

관계 중심의 사고법

# 쉽게 배우는 알고리즘

## 2장. 알고리즘 설계와 분석의 기초

## 2장. 알고리즘 설계와 분석의 기초

전혀 새로운 아이디어를 갑자기  
착상하는 일이 자주 있다.  
하지만 그것을 착상하기까지 오랫동안  
끊임없이 문제를 생각한다.  
오랫동안 생각한 끝에 갑자기  
답을 착상하게 되는 것이다.

– 라이너스 폴링

# 알고리즘이란 무엇인가?

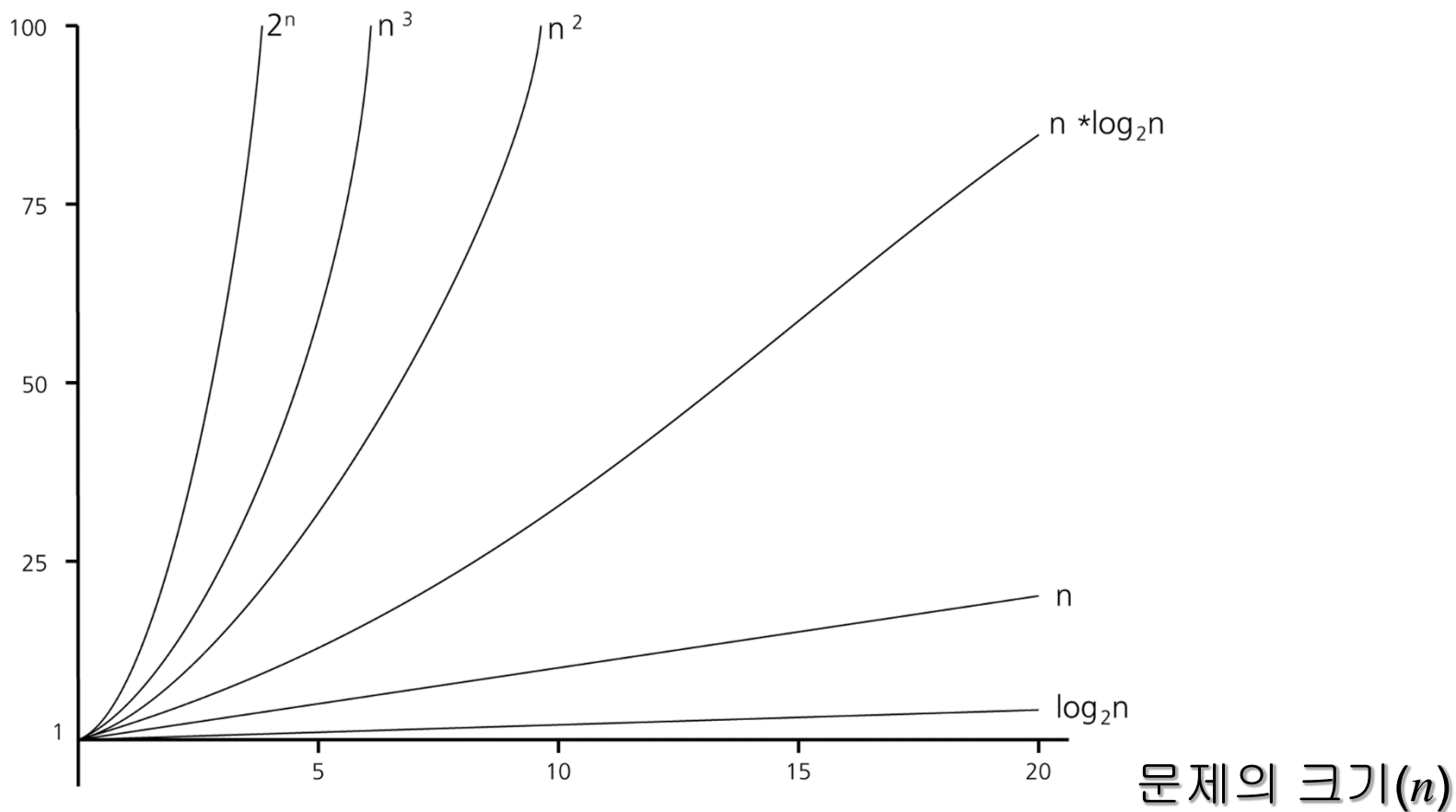
- 문제 해결 절차를 체계적으로 기술한 것
- 문제의 요구조건
  - 입력과 출력으로 명시할 수 있다
  - 알고리즘은 입력으로부터 출력을 만드는 과정을 기술

# 바람직한 알고리즘

- 명확해야 한다
  - 이해하기 쉽고 가능하면 간명하도록
  - 지나친 기호적 표현은 오히려 명확성을 떨어뜨림
  - 명확성을 해치지 않으면 일반언어의 사용도 무방
- 효율적이어야 한다
  - 같은 문제를 해결하는 알고리즘들의 수행 시간이 수백만 배 이상 차이날 수 있다

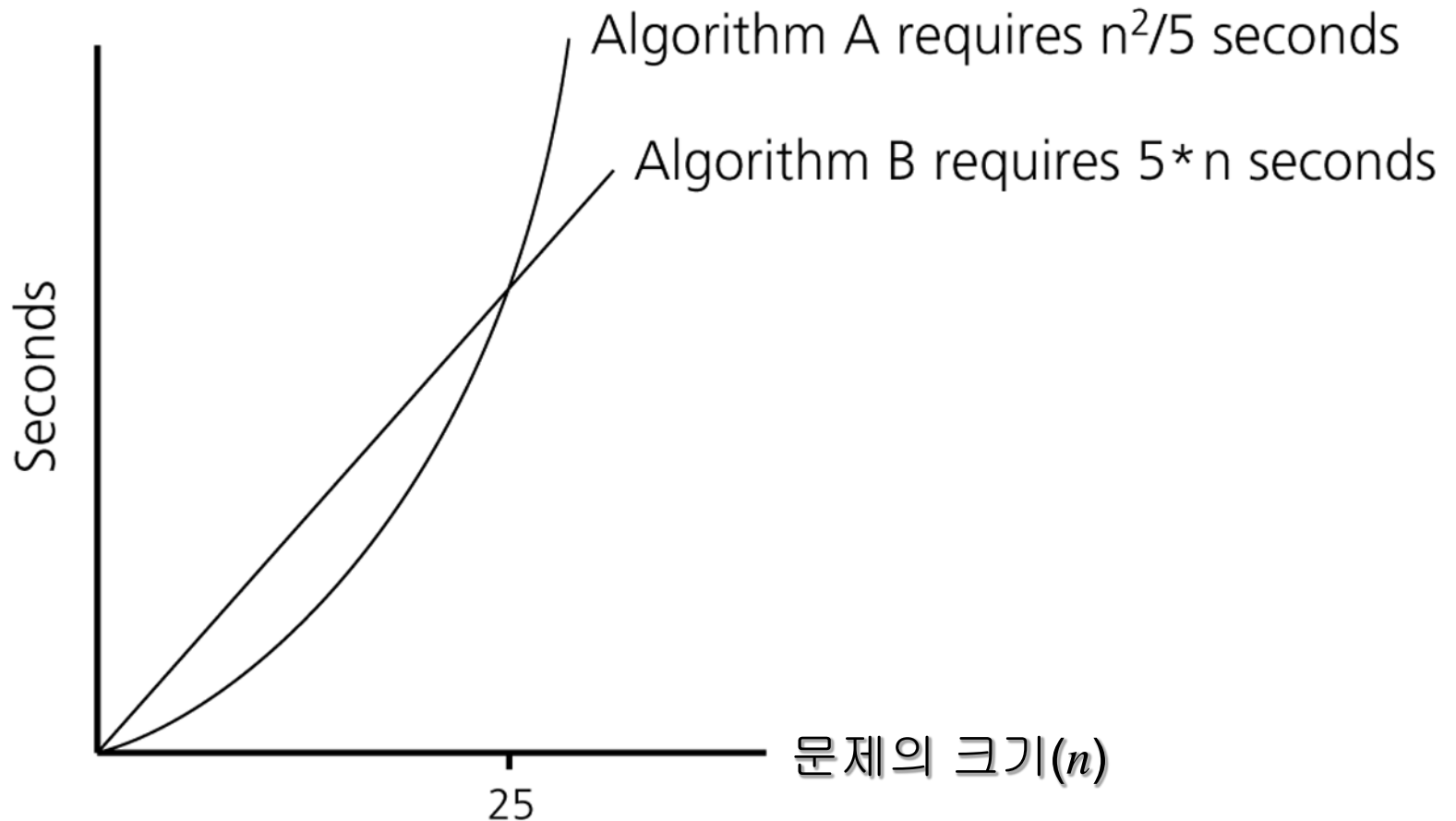
# 알고리즘의 수행 시간

수행 시간



# 알고리즘의 수행 시간

수행 시간



# 알고리즘의 수행 시간

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
$n$	10	$10^2$	$10^3$	$10^4$	$10^5$	$10^6$
$n * \log_2 n$	30	664	9,965	$10^5$	$10^6$	$10^7$
$n^2$	$10^2$	$10^4$	$10^6$	$10^8$	$10^{10}$	$10^{12}$
$n^3$	$10^3$	$10^6$	$10^9$	$10^{12}$	$10^{15}$	$10^{18}$
$2^n$	$10^3$	$10^{30}$	$10^{301}$	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

# 알고리즘의 수행 시간

- 알고리즘의 수행 시간을 좌우하는 기준은 다양하게 잡을 수 있다
  - 예: for 루프의 반복횟수, 특정한 행이 수행되는 횟수, 함수의 호출횟수, ...
- 몇 가지 간단한 경우의 예를 통해 알고리즘의 수행 시간을 살펴본다



## 알고리즘의 수행 시간

```
sample1(A[ ], n)
{
    k =  $\lfloor n/2 \rfloor$ ;
    return A[k];
}
```

✓  $n$ 에 관계없이 상수 시간이 소요된다.

# 알고리즘의 수행 시간

```
sample2(A[ ], n)
{
    sum ← 0 ;
    for  $i \leftarrow 1$  to  $n$ 
        sum ← sum + A[i] ;
    return sum ;
}
```

✓  $n$ 에 비례하는 시간이 소요된다.

# 알고리즘의 수행 시간

```
sample3(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        for j ← 1 to n
            sum ← sum + A[i]*A[j] ;
    return sum ;
}
```

✓  $n^2$ 에 비례하는 시간이 소요된다.

# 알고리즘의 수행 시간

```
sample4(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n
        for j ← 1 to n {
            k ← A[1 ... n]에서 임의로  $\lfloor n/2 \rfloor$ 개를 뽑을 때 이들 중 최댓값 ;
            sum ← sum + k ;
        }
    return sum ;
}
```

$n^{3/2}$ 의 시간이므로 ->

✓  $n^3$ 에 비례하는 시간이 소요된다.

# 알고리즘의 수행 시간

```
sample5(A[ ], n)
{
    sum ← 0 ;
    for i ← 1 to n-1          n*2/2
        for j ← i+1 to n
            sum ← sum + A[i]*A[j] ;
    return sum ;
}
```

✓  $n^2$ 에 비례하는 시간이 소요된다.

# 알고리즘의 수행 시간

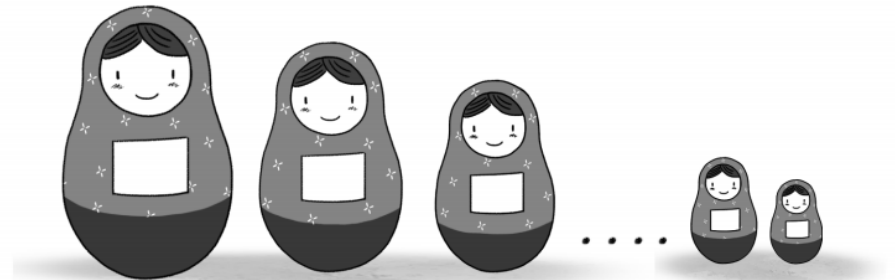
```
factorial( $n$ )  
{  
    if ( $n=1$ ) return 1 ;  
    return  $n$ *factorial( $n-1$ ) ;  
}
```

필기 참조,

✓  $n$ 에 비례하는 시간이 소요된다.

# 재귀와 귀납적 사고

- 재귀=자기호출(recursion)
- 재귀적 구조
  - 어떤 문제 안에 크기만 다를 뿐 성격이 똑같은 작은 문제(들)가 포함되어 있는 것
  - 예1: factorial
    - $N! = N \times (N-1)!$
  - 예2: 수열의 점화식
    - $a_n = a_{n-1} + 2$



## 재귀의 예: 병합 정렬

`mergeSort(A[ ], p, r)`    ▷  $A[p \dots r]$ 을 정렬한다.

```
{  
  if ( $p < r$ ) then {  
     $q \leftarrow \lfloor (p + r) / 2 \rfloor$ ; ----- ①    ▷  $p, r$ 의 중간 지점 계산  
    mergeSort(A, p, q); ----- ②    ▷ 전반부 정렬  
    mergeSort(A, q+1, r); ----- ③    ▷ 후반부 정렬  
    merge(A, p, q, r); ----- ④    ▷ 병합  
  }  
}
```

`merge(A[ ], p, q, r)`  
{  
 정렬되어 있는 두 배열  $A[p \dots q]$ 와  $A[q+1 \dots r]$ 을 합쳐  
 정렬된 하나의 배열  $A[p \dots r]$ 을 만든다.  
}



mergeSort(A[ ],  $p, r$ )    ▷  $A[p \dots r]$ 을 정렬한다.

```
{
  if ( $p < r$ ) then {
     $q \leftarrow \lfloor (p + r) / 2 \rfloor$ ; ----- ①    ▷  $p, r$ 의 중간 지점 계산
    mergeSort(A,  $p, q$ ); ----- ②    ▷ 전반부 정렬
    mergeSort(A,  $q+1, r$ ); ----- ③    ▷ 후반부 정렬
    merge(A,  $p, q, r$ ); ----- ④    ▷ 병합
  }
}
```

✓ ②, ③은 재귀호출

✓ ①, ④는 재귀적 관계를 드러내기 위한 오버헤드

# 다양한 알고리즘의 적용 주제들

- 카 네비게이션
- 스케줄링
  - TSP, 차량 라우팅, 작업공정, ...
- Human Genome Project
  - 매칭, 계통도, functional analyses, ...
- 검색
  - 데이터베이스, 웹페이지들, ...
- 자원의 배치
- 반도체 설계
  - Partitioning, placement, routing, ...
- ...

# 알고리즘을 왜 분석하는가

- 무결성 확인
- 자원 사용의 효율성 파악
  - 자원
    - 시간
    - 메모리, 통신대역, ...

# 알고리즘의 분석

- 크기가 작은 문제
  - 알고리즘의 효율성이 중요하지 않다
  - 비효율적인 알고리즘도 무방
- 크기가 충분히 큰 문제
  - 알고리즘의 효율성이 중요하다
  - 비효율적인 알고리즘은 치명적
- 입력의 크기가 충분히 큰 경우에 대한 분석을  
점근적 분석이라 한다

# 점근적 분석 Asymptotic Analysis

- 입력의 크기가 충분히 큰 경우에 대한 분석
- 이미 알고있는 점근적 개념의 예

$$\lim_{n \rightarrow \infty} f(n)$$

- $O$ ,  $\Omega$ ,  $\Theta$ ,  $\omega$ ,  $o$  표기법

# 점근법 표기법 Asymptotic Notations

$O(g(n))$

- 기껏해야  $g(n)$ 의 비율로 증가하는 함수
- e.g.,  $O(n)$ ,  $O(n \log n)$ ,  $O(n^2)$ ,  $O(2^n)$ , ...

- Formal definition

- $O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, cg(n) \geq f(n) \}$
- $f(n) \in O(g(n))$ 을 관행적으로  $f(n) = O(g(n))$ 이라고 쓴다.

- 직관적 의미

- $f(n) = O(g(n)) \Rightarrow f$ 는  $g$ 보다 빠르게 증가하지 않는다
- 상수 비율의 차이는 무시

## 점근적 표기법

- 예,  $O(n^2)$ 
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $n \log n + 5n$
  - $3n$
- 알 수 있는 한 최대한 tight 하게
  - $n \log n + 5n = O(n \log n)$  인데 굳이  $O(n^2)$ 으로 쓸 필요없다
  - 엄밀하지 않은 만큼 정보의 손실이 일어난다

$\Omega(g(n))$

- 적어도  $g(n)$ 의 비율로 증가하는 함수
- $O(g(n))$ 과 대칭적

- Formal definition

- $\Omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, cg(n) \leq f(n) \}$

- 직관적 의미

- $f(n) = \Omega(g(n)) \Rightarrow f$ 는  $g$ 보다 느리게 증가하지 않는다



$\Theta( g(n) )$

–  $g(n)$ 의 비율로 증가하는 함수

- Formal definition

–  $\Theta( g(n) ) = O( g(n) ) \cap \Omega( g(n) )$

- 직관적 의미

–  $f(n) = \Theta(g(n)) \Rightarrow f$ 는  $g$ 와 같은 정도로 증가한다

# 점근적 표기법

$o(g(n))$

–  $g(n)$ 보다 느린 비율로 증가하는 함수

- Formal definition

- $o(g(n)) = \{f(n) \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0\}$

- 직관적 의미

- $f(n) = o(g(n)) \Rightarrow f$ 는  $g$ 보다 느리게 증가한다

# 점근적 표기법

$\omega( g(n) )$

–  $g(n)$ 보다 빠른 비율로 증가하는 함수

- Formal definition

–  $\omega(g(n)) = \{ f(n) \mid \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \}$

- 직관적 의미

–  $f(n) = \omega(g(n)) \Rightarrow f$ 는  $g$ 보다 빠르게 증가한다

# 각 점근적 표기법의 직관적 의미

- $O( g(n) )$ 
  - Tight or loose upper bound
- $\Omega( g(n) )$ 
  - Tight or loose lower bound
- $\Theta( g(n) )$ 
  - Tight bound
- $o( g(n) )$ 
  - Loose upper bound
- $\omega( g(n) )$ 
  - Loose lower bound




## 점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현 예 (4장에서 공부함)
  - 선택정렬
    - $\Theta(n^2)$
  - 힙정렬
    - $O(n \log n)$
  - 퀵정렬
    - $O(n^2)$
    - 평균  $\Theta(n \log n)$

# 시간 복잡도 분석의 종류

- **Worst-case**
  - Analysis for the worst-case input(s)
- **Average-case**
  - Analysis for all inputs
  - More difficult to analyze
- **Best-case**
  - Analysis for the best-case input(s)
  - 별로 유용하지 않음

# 저장/검색의 복잡도

- 
- 배열
    - $O(n)$
  - Binary search trees
    - 최악의 경우  $\Theta(n)$
    - 평균  $\Theta(\log n)$
  - Balanced binary search trees
    - 최악의 경우  $\Theta(\log n)$
  - B-trees
    - 최악의 경우  $\Theta(\log n)$
  - Hash table
    - 평균  $\Theta(1)$

# 크기 $n$ 인 배열에서 원소 찾기

- Sequential search
  - 배열이 아무렇게나 저장되어 있을 때
  - Worst case:  $\Theta(n)$
  - Average case:  $\Theta(n)$
- Binary search
  - 배열이 정렬되어 있을 때
  - Worst case:  $\Theta(\log n)$
  - Average case:  $\Theta(\log n)$