1 알고리즘의 수행시간

1 알고리즘의 수행시간

1 알고리즘을 통한 문제 해결

- 수학에서는 문제를 풀기 위해 정의나 정리들을 활용
- 컴퓨터에서는 문제해결을 위해 알고리즘 이용
- 문제를 철저하게 분석한 후 알고리즘을 거쳐 프로그램 작성

1 알고리즘의 수행시간

2 바람직한 알고리즘

- 명확해야 함
  - 이해하기 쉽고 가능하면 간명하도록 해야 함
  - 지나친 기호적 표현은 오히려 명확성을 떨어뜨림
  - 명확성을 해치지 않으면 일반 언어의 사용도 무방

- 효율적이어야 함
  - 같은 문제를 해결하는 알고리즘들의 수행 시간이 수백만 배 이상 차이 날 수 있음

1 알고리즘의 수행시간

3 알고리즘 공부의 목적

- 알고리즘은 문제 해결 절차를 체계적으로 기술한 것
- 알고리즘 공부의 목적은 특정한 문제를 위한 알고리즘의 습득, 체계적으로 생각하는 훈련, 지적 추상화의 레벨 상승을 위함
- 같은 문제에서도 효율이 아주 큰 차이가 나는 다양한 알고리즘이 존재함
- 이들에 대해 학습하는 과정에서 얻을 수 있는 여러 가지 기법과 생각하는 방법도 중요

1 알고리즘의 수행시간

4 알고리즘 분석의 필요성

- 바람직한 알고리즘은 **명확**하고 **효율적**이어야 함
- 알고리즘을 설계하고 나면 이 알고리즘이 자원을 얼마나 소모하는지 분석해야 함
- 자원은 소요 시간, 메모리, 통신 대역 등

1 알고리즘의 수행시간

4 알고리즘 분석의 필요성

- 자원의 가장 중심 대상은 **소요시간**
- 시간의 분석은 최악의 경우와 평균적인 경우에 대한 분석이 대표적

↳ 시간 분석을 하면 알고리즘이 어느 정도의 입력에서 어느 정도의 시간이 필요한지 미리 짐작 가능하며 주어진 시간에 요구하는 작업이 완료 가능한지를 알 수 있음

5 알고리즘의 수행시간

- 알고리즘의 수행 시간은 입력의 크기에 대해 시간이 어떤 비율로 소요되는지로 표현됨
- 입력의 크기
  - 정렬의 경우 정렬하고자 하는 개체의 수
  - 도시 간의 거리를 구하는 경우 계산에 관여되는 도시의 총수와 도시 간 간선(도로)의 총수
  - 계승을 구하는 경우에는 계승치를 구하고자 하는 자연수의 크기

5 알고리즘의 수행시간

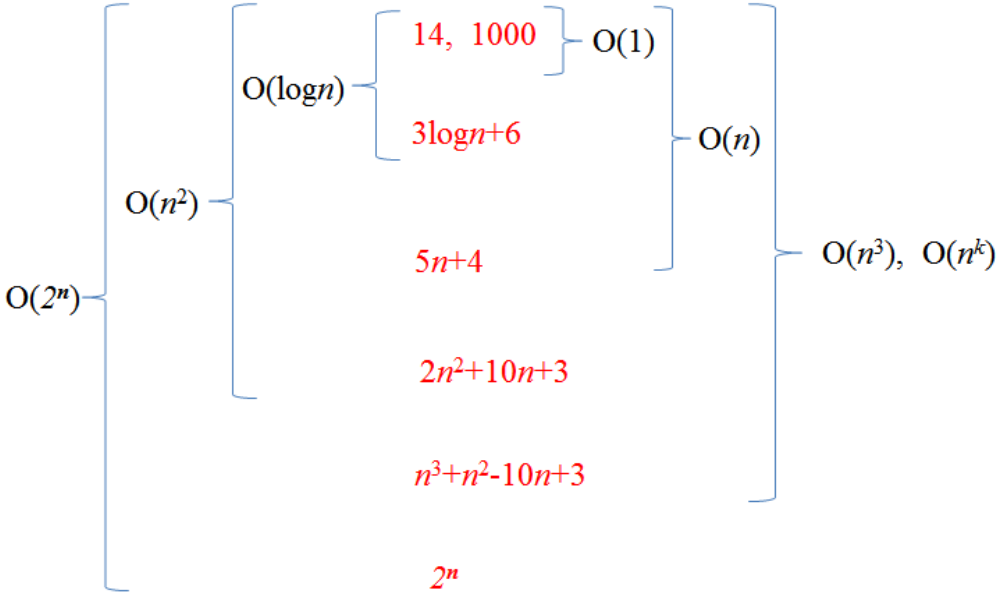
◆ 입력값 n에 따른 알고리즘 수행 시간

$$O(1) < O(\log_2 n) < O(n) < O(n\log_2 n) < O(n^2) < O(n^3) < O(2^n)$$

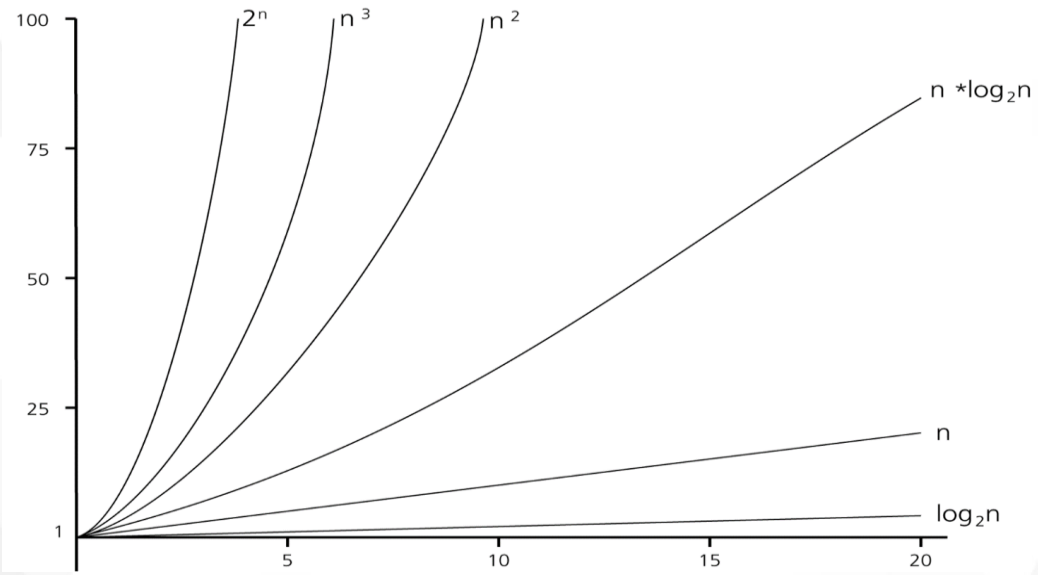
O(1)	상수 시간(Constant time)
O(logn)	로그 시간(Logarithmic time)
O(n)	선형 시간(Linear time)
O(nlogn)	로그 선형 시간(Log-linear time)
$O(n^2)$	제곱 시간(Quadratic time)
$O(n^3)$	세제곱 시간(Cubic time)
$O(2^n)$	지수 시간(Exponential time)

5 알고리즘의 수행시간

수행 시간의 포함 관계



5 알고리즘의 수행시간



5 알고리즘의 수행시간

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

5 알고리즘의 수행시간

- **알고리즘의 수행시간을 좌우하는 기준**
  예) for 루프의 반복횟수, 특정한 행이 수행되는 횟수,
  함수의 호출횟수 등

5 알고리즘의 수행시간

◆ 예) 입력으로 n개의 데이터가 저장된 배열 data가 주어지고 그 중 $\frac{n}{2}$번째 데이터를 반환한다.

```
int sample(int data[ ], int n)
{
  int k= n/2;
  return data[k];
}
```

대입연산: 1번

풀이) n에 관계없이 상수 시간이 소요됨
이 경우 알고리즘의 시간복잡도는 O(1)

5 알고리즘의 수행시간

◆ 예) 입력으로 n개의 데이터가 저장된 배열 data가 주어지고 그 합을 구하여 반환한다.

```
int sample(int data[ ], int n)
 {
   int sum = 0;
   for( int i = 0 ; i < n ;  i++)
      sum = sum + data[i];
   return sum;
 }
```

풀이)
이 알고리즘에서 가장 자주 실행되는 문장이며, 0부터 n-1까지 실행하여 실행 횟수는 항상 n번

가장 자주 실행되는 문장의 실행횟수가 n번이라면 모든 문장의 실행 횟수의 합은 n에 선형적으로 비례하며 모든 연산들의 실행횟수의 합도 역시 n에 선형적으로 비례
(대입연산: n+1번, 덧셈연산: n번)
따라서 O(n)의 시간복잡도를 갖음

5 알고리즘의 수행시간

◆ 예)

```
sample(A[ ], n)
{
   sum = 0 ;
   for ( int i = 1 ; i <= n ;  i++)
      for ( int j = 1 ; j <= n ;  j++)
         sum = sum+ A[i]*A[j] ;
    return sum ;
}
```

풀이)
for 루프가 $n \times n = n^2$ 번
반복되고 각 루프에서는
덧셈 한번과 곱셈 한 번
즉, 상수 시간 작업이 수행됨

대입연산: $n^2 + 1$번,
덧셈연산: n^2번,
곱셈연산: n^2번

따라서 총 수행시간은 $O(n^2)$

6 재귀적(Recursive) 알고리즘

- 재귀 = 자기호출
- 재귀적 사고는 해법을 알고 그것을 반복적으로 적용시킴
- 어떤 문제 안에 크기만 다를 뿐 성격이 똑같은 작은 문제들이 포함되어 있는 것
- 병렬화 개념은 하나의 일을 나누어 동시에 병렬로 처리
- 팩토리얼, 수열의 점화식, 피보나치, 병합 정렬 등

6 재귀적(Recursive) 알고리즘

◆ 예) 팩토리얼
재귀적 정의에 따라 n=5일 때 n!의 값 구하기

$$n! = \begin{cases} 1 & \text{if } n = 0, 1 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

풀이) $5! = 5 \times 4!$
$= 5 \times 4 \times 3!$
$= 5 \times 4 \times 3 \times 2!$
$= 5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4 \times 3 \times 2 \times 1$
$= 120$

2 알고리즘의 효율성 분석

2 알고리즘의 효율성 분석

1 알고리즘의 분석

- 알고리즘의 **자원(Resource) 사용량**을 분석
- 자원이란 실행 시간, 메모리, 저장장치, 통신 등
- 여기서는 **실행시간의 분석**에 대해서 다룸

2 알고리즘의 복잡도(Complexities of Algorithm)

- 여러 알고리즘 중 처리 시간이나 차지하는 메모리 용량이 작은 알고리즘이 프로그램 효율성도 높음

- 알고리즘을 어떤 식으로 작성하느냐에 따라 실행되는 시간과 공간이 다르며 이것을 알고리즘의 복잡도라고 함

2 알고리즘의 복잡도(Complexities of Algorithm)

◆ 알고리즘 성능분석

- 성능 측정: 실제 구현 필요
  구체적으로 실행시켜 시간을 확인

- 성능 분석: 구현 불필요
  시간복잡도: 입력 개수(n)에 따른 실행횟수
  공간복잡도

※ 성능 측정 방법은 구현해야 하므로 잘 안씀

2 알고리즘의 복잡도(Complexities of Algorithm)

알고리즘 성능분석

입력 자료수	n^2초	2^n초
n=4	16초	16초
n=100	10,000초	2^{100}초 = 4X10^{22}년

2 알고리즘의 효율성 분석

2 알고리즘의 복잡도(Complexities of Algorithm)

- 시간복잡도, 공간복잡도가 있음

시간복잡도	▪ 알고리즘이 수행되는 시간
공간복잡도	▪ 알고리즘이 수행될 때 필요한 메모리 공간

- 일반적으로 알고리즘들을 비교할 때에는 시간복잡도가 주로 사용됨

2 알고리즘의 복잡도(Complexities of Algorithm)

- 알고리즘의 복잡도는 처리해야 하는 데이터의 양이나 표현 방법, 컴파일러 등에 따라 달라지므로 성능 측정이 어려움

- 입력 데이터가 많을수록 실행 속도나 성능이 저하되므로 입력 데이터가 무한히 많아질 때의 알고리즘의 성능을 주로 평가

2 알고리즘의 효율성 분석

3 시간 복잡도

- 실행시간은 실행환경에 따라 달라짐
  (하드웨어, 운영체제, 언어, 컴파일러 등)
- 실행 시간을 측정하는 대신 **연산의 실행 횟수**를 카운트
- 연산의 실행 횟수는 입력 **데이터의 크기에 관한 함수**로 표현
- 데이터의 크기가 같더라도 실제 데이터에 따라서 달라짐

2 알고리즘의 효율성 분석

3 시간 복잡도

- 시간복잡도는 알고리즘이 수행하는 기본적인 연산 횟수를 입력 크기에 대한 함수로 표현
- 예) 10장의 숫자 카드 중에서 최대 숫자 찾기

  순차탐색으로 찾는 경우에 숫자 비교가 기본적인 연산이고, 총 비교 횟수는 9 임

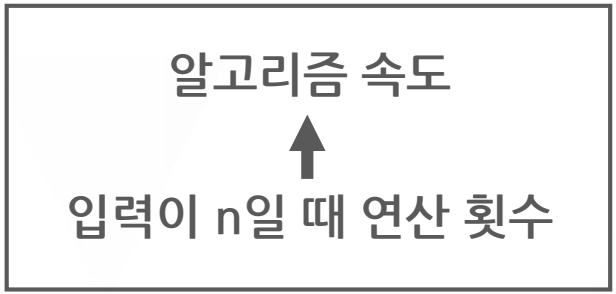
  n장의 카드가 있다면, (n-1)번의 비교 수행

  ➔ 시간복잡도는 (n-1)

2 알고리즘의 효율성 분석

3 시간 복잡도

- 알고리즘의 복잡도를 표현하는 데는 다음과 같은 분석 방법들이 있음
  - **최악의 경우** 시간 분석(Worst Case Analysis)
  - **평균의 경우** 분석(Average Case Analysis)
  - **최선의 경우** 분석 (Best Case Analysis)



3 점근적 표기

3 점근적 표기

1 점근적 분석

- 입력크기가 작은 문제는 알고리즘의 효율성이 중요하지 않지만 입력 크기가 충분히 큰 문제는 비효율적인 알고리즘은 치명적임
- 입력 크기 **n이 무한대로 커질 때**의 복잡도를 간단히 표현하기 위해 사용하는 표기법
- 이미 알고 있는 점근적 개념의 예 $\lim_{n\to\infty} f(n)$
- O(Big-Oh) 표기
  Ω(Big-Omega) 표기
  Θ(Big-Theta) 표기

3 점근적 표기

2 점근적 표기법

- 데이터의 개수 $n \to \infty$ 일 때 수행시간이 증가하는 증가율로 시간복잡도를 표현하는 기법
- 알고리즘에 포함된 연산들의 실행 횟수를 표기하는 하나의 기법
- 최고차항의 차수만으로 표시

↳ 따라서 **가장 자주 실행되는 연산 혹은 문장**의 실행횟수를 고려하는 것으로 충분

3 점근적 표기

2 점근적 표기법

- 유일한 분석법도 아니고 가장 좋은 분석법도 아님
  - 다만 (상대적으로) 가장 간단함
  - 알고리즘의 실행환경에 비의존적임
  - 그래서 가장 광범위하게 사용됨

3 점근적 표기

3 Big-O 표기법

- 복잡도의 점근적 상한을 나타냄(상한선, 최악의 시간)
- (예) O(n), O(n logn), $O(n^2)$, $O(2^n)$, ...
- 기껏해야 g(n)의 비율로 증가하는 함수
- 정의
  - f(n)과 g(n)이 주어졌을 때 모든 $n \geq n_0$ 에 대하여 **$f(n) \leq cg(n)$** 을 만족하는 상수 c와 n_0 가 존재하면 f(n) = O(g(n))임

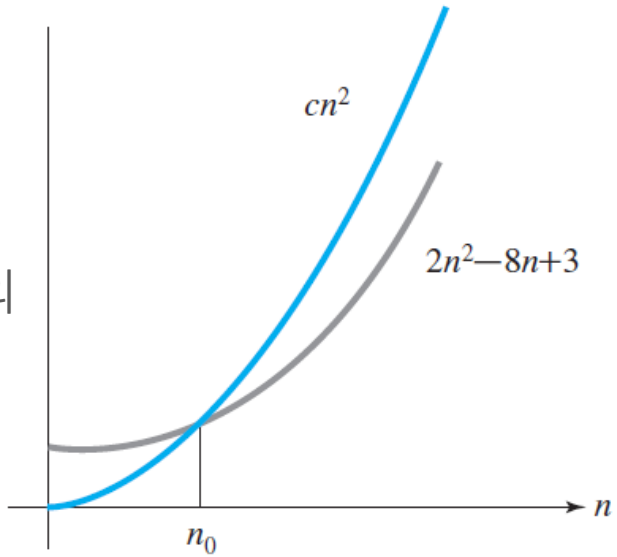
3 점근적 표기

3 Big-O 표기법

- 복잡도가 **f(n) = $2n^2-8n+3$** 이라면
  f(n)의 O-표기는 **$O(n^2)$**임

  먼저 f(n)의 단순화된 표현은 n^2 임

  단순화된 함수 n^2 에 임의의 상수 c를 곱한 cn^2 이 n이 증가함에 따라 f(n)의 상한이 됨 (단, c>0)

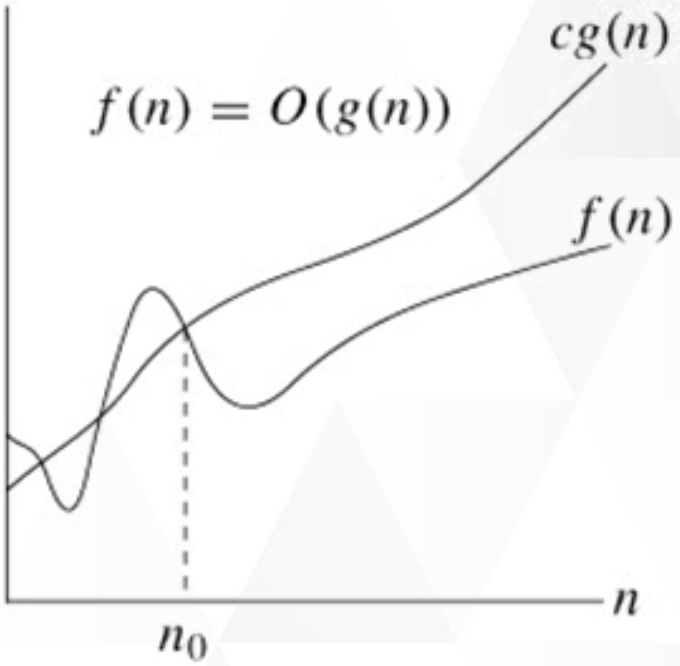


3 점근적 표기

3 Big-O 표기법

- 복잡도 f(n)과 O-표기를 그래프로 나타냄

  n이 증가함에 따라 O(g(n))이 점근적 상한이라는 것을 보여줌
  즉, **g(n)이 n_0보다 큰 모든 n에 대해서 항상 f(n)보다 크다는 것**을 보여줌



3 점근적 표기

3 Big-O 표기법

- 예) $O(n^2)$은 $3n^2 + 2n, 7n^2 - 100n, n \log n + 5n, 3n$ 등을 포함
  - $n \log n + 5n = O(n \log n)$ 인데 굳이 $O(n^2)$으로 쓸 필요 없음
  - 엄밀하지 않은 만큼 정보의 손실이 일어남

3 Big-O 표기법

◆ 예) 다음 알고리즘의 복잡도를 구하시오.

```
algorithm sum(int n)
{
  x = 0;
  for i = 0 to n
    x = x + 1;
  next i
}
```

풀이) 이 알고리즘은
0부터 n까지 반복하여 x=x+1 문장을 수행함
이때 x=x+1 문장에서 덧셈 연산은 1번 수행되고
반복문은 0부터 n이 될 때까지 n+1번 수행
따라서 복잡도는 f(n)=n+1=O(n)이 됨

3 점근적 표기

4 Big-Ω 표기법

- 복잡도의 점근적 하한을 나타냄(하한선, 최상, 최선)
- O(g(n))과 대칭적
- 적어도 g(n)의 비율로 증가하는 함수
- 정의
  - f(n)과 g(n)이 주어졌을 때 모든 $n \geq n_0$ 에 대하여 $f(n) \geq cg(n)$ 을 만족하는 상수 c와 n_0가 존재하면 $f(n) = \Omega(g(n))$

4 Big-Ω 표기법

- **$f(n) = 2n^2-8n+3$**의 Ω-표기는 **$\Omega(n^2)$**임

  f(n)=Ω(n^2)은 “n이 증가함에 따라 $2n^2-8n+3$이 cn^2보다 작을 수 없다”라는 의미임

  이때 상수 c=1로 놓으면 됨

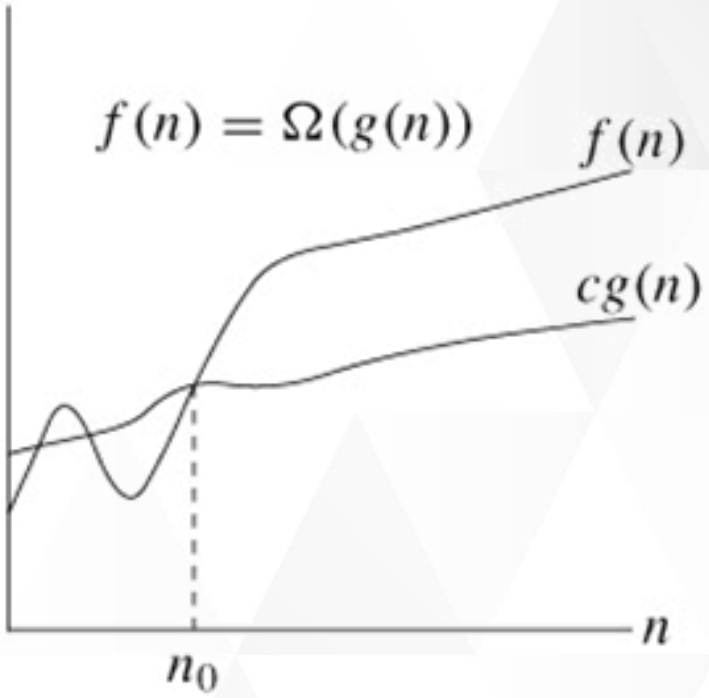
  O-표기와 마찬가지로 Ω-표기도 복잡도 다항식의 최고 차항만 계수 없이 취하면 됨

3 점근적 표기

4 Big-Ω 표기법

- 복잡도 f(n)과 Ω-표기를 그래프로 나타냄

n이 증가함에 따라 Ω(g(n))이 점근적 하한이라는 것
즉, **g(n)이 n_0보다 큰 모든 n에 대해서 항상 f(n)보다 작다는 것**을 보여줌



3 점근적 표기

5 Big-Θ 표기법

- 복잡도의 상한과 하한이 동시에 적용되는 경우를 나타냄
- 동일한 증가율
- O-표기와 Ω-표기가 같은 경우에 사용
- 정의
  - f(n)과 g(n)이 주어졌을 때 모든 $n \geq n_0$ 에 대하여 $c_1 g(n) \leq f(n) \leq c_2 g(n)$ 을 만족하는 상수 c_1, c_2와 n_0가 존재하면 f(n) = Θ(g(n))

3 점근적 표기

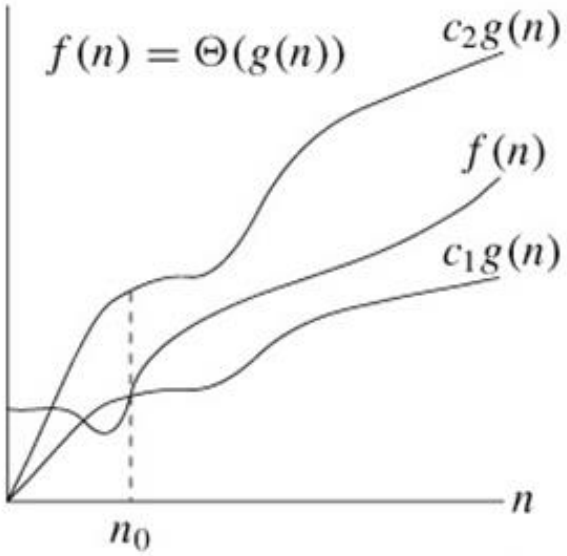
5 Big-Θ 표기법

- $f(n) = 2n^2+10n+3 = O(n^2) = \Omega(n^2)$이므로
  $f(n)=\Theta(n^2)$임

  “f(n)은 n이 증가함에 따라 n^2과 **동일한** 증가율을 가진다”라는 의미

5 Big-Θ 표기법

- 복잡도 f(n)과 Θ-표기를 그래프로 n_0보다 큰 모든 n에 대해서 Θ-표기가 **상한과 하한 동시에** 만족한다는 것을 보여줌



6 효율적인 알고리즘

왜 효율적인 알고리즘이 필요한가?

10억 개의 숫자를 정렬하는데 PC에서 $O(n^2)$ 알고리즘은 300여년이 걸리는 반면에 O(n logn) 알고리즘은 5분 만에 정렬함

$O(n^2)$	1,000	1백만	10억
PC	< 1초	2시간	300년
슈퍼컴	< 1초	1초	1주일

O(nlogn)	1,000	1백만	10억
PC	< 1초	<1초	5분
슈퍼컴	< 1초	<1초	<1초

3 점근적 표기

6 효율적인 알고리즘

- 효율적인 알고리즘은 슈퍼컴퓨터보다
  더 큰 가치가 있음
- 값 비싼 H/W의 기술 개발보다
  효율적인 알고리즘 개발이 훨씬 더 경제적