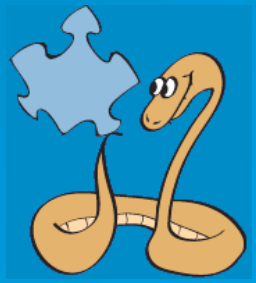
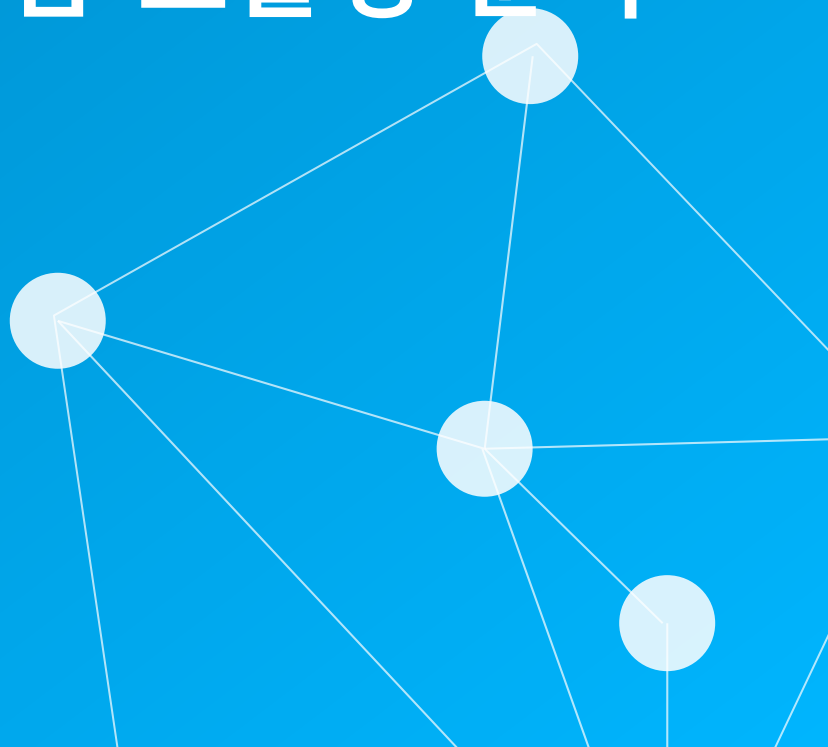
파이썬 알고리즘

ALGORITHMS IN PYTHON 최영규 지음

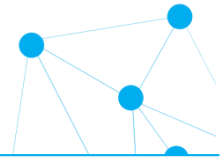


CHAPTER 02

알고리즘 효율성 분석



학습 내용



2.1 효율성 분석의 기초

2.2 점근적 성능 분석 방법

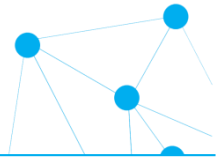
2.3 복잡도 분석 예: 반복 알고리즘

2.4 복잡도 분석 예: 순환 알고리즘

좋은 알고리즘은?

- 시간 효율성(Time efficiency)
- 공간 효율성(Space efficiency)

2.1 효율성 분석의 기초



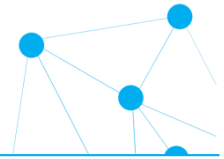
- 실제 실행시간 측정 방법 (파이썬)

```
import time                          # time 모듈 불러오기
...
start = time.time()                  # 현재 시각을 start에 저장(시작 시각)
testAlgorithm(input)                 # 실행시간을 측정하려는 알고리즘 함수 호출
end = time.time()                    # 현재 시각을 end에 저장(종료 시각)
print("실행시간 = ", end-start)       # 실제 실행시간(종료-시작)을 출력
```

  - 문제점은?
    - 반드시 구현?
    - 같은 조건의 실행시간?
    - 소프트웨어 환경, 사용한 언어?
    - 모든 데이터에 대해?

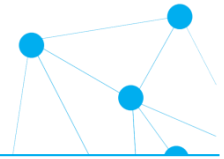
- 절대적인 시간 측정 → 이론적인 복잡도 분석

알고리즘 복잡도 분석에서 중요한 점



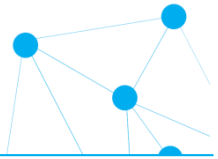
- 알고리즘에서 입력의 크기는 무엇인가?
- 복잡도에 영향을 미치는 가장 핵심적인 기본 연산은 무엇인가?
- 입력의 크기가 증가함에 따라 처리시간은 어떤 형태로 증가하는가?
- 입력의 특성에 따라 알고리즘 효율성에는 어떤 차이가 있는가?

입력의 크기



- 알고리즘의 효율성은 입력 크기의 함수 형태로 표현
- 무엇이 입력의 크기를 나타내는지를 먼저 명확히 결정
- 예
  - 리스트에서 어떤 값을 찾는 문제
  - x의 n 거듭제곱
  - 다항식의 연산
  - Row x Col 의 행렬 연산
  - 그래프 연산 : 인접 행렬 표현 / 인접 리스트 표현

실행시간 측정의 단위(기본 연산)



- **기본 연산(basic operation)**
  - 알고리즘에서 가장 중요한 연산
  - 이 연산이 실행되는 횟수 만을 계산
  - 예) 다중 루프의 경우 가장 안쪽 루프에 있는 연산

3행: + 와 대입 연산 한번씩
2행에 의해 n번 반복(대입n번, 덧셈n번)

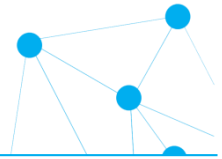
- n의 거듭제곱

대입

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 덧셈과 대입	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: $2n+1$	대입 연산: n^2+n+2 덧셈 연산: n^2 전체 횟수: $2n^2+n+2$

복잡도 함수와 증가 속도



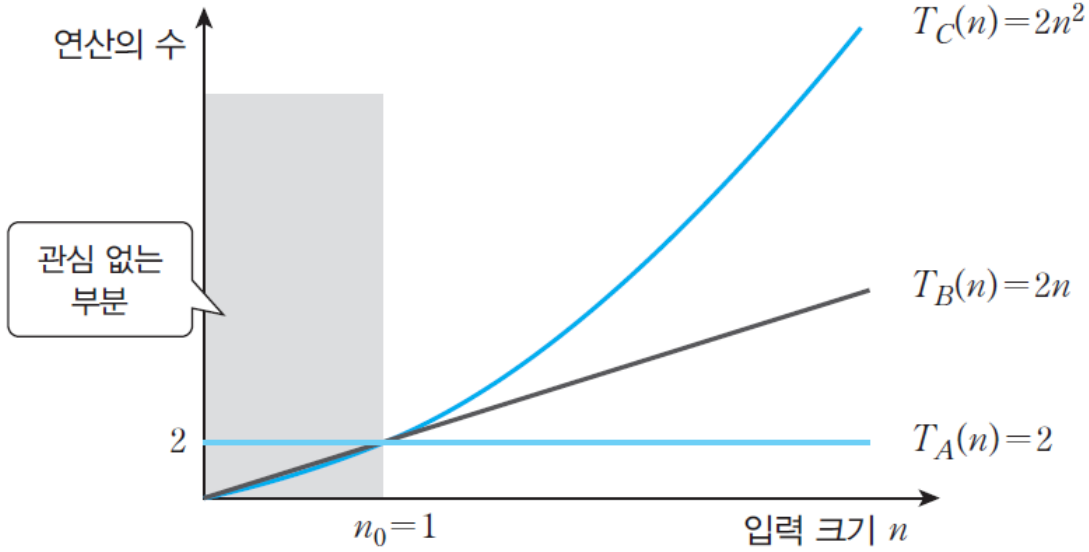
- 복잡도 함수

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

- n이 작은 경우: 예 n=1

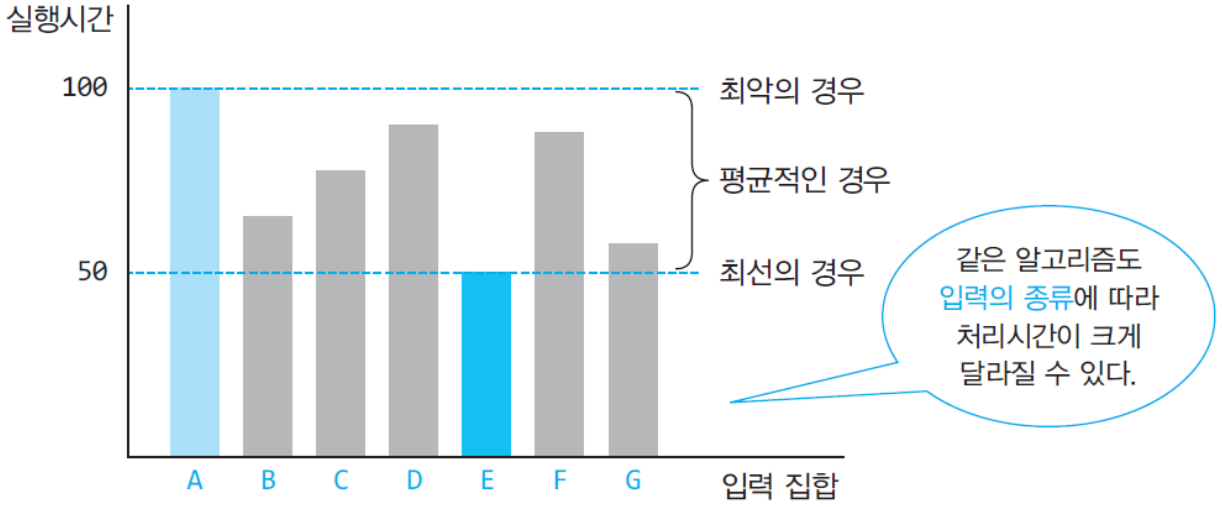
$$T_A(1) = 2,\ T_B(1) = 2,\ T_C(1) = 2$$

- **n이 충분히 큰 경우에만 관심 있음**



최선, 최악, 평균적인 효율성

- 입력의 종류 또는 구성에 따라 다른 특성의 실행시간

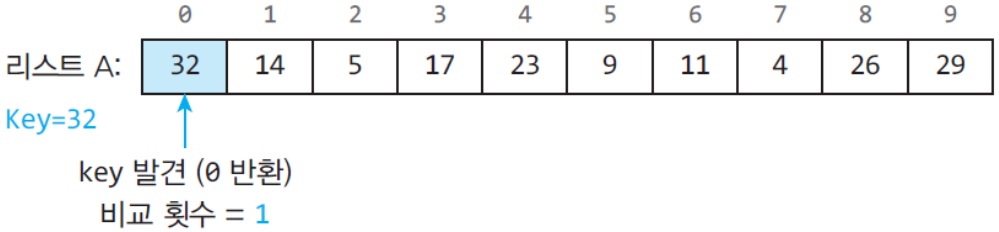


- **최선의 경우**(best case): 실행시간이 가장 적은 경우를 말하는데, 알고리즘 분석에서는 큰 의미가 없다.
- **평균적인 경우**(average case): 알고리즘의 모든 입력을 고려하고 각 입력이 발생할 확률을 고려한 평균적인 실행시간을 의미하는데 정확히 계산하기가 어렵다.
- **최악의 경우**(worst case): 입력의 구성이 알고리즘의 실행시간을 가장 많이 요구하는 경우를 말하는데, 가장 중요하게 사용된다.

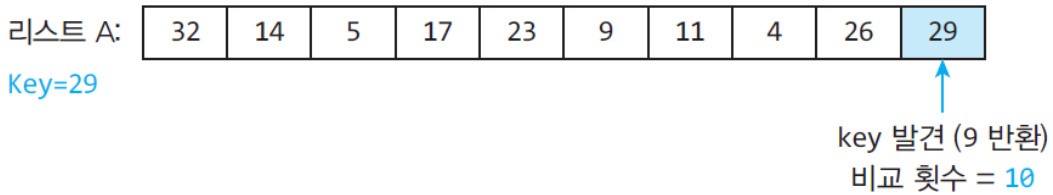
예) 순차 탐색

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

- 최선



- 최악

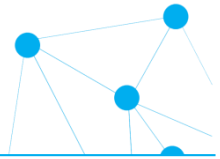


- 평균: 가정이 필요 **리스트내의 모든 숫자가 균일하게 탐색된다고 가정**

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

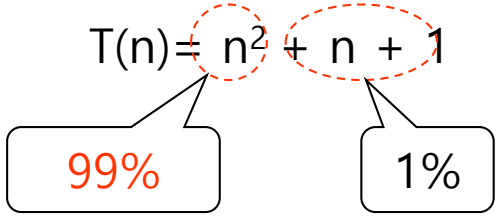
자율주행 자동차, 항공 관제 업무에 사용되는 알고리즘은
아무리 불리한 입력이 입력되도 일정한 시간 안에 반드시 계산을 마쳐야 중대한 사고 X

2.2 점근적 성능 분석 방법

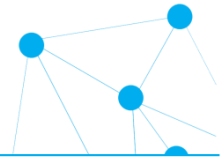


- 차수가 가장 큰 항이 절대적인 영향
  - 예: $T(n) = n^2 + n + 1$
    - n=1일때 : $T(n) = 1 + 1 + 1 = 3$ (n^2 항이 33.3%)
    - n=10일때 : $T(n) = 100 + 10 + 1 = 111$ (n^2 항이 90%)
    - n=100일때 : $T(n) = 10000 + 100 + 1 = 10101$ (n^2 항이 99%)
    - n=1,000일때 : $T(n) = 1000000 + 1000 + 1 = 1001001$ (n^2 항이 99.9%)

n=100인 경우



점근적 표기(asymptotic notation)



- n 이 무한대로 커질 때의 복잡도를 간단히 표현
  - 복잡도 함수를 최고차항만을 계수 없이 취해 단순화 함

- 빅오 : 복잡도 함수의 상한

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

- 빅오메가 : 하한

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

- 빅세타 : 상한인 동시에 하한

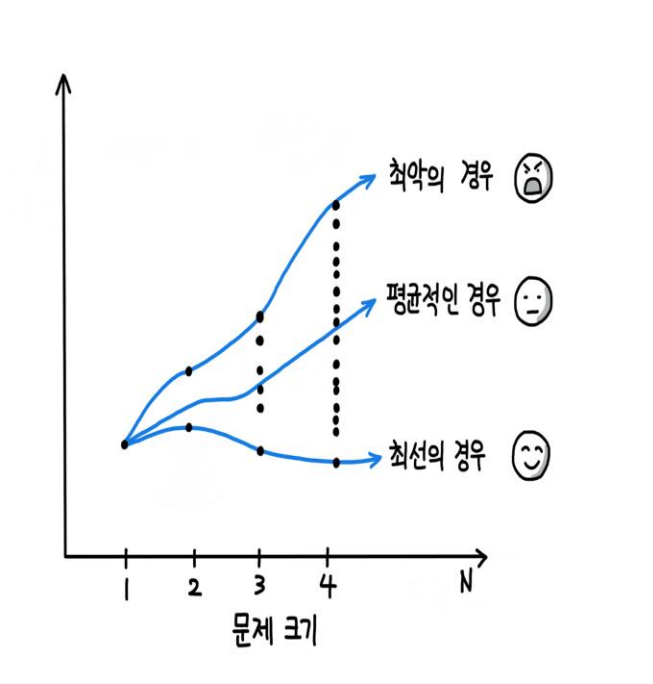
$$2n^3+3n \in \Theta(n^3), \quad 2n^3+3n \notin \Theta(n^2), \quad 2n^3+3n \notin \Theta(n^4)$$

시간 복잡도와 공간 복잡도의 트레이드오프

Ω(빅오메가): 최선의 경우, best case

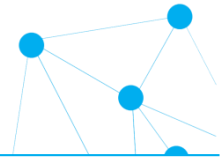
Θ(빅세타): 평균적인 경우, average case

O(빅오): 최악의 경우, worst case



병렬화로 실행속도 ⬆

다단계 알고리즘의 복잡도



만약 $f_1(n) \in O(g_1(n))$ 이고 $f_2(n) \in O(g_2(n))$ 이면 다음이 성립한다.

$$f_1(n)+f_2(n) \in O(\max\{g_1(n),\ g_2(n)\})$$

이것은 Ω – 표기와 Θ – 표기에서도 동일하게 적용된다.

- 증명? 심화학습

- 예) 리스트의 중복 항목 검사

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

중복된 항목 있음(14)

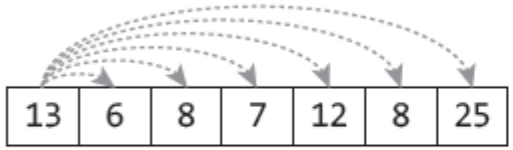
	0	1	2	3	4	5	6	7
리스트 B	22	16	14	12	7	13	27	17

중복된 항목 없음

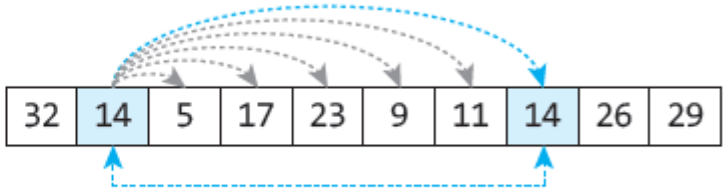
알고리즘 A : 리스트의 각 항목을 다른 모든 항목과 비교하여 같은 항목이 있으면 True 모두 다르면 False

- 알고리즘 A: 이중 루프 사용 O(n^2)
- 알고리즘 B: 2-단계 알고리즘→ 정렬 + 단일 루프

리스트의 중복 항목 탐색



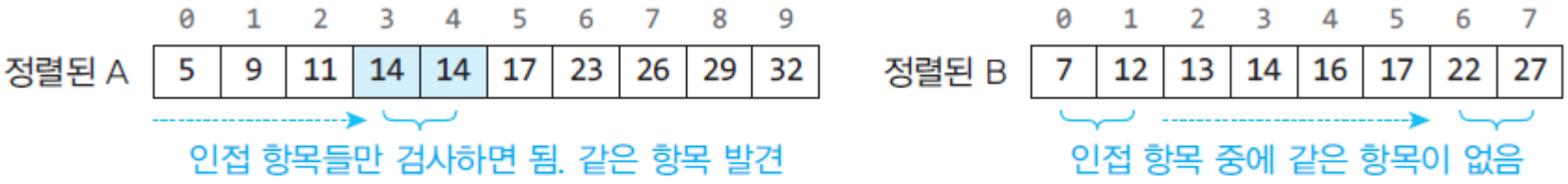
중복된 항목 없음




```
01  def unique_elements(A) :          # 리스트 A 입력
02      n = len(A)                    # 입력의 크기 = 리스트의 크기
03      for i in range(n-1) :         # i : 0, 1, ... n-2
04          for j in range(i+1,n) :   # j : i+1, i+2, ... n-1
05              if A[i] == A[j] :     # 기본 연산
06                  return False      # 같은 항목이 있으면 False 반환
07      return True                   # 같은 항목이 없으면 True 반환
```

- 입력의 크기?
- 기본 연산?
- 최선/최악/평균의 복잡도 ?

- 2단계 알고리즘
- 알고리즘 B: 2-단계 알고리즘→ 정렬 + 단일 루프



[그림 2.8] 2-단계 알고리즘: 리스트를 먼저 정렬하고, 인접 항목들만을 비교

- 복잡도 계산
  - 1단계: O(nlogn) 정렬(효율적인 정렬알고리즘 사용)
  - 2단계: O(n)
  - 전체: $O(\max\{n\log_2 n, n\}) = O(n\log_2 n)$

점근적 성능 클래스들

$O(1)$: 상수형

$O(\log n)$: 로그형

$O(n)$: 선형

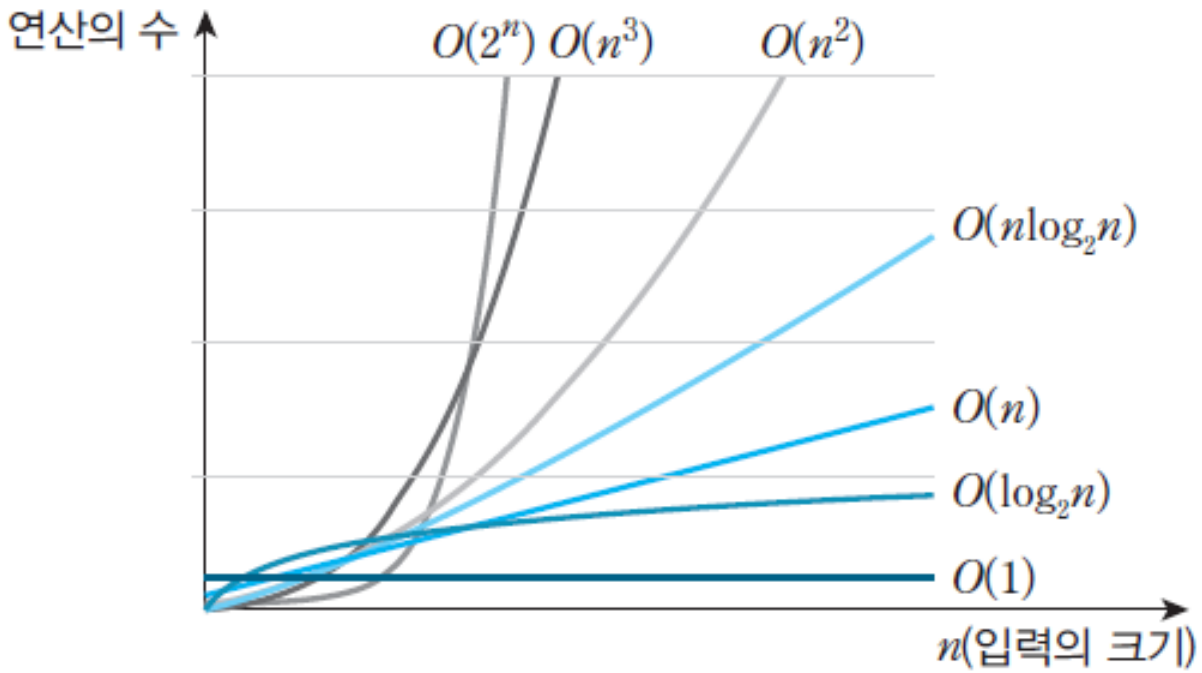
$O(n\log n)$: 선형로그형

$O(n^2)$: 2차형

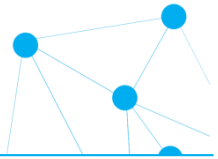
$O(n^3)$: 3차형

$O(2^n)$: 지수형

$O(n!)$: 팩토리얼형

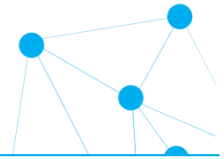


2.3 복잡도 분석 예: 반복 알고리즘



(1) 입력의 크기를 나타내는 파라미터를 결정한다.
(2) 기본 연산을 찾는다. 보통 반복 루프의 가장 안쪽에 있다.
(3) 연산의 횟수가 입력 크기에 의해서만 결정되는지 살핀다. 만약 입력의 종류에 따라서도 다를 수 있다면 최선, 최악, 평균의 경우에 대해 독립적으로 복잡도를 분석해야 한다.
(4) 기본 연산의 전체 실행 횟수를 구하는 복잡도 함수 $T(n)$을 구한다.
(5) 알려진 공식 등을 이용해 $T(n)$을 풀고, 증가 속도를 계산한다.

자연수의 제곱 계산



```
01  def compute_square_A(n) :
02      return n*n
```

O(1)

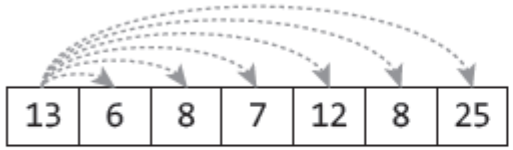
```
01  def compute_square_B(n) :
02      sum = 0
03      for i in range(n) :
04          sum = sum + n
05      return sum
```

O(n)

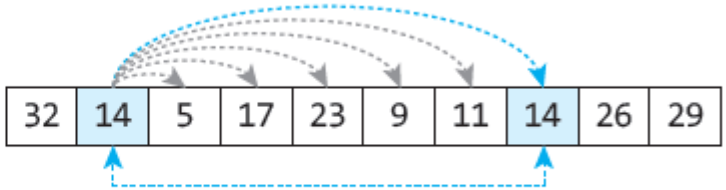
```
01  def compute_square_C(n) :
02      sum = 0
03      for i in range(n) :
04          for j in range(n) :
05              sum = sum + 1
06      return sum
```

O(n^2)

리스트의 중복 항목 탐색



중복된 항목 없음




```
01  def unique_elements(A) :        # 리스트 A 입력
02      n = len(A)                   # 입력의 크기 = 리스트의 크기
03      for i in range(n-1) :        # i : 0, 1, ... n-2
04          for j in range(i+1,n) :  # j : i+1, i+2, ... n-1
05              if A[i] == A[j] :    # 기본 연산
06                  return False     # 같은 항목이 있으면 False 반환
07      return True                  # 같은 항목이 없으면 True 반환
```

- 입력의 크기? 전체 리스트 항목수
- 기본 연산? 5행 A[i] == A[j]
- 최선/최악/평균의 복잡도 ?
  - 최선:A[0] == A[1] 같다면 한번만 처리되고 종료 O(1)
  - 최악: O(n^2)
  - 만약 2-단계 알고리즘(정렬 + 단일 루프) → $O(n \log_2 n)$

- 최악의 경우?

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{n(n-1)}{2} \in O(n^2)$$

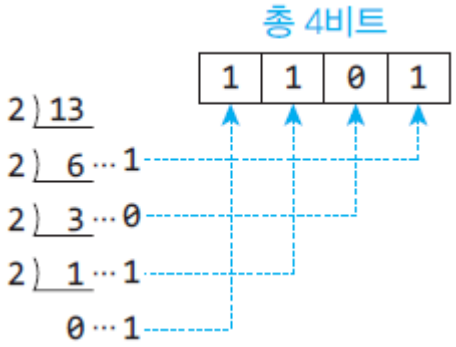
잠깐만!! 유용한 합 공식들

$$\sum_{i=1}^{n} 1 = n$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} (k+i) = k \sum_{i=1}^{n} 1 + \sum_{i=1}^{n} i$$

자연수의 2진수 변환시 비트 수 (반복 구조)




```
01  def binary_digits(n) :
02      count = 1
03      while n > 1 :
04          count = count + 1
05          n = n // 2
06      return count
```

- 입력의 크기? n
- 기본 연산? n // 2
- 최선/최악/평균의 복잡도 ? → 동일
- 복잡도: $O(\log_2 n)$ n = 2^k

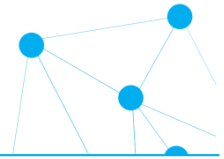
$$2^k \to 2^{k-1} \to 2^{k-2} \to \cdots \to 2^1 \to 2^0$$

총 3번 나누기 6(13/2) → 3(6/2) → 1(3/2)

2.4 복잡도 분석 예: 순환 알고리즘

(1) 입력의 크기를 나타내는 파라미터를 결정한다.
(2) 기본 연산을 찾는다.
(3) 연산의 횟수가 입력 크기에 의해서만 결정되는지 살핀다.
(4) 기본 연산의 실행 횟수를 구하기 위한 순환 관계식(recurrence relation) $T(n)$을 구한다. 이때, 초기 조건도 찾아야 한다.
(5) 순환 관계식(점화식)을 풀거나 증가 속도(order of growth)를 계산한다.

시간 복잡도 분석: 순환 알고리즘



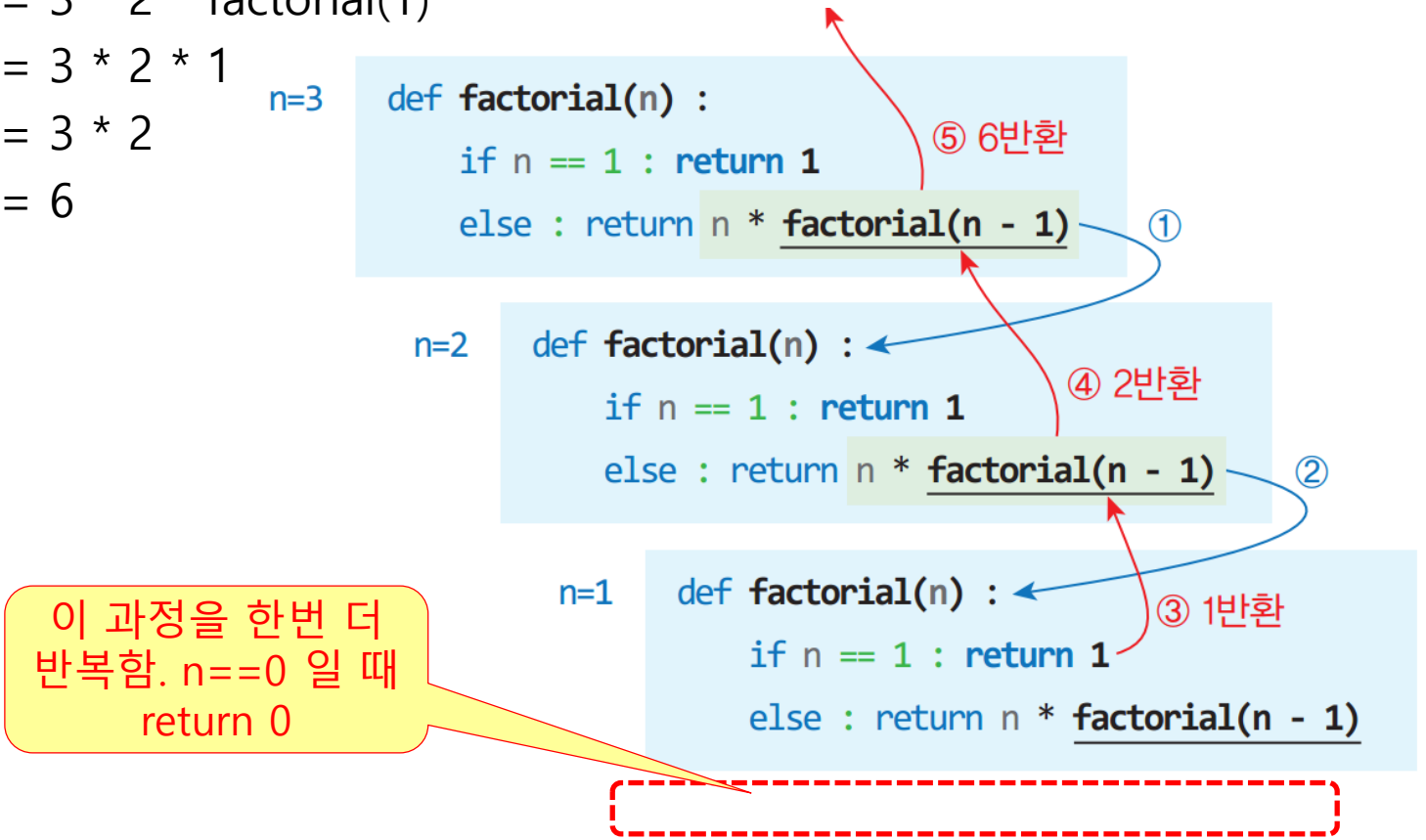
- 순환 알고리즘
  - 알고리즘이나 함수가 수행 도중에 자기 자신을 다시 호출하여 문제를 해결하는 기법
  - 정의자체가 순환적으로 되어 있는 경우에 적합
  - 팩토리얼 구하기 $n! = \begin{cases} 1 & n=1 \\ n*(n-1)! & n>1 \end{cases}$
  - 피보나치 수열 $fib(n) = \begin{cases} 0 & if \quad n=0 \\ 1 & if \quad n=1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$
  - 이항 계수, 하노이의 탑, 이진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

팩토리얼 구하기

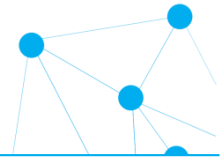
- 순환적인 함수 호출 순서

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

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



팩토리얼: 순환과 반복



- n의 팩토리얼 구하기

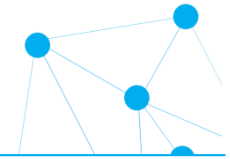
순환 구조		반복 구조
$n!=n*(n-1)!$	↔	$n!=n*(n-1)*(n-2)*\cdots*1$

수행시간과 기억공간의 비효율

- 순환(recursion): $O(n)$
  - 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

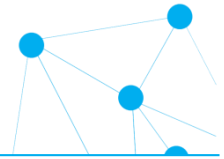


- 방법 1: 반복 구조

```
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

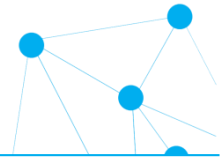
```
power(x, n)

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);
```

$$\begin{aligned} \text{power}(x, n) &= \text{power}(x^2, n / 2) \\ &= (x^2)^{n/2} \\ &= x^{2(n/2)} \\ &= x^n \end{aligned}$$

$$\begin{aligned} \text{power}(x, n) &= x \cdot \text{power}(x^2, (n-1) / 2) \\ &= x \cdot (x^2)^{(n-1)/2} \\ &= x \cdot x^{n-1} \\ &= x^n \end{aligned}$$

```
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.

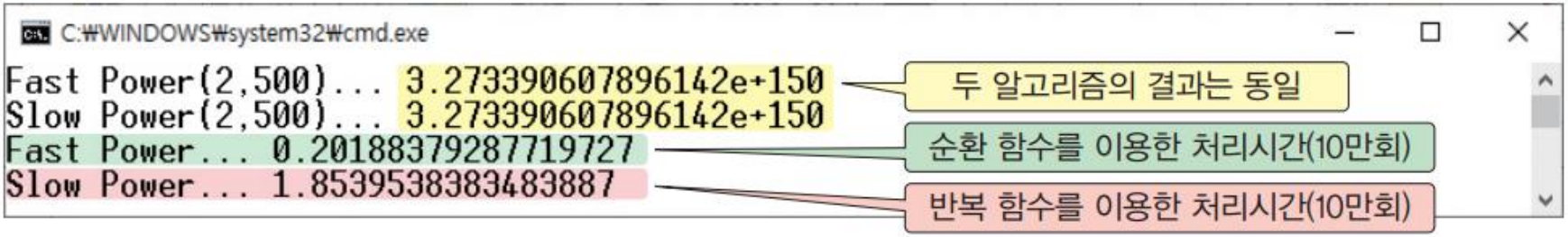
$$2^n \rightarrow 2^{n-1} \rightarrow \cdots 2^2 \rightarrow 2^1 \rightarrow 2^0$$

- 시간 복잡도
  - 순환적인 함수: $O(log_2 n)$
  - 반복적인 함수: $O(n)$

n을 2의 거듭제곱인 $2^k \rightarrow 2^{k-1} \rightarrow 2^{k-2} \rightarrow 2^0$

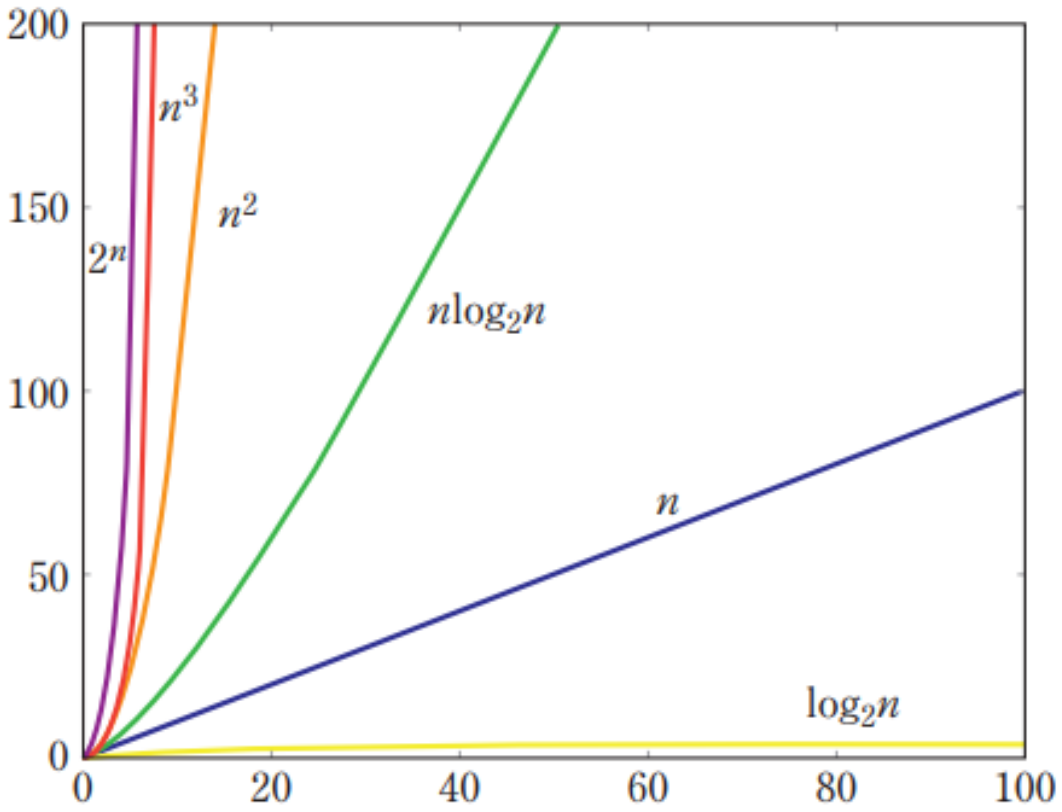
$n = 2^k$ 양변에 log취하면 $log 2^n$=k

1번의 순환 호출이 일어날때마다
1번의 곱셈과 1번의 나눗셈이 일어나므로
전체 연산 개수는 $log 2^n$=k에 비례

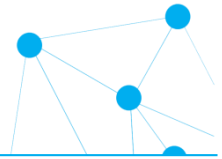


빅오 표기법의 종류

$O(1)$: 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



순환이 느린 예: 피보나치 수열



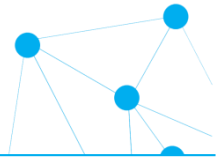
- 순환 호출을 사용하면 비효율적인 예
- 피보나치 수열: 0,1,1,2,3,5,8,13,21,...

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

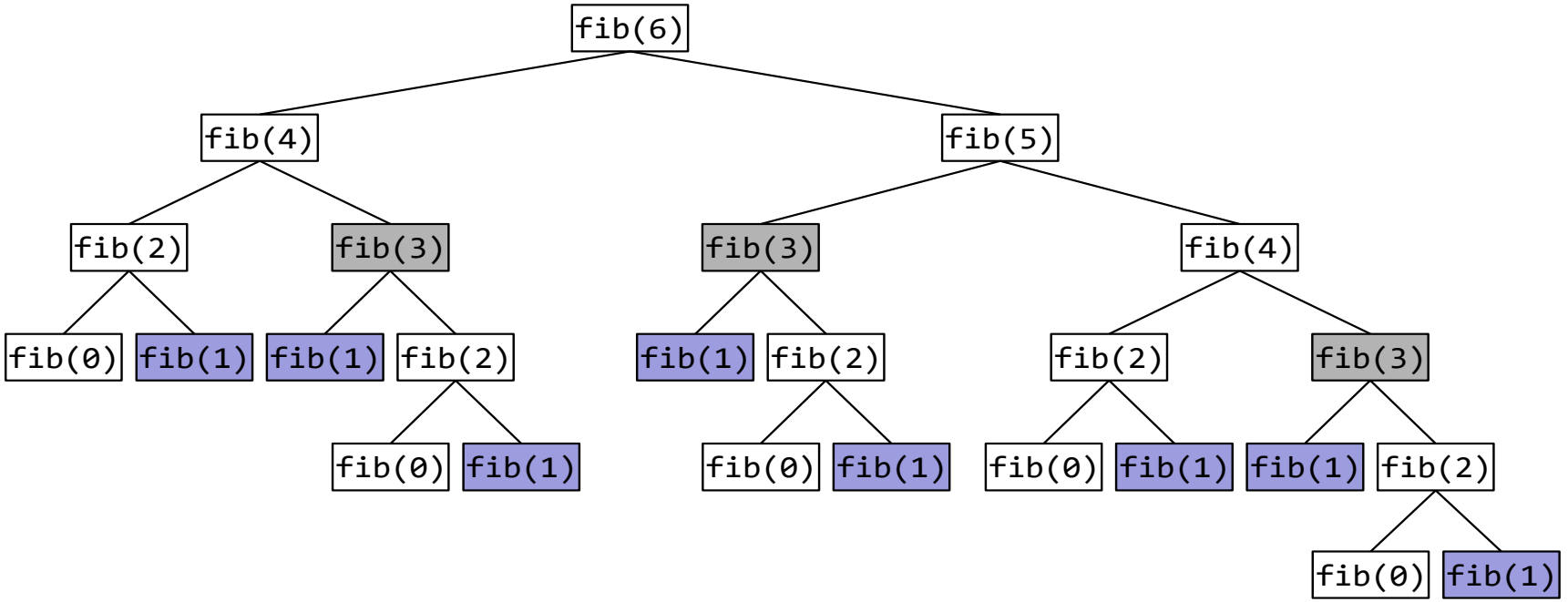
- 순환적인 구현

```
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

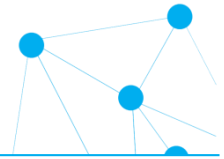


- 같은 항이 중복해서 계산됨!
  - n이 커지면 더욱 심각



- 시간 복잡도: $O(2^n)$

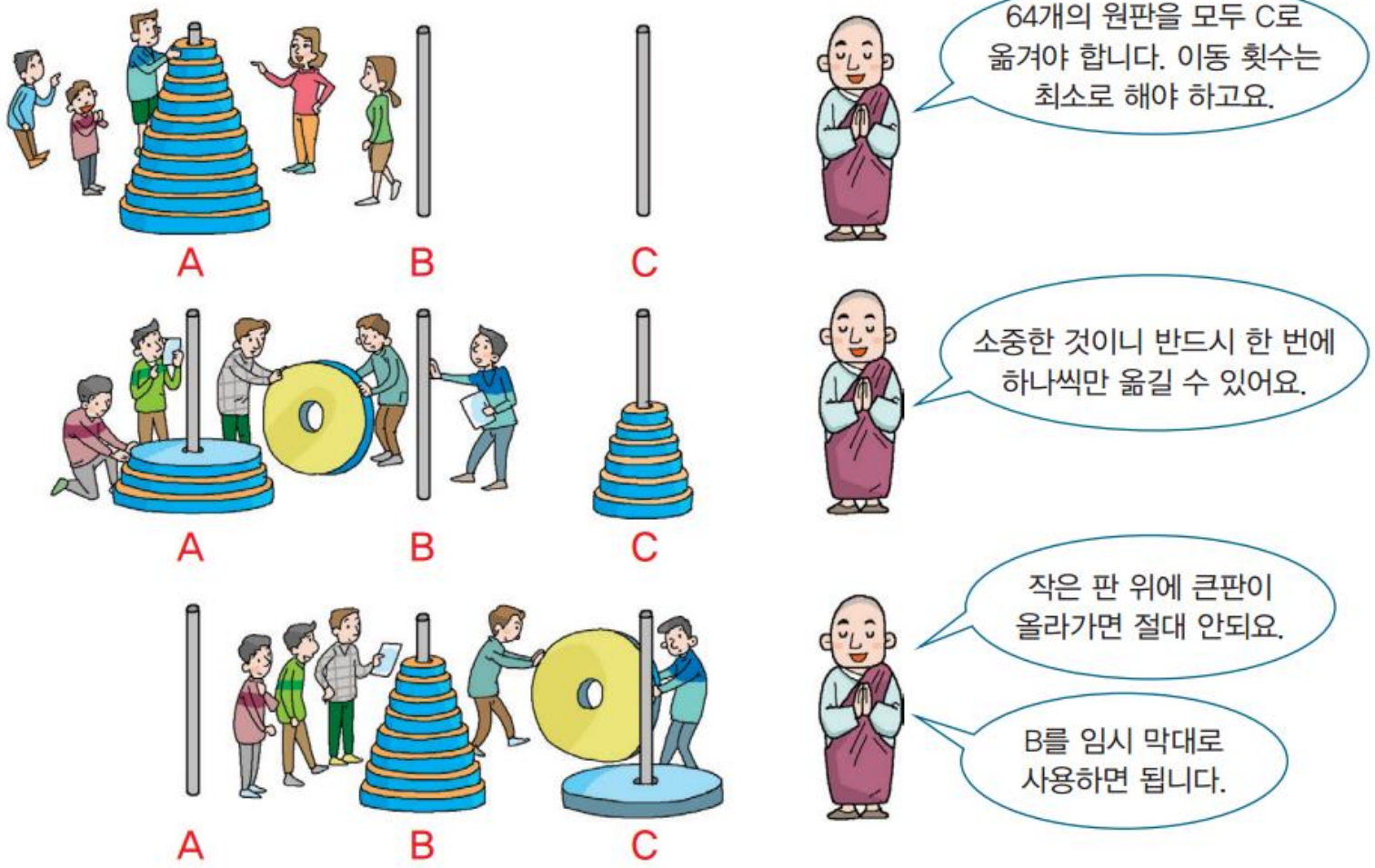
반복적인 피보나치 수열



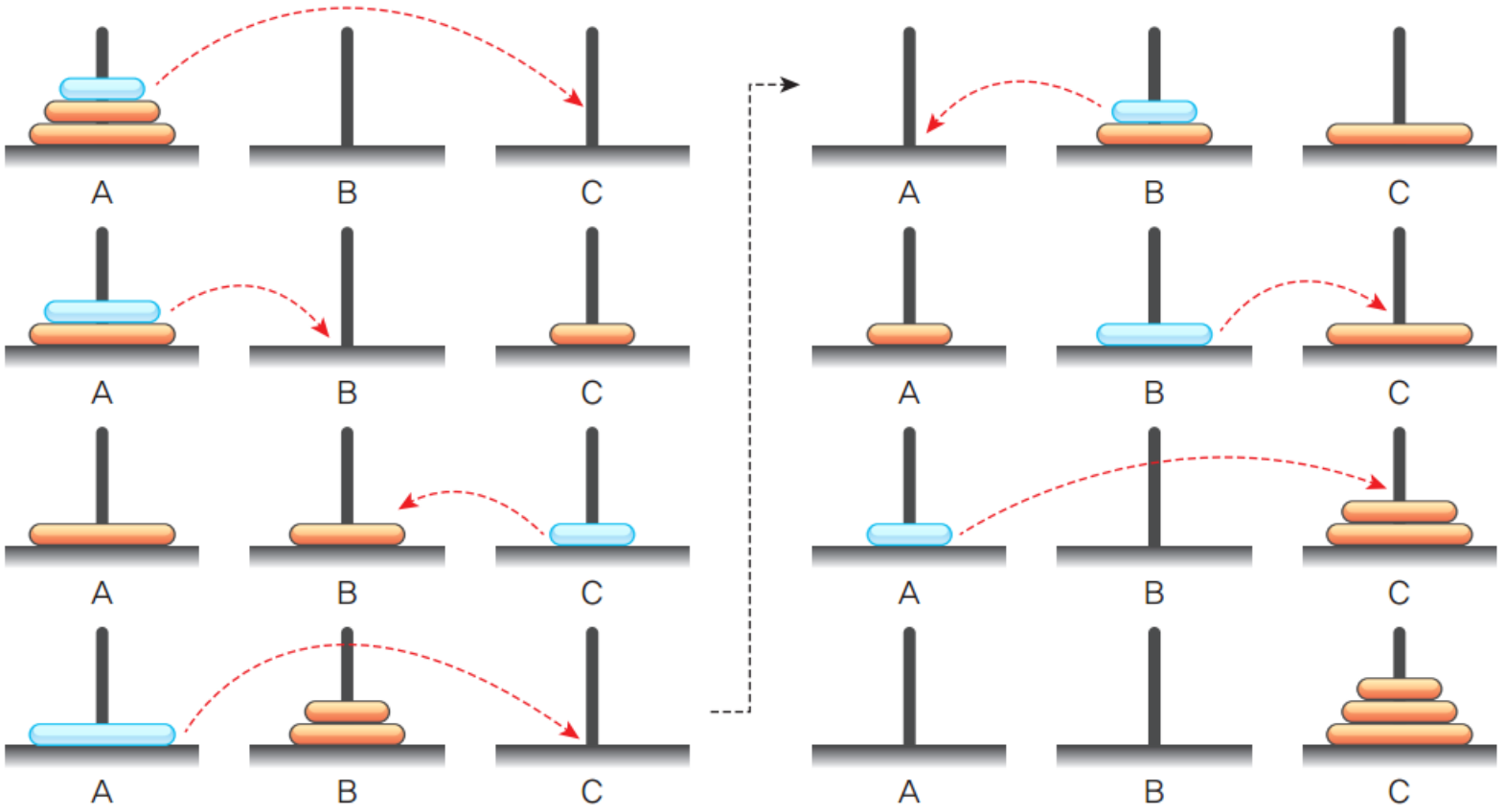
```
def fib_iter(n) :                    # 반복으로 구현한 피보나치 수열
    if (n < 2): return n

    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

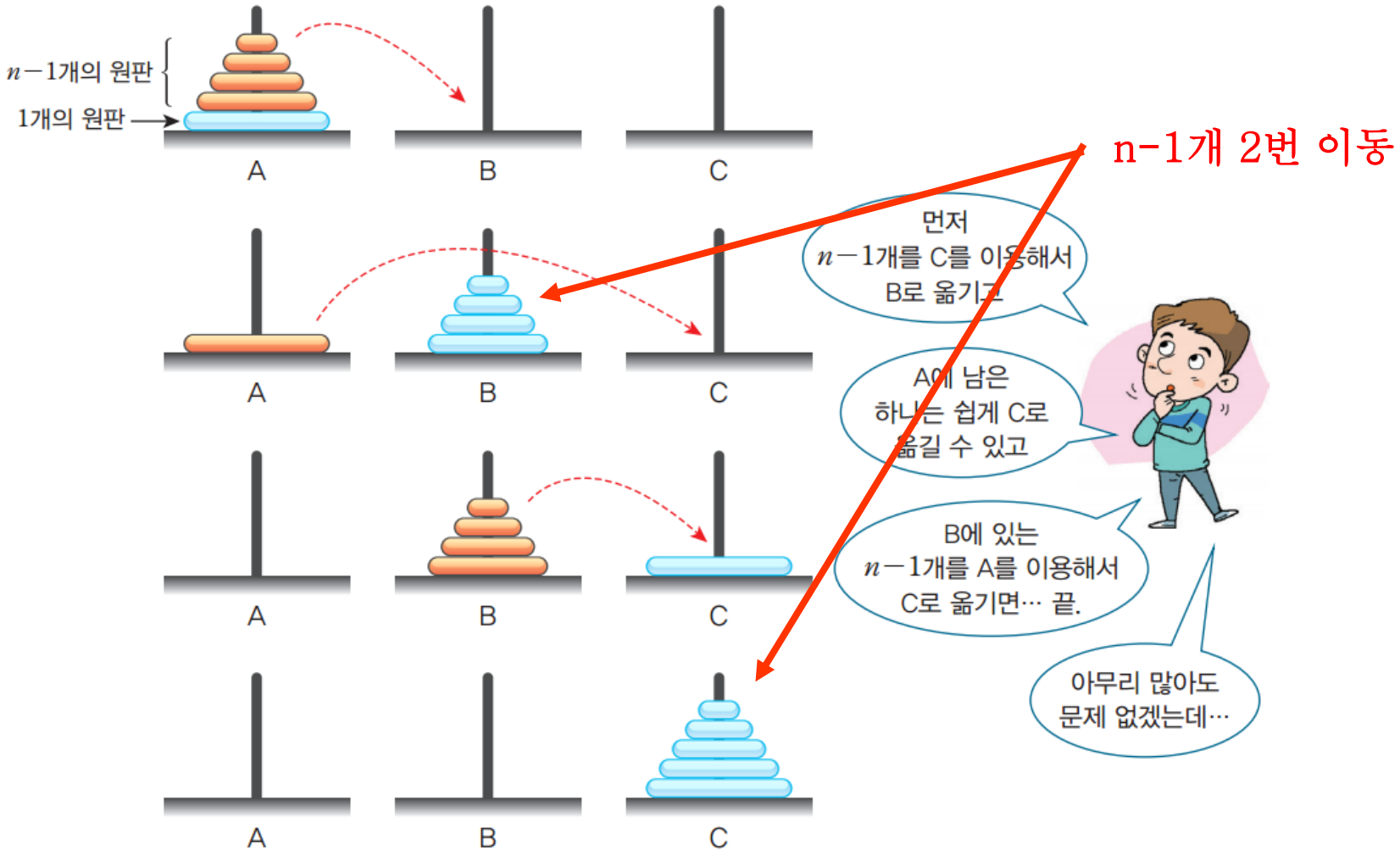

A
B
C
A
B
C
A
B
C
64개의 원판을 모두 C로 옮겨야 합니다. 이동 횟수는 최소로 해야 하고요.
소중한 것이니 반드시 한 번에 하나씩만 옮길 수 있어요.
작은 판 위에 큰판이 올라가면 절대 안되요.
B를 임시 막대로 사용하면 됩니다.

n=3인 경우의 해답

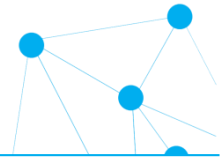


https://www.mathplayground.com/logic_tower_of_hanoi.html

일반적인 경우에는?


n−1개의 원판
1개의 원판
A
B
C
n-1개 2번 이동
먼저
n−1개를 C를 이용해서
B로 옮기고
A에 남은
하나는 쉽게 C로
옮길 수 있고
B에 있는
n−1개를 A를 이용해서
C로 옮기면… 끝.
아무리 많아도
문제 없겠는데…

구현

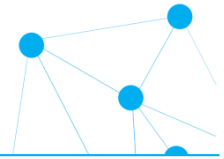


- 어떻게 n-1개의 원판을 A에서 B로, 또 B에서 C로 이동하는가?
  - **순환을 이용**

```
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.)

n-1개 2번 이동

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

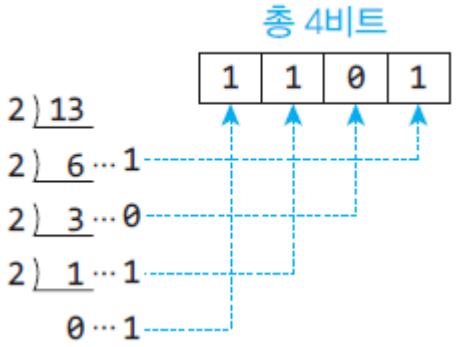
- 복잡도 함수의 순환 관계식
  - *T(n) = T(n-1) + 1 + T(n-1)*
  - *T(1) = 1*

  - 연속 대치법에 의한 풀이

$$T(n) = 2T(n-1)+1$$
$$= 2[2T(n-2)+1]+1 = 2^2 T(n-2)+2+1$$
$$= 2^n - 1$$

  - 복잡도: $O(2^n)$

자연수의 2진수 변환시 비트 수 (순환 구조)




```
01  def binary_digits(n) :
02      if n <= 1 :
03          return 1
04      else :
05          return 1 + binary_digits(n//2)
```

- 복잡도 순환 관계식

$$T(n) = T(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1$$

$$\begin{aligned} T(2^k) &= T(2^{k-1}) + 1 \\ &= [T(2^{k-2}) + 1] + 1 = T(2^{k-2}) + 2 \\ &= [T(2^{k-3}) + 1] + 1 = T(2^{k-3}) + 3 \\ &\cdots \\ &= T(2^{k-k}) + k \\ &= T(2^0) + k \\ &= T(1) + k \\ &= k \end{aligned}$$

$$n = 2^k$$

$$O(\log_2 n)$$