

제1장 자료구조를 배우기 위한 준비

자료구조

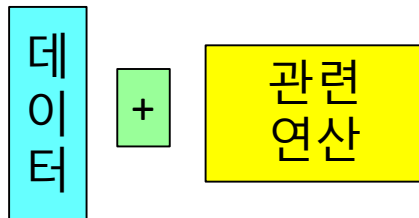
- 자료구조(Data Structure): 일련의 동일한 타입의 데이터를 정돈하여 저장한 구성체
- 데이터를 정돈하는 목적: 프로그램에서 저장하는 데이터에 대해 탐색, 삽입, 삭제 등의 연산을 효율적으로 수행하기 위해서
- 자료구조를 설계할 때에는 데이터와 데이터에 관련된 연산들을 함께 고려해야

추상데이터타입

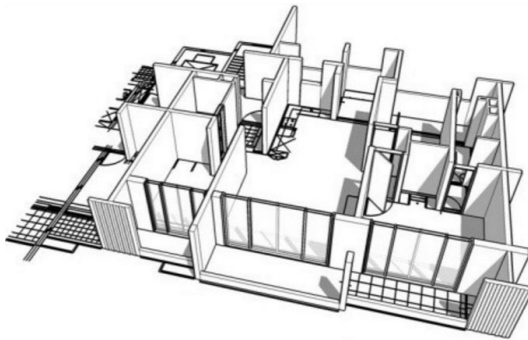
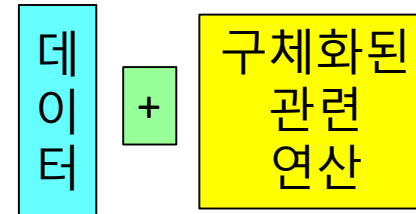
- 추상데이터타입(Abstract Data Type)은 데이터와 그 데이터에 대한 추상적인 연산들로써 구성
- ‘추상’의 의미: 연산을 구체적으로 어떻게 구현하여야 한다는 세부 명세를 포함하고 있지 않다는 의미
- 자료구조는 추상데이터타입을 구체적(실제 프로그램)으로 구현한 것

추상데이터타입과 자료구조 관계

추상데이터타입
(ADT)



자료구조



1-2 수행시간의 분석

- 자료구조의 효율성은 자료구조에 대해 수행되는 연산의 수행시간으로 측정
- 자료구조에 대한 연산 수행시간 측정 방식은 알고리즘의 성능을 측정하는 방식과 동일
- 알고리즘의 성능: 수행시간을 나타내는 시간복잡도(Time Complexity)와 알고리즘이 수행되는 동안 사용되는 메모리 공간의 크기를 나타내는 공간복잡도(Space Complexity)에 기반하여 분석

- 대부분의 경우 시간복잡도만을 사용하여 알고리즘의 성능을 분석, 주어진 문제를 해결하기 위한 대부분의 알고리즘들이 비슷한 크기의 메모리 공간을 사용하므로
- 알고리즘의 성능은 알고리즘을 구현한 프로그램을 실제로 컴퓨터에서 실행시킨 후 실행 완료까지 소요된 시간으로 측정할 수도 있다.
- 실제 측정된 시간으로 알고리즘의 성능을 객관적으로 평가하는 데는 한계가 존재한다.
- 프로그래머의 숙련도, 구현에 사용된 프로그래밍 언어의 종류 그리고 알고리즘을 실행한 컴퓨터의 성능에 따라서 수행시간은 얼마든지 달라질 수 있기 때문

시간복잡도

- 시간복잡도는 알고리즘(연산)이 실행되는 동안에 사용된 기본적인 연산 횟수를 입력 크기의 함수로 나타낸다.
- 기본 연산(Elementary Operation)이란 탐색, 삽입이나 삭제와 같은 연산이 아닌, 데이터 간 크기 비교, 데이터 읽기 및 갱신, 숫자 계산 등과 같은 단순한 연산을 의미

3 종류의 분석

- 최악경우 분석(Worst-case Analysis)
- 평균경우 분석(Average-case Analysis)
- 최선경우 분석(Best-case Analysis)

- 일반적으로 알고리즘의 수행시간은 최악경우 분석으로 표현
- **최악경우 분석**: ‘어떤 입력이 주어지더라도 알고리즘의 수행시간이 얼마 이상은 넘지 않는다’라는 상한(Upper Bound)의 의미
- **평균경우 분석**: 입력의 확률 분포를 가정하여 분석하는데, 일반적으로 균등분포(Uniform Distribution)를 가정
- **최선경우 분석**: 가장 빠른 수행시간을 분석
 - 최적(Optimal) 알고리즘을 찾는데 활용

등교 시간 분석

- 집을 나와서 지하철역까지는 5분, 지하철을 타면 학교까지 20분, 강의실까지는 걸어서 10분 걸린다
- **최선경우:** 집을 나와서 5분 후 지하철역에 도착하고, 운이 좋게 바로 열차를 탄 경우를 의미한다. 따라서 최선경우 시간은 $5 + 20 + 10 = 35$ 분
- **최악경우:** 열차에 승차하려는 순간, 열차의 문이 닫혀서 다음 열차를 기다려야 하고 다음 열차가 10분 후에 도착한다면, 최악경우는 $5 + 10 + 20 + 10 = 45$ 분

- 평균 시간: 대략 최악과 최선의 중간이라고 가정했을 때, 40분이 된다.



(a) 최선 경우



(b) 최악 경우



(c) 평균 경우

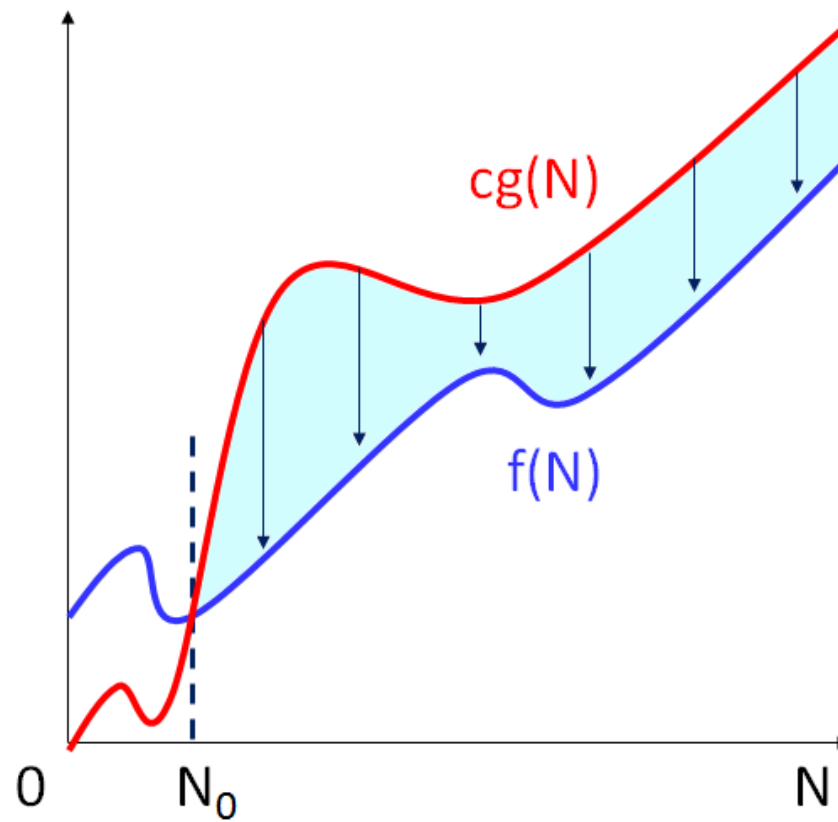
1-3 수행시간의 점근표기법

- 수행시간은 알고리즘이 수행하는 기본 연산 횟수를 입력 크기에 대한 함수로 표현
- 이러한 함수는 다항식으로 표현되며 이를 입력의 크기에 대한 함수로 표현하기 위해 점근표기법(Asymptotic Notation)이 사용
- O (Big-Oh)-표기법
- Ω (Big-Omega)-표기법
- Θ (Theta)-표기법

O (Big-Oh)-표기법

[O-표기의 정의]

- 모든 $N \geq N_0$ 에 대해서 $f(N) \leq cg(N)$ 이 성립하는 양의 상수 c 와 N_0 이 존재하면, $f(N) = O(g(N))$ 이다.
- O-표기의 의미: N_0 과 같거나 큰 모든 N (즉, N_0 이후의 모든 N)에 대해서 $f(N)$ 이 $cg(N)$ 보다 크지 않다는 것
- $f(N) = O(g(N))$ 은 N_0 보다 큰 모든 N 에 대해서 $f(N)$ 이 양의 상수를 곱한 $g(N)$ 에 미치지 못한다는 뜻
- $g(N)$ 을 $f(N)$ 의 **상한(Upper Bound)**이라고 한다.



$$f(N) = O(g(N))$$

[예제]

- $f(N) = 2N^2 + 3N + 5$ 이면, 양의 상수 c 값을 최고 차항의 계수인 2보다 큰 4를 택하고 $g(N) = N^2$ 으로 정하면, 3보다 큰 모든 N 에 대해 $2N^2 + 3N + 5 < 4N^2$ 이 성립, 즉, $f(N) = O(N^2)$
- 물론 $2N^2 + 3N + 5 = O(N^3)$ 도 성립하고, $2N^2 + 3N + 5 = O(2^N)$ 도 성립한다. 그러나 $g(N)$ 을 선택할 때에는 정의를 만족하는 가장 차수가 낮은 함수를 선택하는 것이 바람직함
- $f(N) \leq cg(N)$ 을 만족하는 가장 작은 c 값을 찾지 않아도 됨. 왜냐하면 $f(N) \leq cg(N)$ 을 만족하는 양의 상수 c 와 N_0 가 존재하기만 하면 되기 때문

- 주어진 수행시간의 다항식에 대해 o-표기를 찾기 위해 간단한 방법은 다항식에서 최고 차수 항만을 취한 뒤, 그 항의 계수를 제거하여 $g(N)$ 을 정한다.
- 예를 들어, $2N^2 + 3N + 5$ 에서 최고 차수항은 $2N^2$ 이고, 여기서 계수인 2를 제거하면 N^2 이다.

$$2N^2 + 3N + 5 = O(N^2)$$

Ω-표기법

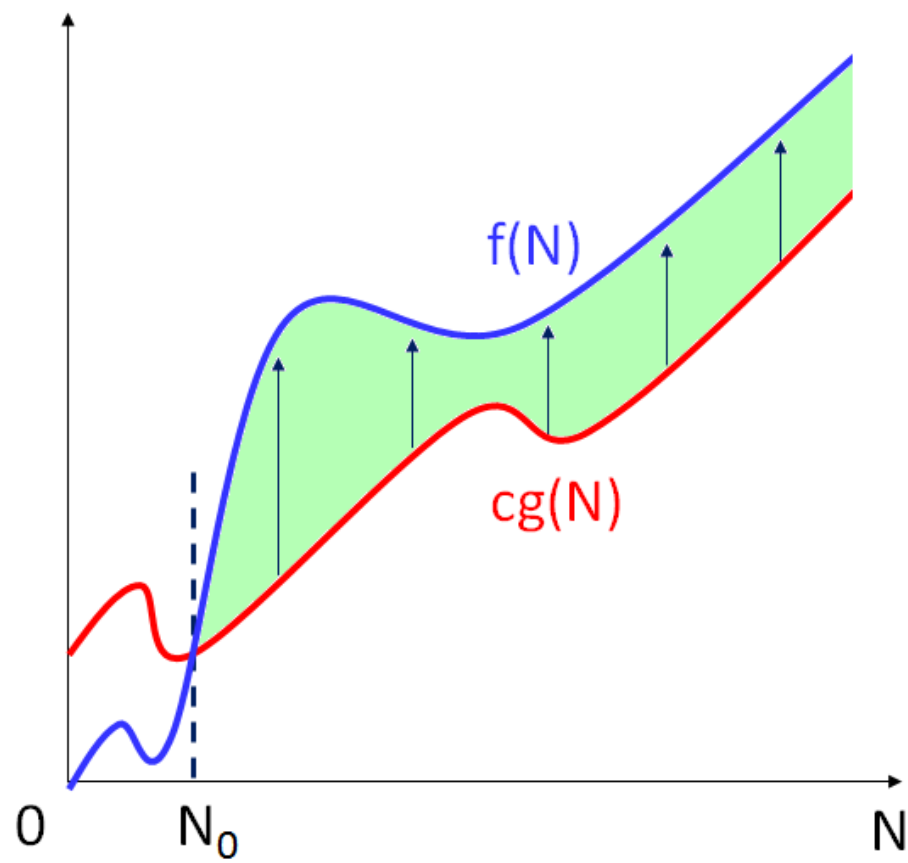
[Ω-표기의 정의]

- 모든 $N \geq N_0$ 에 대해서 $f(N) \geq cg(N)$ 이 성립하는 양의 상수 c 와 N_0 이 존재하면, $f(N) = \Omega(g(N))$ 이다.

- Ω-표기의 의미는 N_0 보다 큰 모든 N 대해서 $f(N)$ 이 $cg(N)$ 보다 작지 않다는 것
- $f(N) = \Omega(g(N))$ 은 양의 상수를 곱한 $g(N)$ 이 $f(N)$ 에 미치지 못한다는 뜻
- $g(N)$ 을 $f(N)$ 의 **하한(Lower Bound)**이라고 함

[예제]

- $f(N) = 2N^2 + 3N + 5$ 일 때, 양의 상수 $c = 1$ 로 택하고 $g(N) = N^2$ 으로 정하면, 1보다 큰 모든 N 에 대해 $2N^2 + 3N + 5 > N^2$ 이 성립한다. 따라서 $f(N) = \Omega(N^2)$
- 물론 $2N^2 + 3N + 5 = \Omega(N)$ 도 성립하고, $2N^2 + 3N + 5 = \Omega(\log N)$ 도 성립한다. 그러나 $g(N)$ 을 선택할 때에는 정의를 만족하는 가장 높은 차수의 함수를 선택하는 것이 바람직함
- $f(N) \geq cg(N)$ 을 만족하는 가장 작은 양의 c 값을 찾아야 하는 것은 아니다. 왜냐하면 $f(N) \geq cg(N)$ 을 만족하는 양의 상수 c 와 N_0 가 존재하기만 하면 되기 때문

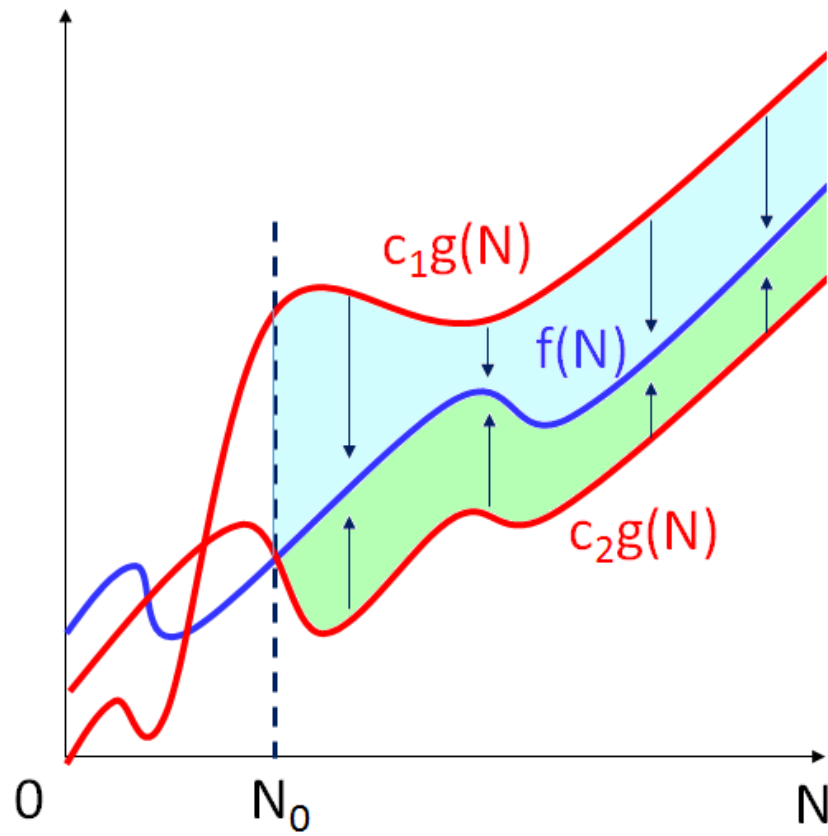


$$f(N) = \Omega(g(N))$$

Θ-표기법

[Θ-표기의 정의]

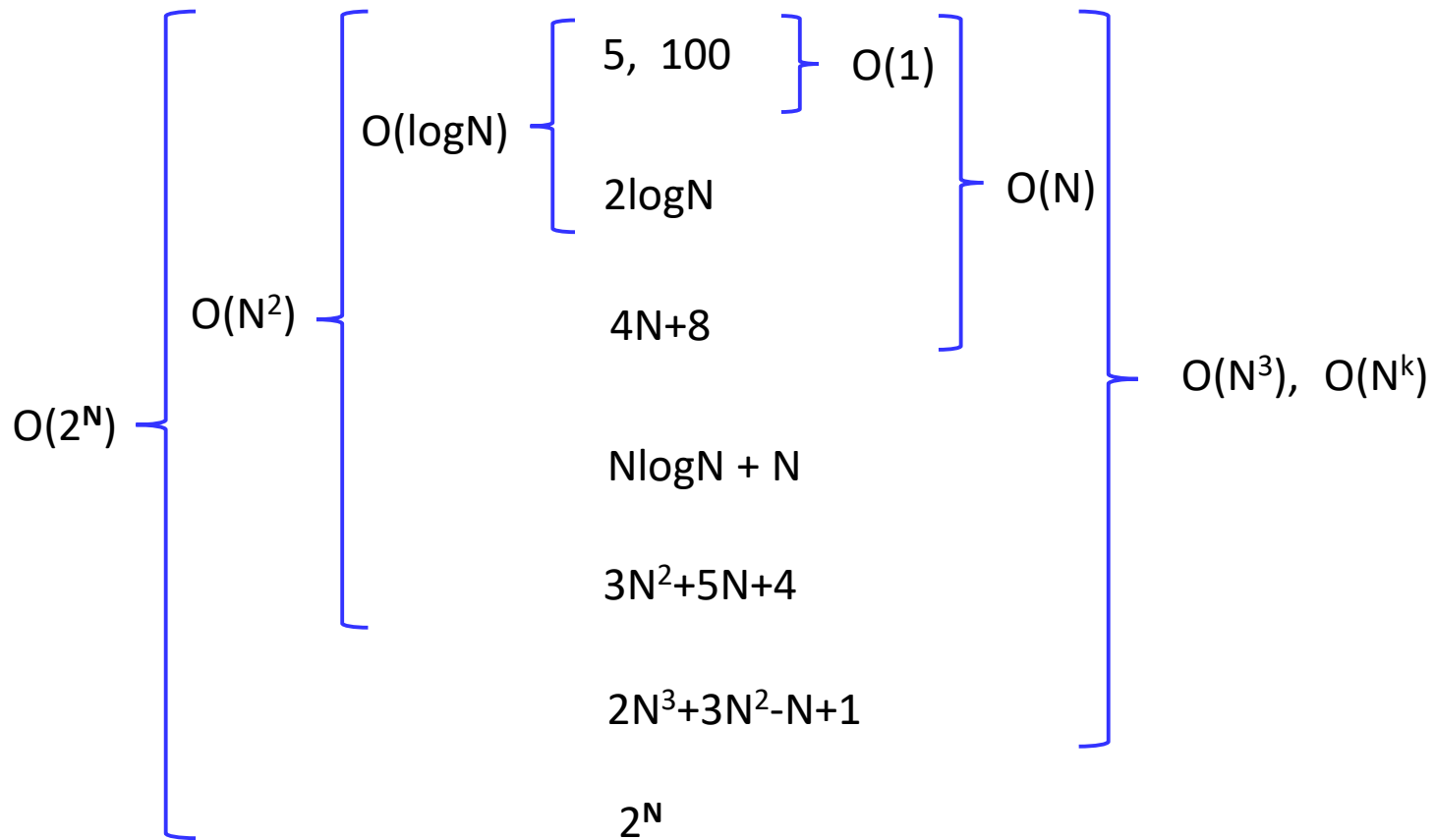
- 모든 $N \geq N_0$ 에 대해서 $c_1g(N) \geq f(N) \geq c_2g(N)$ 이 성립하는 양의 상수 c_1, c_2, N_0 가 존재하면, $f(N) = \Theta(g(N))$ 이다.
- Θ-표기는 수행시간의 o-표기와 Ω-표기가 동일한 경우에 사용
- $2N^2 + 3N + 5 = O(N^2)$ 과 동시에 $2N^2 + 3N + 5 = \Omega(N^2)$ 이므로, $2N^2 + 3N + 5 = \Theta(N^2)$
- $\Theta(N^2)$ 은 N^2 과 $(2N^2 + 3N + 5)$ 이 유사한 증가율을 가지고 있다는 뜻
- $2N^2 + 3N + 5 \neq \Theta(N^3), 2N^2 + 3N + 5 \neq \Theta(N)$



$$f(N) = \Theta(g(N))$$

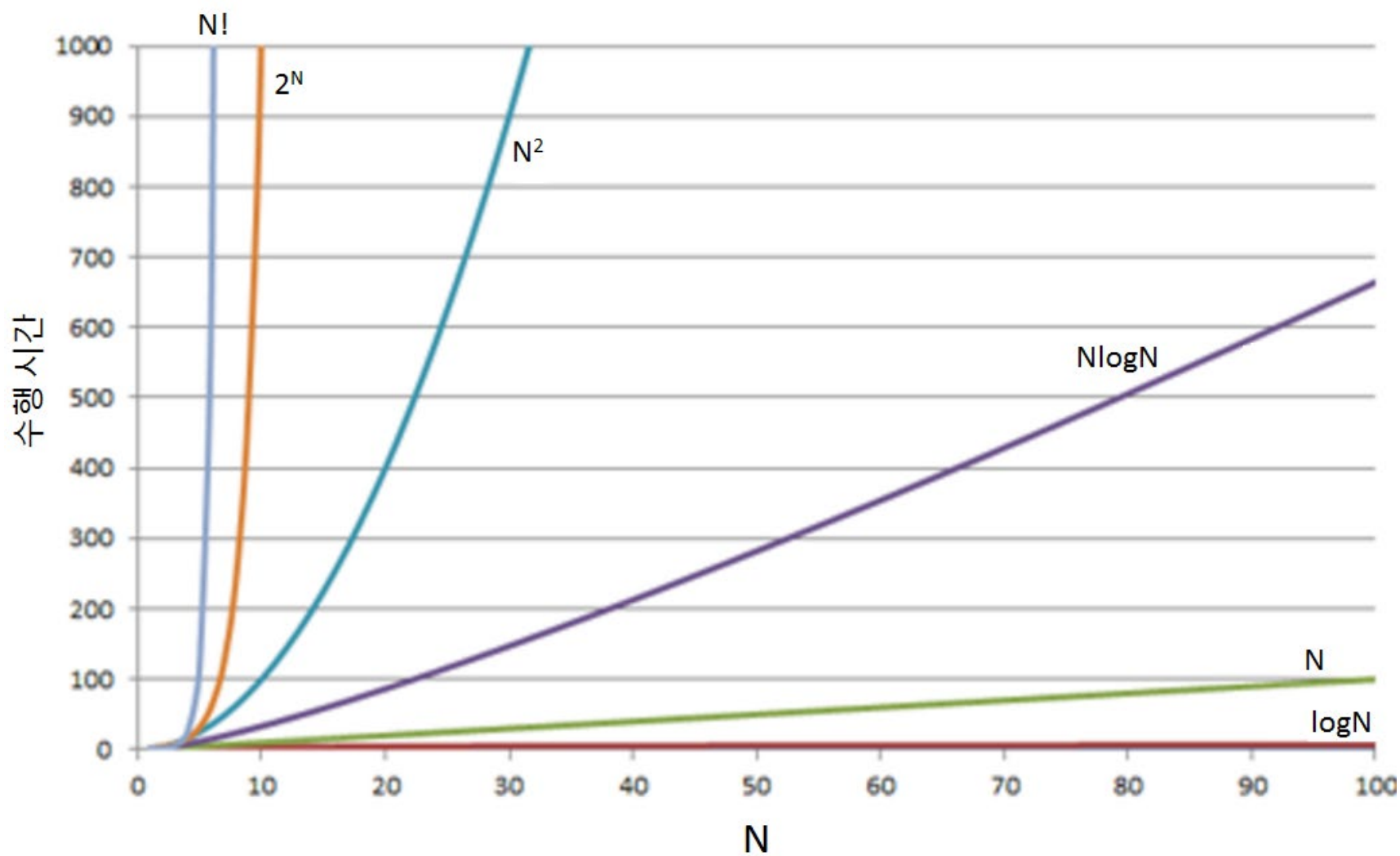
자주 사용되는 함수의 O -표기와 이름

- 알고리즘의 수행시간은 주로 O -표기를 사용하며, 보다 정확히 표현하기 위해 θ -표기를 사용하기도 한다.
- $O(1)$ 상수시간(Constant Time)
- $O(\log N)$ 로그(대수)시간(Logarithmic Time)
- $O(N)$ 선형시간(Linear Time)
- $O(N \log N)$ 로그선형시간(Log-linear Time)
- $O(N^2)$ 제곱시간(Quadratic Time)
- $O(N^3)$ 세제곱시간(Cubic Time)
- $O(2^N)$ 지수시간(Exponential Time)



O-표기의 포함 관계

함수의 증가율 비교



1-4 파이썬 언어에 대한 기본적인 지식

```
class 클래스이름:
    def __init__(self, 인자1, ...): # 객체 생성자
        self.인스턴스 변수1 = 인자1
        :
    def
    :
    def
```

— 객체에 대한 메소드

- 인스턴스 변수: 객체에 정보를 저장
- 객체 생성자(Constructor): 객체를 생성
- 메소드: 객체 혹은 객체 내부 인스턴스 변수에 대한 연산 수행

[예제] 학생을 객체로 표현하기 위해 Student 클래스

- 각 Student 객체는 이름과 학번을 각각 저장하는 인스턴스 변수 name과 id를 가진다고 가정

```
01 class Student:
02     def __init__(self, name, id):
03         self.name = name
04         self.id = id
05     def get_name(self):
06         return self.name
07     def get_id(self):
08         return self.id
09
10 best = Student('Lee', 101)
11 print(best.get_name())
12 print(best.get_id())
```

Student 객체
생성자


인스턴스 변수

객체의 name을
리턴하는 메소드

객체의 id를
리턴하는 메소드

best 학생의
name과 id 출력

[프로그램 1-1]

Console  PyUnit

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

Lee


101

[프로그램 1-1]의 수행 결과

- 리스트(List)는 C나 자바 언어의 배열과 유사
- 배열은 동일한 타입의 항목들을 저장하는 반면에
리스트는 서로 다른 타입의 항목들을 저장할 수 있음

```
01 a = []                # 비어있는 리스트 a 선언
02 b = [None] * 10       # 크기가 10이고 각 원소가 None으로 초기화 된 리스트
03 c = [40, 10, 70, 60]  # 크기가 4이고 4개의 정수로 초기화된 리스트
04 print(c[0])           # 40 출력
05 print(c[-1])          # 60 출력
06 c.pop()               # 리스트 마지막 항목인 60 제거
07 c.pop(0)              # 리스트 첫 항목인 40 제거
08 c.append(90)           # 리스트 맨 뒤에 90 추가
09 print(len(c))         # 내장 함수인 len()은 리스트의 크기 리턴
```

[프로그램 1-2]

Console  PyUnit

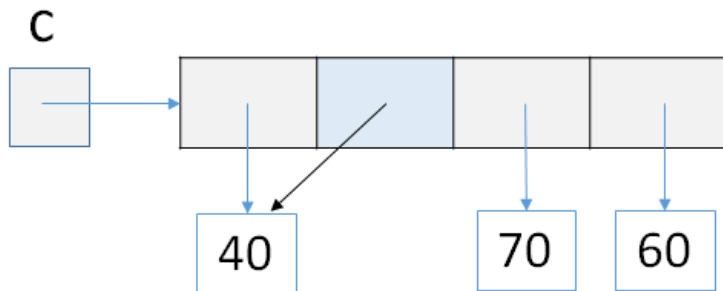
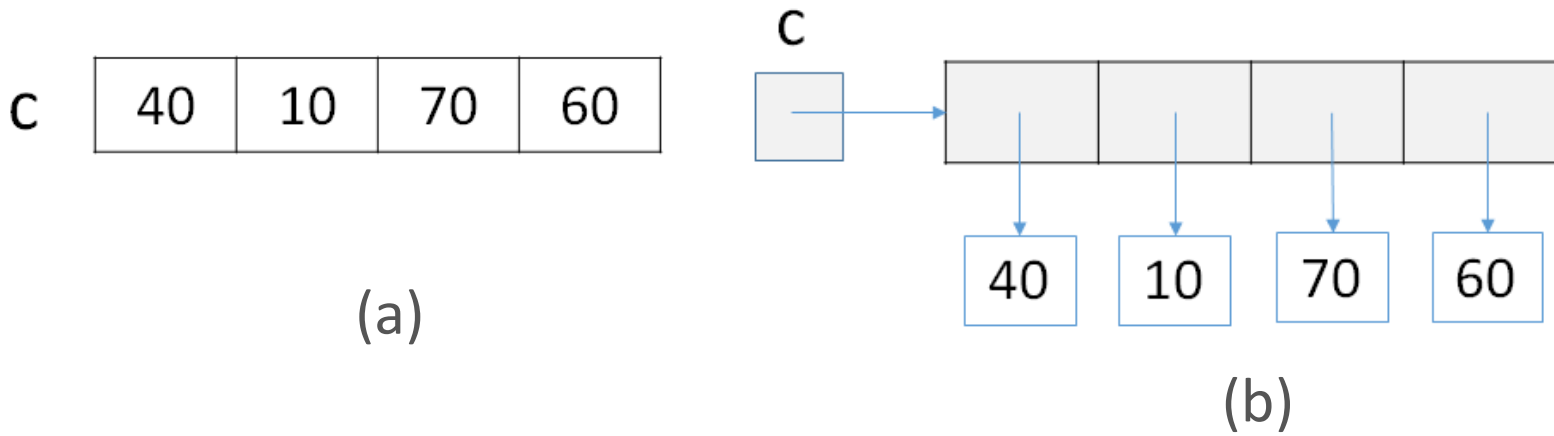
<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\

40

60

3

[프로그램 1-2] line 03에서 $c = [40, 10, 70, 60]$ 이 수행되면, 4개의 정수들이 (a)와 같이 저장되는 것이 아니라 실제로 (b)와 같이 메모리에 저장된다.



$c[1] = 40$ 이 수행된 후의
리스트 c

- `if`-문은 조건문으로서 하나 또는 여러가지 조건을 검사하도록 선언
- 조건식이 `True`이면 해당 명령문을 수행하고, `False`이면 `else`의 명령문을 수행
- 여러 조건으로 세분화시켜야 하는 경우에는 `if – elif – ... – else` 문을 사용

```
if 조건식:  
    명령문  
else:  
    명령문
```

```
if 조건식:  
    명령문  
elif 조건식:  
    명령문  
...  
else:  
    명령문
```

- 반복문에는 `for`-문과 `while`-문이 있다.
- `for`-문은 리스트, 튜플, 스트링 등의 순서형 자료의 항목들을 차례로 읽으며 명령문을 처리하는 방식과 내장 함수인 `range()`를 사용하는 방식
- `while`-문은 조건식이 만족된 동안에만 반복 수행

`for` 변수 `in` 리스트:

명령문

`while` 조건식:

명령문

`for` 변수 `in range(N)`:

명령문

- `range()` 함수는 다음과 같은 형식을 갖는다.

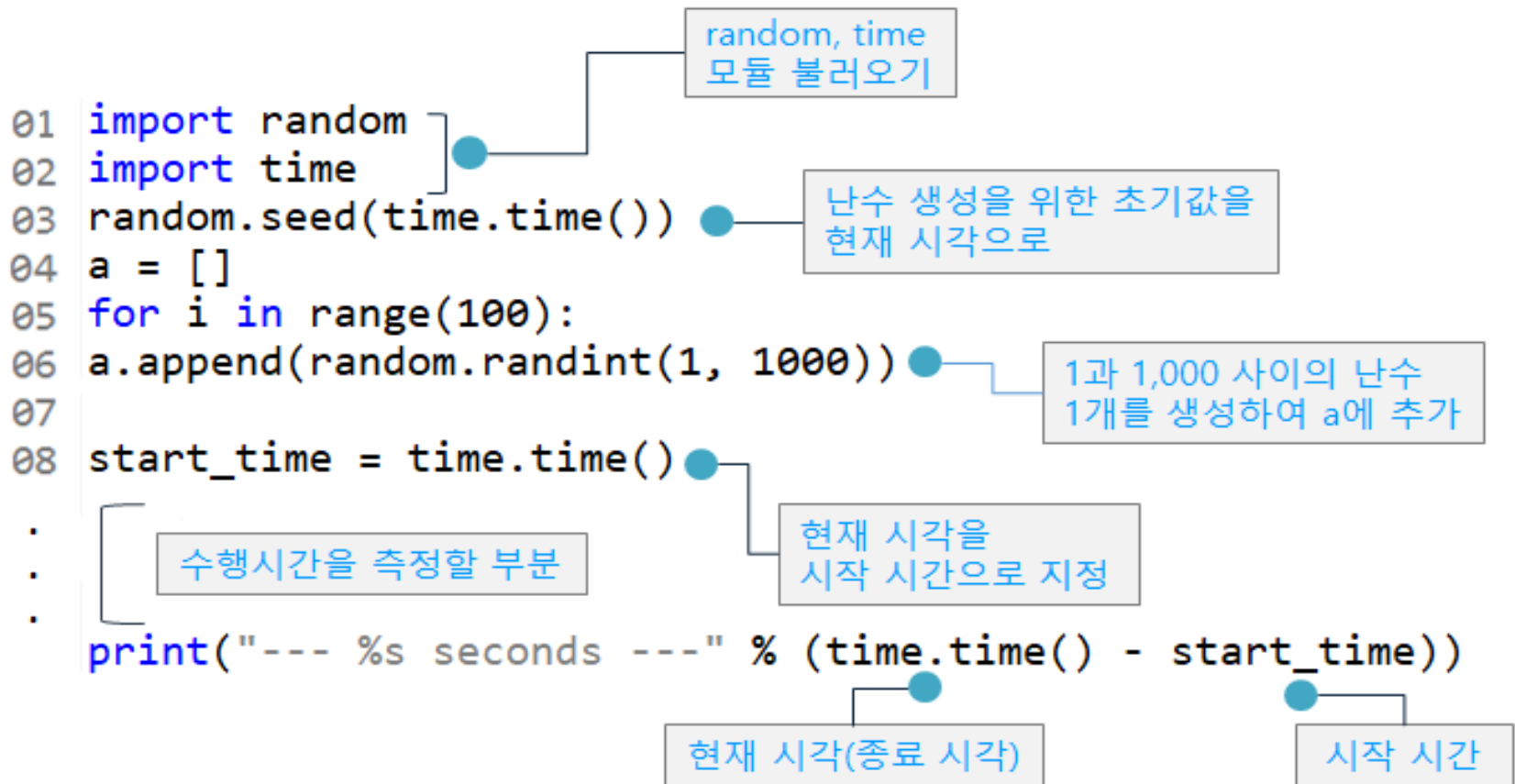
`range(N)`

`range(시작, N)`

`range(시작, N, 증감)`

01	<code>range(10)</code>	# 0, 1, 2, ..., 9 리턴
02	<code>range(1, 11)</code>	# 1, 2, ..., 10 리턴
03	<code>range(10, 20, 2)</code>	# 10, 12, 14, 16, 18 리턴
04	<code>range(10, 1, -1)</code>	# 10, 9, 8, ..., 2 리턴
05	<code>range(len(리스트))</code>	# 0부터 (리스트의 크기-1)까지 리턴

- 의사 난수(Pseudo-random Number) 생성을 이용해 리스트 만들기과 프로그램의 실행시간을 측정하기



`time.time()`은 1970년 1 월 1일 0시 0분 0초로부터 현재 시각을 몇 초인 지를 리턴

내장 함수

- `ord('문자')`

문자의 Unicode 값을 리턴: 예, `ord('A')`는 65를 리턴

- `list(reversed(리스트))`

역순으로 된 리스트를 리턴

- `리스트.reverse()`

리스트를 역순으로

예, `a.reverse()`는 리스트 `a`를 역순으로

- **lambda 함수**
함수의 이름도 return도 없이 수행되는 함수

lambda 인자(arguments): 식(expression)

lambda 함수는 filter() 또는 map() 함수의 인자로 사용

- **filter(expression, sequence)**
- **map(expression, sequence)**
- lambda 함수는 일반적으로 간단한 함수를 대신할 수 있어 매우 유용

```

01 a = [1, 5, 4, 6, 8, 11, 3, 12]
02 even = list(filter(lambda x: (x%2 == 0) , a))
03 print(even)
04 ten_times = list(map(lambda x: x * 10, a))
05 print(ten_times)
06
07 b = [[0, 1, 8], [7, 2, 2], [5, 3, 10], [1, 4, 5]]
08 b.sort(key = lambda x: x[2])
09 print(b)

```

a에서 짝수만 선택하여
리스트로 리턴

a의 각 숫자를 10 배로
만들어 리스트로 리턴

b의 각 원소(리스트)의 마지막
숫자를 기준으로 정렬

[프로그램 1-4]

Console PyUnit

```

<terminated> main.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32
[4, 6, 8, 12]
[10, 50, 40, 60, 80, 110, 30, 120]
[[7, 2, 2], [1, 4, 5], [0, 1, 8], [5, 3, 10]]

```

1-5 순환(Recursion)

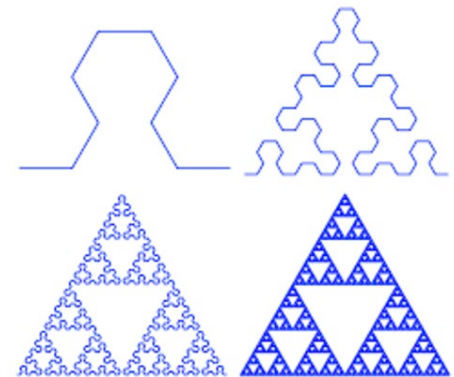
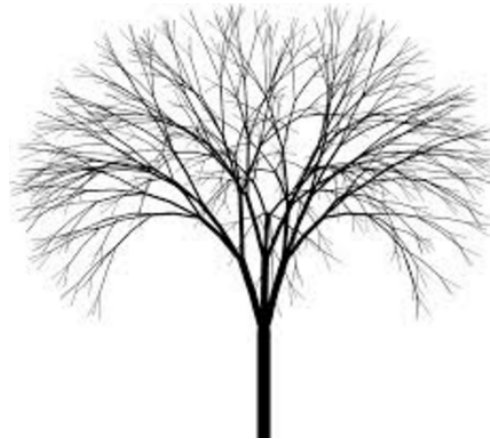
- 순환(Recursion): 함수/메소드의 실행 과정 중 스스로를 호출하는 것
- 순환은 팩토리얼, 조합을 계산하기 위한 식의 표현, 무한한 길이의 숫자 스트림을 만들기, 분기하여 자라나는 트리 자료구조, 프랙털(Fractal) 등의 기본 개념으로 사용

$$n! = n \cdot (n - 1)!$$

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$F_n = F_{n-1} + F_{n-2}, \quad F_0=0, \quad F_1=1$$

$$X_n = (aX_{n-1} + c) \% m, \quad X_0=seed$$



- 함수가 자기 자신을 호출할 때 무한 호출을 방지해야 함
- 다음의 프로그램을 실행시키면 `RecursionError` 발생
- 이는 스스로의 호출을 중단시킬 수 있는 조건문이 없기 때문

```
01 def recurse():  
02     print(' * ')  
03     recurse()  
04  
05 recurse()
```

Console x PyUnit

<terminated> main.py [C:\Users\sbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

recurse()

File "C:\Users\sbyang\Desktop\pyworkspace\test\main.py", line 3, in recurse

recurse()

File "C:\Users\sbyang\Desktop\pyworkspace\test\main.py", line 3, in recurse

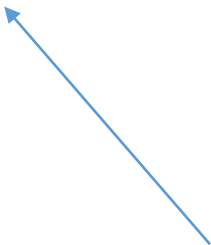
recurse()

[Previous line repeated 993 more times]

File "C:\Users\sbyang\Desktop\pyworkspace\test\main.py", line 2, in recurse

print(' *')

RecursionError maximum recursion depth exceeded while calling a Python object



```
01 def recurse(count):
02     if count <= 0:
03         print('.')
04     else:
05         print(count, ' *')
06         recurse(count-1)
07
08 recurse(5)
```

Console

PyUnit

<terminated> main.py [C:\WUsers

5 *

4 *

3 *

2 *

1 *

.

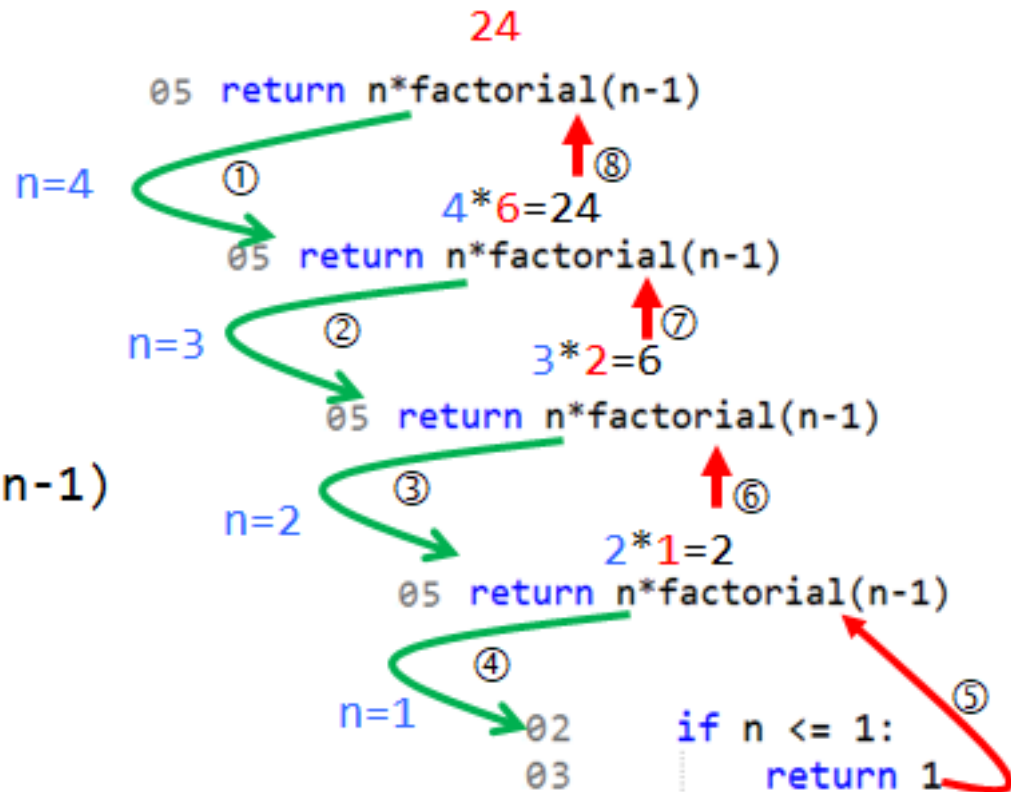
순환으로 구현된 함수는 두 부분으로 구성:

- 기본(Base) case: 스스로를 더 이상 호출하지 않는 부분
- 순환 case: 스스로를 호출하는 부분
- 무한 호출을 방지하기 위해 선언한 변수 또는 수식의 값이 호출이 일어날 때마다 순환 case에서 감소되어 최종적으로 if-문의 조건식에서 기본 case를 실행하도록 제어해야 함

팩토리얼 계산

- Line 07에서 factorial(4)로 함수 호출
- [그림]의 번호 순서대로 수행되어 24를 출력

```
01 def factorial(n):  
02     if n <= 1:  
03         return 1  
04     else:  
05         return n*factorial(n-1)  
06  
07 print(factorial(4))
```



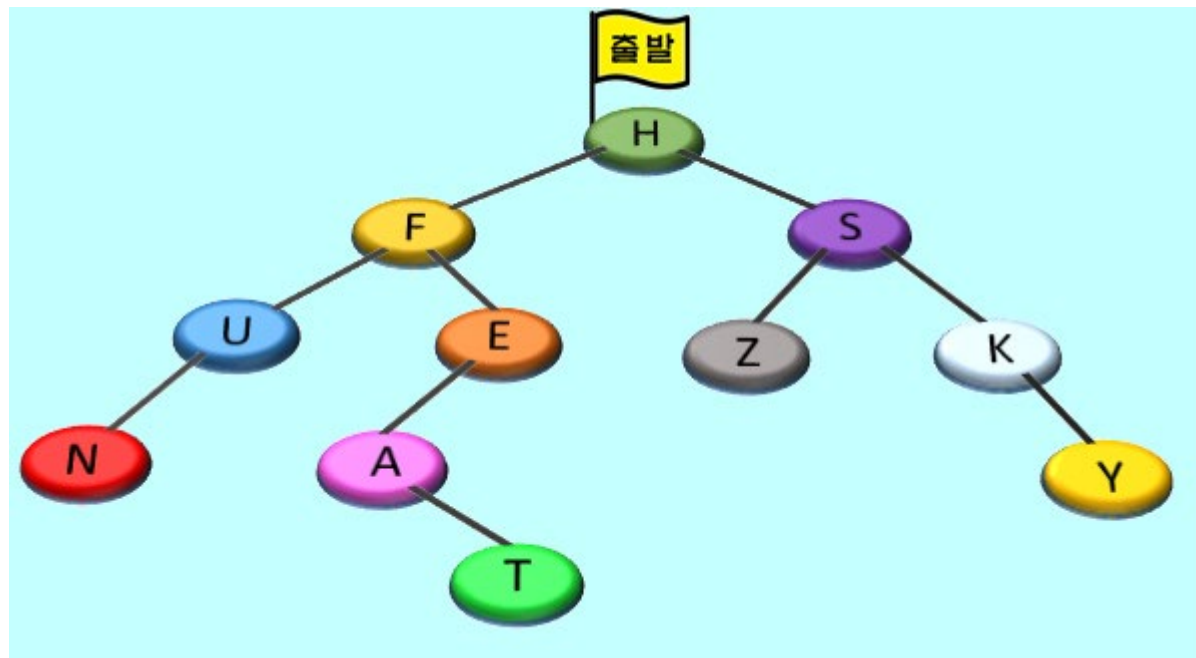
반복문으로 팩토리얼을 계산하는 프로그램

- 반복문을 이용한 계산은 함수 호출로 인해 시스템 스택을 사용 하지 않으므로 순환을 이용한 계산보다 매우 간단하며 메모리도 적게 사용

```
01 factorial = 1
02 for i in range(1, 5):
03     factorial *= i
04
05 print(factorial)
```

- 꼬리 순환(Tail Recursion): 함수의 마지막 부분에서 순환하는 것
- 꼬리 순환은 반복문으로 변환하는 것이 수행 속도와 메모리 사용 측면에서 효율적임

[예제] 남태평양에 있는 어느 나라에 11개의 섬이 그림과 같이 다리로 연결되어 있다. 이 나라의 관광청에서는 관광객들이 모든 11개의 섬들의 방문 순서가 다른 3개의 관광코스를 개설



- 각 코스의 관광은 섬 H에서 시작하며, 관광객에서는 각 관광 코스의 방문 순서를 다음과 같은 규칙 하에 만들었다.

A-코스: 섬에 도착하면 항상 도착한 섬을 먼저 관광하고, 그 다음엔 왼쪽 섬으로 관광을 진행하고 왼쪽 방향의 모든 섬들을 방문한 후에 오른쪽 섬으로 관광을 진행

```
def A_course(n):    # A-코스
    if n != None:
        print(n.name, '-> ', end='') # 섬 n 방문
        A_course(n.left)              # n의 왼쪽으로 진행
        A_course(n.right)             # n의 오른쪽으로 진행
```

B-코스: 섬에 도착하면 도착한 섬의 관광을 미루고, 먼저 왼쪽 섬으로 관광을 진행하고 왼쪽 방향의 모든 섬들을 방문한 후에 돌아와서 섬을 관광한다. 그 다음엔 오른쪽 섬으로 관광을 진행

- 섬 H는 왼쪽 방향의 섬 F, U, N, E, A, T를 모두 관광한 다음에 관광

```
def B_course(n): # B-코스
    if n != None:
        B_course(n.left)      # n의 왼쪽으로 진행
        print(n.name, '-> ', end='') # 섬 n 방문
        B_course(n.right)     # n의 오른쪽으로 진행
```

C-코스: 섬에 도착하면 도착한 섬의 관광을 미루고, 먼저 왼쪽 섬으로 관광을 진행하고 왼쪽 방향의 모든 섬들을 관광한 후에 돌아와서, 오른쪽 섬으로 관광을 진행하고 오른쪽 방향의 모든 섬들을 관광한 후에 돌아와서, 드디어 섬을 관광

- 섬 H는 왼쪽 방향의 섬 F, U, N, E, A, T를 모두 관광하고 오른쪽 방향의 모든 섬 S, Z, K, Y를 관광한 다음에 마지막으로 관광

```
def C_course(n): # C-코스
    if n != None:
        C_course(n.left)      # n의 왼쪽으로 진행
        C_course(n.right)     # n의 오른쪽으로 진행
        print(n.name, '-> ', end='') # 섬 n 방문
```



```

01 class Node:
02     def __init__(self, name, left=None, right=None): # 섬 생성자
03         self.name = name
04         self.left = left
05         self.right = right
06
07 def map(): # 지도 만들기
08     n1 = Node('H')
09     n2 = Node('F')
10     n3 = Node('S')
11     n4 = Node('U')
12     n5 = Node('E')
13     n6 = Node('Z')
14     n7 = Node('K')
15     n8 = Node('N')
16     n9 = Node('A')
17     n10 = Node('Y')
18     n11 = Node('T')
19
20     n1.left = n2
21     n1.right = n3
22     n2.left = n4
23     n2.right = n5
24     n3.left = n6
25     n3.right = n7
26     n4.left = n8
27     n5.left = n9
28     n7.right = n10
29     n9.right = n11
30     return n1 # 시작 섬 리턴

```

11개의
섬 만들기


11개의
섬
교량으로
잇기

```

31 def A_course(n): # A-코스
32     if n != None:
33         print(n.name, '-> ', end='') # 섬 n 방문
34         A_course(n.left)             # n의 왼쪽으로 진행
35         A_course(n.right)            # n의 오른쪽으로 진행
36
37 def B_course(n): # B-코스
38     if n != None:
39         B_course(n.left)             # n의 왼쪽으로 진행
40         print(n.name, '-> ', end='') # 섬 n 방문
41         B_course(n.right)            # n의 오른쪽으로 진행
42
43 def C_course(n): # C-코스
44     if n != None:
45         C_course(n.left)             # n의 왼쪽으로 진행
46         C_course(n.right)            # n의 오른쪽으로 진행
47         print(n.name, '-> ', end='') # 섬 n 방문
48
49 start = map()                        # 시작 섬을 n1으로
50 print('A-코스:\t', end='')
51 A_course(start)
52 print('\nB-코스:\t', end='')
53 B_course(start)
54 print('\nC-코스:\t', end='')
55 C_course(start)

```

수행 결과

Console  PyUnit

<terminated> travel.py [C:\Users\wsbyang\AppData\Local\Programs\Python\Python36-32\python.exe]

A-코스: H -> F -> U -> N -> E -> A -> T -> S -> Z -> K -> Y ->

B-코스: N -> U -> F -> A -> T -> E -> H -> Z -> S -> K -> Y ->

C-코스: N -> U -> T -> A -> E -> F -> Z -> Y -> K -> S -> H ->

- 4장의 이진트리에서 트리를 순회하는 방법을 공부하는데
관광코스가 이진트리의 3가지 순회 방법
 - A-코스 = 전위순회
 - B-코스 = 중위순회
 - C-코스 = 후위순회

- 일반적으로 순환은 프로그램(알고리즘)의 **가독성**을 높일 수 있는 장점을 갖지만 시스템 스택을 사용하기 때문에 **메모리 사용 측면에서 비효율적**임
- 반복문으로 변환하기 어려운 순환도 존재하며, 억지로 반복문으로 변환하는 경우 프로그래머가 시스템 스택의 수행을 처리해야 하므로 프로그램이 매우 복잡해질 수 밖에 없음
- 반복문으로 변환된 프로그램의 수행 속도가 순환으로 구현된 프로그램보다 항상 빠르다는 보장 없음



요약

- **자료구조**: 일련의 동일한 타입의 데이터를 정돈하여 저장한 구성체
- **추상데이터타입**은 데이터와 그 데이터에 관련된 추상적인 연산들로서 구성된다. 추상적이란 연산을 구체적으로 어떻게 구현하여야 한다는 상세를 포함하고 있지 않다는 뜻이다. 자료구조는 추상데이터타입을 구체적(실제 프로그램)으로 구현한 것
- **수행시간**은 알고리즘이 수행하는 기본적인 연산 횟수를 입력 크기에 대한 함수로 표현

- 알고리즘의 수행시간 분석 방법: 최악경우, 평균경우, 최선경우 분석
- 점근표기법: 입력 크기가 증가함에 따른 수행시간의 간단한 표기법
- O (Big-Oh)-표기: 점근적 상한
- Ω (Big-Omega)-표기: 점근적 하한
- Θ (Theta)-표기: 동일한 증가율

- 파이썬 언어는 객체지향 프로그래밍 언어로서 클래스를 선언하여 데이터를 객체에 저장하고 메소드를 선언하여 객체들에 대한 연산 구현

- **순환(Recursion)**: 함수가 스스로를 호출하는 것
- 순환으로 구현된 함수는 두 부분으로 구성
 - **기본(Base) case**: 스스로를 더 이상 호출하지 않는 부분
 - **순환 case**: 스스로를 호출하는 부분
- 무한 호출을 방지하기 위해 선언한 변수 또는 수식의 값이 호출이 일어날 때마다 순환 case에서 감소되어 최종적으로 if-문의 조건식에서 기본 case를 실행하도록 제어해야 함
- **꼬리 순환(Tail Recursion)**: 함수의 마지막 부분에서 순환하는 것. 꼬리 순환은 반복문으로 변환하는 것이 수행
- 순환은 프로그램(알고리즘)의 **가독성**을 높일 수 있다는 장점을 갖지만, 시스템 스택을 사용하기 때문에 메모리 사용 측면에서는 비효율적임