01 CHAPTER

자료구조와 알고리즘

1장. 학습 목표

- 자료구조와 알고리즘의 개념과 관계를 이해한다.
- 추상 자료형의 개념을 이해하고 Bag 자료형에 적용해 본다.
- 알고리즘의 실행 시간 측정 방법을 이해하고 활용할 수 있다.
- 알고리즘의 시간 복잡도 개념과 빅오 표기법 등을 이해한다.
- 순환의 개념과 구조를 이해하고, 다양한 순환 문제를 살펴본다.
- 순환을 통해 알고리즘의 시간 복잡도 분석 능력을 기른다.
- 2장에서 학습할 파이썬 코드의 형식에 미리 익숙해진다.

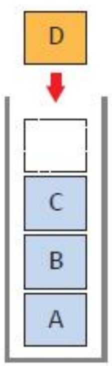
1.1 자료구조와 알고리즘

- 자료구조란?
- 알고리즘이란?
- 알고리즘의 조건
- 알고리즘의 기술 방법

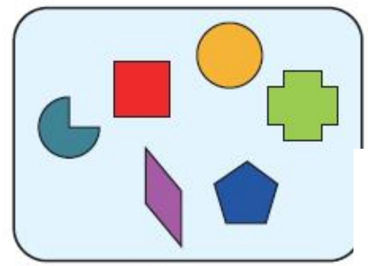
자료구조란?

- 일상 생활에서 자료를 정리하고 조직화하는 이유는?
  - 사물을 편리하고 효율적으로 사용하기 위함
  - 다양한 자료를 효율적인 규칙에 따라 정리한 예

스택

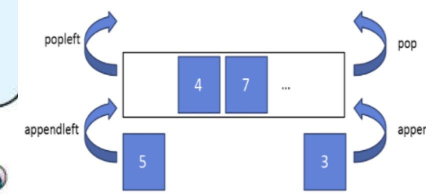


큐



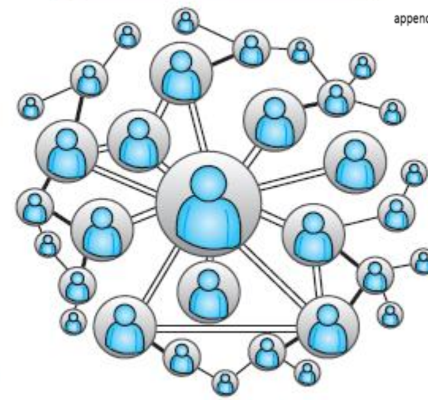
정렬, 탐색

덱 (Deque)



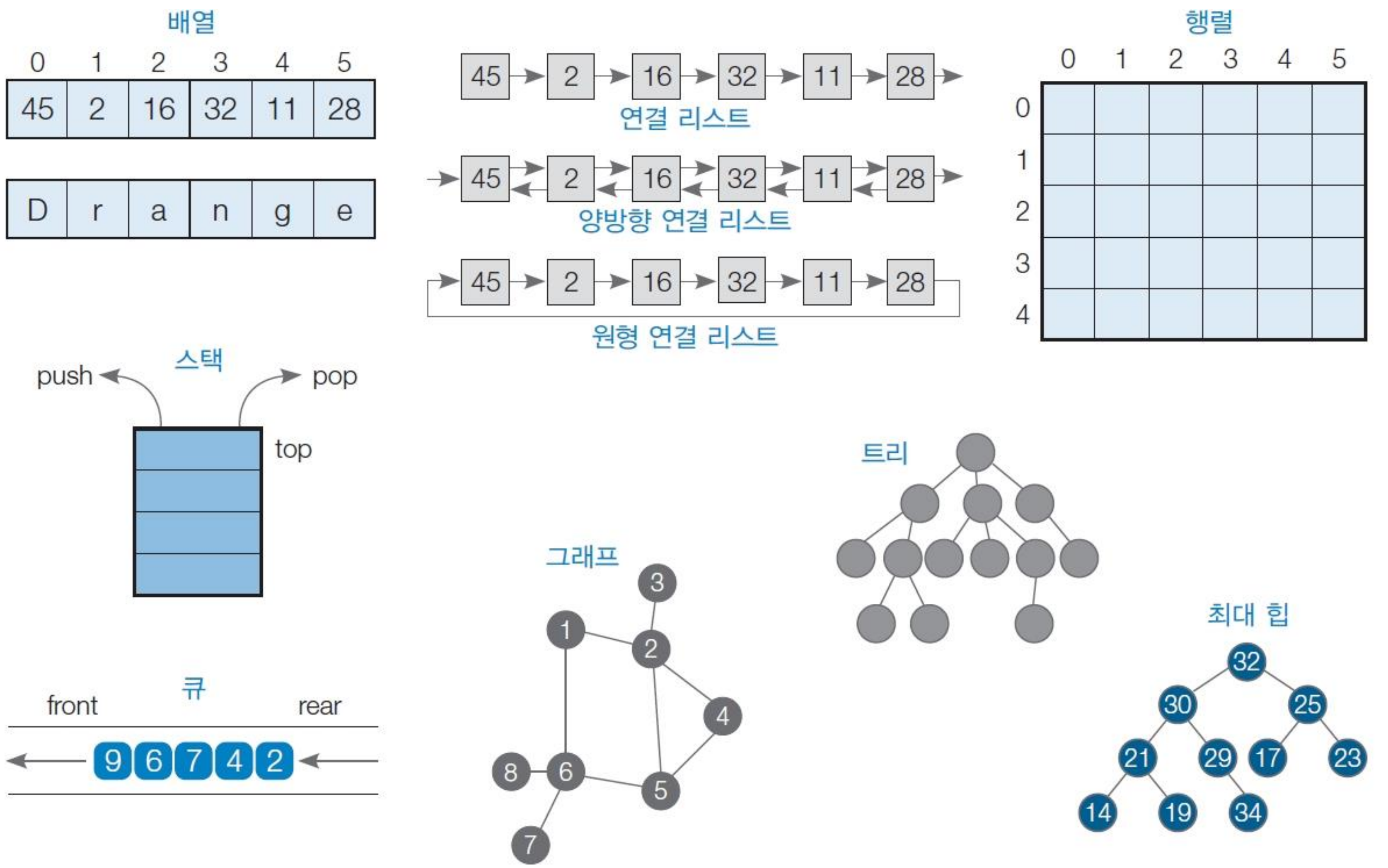
리스트

트리구조



그래프

자료구조 종류

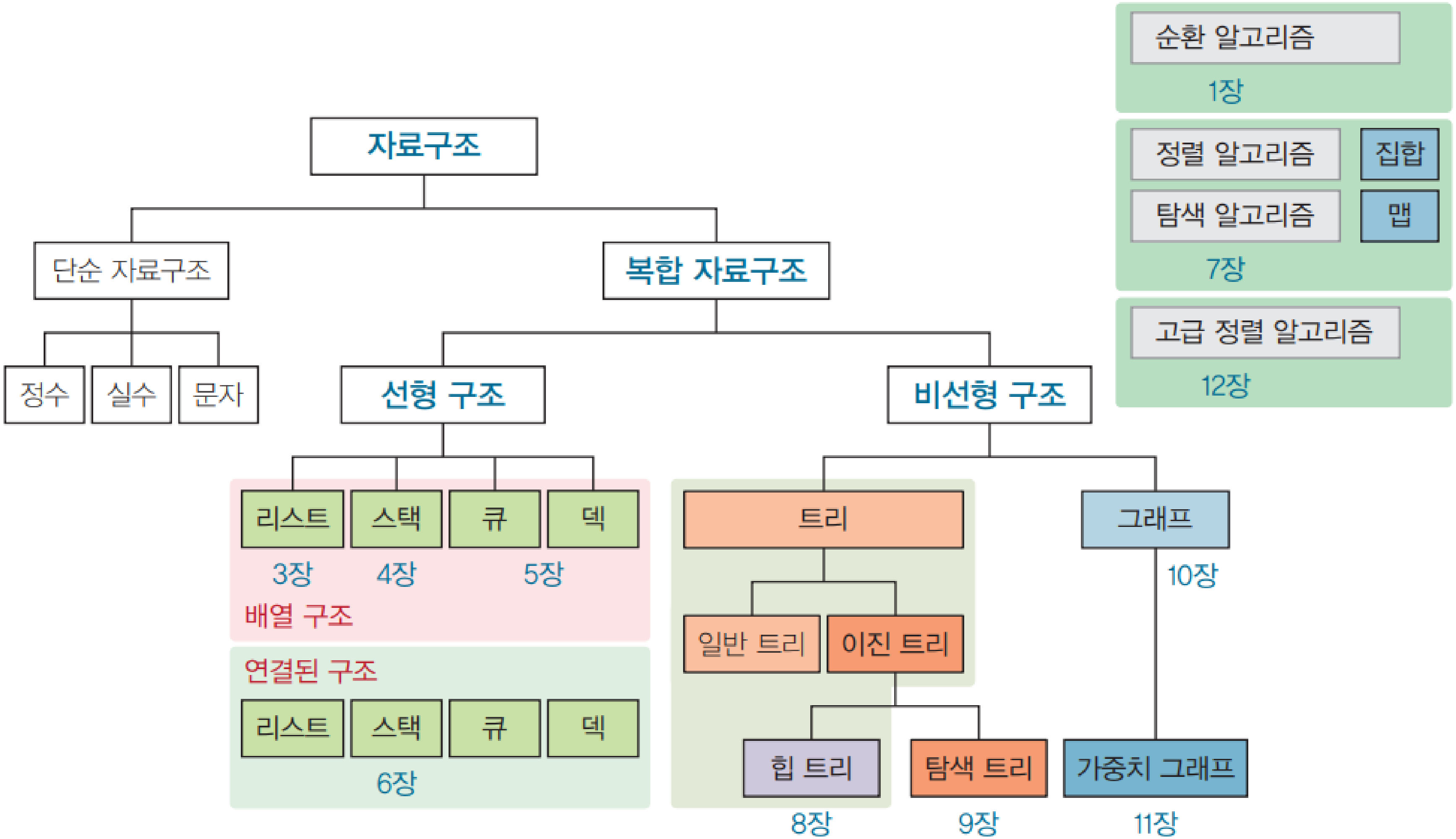


배열
0 1 2 3 4 5
45 2 16 32 11 28
D r a n g e
45 2 16 32 11 28
연결 리스트
45 2 16 32 11 28
양방향 연결 리스트
45 2 16 32 11 28
원형 연결 리스트
행렬
0 1 2 3 4 5
0 1 2 3 4
스택
push
pop
top
큐
front
rear
9 6 7 4 2
그래프
1 2 3 4 5 6 7 8
트리
최대 힙
32 30 25 21 29 17 23 14 19 34

컴퓨터에서의 자료구조

- 자료구조(Data Structure)
  – 컴퓨터에서 자료를 정리하고 조직화하는 다양한 구조
- 선형 자료구조
  – 항목들을 순서적으로 나열하여 저장하는 창고
  – 항목 접근 방법에 따라 다시 세분화
    - 리스트: 가장 자유로운 선형 자료구조
    - 스택, 큐, 덱: 항목의 접근이 맨 앞(전단)이 나 맨 뒤(후단)로 제한
- 비선형 자료구조
  – 항목들이 보다 복잡한 연결 관계
  – 트리: 회사의 조직도나 컴퓨터의 폴더와 같은 계층 구조
    - 힙 트리는 우선순위 큐
    - 이진 탐색트리나 AVL트리: 탐색을 위한 트리 구조
  – 그래프: 가장 복잡한 연결 관계를 표현
    - 다양한 문제를 해결하기 위한 기본 구조로 사용된다.

이 책의 구성



순환 알고리즘
1장
정렬 알고리즘
집합
탐색 알고리즘
맵
7장
고급 정렬 알고리즘
12장
자료구조
단순 자료구조
복합 자료구조
정수
실수
문자
선형 구조
비선형 구조
리스트
스택
큐
덱
3장
4장
5장
배열 구조
연결된 구조
리스트
스택
큐
덱
6장
트리
일반 트리
이진 트리
힙 트리
탐색 트리
8장
9장
그래프
10장
가중치 그래프
11장

자료구조는 생각하는 방법을 훈련하는 도구

자료구조는 그 자체로 중요하다

못지 않게, 생각하는 방법 훈련도 중요하다

- 자료구조를 이용해서 문제를 해결하는 과정
- 문제 해결 과정에서 논리의 골격이 구성되는 방법/스타일
- 의미의 단위(의미의 매듭)를 설정하는 방법

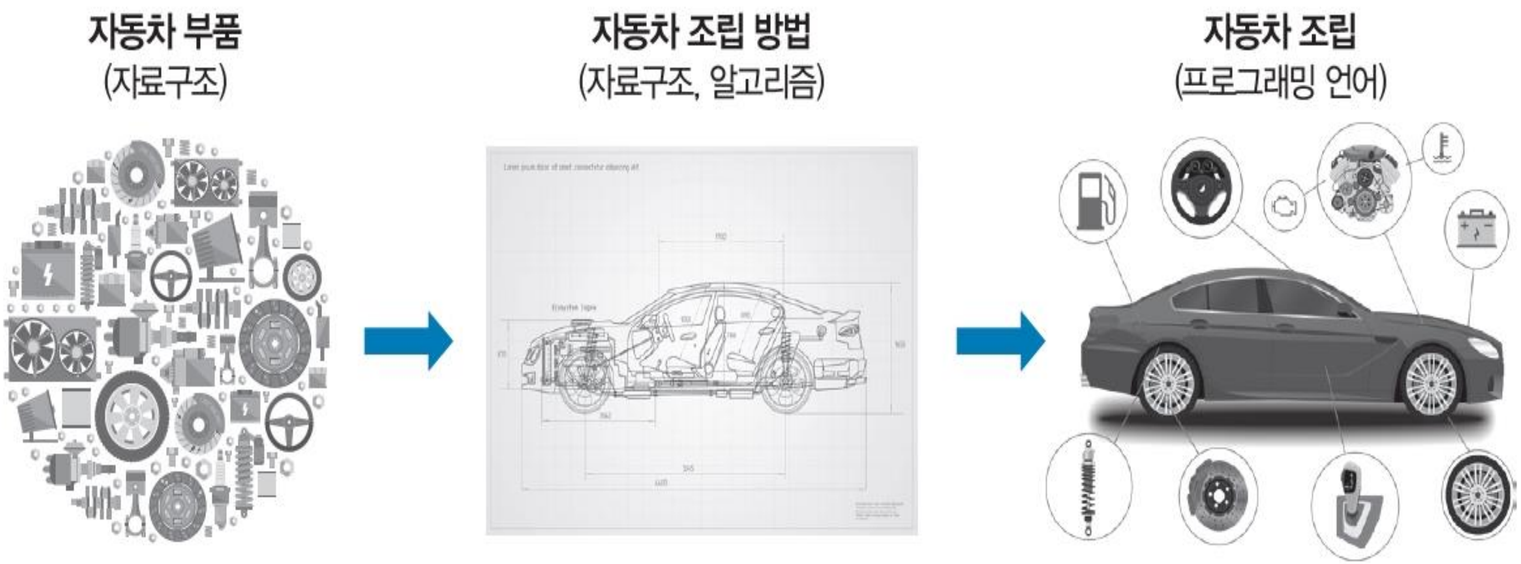
자료구조는 건축의 부품이나 모듈 같은 것

건축물을 만들려면
- 건축 재료와 구조 모듈에 대한 이해가 필요하다
- 철근, 시멘트, 강화 유리, 벽돌, ...
- 샤시, 철골, 거푸집, 배수 구조, 전기/인터넷 연결 구조, ...

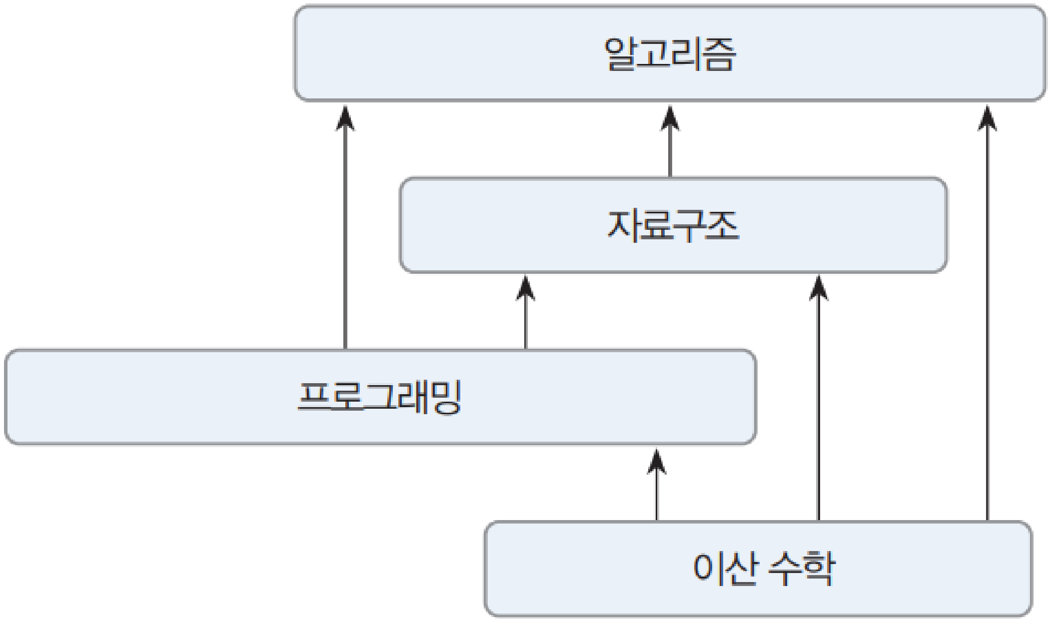
프로그래밍과 문제 해결도
- 데이터와 구조 모듈에 대한 이해가 필요하다
- 프로그래밍 언어, 정수, 문자열, ...
- 리스트, 스택, 큐, 우선순위 큐, 검색 트리, 해시 테이블, 그래프, ...

자료구조와 알고리즘의 관계

- 자료구조는 알고리즘 과목의 직전 단계이기도 하고, 그 자체로 여러 알고리즘을 포함



프로그래밍, 자료구조, 알고리즘의 관계



알고리즘이란?

- 컴퓨터로 문제를 풀기 위한 단계적인 절차
  - 예) 사전에서 단어 찾기

- 프로그램 = 자료구조 + 알고리즘

알고리즘의 조건

- 입 력 : 0개 이상의 입력이 존재하여야 한다.
- 출 력 : 1개 이상의 출력이 존재하여야 한다.
- 명백성 : 각 명령어의 의미는 모호하지 않고 명확해야 한다.
- 유한성 : 한정된 수의 단계 후에는 반드시 종료되어야 한다.
- 유효성 : 각 명령어들은 실행 가능한 연산이어야 한다.

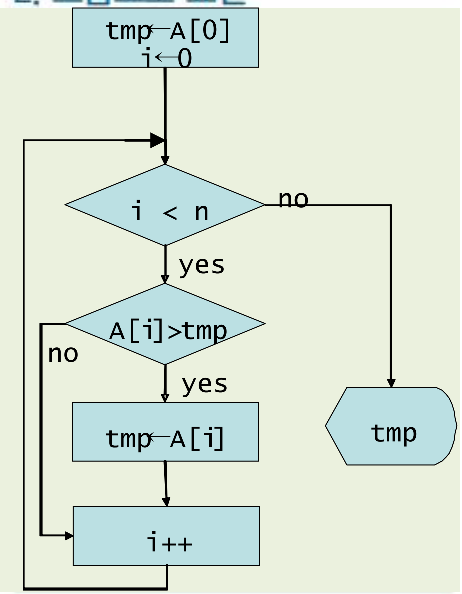
알고리즘의 기술 방법

1. 자연어로 표현

find_max(A)

1. 배열 A의 첫 번째 요소를 변수 tmp에 복사한다.
2. 배열 A의 다음 요소들을 차례대로 tmp와 비교하여, 더 크면 그 값을 tmp로 복사한다.
3. 배열 A의 모든 요소를 비교했으면 tmp를 반환한다.

2. 흐름도로 표현



3. 유사 코드로 표현

```
find_max(A)
tmp ← A[0];
for i←1 to size(A) do
    if tmp < A[i] then
        tmp ← A[i]
return tmp
```

4. 파이썬으로 표현

```
def find_max( A ):
    tmp = A[0]
    for item in A :
        if item > tmp :
            tmp = item
    return tmp
```

알고리즘의 기술 방법

(1) 자연어
- 읽기 쉬움. 단어들을 정확하게 정의하지 않으면 의미 모호.

(2) 흐름도
- 직관적. 이해하기 쉬움. 복잡한 알고리즘→상당히 복잡!

(3) 유사코드
- 프로그램을 구현할 때의 여러 가지 문제들을 감출 수 있음
- 알고리즘의 핵심적인 내용에만 집중 가능

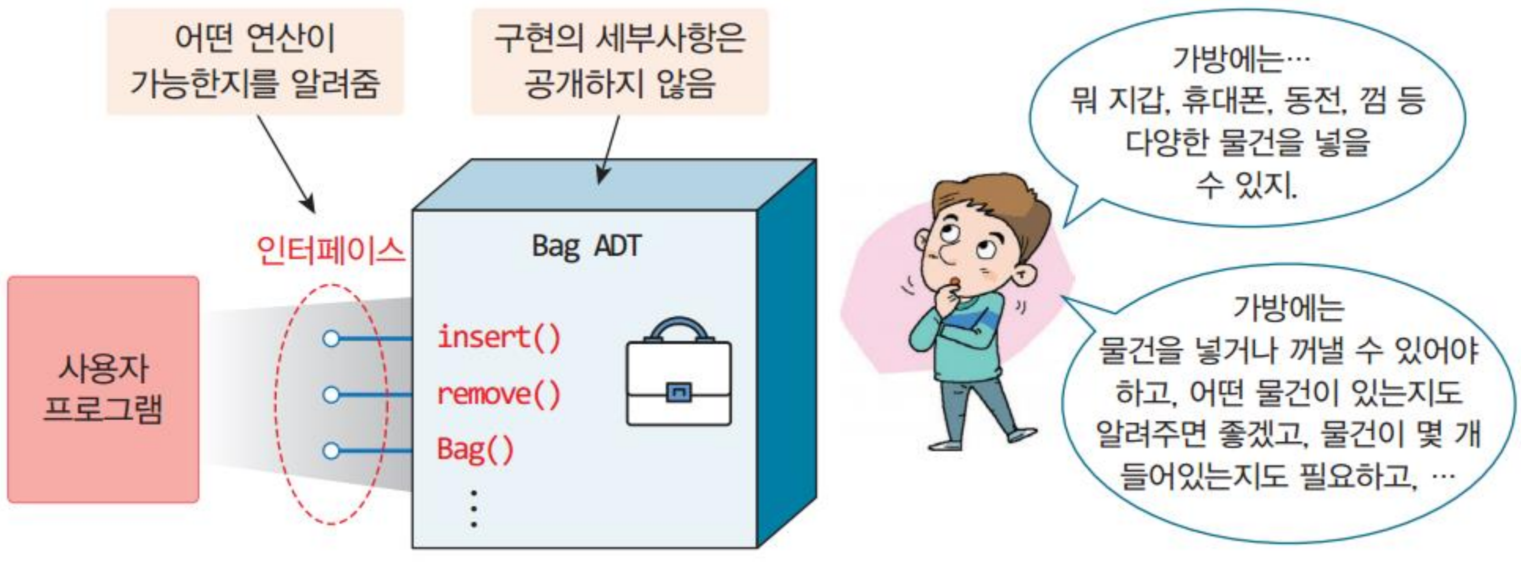
(4) 특정 언어
- 알고리즘의 가장 정확한 기술 가능
- 구현시의 사항들이 알고리즘의 핵심적인 내용들의 이해를 방해
- **파이썬**: C나 자바보다 훨씬 간결한 표현 가능

1.2 추상 자료형

- 추상 자료형(Abstract Data Type, ADT)이란?
- 예) Bag 추상 자료형의 정의
- <실습> Bag 추상 자료형의 구현과 활용

추상 자료형(ADT)이란?

- 프로그래머가 추상적으로 정의한 자료형
  - 데이터 타입을 추상적(수학적)으로 정의한 것
    - 데이터나 연산이 **무엇(what)**인가를 정의함
    - 데이터나 연산을 **어떻게(how)** 구현할 것인지는 정의하지 않음
  - 시스템의 정말 핵심적인 구조나 동작에만 집중



[그림 1.3] 가방(Bag)의 추상 자료형

예) Bag의 추상 자료형

데이터: 중복된 항목을 허용하는 자료들의 저장소. 항목들은 특별한 순서가 없이 개별적으로 저장되지만 항목간의 비교는 가능해야 함.

연산:

- Bag(): 비어있는 가방을 새로 만든다.
- insert(e): 가방에 항목 e를 넣는다.
- remove(e): 가방에 e가 있는지 검사하여 있으면 이 항목을 꺼낸다.
- contains(e): e가 들어있으면 True를 없으면 False를 반환한다.
- count(): 가방에 들어 있는 항목들의 수를 반환한다.

예) Bag 추상 자료형의 구현

- 함수를 이용한 Bag 연산들의 구현 예(파이썬)

```
def contains(bag, e) :          # bag에 항목 e가 있는지 검사하는 함수
    return e in bag             # 파이썬의 in 연산자 사용

def insert(bag, e) :            # bag에 항목 e를 넣는 함수
    bag.append(e)               # 파이썬 리스트의 append메소드 사용

def remove(bag, e) :            # bag에서 항목 e를 삭제하는 함수
    bag.remove(e)               # 파이썬 리스트의 remove메소드 사용

def count(bag):                 # bag의 전체 항목 수를 계산하는 함수
    return len(bag)             # 파이썬의 len 함수 사용
```

Bag을 위한 데이터로는 파이썬의 리스트 사용하여 함수로 구현

예) Bag의 활용

- Bag을 이용한 자료 관리 예

테스트를 위한 코드

```
myBag = []                              # Bag을 위한 빈 리스트를 만듦
insert(myBag, '휴대폰')                 # Bag에 휴대폰 삽입
insert(myBag, '지갑')                   # Bag에 지갑 삽입
insert(myBag, '손수건')                 # Bag에 손수건 삽입
insert(myBag, '빗')                     # Bag에 빗 삽입
insert(myBag, '자료구조')               # Bag에 자료구조 삽입
insert(myBag, '야구공')                 # Bag에 야구공 삽입
print('가방속의 물건:', myBag)          # Bag의 내용 출력

insert(myBag, '빗')                     # Bag에서 '빗'삽입(중복)
remove(myBag, '손수건')                 # Bag에서 '손수건'삭제
print('가방속의 물건:', myBag)          # Bag의 내용 출력
```

C:\WINDOWS\system32\cmd.exe

```
내 가방속의 물건: ['휴대폰', '지갑', '손수건', '빗', '자료구조', '야구공']
내 가방속의 물건: ['휴대폰', '지갑', '빗', '자료구조', '야구공', '빗']
```

1.3 알고리즘의 성능 분석

- 알고리즘의 실행시간을 측정해 보자.

- 알고리즘의 복잡도 분석이란?
  - Bag의 삽입연산
  - n^2 을 구하는 세 알고리즘 비교

- 빅오, 빅오메가, 빅세타 표기법

- 입력 데이터에 따른 성능 차이

알고리즘의 성능분석

- 알고리즘의 성능 분석 기법
  - **실행 시간을 측정하는 방법**
    - 두 개의 알고리즘의 실제 실행 시간을 측정하는 것
    - 실제로 구현하는 것이 필요
    - 동일한 하드웨어를 사용하여야 함
  - **알고리즘의 복잡도를 분석하는 방법**
    - 직접 구현하지 않고서도 수행 시간을 분석하는 것
    - 알고리즘이 수행하는 연산의 횟수를 측정하여 비교
    - 일반적으로 연산의 횟수는 n의 함수
    - **시간 복잡도 분석** : 수행 시간 분석
    - **공간 복잡도 분석** : 수행시 필요로 하는 메모리 공간 분석

(1) 실행시간 측정

- 파이썬의 실행시간 측정 코드 예

```
import time                       # time 모듈 불러오기

myBag = []                        # 비어있는 새로운 가방을 하나 만듦
start = time.time()               # 현재 시각을 start에 저장
insert(myBag, '축구공')           # 실행시간을 측정하려는 코드
...                               # ...
end = time.time()                 # 현재 시각을 end에 저장
print("실행시간 = ", end-start)   # 실행시간(종료-시작)을 출력
```

(2) 복잡도 분석

- 시간 복잡도

  알고리즘(연산)이 실행되는 동안에 사용된 **기본적인 연산 횟수를 입력 크기의 함수**로 나타낸다.

  – 산술, 대입, 비교, 이동의 **기본적인 연산** 고려
  – 알고리즘 수행에 필요한 연산의 개수를 계산
  – 입력의 개수 n에 대한 함수->**시간복잡도 함수**, T(n)

복잡도 분석 예: Bag의 삽입연산

- 방법 1: 리스트의 맨 뒤에 삽입
  - append() 함수 사용

```
def insert(bag, e) :            # bag에 항목 e를 넣는 함수
    bag.append(e)               # 파이썬 리스트의 맨 뒤에 추가
```

  - 효율적 → 바로 삽입 가능

- 방법 2: 리스트의 맨 앞에 삽입
  - Insert() 함수 사용

```
def insert(bag, e) :            # bag에 항목 e를 넣는 함수
    bag.insert(0, e)            # 파이썬 리스트의 맨 앞에 추가
```

  - 비 효율적 → 가방의 모든 물건을 먼저 이동해야 삽입 가능

n^2 을 구하는 문제

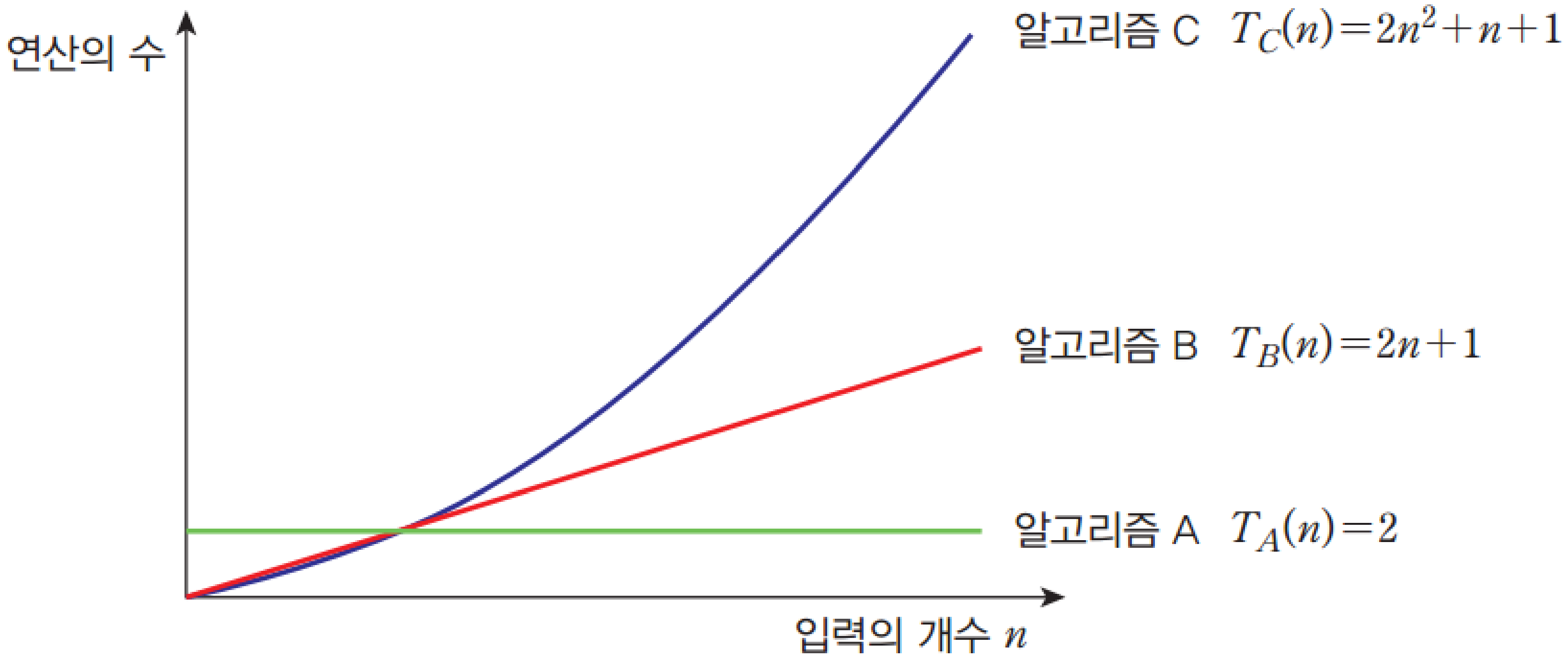
- 3 가지 알고리즘
  - 각 알고리즘이 수행하는 연산의 개수 계산
  - 단 for 루프 제어 연산은 고려하지 않음

알고리즘	A	B	C

유사 코드	`sum ← n*n`	`for i←1 to n do`
		`  sum ← sum + n`
			`for i←1 to n do`
			`  for j←1 to n do`
			`    sum ← sum + 1`

연산 횟수	대입연산: 1 곱셈연산: 1	대입연산: n+1 덧셈연산: n	대입연산: n^2+n+1 덧셈연산: n^2
복잡도 함수	$T_A(n)=2$	$T_B(n)=2n+1$	$T_C(n)=2n^2+n+1$

n^2 을 구하는 세 알고리즘 비교



빅오 표기법

- 차수가 가장 큰 항이 절대적인 영향
  - 다른 항들은 상대적으로 무시
  - 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=100인 경우

T(n)= n² + n + 1

99%

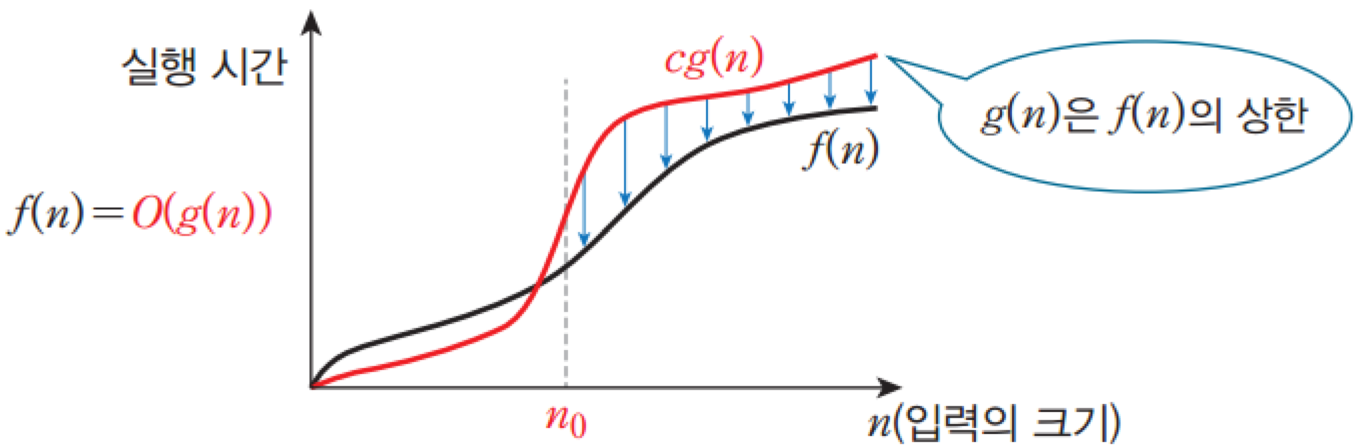
1%

빅오 표기법의 정의

정의 1.3 빅오 표기법

> 두개의 함수 $f(n)$과 $g(n)$이 주어졌을 때 모든 $n > n_0$에 대해 $|f(n)| \leq c\,|g(n)|$을 만족하는 상수 c와 n_0가 존재하면 $f(n) = O(g(n))$이다.

– 연산의 횟수를 대략적(점근적)으로 표기한 것



빅오 표기법의 예

- $f(n)$=5이면 $O(1)$이다. 왜냐하면 $n0$=1, c=10일 때, n≥1에 대하여 5≤10·1이 되기 때문이다.

- $f(n)$=2n+1이면 $O(n)$이다. 왜냐하면 $n0$=2, c=3일 때, $n \geq 2$에 대하여 2n+1≤ 3n이 되기 때문이다.

- $f(n)$=3n^2 +100 이면 $O(n^2)$이다. 왜냐하면 $n0$=100, c=5일 때, $n \geq 100$에 대하여 $3n^2 + 100 \leq 5^2$이 되기 때문이다.

- $f(n)$=$5 \cdot 2^n + 10n^2$+100 이면 $O(2^n)$이다. 왜냐하면 $n0$=1000, c=10일 때, $n \geq 1000$에 대하여 $5 \cdot 2^n + 10n^2 + 100 < 10 \cdot 2^n$이 되기 때문이다.

빅오 표기법의 종류

$O(1)$: 상수형

$O(\log n)$: 로그형

$O(n)$: 선형

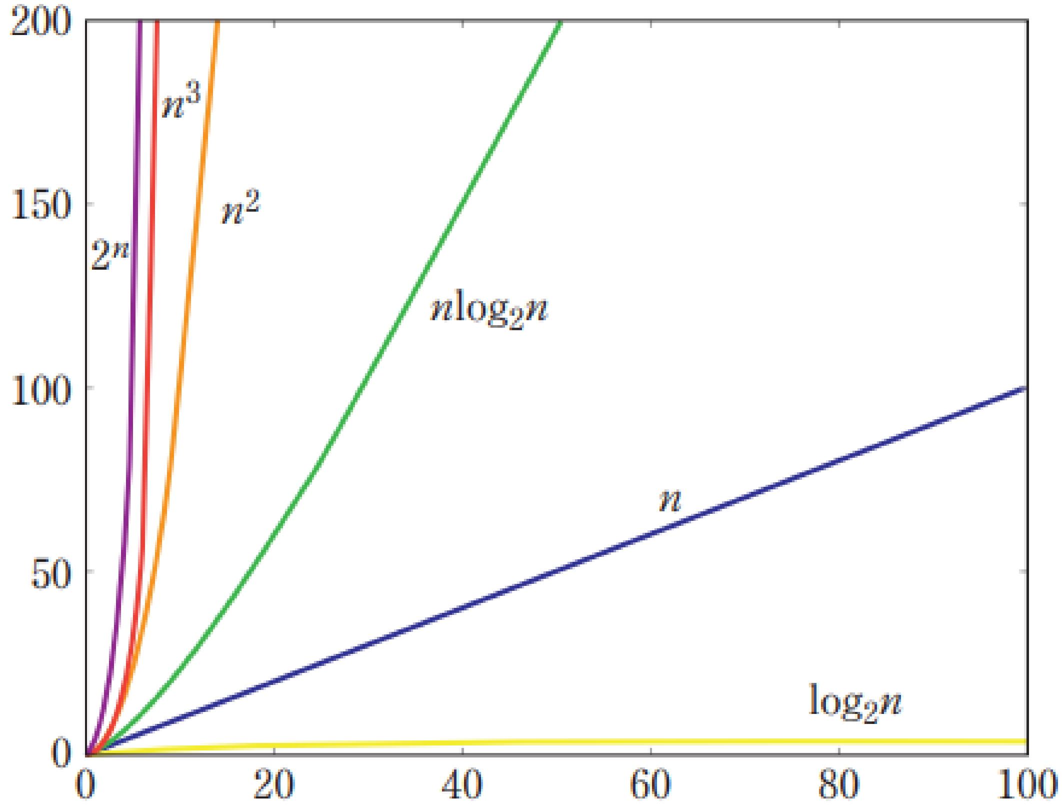
$O(n\log n)$: 선형로그형

$O(n^2)$: 2차형

$O(n^3)$: 3차형

$O(2^n)$: 지수형

$O(n!)$: 팩토리얼형



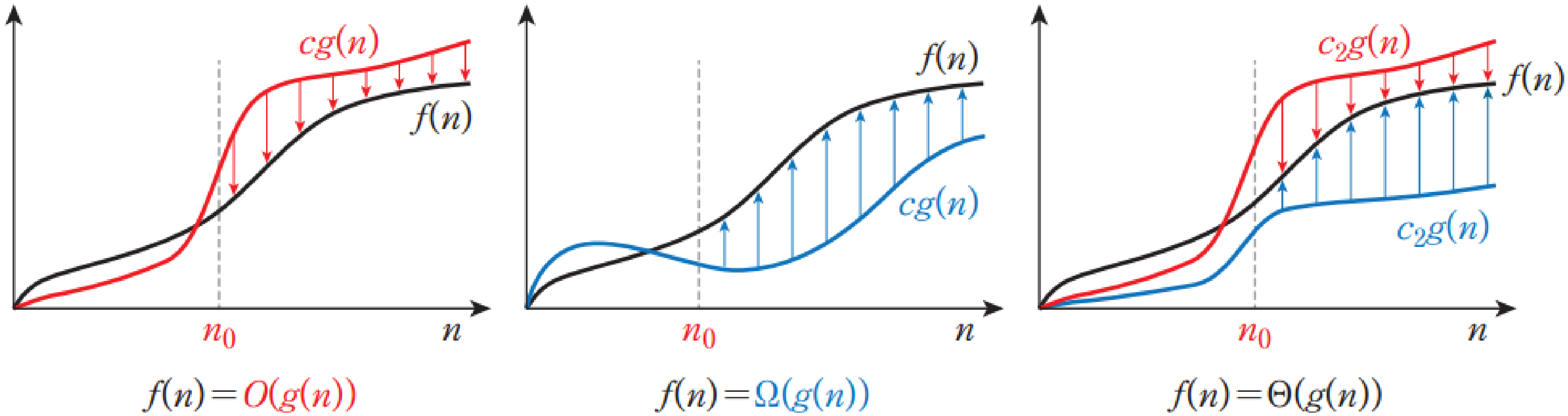
빅오메가와 빅세타 표기법

정의 1.4 빅오메가 표기법

두개의 함수 $f(n)$과 $g(n)$이 주어졌을 때 모든 $n > n_0$에 대해 $|f(n)| \geq c\,|g(n)|$을 만족하는 상수 c와 n_0가 존재하면 $f(n) = \Omega(g(n))$이다.

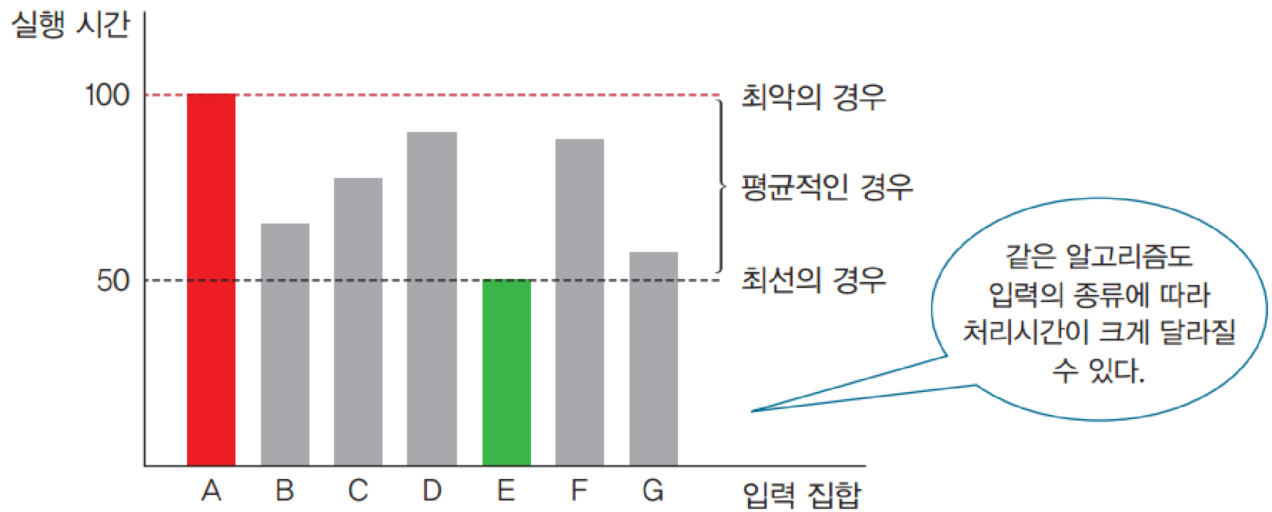
정의 1.5 빅세타 표기법

두개의 함수 $f(n)$과 $g(n)$이 주어졌을 때 모든 $n > n_0$에 대해 $c_1|g(n)| \leq f(n) \leq c_2\,|g(n)|$을 만족하는 상수 c_1, c_2와 n_0가 존재하면 $f(n) = \Theta(g(n))$이다.



최선, 평균, 최악의 경우

- 실행시간은 입력 집합에 따라 다를 수 있음
  - **최악의 경우(worst case)**: 수행 시간이 가장 늦은 경우
    - 가장 널리 사용됨. 계산하기 쉽고 응용에 따라서 중요한 의미를 가짐. (예) 비행기 관제업무, 게임, 로보틱스
  - 최선의 경우(best case): 수행 시간이 가장 빠른 경우
    - 의미가 없는 경우가 많다.
  - 평균의 경우(average case): 수행시간이 평균적인 경우
    - 계산하기가 상당히 어려움.



3 종류의 분석

- 입력의 크기가 충분히 클때의 복잡도
- 입력의 크기: n
- 최악경우 분석(Worst-case Analysis)
  - $O(n^2)$: 알고리즘이 기껏해야 n^2에 비례하는 시간이 든다(빅오)
- 최선경우 분석(Best-case Analysis)
  - $\Omega(n^2)$: 알고리즘이 적어도 n^2에 비례하는 시간이 든다(빅오메가)
- 평균경우 분석(Average-case Analysis)
  - $\Theta(n^2)$: 알고리즘이 항상 n^2에 비례하는 시간이 든다(빅세타)

- 일반적으로 알고리즘의 수행시간은 최악경우 분석으로 표현

- 최악경우 분석: '어떤 입력이 주어지더라도 알고리즘의 수행시간이 얼마 이상은 넘지 않는다'라는 상한(Upper Bound)의 의미
- 최선경우 분석: 가장 빠른 수행시간을 분석

  \- 최적(Optimal) 알고리즘을 찾는데 활용
- 평균경우 분석: 입력의 확률 분포를 가정하여 분석하는데, 일반적으로 균등분포(Uniform Distribution)를 가정

등교 시간 분석

- 집을 나와서 지하철역까지는 5분, 지하철을 타면 학교까지 30분, 강의실까지는 걸어서 10분 걸린다
- 최선경우: 집을 나와서 5분 후 지하철역에 도착하고, 운이 좋게 바로 열차를 탄 경우를 의미한다. 따라서 최선경우 시간은 5 + 20 + 10 = 35분
- 최악경우: 열차에 승차하려는 순간, 열차의 문이 닫혀서 다음 열차를 기다려야 하고 다음 열차가 10분 후에 도착한다면, 최악경우는 5 + 10 + 20 + 10 = 45분

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

(a) 최선 경우 (b) 최악 경우 (c) 평균 경우

1.4 시간 복잡도 분석: 순환 알고리즘

- 순환 알고리즘이란?
- 순환이 더 빠른 예도 있다: 거듭제곱 구하기
- 순환이 훨씬 느린 경우가 많다: 피보나치 수열의 계산
- 복잡한 문제를 쉽게 해결할 수 있다: 하노이의 탑

시간 복잡도 분석: 순환 알고리즘

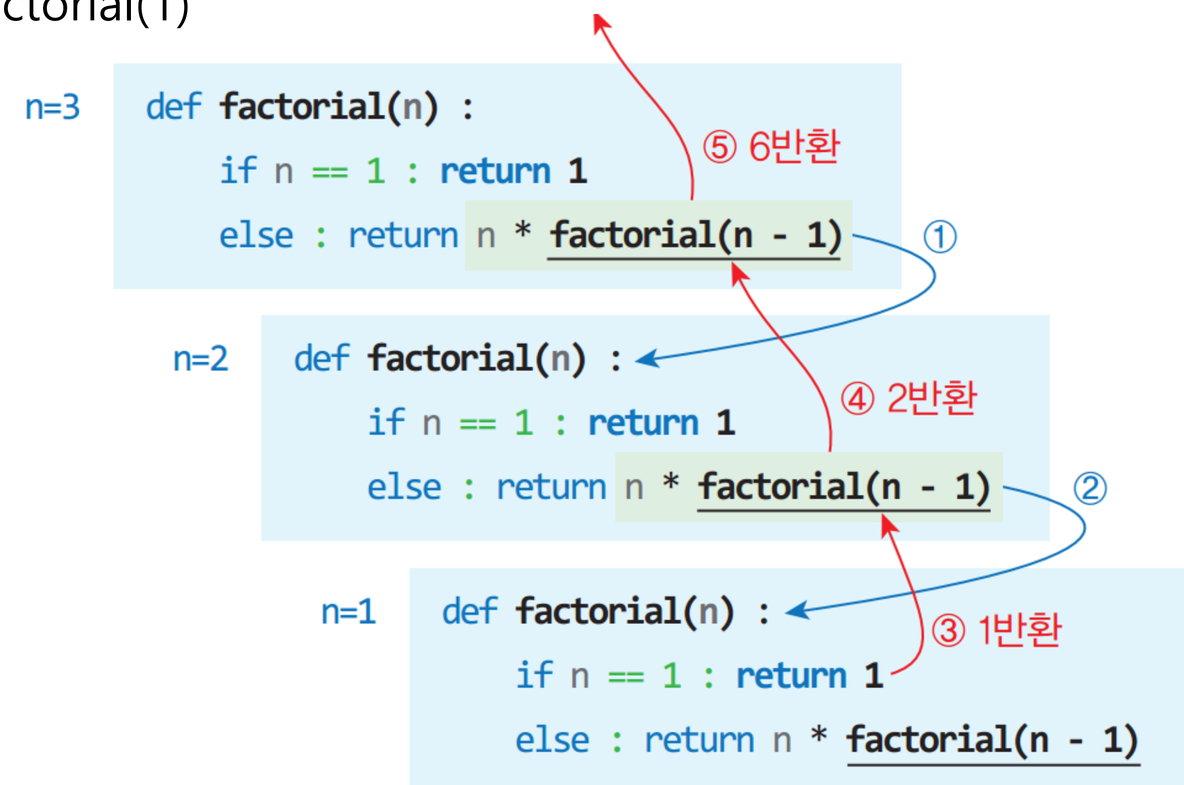
- 순환 알고리즘
  - 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법
  - 정의자체가 순환적으로 되어 있는 경우에 적합
  - 팩토리얼 구하기 $n! = \begin{cases} 1 & n=1 \\ n*(n-1)! & n>1 \end{cases}$
  - 피보나치 수열 $fib(n) = \begin{cases} 0 & if \quad n=0 \\ 1 & if \quad n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$
  - 이항 계수, 하노이의 탑, 이진 탐색, …

팩토리얼 구하기

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

- 순환적인 함수 호출 순서

factorial(3) = 3 * factorial(2)
= 3 * 2 * factorial(1)
= 3 * 2 * 1
= 3 * 2
= 6



팩토리얼: 순환과 반복

- n의 팩토리얼 구하기

순환 구조		반복 구조
$n! = n*(n-1)!$	$\leftrightarrow$	$n! = n*(n-1)*(n-2)*\cdots*1$

- 순환(recursion): $O(n)$ 수행시간과 기억공간의 비효율
  - 순환적인 문제에서는 자연스러운 방법
  - 함수 호출의 오버헤드
- 반복(iteration): $O(n)$
  - for나 while문 이용. 수행속도가 빠름.
  - 순환적인 문제에서는 프로그램 작성이 어려울 수도 있음.

- 대부분의 순환은 반복으로 바꾸어 작성할 수 있음

순환이 더 빠른 예: 거듭제곱 계산

- 방법 1: 반복 구조

```
def power_iter(x, n) :                # 반복으로 xn을 구하는 함수
    result = 1.0
    for i in range(n):                # 루브: n번 반복
        result = result * x
    return result
```

  - 내부 반복문 : $O(n)$

순환적인 거듭제곱 함수

- 방법 2: 순환 구조

```
power(x, n)

if n = 0
    then return 1;
else if n이 짝수
    then return power(x2, n/2);
else if n이 홀수
    then return x*power(x2, (n-1)/2);
```

power(x, n) = power(x^2, n / 2)
$= (x^2)^{n/2}$
$= x^{2(n/2)}$
$= x^n$

power(x, n) = x • power(x^2, (n-1) / 2)
$= x \cdot (x^2)^{(n-1)/2}$
$= x \cdot x^{n-1}$
$= x^n$

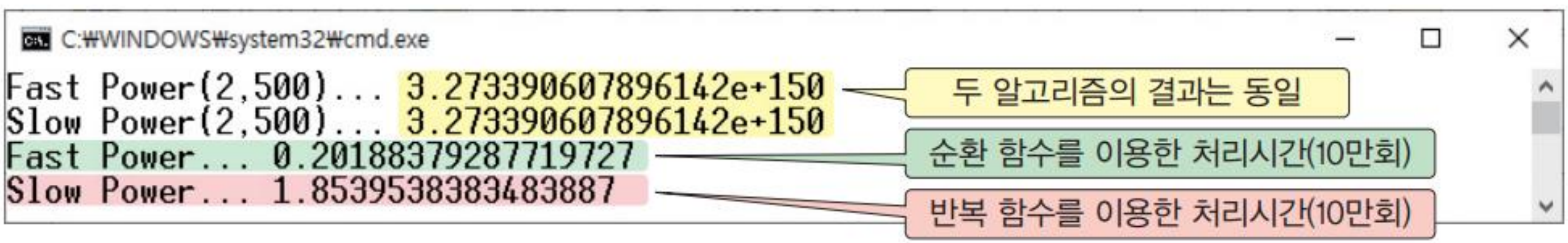
```
def power(x, n) :
    if n == 0 : return 1
    elif (n % 2) == 0 :                  # n이 짝수
        return power(x*x, n//2)          # 정수의 나눗셈 (2.3절 참조)
    else :
        return x * power(x*x, (n-1)//2)  # n이 홀수
```

복잡도 분석

- 순환적인 방법의 시간 복잡도
  - n이 2의 제곱이라면 문제의 크기가 절반씩 줄어든다.

$$2^n \rightarrow 2^{n-1} \rightarrow \cdots 2^2 \rightarrow 2^1 \rightarrow 2^0$$

- 시간 복잡도
  - 순환적인 함수: $\boldsymbol{O(log_2 n)}$
  - 반복적인 함수: $\boldsymbol{O(n)}$



순환이 느린 예: 피보나치 수열

- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열: 0,1,1,2,3,5,8,13,21,…

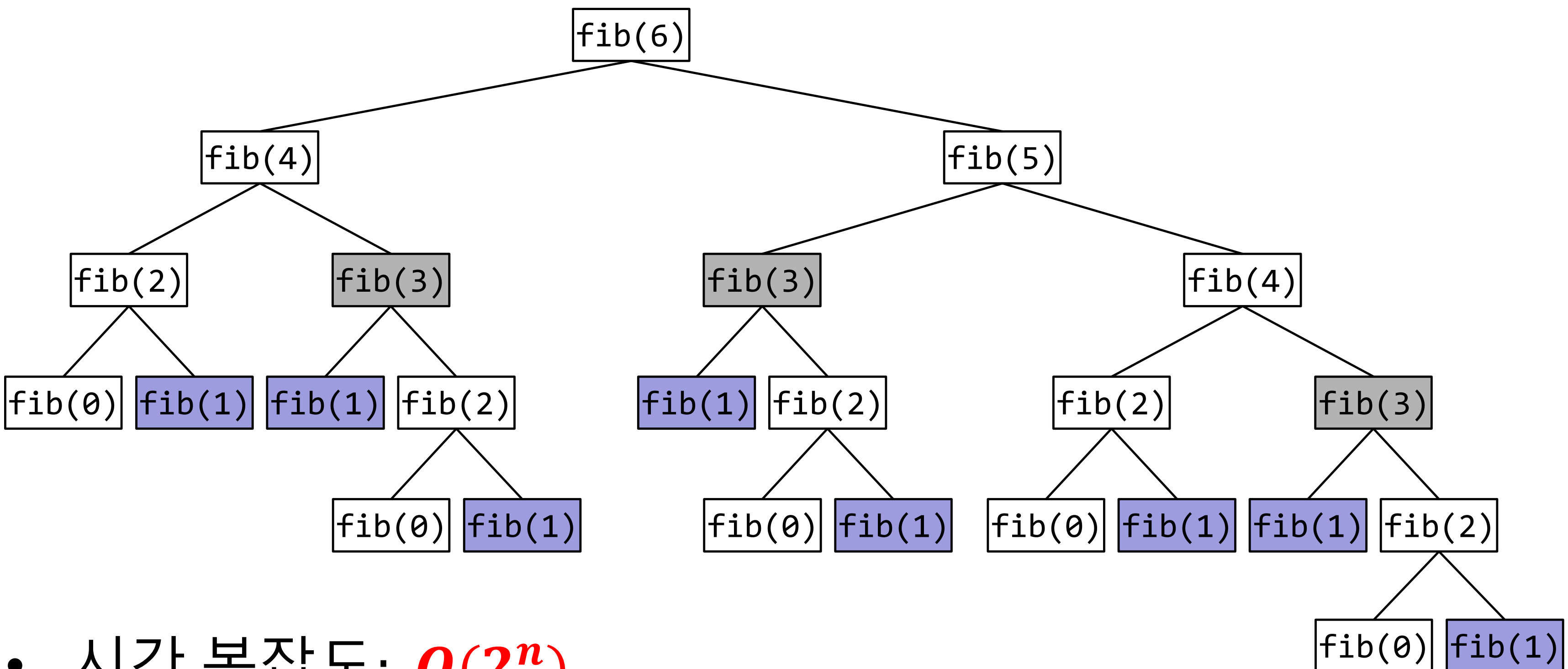
$$fib(n)\begin{cases} 0 & n=0 \\ 1 & n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$$

- 순환적인 구현

```
def fib(n) :                              # 순환으로 구현한 피보나치 수열
    if n == 0 : return 0                  # 종료조건
    elif n == 1 : return 1                # 종료조건
    else :
        return fib(n - 1) + fib(n - 2)    # 순환호출
```

순환적인 피보나치의 비효율성

- 같은 항이 중복해서 계산됨!
  - n이 커지면 더욱 심각



- 시간 복잡도: $O(2^n)$

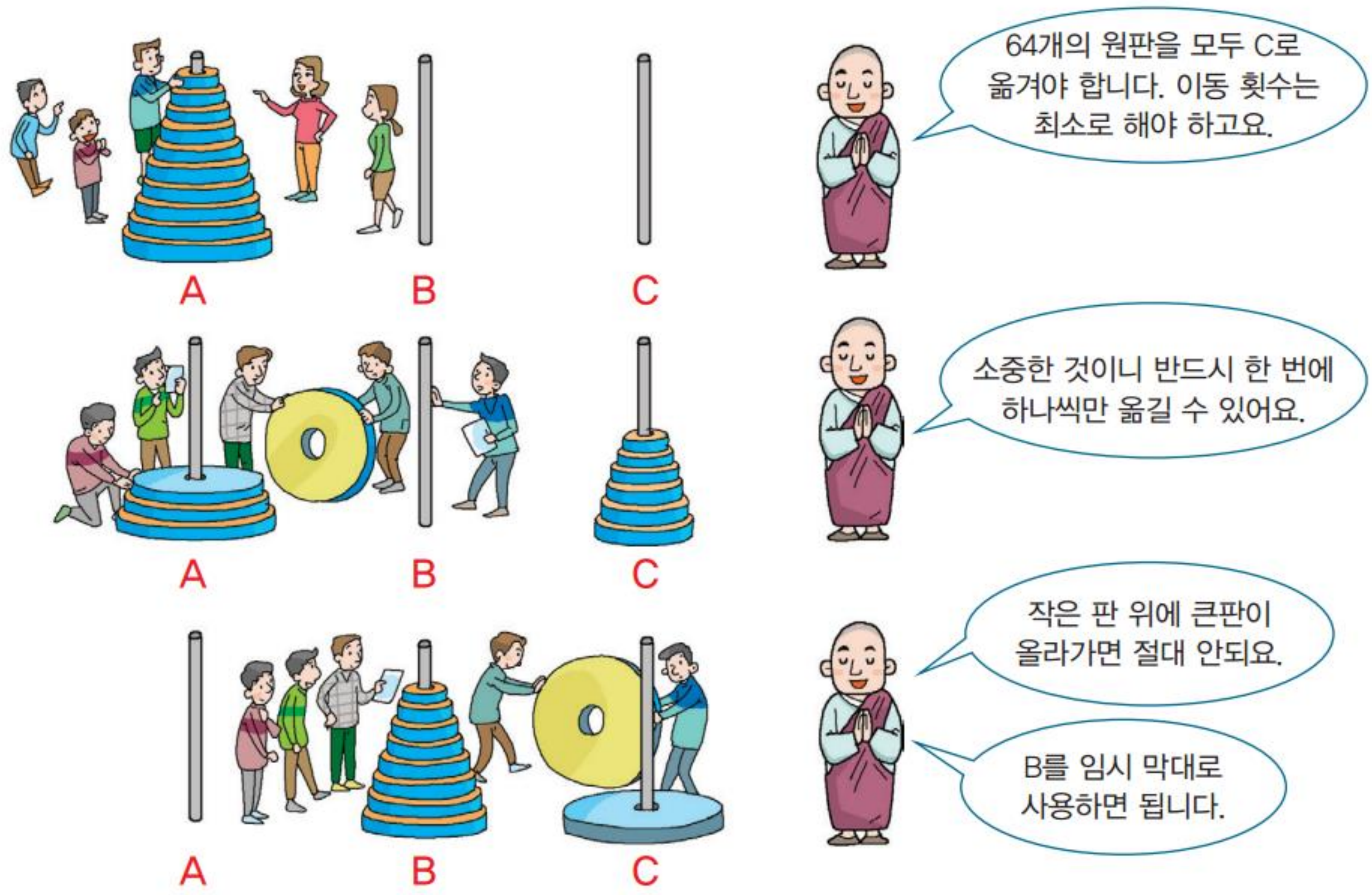
반복적인 피보나치 수열

```
def fib_iter(n) :                       # 반복으로 구현한 피보나치 수열
    if (n < 2): return n

    last = 0
    current = 1
    for i in range(2, n+1) :            # 반복 루프
        tmp = current
        current += last
        last = tmp
    return current
```

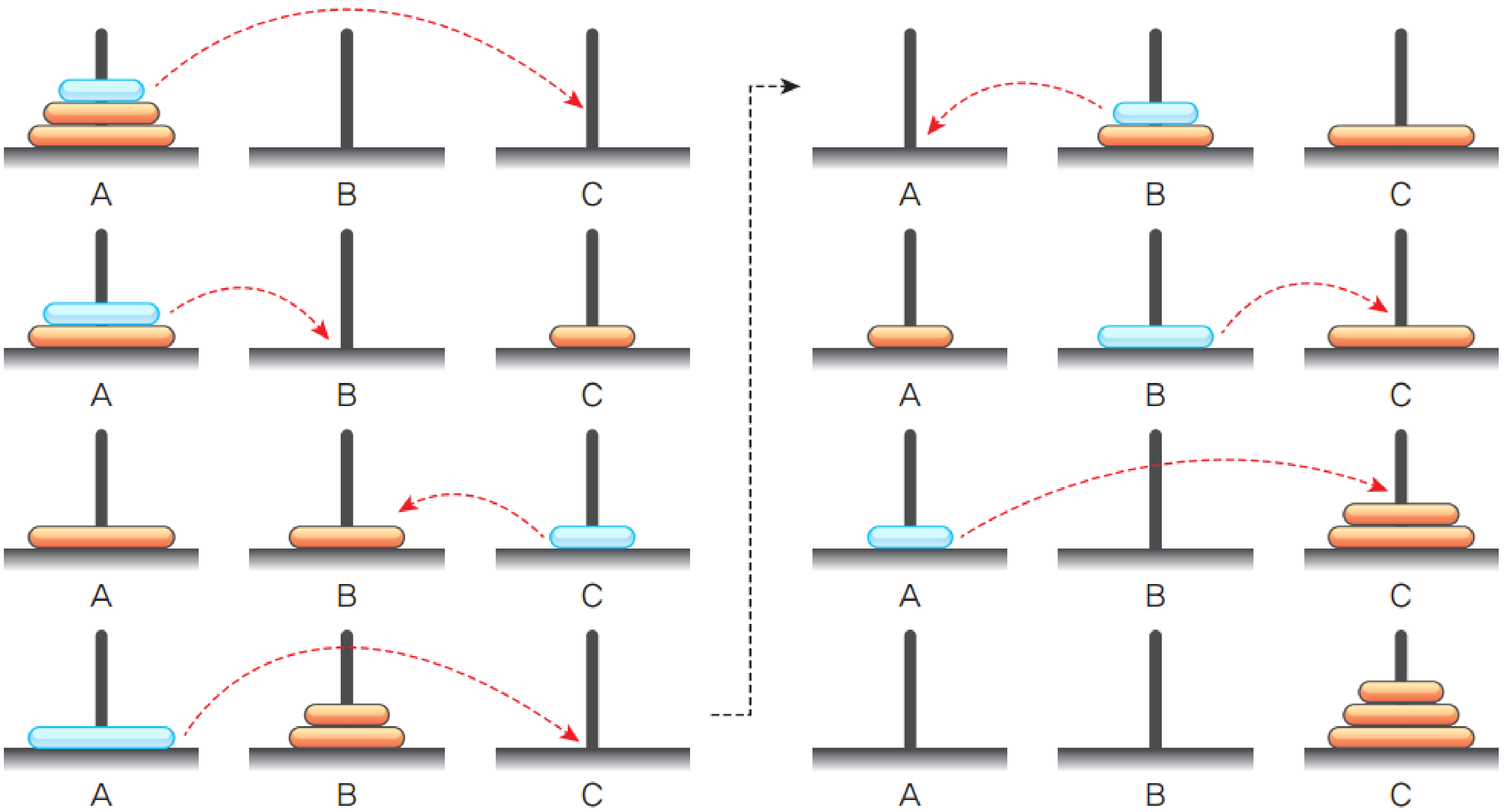
- 시간 복잡도: $O(n)$

하노이 탑 문제

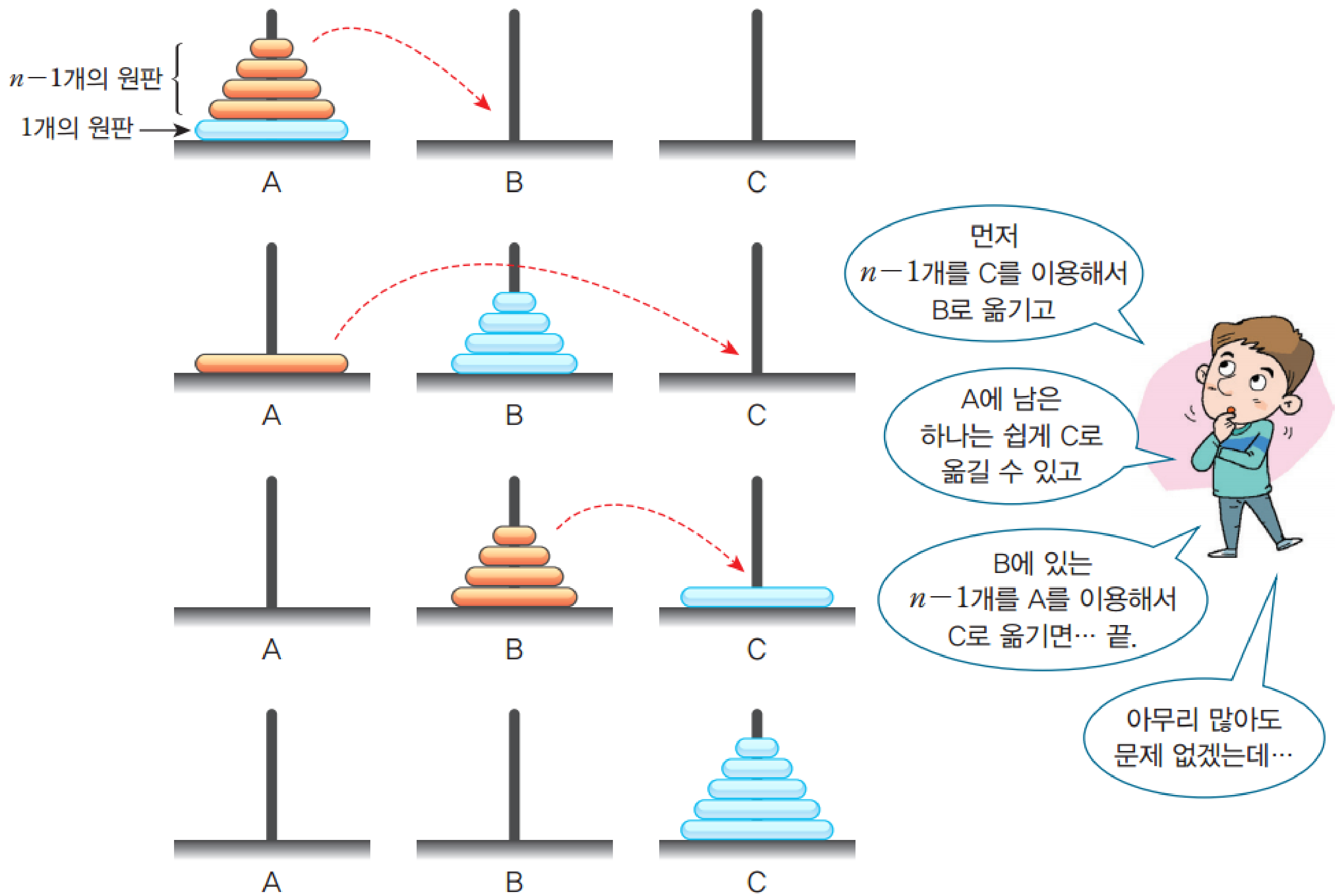


A
B
C
64개의 원판을 모두 C로 옮겨야 합니다. 이동 횟수는 최소로 해야 하고요.
A
B
C
소중한 것이니 반드시 한 번에 하나씩만 옮길 수 있어요.
A
B
C
작은 판 위에 큰판이 올라가면 절대 안되요.
B를 임시 막대로 사용하면 됩니다.

n=3인 경우의 해답



일반적인 경우에는?



$n-1$개의 원판
1개의 원판
A
B
C
먼저 $n-1$개를 C를 이용해서 B로 옮기고
A에 남은 하나는 쉽게 C로 옮길 수 있고
B에 있는 $n-1$개를 A를 이용해서 C로 옮기면… 끝.
아무리 많아도 문제 없겠는데…

구현

- 어떻게 n-1개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
  - **순환을 이용**

```
def hanoi_tower(n, fr, tmp, to) :                    # Hanoi Tower 순환 함수

    if (n == 1) :                                     # 종료 조건
        print("원판 1: %s --> %s" % (fr, to))         # 가장 작은 원판을 옮김
    else :
        hanoi_tower(n - 1, fr, to, tmp)               # n-1개를  to를 이용해 tmp로
        print("원판 %d: %s --> %s" % (n,fr,to))       # 하나의 원판을 옮김
        hanoi_tower(n - 1, tmp, fr, to)               # n-1개를  fr을 이용해 to로
```

```
hanoi_tower(4, 'A', 'B', 'C')                         # 4개의 원판이 있는 경우
```

하노이탑(n=3) 실행 결과

C:₩WINDOWS₩system32₩cmd.exe

```
원판 1: A --> B
원판 2: A --> C
원판 1: B --> C
원판 3: A --> B
원판 1: C --> A
원판 2: C --> B
원판 1: A --> B
원판 4: A --> C
원판 1: B --> C
원판 2: B --> A
원판 1: C --> A
원판 3: B --> C
원판 1: A --> B
원판 2: A --> C
원판 1: B --> C
```

원판의 이동(예 1번 원판을 A에서 B로 이동한다.)

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
&= 2(\ 2T(n-2)\ +1\ ) + 1 \\
&= 2(\ 2(\ 2(T(n-3)\ +1\ )\ +1\ )\ +1 \\
&= 2^{n-1}T(1) + \ldots \\
&= 2^{n-1} + \ldots \\
&= O(2^n)
\end{aligned}
$$

1장 연습문제, 실습문제

감사합니다!