자료구조: 2022년 1학기 [강의]

Algorithm Performance Analysis

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터융합학부

알고리즘 (Algorithm)

프로그램과 알고리즘

프로그램

- 컴퓨터에게 우리가 원하는 일을 시키기 위해서 사용하는 구체적인 방법
- 실제로 컴퓨터가 일을 할 수 있는 상태의 매우 세세한 것까지 표현

알고리즘

- 컴퓨터에게 편한 관점이 아닌, 우리 사람에게 유용한 관점에서 어떤 원하는 일을 처리하는 방법을 표현한 것

프로그램과 알고리즘

- 컴퓨터를 활용한다는 면에서는 알고리즘과 프로그램은 서로 불가분의 관계
  - 알고리즘은 프로그램으로 표현되어야 컴퓨터가 처리할 수 있다.
  - 프로그램도 결국은 우리가 원하는 일을 표현한 것이므로, (알고리즘의 조건을 만족하는 한) 알고리즘이라고 할 수 있다.

알고리즘이란?

- 알고리즘이란, 그 지시대로 실행하여 특정한 일을 달성하려는 명령어들의 유한집합이다. 그리고 다음의 조건을 만족해야 한다.
  - 입력 (Input)
  - 출력 (Output)
  - 명확성 (Definiteness)
  - 유한성 (Finiteness)
  - 유효성 (Effectiveness)

알고리즘의 조건

- 입력 (Input): 0 개 이상이어야 한다
  - 일반적으로는 무엇인가 외부에서 입력이 주어진다.
  - 어떤 경우에는 외부의 입력 없이 자체적으로 가지고 있는 자료를 활용하여 일을 할 수도 있다.
- 출력 (Output): 한 개 이상 있어야 한다
  - 알고리즘을 만드는 것은 무슨 일을 하여 원하는 결과를 얻고자 하는 것이다.
  - 그러므로 반드시 결과로서의 출력이 있어야 한다.

알고리즘의 조건

명확성 (Definiteness):

- 명령이 실행 시점에 이렇게도 할 수 있고 저렇게도 할 수 있다면, 우리는 그 결과를 전혀 예측할 수 없다.
- 마치 신뢰할 수 없는 사람과 함께 일하는 것 같다고 할 수 있다.

❑ 알고리즘의 조건

유한성 (Finiteness):

- 유한성이란, 명령을 유한 개수 만큼 실행하면 우리가 원하는 일이 달성되어야 한다는 것이다.
- 종료되지 않는 것은 알고리즘이라고 할 수 없다.
  - 잘못 작성되어 무한반복에 빠지는 프로그램
- 유용하지만, 종료되지 않는 프로그램
  - 운영체제:
    - MS Windows, Mac OS, Linux, iOS, Android 등등
  - 운영체제는 다음 일이 계속 들어 오기를 무한히 기다리고 있다. 엄밀한 의미에서 알고리즘이라고 할 수는 없다.

알고리즘의 조건

유효성 (Effectiveness):

- 명령 하나하나는 컴퓨터가 쉽게 간단히 짧은 시간 내에 실행할 수 있는 것이어야 한다.
  - 사칙연산, 메모리 읽기/쓰기, 등등.
- 적분과 같이 상당히 복잡하여 긴 시간이 필요한 일을 해야 하는 것을 하나의 명령으로 표현한 것은, 유효한 명령이라고 할 수 없다.

알고리즘은 어떻게 표현할까?

알고리즘 표현법

- 한글이나 영어와 같은 자연어
  - 사람에게 편리
  - 알고리즘 개발 과정에서 모호성을 점진적으로 제거하여, 궁극적으로 모호성이 전혀 없이 명쾌하게 표현이 되어야 한다.
- Java:
  - 컴퓨터에게 아주 적합
  - 사람에게는 약간 불편

- 여기서는 자연어와 Java 를 섞어서 사용하기로 한다

성능 분석과 측정

프로그램은 어떻게 판단할까?

- 생각해 볼 수 있는 판단 기준들은?
  - 설계시의 요구사항을 만족하는가?
  - 바르게 작동하는가?
  - 사용법과 작동법을 설명하는 문서를 포함하고 있는가?
  - 함수가 적정하게 설계되어 사용되는가?
  - 프로그램을 읽을 때 이해하기 쉬운가?
  - 메모리와 디스크를 효율적으로 사용하는가?
  - 주어진 일을 적정 시간 안에 수행할 수 있는가?

분석과 측정

- 성능 분석 (Analysis)
  - 사용할 컴퓨터와 무관하게 필요한 시간과 공간을 이론적으로 추정

- 성능 측정 (Measurement)
  - 특정 컴퓨터에서 시간과 공간을 실제로 측정

성능 분석과 프로그램의 복잡도

- 성능을 추정한 결과는 복잡도로 나타낸다

- 공간 복잡도 (Space Complexity)
  - 프로그램이 실행을 마칠 때까지 필요한 메모리의 양

- 시간 복잡도 (Time Complexity)
  - 프로그램 실행에 필요한 시간

공간 복잡도 (Space Complexity)

- 크기가 고정된 공간

- 크기가 가변적인 공간

공간 복잡도 (Space Complexity)

- 크기가 고정된 공간
  - 프로그램의 입력과 출력의 수와 크기와 무관
  - 프로그램 코드 크기
  - 컴파일 할 때 크기가 정해지는 변수나 상수
- 크기가 가변적인 공간
  - 프로그램 실행 시점의 인스턴스의 특성: 입출력의 개수, 값의 크기 등
    - 필요한 배열의 크기, 연결 체인의 크기
  - 재귀함수가 실행될 때 추가로 필요한 공간

공간 복잡도 (Space Complexity)

- 크기가 고정된 공간
  - 프로그램의 입력과 출력의 수와 크기와 무관
  - 프로그램 코드 크기
  - 컴파일 할 때 크기가 정해지는 변수나 상수
- 크기가 가변적인 공간
  - 프로그램 실행 시점의 인스턴스의 특성: 입출력의 개수, 값의 크기 등
    - 필요한 배열의 크기, 연결 체인의 크기
  - 재귀함수가 실행될 때 추가로 필요한 공간

공간 복잡도 (Space Complexity)

- 크기가 고정된 공간
  - 프로그램의 입력과 출력의 수와 크기와 무관
  - 프로그램 코드 크기
  - 컴파일 할 때 크기가 정해지는 변수나 상수
- 크기가 가변적인 공간
  - 프로그램 실행 시점의 인스턴스의 특성: 입출력의 개수, 값의 크기 등
    - 필요한 배열의 크기, 연결 체인의 크기
  - 재귀함수가 실행될 때 추가로 필요한 공간
- 공간복잡도는 크기가 가변적인 공간이 중요

크기가 고정된 공간

```
public float average (float a, float b, float c)
{
      return  (a+b+c) / 3.0  ;
}
```

➡ 프로그램에 필요한 공간 크기는,
컴파일 시점에 미리 결정되어 고정된다.

크기가 가변적인 공간

```
public void main ()
{
    Scanner  scanner = new  Scanner() ;
    int  numberOfStudents = scanner.nextInt() ;
    int[]  scores = new int[numberOfStudents] ;
    ……
}
```

➡ 프로그램에 필요한 배열 scores[]의 공간 크기는, scanner.nextInt() 의 값에 따라 프로그램이 실행되는 동안에 동적으로 결정된다.

재귀함수로 인한 공간복잡도

```
pubic int  nfact (int n)
{
    if ( n==0 ) {
        return  0 ;
    }
    else {
        return  n * nfact(n-1) ;
    }
}
```

➡ 프로그램에 필요한 공간 크기는,
프로그램 실행 중에
n 의 값의 크기에 따라 달라진다.

□ 시간 복잡도 (Time Complexity)

- 프로그램의 실행 시간이 중요
  - 컴파일 시간은 중요 관심 대상이 아니다
- 추정에서 수치화 방법은?
  - 연산의 개수를 센다.
    - 특정 컴퓨터에 의존적이지 않게 된다.
  - 그러나, 엄격하고 정확하게 찾는 것은 매우 어렵다.
- 연산의 단위는?
  - 하나의 연산을 실행하는 데 걸리는 시간이 실행 시점의 특정 상황에 영향 받지 않아야 한다
    - 사칙 연산, 비교연산 등 기본 연산
    - 메모리에 저장, 메모리로부터 읽어 오기
    - 배열의 인덱싱
    - 입출력

예: 시간복잡도

```
public double  sum (double[] a, int n)
{
    double  sum = 0.0 ;
    int  i = 0 ;
    while ( i < n ) {

        sum = sum + a[i] ;

        i++ ;
    }


    return  sum ;
}
```

예: 연산 스텝 세기

```
public double  sum (double[] a, int n)
{
   double  sum = 0.0 ; count++; // 메모리에 저장
   int  i = 0 ; count++ ; // 메모리에 저장
   while ( i < n ) {
         count++ ; //  while 비교
      sum = sum + a[i] ;
         count+=3 ; // 배열 첨자 처리, 덧셈, 저장
      i++ ; count++ // 증가
   }
   count++ ; // 마지막 while 비교
   count++ ; // return
   return  sum ;
}
```

예: 연산 스텝 세기

```
public double  sum (double[] a, int n)
{
    int  i = 0 ;
    count += 2 ; // first 2 assignments
    while ( i < n ) {
        count += 5 ; // total numbers in each loop


        i++ ;
    }
    count += 2 ; // last while comparison & return

    return  0 ;
}
```

➡ 스텝의 총 회수: $5n + 4$

재귀의 경우 [1]: f()

```
pubic int  f (int n)
{
    if ( n==0 ) { // count++
        return  0 ; // count++
    }
    else {
        return  f (n/2) + f(n/2) + 1 ; // count+=7 (나눗셈 2, 덧셈 2, call 2, return)
    }
}
```

$$T(n) = \begin{cases} 2 & if\ n = 0 \\ 2 \times T(n/2) + 8 & if\ n > 0 \end{cases}$$

재귀의 경우 [2]: nfact()

```
pubic int  nfact (int n)
{
    if ( n==0 ) { // count++
        return  0 ; // count++
    }
    else {
        return  n * nfact (n-1) ; // count+=4 (곱셈, 뺄셈, call, return)
    }
}
```

$$T(n) = \begin{cases} 2 & if\ n = 0 \\ T(n-1) + 5 & if\ n > 0 \end{cases}$$

❑ 시간복잡도의 실용적 관점

- 측정이 아닌 "추정" 을 하려는 것
- 중요:
  n 의 값이 매우 큰 상황을 고려한다.

- 스텝의 총 회수

  5n + 4

  ≈ 5n (n 이 매우 크므로 4 는 무시할 수 있다)

  ≈ n (크기에 따른 현상을 이해하는 데는, 단지 비례한다는 정도면 충분)

시간복잡도에 영향을 주는 것들은?

- 입력의 크기:
  - 입력의 크기 : factorial
  - 입력의 개수: 정렬

- 출력의 크기:
  - 출력의 크기
  - 출력의 개수

더 고려할 점들

- 특정 입력 데이터 자체가 시간복잡도에 영향을 줄 수 있다.
  - 퀵 정렬의 경우
    - 입력 데이터가 이미 정렬되어 있으면 최악의 시간복잡도를 보여준다.
    - 입력 데이터 순서가 무작위적이면 최선의 시간복잡도를 보여준다.
- 3 가지 경우를 고려할 필요가 있다.
  - 최선의 경우 / 최악의 경우 / 평균적인 경우

복잡도 표기

점근 표기 (Asymptotic Notation)

- 성능 분석을 하는 목적
  - 같은 기능을 하는 두 프로그램의 시간 복잡도를 비교
  - 프로그램의 실행 상황의 특성 (입력의 크기 등)이 변함에 따라 실행 시간이 어떻게 변하는지를 예측

- 스텝의 수 표현의 문제점
  - 스텝의 수를 정확히 세는 일은 매우 어렵다.
  - 세는 행위가 부정확한 면이 있다.
  - 따라서, 엄격한 표현을 사용하는 것이 오히려 유용하지 않을 수 있다.

상한 (Upper Bound) 을 나타내는 Big-Oh

- 예: $f(n) = 5n+4$

  그러면 $f(n) = O(n)$ 이라고 한다.

  - n 이 상당히 클 때, f(n) 은 어떤 상수 c 에 대해 c · n 이상은 아니라는 의미이다.
  - 상한을 나타내기 위한 것이므로, 상수 c 의 크기는 큰 의미를 갖지 않는다.

$$5n+4 < \mathbf{6} \cdot n$$

$$5n+4 < \mathbf{7} \cdot n$$

$$5n+4 < \mathbf{1000} \cdot n$$

Big-Oh 의 정의

- $f(n) = O(g(n))$

  iff n_0 보다 크거나 같은 모든 n 에 대해,
  부등식 $f(n) \le c \cdot g(n)$ 을 만족시키는
  양의 정수 c 와 n_0 가 존재한다.

- 예: $f(n) = 5n+4$
  - $c=6$, $n_0 = 4$ 라고 하자
  - 그러면 $4(=n_0)$ 보다 크거나 같은 모든 n 에 대해,
    $f(n) \le 6 \cdot n$ 이 성립한다.
    - 왜냐하면, 4 보다 크거나 같은 모든 n 에 대해

      $$6n - (5n+4) = n-4 \ge 0$$

      이다.
  - g(n) 은 n 이 된다.
  - 그러므로, $f(n) = O(g(n))$, 즉, $f(n) = O(n)$ 이다.

Big-Oh 증명의 예:

- $f(n) = 2n^2+6n-8$ 은 $O(n^2)$ 임을 증명하라.

(증명)

$g(n) = n^2$ 이라 하자. 그러면,

$3\cdot g(n) - f(n) = 3n^2 - (2n^2+6n-8) = n^2 -6n+8 = (n-3)^2-1.$

4 보다 크거나 같은 모든 n 에 대해, 다음이 성립한다.

$3\cdot g(n) - f(n) = (n-3)^2 - 1 \geq 0$

그러므로, 다음 부등식을 만족하는 두 양수 $c=3$ 과 $n_0=4$ 를 발견하였다:

4 보다 큰 모든 n 에 대해, $f(n) \leq 3\cdot g(n)$

따라서, Big Oh 의 정의에 따라,

$f(n) = O(n^2)$

[증명 끝]

예제

- $3n + 2 = O(n)$
  왜냐하면, 2 보다 크거나 같은 모든 n 에 대해,
  $3n + 2 \le 4n$.
- $3n + 3 = O(n)$
  왜냐하면, 3 보다 크거나 같은 모든 n 에 대해,
  $3n + 3 \le 4$.
- $10n^2 + 4n + 2 = O(n^2)$
  왜냐하면, 5 보다 크거나 같은 모든 n 에 대해,
  $10n^2 + 4n + 2 \le 11n^2$.
- $1000n^2 + 100n - 6 = O(n^2)$
  왜냐하면, 100 보다 크거나 같은 모든 n 에 대해,
  $1000n^2 + 100n - 6 \le 1001n^2$.
- $6 \cdot 2^n + n^2 = O(2^n)$
  왜냐하면, 100 보다 크거나 같은 모든 n 에 대해,
  $6 \cdot 2^n + n^2 \le 7 \cdot 2^n$.

상한은 작을수록!

- O (Big-Oh)는 상한 (upper bound)을 나타낸다.
- $3n + 3 = O(n^2)$
  - 왜냐하면, 2보다 크거나 같은 모든 n 에 대해, $3n + 3 \le 3n^2$ 이다.
- 그렇지만, $3n + 3 = O(n)$ 이라고 보통 말하지, $3n + 3 = O(n^2)$ 이라고는 거의 말하지 않는다.
- 인간의 키의 상한을 말할 때, 다음 중 어느 표현이 더 의미 있는가?
  - "인간의 키는 3M를 넘지 않는다"
  - "인간의 키는 30M를 넘지 않는다"
- $10n^2 + 4n + 2 = O(n^4)$ 는 맞는 말이지만, 그렇게 말하지는 않는다.
  - $10n^2 + 4n + 2 = O(n^2)$ 이라고 하는 것이 더 의미 있기 때문이다.

잘못된 상한

- $3n + 2 \neq O(1)$
  - 왜냐하면, 어떤 상수 c 와 어떤 n_0 에 대해서도, n_0 보다 크거나 같은 모든 n 에 대해, $(3n+2) \leq c \cdot 1$ 는 성립할 수 없다.
  - $10n^2 + 4n + 2 \neq O(n)$

❑ 자주 사용하는 Big-Oh

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

다항식의 Big-Oh

- $f(n) = a_m n^m + \ldots + a_1 n + a_0$ 이면, $f(n) = O(n^m)$ 이다

(증명)

n ≥ 1 인 모든 n 에 대해, 다음 부등식이 성립한다.

$$
\begin{aligned}
f(n) &\le |a_m| n^m + \ldots + |a_1| n + |a_0| \\
&\le |a_m| n^m + \ldots + |a_1| n^m + |a_0| n^m \\
&= (|a_m| + \ldots + |a_1| + |a_0|)\, n^m
\end{aligned}
$$

따라서, $f(n) = O(n^m)$.

[증명 끝]

하한 (Lower Bound) 인 Big-Omega

- $f(n) = \Omega(g(n))$
  iff n_0 보다 크거나 같은 모든 n 에 대해 부등식
  $f(n) \ge c \cdot g(n)$ 을 만족시키는 양의 정수 c 와
  n_0 가 존재한다.

- 예제:
  - $3n + 3 = \Omega(n)$
    왜냐하면, $n \ge 1$ 인 모든 n 에 대해
    $3n + 3 \ge 3n$ 이다.
  - $100n + 6 = \Omega(n)$
  - $6 \cdot 2^n + n^2 = \Omega(2^n)$

❑ 하한은 클수록!

- Ω (Big-Omega) 는 하한 (lower bound)을 나타낸다.
- $3n + 3 = \Omega(1)$ 이다.
  왜냐하면, $n \geq 1$ 인 모든 n 에 대해, $3n+3 \geq 3 \cdot 1$
- 그렇지만, $3n + 3 = \Omega(n)$ 이라고는 말하지만,
  $3n + 3 = O(1)$ 이라고는 거의 말하지 않는다.

- $(6 \cdot 2^n + n^2)$의 가능한 하한들
  - $6 \cdot 2^n + n^2 = \Omega(2^n)$
  - $6 \cdot 2^n + n^2 = \Omega(n^{50.2})$
  - $6 \cdot 2^n + n^2 = \Omega(n)$
  - $6 \cdot 2^n + n^2 = \Omega(1)$
  - 그러나 $(6 \cdot 2^n + n^2) = \Omega(2^n)$ 이라고 보통 말한다.

다항식의 하한

- $f(n) = a_m n^m + \ldots + a_1 n + a_0$ $(a_m > 0)$ 이면,
  $f(n) = \Omega(n^m)$ 이다.

Big-Theta

- $f(n) = \Theta(g(n))$
  iff $n \geq n_0$ 인 모든 n 에 대해,
  부등식 $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ 을 만족시키는
  양의 정수 c_1 , c_2 와 n_0 가 존재한다.

- Big-Theta 는 Big-Oh 와 Big-Omega 보다 더 섬세한 표현

- $f(n) = a_m n^m + \ldots + a_1 n + a_0$ 이면, $f(n) = \Theta(n^m)$ 이다.

Big-Theta

- 예제:
  - $3n + 2 = \Theta(n)$
    - $n \geq 2$ 인 모든 n 에 대해, $3n \leq 3n + 2 \leq 4n$ 이므로.
  - $10n^2 + 4n + 2 = \Theta(n^2)$
  - $6 \cdot 2^n + n^2 = \Theta(2^n)$
  - $3n + 2 \neq \Theta(1)$
  - $10n^2 + 4n + 2 \neq \Theta(n)$
  - $6 \cdot 2^n + n^2 \neq \Theta(n^2)$
  - $6 \cdot 2^n + n^2 \neq \Theta(n^{100})$
  - $6 \cdot 2^n + n^2 \neq \Theta(1)$

점근 표현의 계수

- 점근 표현 (O, Ω, ,Θ) 에서 사용된 g(n) 의 계수는 언제나 1 이었다.
  - $3n + 3 = O(3n)$ 은 맞는 표현이지만, 그렇게 말하지는 않는다. 보통 $3n + 3 = O(n)$ 이라고 한다.

점근 표기법 요약

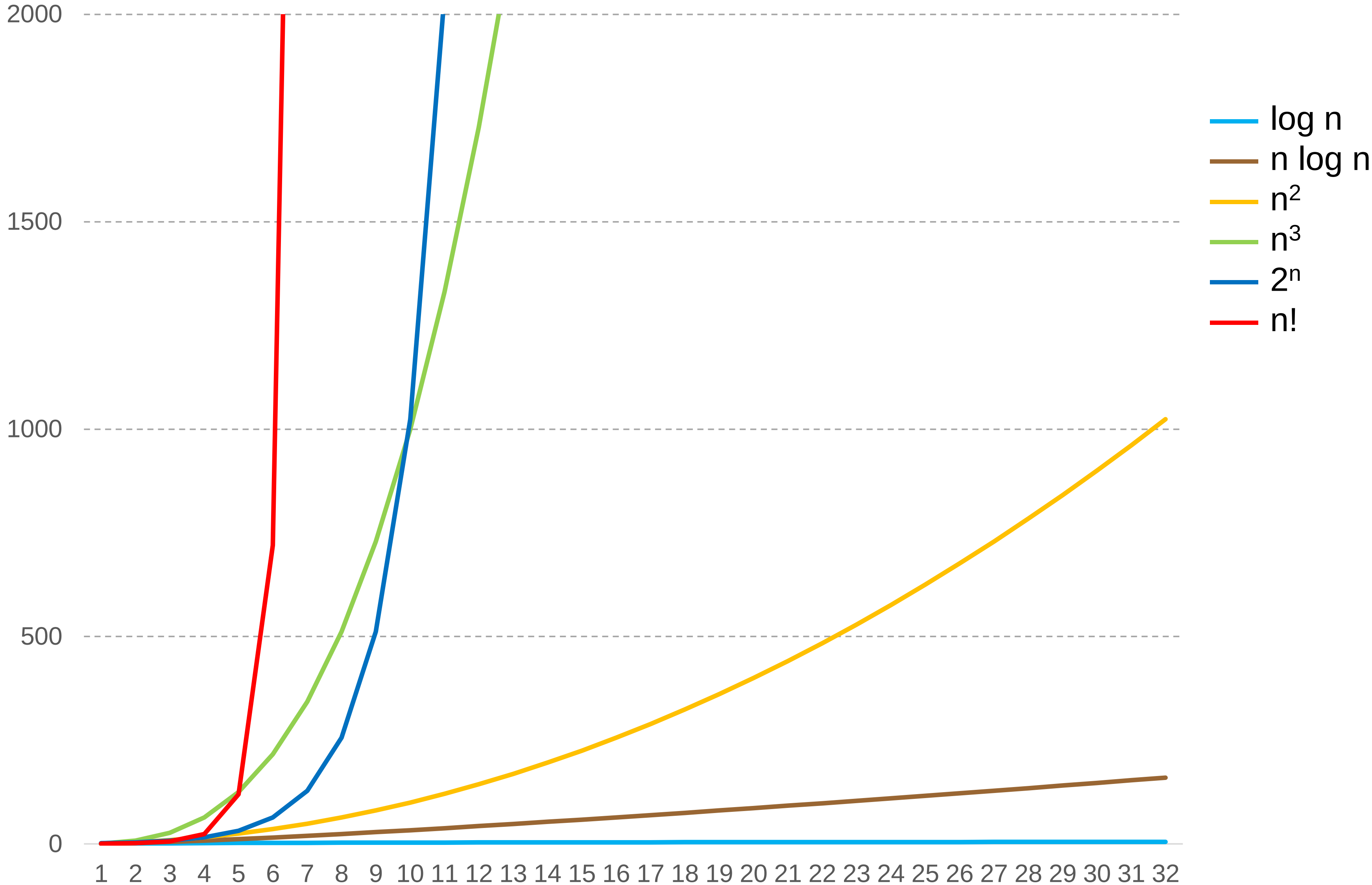
- 직관적 이해가 필요
  - n 이 상당히 클 때의 상황에 대한 표현
- 주로 중요 g(n) 을 사용
  - $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- 특히 상한을 나타내는 Big-Oh 가 중요하며, 주로 많이 사용된다.

자주 사용하는 복잡도 함수

함수 값의 비교

$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	40326
4	16	64	256	4,096	65,536	2,092,278,988,800
5	32	160	1,024	32,768	4,294,967,297	26313×10^{33}

복잡도 함수의 그래프


2000
1500
1000
500
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
log n
n log n
n2
n3
2n
n!

❑ 컴퓨터에서의 실행 시간

■ 처리 속도: 초당 10^9 스텝

	log n	n	n $\log_2$ n	n^2	n^3	2^n	n!
10	0.003 μs	0.01 μs	0.033 μs	0.10 μs	1.0 μs	1 μs	10.87 ms
20	0.004 μs	0.02 μs	0.086 μs	0.40 μs	8.0 μs	1 ms	23 years
30	0.005 μs	0.03 μs	0.147 μs	0.90 μs	27.0 μs	1 s	2.5 x 10^{16} years
40	0.005 μs	0.04 μs	0.213 μs	1.60 μs	64.0 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.50 μs	125.0 μs	13 days	
10^2	0.007 μs	0.10 μs	0.664 μs	10.00 μs	1.0 ms	4 x 10^{13} years	
10^3	0.010 μs	1.00 μs	9.966 μs	1.00 ms	1.0 s		
10^4	0.013 μs	10.00 μs	130.000 μs	100.00 ms	16.7 min		
10^5	0.017 μs	0.10 ms	1.670 ms	10.00 s	11.6 days		
10^6	0.020 μs	1.00 ms	19.930 ms	16.70 min	31.7 years		
10^7	0.023 μs	0.01 s	0.222 s	1.16 days	3.17 x 10^4 years		
10^8	0.027 μs	0.10 s	2.660 s	115.7 days	3.17 x 10^7 years		
10^9	0.030 μs	1.00 s	29.900 s	31.7 years			

μs = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
s = seconds; m = minutes; h = hours; d = days; y = years

성능 측정

- 실행시간 측정
  - Java 프레임워크 사용
  -
- 실험 데이터의 생성
  - 최악의 경우의 실험 데이터
  - 평균 경우의 실험 데이터
    - 난수를 생성하여 만든다

End of "Algorithm, Performance Analysis"