[CSE1017]
프로그래밍기초

Recursion and Iteration on Natural Numbers

#04. 재귀와 반복 : 자연수 계산

김현하

한양대학교 ERICA 소프트웨어학부

2024년도 1학기

목차

자연수의 귀납 정의

(1) 기초base : 0은 자연수이다.

(2) 귀납induction : *n* 이 자연수이면 *n+1* 도 자연수이다.

(3) 그 외에 다른 자연수는 없다.

- 귀납 정의 (1)의 기초에 의하여 0은 자연수이다.
- 그런데 0이 자연수이니 (2)의 귀납에 의해서 0 + 1 = 1 도 자연수이다.
- 이어서 1이 자연수이니 마찬가지로 (2)의 귀납에 의해서 1 + 1 = 2도 자연수이다.
- 그런데 2가 자연수이니 마찬가지로 (2)의 귀납에 의해서 2 + 1 = 3도 자연수이다.
- 이런 식으로 계속해서, 무한히 많은 자연수를 시간만 충분히 주면 모두 확인할 수 있다.
- 그리고 (1)과 (2)를 사용하여 만든 수 말고는 자연수가 없음을 (3)에서 못 박는다.

자연수 수열의 합

순진무구 알고리즘

(1) 기초base : 0은 자연수이다.

(2) 귀납induction : *n* 이 자연수이면 *n+1* 도 자연수이다.

(3) 그 외에 다른 자연수는 없다.

$$\text{sigma}(0) = 0 \qquad [base]$$

$$\text{sigma}(n) = n + \text{sigma}(n-1) \qquad [induction]$$

$$\text{sigma}(n) = \begin{cases} n + \text{sigma}(n-1) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

다시쓰기|rewrite

$$\text{sigma}(n) = \begin{cases} n + \text{sigma}(n-1) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

$$\begin{aligned}
\text{sigma}(5) &= 5 + \text{sigma}(4) \\
&= 5 + (4 + \text{sigma}(3)) \\
&= 5 + (4 + (3 + \text{sigma}(2))) \\
&= 5 + (4 + (3 + (2 + \text{sigma}(1)))) \\
&= 5 + (4 + (3 + (2 + (1 + \text{sigma}(0))))) \\
&= 5 + (4 + (3 + (2 + (1 + 0)))) \\
&= 5 + (4 + (3 + (2 + 1))) \\
&= 5 + (4 + (3 + 3)) \\
&= 5 + (4 + 6) \\
&= 5 + 10 \\
&= 15
\end{aligned}$$

순진무구 알고리즘

$$\text{sigma}(n) = \begin{cases} n + \text{sigma}(n-1) & \text{if } n > 0 \\ 0 & \text{if } n = 0 \end{cases}$$

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

재귀recursion 함수 : sigma와 같이 자기 자신을 호출하는 함수

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

함수 호출의 실행 추적

```
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
```

재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- sigma 함수의 계산 복잡도
  - 시간
    - 덧셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 인수가 n 일 때 덧셈을 총 n번(= 재귀 호출을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - 재귀 함수를 호출하는 횟수에 비례 (답을 구해온 뒤에 더해야 할 수를 기억해둘 공간 필요)
    - 인수가 n 일 때 재귀 호출을 총 n번 하므로 필요 공간은 n에 비례

꼬리재귀 함수

- 앞에서 공부한 재귀 함수는 재귀 호출로 계산한 결과에 더할 수를 기억해 두어야 함
- 꼬리재귀tail recursion 함수 : 재귀 호출을 할 때 더이상 기억해둘 것이 없도록 하는 재귀 함수

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

n : 재귀 호출의 종료를 제어하기 위한 카운터counter 역할
total : 계산 결과를 누적하는 누적기accumulator 역할

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

함수 호출의 실행 추적

```
sigma(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
```

꼬리재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- 꼬리재귀 sigma 함수의 계산 복잡도
  - 시간
    - 덧셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 인수가 n 일 때 덧셈을 총 n번(= 재귀 호출을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - **인수의 크기와 상관없이 일정**

보조 함수의 지역화

- loop 함수는 sigma 함수가 전용으로 사용하는 보조 함수 임
- loop 함수를 sigma 함수의 내부에 정의하면 내부용으로만 사용할 수 있음
- 지역함수local function : 함수 내부에 정의하여 호출 가능 범위를 함수 내부로 한정한 함수

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

캡슐화encapsulation : 외부에서 호출할 수 없도록 가려놓음

재귀 함수 vs. 꼬리재귀 함수

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

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

꼬리재귀 함수 → while 루프

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

```
1  def sigma(n):
2      total = 0
3      while n > 0:
4          n, total = n - 1, n + total
5      return total
```

while 루프의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- while 루프로 작성한 sigma의 계산 복잡도
  - 시간
    - 덧셈의 횟수 (= 루프를 반복하는 횟수)에 비례
    - 인수가 n 일 때 덧셈을 총 n번(= 루프 반복을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - **인수의 크기와 상관없이 일정**

재귀 함수 vs. while 루프

재귀 함수

하향식
Top-down

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

while 루프

상향식
Bottom-up

```
1  def sigma(n):
2      total = 0
3      while n > 0:
4          n, total = n - 1, n + total
5      return total
```

지정문의 실행 순서

```
1  def sigma(n):
2      total = 0
3      while n > 0:
4          n = n - 1
5          total = n + total
6      return total
```

```
1  def sigma(n):
2      total = 0
3      while n > 0:
4          total = n + total
5          n = n - 1
6      return total
```

정리

- 재귀 함수 (하향식)
  - \+ 직관적인 실행 논리 표현, 코딩이 쉬움
  - \- 공간 효율이 떨어질 수 있음
  - 상향식인 꼬리재귀 함수나 `while` 루프로 변환해서 공간 절약 가능
- 상대적으로 사고하기 쉬운 하향식으로 작성 후, 꼬리재귀 → `while` 루프로 변환해서 코드를 다듬는 습관 권장

Python과 재귀의 궁합

- Python을 설계/개발한 귀도 반 로섬(Guido van Rossum, 1956~)은 재귀함수의 신봉자가 아닌듯?
- 재귀/꼬리 재귀/`while` 루프로 `sigma`(1000)을 각각 실행
- Python은 꼬리 재귀임에도 불구하고 재귀 호출을 제한
  - `RecursionError: ...`
- Python에서는 `while` 루프나 (후에 배울) `for` 루프를 추천

가우스 알고리즘

$$\begin{aligned} \text{sigma}(n) &= 1 + 2 + \cdots + n \\ +\ \text{sigma}(n) &= n + (n-1) + \cdots + 1 \end{aligned}$$

$$\text{sigma}(n) + \text{sigma}(n) = (n+1) + (n+1) + \cdots + (n+1)$$

$$2 \times \text{sigma}(n) = n \times (n+1)$$

$$\text{sigma}(n) = \frac{n \times (n+1)}{2}$$

가우스 알고리즘

$$\text{sigma}(n) = \frac{n \times (n+1)}{2}$$

```
1  def sigma(n):
2      return n * (n + 1) // 2
```

증명?

수학적 귀납법 : 정리 4.1.1 pp.164-165

거듭제곱

거듭제곱

- `2 ** 5`
- `-3 ** 7`
- `123 ** 0`
- `123 ** -3`

- `pow(2, 5)`
- `pow(-3, 7)`
- `pow(123, 0)`
- `pow(123, -3)`

목표 : b^n 를 구하는 함수를 제작
b는 정수, n은 자연수로 제한(음수는 0으로 취급)

순진무구 알고리즘

$$b^n = \begin{cases} b \times b^{n-1} & \text{if } n > 0 \\ 1 & \text{if } n \leq 0 \end{cases}$$

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

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

함수 호출의 실행 추적

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

재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- power 함수의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 인수가 n 일 때 곱셈을 총 n번(= 재귀 호출을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - 재귀 함수를 호출하는 횟수에 비례 (답을 구해온 뒤에 곱해야 할 수를 기억해둘 공간 필요)
    - 인수가 n 일 때 재귀 호출을 총 n번 하므로 필요 공간은 n에 비례

꼬리재귀 함수

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

꼬리재귀 함수

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

함수 호출의 실행 추적

```
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
```

꼬리재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- 꼬리재귀 power 함수의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 인수가 n 일 때 곱셈을 총 n번(= 재귀 호출을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - **인수의 크기와 상관없이 일정**

꼬리재귀 함수 → while 루프

```
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      prod = 1
3      while n > 0:
4          n, prod = n - 1, b * prod
5      return prod
```

while 루프의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- while 루프로 작성한 power의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 루프를 반복하는 횟수)에 비례
    - 인수가 n 일 때 곱셈을 총 n번(= 루프 반복을 총 n번) 하므로 계산시간은 n에 비례
  - 공간
    - **인수의 크기와 상관없이 일정**

나눠 풀기|divide-and-conquer, 분할정복 알고리즘

power 함수의 시간 기준 계산복잡도를 줄이기

n이 짝수일 때, $b^n = (b^2)^{\frac{n}{2}}$

$$b^n = \begin{cases} (b \times b)^{\frac{n}{2}} & \text{if } n > 0 \ and \ even(n) \\ b \times b^{n-1} & \text{if } n > 0 \ and \ odd(n) \\ 1 & \text{if } n \leq 0 \end{cases}$$

나눠 풀기|divide-and-conquer, 분할정복 알고리즘

$$b^n = \begin{cases} (b \times b)^{\frac{n}{2}} & \text{if } n > 0 \text{ and } even(n) \\ b \times b^{n-1} & \text{if } n > 0 \text{ and } odd(n) \\ 1 & \text{if } n \leq 0 \end{cases}$$

```
1  def power(b, n):
2      if n > 0:
3          if n % 2 == 0:
4              return power(b * b, n // 2)
5          else:
6              return b * power(b, n - 1)
7      else:
8          return 1
```

```
1  def power(b, n):
2      if n > 0:
3          if n % 2 == 0:
4              return power(b * b, n // 2)
5          else:
6              return b * power(b, n - 1)
7      else:
8          return 1
```

함수 호출의 실행 추적

```
power(2, 7)
⇒ 2 * power(2, 6)
⇒ 2 * power(2 * 2, 6 // 2)
⇒ 2 * power(4, 3)
⇒ 2 * 4 * power(4, 2)
⇒ 2 * 4 * power(4 * 4, 2 // 2)
⇒ 2 * 4 * power(16, 1)
⇒ 2 * 4 * 16 * power(16, 0)
⇒ 2 * 4 * 16 * 1
⇒ 2 * 4 * 16
⇒ 2 * 64
⇒ 128
```

재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- `power` 함수의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 둘째 인수 n이 짝수 일 때 인수의 크기가 반으로 작아지므로 호출 횟수를 대략 따져보면 약 $\log_2 n$ 번이 되므로 계산시간은 $\log_2 n$에 비례함
  - 공간
    - 재귀 함수를 호출하는 횟수에 비례 (답을 구해온 뒤에 곱해야 할 수를 기억해둘 공간 필요)
    - 둘째 인수 n이 짝수 일 때 위와 마찬가지로 재귀 호출을 약 $\log_2 n$ 번 하므로 계산시간은 $\log_2 n$에 비례함

꼬리재귀 함수

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

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

함수 호출의 실행 추적

power(2,7)

⇒ loop(2,7,1)

⇒ 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*4,2//2,8) = loop(16,1,8)

⇒ loop(16,1-1,16*8) = loop(16,0,128)

⇒ 128

꼬리재귀 함수의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- 꼬리재귀 `power` 함수의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 재귀 함수를 호출하는 횟수)에 비례
    - 둘째 인수 n이 짝수 일 때 인수의 크기가 반으로 작아지므로 호출 횟수를 대략 따져 보면 약 $\log_2 n$ 번이 되므로 계산시간은 $\log_2 n$에 비례함
  - 공간
    - **인수의 크기와 상관없이 일정**

n vs. $\log_2 n$

n	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768	65536
$\log_2 n$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

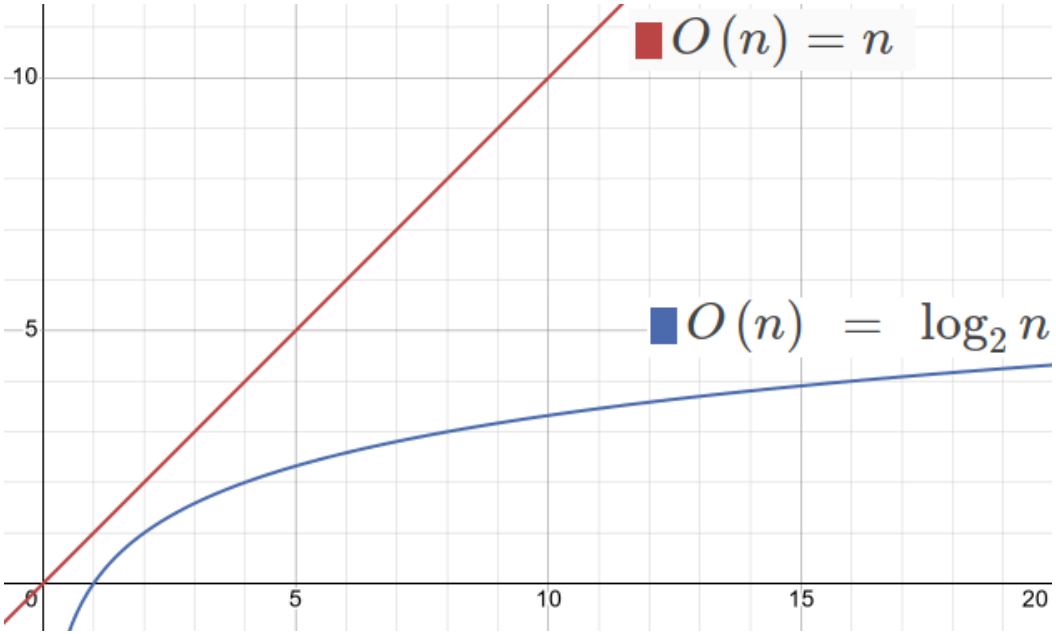

그림 출처 : https://www.kernel.bz/boardPost/118679/15

꼬리재귀 함수 → while 루프

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

```
1  def power(b, n):
2      prod = 1
3      while n > 0:
4          if n % 2 == 0:
5              b = b * b
6              n = n // 2
7          else:
8              n = n - 1
9              prod = b * prod
10     return prod
```

while 루프의 계산비용 분석

- 계산복잡도computational complexity
  - 시간 : 프로그램이 얼마나 빨리 답을 계산하는가?
  - 공간 : 답을 계산하면서 얼마나 많은 공간을 사용하는가?
- while 루프로 작성한 power의 계산 복잡도
  - 시간
    - 곱셈의 횟수 (= 루프를 반복하는 횟수)에 비례
    - 둘째 인수 n이 짝수 일 때 인수의 크기가 반으로 작아지므로 계산시간은 $\log_2 n$에 비례함
  - 공간
    - **인수의 크기와 상관없이 일정**

최대공약수

최대공약수

- 자연수 n의 약수divisor : n을 나누어서 나머지 없이 떨어지는 양수
  - 54의 약수 : 1, 2, 3, 6, 9, 18, 27, 54
  - 24의 약수 : 1, 2, 3, 4, 6, 8, 12, 24
- 두 자연수 m과 n의 공약수common divisor : m의 약수와 n의 약수 중에서 공통이 되는 수
  - 54와 24의 공약수 : 1, 2, 3, 6
- 두 자연수 m과 n의 최대공약수greatest common divisor : m과 n의 공약수 중에서 가장 큰 수
  - 54와 24의 최대공약수 : 6
- 양수 n과 0의 최대공약수는 n (0의 약수는 모든 양수 임)
- 0과 0의 최대공약수 : 따져보면 무한대로 큰 수가 되겠지만 편의상 0으로 정함
- `math.gcd`

유클리드 알고리즘

$$gcd(m,n) = \begin{cases} gcd(n, m \bmod n) & \text{if } n \neq 0 \\ m & \text{if } n = 0 \end{cases}$$

- 48과 18의 최대공약수?
  - 48 % 18 = 12, 18과 12의 최대공약수
- 18과 12의 최대공약수?
  - 18 % 12 = 6 , 12와 6의 최대공약수
- 12와 6의 최대공약수?
  - 12 % 6 = 0, 48과 18의 최대공약수는 6.

유클리드 알고리즘

$$gcd(m,n) = \begin{cases} gcd(n, m \bmod n) & \text{if } n \neq 0 \\ m & \text{if } n = 0 \end{cases}$$

```
1  def gcd(m, n):
2      if n != 0:
3          return gcd(n, m % n)
4      else:
5          return m
```

```
1  def gcd(m, n):
2      if n != 0:
3          return gcd(n, m % n)
4      else:
5          return m
```

함수 호출의 실행 추적

```
   gcd(18, 48)
=> gcd(48, 18%48) == gcd(48, 18)
=> gcd(18, 48%18) == gcd(18, 12)
=> gcd(12, 18%12) == gcd(12, 6)
=> gcd(6, 12%6) == gcd(6, 0)
=> 6
```

분할정복 알고리즘

$$gcd(m,n) = \begin{cases} 2 \times gcd(\frac{m}{2}, \frac{n}{2}) & \text{if } even(m) \text{ and } even(n) \\ gcd(\frac{m}{2}, n) & \text{if } even(m) \text{ and } odd(n) \\ gcd(m, \frac{n}{2}) & \text{if } odd(m) \text{ and } even(n) \\ gcd(m, \frac{n-m}{2}) & \text{if } odd(m) \text{ and } odd(n) \text{ and } m \leq n \\ gcd(n, \frac{m-n}{2}) & \text{if } odd(m) \text{ and } odd(n) \text{ and } m > n \\ n & \text{if } m = 0 \\ m & \text{if } n = 0 \end{cases}$$

$$gcd(m,n) = \begin{cases} 2 \times gcd(\frac{m}{2}, \frac{n}{2}) & \text{if } even(m) \text{ and } even(n) \\ gcd(\frac{m}{2}, n) & \text{if } even(m) \text{ and } odd(n) \\ gcd(m, \frac{n}{2}) & \text{if } odd(m) \text{ and } even(n) \\ gcd(m, \frac{n-m}{2}) & \text{if } odd(m) \text{ and } odd(n) \text{ and } m \leq n \\ gcd(n, \frac{m-n}{2}) & \text{if } odd(m) \text{ and } odd(n) \text{ and } m > n \\ n & \text{if } m = 0 \\ m & \text{if } n = 0 \end{cases}$$

```
1  def even(n):
2      return n % 2 == 0
3
4  def odd(n):
5      return n % 2 == 1
```

```
1  def gcd(m, n):
2      if not (m == 0 or n == 0):
3          if even(m) and even(n):
4              return 2 * gcd(m//2, n//2)
5          elif even(m) and odd(n):
6              return gcd(m//2, n)
7          elif odd(m) and even(n):
8              return gcd(m, n//2)
9          elif m <= n:
10             return gcd(m, (n-m)//2)
11         else:
12             return gcd(n, (m-n)//2)
13     else:
14         if m == 0:
15             return n
16         else:
17             return m
```

```
1  def gcd(m, n):
2      if not (m == 0 or n == 0):
3          if even(m) and even(n):
4              return 2 * gcd(m//2, n//2)
5          elif even(m) and odd(n):
6              return gcd(m//2, n)
7          elif odd(m) and even(n):
8              return gcd(m, n//2)
9          elif m <= n:
10             return gcd(m, (n-m)//2)
11         else:
12             return gcd(n, (m-n)//2)
13     else:
14         if m == 0:
15             return n
16         else:
17             return m
```

함수 호출의 실행 추적

gcd(18,48)

⇒ 2 * gcd(18//2,48//2) == 2 * gcd(9,24)

⇒ 2 * gcd(9,24//2) == 2 * gcd(9,12)

⇒ 2 * gcd(9,12//2) == 2 * gcd(9,6)

⇒ 2 * gcd(9,6//2) == 2 * gcd(9,3)

⇒ 2 * gcd(3,(9-3)//2) == 2 * gcd(3,3)

⇒ 2 * gcd(3,(3-3)//2) == 2 * gcd(3,0)

⇒ 2 * 3

⇒ 6

곱셈

덧셈/뺄셈 알고리즘

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

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

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

함수 호출의 실행 추적

```
mult(3, 4)
⇒ 3 + mult(3, 3)
⇒ 3 + 3 + mult(3, 2)
⇒ 3 + 3 + 3 + mult(3, 1)
⇒ 3 + 3 + 3 + 3 + mult(3, 0)
⇒ 3 + 3 + 3 + 3 + 0
⇒ 3 + 3 + 3 + 3
⇒ 3 + 3 + 6
⇒ 3 + 9
⇒ 12
```

분할정복 알고리즘

n이 짝수일 때, $m \times n = (m + m) \times (n \div 2)$

$$m \times n = \begin{cases} (m+m) \times (n//2) & \text{if } n > 0 \text{ and } n \text{ is even} \\ m + m \times (n-1) & \text{if } n > 0 \text{ and } n \text{ is odd} \\ 0 & \text{if } n = 0 \end{cases}$$

```
1  def fastmult(m, n):
2      if n > 0:
3          if n % 2 == 0: # even(n)
4              return fastmult(m + m, n // 2)
5          else:
6              return m + fastmult(m, n - 1)
7      else:
8          return 0
```

러시아 농부 알고리즘

- 구구단 없이 덧셈, 두배 하기, 반 나누기만 사용
  - 곱할 두 수를 나란히 적는다.
  - 첫째 수는 두배를 하고, 둘째 수는 반으로 나누어 (나머지는 버림) 다음 줄에 나란히 적는다.
  - 이 과정을 둘째 수가 1이 될 때까지 계속한다.
  - 둘째 수가 짝수인 줄은 모두 지운다.
  - 남은 줄의 첫째 수를 모두 더한 값이 답이다.

러시아 농부 알고리즘

- 곱할 두 수를 나란히 적는다.
- 첫째 수는 두배를 하고, 둘째 수는 반으로 나누어 (나머지는 버림) 다음 줄에 나란히 적는다.
- 이 과정을 둘째 수가 1이 될 때까지 계속한다.
- 둘째 수가 짝수인 줄은 모두 지운다.
- 남은 줄의 첫째 수를 모두 더한 값이 답이다.

	~~57~~	~~86~~
	114	43
	228	21
	~~456~~	~~10~~
	912	5
	~~1824~~	~~2~~
+	3648	1
	4902	