KOREATECH
한국기술교육대학교

02장. 알고리즘 설계와 분석의 기초

Youn-Hee Han
LINK@KOREATECH
http://link.koreatech.ac.kr

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

학습 목표

◈ 알고리즘을 설계하고 분석하는 몇 가지 기초 개념 이해

◈ 기초적인 알고리즘 수행 시간 분석

◈ 점근적 표기법을 이해한다.

01. 몇 가지 기초 사항들

알고리즘 분석의 필요성

◈ 알고리즘 분석
- 알고리즘이 자원을 얼마나 소모하는 지에 대한 분석
- 자원
  - 소요 시간, 메모리, 통신 대역 등
- 평균적인 경우 or 최악의 경우 분석

◈ 사용하는 알고리즘의 소요 시간이 입력의 크기에 대해 어떤 비율로 비례하는지 안다면 주어진 시간에 요구하는 작업을 완료할 수 있을 지 대략 짐작할 수 있음.

◈ 입력의 크기가 n일 때 최악의 경우
- n^2에 비례하는 시간을 소모하는 알고리즘
- $nlogn$에 비례하는 시간을 소모하는 알고리즘
- n에 비례하는 시간을 소모하는 알고리즘

알고리즘의 수행 시간

◈ 알고리즘의 수행 시간을 좌우하는 기준은 다양하게 잡을 수 있다
  - 예
    - for 루프의 반복횟수, 특정한 행이 수행되는 횟수, 함수의 호출횟수, …
    - 핵심 연산 수행 횟수

◈ 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.

```
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

```
sample4(A[ ], n)
{
        sum ← 0 ;
        for i ← 1 to n
                for j ← 1 to n {
                        k ← A[1 … n]에서 임의로 ⌊n/2⌋ 개를 뽑을 때 이
    들 중 최댓값 ;
                        sum ← sum + k ;
                }
        return sum ;
}
```

✓ n^3에 비례하는 시간이 소요된다.

알고리즘의 수행 시간

```
sample5(A[ ], n)
{
        sum ← 0 ;
        for  i ← 1 to n-1
                for  j ←  i+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.

검색 알고리즘

◈ 순차 검색 알고리즘(의사코드)

```
index seqsearch(int n,                    // 입력(1)
                keytype[] S,              // 입력(2)
                keytype x)                // 입력(3)
{
    location = 0;
    while (location < n && S[location] != x)
       location++;
    if (location >= n)
       location = -1;
    return location;
}
```

- while-루프: 아직 검사할 항목이 있고, x를 찾지 못하였나?
- if-문: 모두 검사하였으나, x를 찾지 못했나?

검색 알고리즘

◈ 순차 검색 알고리즘(의사코드)

- 순차검색 알고리즘으로 키를 찾기 위해서 S에 있는 항목을 몇 개나 검색해야 하는가?
  - 키와 같은 항목의 위치에 따라 다름
  - 최악의 경우: n
  - 평균의 경우: $(n+1)/2$ ← 항목이 배열 안에 있는 경우

- 좀 더 빨리 찾을 수는 없는가?
  - 사실상 더 이상 빨리 찾을 수 있는 알고리즘은 존재하지 않는다.
    - 배열 S에 있는 항목에 대한 정보가 전혀 없는 상황에서, 모든 항목을 검색하지 않고 임의의 항목 x를 항상 찾을 수 있다는 보장이 없음.
  - 만약, 배열 S가 정렬되어 있다는 정보가 존재한다면? → 이진검색 알고리즘 적용가능

검색 알고리즘

◈ 이진 검색 알고리즘(의사코드)

```
index binsearch(int n,                    // 입력(1)
                keytype[] S,              // 입력(2)(정렬된 배열)
                keytype x)                // 입력(3)
{
    low = 0; high = n - 1;
    location = -1;
    while (low <= high && location == -1) {
        mid = ⌊(low + high) / 2⌋;         // 나눗셈 & floor
        if (x == S[mid]) location = mid;
        else if (x < S[mid]) high = mid - 1;
        else low = mid + 1;
    }
    return location;
}
```

- while-루프: 아직 검사할 항목이 있고, x를 찾지 못하였나?

검색 알고리즘

◈ 이진 검색 알고리즘(의사코드)

```
index binsearch(int n, keytype[] S, keytype x)
{
     low = 0; high = n - 1;
     location = -1;
     while (low <= high && location == -1) {
          mid = ⌊(low + high) / 2⌋;                    // 나눗셈 & floor
          if (x == S[mid]) location = mid;
          else if (x < S[mid]) high = mid - 1;
          else low = mid + 1;
     }
     return location;
}
```

[초기입력] n=8 | 2 5 10 11 28 34 47 49 | x=50

low=0, high=7, location=-1 mid=floor((0+7)/2)=3 x == S[3] 누적비교횟수: 1

low=4, high=7, location=-1 mid=floor((4+7)/2)=5 x == S[5] 누적비교횟수: 2

low=6, high=7, location=-1 mid=floor((6+7)/2)=6 x == S[6] 누적비교횟수: 3

low=7, high=7, location=-1 mid=floor((7+7)/2)=7 x == S[7] 누적비교횟수: 4

low=8, high=7, location=-1 ← while 조건문 만족 못함

검색 알고리즘

◈ 이진 검색 알고리즘(의사코드)

- 최악의 경우에 대한 비교 횟수를 (개략적으로) 분석해 본다.

배열의 크기	순차검색	이진검색	비고
n	n	$\lg n + 1$	두 방법 모두 최악의 경우에 대한 비교 횟수임
128	128	8	
1,024	1,024	11	
1,048,576	1,048,576	21	
4,294,967,296	4,294,967,296	33	

검색 알고리즘

◈ 이진 검색 알고리즘(의사코드)

- 최악의 경우에 대한 비교 횟수를 (개략적으로) 분석해 본다.
  - 최악의 경우: x가 배열 S에 저장된 값들보다 클 때 (즉, 검색을 실패할 때)
    - while 문을 수행할 때마다 검색 대상의 크기가 절반으로 감소
  - 배열의 크기가 16라면 → 5번의 비교가 필요하다.
    - $5 = \log_2 16 + 1$
  - 배열의 크기가 32라면 → 6번의 비교가 필요하다.
    - $6 = \log_2 32 + 1$

S[15]	S[23]	S[27]	S[29]	S[30]	S[31]
↑	↑	↑	↑	↑	↑
1st	2nd	3rd	4th	5th	6th

  - 배열의 크기가 64라면 → 7번의 비교가 필요하다.
    - $7 = \log_2 64 + 1$
  - …
  - 배열의 크기가 2^k라면 → $k + 1 (= W)$ 번의 비교가 필요하다.
    이때, $\boldsymbol{W = k + 1 = log_2\, 2^k + 1}$

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
  - $N! = N(N-1)!$

- 예2: 수열의 점화식
  - $a_n = a_{n-1} + 2$

재귀의 예: 피보나치 수열

◈ 피보나찌 (Fibonacci) 수열의 정의

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2}, \text{ for } n \geq 2$$

- 예: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …

레오나르도 피보나치

위키백과, 우리 모두의 백과사전.

레오나르도 피보나치(이탈리아어: Leonardo Fibonacci, 1170년 ~ 1240년-1250년) 또는 **레오나르도피사노** (Leonardo da Pisa, Leonardo Pisano)는 이탈리아의 수학자로 피보나치 수에 대한 연구로 유명하다. 또한 유럽에 아라비아 수 체계를 소개하기도 했다.

재귀의 예: 피보나치 수열

◈ 문제: n번째 피보나찌 수를 구하라.

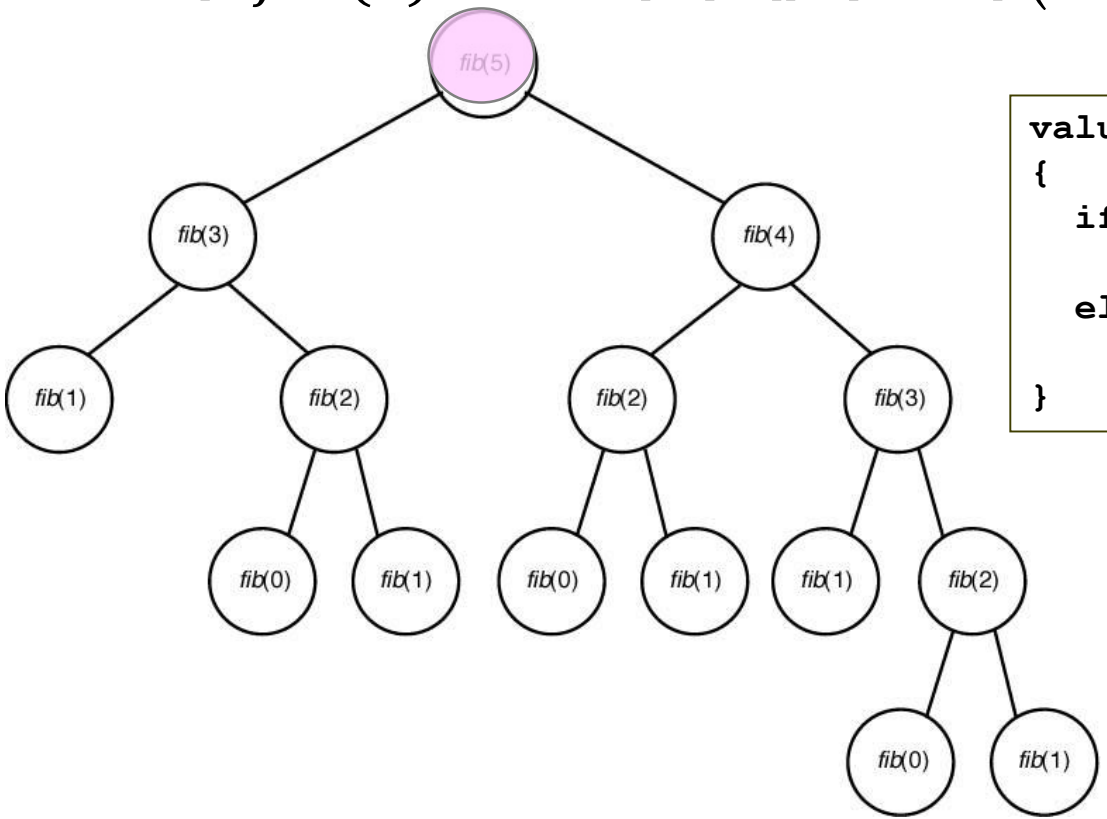
- 입력: 음이 아닌 정수 n
- 출력: n번째 피보나찌 수 (주의: 0번째 부터 시작)

◈ 재귀(recursive) 알고리즘:

```
value fib(index n)
{
  if (n <= 1)
    return n;
  else
    return (fib(n-1) + fib(n-2));
}
```

재귀의 예: 피보나치 수열

- 분석: 피보나찌 수 구하기 재귀 알고리즘은 수행속도가 매우 느리다.
  - 이유: 계산과정에서 동일한 피보나찌 수를 중복하여 계산한다.
  - 예: $fib(5)$ 계산을 위해서는 $fib(2)$를 세 번이나 중복 계산한다.
- 함수 $fib(5)$ 호출 시의 재귀 트리 (recursive tree)




```
value fib(index n)
{
  if (n <= 1)
    return n;
  else
    return (fib(n-1) + fib(n-2));
}
```

실제 호출되는 상황을 봅시다!

재귀의 예: 피보나치 수열

◈ $fib(n)$ 함수 호출 횟수 계산

- $T(n)$: $fib(n)$을 계산하기 위하여 $fib()$ 함수를 호출하는 횟수
- 즉, $T(n)$은 재귀 트리 상의 마디의 개수

$$
\begin{aligned}
&T(0)=1;\\
&T(1)=1;\\
&T(n)=T(n-1)+T(n-2)+1 \quad \text{for } n\geq 2\\
&\quad >2\times T(n-2) \qquad\qquad \text{since } T(n-1)>T(n-2)\\
&\quad >2^2\times T(n-4)\\
&\quad >2^3\times T(n-6)\\
&\quad \cdots\\
&\quad >2^{n/2}\times T(0)\\
&\quad =2^{n/2}
\end{aligned}
$$

피보나치 수열: 재귀 → 반복

◈ 문제: n번째 피보나찌 수를 구하라.

- 입력: 음이 아닌 정수 n
- 출력: n번째 피보나찌 수 (주의: 0번째 부터 시작)

◈ 반복(iterative) 알고리즘:

```
value fib2(index n)
{
  int[] f = new int[n+1];
  f[0] = 0;
  if (n > 0) {
    f[1] = 1;
    for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

피보나치 수열: 재귀 → 반복

◈ 분석: 반복 알고리즘은 수행속도가 훨씬 더 빠르다.
- 이유: 재귀 알고리즘과는 달리 중복 계산이 없다.

◈ $f(i)$ 을 계산하는 연산문 수행수: $T(n)$
- 연산문
  - f[0] = 0
  - f[1] = 1
  - f[i] = f[i−1] + f[i−2]
- $T(n) = n + 1$
  - 즉, f[0]부터 f[n]까지 단 한번씩만 계산한다.

피보나치 수열: 재귀 → 반복

◈ 노드 하나(혹은 연산 하나) 계산에 $1ns$ 걸린다고 가정하자.

n	n+1	$2^{n/2}$	반복 알고리즘	재귀 알고리즘 (하한)
40	41	1,048,576	$41ns$	$1048\mu s$
60	61	1.1×10^{9}	$61ns$	$1s$
80	81	1.1×10^{12}	$81ns$	$18min$
100	101	1.1×10^{15}	$101ns$	$13days$
120	121	1.2×10^{18}	$121ns$	$36years$
160	161	1.2×10^{24}	$161ns$	$3.8\times10^{7}years$
200	201	1.3×10^{30}	$201ns$	$4\times10^{13}years$

재귀 vs. 반복

◈ 재귀 알고리즘 보다는 반복 알고리즘이 항상 효율적이다?
- 많은 경우 그렇다.
- 하지만, 알고리즘 설계 단계에서 재귀 알고리즘은 매우 유용하며 때로는 재귀적으로 설계해도 매우 효율적일 때가 있다
- 대부분의 재귀 알고리즘은 반복 알고리즘으로 변경 가능

◈ 피보나찌 수 구하기의 재귀 알고리즘
- 분할정복(Divide & Conquer)의 전형적인 예이다.

◈ 피보나찌 수 구하기의 반복 알고리즘
- 동적 프로그래밍(Dynamic Programming) (또는 동적 계획법)의 간단한 예이다

재귀를 사용한 분할 정복 예: 병합 정렬

◈ 분할정복(Divide & Conquer)의 예

알고리즘 2-1 병합 정렬

```
mergeSort(A[], p, r)      ▷ A[p ··· r]을 정렬한다.
{
    if (p < r) then {
        ❶ q ← ⌊(p+r)/2⌋;        ▷ p, r의 중간 지점 계산
        ❷ mergeSort(A, p, q);    ▷ 전반부 정렬
        ❸ mergeSort(A, q+1, r);  ▷ 후반부 정렬
        ❹ merge(A, p, q, r);     ▷ 병합
    }
}
```

✓②, ③은 재귀호출
✓①, ④는 재귀적 호출 전후에 수반되는 오버헤드

```
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.
}
```

02. 점근적 표기 (점근적 분석)

알고리즘의 분석

◈ 공간적 효율성(Space Efficiency)과 시간적 효율성 (Time Efficiency)

- 공간적 효율성은 얼마나 많은 메모리 공간을 요하는가를 지칭
- 시간적 효율성은 얼마나 많은 시간을 요하는가를 지칭
- 효율성을 뒤집어 표현하면 복잡도(Complexity)가 된다.
- 즉, 복잡도가 높을수록 효율성은 저하된다.

◈ 일반적으로 시간적 효율성이 공간적 효율성보다 더욱 강조됨

- Why? CPU Cost 가 Memory Cost 보다 비싸기 때문
- 그렇다고, 공간적 효율을 무작정 무시하면 안됨

알고리즘의 분석

◈ 시간적 복잡도 (Time Complexity) 분석

- 하드웨어 환경에 따라 처리시간이 달라진다.
  - CPU 성능 차이
  - GPU (Graphics Processing Unit, 그래픽 처리 장치) 존재유무
  - 곱셈/나눗셈 가속기능 유무
  - 입출력 장비의 성능, 공유여부

- 소프트웨어 환경에 따라 처리시간이 달라진다.
  - 프로그램 언어의 종류, 운영체제, 컴파일러의 종류
  - 운영체제에 현재 어느 정도의 프로세스가 동작하고 어느 정도의 load가 걸리고 있는가?

- 이러한 환경적 차이로 인해 실제 작동시간을 통한 분석 어려움

- 그래서, 실행환경과 무관하게 개략적으로 분석하는 방법 필요

알고리즘의 분석

◈ 그래서… 시간복잡도(Time Complexity) 분석이란?

- 입력 크기에 따라서 단위 연산이 몇 번 수행되는지 결정하는 절차

◈ 복잡도 분석을 위한 주요 요소

- **단위연산(basic operation)**
  - 알고리즘을 수행하는 데 있어서 가장 핵심적인 역할을 담당하는 연산
  - 비교문(comparison)에 있는 비교 연산 또는 지정문(assignment)에 있는 수치연산 후 지정 연산 등…
  - 주관적으로 선택
    - 하지만, 대부분의 전문가들은 동일한 것을 선택하게 되며, 서로 다른 단위연산을 택하더라도 최종적인 복잡도는 동일하게 될 정도로 유사한 중요도를 지닌 단위연산을 택함
- **입력크기(input size)**
  - 배열의 크기, 리스트의 길이, 행렬에서 행과 열의 크기
  - 그래프/트리에서 Vertex와 Edge의 수

알고리즘의 분석 종류

◈ 최악의 경우 분석(Worst-case analysis)

- 입력 크기와 입력 값 모두에 종속
- 단위연산이 수행되는 횟수가 최대(최악)인 경우 선택
- 예: 입력 값이 찾는 대상인 리스트에 존재하지 않을 때

◈ 최선의 경우 분석(Best-case analysis)

- 입력 크기와 입력 값 모두에 종속
- 단위 연산이 수행되는 횟수가 최소(최선)인 경우 선택
- 별로 유용하지 않음

알고리즘의 분석 종류

◈ 평균의 경우 분석(Average-case analysis)

- 입력 크기와 입력 값 모두에 종속
- 모든 입력에 대해서 단위연산이 수행되는 횟수의 기대치(평균)
- 확률적 계산 필요 → 일반적으로 최악의 경우보다 계산이 복잡

◈ 모든 경우 분석(Every-case analysis)

- 입력 값의 내용과 무관(independent)하게 복잡도가 항상 일정한 경우에만 모든 경우 분석이 가능함
  - 즉, 모든 경우 분석이 가능하다면 최악, 최선, 평균의 경우가 모든 경우 분석결과와 동일함
- 복잡도는 입력 크기에만 종속적임

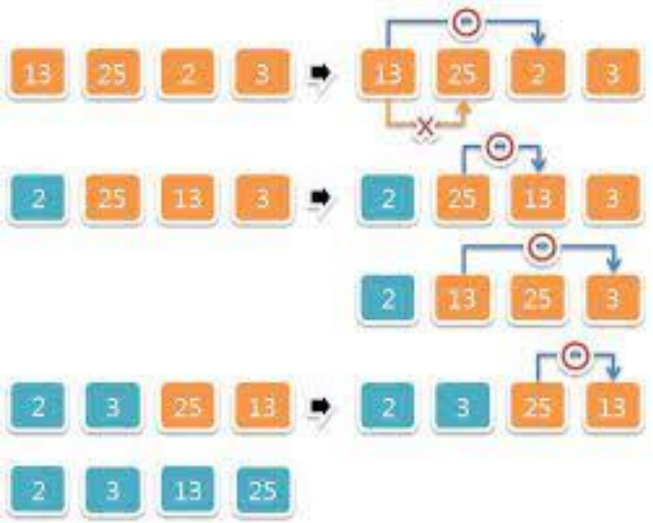
시간 복잡도 분석 예

◈ 교환 정렬 (Exchange Sort)

- 문제: n개의 키를 정렬
- 입력: 양수 n, 배열 $S[1 \ldots n]$
- 출력: 비내림차순으로 정렬된 배열

```
exchangesort (int n, keytype[] S)
{
    index i, j;

    for (i = 1; i <= n-1; i++)
        for (j = i+1; j <= n; j++)
            if (S[j] < S[i])
              exchange S[i] and S[j];
}
```



시간 복잡도 분석 예

```
exchangesort (int n, keytype[] S)
{
      index i, j;

      for (i = 1; i <= n-1; i++)
            for (j = i+1; j <= n; j++)
                  if (S[j] < S[i])
                        exchange S[i] and S[j];
}
```

◈ 교환 정렬 (Exchange Sort)

- 단위연산: 비교문 ($S[j]$와 $S[i]$의 비교)
- 입력크기: 정렬할 항목의 수 n
- 최악의 경우/최선의 경우/평균의 경우 동일 → 모든 경우 분석:
  - j-루프가 수행될 때마다 조건문을 1번씩 수행
  - 조건문의 총 수행횟수
    - ✓ $i = 1$: j-루프 $n-1$ 번 수행
    - ✓ $i = 2$: j-루프 $n-2$ 번 수행
    - ✓ $i = 3$: j-루프 $n-3$ 번 수행
    - ✓ …
    - ✓ $i = n-1$: j-루프 1 번 수행
    - ✓ 따라서

$$T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

시간 복잡도 분석 예

```
exchangesort (int n, keytype[] S)
{
      index i, j;

      for (i = 1; i <= n-1; i++)
            for (j = i+1; j <= n; j++)
                  if (S[j] < S[i])
                        exchange S[i] and S[j];
}
```

◈ 교환 정렬 (Exchange Sort)

- 단위연산: exchange S[i] and S[j];
- 입력크기: 정렬할 항목의 수 n
- 최악의 경우/최선의 경우/평균의 경우 다름:
  - 최악의 경우
    - $T(n) = (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$
  - 최선의 경우
    - $T(n) = 0$
  - 평균의 경우
    - 분석 생략 (확률에 기반에 계산 필요)

따라서, 단위연산 선정 기준에 따라 시간 복잡도 분석 결과가 달라질 수 있음.

하지만, 정렬 알고리즘인 경우 비교연산을 단위연산으로 선정하는 경우가 많음

알고리즘의 점근적 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 (Asymptotic) 분석이라 한다

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

알고리즘의 점근적 분석

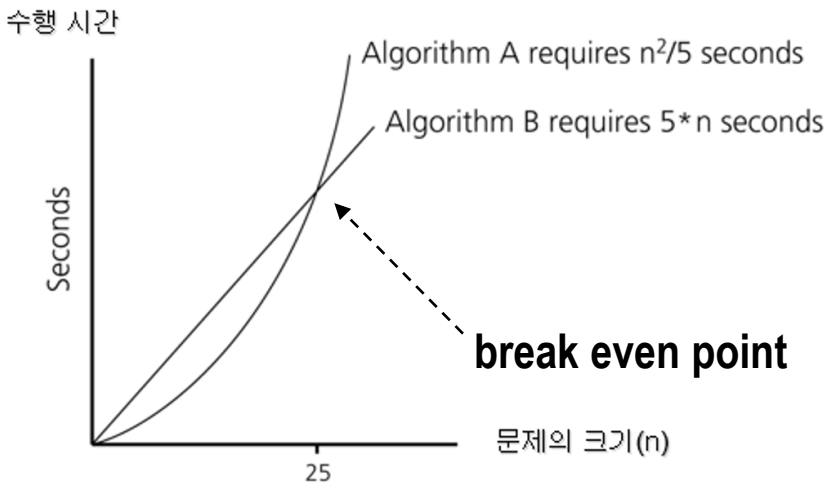
◈ 동일 문제 해결을 위한 알고리즘 A와 B를 생각해 보자
- 아래와 같은 A와 B 의 Complexity라면 A가 당연히 효율적
  - A: n vs. B: n^2
- 그러나, 아래와 같은 A와 B 라면?
  - A: $100n$ vs. B: $0.01n^2$
  - 입력의 크기가 10,000 보다 적으면 알고리즘 A가 좋고 그렇지 않으면 알고리즘 B가 좋다.
  - 그렇다면 어느 알고리즘이 더 좋은 것인가?

◈ Asymptotic Complexity (점근적 복잡도)
- 일반적으로 시간적 효율을 말할 때에는 n이 무한히 큰 경우일 때의 복잡도를 지칭한다.
- 즉, 입력 데이터의 크기 n이 무한대로 갈 때 알고리즘의 실행시간이 어디에 접근하는가?

알고리즘의 점근적 분석

◈ 상수(Constant) vs. 차수(Order)

- Comparing $c_1 n$ with $c_2 n^2$ (c_1 and c_2 are constants)
  - Regardless of c_1 and c_2, there exists a break even point.

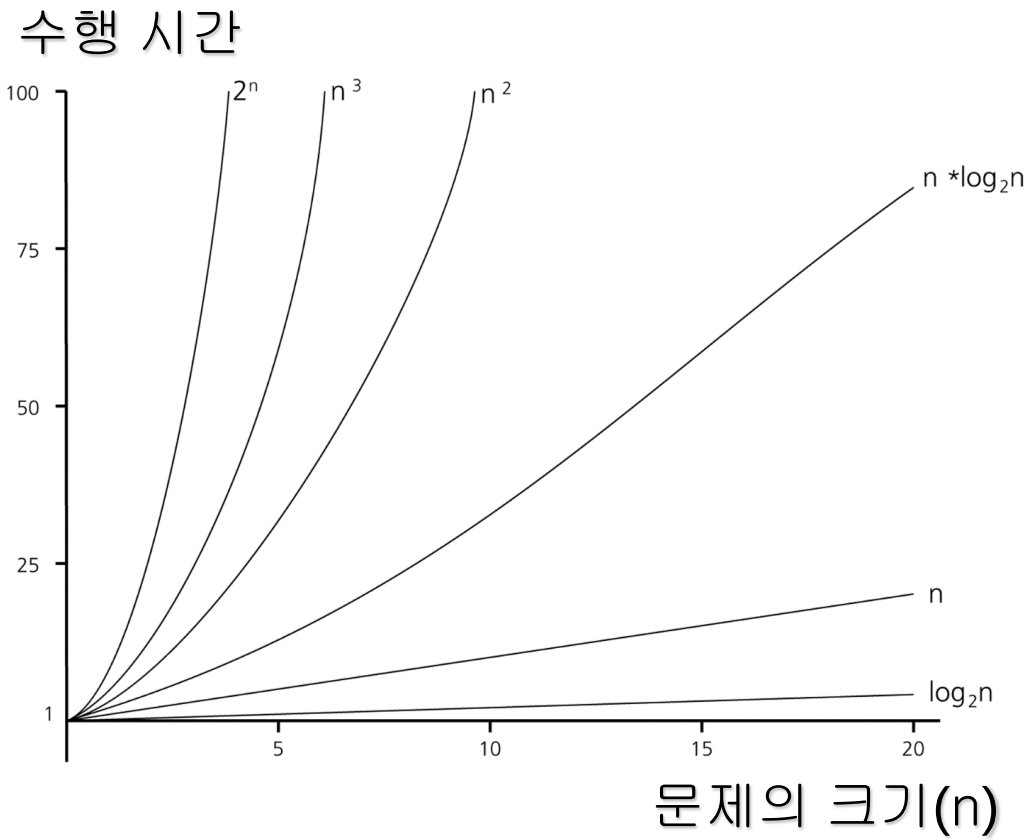


- Consequently (결론적으로)…
  - 차수(Order) is important
  - 상수(Constant) can be negligible
  - 즉, N이 무한대로 갈 때인 Asymptotic Complexity를 기준으로 평가
    - ➢ 입력 데이터가 최악일 때 알고리즘이 보이는 효율 기준

알고리즘의 수행 시간과 복잡도 카테고리

◈ 복잡도 카테고리

- $\Theta(\lg n)$: **로그**(logarithmic)
- $\Theta(n)$: 1**차**(linear)
- $\Theta(n \lg n)$
- $\Theta(n^2)$: 2**차**(quadratic)
- $\Theta(n^3)$: 3**차**(cubic)
- $\Theta(2^n)$: **지수**(exponential)
- $\Theta(n!)$: factorial



알고리즘의 수행 시간과 복잡도 카테고리

◈ 복잡도 카테고리와 수행 시간

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 days	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 days		

[가정] 단위 연산 1회 수행시간 = 10^{-9} second

* 1 μs = 10^{-6} second
† 1 ms = 10^{-3} second

알고리즘의 수행 시간과 복잡도 카테고리

◈ 다음 두 복잡도 식을 가지고 있는 알고리즘은 어느 것이 복잡도가 높은가?

- 알고리즘 A: $0.1n^2$
- 알고리즘 B: $0.1n^2 + n + 100$

◈ 복잡도 수식이 2차 이하의 항으로만 구성된 경우, 2차 항이 궁극적으로 지배한다.

n	$0.1n^2$	$0.1n^2+n+100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

◈ 즉, 위 두 알고리즘은 동일한 복잡도 카테고리에 속한다.

알고리즘의 수행 시간과 복잡도 카테고리

◈ 컴퓨터 과학자들 사이에 고민한 내용…

- 최고차 항의 상수를 무시하고
- 최고차 항보다 작은 차수 항은 무시할 수 있는
- 편리한 점근적 복잡도 표기 방법이 없을까?
- 그래서 창안한 방법 → $O, \Omega, \Theta, \omega, o$ 표기법

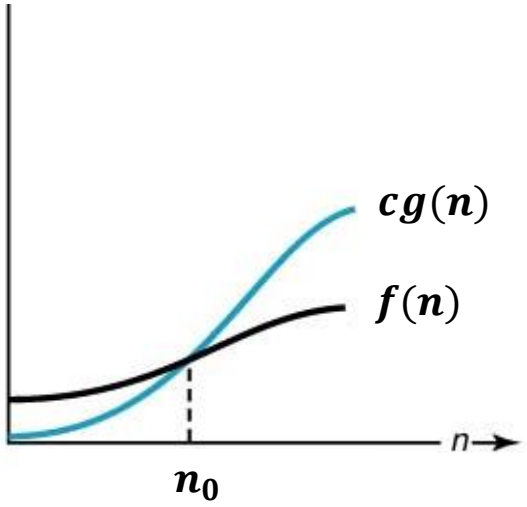
이 알고리즘의 점근적 복잡도는 _____ 이다.

03. 점근적 표기의 엄밀한 정의 ← 공부할 것!

점근법 표기법 Asymptotic Notations: O

◈ $O(g(n))$

- 기껏해야(많아도) $g(n)$의 비율로 증가하는 함수 집합
- e.g., $O(n)$, $O(nlogn)$, $O(n^2)$, $O(2^n)$, …



◈ Formal definition of $O(g(n))$

$$\boldsymbol{O(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \; s.t. \forall n \geq n_0, f(n) \leq cg(n)\}}$$

- $f(n) \in O(g(n))$을 관행적으로 $f(n) = O(g(n))$라고 표기한다.
- $f(n) = O(g(n))$의 직관적 의미
  - n이 커질 때 $f(n)$은 $g(n)$보다 더 높이 증가하지 않고, 아무리 느려도 $c \cdot g(n)$과 같거나 빠르게 수행된다.
  - 상수 비율의 차이는 무시

점근법 표기법 Asymptotic Notations: O

◈ $O(g(n))$ 예시

- $3n^2 + 2n = O(n^2)$
- $7n^2 - 100n = O(n^2)$
- $nlogn + 5n = O(n^2)$
- $3n = O(n^2)$

```
exchangesort (int n, keytype[] S)
{
      index i, j;

      for (i = 1; i <= n-1; i++)
            for (j = i+1; j <= n; j++)
                  if (S[j] < S[i])
                        exchange S[i] and S[j];
}
```

- Exchange Sort 알고리즘의 (점근적) 시간복잡도: $O(n^2)$

◈ 알 수 있는 한 최대한 tight 하게

- $nlogn + 5n = O(nlogn)$ 이므로 굳이 $O(n^2)$으로 표기할 필요 없다.
- 엄밀하지 않은 만큼 정보의 손실이 일어난다.

점근법 표기법 Asymptotic Notations: O

$$\boldsymbol{O(g(n)) = \{f(n) | \exists c > 0, n_0 > 0 \; s.t. \forall n \geq n_0, f(n) \leq cg(n)\}}$$

◈ [**예제** 2-1] $5n^2 = O(n^2)$ **임을 보여라**.

$c = 6$**와** $n_0 = 1$**을 선택하면, 모든** $n \geq n_0$**에 대하여** $5n^2 \leq 6n^2$**이 성립한다. 즉, 정의를 만족하는 상수** c**와** n_0**가 존재한다**.

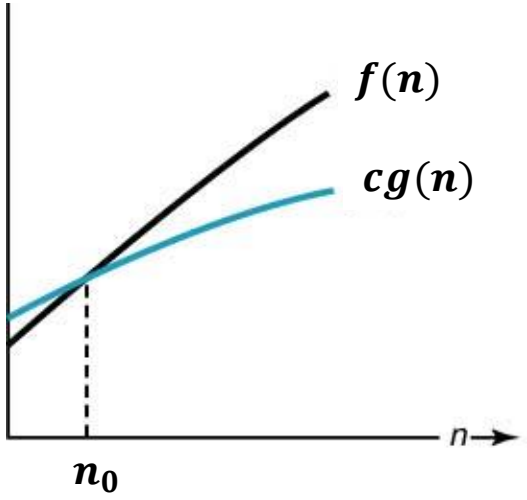
◈ [**예제** 2.3] $\frac{n^2}{2} - 5 = O(n^2)$ **임을 보여라**.

$c = 1$**와** $n_0 = 1$**을 선택하면, 모든** $n \geq n_0$**에 대하여** $\frac{n^2}{2} - 5 \leq n^2$ $(\because -5 \leq \frac{n^2}{2})$ **이 성립한다. 즉, 정의를 만족하는 상수** c**와** n_0**가 존재한다**.

점근법 표기법 Asymptotic Notations: Ω

◈ $\Omega(g(n))$

- 최소한(적어도) $g(n)$의 비율로 증가하는 함수 집합
- e.g., $\Omega(n)$, $\Omega(nlogn)$, $\Omega(n^2)$, $\Omega(2^n)$, …



◈ Formal definition of $\Omega(g(n))$

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

- $f(n) \in \Omega(g(n))$을 관행적으로 $f(n) = \Omega(g(n))$라고 표기한다.
- $f(n) = \Omega(g(n))$의 직관적 의미
  - n이 커질 때 $f(n)$은 $g(n)$보다 더 높이 증가하고, 아무리 빨라도 $c \cdot g(n)$과 같거나 느리게 수행된다.
  - 상수 비율의 차이는 무시

점근법 표기법 Asymptotic Notations: Ω

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

◈ [**예제** 2-6] $5n^2 + 3 = \Omega(n^2)$ **임을 보여라**.

$c = 1$**와** $n_0 = 1$**을 선택하면, 모든** $n \geq n_0$**에 대하여** $n^2 \leq 5n^2 + 3$ ($\because -3 \leq 4n^2$)**이 성립한다. 즉, 정의를 만족하는 상수** c**와** n_0**가 존재한다.**

◈ [**예제** 2-8] $5n^3 + 3 = \Omega(n^2)$ **임을 보여라**.

$c = 1$**와** $n_0 = 1$**을 선택하면, 모든** $n \geq n_0$**에 대하여** $n^2 \leq 5n^3 + 3$**이 명백하게 성립한다. 즉, 정의를 만족하는 상수** c**와** n_0**가 존재한다.**

점근법 표기법 Asymptotic Notations: Θ

◈ $\Theta(g(n))$

- $g(n)$과 동일한 비율로 증가하는 함수 집합
- e.g., $\Theta(n)$, $\Theta(nlogn)$, $\Theta(n^2)$, $\Theta(2^n)$, …

◈ Formal definition of $\Theta(g(n))$

$$\boldsymbol{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$$

$$\boldsymbol{\Theta(g(n)) = \{f(n) | \exists c_1, c_2 > 0, n_0 > 0 \; s.t. \; \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}}$$

- $f(n) \in \Theta(g(n))$을 관행적으로 $f(n) = \Theta(g(n))$라고 표기한다.
- $f(n) = \Theta(g(n))$의 직관적 의미
  - n이 커질 때 $f(n)$과 $g(n)$은 같은 정도로 증가한다.
  - 상수 비율의 차이는 무시
  - $f(n)$는 $g(n)$과 동일한 차수(order) → 일반적으로 $\Theta(g(n))$는 복잡도 카테고리를 나눌 때 사용한다.

점근법 표기법 Asymptotic Notations: Θ

$$\boldsymbol{\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))}$$

◈ [**예제** 2-10] $5n^2 + 3 = \Theta(n^2)$ **임을 보여라**.

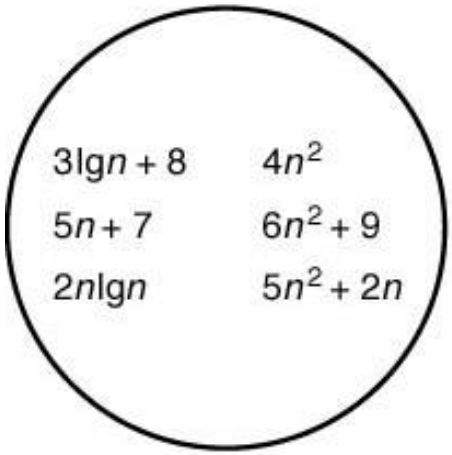
[**예제** 2-2]**와** [**예제** 2-6]**에서** $5n^2 + 3 = O(n^2)$ **및** $5n^2 + 3 = \Omega(n^2)$ **임을 각각 증명하였다. 그러므로,** $5n^2 + 3 = \Theta(n^2)$ **이다**.

◈ [**예제** 2-11] $\frac{n^2}{2} - 5 = \Theta(n^2)$ **임을 보여라**.

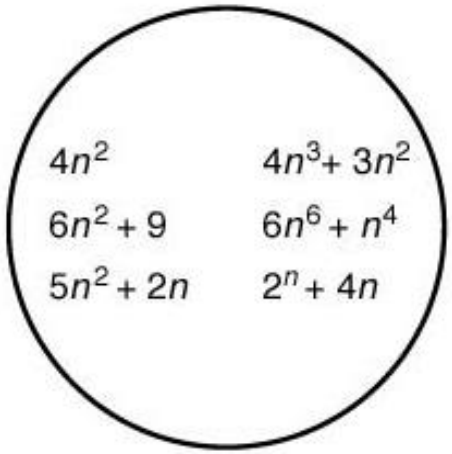
[**예제** 2-3]**과** [**예제** 2-7]**에서** $\frac{n^2}{2} - 5 = O(n^2)$ **및** $\frac{n^2}{2} - 5 = \Omega(n^2)$ **임을 각각 증명하였다. 그러므로,** $\frac{n^2}{2} - 5 = \Theta(n^2)$ **이다**.

점근법 표기법 Asymptotic Notations

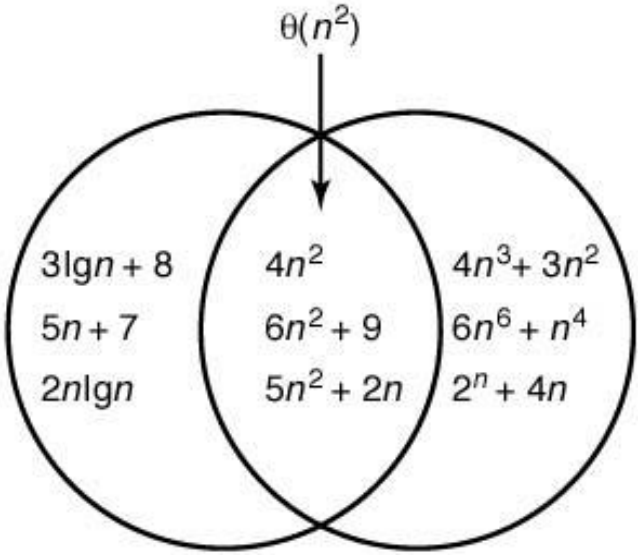
◈ O, Ω, Θ간의 관계



$O(n^2)$

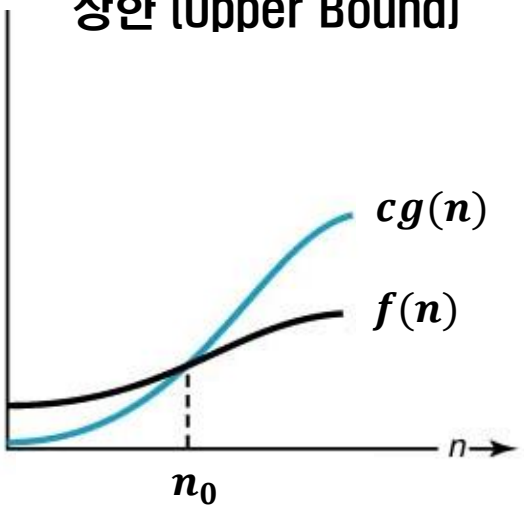


$\Omega(n^2)$

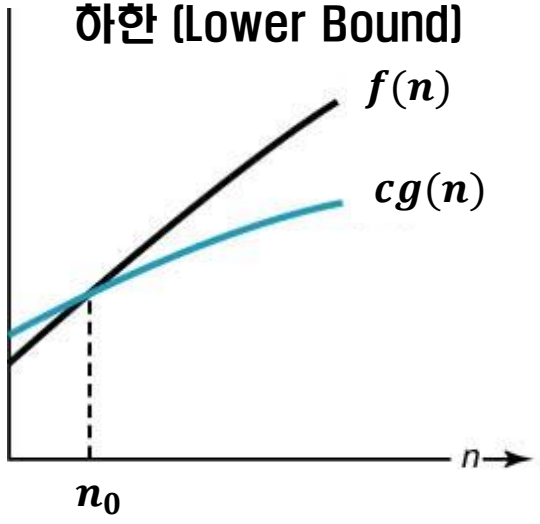


$\Theta(n^2)=O(n^2) \cap \Omega(n^2)$

상한 (Upper Bound)



하한 (Lower Bound)



점근법 표기법 Asymptotic Notations

◈ $o(g(n))$

- $g(n)$보다 느린 비율로 증가하는 함수 집합

◈ Formal definition of $o(g(n))$

$$\boldsymbol{o(g(n)) = \{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\}}$$

- $f(n) \in o(g(n))$을 관행적으로 $f(n) = o(g(n))$라고 표기한다.

- $f(n) = o(g(n))$의 직관적 의미
  - n이 커질 때 $f(n)$은 $g(n)$보다 느리게 증가한다.
  - 상수 비율의 차이는 무시
  - 예: $n = o(n^2)$

점근법 표기법 Asymptotic Notations

◈ $w(g(n))$

- $g(n)$보다 빠른 비율로 증가하는 함수 집합

◈ Formal definition of $w(g(n))$

$$\boldsymbol{w(g(n)) = \{f(n) | \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty\}}$$

- $f(n) \in w(g(n))$을 관행적으로 $f(n) = w(g(n))$라고 표기한다.

- $f(n) = w(g(n))$의 직관적 의미
  - n이 커질 때 $f(n)$은 $g(n)$보다 빠르게 증가한다.
  - 상수 비율의 차이는 무시

복잡도 카테고리의 주요 성질 (1/3)

- 1. $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$
- 2. $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$

- 3. $b > 1$이고 $a > 1$이면, $\log_a n \in \Theta(\log_b n)$은 항상 성립한다.
  → 즉, 임의의 밑에 대한 로그(logarithm) 복잡도 함수는 모두 같은 카테고리에 속한다.

- 4. 지수(exponential) 복잡도 함수 (예 2^n)의 밑이 다르면 이들은 모두 다른 복잡도 카테고리에 속한다.
  - 즉, 임의의 서로 다른 a와 b에 대해서 $a^n \notin \Theta(b^n)$ 이다.
  - 만약 $b > a > 0$ 이면, $a^n \in o(b^n)$ (← Small o)

복잡도 카테고리의 주요 성질 (2/3)

- 5. $n!$은 어떤 지수 (a^n)복잡도 함수보다도 복잡도가 더 높다.
  - ✓ $a^n \in o(n!)$ (← Small o)

- 6. 복잡도 카테고리들은 다음 순서로 나열된다.

$$\Theta(\lg n), \Theta(n), \Theta(n \lg n), \Theta(n^2), \Theta(n^j), \Theta(n^k), \Theta(a^n), \Theta(b^n), \Theta(n!)$$

  여기서 $k>j>2$이고 $b>a>1$이다.

- 7. $c \geq 0$, $d>0$, $f(n) \in O(g(n))$, $h(n) \in \Theta(g(n))$이면, 다음이 성립한다.

$$c \times f(n) + d \times h(n) \in \Theta(g(n))$$

복잡도 카테고리의 주요 성질 (3/3)

- $\Theta(\log_4 n) = \Theta(\log_2 n)$?
  - ***복잡도 카테고리의 주요 성질 3 번 사용***

- $5n + 3\lg n + 10n\lg n + 7n^2 \in \Theta(n^2)$?
  - ***복잡도 카테고리의 주요 성질 6 과 7 을 사용***

최종 정리!!! (1/2)

- 다음 두 복잡도 식을 가지고 있는 알고리즘의 복잡도 표기법은?
  - 알고리즘 A: $0.1n^2$
  - 알고리즘 B: $0.1n^2+n+100$
  - 알고리즘 A, B 둘 다 $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$ 로 표기가능…
    하지만, 각각의 뉘앙스(의미) 및 해석은 다르다.
    - 어떻게 다를까?...

- 모든 경우분석이 가능한 알고리즘에 대한 단위연산 기반 복잡도가 위와 같다면 해당 알고리즘의 점근적 복잡도는 $\Theta(n^2)$ 로 표기한다.
  - 즉, 최선의 경우와 최악의 경우가 구분이 안될 때…

- 모든 경우 분석이 되지 않는 경우에는 최악의 분석을 수행하고, 최악의 분석 결과가 위와 같은 단위연산 기반 복잡도로 나왔다면 점근적 복잡도는 $O(n^2)$
  - 많은 경우 최악의 경우와 최선의 경우는 나뉘어짐.
  - 즉, $O(n^2)$이 가장 많이 사용되는 점근적 복잡도 표기법임

최종 정리!!! (2/2)

- 만약 알고리즘 분석 결과 최악의 경우 $W(n) = n^2$ 이 나왔다면, 해당 알고리즘의 점근적 복잡도는 $O(n^2)$ 이라고 결론 내림.
  - $W(n) = n^2$ 라는 수식이 해당 알고리즘을 분석한 결과 최악의 경우에 나온 수식이며, 임의의 양의 실수 c $(c = 1)$와 음이 아닌 정수 N $(N = 0)$ 및 모든 정수 $n \geq N$에 대하여 $n^2 \leq c \cdot n^2$ 이므로 해당 알고리즘의 점근 적 복잡도는 $O(n^2)$ 는 이다"라고 최종적인 결론을 내린다.

- **만약 알고리즘 분석 결과 모든 경우 $T(n) = n^2$ 이 나왔다면, 해당 알고리즘의 점근적 복잡도는 $\Theta(n^2)$ 이라고 결론 내림. (하지만 가끔은 이런 경우에도 점근적 복잡도를 $O(n^2)$ 로 표기하는 경우가 있음.)**

생각해봅시다.

time(n) = 3 F(n) - 2	is O(F(n))
time(n) = 2 n	is O(n)
time(n) = 4687 n	is O(n)
time(n) = 1,76*10^{25}	is O(1)

- *f* is *O(g)* is transitive
  - If *f* is *O(g)* and *g* is *O(h)* then *f* is *O(h)*
- Product of upper bounds is upper bound for the product
  - If *f* is *O(g)* and *h* is *O(r)* then *fh* is *O(gr)*

생각해봅시다.

- Simple statement sequence

  s_1; s_2; …. ; s_k

  – *O*(1) as long as *k* is constant

- Simple loops

  `for(i=0;i<n;i++) { s; }`

  where `s` is *O*(1)

  – Time complexity is *n O*(1) or *O*(*n*)

- Nested loops

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++) { s; }
```

  – Complexity is *n O*(*n*) or $O(n^2)$

생각해봅시다.

- Loop index doesn't vary linearly

```
h = 1;
while ( h <= n ) {
      s;
      h = 2 * h;
      }
```

  - h takes values 1, 2, 4, … until it exceeds *n*
  - There are 1 + $\log_2 n$ iterations
  - Complexity *O*(log *n*)

생각해봅시다.

- Loop index depends on outer loop index

```
for(j=0;j<n;j++)
    for(k=0;k<j;k++){
      s;
      }
```

  - Inner loop executed
    - 1, 2, 3, …., n times

∴ Complexity $O(n^2)$

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

생각해봅시다.

```
get(x);
get(y);
z:=x+y;
put(z);
```

$f(n)=c \rightarrow O(1)$

```
for i in 1..n loop
  z(i):=x(i)+y(i);
end loop;
```

$f(n)=cn \rightarrow O(n)$

```
for i in 1..n loop
  s(i):=0;          c
  for j in 1..n loop
    s(i):=s(i)+x(i,j);
  end loop;
end loop;           dn
```

$f(n)=n(c+dn) \rightarrow O(n^2)$

```
x:=rnd(100);
if x>90 then
  for i in 1..n loop
    x:=x+z(i);          10%
  end loop;
else
  x:=0;                 90%
end if;
```

$f(n)=(0.1)*cn+(0.9)d \rightarrow O(n)$

```
for i in 1..n loop
  s:=s+x(i);
end loop;
for i in 1..m loop
  t:=t+y(i);
end loop;
```

$f(n,m)=cn+dm \rightarrow O(n+m)$