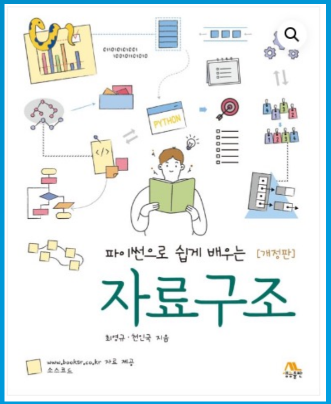


01 CHAPTER

자료구조와 알고리즘

1장. 자료구조와 알고리즘

1.1 자료구조와 알고리즘

1.2 추상 자료형

1.3 알고리즘의 성능 분석

1.4 시간 복잡도 분석: 순환 알고리즘

1.1 자료구조와 알고리즘

- 자료구조(Data Structure)란?
  - 컴퓨터에서 자료를 정리하고 조직화하는 다양한 구조
  - 생활 속의 다양한 사물과 관련된 자료구조의 예

배달할 선물 목록: 리스트(List)

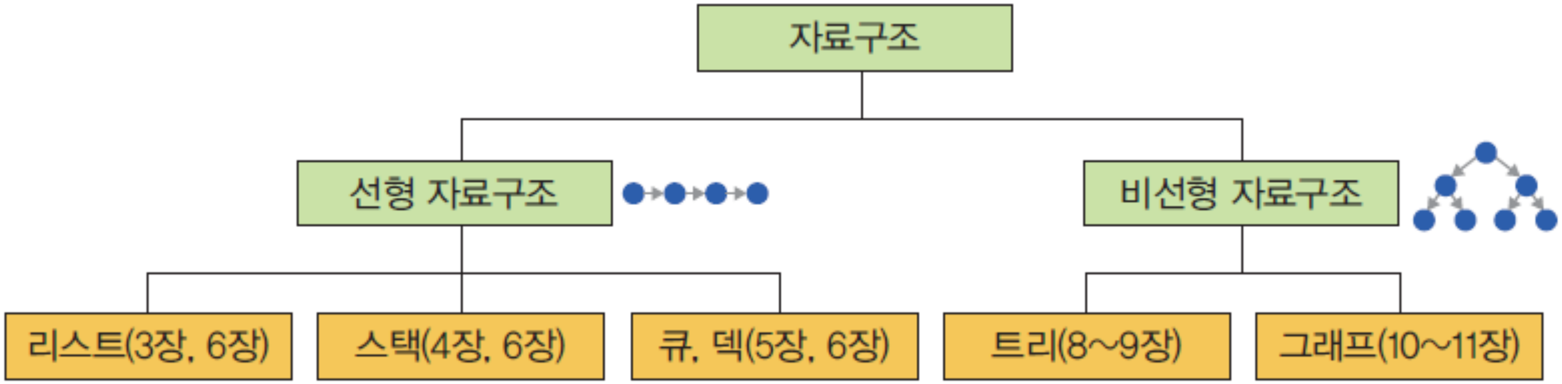
접시나 물건 쌓기: 스택(Stack)

매표소 줄서기: 큐(Queue)

지하철 노선도: 그래프(Graph)

직장의 조직도: 트리(Tree)

자료구조의 분류



- 선형 자료구조
  - 자료를 순서적으로 나열하여 저장하는 창고
  - 접근 방법에 따라 다시 세분화: 리스트, 스택, 큐, 덱 등
- 비선형 자료구조
  - 복잡한 연결 관계의 자료 표현
    - 트리: 회사의 조직도나 컴퓨터의 폴더와 같은 계층 구조
    - 그래프: 가장 복잡한 연결 관계를 표현

알고리즘이란?

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

- 프로그램 = 자료구조 + 알고리즘

알고리즘의 조건

- 입력: 0개 이상의 입력이 존재하여야 한다.
- 출력: 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) 자연어

– 읽기 쉬움. 단어들을 정확하게 정의하지 않으면 의미 모호.

find_max(A)

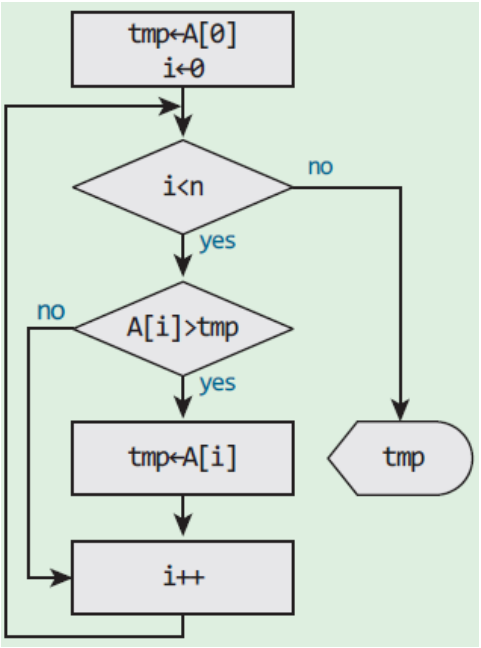
1. 배열 A의 첫 번째 요소를 변수 tmp에 복사한다.
2. 배열 A의 다음 요소들을 차례대로 tmp와 비교하여, 더 크면 그 값을 tmp로 복사한다.
3. 배열 A의 모든 요소를 비교했으면 tmp를 반환한다.

(2) 흐름도

– 직관적.

– 이해하기 쉬움.

– 복잡한 알고리즘→상당히 복잡!



알고리즘의 기술 방법

(3) 유사 코드

- 프로그램을 구현할 때의 여러 가지 문제들을 감출 수 있음
- 알고리즘의 핵심적인 내용에만 집중 가능

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

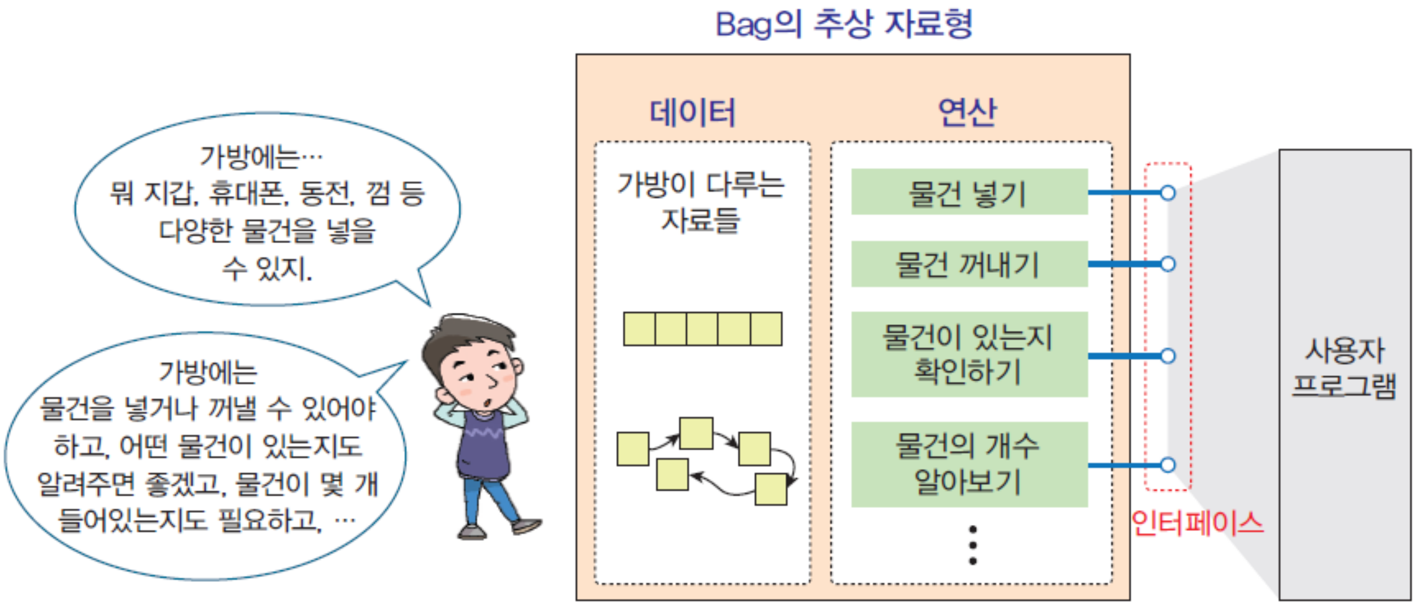
```python
def find_max( A ):
    tmp = A[0]
    for item in A :
        if item > tmp :
            tmp = item
    return tmp
```

(4) 특정 언어

- 구현시의 사항들이 알고리즘의 핵심적인 내용들의 이해를 방해
- **파이썬**: C나 자바보다 훨씬 간결한 표현 가능
  - 바로 실행할 수 있음!

1.2 추상 자료형

- 프로그래머가 추상적으로 정의한 자료형
  - 어떤 자료를 다루고 이들에 대해 어떤 연산이 제공되는지를 기술
    - 데이터나 연산이 **무엇(what)**인가를 정의함
    - 데이터나 연산을 **어떻게(how)** 구현할 것인지는 정의하지 않음
  - 예: 가방(Bag)이 다루는 데이터와 연산들



예) Bag의 추상 자료형

- 데이터
  - 중복된 항목을 허용하는 자료의 모임.
  - 항목들 사이에 순서는 없지만 서로 비교할 수는 있어야 함.

- 연산
  - insert(e): 가방에 항목 e를 넣는다.
  - remove(e): 가방에 e가 있는지 검사해 있으면 이 항목을 꺼낸다.
  - contains(e): e가 들어있으면 True를 없으면 False를 반환한다.
  - count(): 가방에 들어 있는 항목들의 수를 반환한다.

예) Bag 추상 자료형의 구현

- 파이썬 함수를 이용한 Bag 연산 구현 예

```python
def contains(bag, e) :
    return e in bag

def insert(bag, e) :
    bag.append(e)

def remove(bag, e) :
    bag.remove(e)

def count(bag):
    return len(bag)
```

bag에 항목 e가 있는지 검사하는 함수. 파이썬의 in 연산자를 이용했는데, e가 bag에 있으면 True를 없으면 False를 반환함

bag에 새로운 항목 e를 넣는 함수. 파이썬 리스트의 append() 연산을 이용해 리스트의 맨 뒤에 e를 추가함

bag에서 항목 e를 삭제하는 함수. 파이썬 리스트의 remove() 연산을 이용해 구현함

bag에 들어 있는 항목의 수를 반환하는 함수. 파이썬 내장함수 len()을 이용함

예) Bag의 활용

- Bag을 이용한 자료 관리 예

```
myBag = []
insert(myBag, '휴대폰')
insert(myBag, '지갑')
insert(myBag, '손수건')
insert(myBag, '빗')
insert(myBag, '자료구조')
insert(myBag, '야구공')
print('가방속의 물건:', myBag)

insert(myBag, '빗')
remove(myBag, '손수건')
print('가방속의 물건:', myBag)
```

bag을 위한 새로운 배열을 만듦. 자료구조의 데이터를 저장하는 공간

새로운 bag인 myBag에 '휴대폰', '지갑', '손수건', '빗', '자료구조', '야구공'을 순서대로 삽입함

현재 myBag의 내용을 화면에 출력함

myBag에 '빗'을 추가로 삽입하고, '손수건'을 삭제함

변경된 myBag 내용을 화면에 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

- **실행 시간을 측정하는 방법**
  - 알고리즘의 실제 실행 시간을 측정하는 것
  - 알고리즘을 실제로 구현해야 함
  - 동일한 HW/SW 환경을 사용하여야 함

- **알고리즘의 복잡도를 분석하는 방법**
  - 직접 구현하지 않고서도 수행 시간을 분석
  - 알고리즘이 수행하는 연산의 횟수를 측정하여 비교
  - 연산의 횟수는 입력의 크기 n의 함수
    - **시간 복잡도 분석** : 수행 시간 분석
    - **공간 복잡도 분석** : 필요한 메모리 공간 분석

(1) 실행시간 측정

- 파이썬의 실행시간 측정 코드 예

```
import time

myBag = []
start = time.time()
insert(myBag, '축구공')
...
end = time.time()
print("실행시간 = ", end-start)
```

- 시각 측정 함수를 사용하기 위해 time 모듈을 프로그램에 포함시킴
- 현재 시각을 start에 저장(알고리즘 처리 전)
- 실행시간을 측정하려는 코드(알고리즘)가 들어가는 부분
- 현재 시각을 end에 저장(종료 시각)
- 알고리즘의 전체 실행시간(end-start)

(2) 복잡도 분석

- 구현하지 않고 알고리즘의 효율성을 평가
  – 알고리즘의 연산 횟수를 대략적으로 계산
  – 복잡도 함수 T(n): 입력의 크기 n에 대한 알고리즘 연산 횟수

- 예: 1부터 n까지 합을 구하는 두 가지 알고리즘

복잡도 함수

- 알고리즘 1: $T(n) = 2n+1$

```
calc_sum1( n )
    sum ← 0
    for i ← 1 to n then
        sum ← sum + i
    return sum
```

← 연산자 1번 수행

반복 제어 연산은 무시

n번 반복되는 반복문 안에 있으므로 ← 연산자와 + 연산자가 각각 n번씩 수행

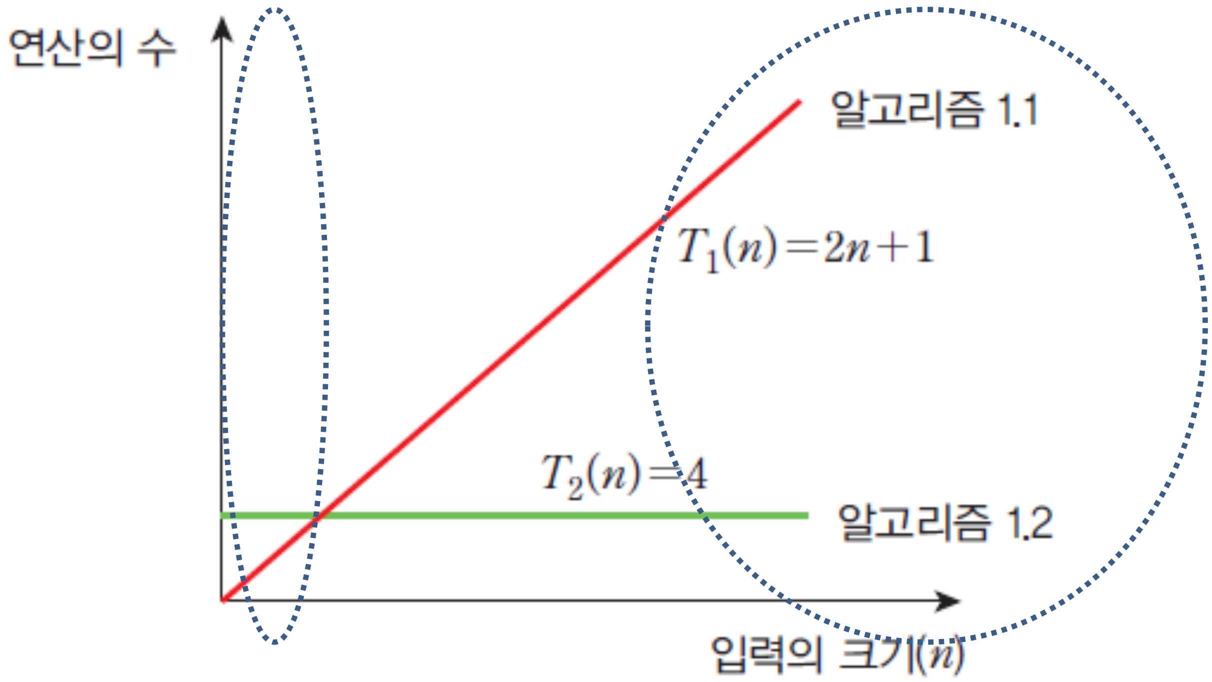
- 알고리즘 2: $T(n) = 4$

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

← 연산자와 *, +, / 연산자가 각각 1번씩 수행

알고리즘 비교

- 두 알고리즘의 성능 비교

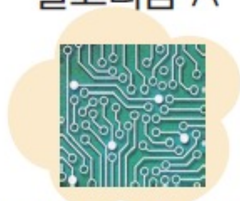


복잡도의 점근적 표기

- 복잡도를 매우 간단한 형태로 단순화하는 이유?
  - 예: 두 정렬 알고리즘의 성능 비교

문제: n개의 숫자를 오름차순으로 정렬해라.

알고리즘 A



$65536n + 2000000$

알고리즘 B

$n^2 + 2n$

n(입력의 크기)	알고리즘 A $65536n+2000000$	비교	알고리즘 B n^2+2n	
10	2,655,360	>	120	n이 작을 때는 B가 효율적인 것처럼 보임
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	n이 커질수록 B가 훨씬 나쁘다는 것이 서서히 드러남
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	

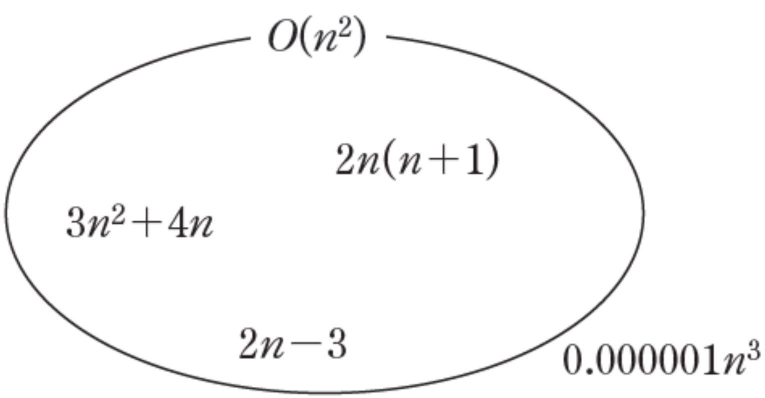
점근적 표기(asymptotic notation)

- 여러 항을 갖는 복잡도 함수를 최고차항 만을 계수 없이 취해 단순하게 표현하는 방법
  - n이 무한대에 가까워지면 최고차항을 제외한 나머지 항의 효과는 거의 없는 것이나 마찬가지
  - 알고리즘의 증가 속도만을 표현
- 예
  - T(n) = 65536n+2000000 -->n
  - T(n) = $n^2 + 2n$ -->n^2
- 점근적 상한/하한/동일 등급
  - 빅 오, 빅 오메가, 빅 세타 표현

빅오 표기법

- O(g(n))
  - 증가속도가 g(n)과 같거나 낮은 모든 복잡도 함수를 모두 포함

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

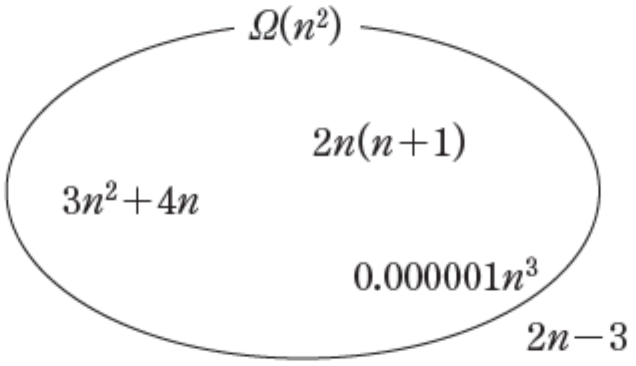


  - 처리 시간의 상한
    - 어떤 경우에도 g(n)에 비례하는 시간 안에는 반드시 완료됨을 의미

빅오메가, 빅세타 표기법

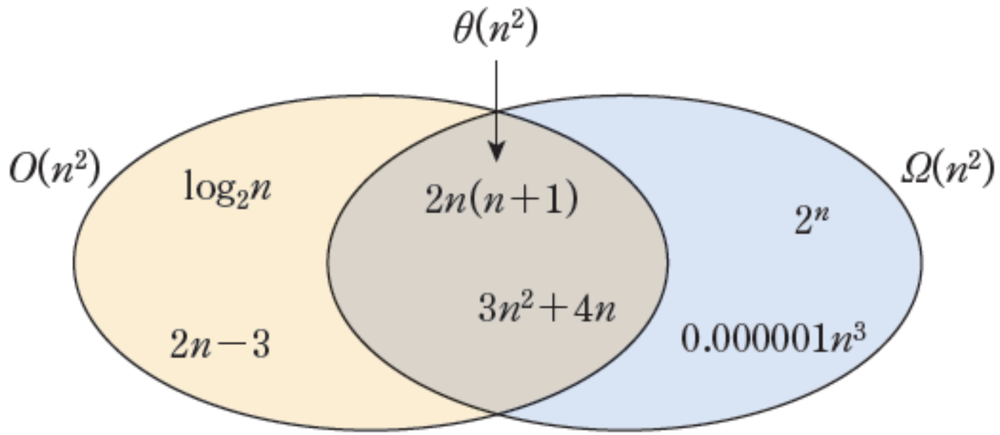
- 빅 오메가: 처리 시간의 하한

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

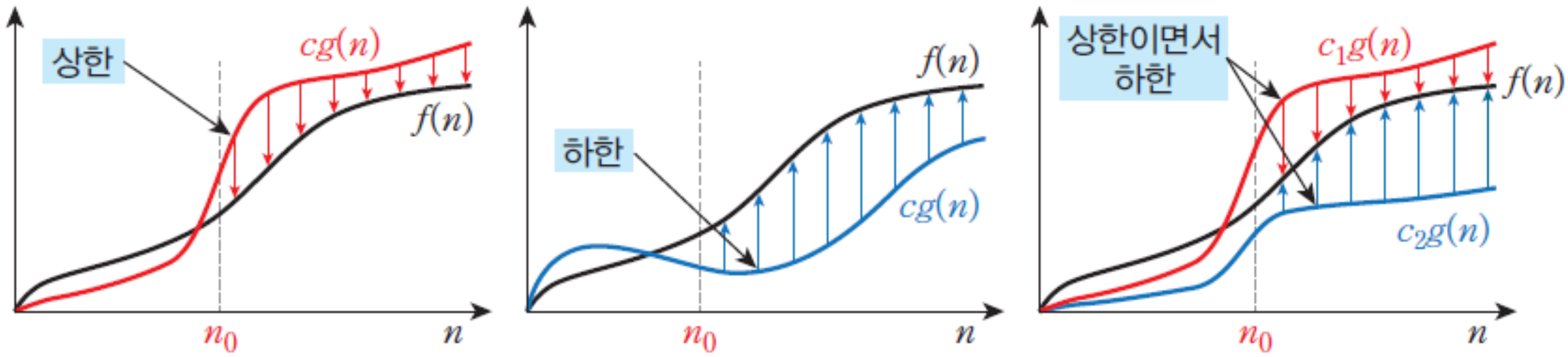


- 빅 세타: 상한이면서 하한

$$3n^2+4n \in \theta(n^2)$$
$$2n-3 \notin \theta(n^2)$$
$$0.000001n^3 \notin \theta(n^2)$$



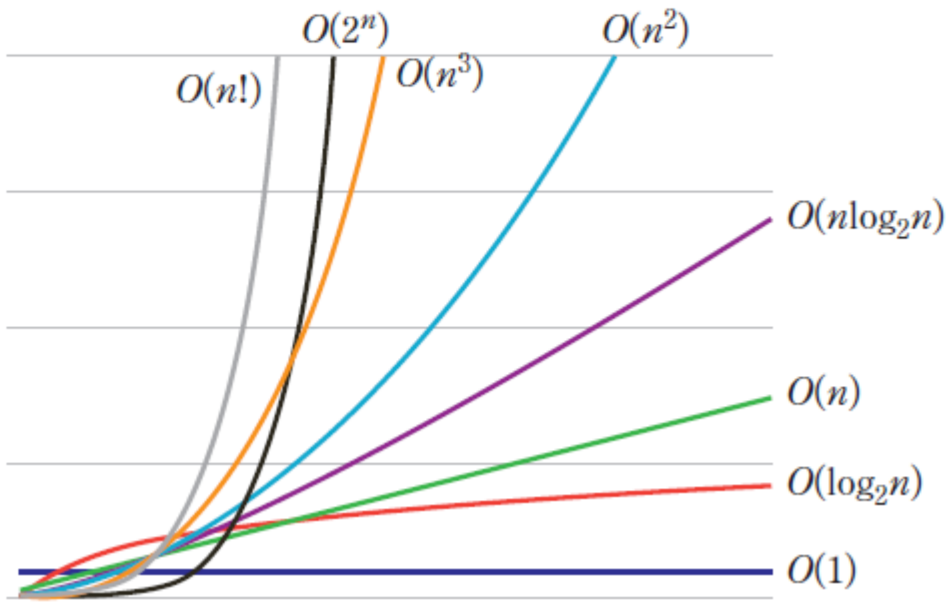
빅오, 빅오메가, 빅세타의 비교



$f(n) = O(g(n))$ $f(n) = \Omega(g(n))$ $f(n) = \Theta(g(n))$

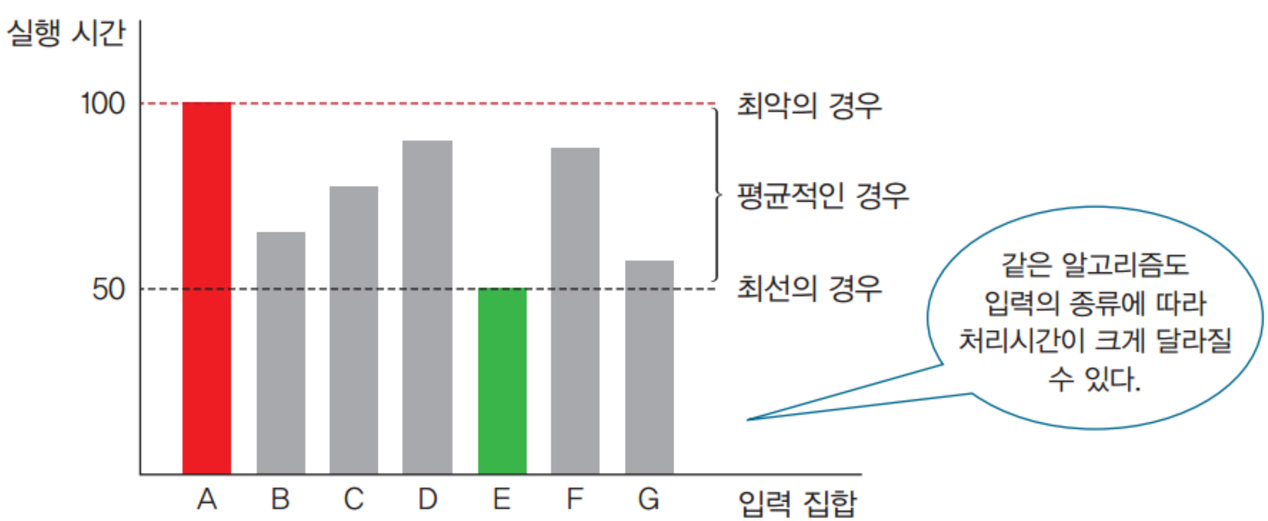
시간 복잡도 함수들의 증가속도

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



최선, 평균, 최악의 경우

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



복잡도 분석의 예: 순차탐색

- 순차 탐색
  – 정렬되지 않은 배열에서 어떤 값을 찾는 알고리즘

```
01  def search(A, key):
02      n = len(A)
03      for i n range(n) :
04          if A[i] == key :
05              return i
06      return -1
```

- 배열 A에서 key인 항목의 위치를 찾는 함수
- 알고리즘의 입력 크기는 배열 A의 크기
- i에 0부터 n-1까지 순서대로 대입
- key인 항목을 찾으면 인덱스 i 반환
  4행이 가장 많이 수행됨(n번)
- A에 key가 없으면(탐색 실패) -1 반환

  – 입력의 크기는?
  – 기준 연산은?
  – 최선/최악/평균의 경우로 나누어 분석해야 할까?

최선, 평균, 최악의 경우

- 최선의 경우(best case)
  - 맨 처음이 찾는 항목인 경우
  - T(n) = 1 → O(1)
- 최악의 경우(worst case)
  - 맨 마지막에 있거나 배열에 찾는 값이 없는 경우
  - T(n) = n → O(n)
- 평균의 경우(average case)
  - "평균"에 대한 가정이 필요
    - 예) 배열의 모든 숫자가 골고루 한 번씩 key로 사용되는 경우

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

  - T(n) = (n+1)/2 → O(n)

1.4 시간 복잡도 분석: 순환 알고리즘

- 같은 일을 되풀이하는 방법: 반복 / 순환
- **반복(iteration)**
  - for나 while 등의 반복문 이용
  - 보통은 수행속도가 빠름
  - 문제에 따라 프로그램 작성이 어려울 수 있음

- **순환(recursion)**
  - 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법
  - 함수 호출의 오버헤드가 있음
  - 간결한 코딩이 가능

순환이 적합한 경우

- 정의자체가 순환적으로 되어 있는 문제나 자료구조
  - 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, 등
  - 이진 트리

예: n의 팩토리얼 구하기

- 반복 구조

$$n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$$

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

- **순환 구조**

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

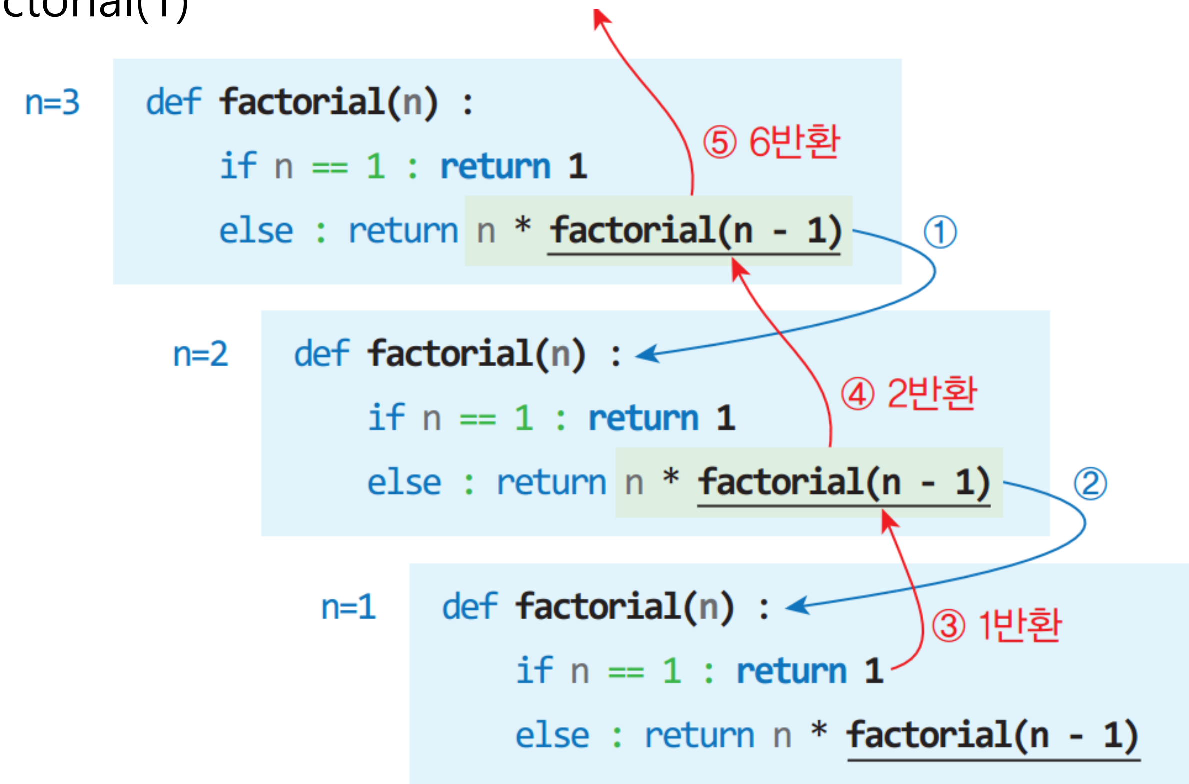
```python
def factorial(n) :
    if n == 1 :
        return 1
    else :
        return n * factorial(n-1)
```

팩토리얼 구하기

- 순환적인 함수 호출 순서

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

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

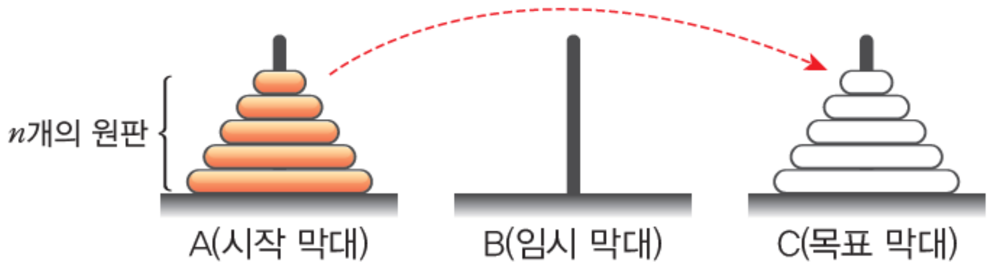


순환과 반복

- 대부분의 순환은 반복으로 바꾸어 작성할 수 있음
- 팩토리얼 문제의 경우 순환과 반복 알고리즘의 시간 복잡도는 $O(n)$ 으로 동일함

- 매우 유명한 알고리즘에서 순환이 흔히 사용됨
  – 예: 이진 탐색, 퀵 정렬 등

하노이 탑 문제

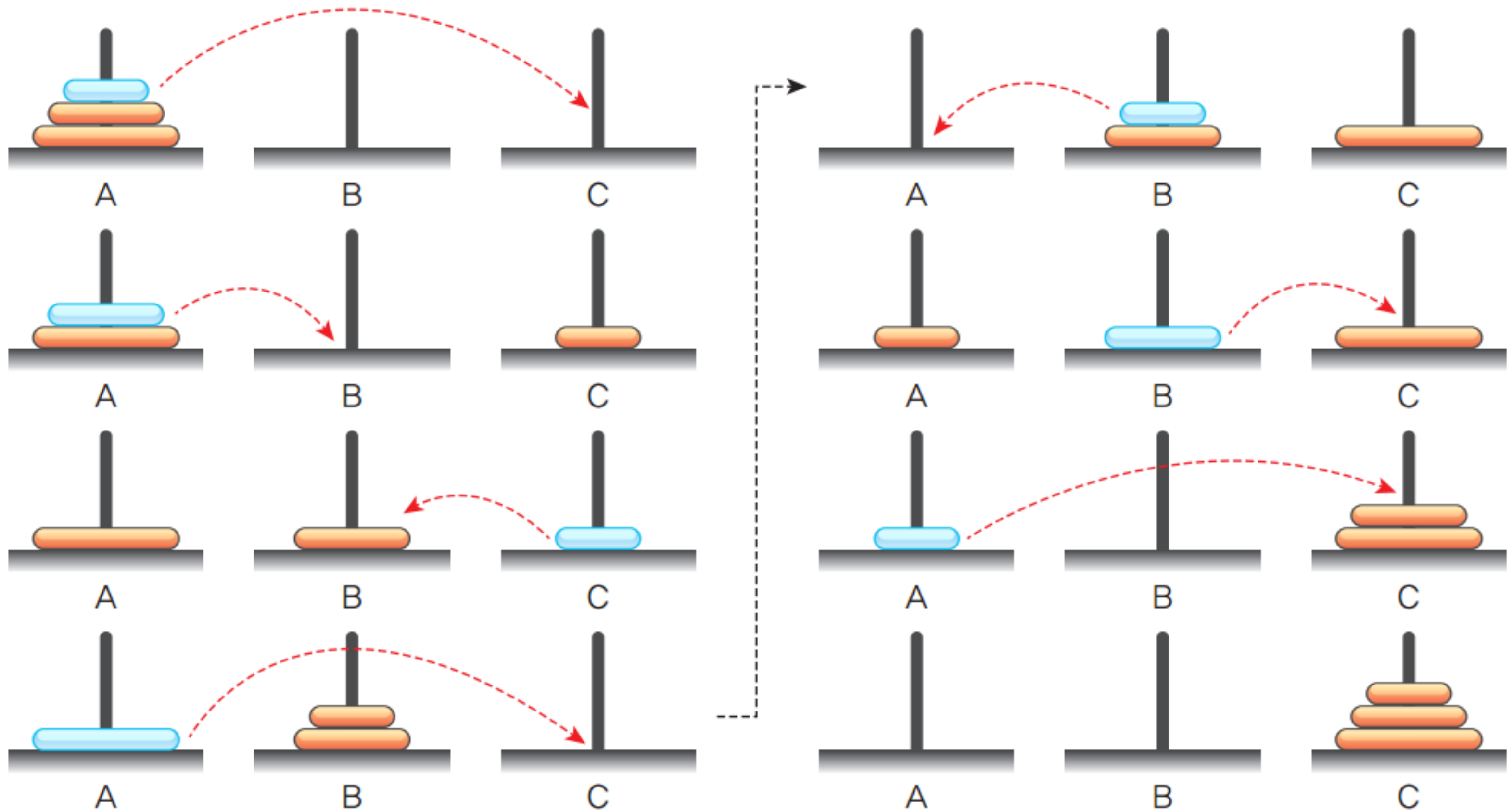


[그림 1.11] n개의 원판을 옮기는 하노이의 탑 퍼즐

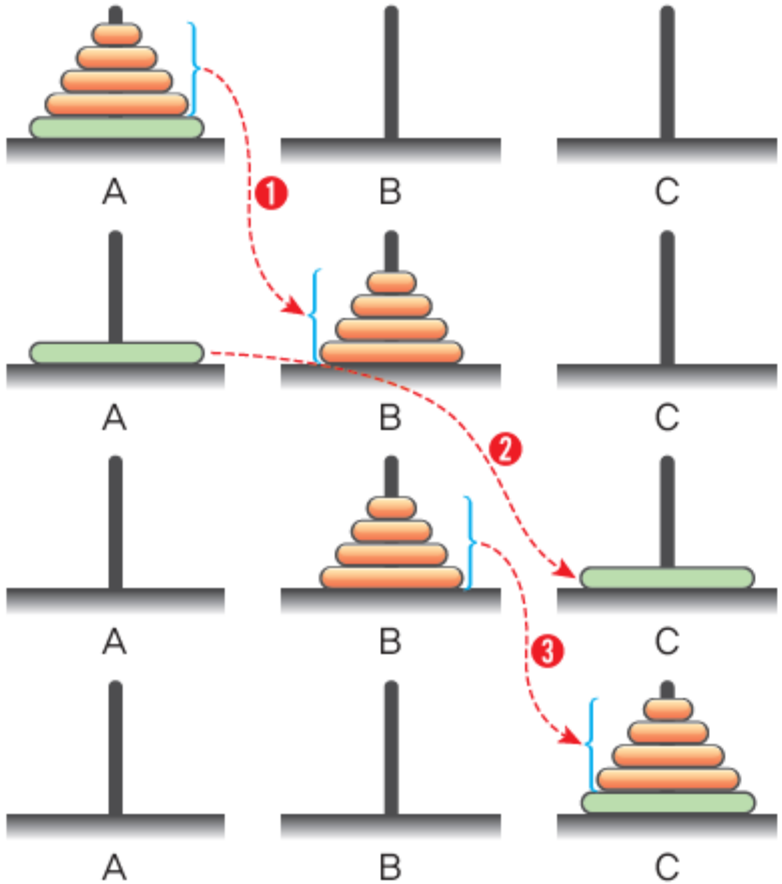
막대 A에 쌓여있는 n개의 원판을 모두 C로 옮겨라. 단, 다음 조건을 만족해야 한다.

- 한 번에 하나의 원판만 옮길 수 있다.
- 맨 위에 있는 원판만 옮길 수 있다.
- 크기가 작은 원판 위에 큰 원판을 쌓을 수는 없다.
- 중간 막대 C를 임시 막대로 사용할 수 있지만 앞의 조건은 지켜야 한다.

n=3인 경우의 해답



일반적인 경우에는?



❶ 단계: A에 있는 $n-1$개의 원판을 C를 임시 막대로 이용해서 B로 이동

❷ 단계: A에 남은 하나의 원판을 C로 이동

❸ 단계: B에 있는 n–1개의 원판을 A를 임시 막대로 이용해서 C로 이동

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.)

순환 알고리즘의 복잡도 계산

- 복잡도 함수가 순환적인 형태
  - T(n) = 2T(n-1) + 1
  - T(1) = 1

- 연속 대치법

$$
\begin{aligned}
T(n) &= 2T(n-1)+1 \\
&= 2(\boxed{2T(n-2)+1})+1 \\
&= 2(\ 2(\boxed{2(T(n-3)+1})+1\ )+1 \\
&= \underline{2^{n-1}T(1)} + \underline{(2^{n-2}+\cdots+2^{1}+2^{0})} \\
&= \underline{2^{n-1}} + \underline{2^{n-1}-1} \\
&= 2^{n}-1 \\
&= O(2^{n})
\end{aligned}
$$

$T(n-1)$를 $2T(n-2)+1$로 대치

$T(n-2)$를 $2T(n-3)+1$로 대치

식 정리

등비수열 합 공식 적용

초기 조건 $T(1)=1$ 대입

감사합니다!