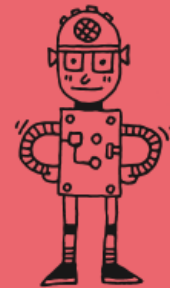


# 모두의 알고리즘 with 파이썬

---

컴퓨팅 사고를 위한 기초 알고리즘



## 둘째 마당 재귀 호출

---

문제04 팩토리얼 구하기

문제05 최대공약수 구하기

문제06 하노이의 탑 옮기기

## 문제 04 팩토리얼 구하기

---

## 문제04 팩토리얼 구하기

0

**1부터  $n$ 까지 연속한 정수의 곱을 구하는 알고리즘을 만들어 보세요.**

- 1부터  $n$ 까지의 곱, 즉 팩토리얼(factorial) 문제

### 1 팩토리얼

#### 팩토리얼

- 1부터  $n$ 까지 연속한 숫자를 차례로 곱한 값, '계승'이라고도 함
- 숫자 뒤에 느낌표(!)를 붙여 표기

$$1! = 1$$

$$3! = 1 \times 2 \times 3 = 6$$

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

단, 0!은 1이라고 약속합니다.

### 1 팩토리얼

#### ■ 프로그램 4-1 팩토리얼을 구하는 알고리즘①

▼ 예제 소스 p04-1-fact.py

```
# 연속한 숫자의 곱을 구하는 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자를 곱한 값

def fact(n):
    f = 1                # 곱을 계산할 변수, 초깃값은 1
    for i in range(1, n + 1): # 1부터 n까지 반복(n+1은 제외)
        f = f * i        # 곱셈 연산으로 수정
    return f
```



## 팩토리얼



```
print(fact(1))  # 1! = 1  
print(fact(5))  # 5! = 120  
print(fact(10)) # 10! = 3628800
```

### ■ 실행 결과

```
1  
120  
3628800
```

## 2 러시아 인형

### 러시아 인형 ‘마트료시카’

- 인형 안에는 자기 자신과 똑같이 생긴, 크기만 약간 작은 인형이 들어 있음
- 인형 안에서 작은 인형이 반복되어 나오다가 인형을 더 작게 만들기 힘들어지면 마지막 인형이 나오면서 반복이 끝남
- 마지막 인형 안에는 사탕이나 초콜릿 같은 작은 상품이 들어 있기도 함



그림 4-1 러시아 인형



### 3 재귀 호출: 다시 돌아가 부르기

- 재귀 호출(再歸呼出, recursion): 어떤 함수 안에서 자기 자신을 부르는 것

```
def hello():  
    print("hello")  
    hello() # hello() 함수 안에서 다시 hello()를 호출  
  
hello() # hello() 함수를 호출
```

- "hello"라는 문장을 화면에 출력한 다음 자기 자신인 hello()를 다시 호출
- "hello"를 출력한 후 다시 자기 자신을 호출하므로 또 다시 "hello"를 출력하고,  
다시 자기 자신을 호출해서 "hello"를 출력하는 과정을 영원히 반복

## 재귀 호출: 다시 돌아가 부르기

```
hello
```

```
hello
```

```
hello
```

```
(...줄임...)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#4>", line 1, in <module>
```

```
    hello()
```

```
  File "<pyshell#3>", line 3, in hello
```

```
    hello()
```

```
  File "<pyshell#3>", line 3, in hello
```

```
    hello()
```

```
  File "<pyshell#3>", line 3, in hello
```

```
    hello()
```

```
[Previous line repeated 974 more times]
```

```
  File "<pyshell#3>", line 2, in hello
```

```
    print("hello")
```

```
RecursionError: maximum recursion depth exceeded while pickling an object
```

### 3 재귀 호출: 다시 돌아가 부르기

- 영원히 hello() 함수를 반복해서 호출하므로 "hello"를 계속 출력하다가 함수 호출에 필요한 기억 장소를 다 써 버리고 나면 에러를 내고 정지  
→ 올바른 재귀 호출 프로그램은 아님(반복을 멈추려면 Ctrl + C)

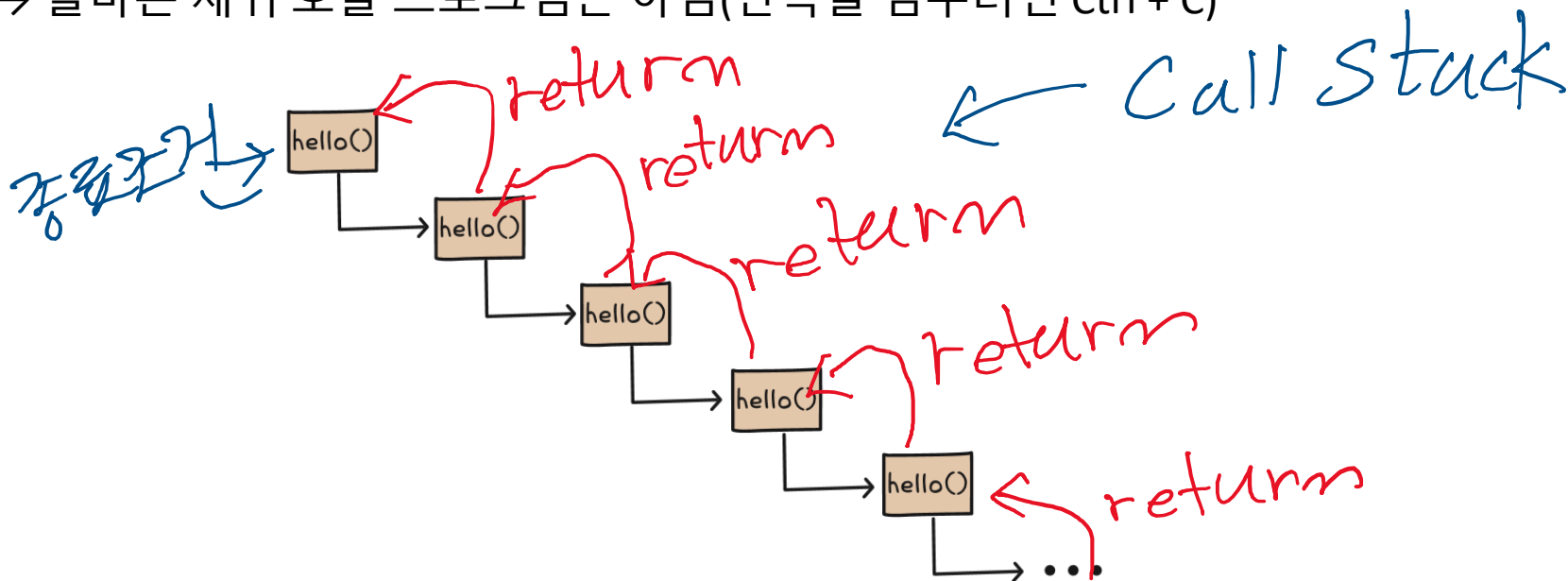


그림 4-2 hello() 함수의 재귀 호출 과정

### 3 재귀 호출: 다시 돌아가 부르기

- 재귀 호출 프로그램이 정상적으로 작동하려면 '종료 조건'이 필요
- 즉, 특정 조건이 되면 더는 자신을 호출하지 않고 멈추도록 설계되어야만 함
- 그렇지 않으면 계속 반복하다가 재귀 에러 발생(RecursionError 발생)
- 재귀 호출 함수가 계산 결과를 돌려줄 때는 return 명령을 사용해서 종료 조건의 결괏값부터 돌려줌
- 종료 조건의 결괏값은 마지막으로 호출된 함수의 결괏값  
→ 마지막 인형 안에 상품으로 들어 있는 사탕과 비슷한 개념

### 4 재귀 호출 알고리즘

- 팩토리얼은 1부터  $n$ 까지 연속한 숫자의 곱
- 팩토리얼을 재귀 호출로 표현하면 다음과 같음

$$1! = 1$$

$$2! = 2 \times 1 = 2 \times 1!$$

$$3! = 3 \times (2 \times 1) = 3 \times 2!$$

$$4! = 4 \times (3 \times 2 \times 1) = 4 \times 3!$$

...

$$n! = n \times (n-1)! \quad \leftarrow \text{팩토리얼을 구하려고 다시 팩토리얼을 구함(재귀적 정의)}$$

- $1! = 1$  그리고  $n! = n \times (n-1)!$ 이라는 팩토리얼의 성질을 이용해서 팩토리얼을 구하는 프로그램을 만들어 보자

## 4 재귀 호출 알고리즘

- 프로그램 4-2 팩토리얼을 구하는 알고리즘②

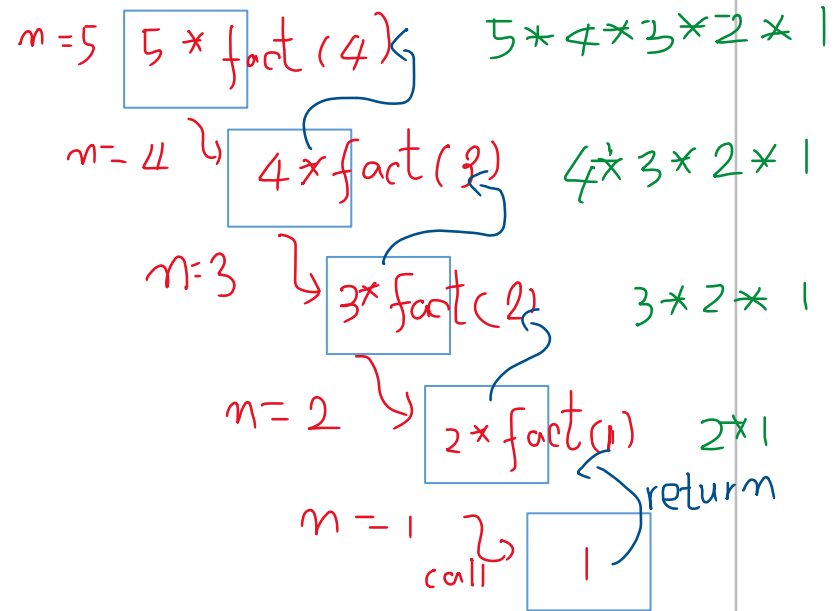
▼ 예제 소스 p04-2-fact.py

```
# 연속한 숫자의 곱을 구하는 알고리즘
# 입력: n
# 출력: 1부터 n까지 연속한 숫자를 곱한 값
```

```
def fact(n):
    if n <= 1:
        return 1
    return n * fact(n - 1)
```

```
print(fact(1)) # 1! = 1
print(fact(5)) # 5! = 120
print(fact(10)) # 10! = 3628800
```

call stack



## 4 재귀 호출 알고리즘

### ■ 실행 결과

```
1
120
3628800
```

- $n$ 이 1 이하인지 비교, 이때 1을 곱값으로 돌려줌 (종료 조건의 결과값)
- 1 이하(0 포함)는 아주 작아서 더는 계산하지 않아도 되는 '종료 조건'
- $n$ 이 1보다 크면  $n! = n \times (n-1)!$ 이므로  $n * \text{fact}(n - 1)$ 을 곱값으로 돌려줌
- 이 과정에서  $n!$ 을 구하기 위해서 약간 더 작은 값인  $(n-1)!$ 을 구하는  $\text{fact}(n-1)$ 이 재귀 호출됨

## 4 재귀 호출 알고리즘

- 호출된  $\text{fact}(n-1)$ 은 어떻게 실행될까?  
→ 다시 종료 조건에 해당하는지 확인
- 종료 조건이 아니라면 이번에는  $\text{fact}(n-2)$ 를 호출
- $\text{fact}(n-3)$ ,  $\text{fact}(n-4)$ ... 이렇게 반복하다 보면 결국  $\text{fact}(1)$ 을 만나게 됨
- 따라서 재귀 호출이 영원히 반복되지 않고 결국 답을 얻게 됨



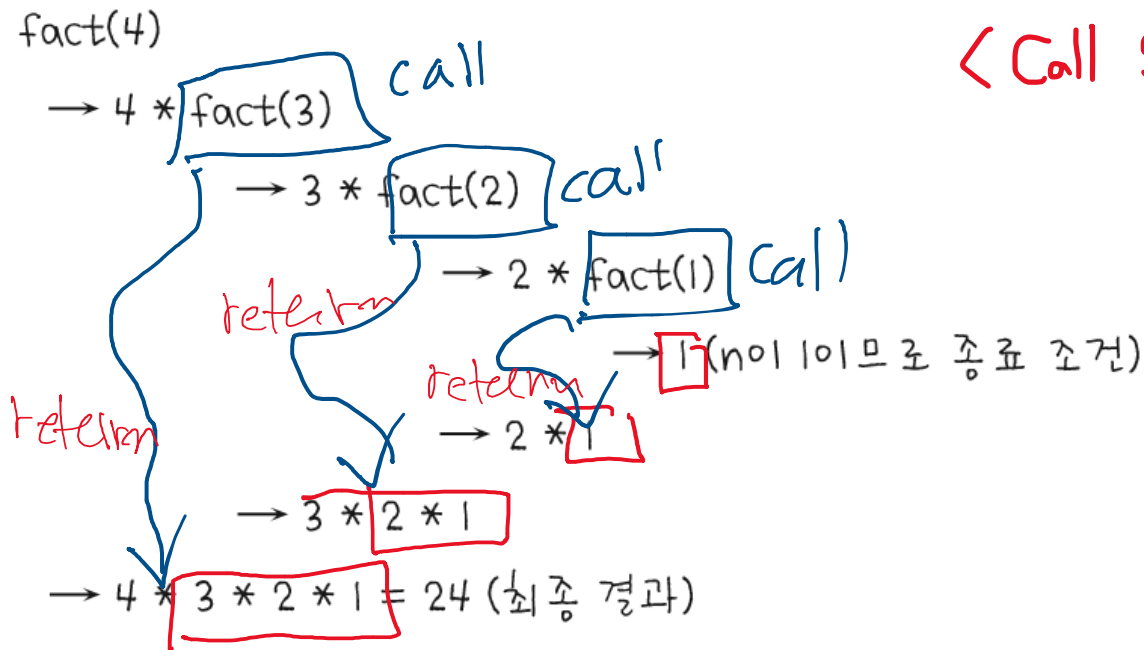
## 4 재귀 호출 알고리즘

■ fact(4)를 호출의 예

- 1) fact(4)는  $4 * \text{fact}(3)$ 이므로 fact(3)을 호출하고
- 2) fact(3)은  $3 * \text{fact}(2)$ 이므로 fact(2)를 호출하고
- 3) fact(2)는  $2 * \text{fact}(1)$ 이므로 fact(1)을 호출
- 4) fact(1)은 종료 조건이므로 fact() 함수를 더 이상 호출하지 않고 1을 돌려줌
- 5) fact(2)는 fact(1)에서 돌려받은 곱값 1에 2를 곱해 2를 돌려주고
- 6) fact(3)은 fact(2)에서 돌려받은 곱값 2에 3을 곱해 6을 돌려주고
- 7) fact(4)는 fact(3)에서 돌려받은 곱값 6에 4를 곱해 24를 돌려줌(최종 결과).

## 4 재귀 호출 알고리즘

- 종이와 연필을 꺼내 여러 번 호출되는 `fact()` 함수에 각각 어떤 값이 입력으로 들어가고 출력으로 반환되는지 직접 4!을 재귀 호출로 계산해 보자



< Call Stack >

## 4 재귀 호출 알고리즘

- 함수 호출을 4! 계산 수식으로 정리

$4!$

$= 4 \times 3!$

$= 4 \times 3 \times 2!$

$= 4 \times 3 \times 2 \times 1!$

$= 4 \times 3 \times 2 \times 1$  (1은 종료 조건이므로 재귀 호출을 멈춤)

$= 4 \times 3 \times 2$

$= 4 \times 6$

$= 24$

## 알고리즘 분석

- 팩토리얼은 연속한 수의 곱  $\rightarrow$  곱셈의 횟수를 기준으로 알고리즘 분석
  - $\text{fact}(4)$ 를 구하려면  $\text{fact}(1)$ 의 종료 조건으로 돌려받은 1을 2와 곱하여 돌려줌
  - 그 값에 다시 3을 곱하여 돌려줌
  - 다시 그 값에 4를 곱하여 돌려주므로 곱셈이 모두 세 번 필요
  - 마찬가지로  $n!$ 을 구하려면 곱셈이 모두  $n-1$ 번 필요
  - 따라서 반복문을 이용한 알고리즘, 재귀 호출을 이용한 알고리즘의 계산 복잡도는 모두  $O(n)$
- 정확한 시간의 복잡도 계산 :

<https://m.blog.naver.com/PostView.naver?isHttpsRedirect=true&blogId=wns7756&logNo=221568348621>

## 알고리즘 분석

- 재귀 호출의 일반적인 형태

```
def func(입력 값):  
    if 입력 값이 충분히 작으면: # 종료 조건  
        return 결괏값  
  
    ...  
    func(더 작은 입력 값) # 더 작은 값으로 자기 자신을 호출  
    ...  
    return 결괏값
```

- 재귀 호출에는 종료 조건이 꼭 필요
- 종료 조건이 없으면 재귀 에러(RecursionError)나 스택 오버플로(Stack Overflow) 등 프로그램 에러가 발생해 비정상적인 동작을 할 수도 있음

### 연습 문제

- 4-1 문제 1의 1부터  $n$ 까지의 합 구하기를 재귀 호출로 만들어 보세요.
- 4-2 문제 2의 숫자  $n$ 개 중에서 최댓값 찾기를 재귀 호출로 만들어 보세요.

## 연습 문제 풀이

부록 A

### 4-1 재귀 호출을 이용해 1부터 n까지의 합 구하기

▼ 예제 소스 e04-1-sum.py

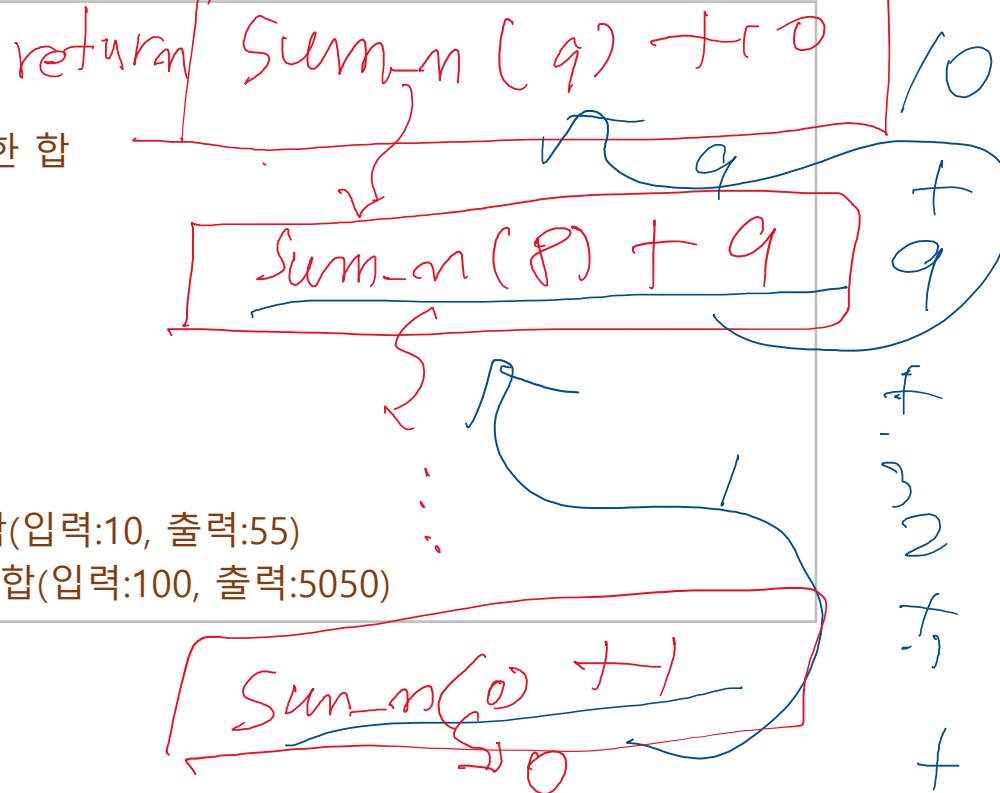
```
# 연속한 숫자의 합을 구하는 알고리즘  
# 입력: n  
# 출력: 1부터 n까지 연속한 숫자를 더한 합
```

```
def sum_n(n):  
    if n == 0:  
        return 0  
    return sum_n(n - 1) + n
```

```
print(sum_n(10)) # 1부터 10까지의 합(입력:10, 출력:55)
```

```
print(sum_n(100)) # 1부터 100까지의 합(입력:100, 출력:5050)
```

call stack



## 연습 문제 풀이

부록 A

- 실행 결과

```
55
5050
```

- 종료 조건:  $n = 0 \rightarrow$  곱값 0
- 재귀 호출 조건:  $n$ 까지의 합 =  $n-1$ 까지의 합 +  $n$



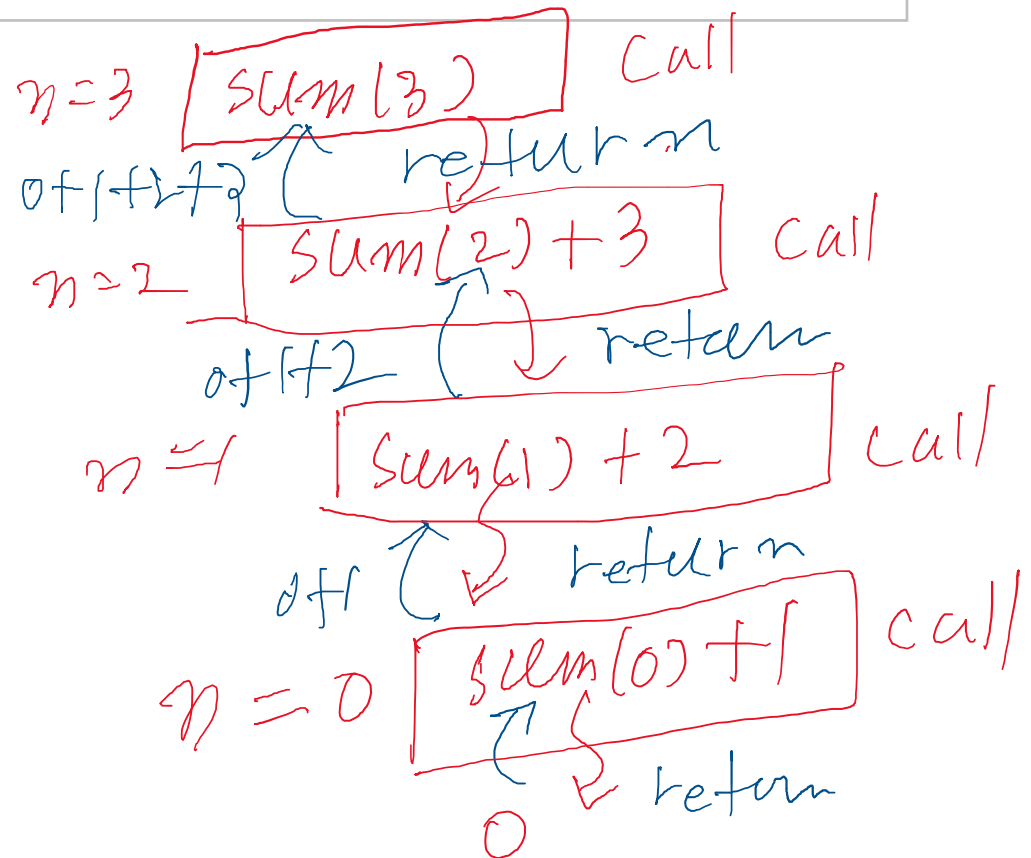
### 연습 문제

- $n = 3$  일 경우 재귀 호출의 call stack 과정을 그림으로 그리시오.

$$1 + 2 + 3 = 6$$

```
def sum(n):  
    if n == 0:  
        return 0  
    return sum(n - 1) + n
```

```
print(sum(3))
```



### 4-2 재귀 호출을 이용한 최댓값 찾기

▼ 예제 소스 e04-2-findmax.py

```
# 최댓값 구하기
# 입력: 숫자가 n개 들어 있는 리스트
# 출력: 숫자 n개 중 최댓값

def find_max(a, n): # 리스트 a의 앞부분 n개 중 최댓값을 구하는 재귀 함수
    if n == 1:
        return a[0]
    max_n_1 = find_max(a, n - 1) # n-1개 중 최댓값을 구함
    if max_n_1 > a[n - 1]:      # n-1개 중 최댓값과 n-1번 위치 값을 비교
        return max_n_1
    else:
        return a[n - 1]
```

```
v = [17, 92, 18, 33, 58, 7, 33, 42]
```

```
print(find_max(v, len(v))) # 함수에 리스트의 자료 갯수를 인자로 추가하여 호출
```

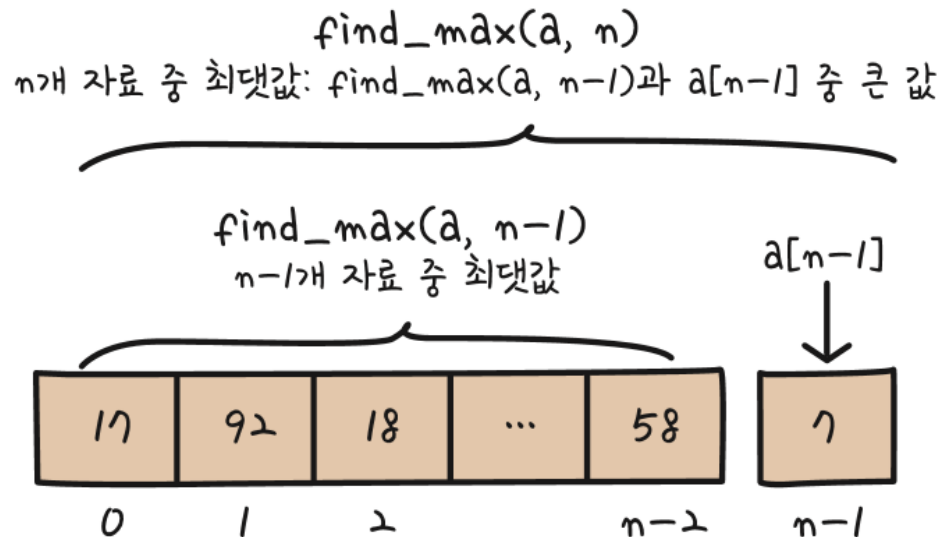
### 실행 결과

92

## 연습 문제 풀이

부록 A

- 종료 조건: 자료 값이 한 개면( $n = 1$ ) 그 값이 최대값
- 재귀 호출 조건:  $n$ 개 자료 중 최대값  $\rightarrow$   $n-1$ 개 자료 중 최대값과  $n-1$ 번 위치 값 중 더 큰 값



## 문제 05 최대공약수 구하기

---

두 자연수  $a$ 와  $b$ 의 최대공약수를 구하는 알고리즘을 만들어 보세요.

- 최대공약수: 두 개 이상의 정수의 공통 약수 중에서 가장 큰 값
- 두 자연수의 최대공약수를 찾으려면 ①두 수의 **약수** 중에서 ②**공통**된 것을 찾아 ③그 값 중 **최댓**값인 것을 찾아야 함
- 3단계 중 밑줄 친 약수, 공, 최대라는 단어를 역순으로 읽으면 '최대공약수'

## 1 최대공약수 알고리즘

- 최대공약수의 성질을 떠올리면서 다음 알고리즘을 생각해 보자
  - 1) 두 수 중 더 작은 값을  $i$ 에 저장
  - 2)  $i$ 가 두 수의 공통된 약수인지 확인
  - 3) 공통된 약수이면 이 값을 곱셈값으로 돌려주고 종료
  - 4) 공통된 약수가 아니면  $i$ 를 1만큼 감소시키고 2번으로 돌아가 반복  
(1은 모든 정수의 약수이므로  $i$ 가 1이 되면 1을 곱셈값으로 돌려주고 종료함)

## 1 최대공약수 알고리즘

■ 4와 6의 최대공약수를 찾는 과정

- 1)  $i$ 에 4를 저장(4와 6 중 작은 값인 4가 최대공약수 후보 중 가장 큰 값)
- 2) 4는  $i$ 로 나누어떨어지지만, 6은 나누어떨어지지 않음
- 3)  $i$ 를 1만큼 감소시켜 3으로 만듦
- 4) 4는  $i$ 로 나누어떨어지지 않음
- 5)  $i$ 를 1만큼 감소시켜 2로 만듦
- 6) 4도  $i$ 로 나누어떨어지고 6도  $i$ 로 나누어떨어지므로  $i$  값 2가 최대공약수

## 1 최대공약수 알고리즘

### ■ 프로그램 5-1 재귀 호출을 이용한 최대값 찾기

▼ 예제 소스 p05-1-gcd.py

```
# 최대공약수 구하기  
# 입력: a, b  
# 출력: a와 b의 최대공약수
```

% 나머지

```
def gcd(a, b):  
    i = min(a, b) # 두 수 중에서 최솟값을 구하는 파이썬 함수  
    while True:  
        if a % i == 0 and b % i == 0:  
            return i  
        i = i - 1 # i를 1만큼 감소시킴
```





## 1 최대공약수 알고리즘



```
print(gcd(1, 5)) # 1  
print(gcd(3, 6)) # 3  
print(gcd(60, 24)) # 12  
print(gcd(81, 27)) # 27
```

### ■ 실행 결과

```
1  
3  
12  
27
```

## 유클리드 알고리즘

### 유클리드가 발견한 최대공약수의 성질

- a와 b의 최대공약수는 'b'와 'a를 b로 나눈 나머지'의 최대공약수와 같음  
즉,  $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$
- 어떤 수와 0의 최대공약수는 자기 자신  
즉,  $\text{gcd}(n, 0) = n$

## 유클리드 알고리즘

- 60과 24의 최대공약수, 81과 27의 최대공약수

→ 유클리드 방식으로 풀어 보기

$$\text{gcd}(60, 24) = \text{gcd}(24, 60 \% 24) = \text{gcd}(24, 12) = \text{gcd}(12, 24 \% 12) = \text{gcd}(12, 0) = 12$$

$$\text{gcd}(81, 27) = \text{gcd}(27, 81 \% 27) = \text{gcd}(27, 0) = 27$$

- 어떤 두 수의 최대공약수를 구하기 위해 다시 다른 두 수의 최대공약수를 구하고 있음 → '재귀 호출'로 이 문제를 풀 수 있다는 힌트

## 유클리드 알고리즘

- $a$ 와  $b$ 의 최대공약수를 구하기 위해서  $(a, b)$ 보다 좀 더 작은 숫자인  $(b, a \% b)$ 의 최대공약수를 구하는 과정을 이용하는 전형적인 재귀 호출 문제 (좀 더 작은 값으로 자기 자신을 호출)
- 재귀 호출이 무한히 반복되지 않도록 하는 데 필요한 종료 조건  
→ '어떤 수와 0의 최대공약수는 자기 자신'이라는 성질
- 즉,  $b$ 가 0이면 재귀 호출을 멈추고 결과를 돌려줌

## 2 유클리드 알고리즘

- 프로그램 5-2 유클리드 방식을 이용해 최대공약수를 구하는 알고리즘

▼ 예제 소스 p05-2-gcd.py

```
# 최대공약수 구하기
# 입력: a, b
# 출력: a와 b의 최대공약수

def gcd(a, b):
    if b == 0: # 종료 조건
        return a
    return gcd(b, a % b) # 좀 더 작은 값으로 자기 자신을 호출

print(gcd(1, 5))    # 1
print(gcd(3, 6))    # 3
print(gcd(60, 24))  # 12
print(gcd(81, 27))  # 27
```

## 유클리드 알고리즘

- 실행 결과

```
1  
3  
12  
27
```

- 세 줄밖에 되지 않는 gcd() 함수로 최대공약수를 구할 수 있음

## 유클리드 알고리즘

- 재귀 호출로 문제를 푸는 과정

$\text{gcd}(60, 24)$

→  $\text{gcd}(24, 12)$

→  $\text{gcd}(12, 0)$

→ 12 (b가 0이므로 종료 조건)

→ 12

→ 12 (최종 결과)

## 2 유클리드 알고리즘

### 재귀 호출의 이해를 돕는 방법

- ① 종이에 직접 함수 호출 과정을 적어 보자 함수가 호출될 때마다 안쪽으로 들여 쓰고, 값이 반환되면 다시 바깥쪽으로 돌아가는 식으로 과정을 적다 보면 중첩된 함수 호출을 이해하는 데 도움이 됨  
(64쪽에서 본 60과 24의 최대공약수를 구하는 과정을 참조)
- ② 예제로 사용할 입력 값은 작은 것이 좋음. 일단 종료 조건에 해당하는 값을 입력으로 넣은 다음 입력 값을 높이면서 재귀 호출 과정을 따라가 보자
- ③ 함수의 입력 값을 화면에 출력하는 것도 도움이 됨. 예를 들어  $\text{gcd}(a, b)$  함수에 `print()` 함수를 추가한 다음  $\text{gcd}(60, 24)$ 를 실행하면  $\text{gcd}(a, b)$  함수가 연속해서 실행되는 과정을 직접 확인할 수 있음



## 유클리드 알고리즘

```
def gcd(a, b):  
    print("gcd: ", a, b) # 함수 호출을 입력(인자) 값과 함께 화면에 표시  
    if b == 0:           # 종료 조건  
        return a  
    return gcd(b, a % b)
```

```
gcd: 60 24  
gcd: 24 12  
gcd: 12 0  
12
```

## 연습 문제

- 5-1 0과 1부터 시작해서 바로 앞의 두 수를 더한 값을 다음 값으로 추가하는 방식으로 만든 수열을 피보나치 수열이라고 합니다. 즉,  $0+1=1$ ,  $1+1=2$ ,  $1+2=3$ 이므로 피보나치 수열은 다음과 같습니다.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

시작점

- 피보나치 수열이 파이썬의 리스트처럼 0번부터 시작한다고 가정할 때 n번째 피보나치 수를 구하는 알고리즘을 재귀 호출을 이용해서 구현해보세요(힌트: 7번 값 = 5번 값 + 6번 값).

# 연습 문제 풀이

부록 A

### 5-1 재귀 호출을 이용한 피보나치 수열 구하기

▼ 예제 소스 e05-1-fibonacci.py

```
# n번째 피보나치 수열 찾기  
# 입력: n값(0부터 시작)  
# 출력: n번째 피보나치 수열 값
```

```
def fibo(n):  
    if n <= 1:  
        return n # n = 0 -> 0 | n = 1 -> 1  
    return fibo(n - 2) + fibo(n - 1)
```

```
print(fibo(7))  
print(fibo(10))
```

index 0 1 2 3 ...  
0, 1, 1, 2, ...

## 연습 문제 풀이

부록 A

### ■ 실행 결과

13

55

- 종료 조건:  $n = 0 \rightarrow$  곱값 0,  $n = 1 \rightarrow$  곱값 1
- 재귀 호출 조건:  $n$ 번 피보나치 수 =  $n-2$ 번 피보나치 수 +  $n-1$ 번 피보나치 수

## 문제 06 하노이의 탑 옮기기

---

원반이  $n$ 개인 하노이의 탑을 옮기기 위한 원반 이동 순서를 출력하는 알고리즘을 만들어 보세요.

- '하노이의 탑'을 알아보고 재귀 호출을 이용해 이 문제를 풀어 보자

## 1 하노이의 탑

- 하노이의 탑(Tower of Hanoi): 간단한 원반(disk) 옮기기 퍼즐

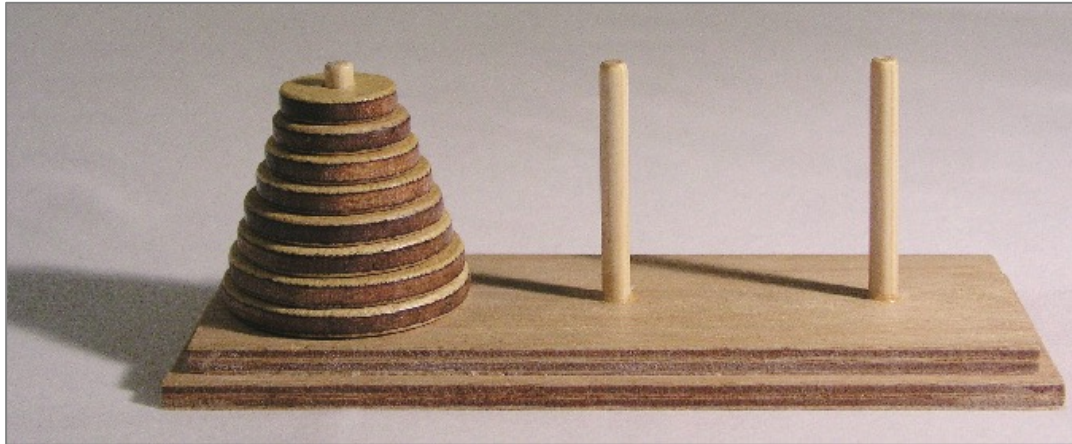


그림 6-1 하노이의 탑(출처: [https://en.wikipedia.org/wiki/Tower\\_of\\_Hanoi](https://en.wikipedia.org/wiki/Tower_of_Hanoi))

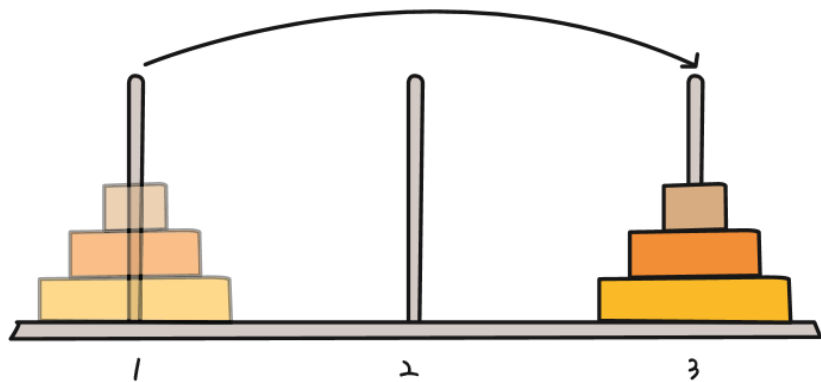
## 하노이의 탑

### 하노이의 탑 규칙

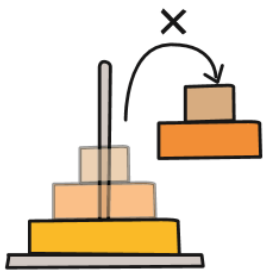
- 크기가 다른 원반  $n$ 개를 출발점 기둥에서 도착점 기둥으로 전부 옮겨야 함
- 원반은 한 번에 한 개씩만 옮길 수 있음
- 원반을 옮길 때는 한 기둥의 맨 위 원반을 뽑아, 다른 기둥의 맨 위로만 옮길 수 있음(기둥의 중간에서 원반을 빼내거나 빼낸 원반을 다른 기둥의 중간으로 끼워 넣을 수 없음).
- 원반을 옮기는 과정에서 큰 원반을 작은 원반 위로 올릴 수 없음



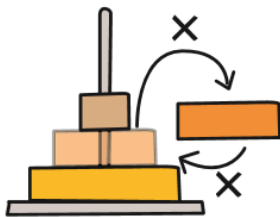
1 하노이의 탑



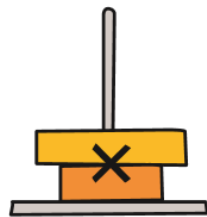
하노이의 탑에 있는 모든 원반을 출발점에서 도착점으로 옮겨야 함



한 번에  
여러 개를  
옮길 수 없음



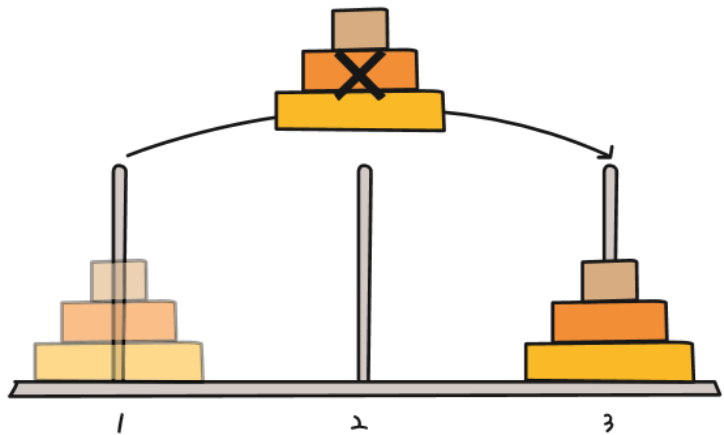
중간에서  
빼거나 중간으로  
넣을 수 없음



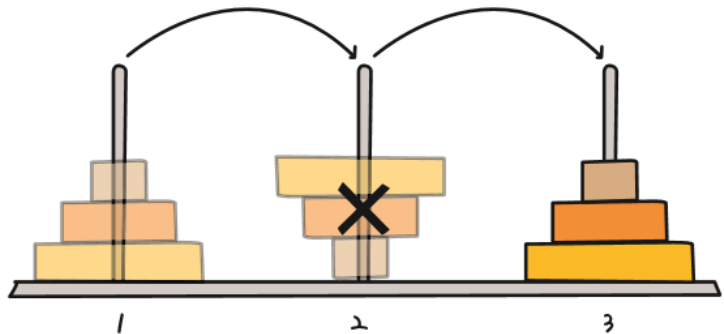
원반의 크기가  
거꾸로 놓일 수  
없음

그림 6-2 하노이의 탑 규칙

1 하노이의 탑



한 번에 여러 개를 옮길 수 없음



원반의 크기가 거꾸로 놓일 수 없음

그림 6-2 하노이의 탑 규칙 제대로 이해하기

## 하노이의 탑

- 661원으로 하노이의 탑 교재를 간단히 만들어 실험할 수 있음

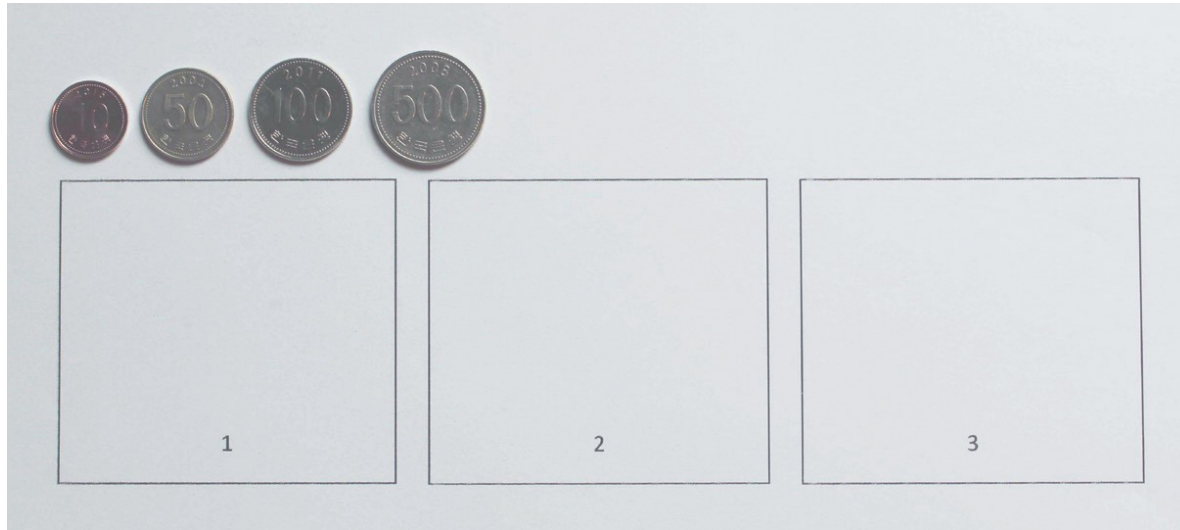


그림 6-4 661원 하노이의 탑 - 동전 네 개와 종이 한 장(1원?)으로 만든 하노이의 탑 교재

## 2 하노이의 탑 풀이

원반이 한 개일 때

- 1번 기둥에 있는 원반을 3번 기둥으로 옮기면 끝(1 → 3)

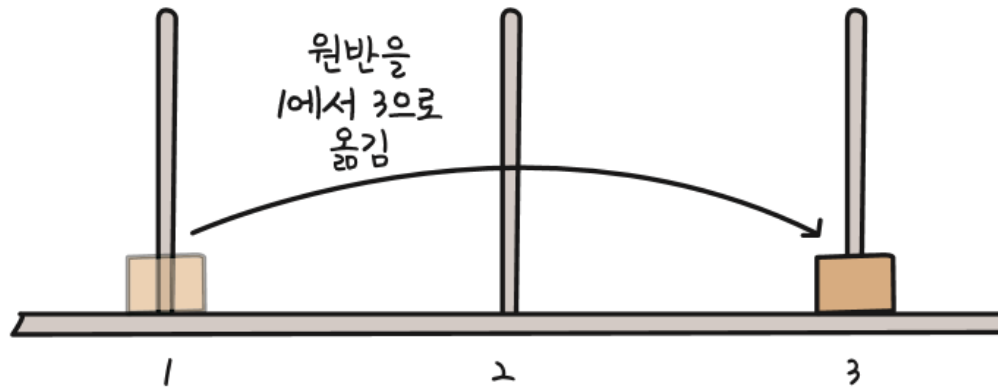


그림 6-5 원반을 1에서 3으로 옮김

## 2 하노이의 탑 풀이

원반이 두 개일 때

- ① 1번 기둥의 맨 위에 있는 원반을 2번 기둥으로 옮김( $1 \rightarrow 2$ )

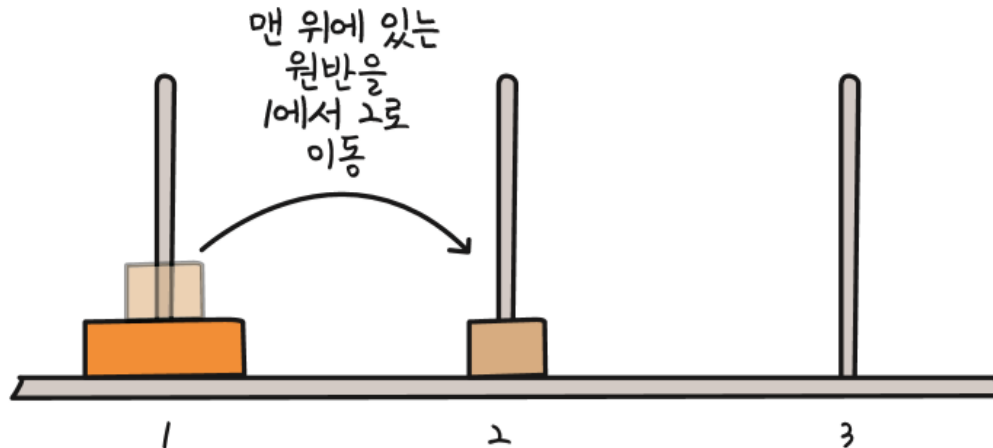


그림 6-6 맨 위의 원반을 1에서 2로 이동

## 2 하노이의 탑 풀이

② 1번 기둥에 남아 있는 큰 원반을 3번 기둥으로 옮김( $1 \rightarrow 3$ )

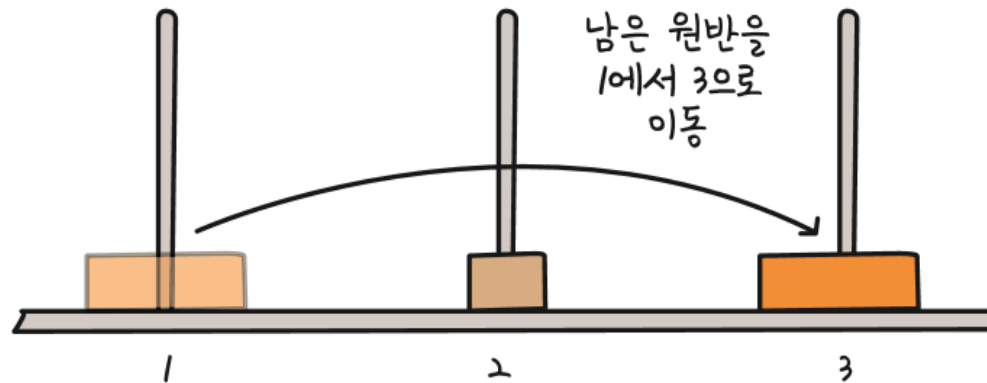


그림 6-7 남은 원반을 1에서 3으로 이동

## 2 하노이의 탑 풀이

③ 2번 기둥에 남아 있는 원반을 3번 기둥으로 옮김(2 → 3)

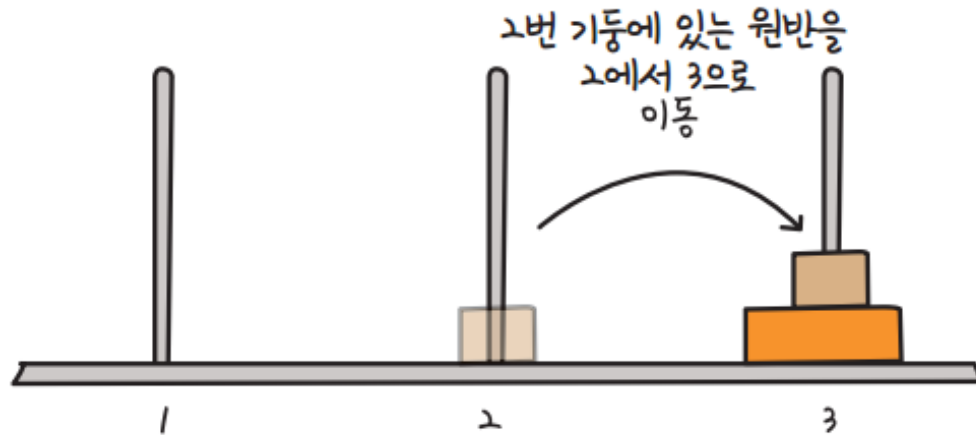


그림 6-8 2번 기둥의 원반을 2에서 3으로 이동

## 2 하노이의 탑 풀이

원반이 세 개일 때

- ① 1번 기둥의 원반 중 위에 있는 원반 두 개를 2번 기둥(보조 기둥)으로 옮김  
(한 번에 원반을 두 개씩 옮길 수 없지만, 원반 두 개일 때 방법 그대로 적용)

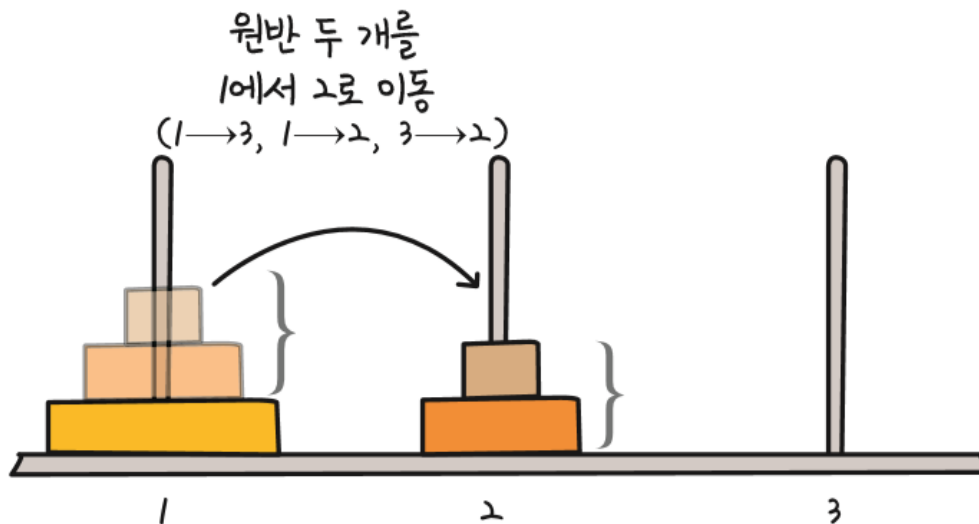


그림 6-9 원반 두 개를 1에서 2로 이동(1 → 3, 1 → 2, 3 → 2)



## 2 하노이의 탑 풀이

원반이 세 개일 때

- ① 1번 기둥의 원반 중 위에 있는 원반 두 개를 2번 기둥(보조 기둥)으로 옮김  
(한 번에 원반을 두 개씩 옮길 수 없지만, 원반 두 개일 때 방법 그대로 적용)

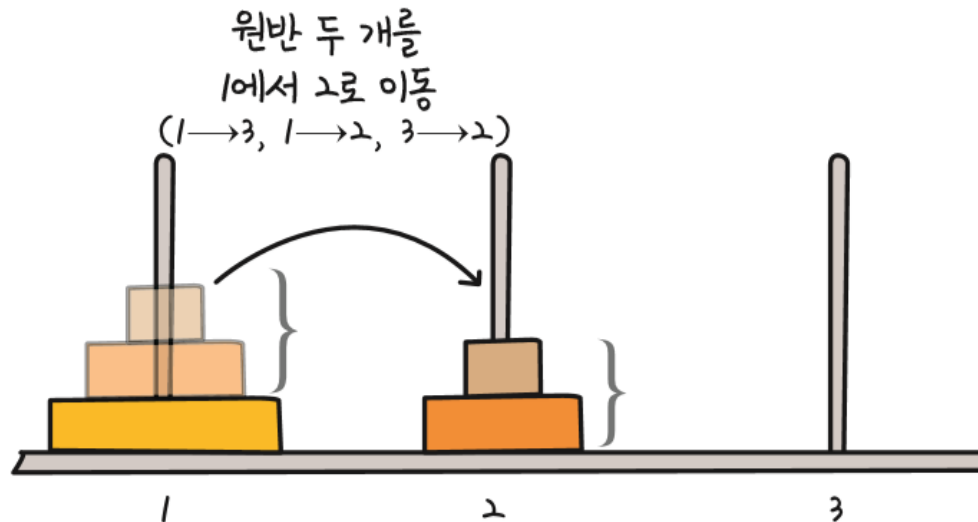


그림 6-9 원반 두 개를 1에서 2로 이동(1 → 3, 1 → 2, 3 → 2)

## 2 하노이의 탑 풀이

② 1번 기둥에 남아 있는 큰 원반을 3번 기둥으로 옮김( $1 \rightarrow 3$ )

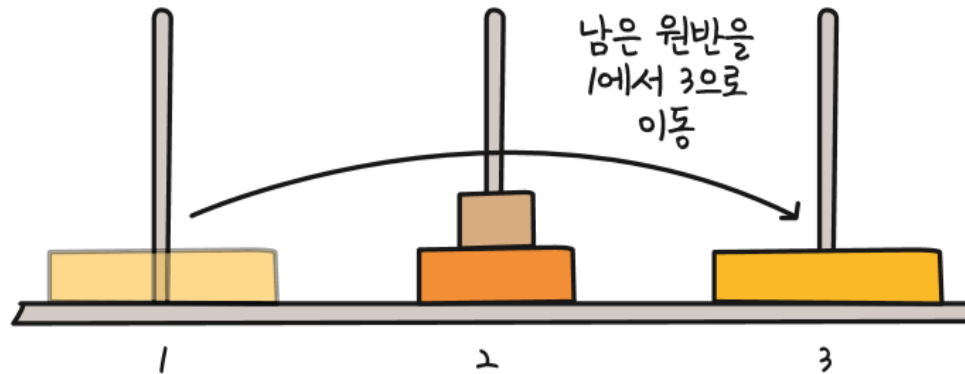


그림 6-10 남은 원반을 1에서 3으로 이동

## 하노이의 탑 풀이

③ 2번 기둥에 있는 원반 두 개를 3번 기둥으로 옮김

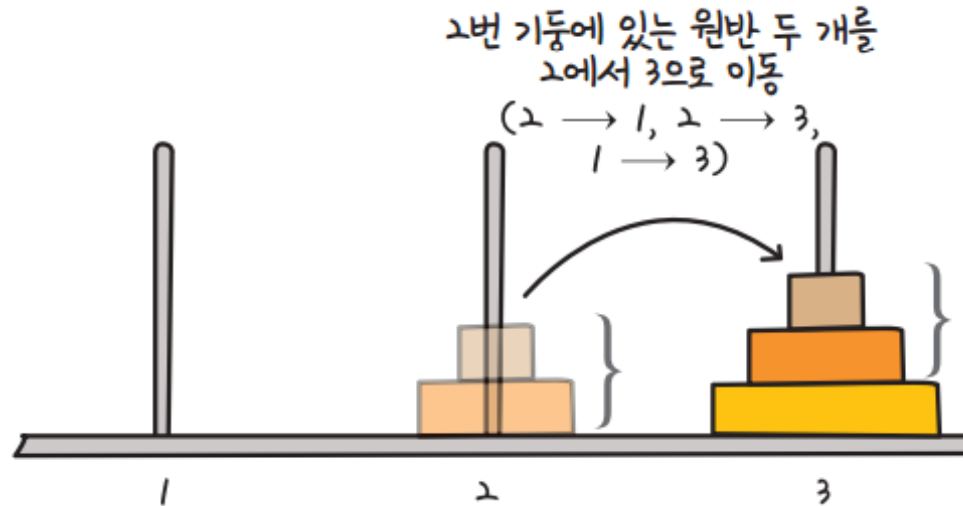


그림 6-11 2번 기둥의 원반 두 개를 3번으로 이동(2 → 1, 2 → 3, 1 → 3)

- 정리하면 원반을 한 개씩 전부 일곱 번 옮기면 문제가 해결( $3 + 1 + 3 = 7$ )

## 2 하노이의 탑 풀이

원반이  $n$ 개일 때

- ① 1번 기둥에 있는  $n-1$ 개 원반을 2번 기둥으로 옮김  
( $n-1$ 개짜리 하노이의 탑 문제 풀기)

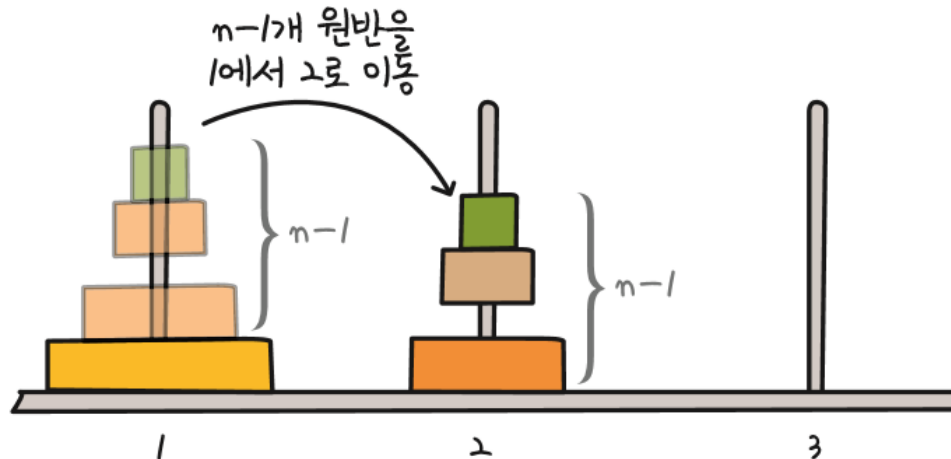


그림 6-12  $n-1$ 개 원반을 1에서 2로 이동

## 2 하노이의 탑 풀이

② 1번 기둥에 남아 있는 가장 큰 원반을 3번 기둥으로 옮김(1 → 3)

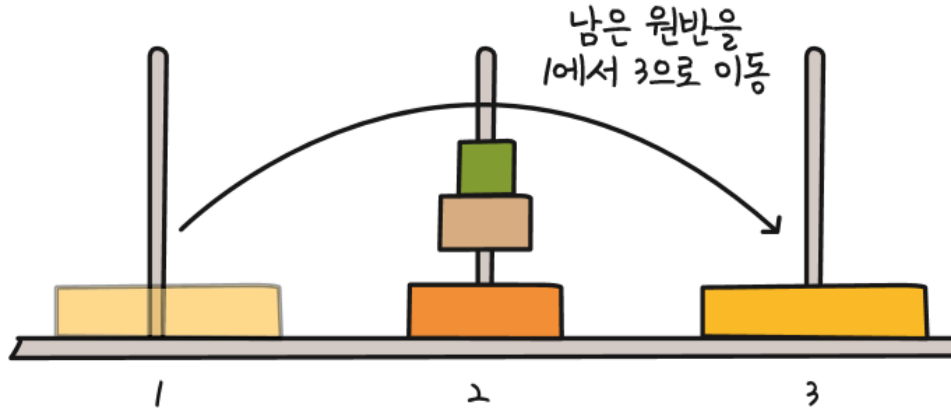


그림 6-13 남은 원반을 1에서 3으로 이동

## 2 하노이의 탑 풀이

- ③ 2번 기둥에 있는  $n-1$ 개 원반을 3번 기둥으로 옮김  
( $n-1$ 개짜리 하노이의 탑 문제 풀기)

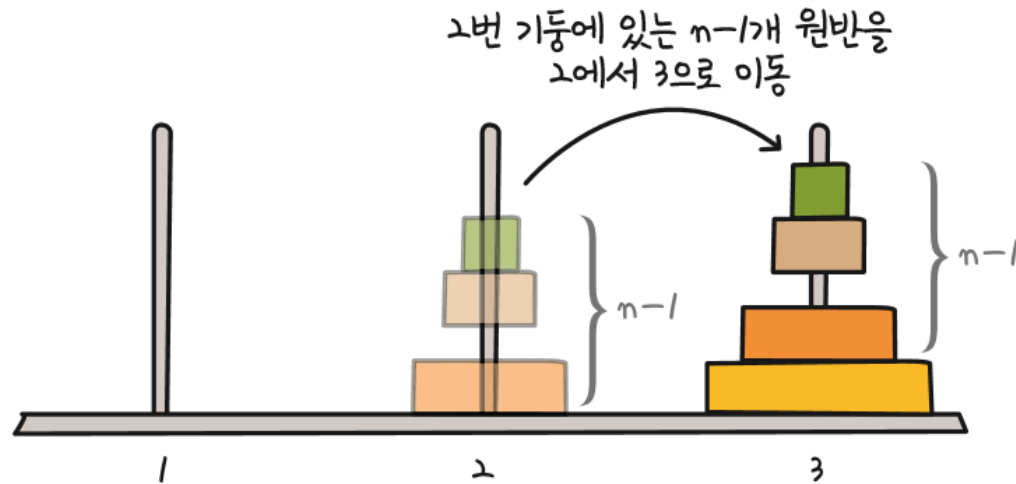


그림 6-14 2번 기둥의  $n-1$ 개 원반을 2에서 3으로 이동

### 3 하노이의 탑 알고리즘

- 하노이의 탑 옮기기 알고리즘

1-A 원반이 한 개면 그냥 옮기면 끝(종료 조건)

1-B 원반이  $n$ 개일 때

- ① 1번 기둥에 있는  $n$ 개 원반 중  $n-1$ 개를 2번 기둥으로 옮김  
(3번 기둥을 보조 기둥으로 사용)
- ② 1번 기둥에 남아 있는 가장 큰 원반을 3번 기둥으로 옮김
- ③ 2번 기둥에 있는  $n-1$ 개 원반을 다시 3번 기둥으로 옮김  
(1번 기둥을 보조 기둥으로 사용)

## 3 하노이의 탑 알고리즘

### ■ 프로그램 6-1 하노이의 탑 알고리즘

▼ 예제 소스 p06-1-hanoi.py

```
# 하노이의 탑
# 입력: 옮기려는 원반의 갯수 n
#      옮길 원반이 현재 있는 출발점 기둥 from_pos
#      원반을 옮길 도착점 기둥 to_pos
#      옮기는 과정에서 사용할 보조 기둥 aux_pos
# 출력: 원반을 옮기는 순서

def hanoi(n, from_pos, to_pos, aux_pos):
    if n == 1: # 원반 한 개를 옮기는 문제면 그냥 옮기면 됨
        print(from_pos, "->", to_pos)
    return
```





### 3 하노이의 탑 알고리즘



```
# 원반 n - 1개를 aux_pos로 이동(to_pos를 보조 기둥으로)
hanoi(n - 1, from_pos, aux_pos, to_pos)
# 가장 큰 원반을 목적지로 이동
print(from_pos, "->", to_pos)
# aux_pos에 있는 원반 n-1개를 목적지로 이동(from_pos를 보조 기둥으로)
hanoi(n - 1, aux_pos, to_pos, from_pos)

print("n = 1")
hanoi(1, 1, 3, 2) # 원반 한 개를 1번 기둥에서 3번 기둥으로 이동(2번을 보조 기둥으로)
print("n = 2")
hanoi(2, 1, 3, 2) # 원반 두 개를 1번 기둥에서 3번 기둥으로 이동(2번을 보조 기둥으로)
print("n = 3")
hanoi(3, 1, 3, 2) # 원반 세 개를 1번 기둥에서 3번 기둥으로 이동(2번을 보조 기둥으로)
```

## 하노이의 탑 알고리즘

### ■ 실행 결과

```
n = 1  
1 -> 3  
n = 2  
1 -> 2  
1 -> 3  
2 -> 3  
n = 3  
1 -> 3  
1 -> 2  
3 -> 2  
1 -> 3  
2 -> 1  
2 -> 3  
1 -> 3
```

## 알고리즘 분석

- 1층짜리 하노이의 탑: 원반을 한 번 이동
- 2층짜리 하노이의 탑: 원반을 세 번 이동
- 3층짜리 하노이의 탑: 원반을 일곱 번 이동
- 입력 크기, 즉 탑의 층수가 높을수록 원반을 더 많이 움직여야 함

## 알고리즘 분석

$n$ 층짜리 하노이의 탑을 옮기려면 원반을 몇 번 움직여야 할까?

- $n$ 층짜리 하노이의 탑을 옮기려면 원반을 모두  $2^n - 1$ 번 옮겨야 함
- 마찬가지로  $n$ 이 커지면  $-1$ 은 큰 의미가 없으므로  $O(2^n)$ 으로 표현할 수 있음
- 이는 2를  $n$ 번 제곱한 값이므로  $n$ 이 커짐에 따라 값이 기하급수적으로 증가

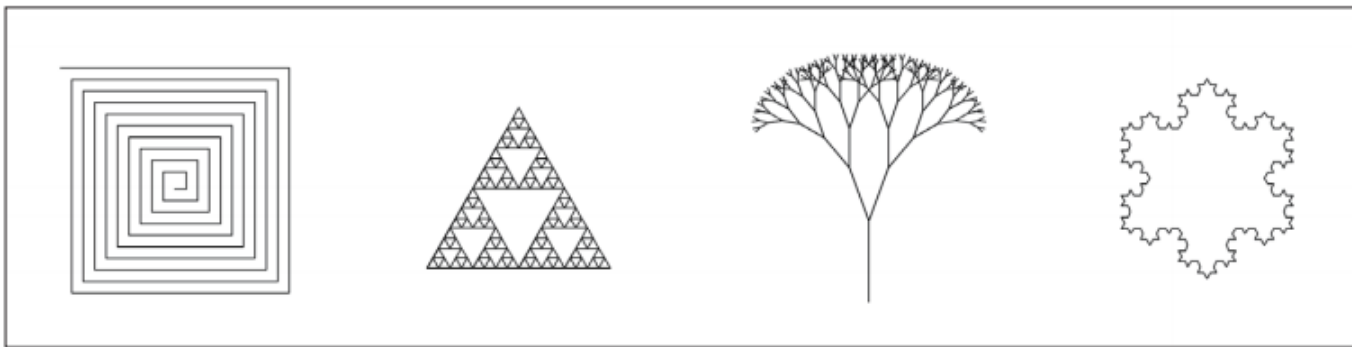
## 알고리즘 분석

### 지금까지 살펴본 계산 복잡도

- $O(1)$ :  $n$ 과 무관하게 일정한 시간이 걸림
- $O(n)$ :  $n$ 과 비례하여 계산 시간이 증가함
- $O(n^2)$ :  $n$ 의 제곱에 비례하여 계산 시간이 증가함
- $O(2^n)$ : 2의  $n$  제곱에 비례하여 계산 시간이 증가함
- 위에서 아래로 갈수록 계산 복잡도가 증가

## 연습 문제

- 6-1 재귀 호출의 원리는 컴퓨터 그래픽에서도 많이 사용됩니다. 다음 그림은 모두 재귀 호출을 이용해서 만든 컴퓨터 그래픽입니다.



재귀 호출로 어떻게 이런 그림을 그릴 수 있을지 상상해 보고 부록 D에 있는 '재귀 호출을 이용한 그림 그리기' 부분을 읽어 보세요.

## 연습 문제 풀이

부록 D

- 6-1 재귀 호출을 이용한 그림 그리기
- 거북이 그래픽: 화면 위에 펜 역할을 하는 거북이를 올린 후, 그 거북이가 움직이도록 명령을 내려 그림을 그리도록 하는 그래픽 시스템
- 어떤 그림 안에 자기 자신과 똑같이 닮았지만, 크기가 조금 더 작은 그림을 반복하여 그림(자기 복제 과정)
- 이 과정을 반복하다가 그려야 할 그림의 크기가 어느 정도로 작아지면 재귀 호출을 멈춤(종료 조건)

# 연습 문제 풀이

부록 D

### ■ 프로그램 D-1 사각 나선을 그리는 프로그램

#### ▼ 예제 소스 e06-1-spiral.py

```
# 재귀 호출을 이용한 사각 나선 그리기
import turtle as t

def spiral(sp_len):
    if sp_len <= 5:
        return
    t.forward(sp_len)
    t.right(90)
    spiral(sp_len - 5)
```







```
t.speed(0)
spiral(200)
t.hideturtle()
t.done()
```

**연습 문제 풀이** 부록 D

■ 실행 결과

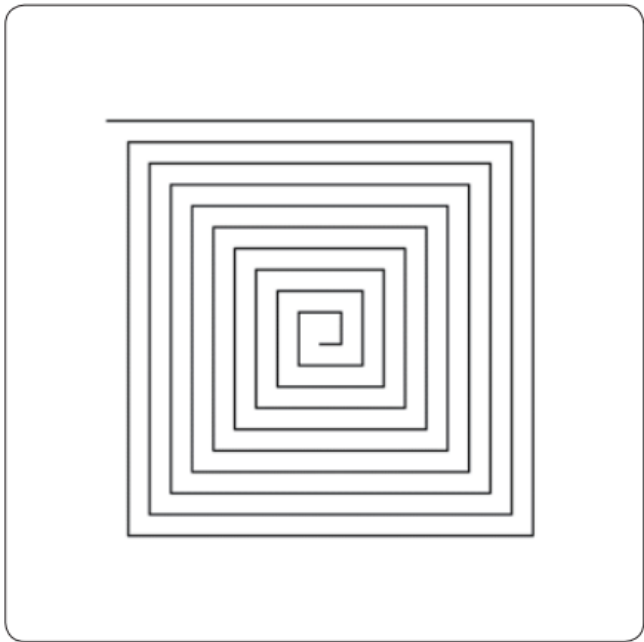


그림 D-1 사각 나선

### ■ 프로그램 D-2 시에르핀스키의 삼각형을 그리는 프로그램

#### ▼ 예제 소스 e06-2-triangle.py

```
# 재귀 호출을 이용한 시에르핀스키(Sierpinski)의 삼각형 그리기
import turtle as t

def tri(tri_len):
    if tri_len <= 10:
        for i in range(0, 3):
            t.forward(tri_len)
            t.left(120)
        return
    new_len = tri_len / 2
    tri(new_len)
```





```
t.forward(new_len)
tri(new_len)
t.backward(new_len)
t.left(60)
t.forward(new_len)
t.right(60)
tri(new_len)
t.left(60)
t.backward(new_len)
t.right(60)

t.speed(0)
tri(160)
t.hideturtle()
t.done()
```

## 연습 문제 풀이

부록 D

### ■ 실행 결과

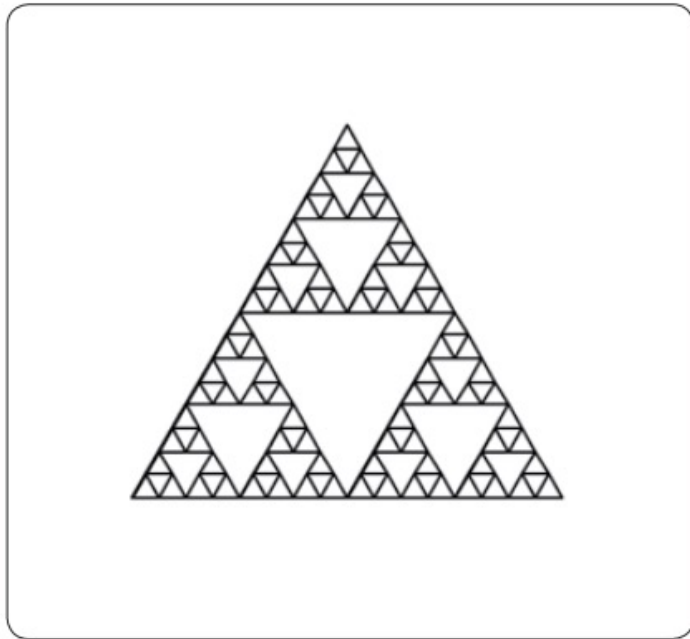


그림 D-2 시에르핀스키의 삼각형

### ■ 프로그램 D-3 나무를 그리는 프로그램

#### ▼ 예제 소스 e06-3-tree.py

```
# 재귀 호출을 이용한 나무 모형 그리기
import turtle as t

def tree(br_len):
    if br_len <= 5:
        return
    new_len = br_len * 0.7
    t.forward(br_len)
    t.right(20)
    tree(new_len)
    t.left(40)
    tree(new_len)
```





```
t.left(40)
tree(new_len)
t.right(20)
t.backward(br_len)

t.speed(0)
t.left(90)
tree(70)
t.hideturtle()
t.done()
```

## 연습 문제 풀이

부록 D

### ■ 실행 결과

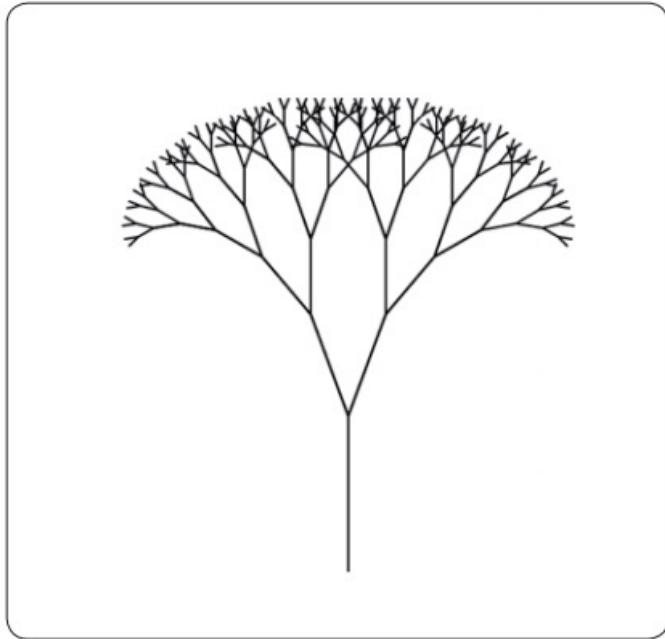


그림 D-3 나무



### ■ 프로그램 D-4 눈꽃을 그리는 프로그램

#### ▼ 예제 소스 e06-4-snow.py

```
# 재귀 호출을 이용한 눈꽃 그리기
import turtle as t

def snow_line(snow_len):
    if snow_len <= 10:
        t.forward(snow_len)
        return
    new_len = snow_len / 3
    snow_line(new_len)
    t.left(60)
    snow_line(new_len)
```





```
t.right(120)
snow_line(new_len)
t.left(60)
snow_line(new_len)

t.speed(0)
snow_line(150)
t.right(120)
snow_line(150)
t.right(120)
```



■ 실행 결과

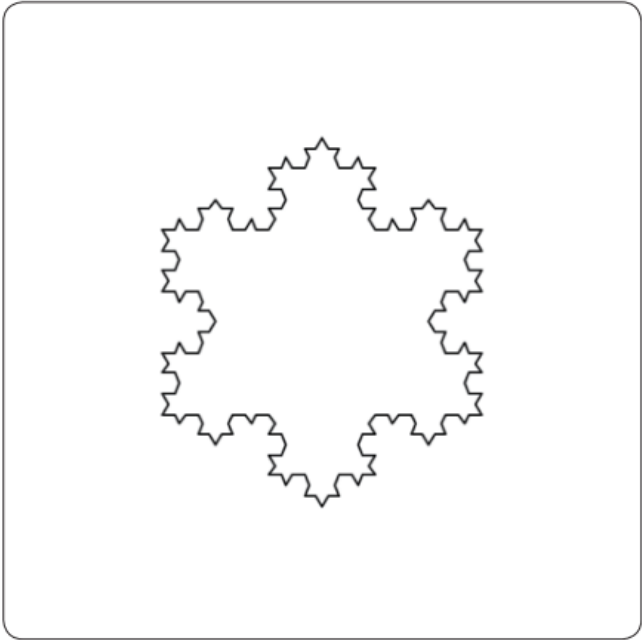


그림 D-4 눈꽃