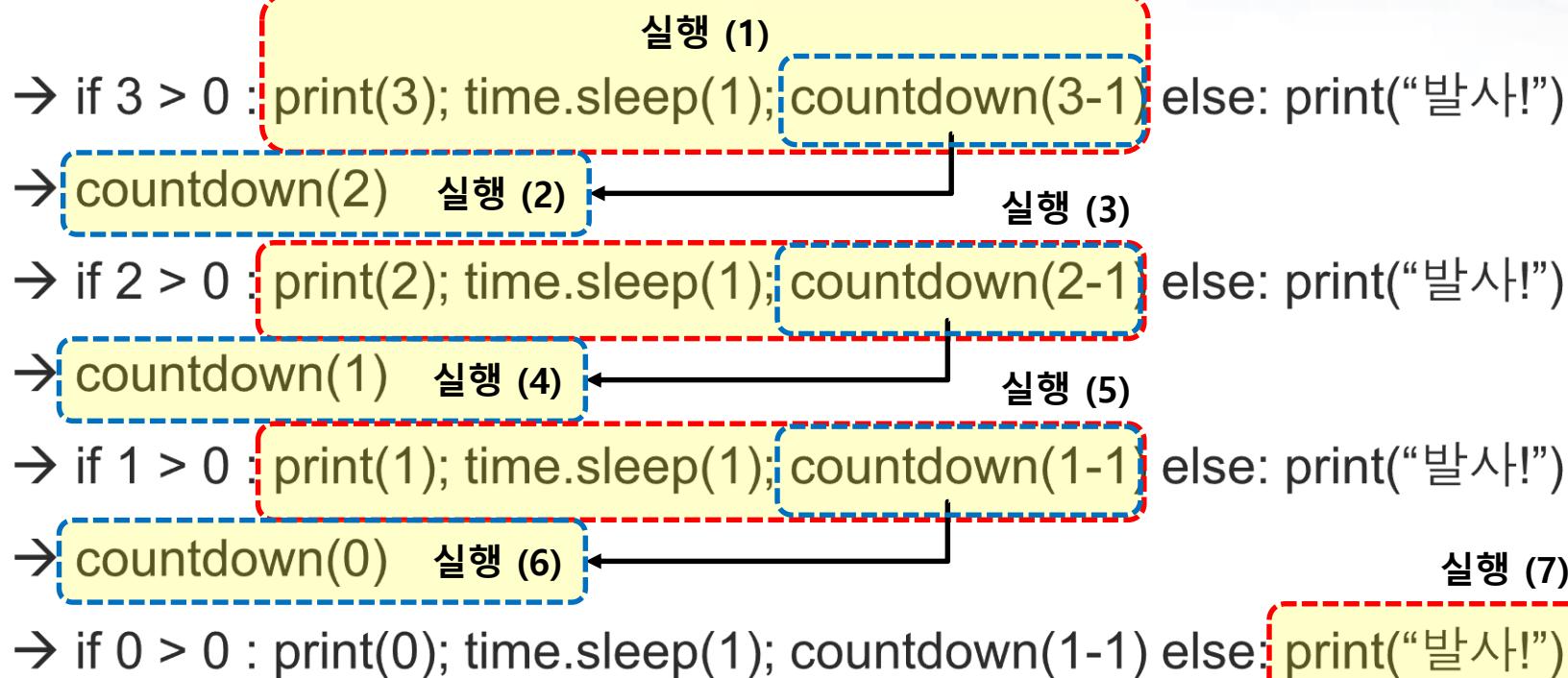
재귀(recursion)

- countdown(3)의 실행 추적

countdown(3)

```

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



02. 첫째 사례: 초 읽기 출력



HANYANG
UNIVERSITY

비교

재귀 / 하향식 (top-down)

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

반복 / 상향식 (bottom-up)

```
1 import time
2 def countdown(n):
3     while n > 0:
4         print(n)
5         time.sleep(1)
6         n = n - 1
7     print("Go !")
```

[재귀방식 countdown(n) 설명]

- n을 프린트하고, 1초 쉬고,
countdown(n-1) 실행
- countdown(0)이면 “발사!” 프린트

[반복방식 countdown(n) 설명]

- n을 프린트하고 1초 쉬고 n을 1만큼
감소하는 작업을 n이 양수인 동안 실행
- n이 0이면 “발사!” 프린트

03. 둘째 사례: 1부터 n까지 자연수 합 계산

문제

- 1부터 n까지 자연수 합을 구하는 **sigma(n)**
- 실행 사례

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

```
>>> sigma(5)
```

```
15
```

```
>>> sigma(0)
```

```
0
```

```
>>> sigma(-3)
```

```
0
```

03. 둘째 사례: 1부터 n까지 자연수 합 계산

재귀(recursion)

- $\text{sigma}(n)$ 의 재귀로 정의

$\text{sigma}(n)$		결과
귀납 - 반복조건	$n > 0$	$n + \text{sigma}(n-1)$
기초 - 종료조건	$n \leq 0$	0

- 재귀 호출 이용한 코드

```

1 def sigma(n):
2     if n > 0:
3         return [REDACTED]
4     else:
5         return 0
  
```

재귀 호출(recursive call):
자신을 다시 호출

03. 둘째 사례: 1부터 n까지 자연수 합 계산

sigma(5) 실행 추적

- sigma(5)
 - if 5 > 0 : return 5 + sigma(5-1) 실행 (1)
 - 5 + sigma(4) 실행 (2)
 - 5 + if 4 > 0 : return 4 + sigma(4-1) 실행 (3)
 - 5 + 4 + sigma(3) 실행 (4)
 - 5 + 4 + if 3 > 0 : return 3 + sigma(3-1) 실행 (5)
 - 5 + 4 + 3 + sigma(2) 실행 (6)
 - 5 + 4 + 3 + if 2 > 0 : return 2 + sigma(2-1) 실행 (7)
 - 5 + 4 + 3 + 2 + sigma(1) 실행 (8)
 - 5 + 4 + 3 + 2 + if 1 > 0 : return 1 + sigma(1-1) 실행 (9)
 - 5 + 4 + 3 + 2 + 1 + sigma(0) 실행 (10)
 - 5 + 4 + 3 + 2 + 1 + if 0 > 0 : return 0 + sigma(0-1) 실행 (11)
 - 5 + 4 + 3 + 2 + 1 + 0 실행 (12)
 - 5 + 4 + 3 + 2 + 1 실행 (13)
 - 5 + 4 + 3 + 3 실행 (14)
 - 5 + 4 + 6 실행 (15)
 - 5 + 10 실행 (16)
 - 15

```

1 def sigma(n):
2     if n > 0:
3         return n + sigma(n-1)
4     else:
5         return 0
  
```

03. 둘째 사례: 1부터 n까지 자연수 합 계산

재귀함수 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수 (=덧셈의 횟수)와 비례
 - 인수가 n 이면 재귀 호출을 총 n 번 (=덧셈을 총 n 번)
 - 즉, 계산 시간은 인수 n 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 비례
 - 재귀 호출 할 때마다 답을 구한 뒤 더해야 할 수를 기억해야 함
 - 인수가 n 이면 재귀 호출을 총 n 번 하기 때문에 필요 공간은 인수 n 에 비례

03. 둘째 사례: 1부터 n까지 자연수 합 계산

꼬리 재귀(tail recursion)

- 꼬리 재귀(tail recursion)
 - 재귀 호출 할 때 더 이상 기억해 둘 것이 없도록 함
 - 즉, 재귀 호출 결과를 가지고 계산할 것이 남아 있지 않음
 - 계산을 남겨두지 않기 때문에 “추가저장공간”이 필요하지 않음
- 꼬리 재귀 함수 만드는 방법
 - 필요한 계산(덧셈)을 미리 하고 그 결과 값을 추가 인수로 가지고 다니도록 함

```

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
  
```

꼬리재귀 함수 sigma1(n)

카운터는 n부터, 누적기는 0부터 시작

중간 계산 결과를 전달하기 위한
보조함수 loop(카운터, 누적기)

loop 호출 시 카운터 1 감소,
누적기에 그때까지의 합 저장

03. 둘째 사례: 1부터 n까지 자연수 합 계산

sigma1(5) 실행 추적

- sigma1(5)

→ loop(5,0)

실행 (1)

→ if 5 > 0 return loop(5-1, 5+0) else : return 0

→ loop(4,5) 실행 (2) ← 실행 (3)

→ if 4 > 0 return loop(4-1, 4+5) else : return 5

→ loop(3,9) 실행 (4) ← 실행 (5)

→ if 3 > 0 return loop(3-1, 3+9) else : return 9

→ loop(2,12) 실행 (6) ← 실행 (7)

→ if 2 > 0 return loop(2-1, 2+12) else : return 12

→ loop(1,14) 실행 (8) ← 실행 (9)

→ if 1 > 0 return loop(1-1, 1+14) else : return 14

→ loop(0,15) 실행 (10) ← 실행 (11)

→ if 0 > 0 : return loop(0-1, 1+15) else : return 15

→ 15 ←

```

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
  
```

03. 둘째 사례: 1부터 n까지 자연수 합 계산



HANYANG
UNIVERSITY

꼬리 재귀 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수와 비례
 - 인수가 n 이면 재귀 호출을 총 $n+1$ 번 이므로,
계산 시간은 인수 n 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 관계없이 일정

03. 둘째 사례: 1부터 n까지 자연수 합 계산

일반재귀 vs 꼬리재귀

[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
```

공간 비효율 향상
일반 재귀 함수는 대부분 꼬리 재귀 함수로 변환 가능

03. 둘째 사례: 1부터 n까지 자연수 합 계산

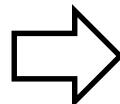


보조 함수의 지역화

- loop 함수를 sigma1 함수 내부안에서만 호출하도록 함
 - 지역 함수(local function)
 - 캡슐화(encapsulation)

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

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



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

03. 둘째 사례: 1부터 n까지 자연수 합 계산



HANYANG
UNIVERSITY

while 반복문

- while 반복문을 이용한 sigma2(n) 함수
 - 합을 누적할 sum 변수 활용
 - n부터 시작하여 sum 변수에 더하고 n을 1씩 감소
 - n이 0이 되면 반복 종료

```
1 def sigma2(n):
```

```
2
```

```
3
```

```
4
```

```
5
```

```
6
```

03. 둘째 사례: 1부터 n까지 자연수 합 계산



HANYANG
UNIVERSITY

꼬리 재귀 vs 반복문

[꼬리재귀]

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

[while반복]

```
def sigma2(n):
```



정리

- 재귀함수(하향식)
 - 직관적이라서 코딩하기 쉬움
 - 꼬리재귀형이 아니면 공간 효율이 떨어질 수 있음
 - 꼬리재귀형 혹은 반복문 (상향식)
 - 공간비효율이 대개 없음
 - 재귀함수 → 꼬리재귀형 함수
 - 꼬리재귀형 함수 → 반복문 함수
- } 보통 기계적으로 변환 가능함

하향식으로 **initial** 프로그래밍

→ 상향식 프로그래밍 방식으로 변환(효율성 향상)

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

문제

- b의 n승인 b^n 을 계산하는 함수 power(b,n)
 - **연산자나 pow 내장함수를 쓰면 간단하게 가능
 - 재귀 함수/반복문 연습을 위해 사용자정의 함수 power(b, n) 구현
- 실행 사례

```
>>> power(2,5)
```

32

```
>>> power(-3,7)
```

-2187

```
>>> power(123,0)
```

1

```
>>> power(123,-3)
```

1

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

재귀 함수

- Power(b, n)의 재귀 정의

$$b^0 = 1$$

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

- 재귀 정의에 기반한 재귀 함수 구현

```
1 def power(b, n):  
2     if n > 0:  
3         [REDACTED]  
4     else:  
5         [REDACTED]
```

04. 셋째 사례: b^n 계산

power(2,5) 실행 추적

- power(2, 5)

실행 (1)

→ if 5 > 0 : return $2 * \text{power}(2, 5-1)$ else : return 1

→ $2 * \text{power}(2, 4)$ 실행 (2) ← 실행 (3)

→ $2 * \text{if } 4 > 0 : \text{return } 2 * \text{power}(2, 4-1)$ else : return 1

→ $2 * [2 * \text{power}(2, 3)]$ 실행 (4) ← 실행 (5)

→ $2 * 2 * \text{if } 3 > 0 : \text{return } 2 * \text{power}(2, 3-1)$ else : return 1

→ $2 * 2 * [2 * \text{power}(2, 2)]$ 실행 (6) ← 실행 (7)

→ $2 * 2 * 2 * \text{if } 2 > 0 : \text{return } 2 * \text{power}(2, 2-1)$ else : return 1

→ $2 * 2 * 2 * [2 * \text{power}(2, 1)]$ 실행 (8) ← 실행 (9)

→ $2 * 2 * 2 * 2 * \text{if } 1 > 0 : \text{return } 2 * \text{power}(2, 1-1)$ else : return 1

→ $2 * 2 * 2 * 2 * [2 * \text{power}(2, 0)]$ 실행 (10) ←

→ $2 * 2 * 2 * 2 * 2 * \text{if } 0 > 0 : \text{return } 2 * \text{power}(2, 0-1)$ else $\text{return } 1$ 실행 (11)

→ $2 * 2 * 2 * 2 * 2 * [2 * 1]$ 실행 (12) ←

→ $2 * 2 * 2 * 2 * 2 * [2 * 2]$ 실행 (13) ←

→ $2 * 2 * 2 * 2 * 2 * [2 * 4]$ 실행 (14) ←

→ $2 * 2 * 2 * 2 * 2 * [2 * 8]$ 실행 (15) ←

→ $2 * 2 * 2 * 2 * 2 * [2 * 16]$ 실행 (16) ←

→ 32

```
def power(b, n):
```

```
    if n > 0:
```



```
    else:
```



04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

재귀 함수 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수와 비례
 - 인수가 n 이면 재귀 호출을 총 $n+1$ 번 하기 때문에, 계산 시간은 인수 n 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 비례
 - 재귀 호출 때마다 답을 구한 뒤에 곱해야 할 수를 저장해 두어야 함
 - 필요 공간은 인수 n 에 비례

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

꼬리 재귀

```
1 def power(b, n):  
2     def loop(n, prod):  
3         if n > 0:  
4             [yellow box]  
5         else:  
6             [yellow box]  
7             [yellow box]
```

카운터

계산 결과
누적기

04. 셋째 사례: b^n 계산

power(2,5) 실행 추적

- power(2, 5)

→ loop(5, 1)

실행 (1)

→ if 5 > 0 : return loop(5-1, 2*1) else : return 1

→ loop(4, 2) 실행 (2)

실행 (3)

→ if 4 > 0 : return loop(4-1, 2*2) else : return 2

→ loop(3, 4) 실행 (4)

실행 (5)

→ if 3 > 0 : return loop(3-1, 2*4) else : return 4

→ loop(2, 8) 실행 (6)

실행 (7)

→ if 2 > 0 : return loop(2-1, 2*8) else : return 8

→ loop(1, 16) 실행 (8)

실행 (9)

→ if 1 > 0 : return loop(1-1, 2*16) else return 16

→ loop(0, 32) 실행 (10)

실행 (11)

→ if 0 > 0 : return loop(0-1, 2*32) else return 32

→ 32 실행 (12)

```
def power(b, n):
    def loop(n, prod):
```

if n > 0:

else:

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

꼬리 재귀 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수와 비례
 - 인수가 n 이면 재귀 호출을 총 $n+1$ 번 하기 때문에,
계산 시간은 인수 n 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 관계없이 일정

04. 셋째 사례: b^n 계산



반복 함수로 변환

```
1 def power(b, n):  
2     def loop(n, prod):  
3         if n > 0:  
4             return loop(n-1, b*prod)  
5         else:  
6             return prod  
7     return loop(n, 1)
```



```
1 def power(b, n):
```

```
2  
3  
4  
5  
6
```

04. 셋째 사례: b^n 계산

실행속도 향상

- n 이 짝수일 경우 곱셈하는 횟수를 절약할 수 있음
 - $b^n = (b^{n/2})^2 = (b^2)^{n/2}$ 성질 이용
- b^n 의 재귀 정의 및 재귀 함수 fastpower(b, n)

$$b^0 = 1$$

$$b^n = (b^2)^{n/2} \quad (n > 0, n \text{ is even})$$

$$b^n = b \times b^{n-1} \quad (n > 0, n \text{ is odd})$$

b^2 를 $n/2 - 1$ 곱함

b 를 n 번 곱함

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

%: 나머지 구하기

//: 몫 구하기

04. 셋째 사례: b^n 계산



fastpower(2,7) 실행 추적

- fastpower(2, 7)

→ 2 * fastpower(2, 6)
실행 (1)

→ 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

```
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
```

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

재귀 함수 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수와 비례
 - 인수가 n 이면 재귀 호출을 총 $\log_2 n$ 번 하기 때문에, 계산 시간은 인수 $\log_2 n$ 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 비례
 - 인수가 n 이면 재귀 호출을 총 $\log_2 n$ 번 하기 때문에, 필요한 공간은 인수 $\log_2 n$ 에 비례

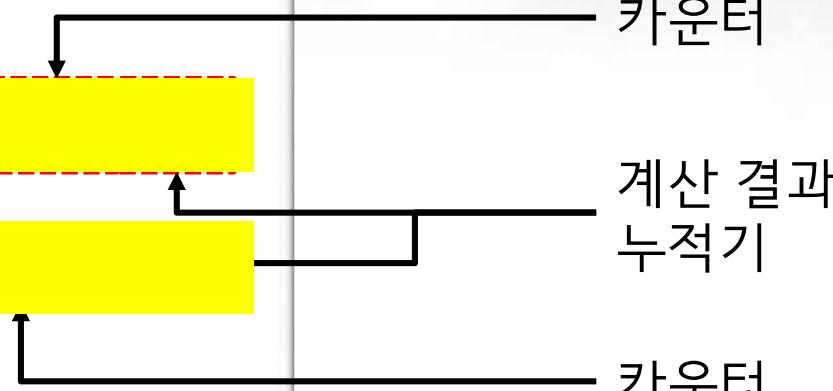
04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

꼬리 재귀

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



04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

fastpower(2,7) 실행 추적

- 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

```
def fastpower(b, n):
    def loop(b, n, prod):
        if n > 0:
            if n % 2 == 0:
                return [REDACTED]
            else:
                return [REDACTED]
        else:
            return prod
    return loop(b, n, 1)
```

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

꼬리 재귀 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: 재귀 호출하는 횟수와 비례
 - 인수가 n 이면 재귀 호출을 총 $\log_2 n$ 번 하기 때문에, 계산 시간은 $\log_2 n$ 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀 호출 횟수에 상관없이 일정
- 참고: n 과 $\log_2 n$ 의 증가속도 비교

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

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

반복 함수로 변환

[꼬리재귀]

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

[while반복]

```
def fastpower(b, n):
```

04. 셋째 사례: b^n 계산



HANYANG
UNIVERSITY

반복함수 계산 비용 분석

- 시간적 측면
 - 답을 구하는데 걸리는 계산 시간: while문의 반복 횟수에 비례
 - 인수가 n이면 $\log_2 n$ 번 반복 하기 때문에, 계산 시간은 $\log_2 n$ 에 비례
- 공간적 측면
 - 답을 구하는데 필요한 공간: 재귀호출 횟수에 상관없이 일정

즉, 꼬리재귀함수와 계산 비용이 동일함

Summary

1. 자연수
2. 첫째 사례: 초 읽기 출력
3. 둘째 사례: 1부터 n까지 자연수 합 계산
4. 셋째 사례: b^n 계산

Thanks

Week 5: Recursion and Iteration - Programming on Numbers
Instructor: Eunil Park (pa1324@hanyang.ac.kr)