

제2장 알고리즘을 배우기 위한 준비

2.1 알고리즘이란

정의

- 알고리즘은 ^①문제를 해결하는 단계적 절차 또는 방법^②
 - 주어지는 문제는 컴퓨터를 이용하여 해결할 수 있어야 함
 - 입력이 주어지고, 수행한 결과인 해를 출력



요건

- 정확성(Correctness)
 - 알고리즘은 주어진 입력에 대해 올바른 해를 주어야 한다.
- 명확성(Definiteness, 수행성)
 - 알고리즘의 각 단계는 컴퓨터에서 수행 가능하여야 한다.
 - 명확하게 표현되어야 함. 모호성 배제
- 유한성(Finiteness)
 - 알고리즘은 일정한 시간 내에 종료되어야 한다.
- 효율성(Efficiency)
 - 알고리즘은 효율적(시간상 혹은 공간상)일수록 그 가치가 높아진다.
 - 구해진 해가 유효해야 함.

2.2 최초의 알고리즘

- 가장 오래된 알고리즘: 기원전 300년경 유클리드 (Euclid)의 최대공약수 알고리즘
 - 최대공약수는 2개 이상의 자연수의 공약수들 중에서 가장 큰 수
 - 유클리드는 2개의 자연수의 최대공약수는 **큰 수에서 작은 수를 뺀 수**와 **작은 수**와의 최대공약수와 같다는 성질을 이용

최대공약수(24, 14)

$$\begin{aligned} &= \text{최대공약수}(24-14, 14) &= \text{최대공약수}(10, 14) \\ &= \text{최대공약수}(14-10, 10) &= \text{최대공약수}(4, 10) \\ &= \text{최대공약수}(10-4, 4) &= \text{최대공약수}(6, 4) \\ &= \text{최대공약수}(6-4, 4) &= \text{최대공약수}(2, 4) \\ &= \text{최대공약수}(4-2, 2) &= \text{최대공약수}(2, 2) \\ &= \text{최대공약수}(2-2, 2) &= \text{최대공약수}(2, 0) \\ &= 2 \end{aligned}$$

2.3 알고리즘의 표현 방법

Algorithm의 표현

- ① 자연어(Natural Language)
- ② Algorithm 언어 의사 코드 (pseudo code) ← 목적언어
- ③ Flow chart
- ④ 특정 Program언어(C, C++, Java 등)

Algorithm 예 (Euclidean Algorithm)

1) 자연어로 표현 //

step1. Divide the larger number by the smaller number, and get the remainder

step2. If the remainder is zero, then the smaller number is GCD and STOP.

otherwise, go to step 3

step3. Replace the larger number by the smaller,
and the smaller by the remainder.

Go to Step1.

2) Algorithm언어(의사코드)로 표현 //

step1. Remainder \leftarrow Larger MOD Smaller

step2. If [Remainder=0] THEN GCD \leftarrow Smaller and STOP

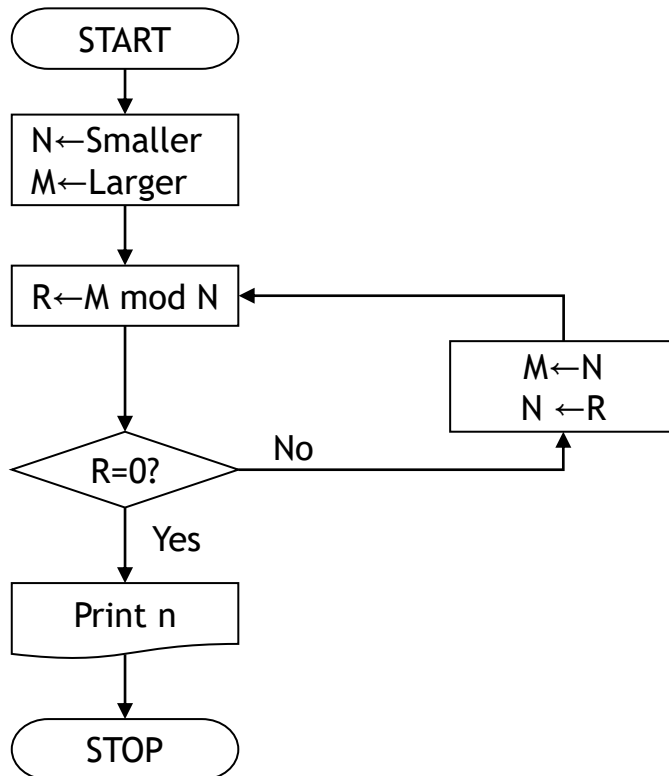
step3. Larger \leftarrow Smaller

Smaller \leftarrow Remainder

Go To Step 1.

Algorithm 예

3) Flow Chart 방법 //



4) Program 언어(c언어) //

```
# include <stdio.h>
# include <math.h>
void main(void)
{
    int m,n,r;
    scanf("%d%d",&m,&n);
    start:
        r=fmod(n,m);
        if(r==0){
            printf("GCD is %d\n",n);
        }
        else {
            m=n;
            n=r;
            goto start;
        }
}
```

알고리즘의 구조화

step1. $\text{Remainder} \leftarrow \text{Larger} \bmod \text{Smaller}$
step2. If $[\text{Remainder}=0]$ THEN **GCD** \leftarrow Smaller and STOP
step3. $\text{Larger} \leftarrow \text{Smaller}$
 $\text{Smaller} \leftarrow \text{Remainder}$
 Go To Step 1.



```
Euclid(Larger, Smaller)
{
    if (Smaller=0) return Larger
    return Euclid(Smaller, Larger mod Smaller)
}
```

알고리즘 수행 예

```
Euclid(Larger, Smaller)
{
    if (Smaller=0) return Larger
    return Euclid(Smaller, Larger mod Smaller)
}
```

- **Euclid(24, 14)**
 - Line 1: smaller=14이므로 if-조건 (**Smaller=0**) 이 '거짓'
 - Line 2: Euclid(14, 24 mod 14) = Euclid(14, 10) 호출
 - Line 1: smaller =10이므로 if-조건이 '거짓'
 - Line 2: Euclid(10, 14 mod 10) = Euclid(10, 4) 호출
 - Line 1: smaller =4이므로 if-조건이 '거짓'
 - Line 2: Euclid(4, 10 mod 4) = Euclid(4, 2) 호출
 - Line 1: smaller =2이므로 if-조건이 '거짓'
 - Line 2: Euclid(2, 4 mod 2) = Euclid(2, 0) 호출
 - Line 1: smaller =0이므로 if-조건이 '참'이 되어 a=2를 최종적으로 리턴

최대 숫자 찾기 문제를 위한 알고리즘

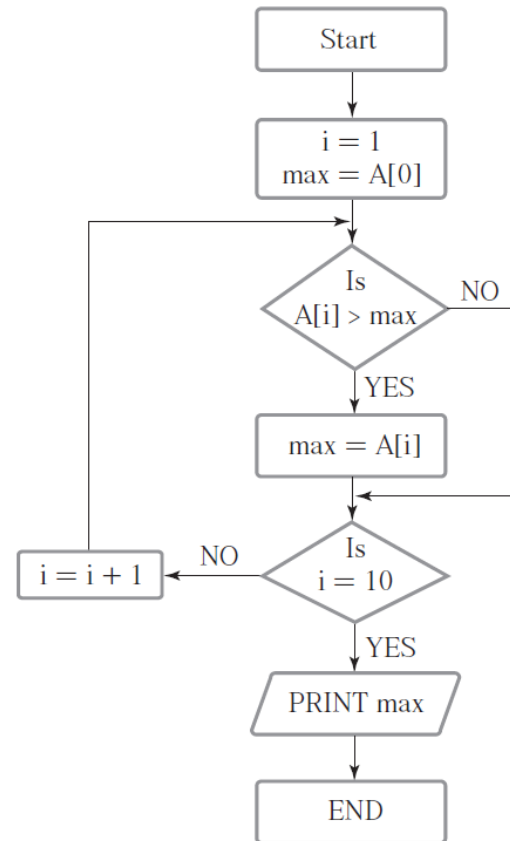
보통 말(자연어)로 표현된 알고리즘:

1. 첫 카드의 숫자를 읽고 머릿속에 기억해 둔다.
2. 다음 카드의 숫자를 읽고, 그 숫자를 머릿속의 숫자와 비교한다.
3. 비교 후 큰 숫자를 머릿속에 기억해 둔다.
4. 다음에 읽을 카드가 남아있으면 2~3을 반복
5. 머릿속에 기억된 숫자가 최대 숫자이다.



의사 코드로 표현된 알고리즘:

배열 A에 입력이 10개의 숫자가 있다고 가정

1. $\text{max} = A[0]$
2. for $i = 1$ to 9
3. if $(A[i] > \text{max})$ $\text{max} = A[i]$
4. return max



2.4 알고리즘의 분류

- 문제의 해결 방식에 따른 분류(일반적인 분류)
 - 분할 정복 (Divide-and-Conquer) 알고리즘 (제3장)
 - 그리디 (Greedy) 알고리즘 (제4장)
 - 동적 계획 (Dynamic Programming) 알고리즘 (제5장) 
 - 근사 (Approximation) 알고리즘 (제8장)
 - 백트래킹 (Backtracking) 기법 (제9장)
 - 분기 한정 (Branch-and-Bound) 기법 (제9장) 

- **문제에 기반한 분류**(특정 문제 해결)
 - 정렬 알고리즘 (제6장)
 - 그래프 알고리즘
 - 기하 알고리즘
- **특정 환경에 따른 분류**(컴퓨팅 환경)
 - 병렬 (Parallel) 알고리즘
 - 분산 (Distributed) 알고리즘
 - 양자 (Quantum) 알고리즘
- **기타 알고리즘들:**
 - 확률 개념이 사용되는 랜덤 (Random) 알고리즘
 - 유전(Genetic) 알고리즘 (제9장)

2.5 알고리즘의 효율성 표현

- 알고리즘의 효율성

- 알고리즘의 수행 시간

- 시간복잡도 (time complexity)

- 알고리즘이 수행하는 동안 사용되는 메모리 공간의 크기 등

- 공간복잡도 (space complexity)

- 일반적으로 시간복잡도가 주로 사용됨

- 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

- n장의 카드가 있다면, (n-1)번의 비교 수행: 시간복잡도는 (n-1) → **n에 비례한다**

Algorithm 복잡도의 의미

Computation time에 의한 측정의 비효율성

- ① Computer 기종마다 능력의 차
- ② program 작성자의 능력의 차
- ③ 측정을 위해 많은 시간 소요(입력사이즈가 큰 경우)

=> Complexity = Amount of computation(계산량)

- 기본연산의 횟수
 - 연산의 회수가 데이터 사이즈에 따라 달라짐
- 보조연산은 무시
 - 초기값 설정, 반복횟수 계산, while문의 조건의 판단, 배열의 첨자 계산 등...

알고리즘의 복잡도를 표현 방법

- 알고리즘의 복잡도 분석 방법
 - 최선 경우 분석 (best case analysis)
 - 평균 경우 분석 (average case analysis) → 사용 X
 - 최악 경우 분석 (worst case analysis)

집에서 강의실까지 가는데 걸리는 시간의 예(전철 간격이 4분이라 하자)

최선 경우



최악 경우



평균 경우



Complexity의 표현

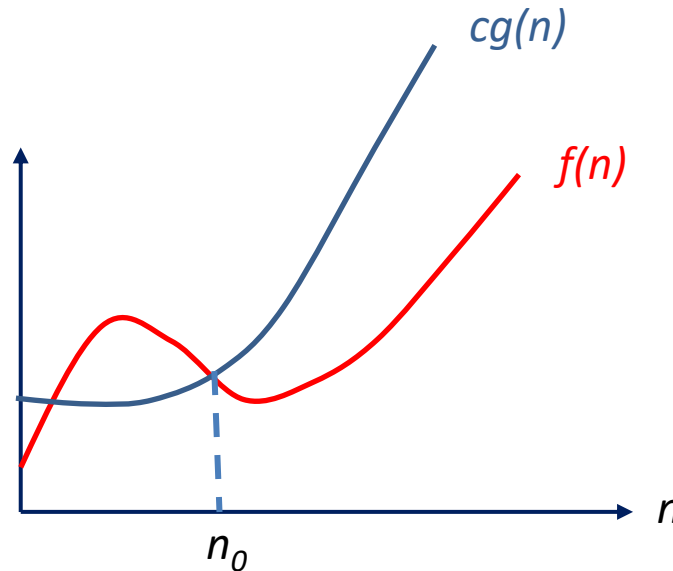
- Best case
 - 타당성(validity)이 부족
 - 예) 탐색 알고리즘: $B(n) = 1$ (n 에 관계 없이)
- Average case
 - 타당성은 있음
 - 계산이 어렵다.
- Worst case
 - 데이터의 사이즈(개수)에 따라 달라짐
 - Average case와 선형적으로 비례하는 성질이 있음
- Complexity
 - 알고리즘을 평가하는 기본 수단
 - Algorithm을 수행할 때 **최악**의 경우에 수행되는 기본 연산 횟수로 표현함

2.6 복잡도의 점근적 표기

- 시간 (또는 공간)복잡도는 입력 크기에 대한 함수로 표기
- 단순한 함수로 표현하기 위해
 - 점근적 표기 (Asymptotic Notation)를 사용
 - 입력 크기 n 이 무한대로 커질 때의 복잡도를 간단히 표현하기 위해 사용
- O (Big-Oh) -표기
- Ω (Big-Omega) -표기
- Θ (Theta) -표기

O(Big-Oh)-표기

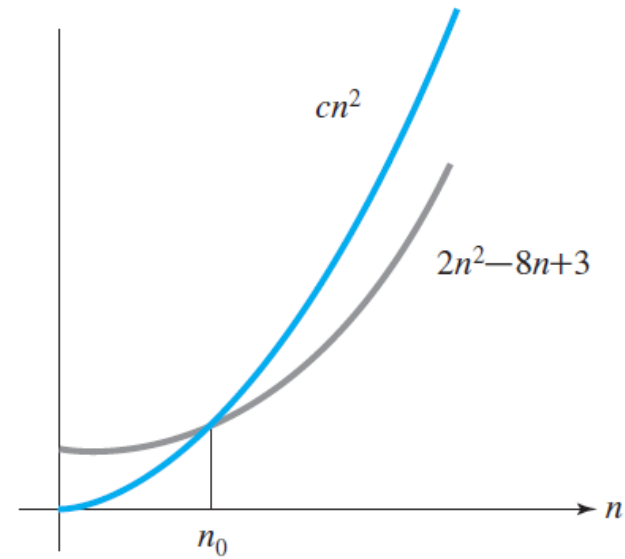
- 복잡도 $f(n)$ 과 O-표기를 그래프로 나타내면 다음과 같음.



- n_0 이 증가함에 따라 $O(g(n))$ 이 점근적 상한
 - $n > n_0$ 에 대하여 $cg(n) > f(n)$ 을 만족하는 c 와 n_0 가 존재하면
 - $f(n) = O(g(n))$ 이라 함
- $O(g(n))$ 이란
 - 충분히 큰 n 에 대하여 $g(n)$ 에 상수만 곱하면 $g(n)$ 이 누를 수 있는 모든 함수들의 집합

O(Big-Oh)의 예

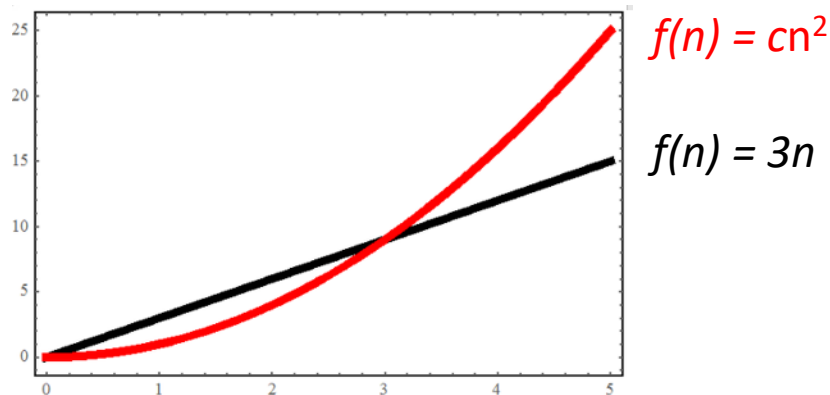
- 복잡도가 $f(n) = 2n^2 - 8n + 3$ 라고 하면
- 교차점 이후의 모든 n 에 대하여
 - $cn^2 > f(n)$ 을 만족하는 c 가 존재. 단, $c > 0$
- 다항식의 최고차 항만 계수 없이 취하면 됨
- 따라서 $f(n)$ 의 O-표기는 $O(n^2)$



- 따라서
- $O(n^2) = \{2n^2 + 3n - 2, 5n + 3, n \log n, \log n, \dots\}$
 - $O(g(n))$ 은 $g(n)$ 보다 증가율이 작은 **모든** 함수들의 집합이다.
 - $O(g(n))$ 은 $f(n)$ 의 최악의 경우임
- $5n + 3$ 은 $O(n^2)$ 이자 $O(n)$
 - **그러나 우리는** $O(n^2)$ 보다는 작은 증가율 함수인 $O(n)$ 이라 표기함
 - $cn > n$ 인 c 가 존재 하므로

o (Small-Oh 혹은 Little-Oh)-표기

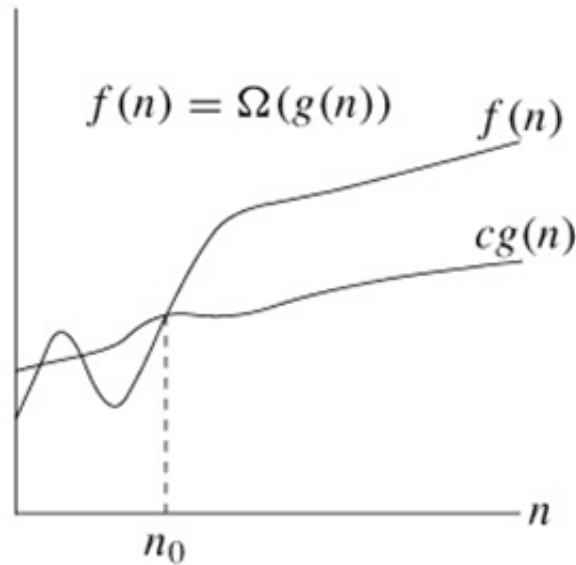
- $n > n_0$ 에 대하여 $cg(n) > f(n)$ 이 모든 c 에 대하여 만족하면
 - $f(n) = o(g(n))$ 이라 함
 - 예를 들어 $o(n^2) = \{\cancel{2n^2+3n-2}, 3n, n \log n, \log n, \dots\}$



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

Ω (Big-Omega)-표기

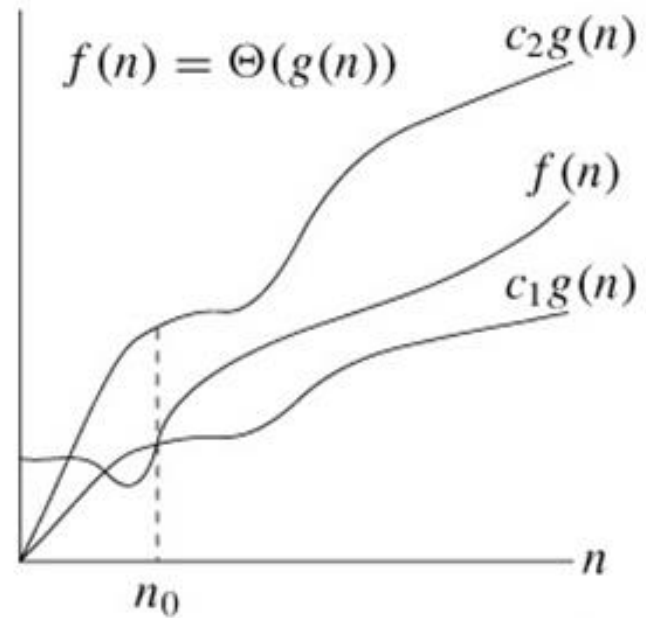
- 복잡도의 점근적 하한을 의미
 - $f(n) = 2n^2 - 8n + 3$ 의 Ω -표기는 $\Omega(n^2)$ 이다.
 - $f(n) = \Omega(n^2)$ 은 "n이 증가함에 따라 $2n^2 - 8n + 3 > cn^2$ 을 만족하는 c가 존재
 - 예를 들어 $c=1$
- O-표기 때와 마찬가지로, Ω -표기도 복잡도 다항식의 최고차항만 계수 없이 취하면 된다.



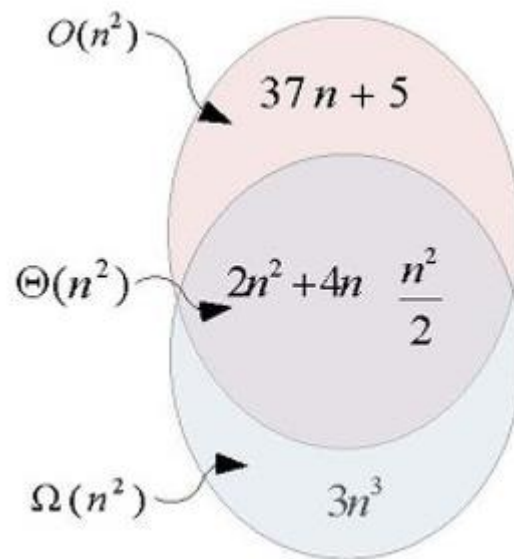
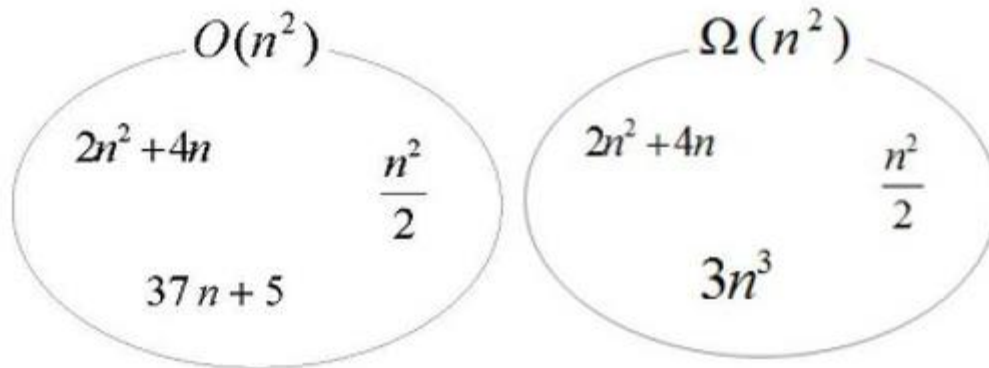
- 복잡도 $f(n)$ 과 Ω -표기를 그래프로 나타낸 것인데, n 이 증가함에 따라 $\Omega(g(n))$ 이 점근적 하한
 - $g(n)$ 이 n_0 보다 큰 모든 n 에 대해서 항상 $f(n)$ 보다 작음

Θ (Theta)-표기

- O -표기와 Ω -표기가 같은 경우에 사용
 - $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$
 - $f(n) = 2n^2 + 10n + 3$ 을 생각해 보자
 - $2n^2 + 10n + 3 = O(n^2)$
 - $2n^2 + 10n + 3 = \Omega(n^2)$
 - $\rightarrow f(n) = \Theta(n^2)$



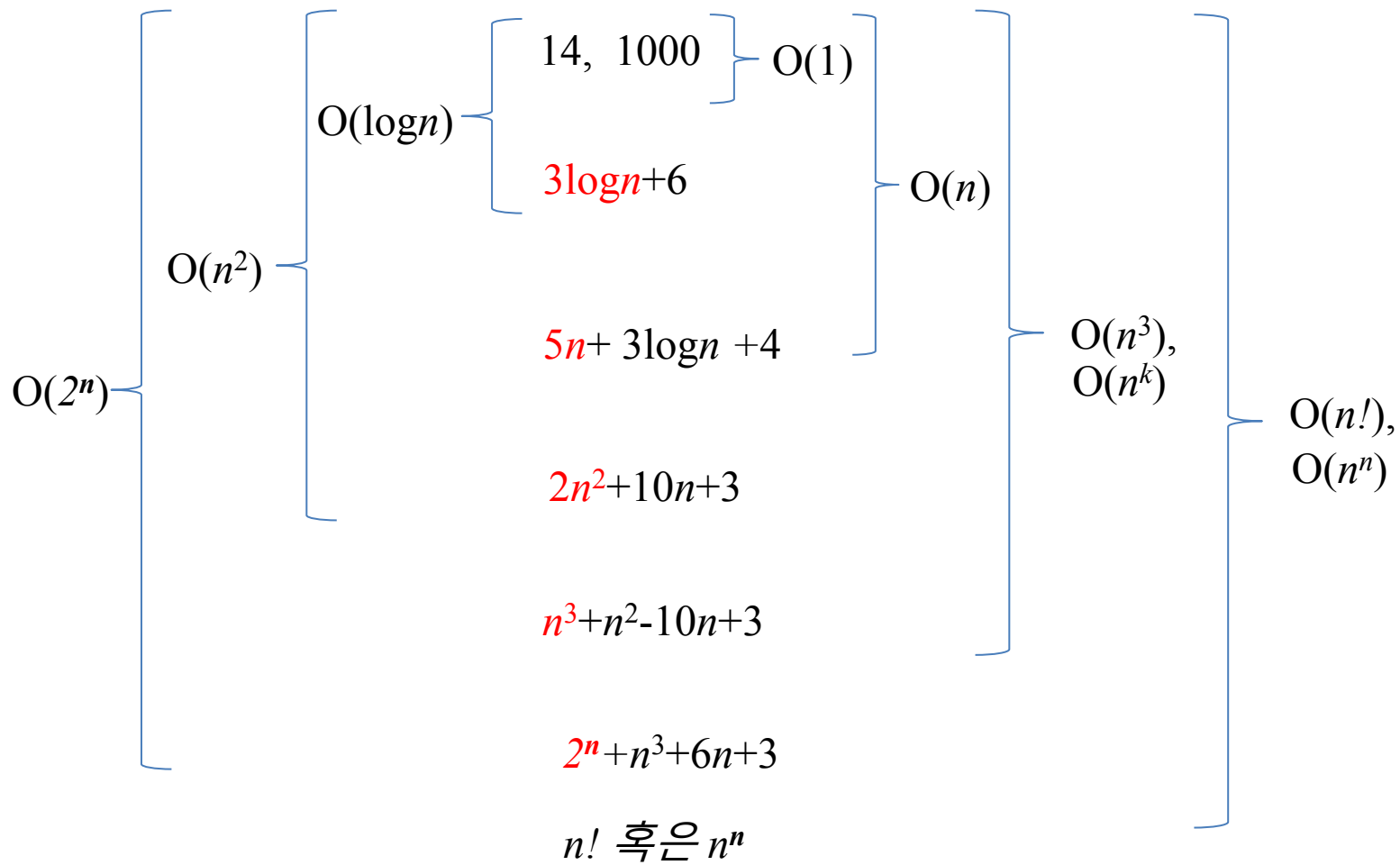
O, Ω, Θ 비교



자주 사용하는 O-표기

- $O(1)$ 상수 시간 (Constant time)
 - 당첨자 뽑기
- $O(\log n)$ 로그(대수) 시간 (Logarithmic time)
 - Binary Search
- $O(n)$ 선형 시간 (Linear time)
 - Linear Search
- $O(n \log n)$ 로그 선형 시간 (Log-linear time)
 - Quick Sorting
- $O(n^2)$ 제곱 시간 (Quadratic time)
 - Bubble Sorting
- $O(2^n)$ 지수 시간 (Exponential time)
 - Knapsack Problem

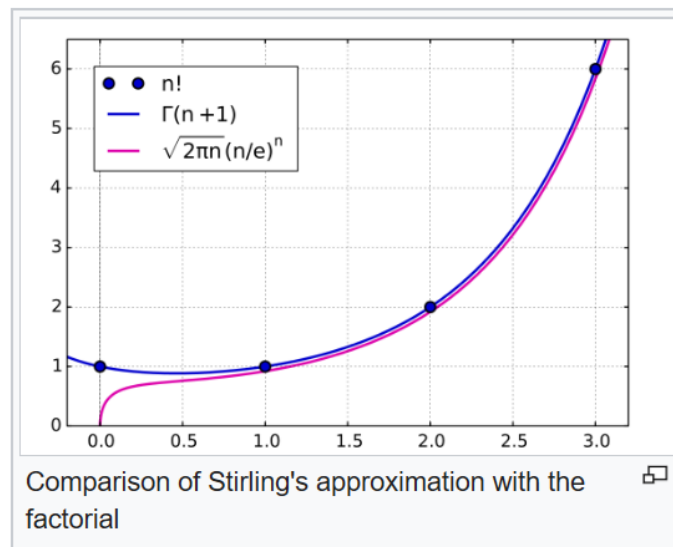
O-표기의 포함 관계



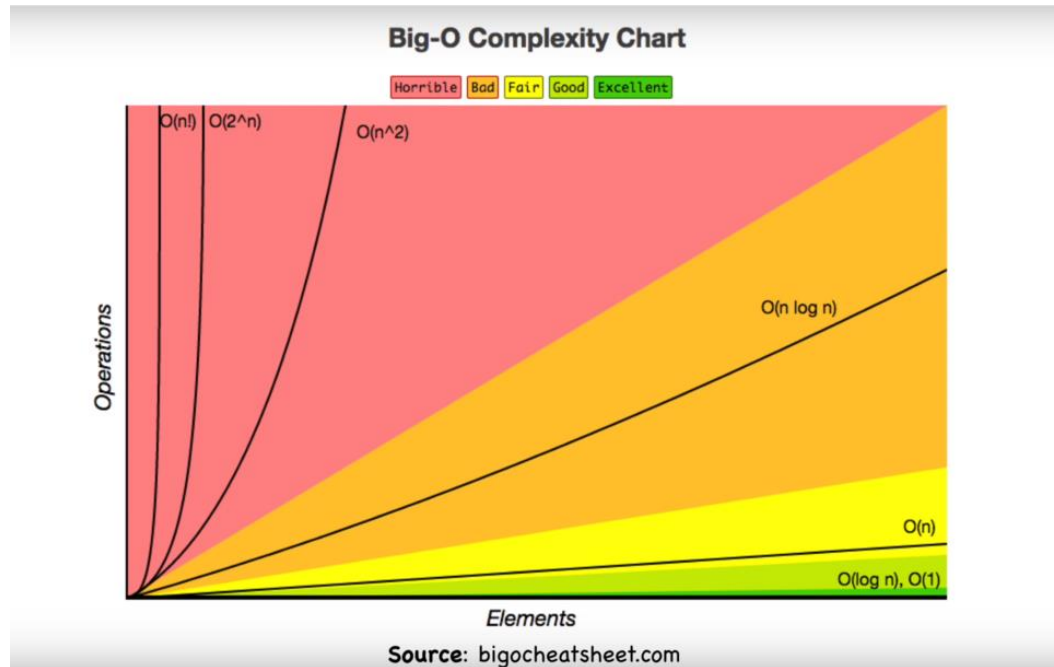
Stirling's approximation (or **Stirling's formula**)

$$\sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n} < n! < e n^{n+\frac{1}{2}}$$

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$



복잡도 비교



n	1	$\log_{10} n$	n	$n \log n$	n^2	2^n
1	1	0	1	1	1	2
10	1	1	10	10	100	$2^{10} = 10^3 = 1,000$
100	1	2	100	200	10,000	$2^{100} = 10^{30}$
1,000	1	3	1,000	3,000	1,000,000	$2^{1,000} = 10^{300}$
10,000	1	4	10,000	40,000	100,000,000	$2^{10,000} = 10^{3,000}$

2.7 왜 효율적인 알고리즘이 필요한가?

- 10억 개의 숫자를 정렬
 - PC에서 $O(n^2)$ 알고리즘
 - 버블정렬
 - 300여년
 - $O(n\log n)$ 알고리즘
 - 퀵정렬
 - 5분

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

$O(n\log n)$	1,000	1백만	10억
PC	< 1초	< 1초	5분
슈퍼컴	< 1초	< 1초	< 1초

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

속제

1. 입력의 크기가 n 일 때 다음 알고리즘의 수행 시간은 어떤 함수에 비례하는가?

```
sample(A[], n)
{
    sum1 ← 0;
    ① for i ← 1 to n
        sum1 ← sum1 + A[i];
    sum2 ← 0;
    ② for i ← 1 to n
        for j ← 1 to n
            sum2 ← sum2 + A[i] * A[j];
    return sum1 + sum2;
}
```

2. 다음 알고리즘의 수행 시간은 n 을 기준으로 어떤 함수에 비례하는가?

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

Q & A