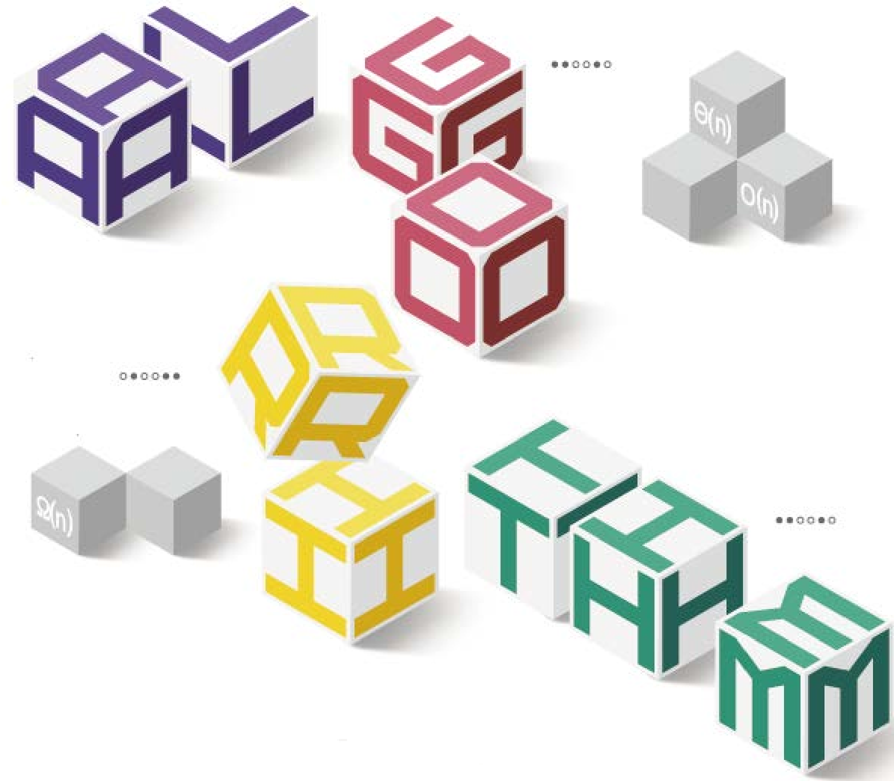


관계 중심의 사고법

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

학습목표

- 알고리즘을 설계하고 분석하는 몇 가지 기초 개념을 이해한다.
- 아주 기초적인 알고리즘의 수행 시간을 분석할 수 있도록 한다.
- 점근적 표기법을 이해한다.

알고리즘이란 무엇인가?

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

입출력의 예

- 문제
  - 정렬(sorting)
- 입력
  - 일련의 숫자들 (25, 17, 52, 36, 11)
- 출력
  - 입력 숫자들을 단조증가 순서로 재배열한 결과 (11, 17, 25, 36, 52)

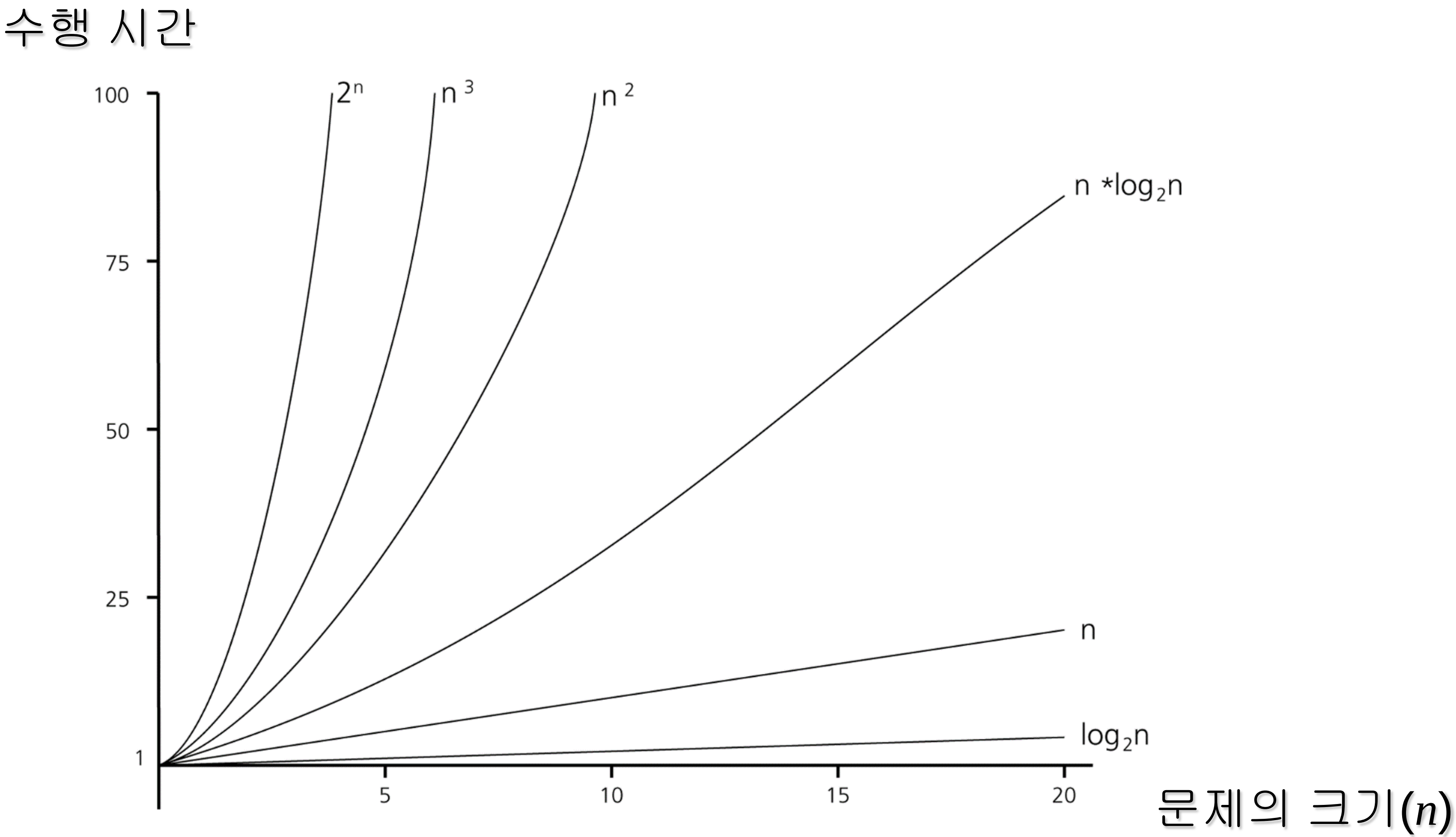
알고리즘 공부의 목적

- 특정한 문제를 위한 알고리즘의 습득
- 체계적으로 생각하는 훈련
- 지적 추상화의 레벨 상승
  - Intellectual abstraction
  - 연구나 개발에 있어 정신적 여유를 유지하기 위해 매우 중요한 요소

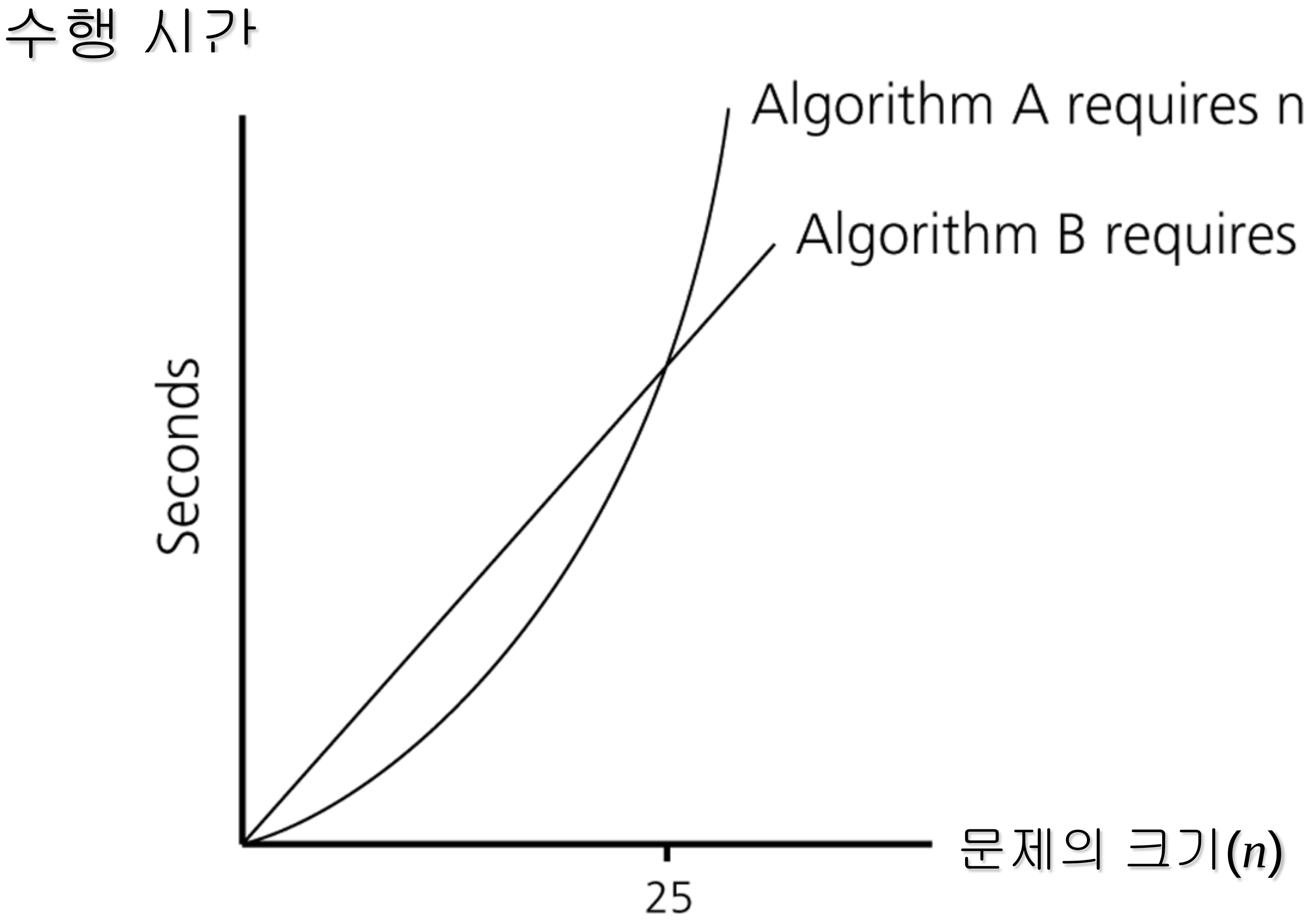
바람직한 알고리즘

- 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



알고리즘의 수행 시간


수행 시간
Algorithm A requires n²/5 seconds
Algorithm B requires 5*n seconds
Seconds
25
문제의 크기(n)

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 = ⌊n/2⌋;
    return A[k];
}
```

✓ n에 관계없이 상수 시간이 소요된다.

알고리즘의 수행 시간

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

✓ n에 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^2에 비례하는 시간이 소요된다.

알고리즘의 수행 시간

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

✓ n에 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
    - $N! = N \times (N\text{-}1)!$
  - 예2: 수열의 점화식
    - $a_n = a_{n-1} + 2$

Merge Sort (Divide-and-Conquer)

```
mergeSort(A[ ], p, r)      ▷ A[p ... r]을 정렬한다.
{
    if (p < r) then {
        q ← ⌊(p + q)/2⌋;  -----------------  ①  ▷ p, q의 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 ... q]와 A[q+1 ... r]을 합쳐
    정렬된 하나의 배열 A[p ... r]을 만든다.
}
```

```
mergeSort(A[ ], p, r)       ▷ A[p ... r]을 정렬한다.
{
      if (p < r) then {
            q ← ⌊(p + q)/2⌋;  -----------------  ①  ▷ p, q의 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
- 스케쥴링
  - 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 (correctness) 확인
- 효율성 (복잡도 complexity) 파악
  - 자원
    - 시간
    - 메모리, 통신대역, …

무결성 (correctness)

- Insertion Sort (Incremental Approach)

```
for j = 2 to n
    key = A[j]
    // insert A[j] into sorted A[1..j-1]
    i = j-1
    while i>0 and A[i]>key
        A[i+1] = A[i]
        i = i-1
    A[i+1] = key
```

Loop Invariant

- for 문이 시작될 때, A[1..j-1]은 입력 A[1..j-1]에 있는 원소들을 정렬된 순서로 가지고 있다.

- 초기: j=2일 때 성립
- 유지: loop가 시작될 때 invariant가 성립한다고 가정하고, 다음 loop가 시작될 때 invariant가 성립함을 증명함.
- 종료: loop가 끝날 때, invariant로부터 correctness를 얻어냄.

복잡도 분석
(Random-Access Machine Model)

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$\quad key = A[j]$	c_2	$n-1$
$\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$\quad i = j-1$	c_4	$n-1$
$\quad$ **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$\qquad A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$\qquad i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$\quad A[i+1] = key$	c_8	$n-1$

복잡도 분석

- $T(n) = c_1 n + c_2(n-1) + \ c_4(n-1)$

$$+c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} \left(t_j \ - 1\right)$$

$$+ \ c_7 \sum_{j=2}^{n} \left(t_j \ - 1\right) \ + c_8(n-1)$$

복잡도 분석

- $T(n) = c_1 n + c_2(n-1) + c_4(n-1)$

$$+c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right)$$

$$+c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n$$

$$-(c_2 + c_4 + c_5 + c_8)$$

$$= an^2 + bn + c$$

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

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

- O, Ω, Θ, ω, 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
  - $O(g(n)) = \{\ f(n) \mid \exists\, c > 0, n_0 \geq 0 \text{ such that } \forall\, n \geq n_0, c\boldsymbol{g}(n) \geq \boldsymbol{f}(n)\ \}$
  - $f(n) \in O(g(n))$을 관행적으로 $f(n) = O(g(n))$이라고 쓴다.
- 직관적 의미
  - $\boldsymbol{f}(n) = O(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$ 는 $\boldsymbol{g}$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
  - $\Omega(g(n)) = \{ f(n) \mid \exists\, c > 0, n_0 \geq 0 \text{ such that } \forall\, n \geq n_0, c\boldsymbol{g}(n) \leq \boldsymbol{f}(n) \}$

- 직관적 의미
  - $\boldsymbol{f}(n) = \Omega(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$는 $\boldsymbol{g}$보다 느리게 증가하지 않는다

점근적 표기법

$\Theta(g(n))$

- $g(n)$의 비율로 증가하는 함수

- Formal definition
  - $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
- 직관적 의미
  - $\boldsymbol{f}(n) = \Theta(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$는 $\boldsymbol{g}$와 같은 정도로 증가한다

$o(\ g(n)\ )$

- $g(n)$보다 느린 비율로 증가하는 함수

- Formal definition
  - $o(\boldsymbol{g}(n)) = \{\ \boldsymbol{f}(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}$

- 직관적 의미
  - $\boldsymbol{f}(n) = \Omega(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$는 $\boldsymbol{g}$보다 느리게 증가한다

$\omega(\ g(n)\ )$

– $g(n)$보다 빠른 비율로 증가하는 함수

- Formal definition
  – $\omega(g(n)) = \{\ f(n) \mid \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}$

- 직관적 의미
  – $\boldsymbol{f}(n) = \Omega(\boldsymbol{g}(n)) \Rightarrow \boldsymbol{f}$는 $\boldsymbol{g}$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
- $o(\ g(\mathrm{n})\ )$
  - Loose upper bound
- $\omega(\ g(n)\ )$
  - Loose lower bound

점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현 예
  - 선택정렬
    - $\Theta(n^2)$
  - 힙정렬 (Algorithm Design using Data Structure)
    - $O(n\log n)$
  - 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: $\Theta(\log n)$

알고리즘 개발 방법론

- Divide-and-Conquer
- Incremental Approach
- Using Data Structure
- Dynamic Programming
- Greedy Approach
- …

Thank you