알고리즘

2장 알고리즘 설계와 분석의 기초

학습 내용

1. 몇가지 기초 사항들
2. 점근적 표기
3. 점근적 표기의 엄밀한 정의

학습 목표

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

학습 내용

1. 몇가지 기초 사항들

2. 점근적 표기

3. 점근적 표기의 엄밀한 정의

바람직한 알고리즘

- 알고리즘이란
  - 주어진 문제의 해결 절차를 체계적으로 기술한 것
- 바람직한 알고리즘은
  - 명확해야 한다.
    - 모호하지 않고 이해하기 쉬워야 함
    - 가능하면 간명하게 표현
    - 지나친 기호적 표현은 오히려 명확성을 떨어뜨림
    - 명확성을 해치지 않으면 일반언어의 사용도 무방
  - 효율적이어야 한다.
    - 같은 문제를 해결하는 알고리즘들의 수행시간이 수백만배 이상 차이 날 수 있다.

알고리즘 분석

- 알고리즘 분석의 필요성
  - 알고리즘의 타당성 확인
  - 자원 사용 효율성 파악
    - 소요 시간 – 대부분 가장 중요한 관심 대상
    - 메모리
    - 통신대역 등

알고리즘 분석

- 알고리즘의 수행 시간 분석
  - 어느 정도의 입력에 대해 어느 정도의 시간이 소요되는지 예측할 수 있다.
  - 최악의 경우 분석, 평균적인 경우 분석 등
  - "입력의 크기에 대해 어떤 비율로 시간이 소요된다"로 표현한다.
  - 수행 시간 분석 예
    - "n개의 자료를 정렬하는 데(즉, 입력의 크기가 n일 때) n^2 에 비례한 시간이 소요된다"
    - "n개의 자료를 정렬하는 데(즉, 입력의 크기가 n일 때) n log n 에 비례한 시간이 소요된다"

알고리즘의 수행시간

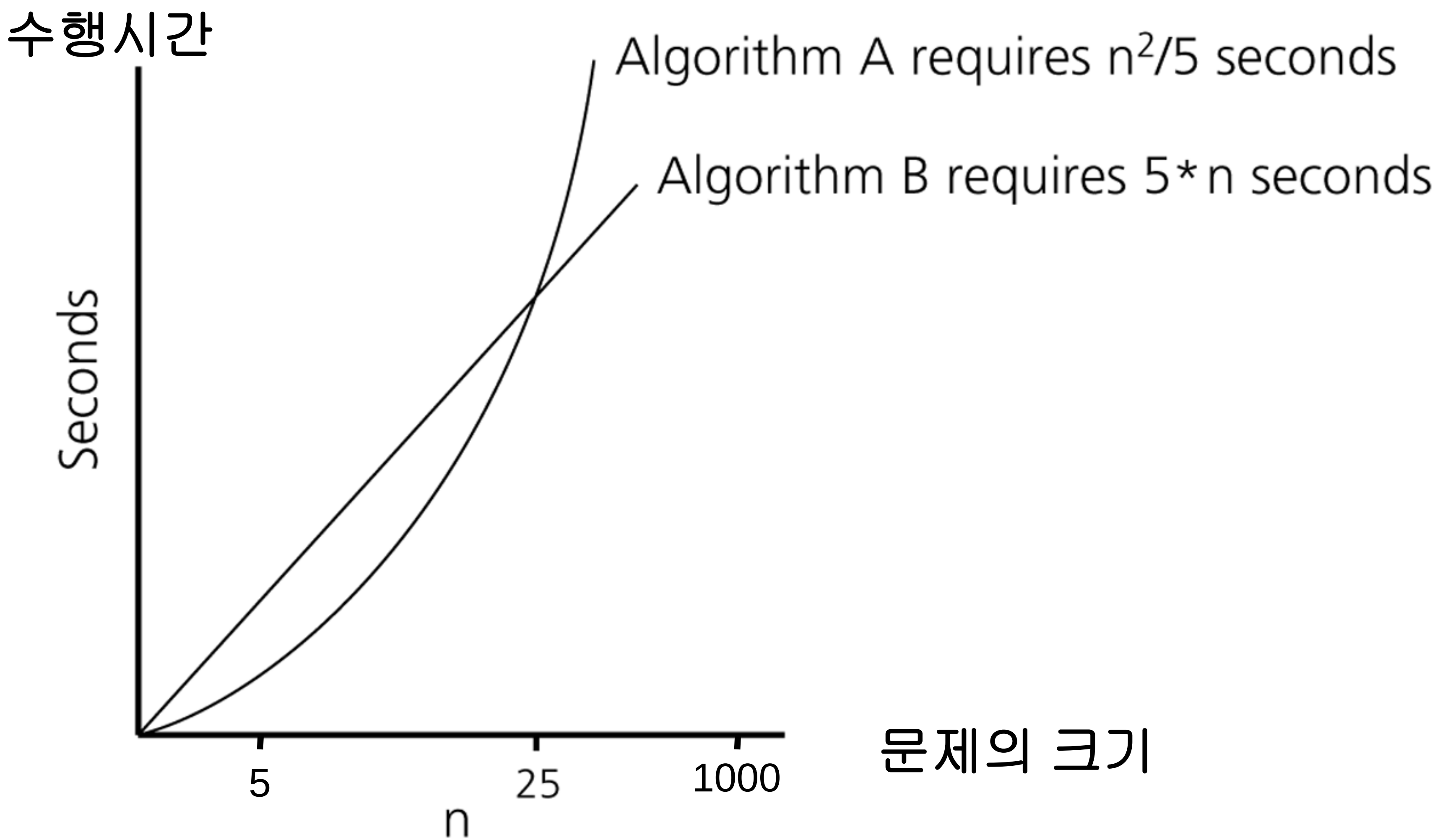
- 동일한 문제를 해결하는 두가지 알고리즘 예
  - 알고리즘 A
    - 입력의 크기가 n일 때 $n^2/5$ 시간이 소요
    - $T_A(n) = n^2/5$
  - 알고리즘 B
    - 입력의 크기가 n일 때 5*n 시간이 소요
    - $T_B(n) = 5*n$

알고리즘의 수행시간

```
Algorithm_A(n)
{
        for i ← 1 to n
                for j ← 1 to n
                        sum ← sum + 1 ;      // 1/5초 걸리는 작업이라고 하자.
}


Algorithm_B(n)
{
        for i ← 1 to n
                sum ← sum + n ;              // 5초 걸리는 작업이라고 하자.
}
```

알고리즘의 수행시간



수행시간
Algorithm A requires n2/5 seconds
Algorithm B requires 5*n seconds
Seconds
5
25
1000
n
문제의 크기

알고리즘의 수행시간

- 알고리즘의 수행 시간을 구하는 기준은 다양하게 잡을 수 있다. 예를 들어
  - for 루프의 반복횟수
  - 특정한 라인이 수행되는 횟수
  - 함수의 호출횟수, …
- 몇 가지 간단한 예를 통해 알고리즘의 수행시간을 살펴보자.
  - 배열 A의 크기(원소 수)를 n이라고 하자.
    - 특정 프로그래밍 언어를 이용한 표현이 아니므로 배열 인덱스는 1, 2, ..., n 로 본다.

알고리즘의 수행시간

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

✓ n에 관계없이 상수 시간이 소요된다.

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

✓ **n**에 비례하는 시간이 소요된다.

알고리즘의 수행시간

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

✓ n^2에 비례하는 시간이 소요된다.

알고리즘의 수행시간

```
sample4(A[ ], n)
{
    sum ← 0 ;
    for i ← 0 to n-1
        for j ← 0 to n-1 {
            k ← A[0..n-1]에서 임의로 n/2개를 뽑아 이 중 최대값;
            sum ← sum + k ;
        }
    return sum ;
}
```

✓ n^3에 비례하는 시간이 소요된다.

알고리즘의 수행시간

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

- 내부 for 루프의 반복횟수(문장 (1)의 수행 횟수)
  $= (n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2$

✓ n^2에 비례하는 시간이 소요된다.

재귀와 귀납적 사고

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

재귀의 예: factorial

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

재귀의 예: mergeSort

```
mergeSort(A[ ], p, r)  ▷ A[p ... r]을 정렬한다.
{
        if (p < r) then {
                q ← (p+r)/2;                    ▷ p, q의 중간 지점 계산
                mergeSort(A, p, q);             ▷ 전반부 정렬
                mergeSort(A, q+1, r);           ▷ 후반부 정렬
                merge(A, p, q, r);              ▷ 병합
        }
}

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

알고리즘의 수행시간

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

✓ **n**에 비례하는 시간이 소요된다.

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

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

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

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

```
main() {
        ... factorial(4) ;
}
```

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

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

```
main() {
        ... factorial(2) ;
}
```

다양한 알고리즘의 적용 주제들

- 차량 네비게이션
  - 최단 경로 알고리즘, …
- ATM 관리
  - TSP(Traveling Salesman Problem)
- Human Genome Project
  - 매칭, 계통도, …
- 데이터베이스, 웹페이지
  - 검색, …
- 고객 정보
  - 정렬, …
- 작업 공정
  - 위상정렬, …
- 신도시 설계시 가스관/수도관
  - 최소신장트리, …
- 반도체 칩 설계
  - partitioning, placement, routing, …

요약

- 알고리즘 공부를 통해 문제 해결 능력 향상
- 알고리즘은 명확해야 함. 즉 모호하지 않고 이해하기 쉬워야 하며, 지나치게 자세한 기술은 피해야 함
- 자기호출(recursion)은 자신과 동일하지만 크기가 다른 문제로 주어진 문제를 정의함으로써 문제 해결에 간명하게 접근하는 방법

학습 내용

1. 몇가지 기초 사항들

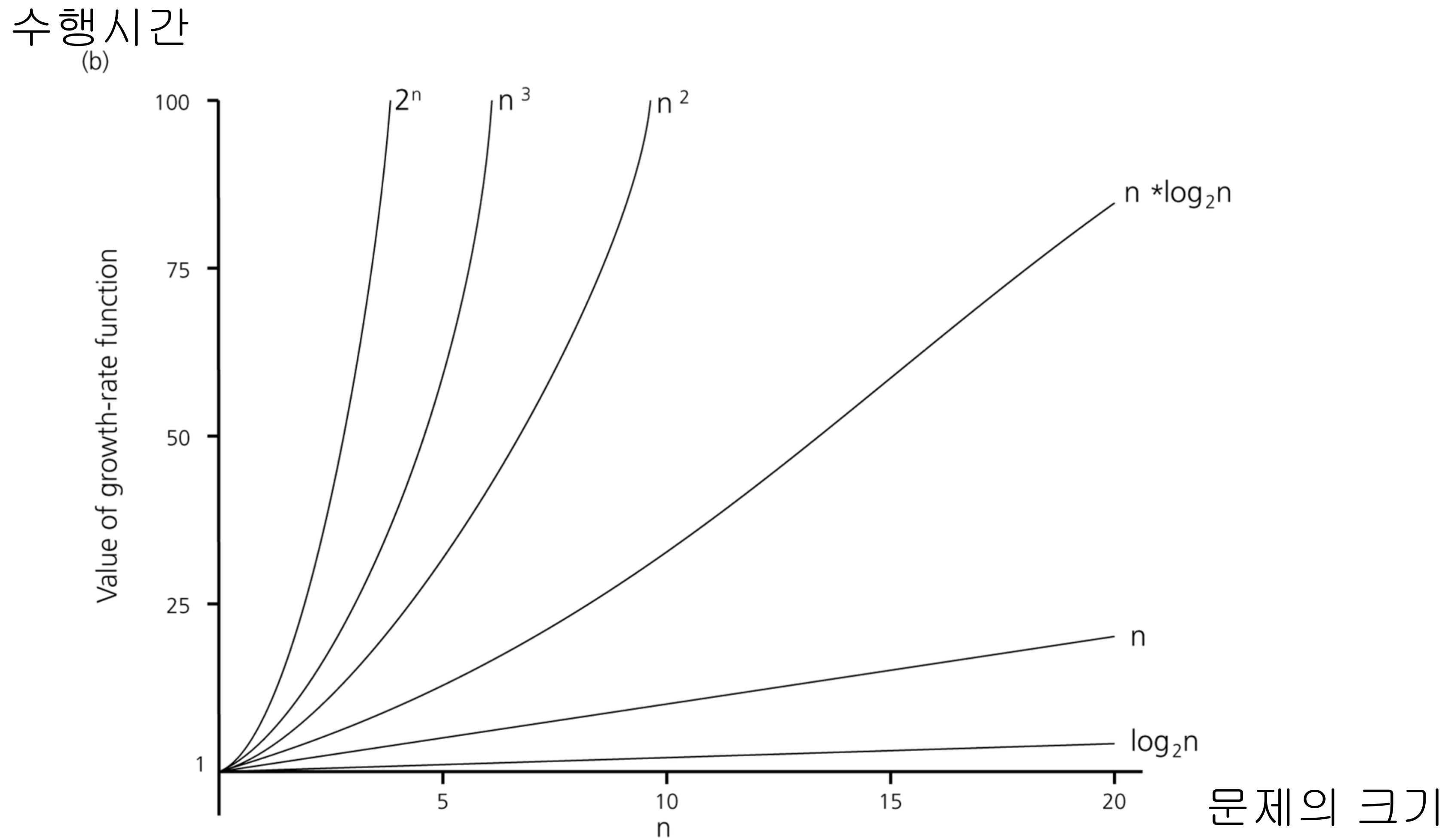
2. 점근적 표기

3. 점근적 표기의 엄밀한 정의

알고리즘의 점근적 분석

- 크기가 작은 문제
  - 알고리즘의 효율성이 중요하지 않다.
  - 비효율적인 알고리즘도 무방하다.
- 크기가 충분히 큰 문제
  - 알고리즘의 효율성이 중요하다.
  - 비효율적인 알고리즘은 치명적이다.
- 입력의 크기가 충분히 큰 경우에 대한 분석을 점근적 분석(asymptotic analysis)이라 한다.

여러 가지 함수의 증가율 비교



여러 가지 함수의 증가율 비교

	n					
Function	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}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

점근적 분석

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

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

- 대표적인 점근적 표기법(asymptotic notation)
  - Θ-표기법 (쎄타)
  - O-표기법 (빅-오, 오)
  - Ω-표기법 (빅-오메가, 오메가)

Θ-표기법

- $\Theta(f(n))$
  - 점근적 증가율이 $f(n)$과 일치하는 모든 함수의 집합
  - 예) 알고리즘의 소요 시간이 $\Theta(n^2)$ 라면 대략 n^2에 비례한 시간이 소요된다는 뜻
- 직관적 의미
  - $g(n) \in \Theta(f(n)) \Rightarrow g$ 는 f 와 같은 비율로 증가한다. 이때 상수 비율 차이는 무시
  - 예) $5n^2+4n \in \Theta(n^2)$
    - $5n^2+4n$의 최고차항에서 계수를 제외한 n^2 를 사용
    - 집합 개념이지만 관행적으로 $5n^2+4n = \Theta(n^2)$ 로 표현

Θ-표기법

- 문제: 다음 함수들 중 $\Theta(n^2)$인 것은?
  - $5n^3 + 2$
  - $n^2 \log n$
  - $3n^2$
  - $100\ n \log n$
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $n \log n + 5n$
  - $3n$

O-표기법

- $O(f(n))$
  - 점근적 증가율이 $f(n)$ 이하인 모든 함수의 집합
  - 예) $O(n\log n)$, $O(n^2)$, $O(2^n)$, …
- 직관적 의미
  - $g(n) \in O(f(n)) \Rightarrow g$ 는 f 보다 빠르게 증가하지 않는다. 이 때 상수 비율 차이는 무시
  - 예)
    - $5n^2+4n \in O(n^2)$ ➔ $5n^2+4n = O(n^2)$
    - $7n \in O(n^2)$ ➔ $7n = O(n^2)$

O-표기법

- 예) 다음 함수들은 모두 $O(n^2)$
  - $8n$
  - $3n^2 + 10n$
  - $7n^2 - 2n + 100$
  - $n \log n + 5n$
- 예) 다음 함수들은 모두 $O(1)$
  - 100
  - 2
- 알 수 있는 한 최대한 tight 하게!
  - $3n = O(n)$인데 굳이 $O(n^2)$로 쓸 필요 없다.
  - $n \log n + n = O(n \log n)$인데 굳이 $O(n^2)$로 쓸 필요 없다.
  - Tight하지 않은 만큼 정보의 손실이 일어난다.

- 문제: 다음 함수들 중 $O(n^2)$인 것은?
  - $\frac{1}{10}n^3$
  - $n^2 \log n$
  - $3n^2$
  - $100\ n \log n$
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $\log n + 5n^2 + 5n$
  - n
  - 5

Ω-표기법

- $\Omega(f(n))$
  - 점근적 증가율이 $f(n)$ 이상인 모든 함수의 집합
  - $O(f(n))$ 과 대칭적
- 직관적 의미
  - $g(n) \in \Omega(f(n)) \Rightarrow g$ 는 f 보다 느리게 증가하지 않는다. 이 때 상수 비율 차이는 무시
  - 예)
    - $5n^2+4n \in \Omega(n^2)$ ➔ $5n^2+4n = \Omega(n^2)$
    - $7n^3 \in \Omega(n^2)$ ➔ $7n^3 = \Omega(n^2)$

Ω-표기법

- 문제 : 다음 함수들 중 $\Omega(n^2)$인 것은?
  - $\frac{1}{10}n^3$
  - $n^2 \log n$
  - $3n^2$
  - $100\, n \log n$
  - $3n^2 + 2n$
  - $7n^2 - 100n$
  - $n \log n + 5n$
  - $3n$

점근적 표기법의 관계

- Θ-표기의 수학적 정의
  - O-표기와 Ω-표기의 정의를 이용해 정의할 수 있다.
  - $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- 예) $5n^2+4n$
  - $5n^2+4n \in O(n^2)$
  - $5n^2+4n \in \Omega(n^2)$

  ➔ $5n^2+4n \in \Theta(n^2)$

학습 내용

1. 몇가지 기초 사항들
2. 점근적 표기
3. **점근적 표기의 엄밀한 정의**

점근적 표기법 Asymptotic Notations

- 5가지 점근적 표기법
  - O-표기법 (빅-오, 오)
  - Ω-표기법 (빅-오메가, 오메가)
  - Θ-표기법 (쎄타)
  - o-표기법 (리틀-오)
  - ω-표기법 (리틀-오메가)

O-표기법

- **수학적** 정의
  - $O(\boldsymbol{f}(n)) = \{\, \boldsymbol{g}(n) \mid \exists c>0, n_0 \geq 0 \text{ s.t.} \forall n \geq n_0, \boldsymbol{g}(n) \leq c\boldsymbol{f}(n) \,\}$
  - $O(\boldsymbol{f}(n)) = \{\, \boldsymbol{g}(n) \mid$ 모든 $n \geq n_0$에 대하여 $\boldsymbol{g}(n) \leq c\boldsymbol{f}(n)$ 인 양의 상수 c와 n_0가 존재한다 $\}$
- 직관적 의미
  - $O(\boldsymbol{f}(n)) = \{\, \boldsymbol{g}(n) \mid$ 충분히 큰 모든 n에 대하여 $\boldsymbol{g}(n) \leq c\boldsymbol{f}(n)$ 인 양의 상수 c가 존재한다 $\}$

O-표기법

- 예) $5n^2 = O(n^2)$ 임을 보여라.
  - 다음 정의를 만족하는 상수 c와 n_0이 존재함을 보이면 된다.

    $O(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid \exists c>0,\ n_0 \geq 0 \text{ s.t.} \forall n \geq n_0,\ \boldsymbol{g}(n) \leq c\boldsymbol{f}(n)\ \}$
  - $g(n) = 5n^2$ 이고,
  - $f(n) = n^2$ 이므로
  - $c=6$, $n_0=1$ 로 잡으면 모든 $n \geq 1$에 대해 $5n^2 \leq 6n^2$

Ω-표기법

- **수학적** 정의
  - $\Omega(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid \exists c>0,\ n_0 \geq 0 \text{ s.t.} \forall n \geq n_0,\ \boldsymbol{g}(n) \geq c\boldsymbol{f}(n)\ \}$
  - $\Omega(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid$ 모든 $n \geq n_0$에 대하여 $\boldsymbol{g}(n) \geq c\boldsymbol{f}(n)$ 인 양의 상수 c와 n_0가 존재한다 }
- 직관적 의미
  - $\Omega(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid$ 충분히 큰 모든 n에 대하여 $\boldsymbol{g}(n) \geq c\boldsymbol{f}(n)$ 인 양의 상수 c가 존재한다 }

Ω-표기법

- 예) $5n^2 = \Omega(n^2)$ 임을 보여라.
  - 다음 정의를 만족하는 상수 c와 n_0이 존재함을 보이면 된다.

    $\Omega(\boldsymbol{f}(n)) = \{\ \boldsymbol{g}(n) \mid \exists c>0,\ n_0 \geq 0 \text{ s.t. } \forall n \geq n_0,\ \boldsymbol{g}(n) \geq c\boldsymbol{f}(n)\ \}$
  - $g(n) = 5n^2$ 이고,
  - $f(n) = n^2$ 이므로
  - $c=4$, $n_0=1$ 로 잡으면 모든 $n \geq 1$에 대해 $5n^2 \geq 4n^2$

Θ-표기법

- 수학적 정의
  - $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$ ➔ $\Theta(f(n))$은 $O(f(n))$ 과 $\Omega(f(n))$ 이 동시에 성립하는 함수
  - $\Theta(f(n)) = \{ g(n) \mid \exists c_1, c_2>0, n_0\geq 0 \text{ s.t.} \forall n\geq n_0, c_1 f(n) \leq g(n) \leq c_2 f(n)\}$
- 직관적 의미
  - $\Theta(f(n)) = \{ g(n) \mid$ 충분히 큰 모든 n에 대하여 $c_1 f(n) \leq g(n) \leq c_2 f(n)$인 양의 상수 c_1, c_2가 존재한다 $\}$

o-표기법

- ***o*(*f*(*n*))**
  - *f*(*n*)보다 느린 비율로 증가하는 함수
  - 함수 증가율이 점근적 의미에서 어느 한계보다 더 작음(<)을 표현
- 형식적 정의
  - $o(f(n)) = \{\ g(n) \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0 \}$
- 직관적 의미
  - ***g***(*n*) = *o*(***f***(*n*)) ⇒ ***g*** 는 ***f*** 보다 느리게 증가한다.
- 예) $n^2 = o(n^3)$

ω-표기법

- $\omega(f(n))$
  - $f(n)$보다 빠른 비율로 증가하는 함수
  - 함수 증가율이 점근적 의미에서 어느 한계보다 더 큼 (>)을 표현
- 형식적 정의
  - $\omega(f(n)) = \{\ g(n) \mid \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty \}$
- 직관적 의미
  - $g(n) = \omega(f(n)) \Rightarrow$ ***g*** 는 ***f*** 보다 빠르게 증가한다.
- 예) $n^2 = \omega(n)$

점근적 표기법

- 점근적 표기법의 직관적 의미

$O(f(n))$	tight or loose upper bound	$\leq$	차수가 $f(n)$ 이하
$\Omega(f(n))$	tight or loose lower bound	$\geq$	차수가 $f(n)$ 이상
$\Theta(f(n))$	tight bound	$=$	차수가 $f(n)$과 같음
$o(f(n))$	loose upper bound	$<$	차수가 $f(n)$ 미만
$\omega(f(n))$	loose lower bound	$>$	차수가 $f(n)$ 초과

점근적 복잡도

- 정렬 알고리즘의 **시간 복잡도(time complexity)**를 예로 살펴보자. – 정렬은 4장에서 다룸
  - 선택정렬
    - $\Theta(n^2)$
  - 힙정렬
    - $O(n \log n)$
  - 퀵정렬
    - $O(n^2)$
    - 평균 $\Theta(n \log n)$
  - 비교에 기반한 정렬
    - $\Omega(n \log n)$

점근적 복잡도 예

```
sampleA(A[ ], n)
{
    sum1 ← 0;

    for i ← 0 to n-1
        sum1 ← sum1 + A[i];

    sum2 ← 0;

    for i ← 0 to n-1
        for j ← 0 to n-1
            sum2 ← sum2 + A[i] * A[j];

    return sum1 + sum2;
}
```

✓수행시간은 n^2에 비례한다. 즉, $\Theta(n^2)$

점근적 복잡도 예

```
matrixMult(A[][], B[][], M[][], n)  // A, B, M은 n x n 행렬
{
    for i ← 0 to n-1
        for j ← 0 to n-1 {
            M[i, j] ← 0;
            for k ← 0 to n-1
                M[i, j] ← M[i, j] + A[i, k] * B[k, j];
        }
}
```

✓ 수행시간은 n^3에 비례한다. 즉, $\Theta(n^3)$

시간 복잡도 분석의 종류

- 최악의 경우(worst-case)
  - worst-case 입력에 대해 수행시간을 분석
- 평균의 경우(average-case)
  - 모든 입력에 대해 분석
  - 최악의 경우보다 분석하기가 까다로움
- 최선의 경우(best-case)
  - best-case 입력에 대해 수행시간을 분석
  - 별 유용하지 않음

시간 복잡도 예

- 자료구조에 따라 검색 연산의 시간 복잡도가 다름
  - 배열(array)
    - $O(n)$
  - 이진검색트리(binary search tree)
    - 최악의 경우 $\Theta(n)$
    - 평균 $\Theta(\log n)$
  - 균형이진검색트리(balanced binary search tree)
    - 최악의 경우 $\Theta(\log n)$
  - B-트리
    - 최악의 경우 $\Theta(\log n)$
  - 해시테이블(hash table)
    - 평균 $\Theta(1)$

낮아짐

시간 복잡도 예

- 크기 n인 배열에서 검색 시간 복잡도는 $O(n)$
- 이를 검색 종류에 따라 구분하면:
  - 순차 검색(sequential search)
    - 배열이 아무렇게나 저장되어 있을 때
    - worst-case: $\Theta(n)$
    - average-case: $\Theta(n)$
  - 이진 검색(binary search)
    - 배열이 정렬되어 있을 때 사용할 수 있는 알고리즘임
    - worst-case: $\Theta(\log n)$
    - average-case: $\Theta(\log n)$

문제

- 각 함수에 해당하는 것을 보기에서 모두 고르세요.

보기: 가. $O(n)$, 나. $\Omega(n)$, 다. $\Theta(n)$, 라. $o(n)$, 마. $\omega(n)$

바. $O(n^2)$, 사. $\Omega(n^2)$, 아. $\Theta(n^2)$, 자. $o(n^2)$, 차. $\omega(n^2)$

(1) $8n - 3$

(2) $5 + 3\log n$

(3) $4n\log n$

(4) $5n^2 + 3$

(5) $n^3 + 3n^2\log n$

(6) $n^2\log n + n^2$

문제

- 다음 두 알고리즘의 시간 복잡도를 m, n에 대한 점근적 표기로 나타내세요.

```
algorithm1(A[], count[])    // A[0..n-1], count[0..m-1]
{
    for i ← 0 to n-1
        count[A[i]] ← count[A[i]] + 1;
}
```

```
algorithm2(A[], count[])    // A[0..n-1], count[0..m-1]
{
    for i ← 0 to n-1
        for j ← 0 to m-1
            if(A[i] = j)
                count[j] ← count[j] + 1;
}
```

요약

- 점근적 분석은 입력의 크기가 충분히 큰 경우에 대한 분석
- 알고리즘의 수행시간 복잡도를 나타내는 점근적 표기법
  - ***O*-**표기 (빅-오, 오)
  - ***Ω*-**표기 (빅-오메가, 오메가)
  - ***Θ*-**표기 (쎄타)
  - ***o*-**표기 (리틀-오)
  - ***ω*-**표기 (리틀-오메가)
- O-표기와 o-표기는 점근적 상한, **Ω**-표기와 ω-표기는 점근적 하한을 나타냄
- O-표기와 **Ω**-표기가 표현하는 한계가 일치할 때는 **Θ**-표기를 사용