가천대학교 Gachon University

03

Algorithm

점화식과 재귀 알고리즘

2025-03-21

조윤실

목 차

※ 가천대학교 컴퓨터공학과 ‘알고리즘’ 과정

알고리즘 분석을 위한 기초

알고리즘 효율성 분석

알고리즘의 필요성

- 알고리즘의 필요성
  - 같은 문제에서도 효율 면에서 아주 큰 차이가 나는 다양한 알고리즘이 존재
  - 알고리즘을 학습하는 과정에서 얻을 수 있는 여러 가지 기법과 생각하는 방법 훈련 가능

좋은 알고리즘이란?

- 알고리즘이 얼마나 효율적인가?

  같은 일을 하면서도 컴퓨터의 자원(resource)을 최소한으로 사용

  - 시간 효율성(Time efficiency)

    $$T : N \to C \; (N = \{n \mid n \geqslant 0,\ n \in \mathbb{N}\},\ C = \{c \mid c \geqslant 0,\ c \in \mathbb{N}\})$$

  - 공간 효율성(Space efficiency)

    $$S : N \to V \; (N = \{n \mid n \geqslant 0,\ n \in \mathbb{N}\},\ V = \{v \mid v \geqslant 0,\ v \in \mathbb{N}\})$$

알고리즘 효율성 분석

▪ 시간 효율성 분석 방법

- 실행 시간 측정 방법(Empirical Analysis)
  - ➢ 실제 실행 시간을 측정하여 성능 비교
  - ➢ 하드웨어 및 환경의 영향을 받을 수 있음

```python
import time
start = time.time()
test(1000)
end = time.time()
print(f"실행시간 = {end-start}")
```

- 복잡도 분석(Asymptotic Analysis, 점근적 분석) 이론적 복잡도
  - ➢ **입력 크기가 매우 클 때의 성능 평가**
  - ➢ 데이터의 크기에 따른 예측을 통해 객관적으로 비교할 수 있는 기준을 제시
  - ➢ 빅오(Big-O) 표기법 사용

[참고] 지난 시간에…

- (Python) 성능 분석을 위한 주요 지표
  - 실행 시간 (time 모듈) → 더 빠른 알고리즘 선택
  - CPU 사용량 (psutil.cpu_percent) → CPU 효율 확인
  - 메모리 사용량 (psutil.memory_info, memory_profiler) → RAM 소모 최적화
  - 라인별 메모리 사용량 (memory_profiler) → 메모리 많이 사용하는 부분 분석

알고리즘 효율성 분석

- 입력 크기에 따른 분석
  - 크기가 작은 문제
    - 알고리즘의 효율성이 중요하지 않음
    - 비효율적인 알고리즘도 무방
  - 크기가 충분히 큰 문제
    - 알고리즘의 효율성이 중요함
    - 비효율적인 알고리즘은 치명적임
    - 복잡도 분석(Asymptotic Analysis, 점근적 분석)

알고리즘 효율성 분석

- 알고리즘의 수행 시간에 영향을 주는 요소
  - 입력의 크기(n)는 대부분 자명함
    - ➢ **정렬** : 정렬할 원소의 수
    - ➢ **색인** : 색인에 포함된 원소의 수
  - 연산의 종류
    - ➢ **일반 연산** : (덧셈, 곱셈, 나눗셈 등)
    - ➢ **반복문** : loop
    - ➢ **분할 정복** : 분할(Divide), 정복(Conquer), 조합(Combine)
    - ➢ **재귀 호출** : recursive
  - 기본 연산 : 알고리즘에서 가장 많이 실행되는 연산 → 이 연산의 횟수를 센다.

실습 : 성능 비교하기

- 데이터를 리스트에 추가하는 다양한 방법의 성능을 측정하고 그래프로 비교해 보세요.

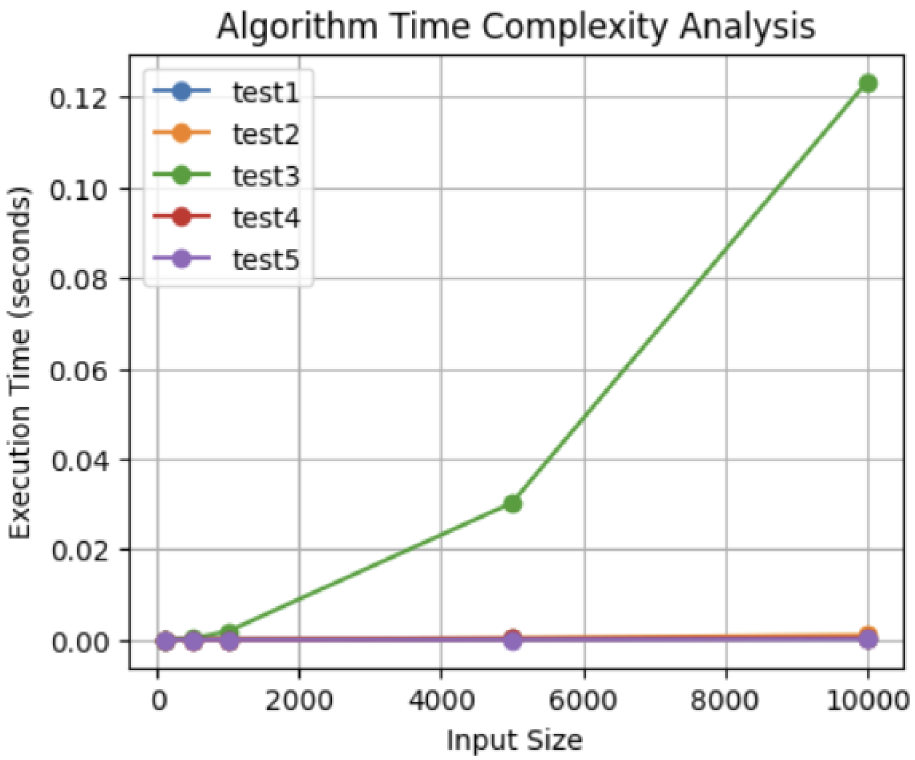
```python
# 1.append() 메서드
def test1(n):
    l = []
    for i in range(n):
        l.append(i)

# 2.extend() 메서드
def test2(n):
    l = []
    for i in range(n):
        l.extend([i])

# 3.리스트 연결 연산자
def test3(n):
    l = []
    for i in range(n):
        l = l + [i]
```

```python
# 4.리스트 조건제시법
def test4(n):
    l = [i for i in range(n)]

# 5.range 객체 활용
def test5(n):
    l = list(range(n))
```



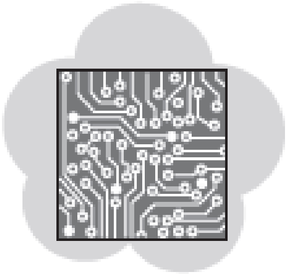
복잡도 분석
(Asymptotic Analysis)

복잡도 분석

- 어느 알고리즘이 효율적일까?

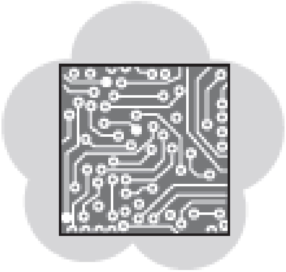
문제: n개의 숫자를
오름차순으로 정렬하라.

알고리즘 A



65536n + 2000000

알고리즘 B



$n^2 + 2n$

[두 함수의 증가율 비교]

n(입력의 크기)	알고리즘 A $65536n + 2000000$	비교	알고리즘 B $n^2 + 2n$
1	2,065,536	>	3
10	2,655,360	>	120
100	8,553,600	>	10,200
1,000	67,536,000	>	1,002,000
10,000	657,360,000	>	100,020,000
100,000	6,555,600,000	<	10,000,200,000
1,000,000	65,538,000,000	<	1,000,002,000,000
10,000,000	655,362,000,000	<	100,000,020,000,000
100,000,000	6,553,602,000,000	<	10,000,000,200,000,000
1,000,000,000	65,536,002,000,000	<	1,000,000,002,000,000,000

n이 커질수록 엄청난
연산이 필요함

[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

알고리즘 복잡도 분석에서 중요한 점

- 알고리즘에서 **입력의 크기**는 무엇인가?
- 복잡도에 영향을 미치는 가장 핵심적인 **기본 연산**은 무엇인가?
- 입력의 크기가 증가함에 따라 처리 시간은 **어떤 형태로 증가**하는가?
- **입력의 특성에 따라 알고리즘 효율성**에는 어떤 차이가 있는가?

입력의 크기

- 알고리즘의 효율성은 입력 크기의 함수 형태로 표현
  - 복잡도 함수

- 무엇이 입력의 크기를 나타내는지를 먼저 명확히 결정
  - 리스트에서 어떤 값을 찾는 문제?
  - x의 n 거듭제곱(x^n)?
  - 다항식의 연산($a_n x^n + a_{n-1} x^{n-1} \cdots$)?
  - Row x Col 의 행렬 연산?
  - 그래프 연산 ?

기본 연산(Basic operation)

- 알고리즘에서 가장 중요한(많이 수행되는) 연산
  - e.g.: 다중 루프의 경우 가장 안쪽 루프에 있는 연산 → 이 연산이 실행되는 횟수 만을 계산

- n의 거듭제곱

알고리즘 A	알고리즘 B	알고리즘 C
n * n	n + n + ⋯ + n	1 + 1 + ⋯ + 1 + 1 + 1 + ⋯ + 1 + ⋯ + 1 + 1 + ⋯ + 1

	알고리즘 A	알고리즘 B	알고리즘 C
유사코드	sum ← n * n	1. sum ← 0 2. for i←1 to n do 3. sum ← sum + n	1. sum ← 0 2. for i←1 to n do 3. for j←1 to n do 4. sum ← sum + 1
전체 연산 횟수	대입 연산: 1 곱셈 연산: 1 전체 횟수: 2	대입 연산: $n+2$ 덧셈 연산: n 전체 횟수: $2n+2$	대입 연산: n^2+n+2 덧셈 연산: n^2 전체 횟수: $2n^2+n+2$

[그림 출처] : 파이썬 알고리즘 by 생능북스

실습 : 얼마나 많은 연산이 실행되는가?

- 1부터 n까지 합을 구하는 두 가지 알고리즘이다 . 각 알고리즘의 연산횟수는?

```
calc_sum1( n )
    sum ← 0                     # 1회 수행
    for i ← 1 to n then# n회 수행(반복 제어부)
        sum ← sum + i       # n회 수행(반복문 내부)
    return sum                  # 1회 수행
```

```
calc_sum2( n )
    sum ← n * (n+1) / 2      # 1회 수행
    return sum                   # 1회 수행
```

복잡도 함수

- 어떤 형태로 증가하는가를 나타내는 함수
- 입력의 크기에 대한 기본 연산의 수행 횟수를 나타냄

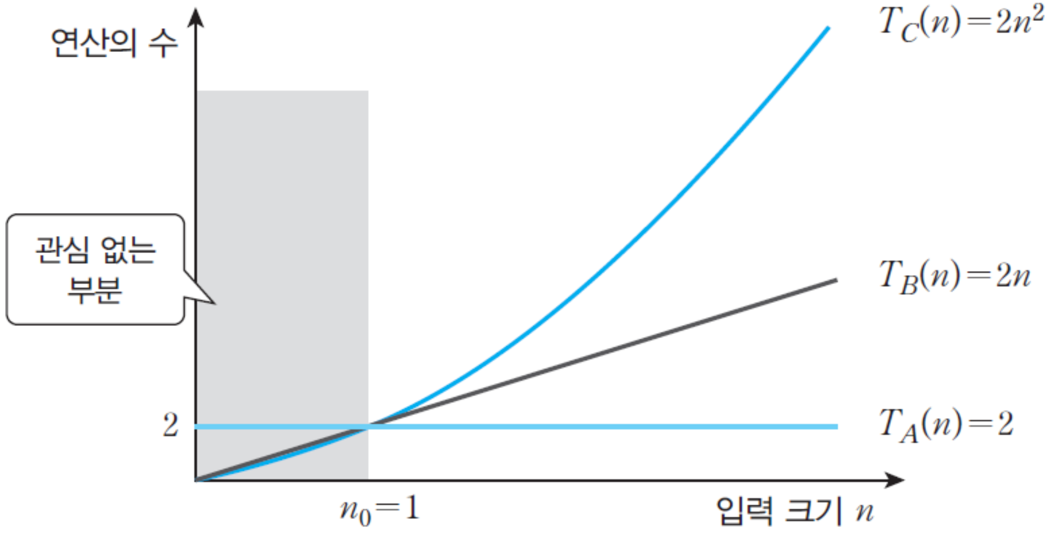
$$T_A(n) = 2,\ T_B(n) = 2n,\ T_C(n) = 2n^2$$

근사적으로 계산된 것

- n이 작은 경우: 예 n=1

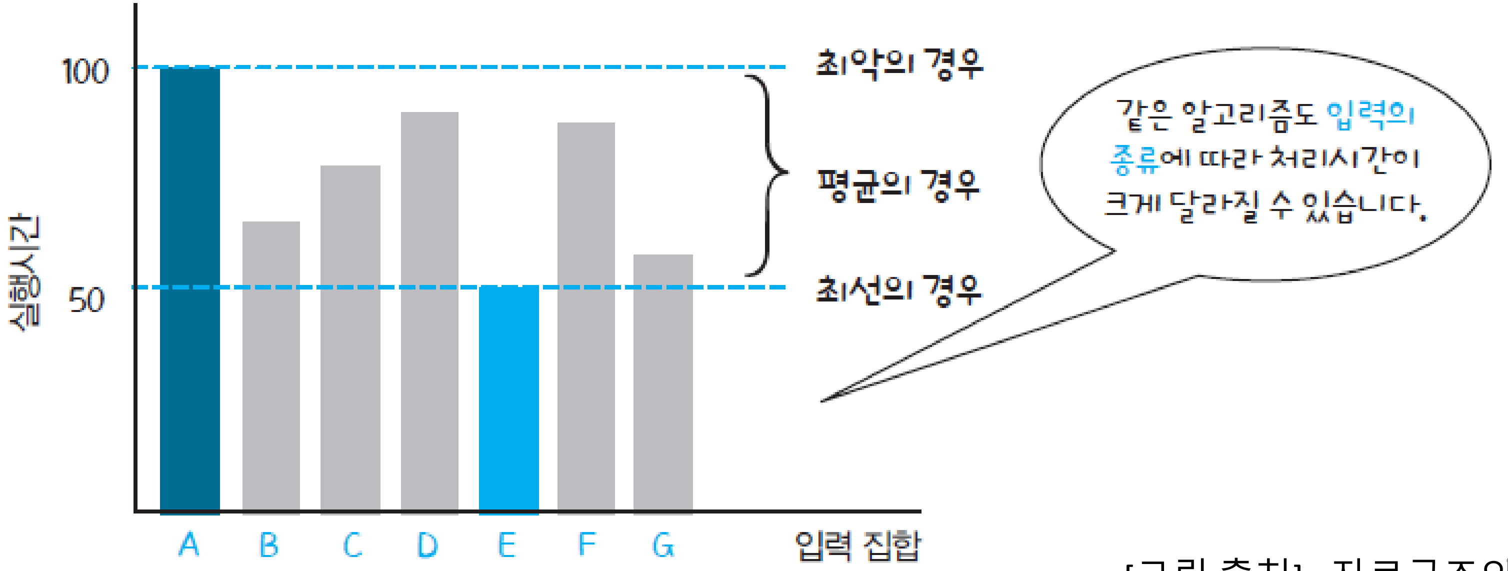
$$T_A(1) = 2,\ T_B(1) = 2,\ T_C(1) = 2$$

- 보통, n이 충분히 큰 경우에만 관심 있음



최선, 최악, 평균적 효율성

- 입력의 종류 또는 구성에 따라 다른 특성의 실행시간
  - 최선의 경우(best case) → 실행시간이 가장 짧음, 큰 의미 없음
  - 평균적인 경우(average case) → 평균적 실행시간, 정확히 계산하기 어렵다
  - 최악의 경우(worst case) → 입력의 구성에 따른 실행시간, 가장 중요



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

최선, 최악, 평균적 효율성

```
def sequential_search(A, key):    # 순차 탐색. A는 리스트, key는 탐색키
    for i in range(len(A)) :      # i : 0, 1, ... len(A)-1
        if A[i] == key :          # 탐색 성공하면 (비교 연산. 기본 연산임)
            return i              # 인덱스 반환
    return -1                     # 탐색에 실패하면 -1 반환
```

- **최선**

	0	1	2	3	4	5	6	7	8	9
리스트 A:	32	14	5	17	23	9	11	4	26	29

Key=32

key 발견 (0 반환)
비교 횟수 = 1

- **최악**

리스트 A:	32	14	5	17	23	9	11	4	26	29

Key=29

key 발견 (9 반환)
비교 횟수 = 10

- **평균**

$$T_{avg}(n) = \frac{1+2+\cdots+n}{n} = \frac{n(n+1)/2}{n} = \frac{n+1}{2}$$

[그림 출처] : 파이썬 알고리즘by 생능북스

점근적 표기법
(Asymptotic Notation)

복잡도의 점근적 표기

▪ 점근적 표기법(Asymptotic Notation)

- 복잡도 함수의 최고차항만을 계수 없이 취해 단순하게 표현
- 복잡도 함수에서 차수가 가장 큰 항이 절대적인 영향

  예: $T(n) = n^2 + n + 1$

  - n=1일때 : $T(n) = 1 + 1 + 1 = 3$ (n^2 항이 33.3%)
  - n=10일때 : $T(n) = 100 + 10 + 1 = 111$ (n^2 항이 90%)
  - n=100일때 : $T(n) = 10000 + 100 + 1 = 10101$ (n^2 항이 99%)
  - n=1,000일때 : $T(n) = 1000000 + 1000 + 1 = 1001001$ (n^2 항이 99.9%)

- n 이 무한대로 커질 때의 복잡도를 간단히 표현

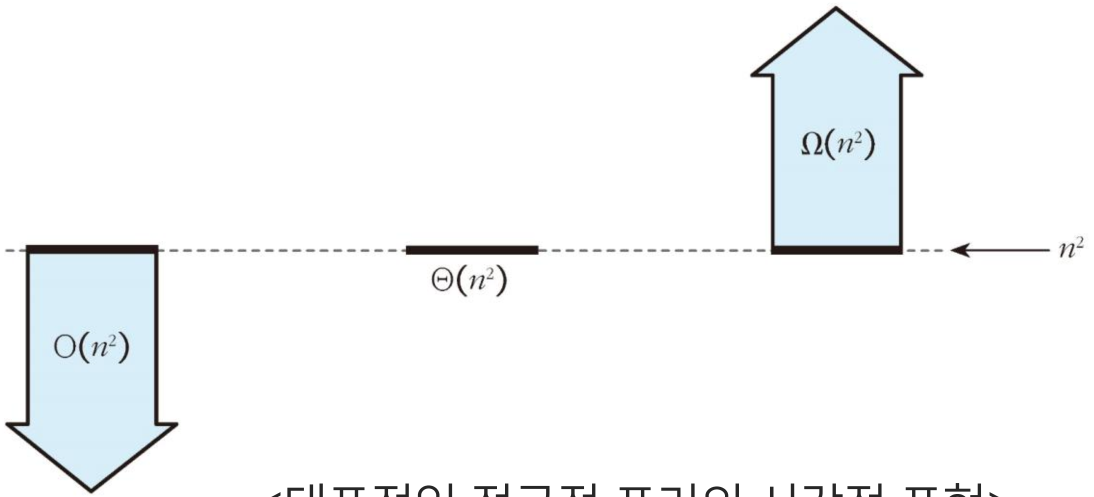
n=100인 경우

$T(n)= \underbrace{n^2}_{99\%} + \underbrace{n + 1}_{1\%}$

[그림 출처] : 파이썬 알고리즘by 생능북스

점근적 표기법

- **대표적인 점근적 표기법**
  - 최악의 경우(Worst Case) : 빅 오(Big-O, O) 표기법 (점근적 상한)
  - 평균적인 경우(Average Case) : 빅 세타(Big-Theta Ω) 표기법 (점근적 동일)
  - 최선의 경우(Best Case) : 빅 오메가(Big-Omega , Θ) 표기법 (점근적 하한)



효율성 분석에서 중요한 것은 n에 대해 연산이 정확히 몇 번 필요한가? 가 아니라 n이 증가함에 따라
'무엇에 비례하는 수의 연산이 필요한가?'

<대표적인 점근적 표기의 시각적 표현>

점근적 표기법

▪ 대표적인 점근적 표기법

- **빅 오(Big-O, O)** : 복잡도 함수의 상한

$$3n^2+4n \in O(n^2),\quad 2n-3 \in O(n^2),\quad 2n(n+1) \in O(n^2)$$
$$3n^2+4n \notin O(n),\quad 0.000001n^3 \notin O(n^2),\quad 1000^n \in O(n!)$$

- **빅 세타(Big-Theta Ω)** : 상한인 동시에 하한

$$2n^3+3n \in \Theta(n^3),\ 2n^3+3n \notin \Theta(n^2),\ 2n^3+3n \notin \Theta(n^4)$$

- **빅 오메가(Big-Omega , Θ)** : 하한

$$2n^3+3n \in \Omega(n^2),\ 2n(n+1) \in \Omega(n^2),\ 100000n+8 \notin \Omega(n^2)$$

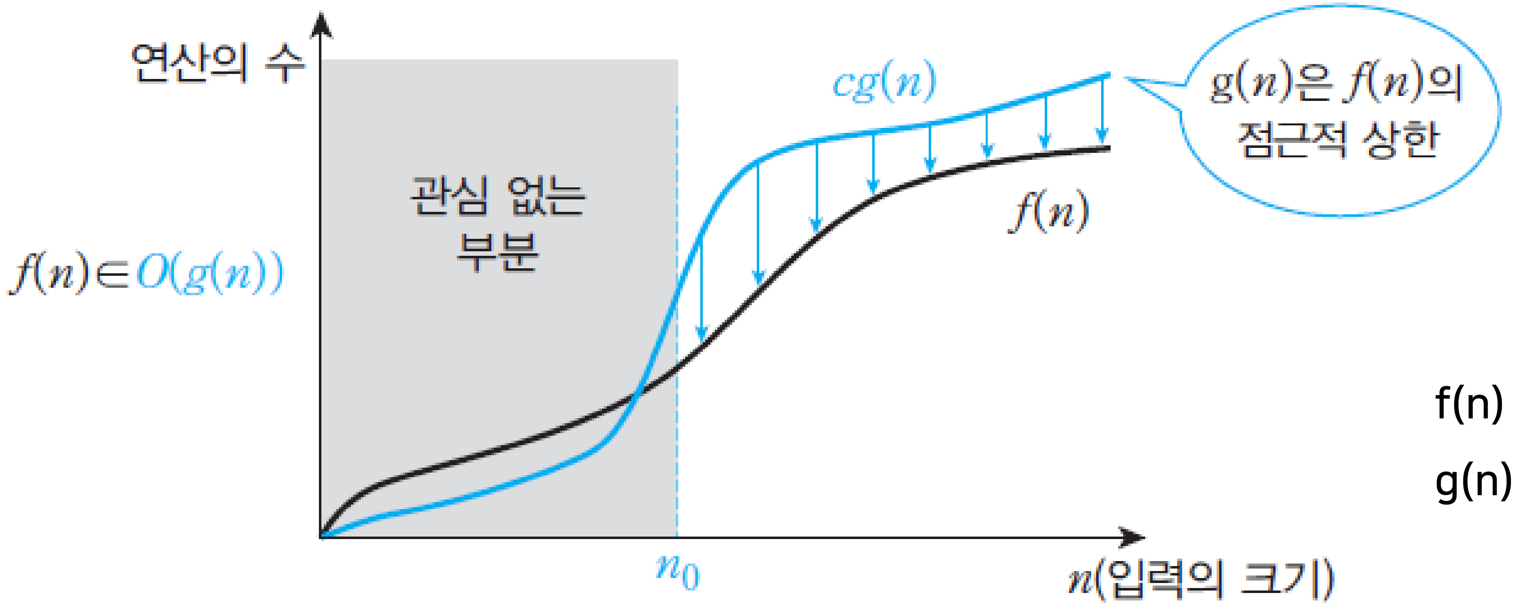
[그림 출처] : 파이썬 알고리즘by 생능북스

빅오 표기법 : $O(g(n))$

- 어떤 함수의 점근적 상한(upper bound) 을 표시하는 방법
  - 증가속도가 g(n)과 같거나 낮은 모든 복잡도 함수를 포함하는 집합

    → $O(n^2)$: 어떤 경우에도 n^2에 비례하는 시간 안에는 반드시 완료됨

worst-case



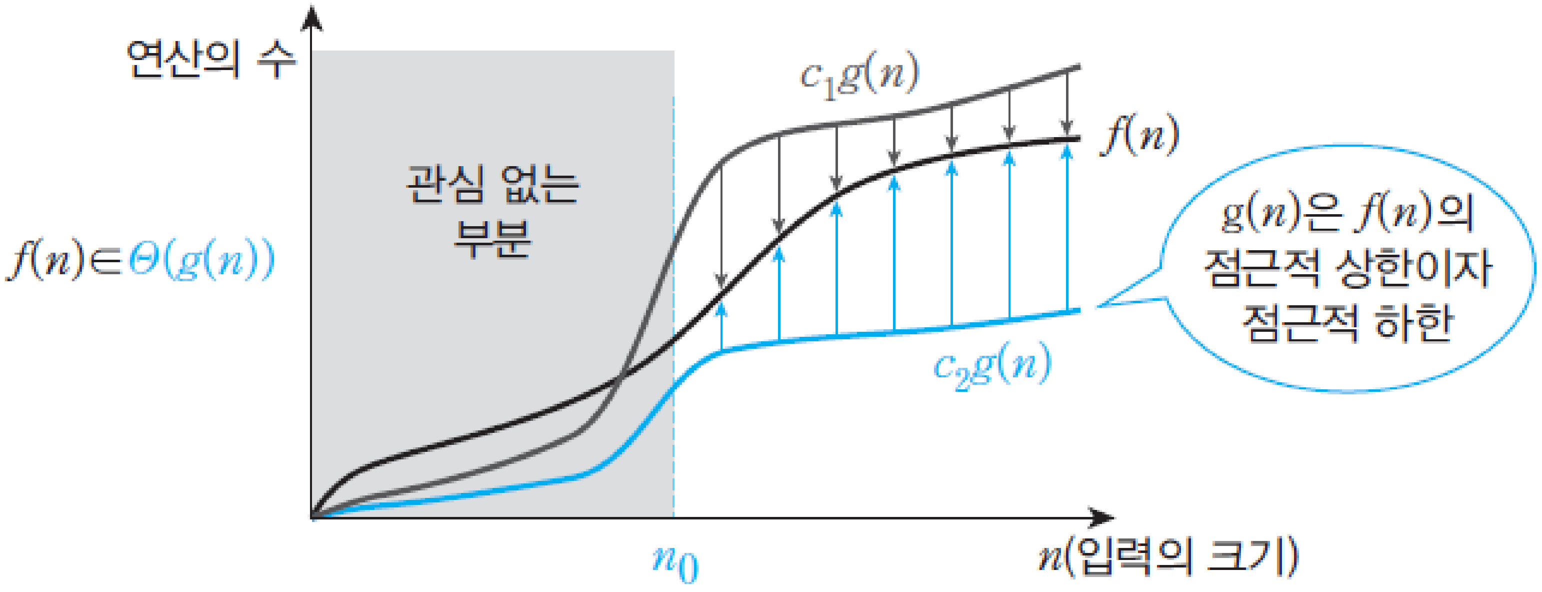
f(n) : 실제 연산량

g(n): f(n)의 점근적 상한

[그림 출처] 파이썬 알고리즘 by 생능북스

빅 세타 : Θ $(g(n))$

- 어떤 함수의 점근적 상한인 동시에 하한을 표시하는 방법 　　average case
  - 증가속도가 g(n)과 같은 모든 복잡도 함수를 포함하는 집합



f(n) : 실제 연산량

g(n): f(n)의 점근적 상한 또는 하한

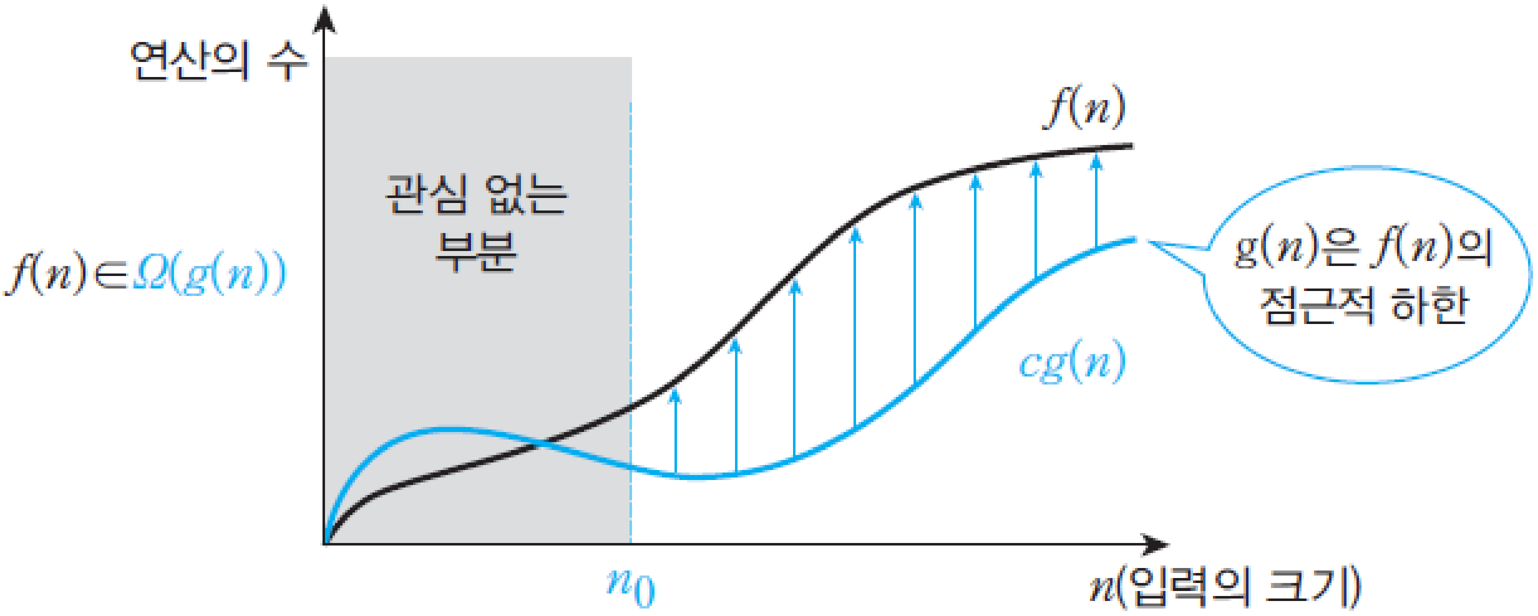
[그림 출처] 파이썬 알고리즘 by 생능북스

빅 오메가 : $\boldsymbol{\Omega(g(n))}$

- **어떤 함수의 점근적 하한 을 표시하는 방법**
  - 증가속도가 g(n)과 갈거나 높은 모든 복잡도 함수를 포함하는 집합

    → $\Omega(n^2)$: 아무리 빨리 처리하더라도 n^2에 비례하는 시간 이상은 반드시 걸린다

best-case



f(n) : 실제 연산량

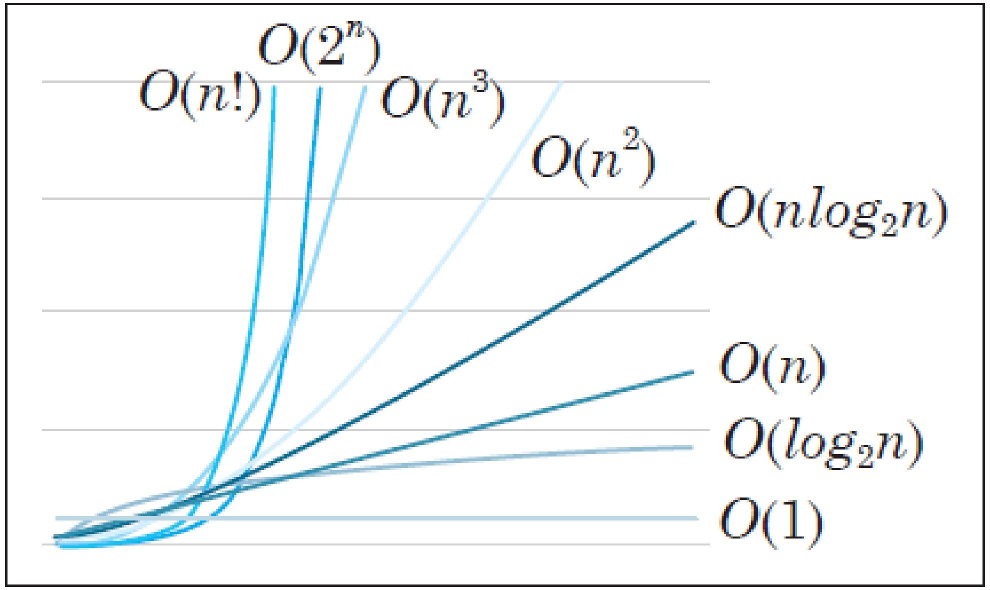
g(n): f(n)의 점근적 하한

[그림 출처] 파이썬 알고리즘 by 생능북스

점근적 성능 클래스

- 자주 사용되는 빅오 표기의 시간 복잡도

$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n!)$$



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

점근적 성능 클래스

- 시간 복잡도와 알고리즘 예 :

시간 복잡도	알고리즘 예시	성능 특징
O(1)	해시 테이블 조회	입력 크기에 무관
O(log n)	이진 탐색	데이터가 커도 빠름
O(n)	선형 탐색	데이터 크기에 비례
O(n log n)	병합 정렬	비교적 효율적
O(n^2)	버블 정렬	데이터 크면 매우 느림
O(2^n)	피보나치(재귀)	지수적으로 증가

[Quiz]

- 다음 시간 복잡도 함수들의 증가 속도를 =,>,< 기호와 빅오 표기법으로 표시

1) $n(n-100)(n-2000)$ $\boxed{}$ $50000n^3$ $\boxed{O(\quad)}$

2) $0.00001n^2$ $\boxed{}$ $100000n$ $\boxed{O(\quad)}$

3) 1000^n $\boxed{}$ $n!$ $\boxed{O(\quad)}$

[Quiz]

- 다음 알고리즘의 성능을 분석하시오.

```
01:  def find_max( A ):
02:      n = len(A)
03:      max = A[0]
04:      for i in range(n) :
05:          if A[i] > max :
06:              max = A[i]
07:      return max
```

1) 입력의 크기는?
2) 가장 많이 처리되는 행은?
3) 입력에 따른 효율성의 차이가 있는가?

[Quiz]

- 다음 알고리즘의 성능을 분석하시오

```
def find_key( A, key ):
    n = len(A)
    for i in range(n) :
        if A[i] == key :
            return i
    return -1
```

1) 입력의 크기는?
2) 가장 많이 처리되는 행은?
3) 입력에 따른 효율성의 차이가 있는가?

※ 가천대학교 컴퓨터공학과 '알고리즘' 과정

재귀 알고리즘

점화식

수열

- 수열(sequence)
  - 수 또는 다른 대상들이 한 줄로 배열된 순서 있는 나열로
    $a1, a2, a3, \cdots, a_n$의 형태로 표시
  - 반드시 일정한 규칙을 갖고 수가 나열되어야 할 필요는 없다
  - 일반적으로 일정한 규칙을 가진 수열을 많이 다루며,
    규칙에 따라 등차수열, 등비수열, 조화수열, 점화수열 등으로 구분한다

점화관계

- 점화관계(recurrence relation)란?

점화 관계recurrence relation 혹은 **점화식**recurrence formula은 수열 $a_0, a_1, \cdots, a_n$에서 a_n과 그 앞의 항들인 $a_0, a_1, a_2, \cdots, a_{n-1}$과의 종속 관계를 말한다. 즉, 수열에서 n번째 원소와 그 앞의 원소와의 관계를 나타낸다.

- 수학적 귀납법과 재귀적 알고리즘과 매우 밀접한 관계를 가지고 있다.

P(k)가 참이면,
n ≥k인 모든 정수 n에 의해 P(n)가 참이라고 가정하면 P(n+1)도 참이다

점화식

- 점화식(recurrence formula)이란?
  - 어떤 함수를 자신보다 더 작은 변수에 대한 함수와의 관계로 표현한 것

$$a_n = a_{n-1} + 2$$
$$f(n) = n\,f(n-1)$$
$$f(n) = f(n-1) + f(n-2)$$
$$f(n) = f(n/2) + n$$
$$\dots$$

점화식 분석 방법

- 대표적 점화식 분석 방법
  - 반복 대치 (Iteration Method)
    - 작은 문제에 대한 값을 계속 대입하여 패턴을 찾고 일반적인 점근적 표현을 유도하는 방법
    - 재귀 함수의 반복적인 호출을 따라가면서 직접 계산하여 결과를 도출
  - 추정 후 증명 (Guess & Verification)
    - 점화식의 해를 추정(Guessing) 한 후, 수학적 귀납법을 이용해 검증하는 방법
    - 주어진 점화식을 해결하기 어려울 때, 특정한 패턴을 예상하고 이를 증명하여 점근적 경계를 찾음
  - 마스터 정리(Master Theorem)
    - 점화식이 특정한 형태일 때 바로 복잡도를 결정할 수 있음

$$T(n) = aT(n/b) + O(n^d)$$

점화식 분석 방법 예

- **점화식 예 : 병합 정렬의 수행시간**
  - 재귀적 관계를 이용해 알고리즘의 수행 시간을 점화식으로 표현할 수 있음

```
mergeSort(A[], p, r):  ▷ A[p...r]을 정렬한다.
    ❶ if (p < r)
        ❷ q ← ⌊ (p+r)/2 ⌋            ▷ p, r의 중간 지점 계산
        ❸ mergeSort(A, p, q)         ▷ 전반부 정렬
        ❹ mergeSort(A, q+1, r)       ▷ 후반부 정렬
        ❺ merge(A, p, q, r)          ▷ 병합
```

  - 수행 시간의 점화식 (크기가 n인 병합 정렬 시간)

= 크기가 $\frac{n}{2}$인 병합 정렬을 두 번하는 시간과 나머지 오버헤드를 더한 시간

$$\boxed{T(n) = 2T(\frac{n}{2}) + \text{후처리 시간}}$$

점화식에서 사용되는 연산

- 연산 유형별 점화식 예 :

연산 유형	점화식	빅오 표기법
상수 연산	$T(n) = O(1)$	$O(1)$
선형 반복문	$T(n) = T(n-1) + O(1)$	$O(n)$
이중 루프	$T(n) = T(n-1) + O(n)$	$O(n^2)$
로그 연산	$T(n) = T(n/2) + O(1)$	$O(logn)$
분할 정복	$T(n) = 2T(n/2) + O(n)$	$O(nlogn)$
지수 연산	$T(n) =T(n-1) +T(n-2) + O(1)$	$O(2^n)$

실습 : 피보나치의 토끼 번식 문제

- 점화 수열의 대표적인 문제인 피보나치의 토끼 번식 문제

1) 1년이 지난 후(13월) 전체 토끼의 쌍의 수는?

개월	처음	1개월	2개월	3개월	4개월	5개월	…
어미 토끼(쌍)	1	1	1	2	3	5	…
새끼 토끼(쌍)	–	–	1	1	2	3	…
전체 쌍의 수	1	1	2	3	5	8	…

2) 점화식은?

3) 빅오 표기법은?

4) 파이썬으로 구현 하시오.

재귀 알고리즘

순환

- 순환(Recursion)이란?
  - 어떤 함수가 자기 자신을 다시 호출하여 문제를 해결하는 프로그래밍 기법
    - ➢ 문제 해결을 위한 독특한 구조를 제공
    - ➢ 많은 효율적인 알고리즘 들에서 사용됨
  - 문제 자체가 순환적이거나 순환적으로 정의되는 자료구조를 다루는데 적합
    - ➢ 문제 자체가 순환적인 경우 (예) 팩토리얼 계산, 하노이 탑 등
    - ➢ 순환적으로 정의되는 자료구조인 경우 (예) 이진 트리

순환

- **반복 vs 순환**
  - n! 구하기

반복

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

```python
def factorial_iter(n) :
    result = 1
    for k in range(2, n+1) :
        result = result * k
    return result
```

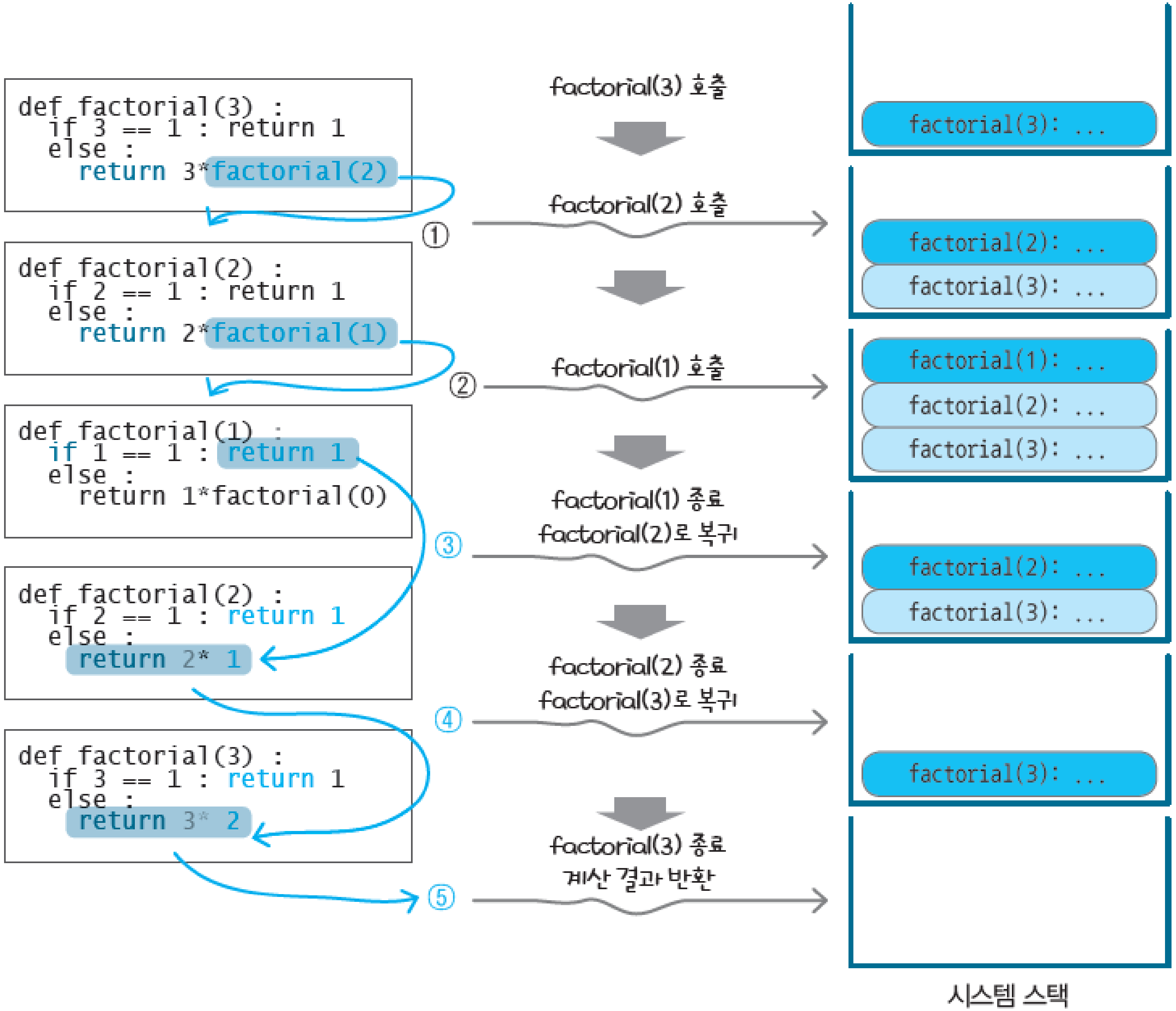
순환

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

```python
def factorial(n) :
    if n == 1 : return 1
    else :
        return n * factorial(n - 1)
```

순환 호출의 예

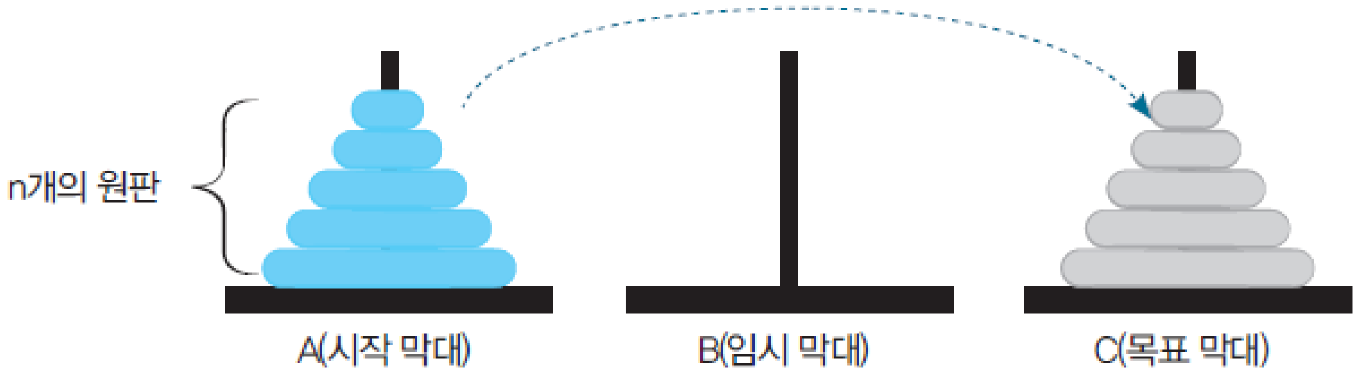
- 순환적인 팩토리얼 함수 동작



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

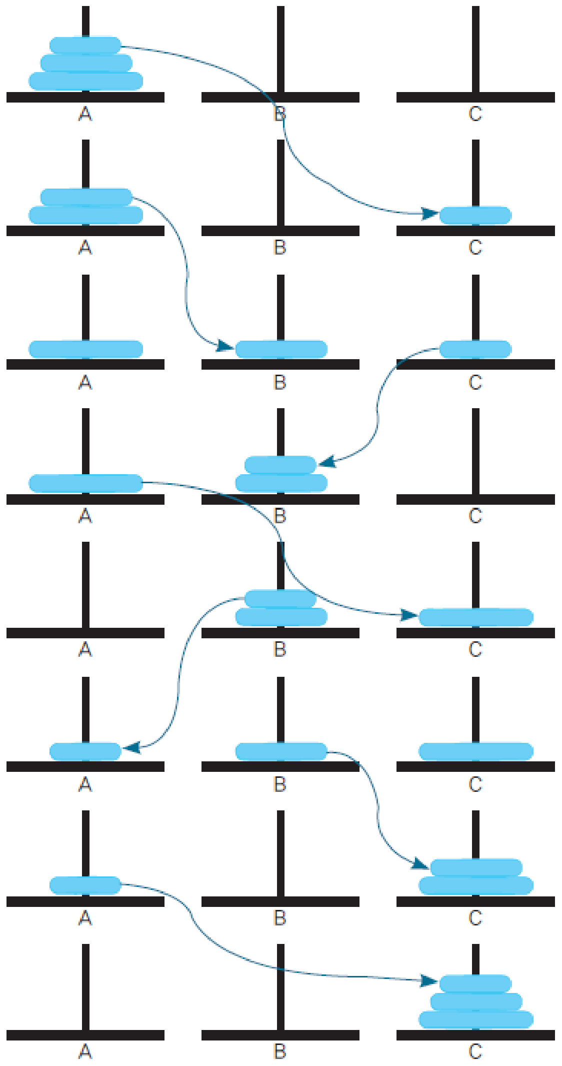
순환 호출의 예

- 하노이의 탑



막대 A에 쌓여 있는 n개의 원판을 모두 C로 옮기는 문제입니다. 단, 다음과 같은 조건을 만족해야 합니다.

- 한 번에 하나의 원판만 옮길 수 있습니다.
- 맨 위에 있는 원판만 옮길 수 있습니다.
- 크기가 작은 원판 위에 큰 원판을 쌓을 수는 없습니다.
- 중간 막대 B를 임시 막대로 사용할 수 있지만 앞의 조건은 지켜야 합니다.



[그림 출처] : 자료구조와 알고리즘 with 파이썬 by 생능북스

재귀 알고리즘

- 재귀 알고리즘(recursive algorithm)
  - 문제를 해결을 위해 동일한 형태의 작은 문제로 반복적으로 축소시켜 나가는 알고리즘.
  - 재귀(순환) 호출(Recursion: 자신을 다시 호출)을 사용한다.

재귀 알고리즘의 주요 구성

- 재귀 알고리즘의 주요 구성
  1. 기본 사례(Base Case): 가장 작은 문제에 대한 해답을 미리 정의해 놓음 (stop 조건)
  2. 재귀 단계(Recursive Step): 현재 문제를 작은 인스턴스의 문제로 분할하고,
     그 부분 문제에 대해 자기 자신을 호출함
  3. 반환(Return) : 상태를 변경하고 기본 사례로 이동해야 한다.

```
def factorial(n):
    if n == 1:                # 1.Base Case
        return 1
    return n * factorial(n-1) # 2.Recursive Step

factorial(5)
```

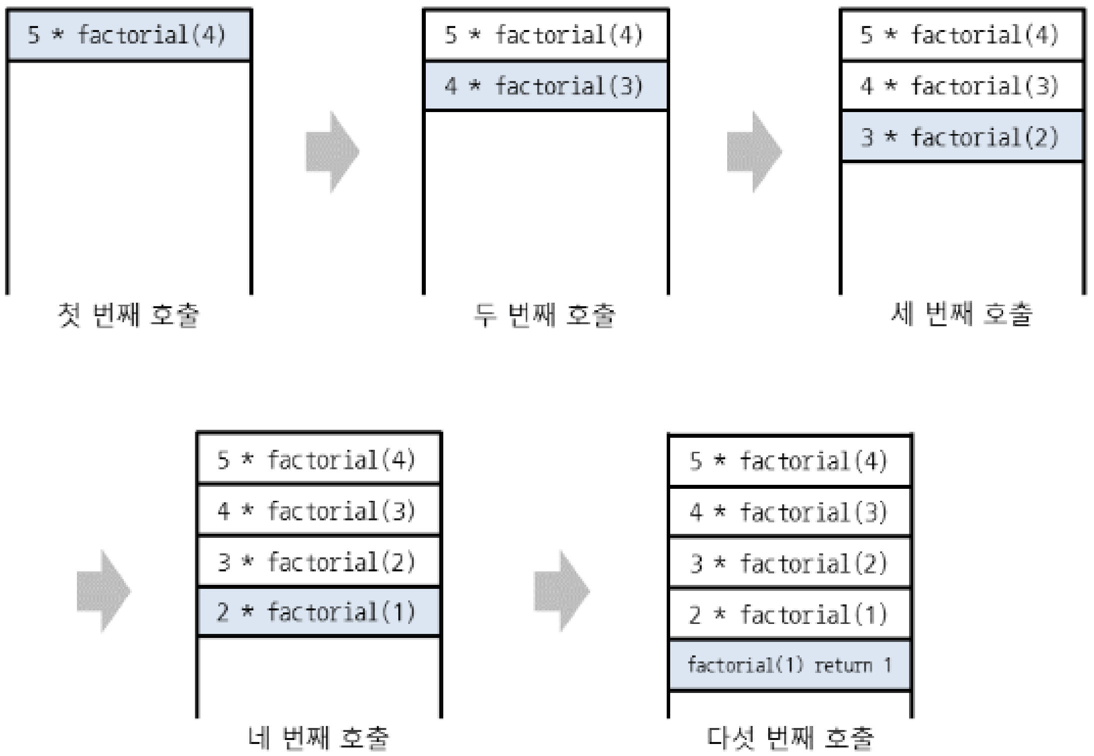
재귀 알고리즘 동작 원리

- 재귀 알고리즘 동작 원리
  ① 재귀 호출 과정
  ② 기본 사례 도달
  ③ 스택에서 값 반환
  ④ 재귀 종료

```python
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

factorial(5)
```

[호출 스택(Call Stack)]



https://pythontutor.com/

재귀 알고리즘의 특징

- 주요 특징
  - **간결성**: 반복문을 사용하지 않고도 복잡한 문제를 간결하게 표현할 수 있음.
  - **직관성**: 문제의 구조를 직관적으로 반영하여 코드를 작성할 수 있음
  - **유연성**: 다양한 문제에 적용할 수 있으며, 특히 분할 정복 방식 문제에 효과적임
  - 종료조건: 무한 루프를 방지하기 위해 반드시 종료 조건을 명확하게 정의해야 함
- 단점
  - **효율성**: 반복문을 사용하는 알고리즘보다 실행 속도가 느릴 수 있음
  - **메모리 사용량**: 재귀 호출은 스택 메모리 사용으로 메모리 사용량이 많아질 수 있음.
  - **디버깅 어려움**: 재귀 알고리즘은 디버깅하기 어려울 수 있음

실습문제 : 재귀 알고리즘 적용하기

- 파이썬으로 재귀 알고리즘을 구현하세요.(<u>without using loops</u>)
  1) countdown(n) : 5->4->3->2->1->'발사' 순서로 출력하기
  2) printStar(n) : 별 모양 출력하기
  3) addNumber(n) : 1~10까지 합계 구하기
  4) sum_range(start, end) : 임의의 두 수 사이의 정수 합계 구하기 (1~100 정수)
  5) reverse(s) : 문자열 뒤집기

회문(Palindrome) 여부 판단하기

회문 여부 판단하기

- 회문(Palindrome) 여부 판단하기
  - 회문(Palindrome): 앞에서부터 읽든, 뒤에서부터 읽든 동일한 단어나 문장을 의미
  - ex)

```
level
kayak
radar
Borrow or rob
I prefer pi
기러기
일요일
주유소의 소유
다 큰 도라지일지라도 큰다
야 너 이번 주 주번이 너야
야 이 달은 밝은 달이야
마지막 날 날 막지 마
```

실습문제 : 문자열 회문 여부 판단하기

- 아래 문자열이 회문인지 판단하는 파이썬 함수를 구현하세요.

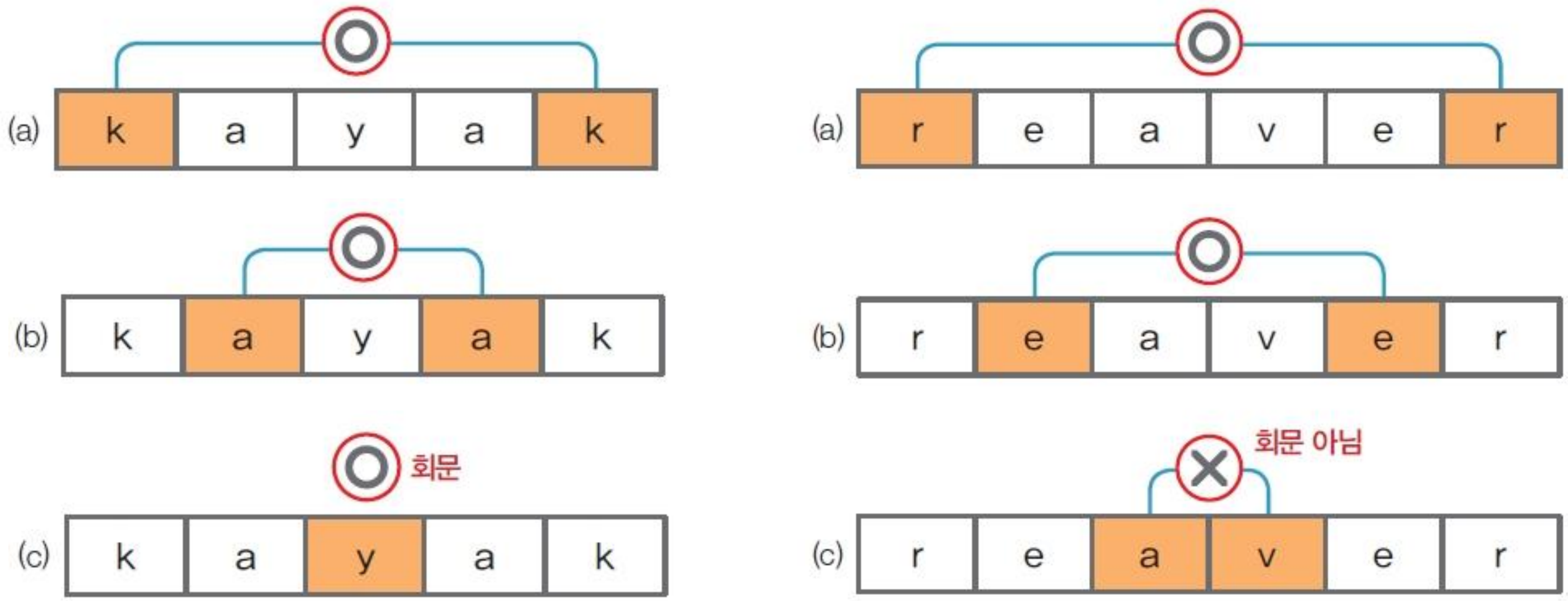
```
def is_palindrome(tStrings):   # 회문 여부 판별
```

```
is_palindrome(['reaver', 'level','기러기','살금 살금'])
```

```
reaver   --> False
level    --> True
기러기   --> True
살금살금 --> False
```

회문 여부 판단하기

- 회문(Palindrome) 여부 판단 동작 원리
  - ex) kayak (O),   reaver (X)



실습문제 : 문자열 회문 여부 판단하기

- 앞에서 확인한 회문 판단 동작 원리를 반영하여 재귀 알고리즘으로 회문 여부를 판단하는 파이썬 함수(*palindrome()*)를 구현하세요.

```python
if tStr == tStr[::-1]:
    print(f"{tStr}\t--> True")
```

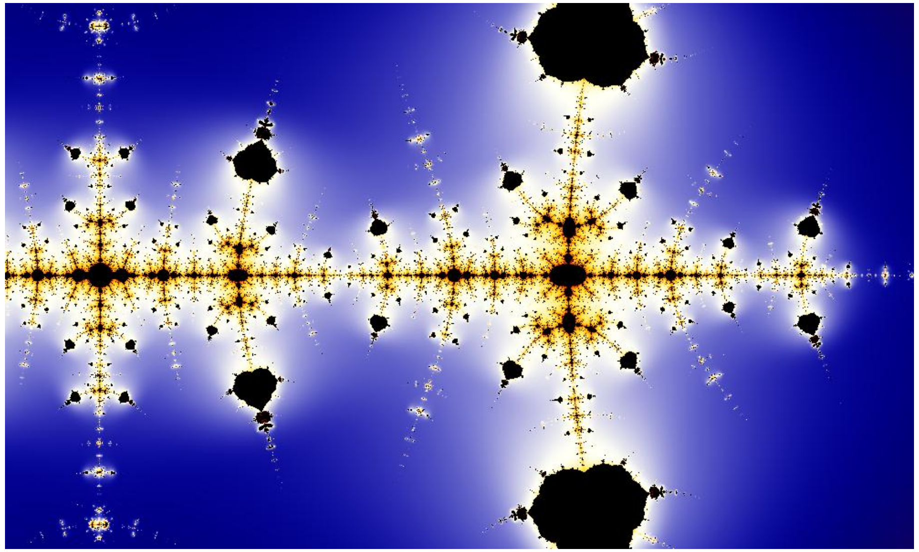
↓

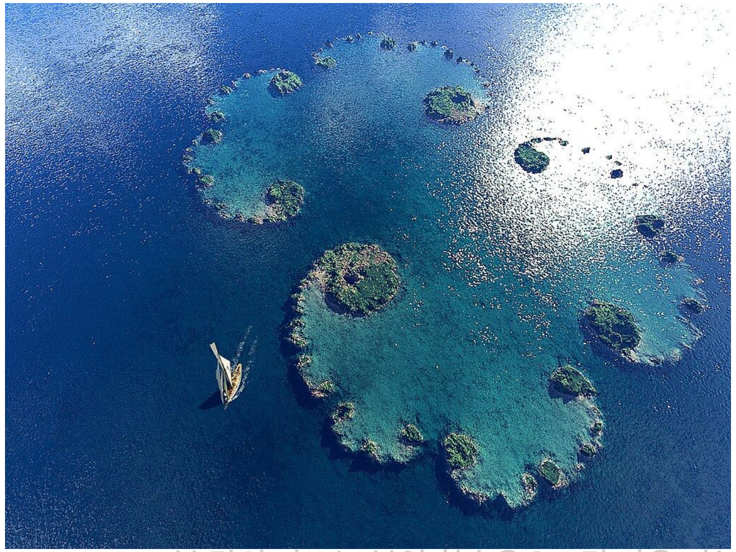
```python
if palindrome(tStr):
    print(f"{tStr} --> True")
```

프랙탈 그리기

프랙탈

- 프랙탈(Fractal)
  - 프랙탈은 작은 조각이 전체와 비슷한 기하학적인 형태를 의미
  - 자기 유사성(Self-Similarity)을 전제로 끊임없이 자기 복제를 반복하는 특징을 가짐
  - 부분을 확대하면 전체와 동일한(or 닮은 꼴 모습)을 나타내는 성질이 있음





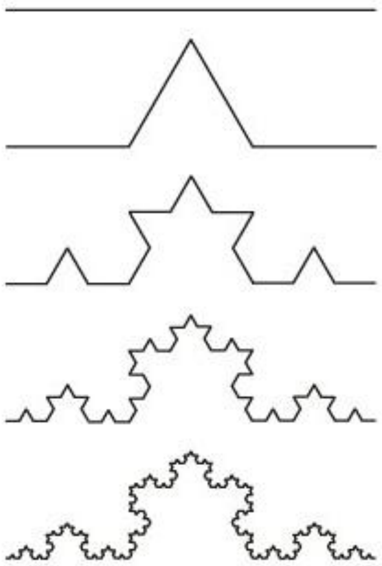
이미지 출처: 위키 피디아

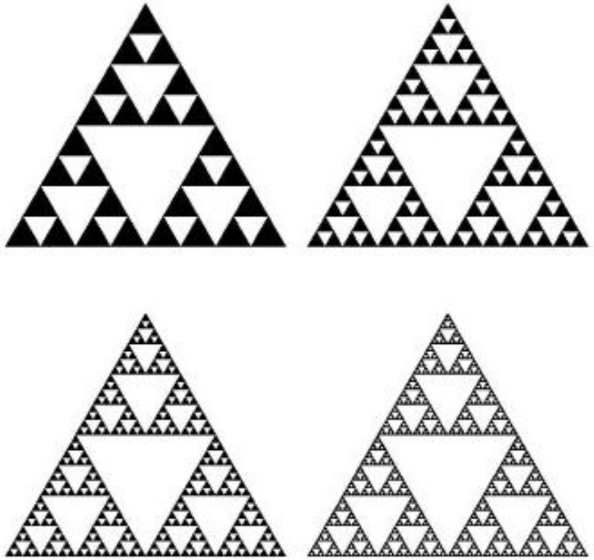
재귀 알고리즘과 프랙탈의 관계

- 재귀 알고리즘과 프랙탈의 관계
  - **재귀 알고리즘 :** 문제를 더 작은 하위 문제로 분할하고 해결하는 데 사용
  - **프랙탈 :** 재귀적 구조를 가진 도형 또는 패턴, 자기 유사성(self-similarity)을 가짐

  → 프랙탈은 재귀 알고리즘을 사용하여 만들 수 있다.

코흐 곡선
(Koch Curve)

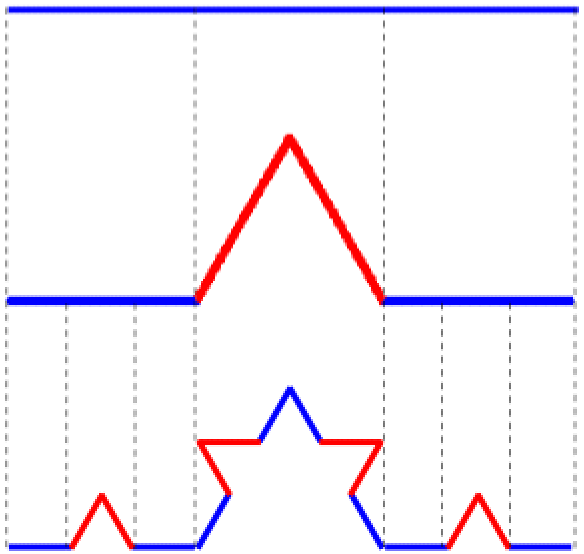


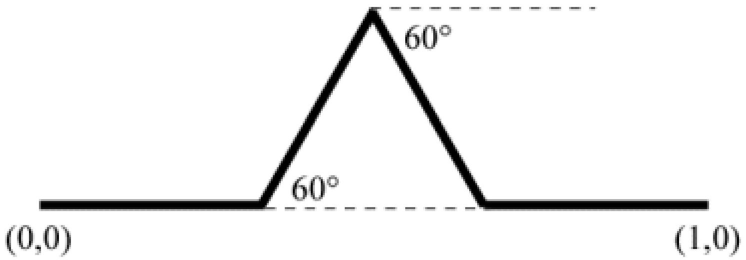


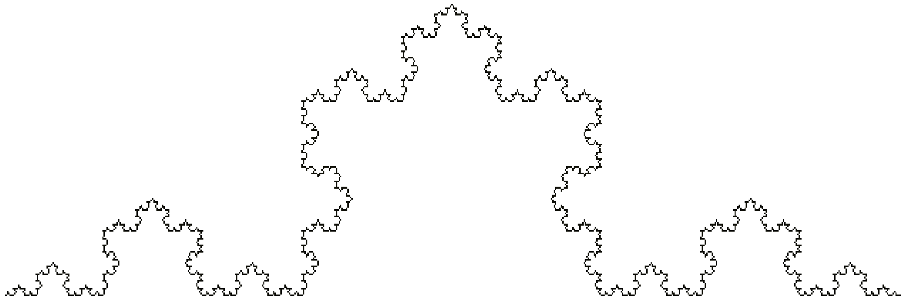
시어핀스키 삼각형
(Sierpinski Gasket)

코흐 곡선

- 코흐 곡선(Koch Curve)
  - 프랙탈 기하학에서 자주 사용되는 대표적인 예시
  - 둘레의 길이는 무한대로 늘어나는 반면 넓이는 유한하다는 특성이 있다.



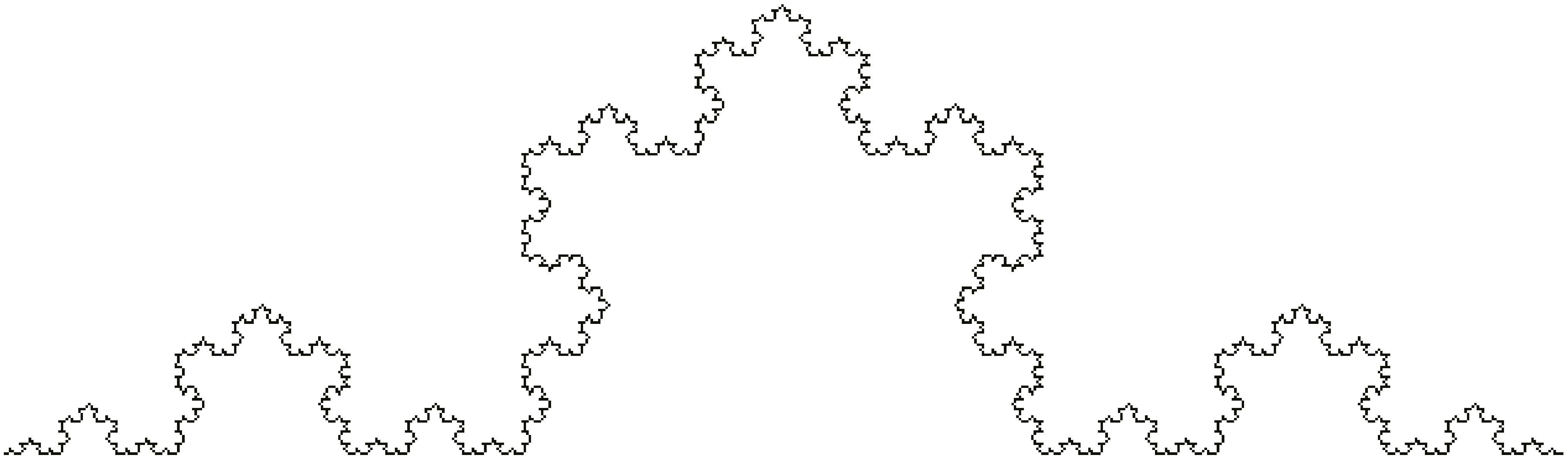




https://larryriddle.agnesscott.org/ifs/kcurve/kcurve.htm

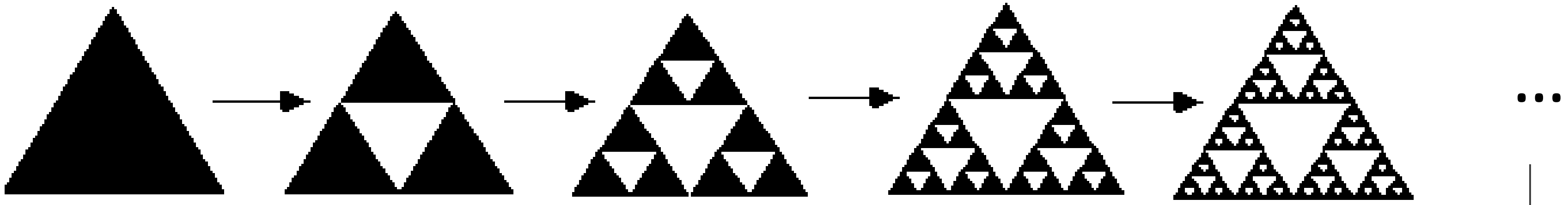
실습문제 : 코흐 곡선 그리기

- 재귀 알고리즘을 이용하여 파이썬으로 코흐 곡선을 그리시오.



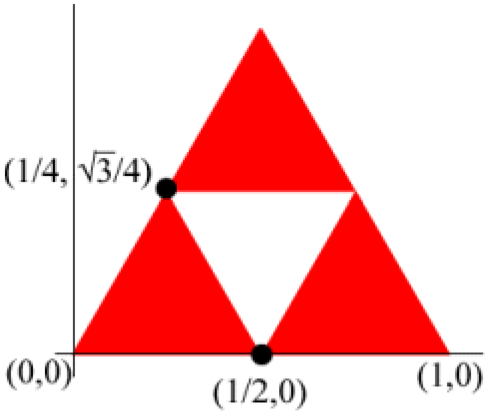
시어핀스키 삼각형

- 시어핀스키 삼각형(Sierpinski Gasket)
  - 프랙탈 기하학에서 자주 사용되는 대표적인 예시
  - 둘레의 길이는 무한대로 늘어나는 반면 넓이는 0으로 수렴하는 특성이 있다.



- 처음 삼각형의 면적 = S
- 두 번째 단계 = ¾ * S
- 세 번째 단계 = ¾ * ¾ * S
- …
- 결국에는 0으로 수렴

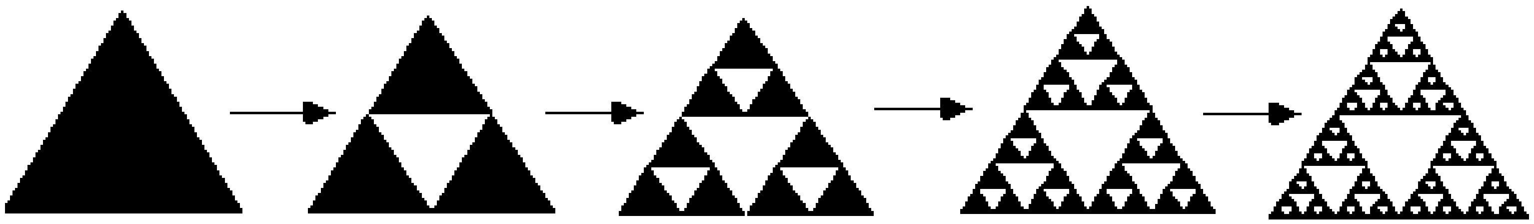
- 처음 삼각형의 둘레 = 3 * L
- 두 번째 단계 = 3 * ½ *L * 3
- 세 번째 단계=
- …
- 결국에는 무한대로 수렴

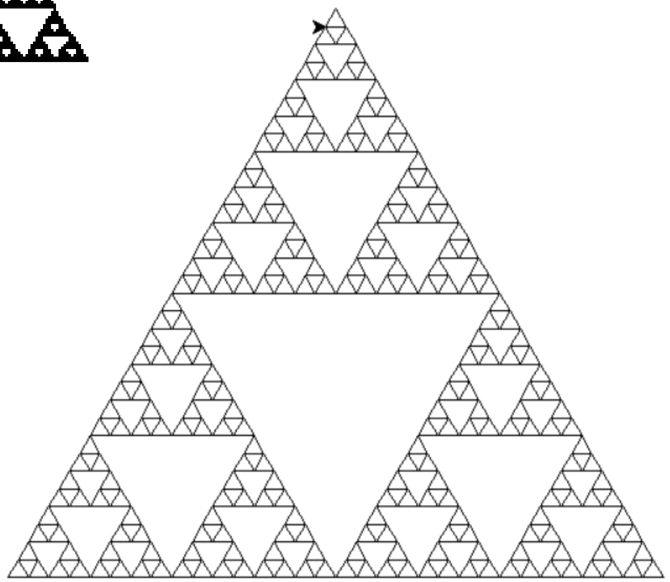


https://larryriddle.agnesscott.org/ifs/siertri/siertri.htm

실습문제 : 시어핀스키 그리기

- 재귀 알고리즘을 이용하여 파이썬으로 시어핀스키 삼각형을 그리시오.





Q & A

Next Topic

- 정렬 알고리즘(기초)

Keep learning, see you soon!