

# 알고리즘의 이해

강의 1

# 알고리즘이란 무엇인가?

- 어떤 값이나 값의 집합을 입력으로 받아 또 다른 값이나 값의 집합을 출력하는 잘 정의된 계산 절차.
- 어떤 입력을 어떤 출력으로 변환하는 일련의 계산과정.
- 문제 해결 절차를 체계적으로 기술한 것
- 문제의 요구조건
  - 입력과 출력으로 명시할 수 있다
  - 알고리즘은 입력으로부터 출력을 만드는 과정을 기술

# 입출력의 예

- 문제
  - 100명의 학생의 시험 점수의 최댓값을 찾으라
- 입력
  - 100명의 학생들의 시험 점수
- 출력
  - 위 100개의 시험 점수들 중 최댓값

# 알고리즘 예시

- 어떤 값이나 값의 집합을 입력으로 받아 또 다른 값이나 값의 집합을 출력하는 잘 정의된 계산 절차.
- 어떤 입력을 어떤 출력으로 변환하는 일련의 계산과정.
- 예시 – 최대값 찾기.

60    80    70    90    40    20    60    100

# 알고리즘 예시

- 예시 - 최대값 찾기

60 80 70 90 40 20 60 100



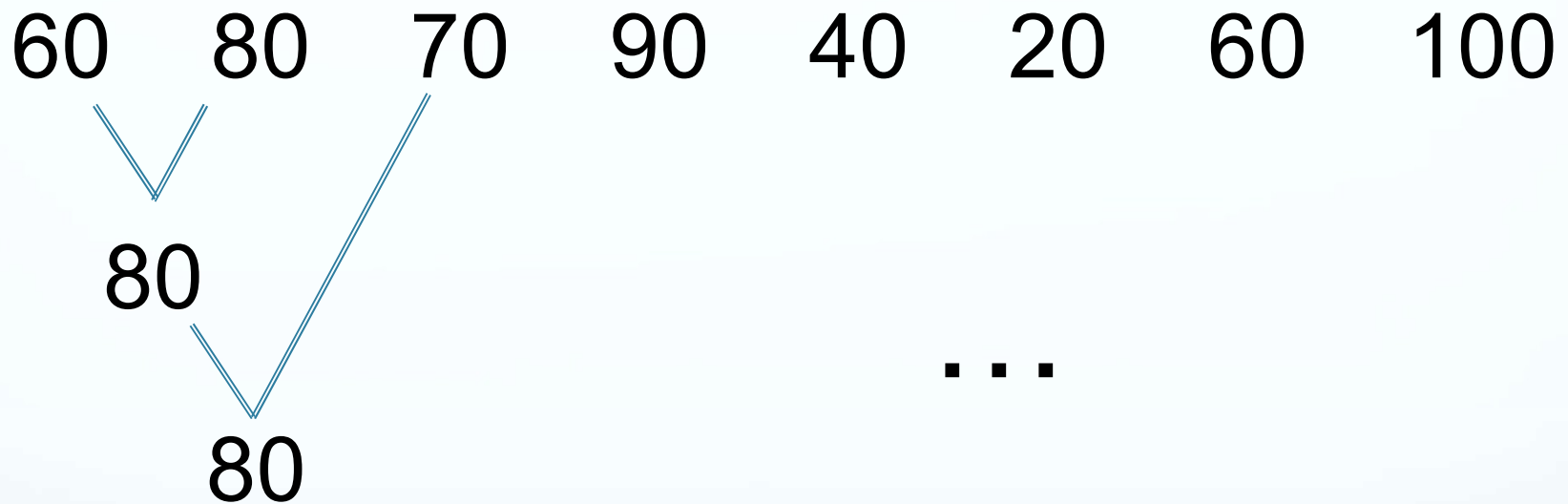
80

...

- 입력: 60 80 70 90 40 20 60 100
- 출력: 100

# 알고리즘 예시

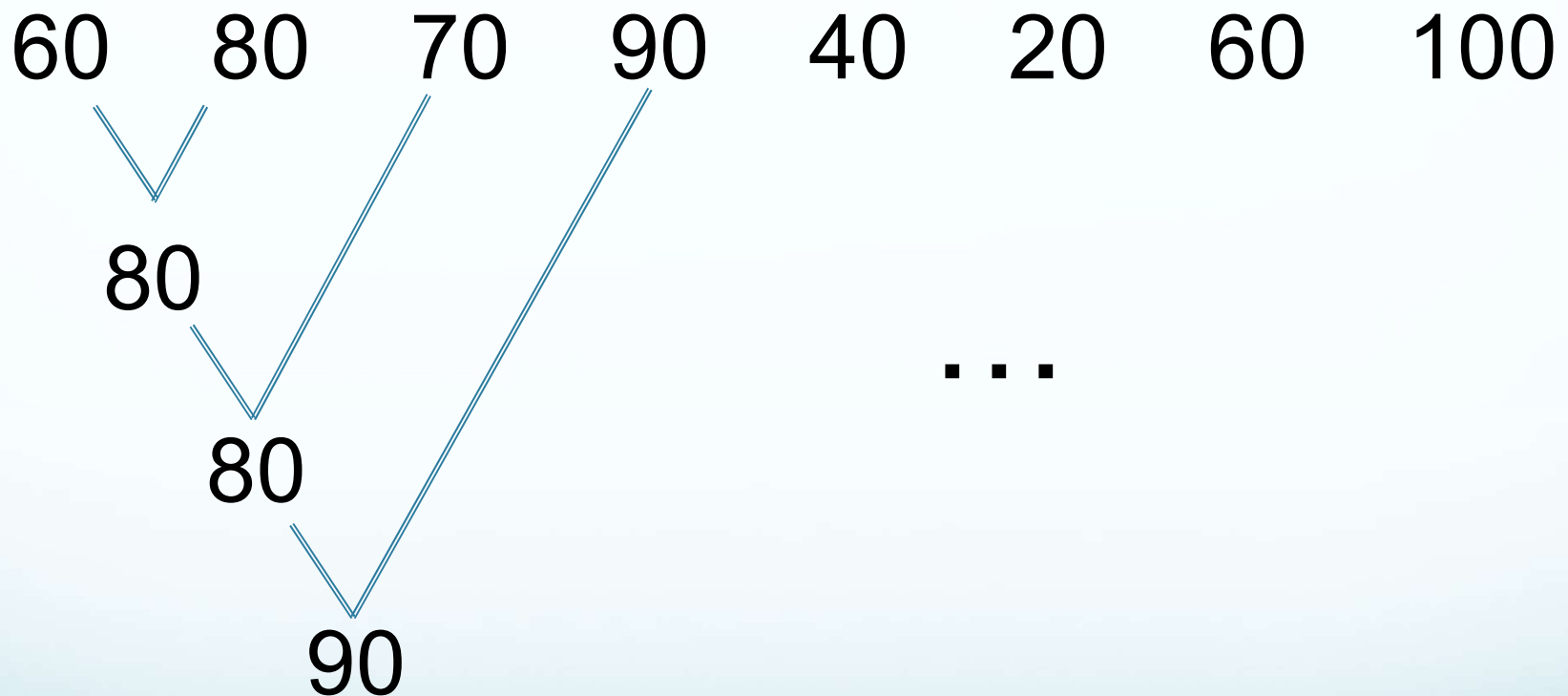
- 예시 - 최대값 찾기



- 입력: 60 80 70 90 40 20 60 100
- 출력: 100

# 알고리즘 예시

- 예시 - 최대값 찾기



- 입력: 60 80 70 90 40 20 60 100
- 출력: 100

# 바람직한 알고리즘

- 명확해야 한다
  - 이해하기 쉽고 가능하면 간명하도록
  - 지나친 기호적 표현은 오히려 명확성을 떨어뜨림
  - 명확성을 해치지 않으면 일반언어의 사용도 무방
- 효율적이어야 한다
  - 같은 문제를 해결하는 알고리즘들의 수행 시간이 수백만 배 이상 차이 날 수 있다



# 알고리즘의 기술 언어

- 알고리즘 표현방법
  - 자연어 표현
  - 흐름도(flow chart) 표현
  - 유사코드(pseudo code) 표현
  - 프로그래밍 언어 표현

# 알고리즘의 기술 언어

- 최대값 찾기 문제 . 주어진 수들 중 최대값을 찾는 방법을 어떻게 표현 할 수 있을 까?

60    80    70    90    40    20    60    100

# 알고리즘의 기술 언어

## 자연어로 표현된 알고리즘

- 장점:
- 단점:

60 80 70 90 40 20 60 100

ArrayMax(A, n)

1.  
2.  
3.

# 알고리즘의 기술 언어

## 자연어로 표현된 알고리즘

- 장점: 가장 읽기 쉽다.
- 단점: 의미 전달이 모호해질 우려가 있다.

60    80    70    90    40    20    60    100

ArrayMax(A, n)

1. 배열 A의 첫 요소를 변수 tmp에 복사
2. 배열 A의 다음 요소들을 차례대로 tmp와 비교하며  
   더 크면 tmp로 복사
3. 배열 A의 모든 요소를 비교했으면 tmp를 반환

# 알고리즘의 기술 언어

## 흐름도(flowchart)로 표현된 알고리즘

- 장점:
- 단점:



배열 A: 60   80   70   90   40   20   60   100

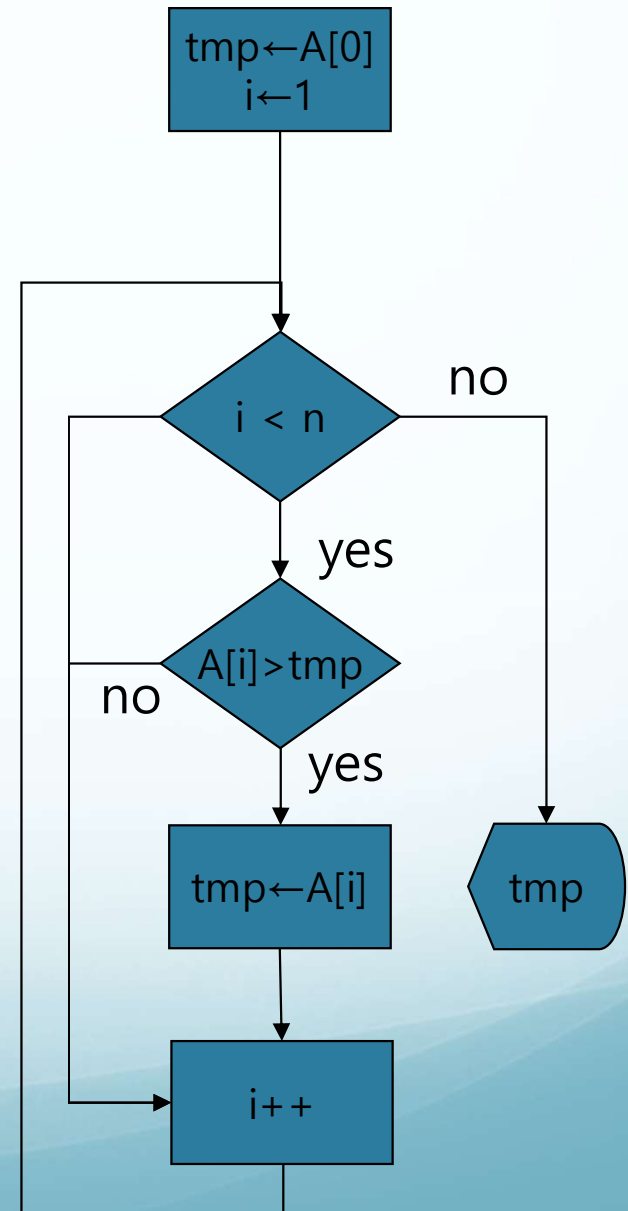


# 알고리즘의 기술 언어

## 흐름도(flowchart)로 표현된 알고리즘

- 장점: 이해하기 쉽다.
- 단점: 알고리즘이 복잡하면  
흐름도가 매우 복잡해지며  
작성에도 많은 시간이 소요된다.

배열 A: 60   80   70   90   40   20   60   100



# 알고리즘의 기술 언어

## 프로그래밍 언어로 표현된 알고리즘

- 장점:
- 단점:

```
#define MAX_ELEMENTS 100
int score[MAX_ELEMENTS];
int find_max_score(int n)
{
}
}
```

# 알고리즘의 기술 언어

## 프로그래밍 언어로 표현된 알고리즘

- 장점: 가장 정확하다.
- 단점: 구현관련 세부 사항들로 인해 알고리즘 핵심내용을 파악하는 데에는 오히려 방해가 될 수 있다.

```
#define MAX_ELEMENTS 100
int score[MAX_ELEMENTS];
int find_max_score(int n)
{
    int i, tmp;
    tmp=score[0];
    for(i=1;i<n;i++){
        if( score[i] > tmp ){
            tmp = score[i];
        }
    }
    return tmp;
}
```



# 알고리즘의 기술 언어

## 유사코드(pseudo code)

- 프로그래밍 언어와 유사하나 세부 표현은 생략하고 조금 더 간략하게 표현한 언어로 정해진 법칙은 없음.

- 변수 · 보통은 선언 없이 그냥 사용 i=1    A[i]=1

- 반복문

```
while M>0  
    i=i+1  
...
```

```
for i=1 to M  
    ...
```

```
for (i=1; i<N; i=i+2) {  
    ...  
}
```

- 조건문

```
if i<M  
    statement  
else  
    statement
```

- 함수

```
add(a,b)  
    return (a+b)
```

# 알고리즘의 기술 언어

유사코드(pseudo code)로 표현된 알고리즘

- 장점:
- 단점:
- 특징:

ArrayMax(A,n)

# 알고리즘의 기술 언어

유사코드(pseudo code)로 표현된 알고리즘

- 장점: 고수준의 구조적 표현법. 구현상 세부 문제를 감춤으로써 알고리즘 핵심내용에 보다 집중할 수 있음.
- 단점: 프로그래밍 언어에 비해 덜 구체적임.
- 특징: 알고리즘 기술에 가장 많이 사용됨.

```
ArrayMax(A,n)
  tmp ← A[0]
  for i ← 1 to n-1
    if tmp < A[i]
      tmp ← A[i]
  return tmp
```

# 알고리즘의 기술 언어

## 기본 자료구조

- 데이터 구조는 알고리즘 효율에 크게 영향을 미침.
- 일반적으로 데이터 구조가 복잡할 경우 연산의 횟수가 줄어들고, 데이터 구조가 단순할 경우 연산 횟수가 많아져 수행시간이 늘어나는 경향이 있다.
- 배열 (array) 

10	50	30	40	20	15
----	----	----	----	----	----

  - 장점 · 각 원소의 접근 시간이 동일, 임의의 원소에 접근 속도가 빠르다.
  - 단점 · 새로운 원소를 중간에 삽입/삭제가 용이하지 않음
- 연결리스트 (linked list) ... 

10	→	50	→	30	→	40	→	...
----	---	----	---	----	---	----	---	-----

  - 장점 - 원소의 삽입 삭제 연산이 간단하다.
  - 단점 - 자료접근이 어렵다(링크를 따라가야함), 자료를 위한 메모리 외에 링크를 위한 메모리 필요하다.

# 알고리즘의 기술 언어

## 기본 자료구조

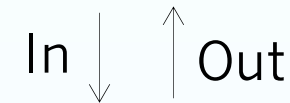
**Insert 10, 20, 30, 40**

- 큐 (queue)



FIFO (First In First Out)

- 스택 (stack)



LIFO (Last In First Out)

# 알고리즘의 기술 언어

## 기본 자료구조

- 그래프 (graph)

- $G=(V, E)$

- $V$ : 정점, vertex,

- $E$ : 간선, edge

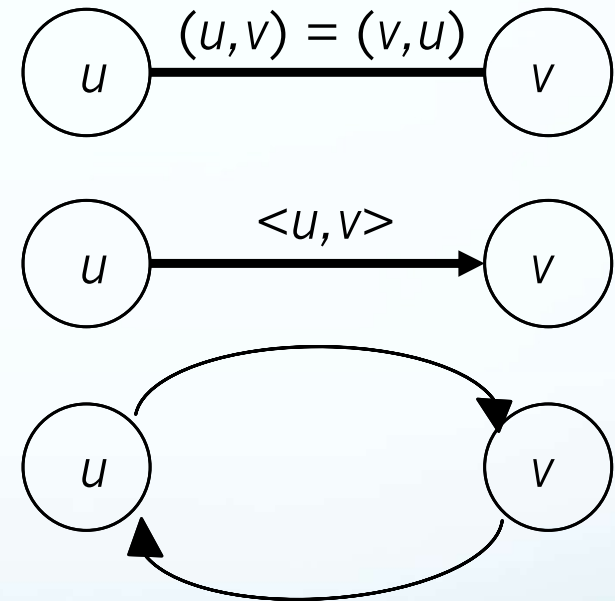
- 무방향 그래프 (undirected graph)

- 방향 그래프 (directed graph)

- 루프 (loop)

- 정점의 차수 (degree, in-degree, out-degree) – 정점에 연결된 간선의 갯수



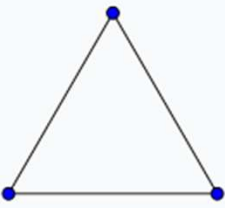
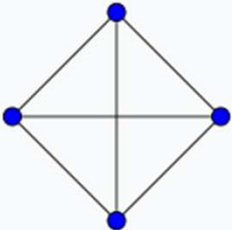
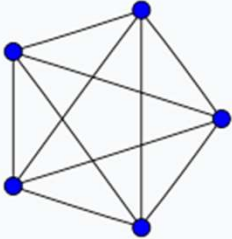
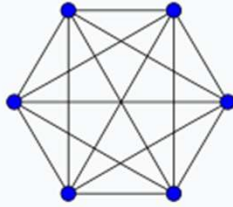
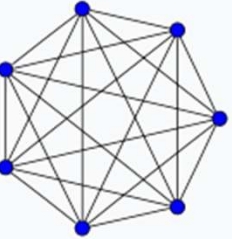
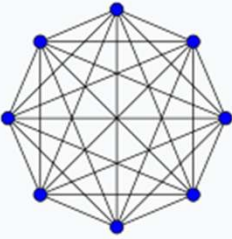
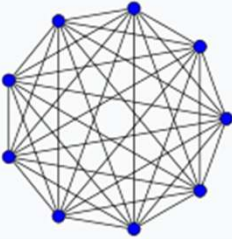
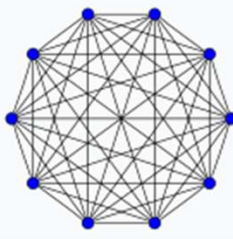
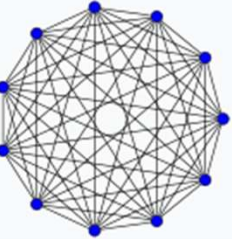
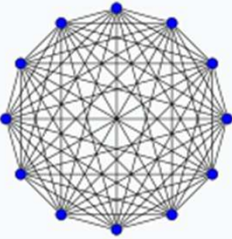
- 정점  $v_1$  에서  $v_2$  까지의 경로 (path) – 간선으로 연결된 정점들의 순차열



# 알고리즘의 기술 언어

## 기본 자료구조

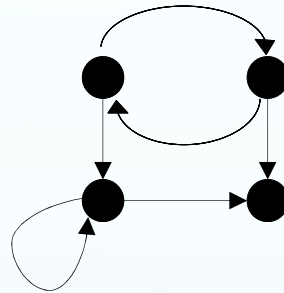
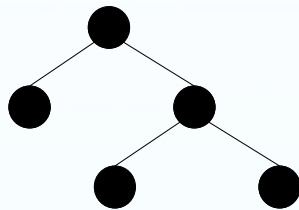
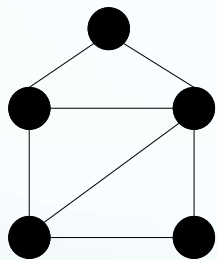
- 그래프 (graph)
  - 사이클(cycle) - 시작 정점과 끝정점이 같은 경로
  - 완전 그래프 (complete graph) - 모든 정점끼리 간선으로 연결된 그래프

$K_1: 0$	$K_2: 1$	$K_3: 3$	$K_4: 6$
			
$K_5: 10$	$K_6: 15$	$K_7: 21$	$K_8: 28$
			
$K_9: 36$	$K_{10}: 45$	$K_{11}: 55$	$K_{12}: 66$
			

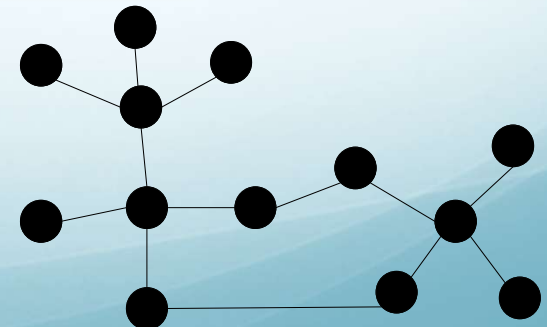
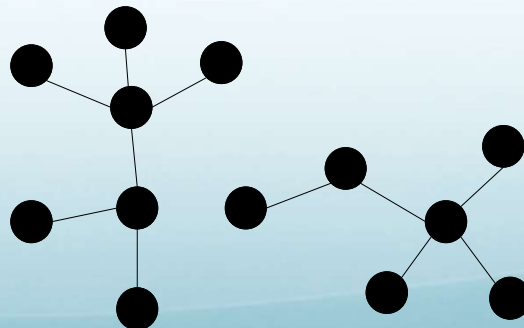
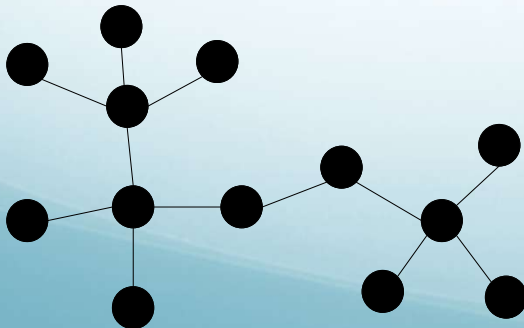
# 알고리즘의 기술 언어

## 기본 자료구조

- 그래프 (graph)
  - 사이클(cycle) - 시작 정점과 끝정점이 같은 경로
  - 완전 그래프 (complete graph) - 모든 정점끼리 간선으로 연결된 그래프
  - 여러가지 그래프의 예



- 나무 (tree) - 연결된 무사이클 무방향 그래프





# 알고리즘의 기술 언어

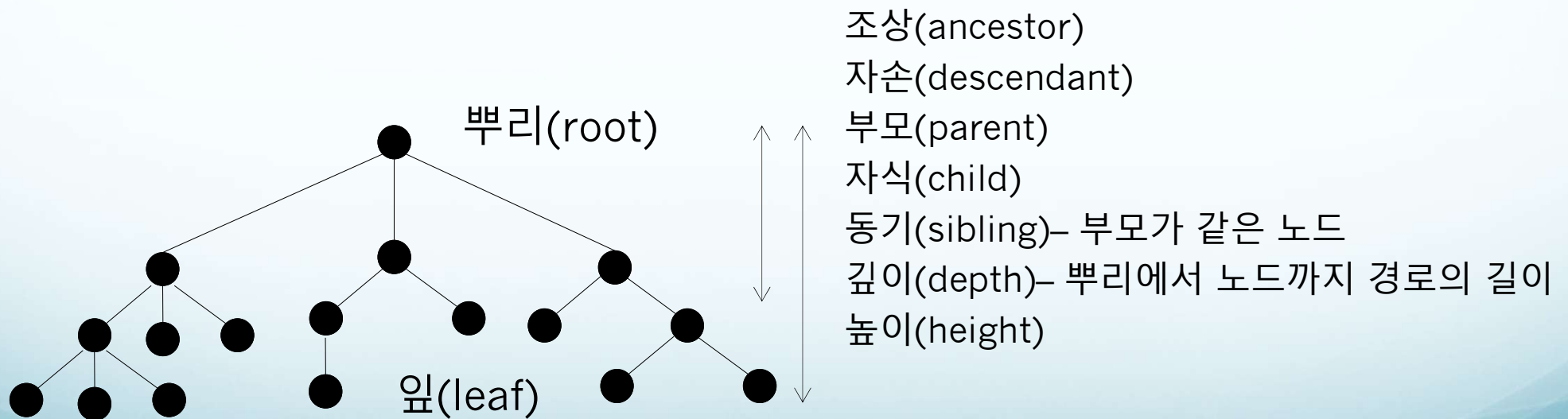
## 기본 자료구조

- 그래프 (graph)

- 나무 (tree)

뿌리나무 (rooted tree) – 나무의 정점 중 하나가 뿌리로 지정된 나무

이진 나무 (binary tree) – 각 노드의 자식이 2개 이하인 나무



# 알고리즘의 평가기준

- 정당성 (Accuracy)

An algorithm is said to be correct(정당하다) if it **halts** with the **correct output** for **every** input instance.

A correct algorithm solves the given problem.

An incorrect algorithm might not halt at all or it might halt with an answer other than the desired one.

- 효율성 (Efficiency)

- 컴퓨터가 상당히 빠를수는 있지만 무한히 빠를 수는 없고 메모리도 매우 저렴할 수 있지만 비용이 전혀 들지 않을 수는 없다.

- 한정된 자원 (계산시간과 메모리 공간) 을 가지고 **시간(time)**과 **공간(space)**측면에서 효율적인 알고리즘이 필요하다.

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- N 을 N 번 더하는 문제

각 알고리즘이 수행하는 연산의 갯수를 세어 본다  
(단, for 루프 제어 연산 등 세세한 것은 무시)

알고리즘 A	알고리즘 B	알고리즘 C
$\text{sum} \leftarrow n * n;$	$\text{sum} \leftarrow 0;$ for i $\leftarrow$ 1 to n do $\text{sum} \leftarrow \text{sum} + n;$	$\text{sum} \leftarrow 0;$ for i $\leftarrow$ 1 to n do for j $\leftarrow$ 1 to n do $\text{sum} \leftarrow \text{sum} + 1;$

	알고리즘 A	알고리즘 B	알고리즘 C
대입 연산			
덧셈 연산			
곱셈 연산			
나눗셈 연산			
전체 연산수			

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- N 을 N 번 더하는 문제

각 알고리즘이 수행하는 연산의 갯수를 세어 본다  
(단, for 루프 제어 연산 등 세세한 것은 무시)

알고리즘 A	알고리즘 B	알고리즘 C
$\text{sum} \leftarrow n * n;$	$\text{sum} \leftarrow 0;$ for $i \leftarrow 1$ to $n$ do $\text{sum} \leftarrow \text{sum} + n;$	$\text{sum} \leftarrow 0;$ for $i \leftarrow 1$ to $n$ do for $j \leftarrow 1$ to $n$ do $\text{sum} \leftarrow \text{sum} + 1;$

	알고리즘 A	알고리즘 B	알고리즘 C
대입 연산	1	$n + 1$	$n * n + 1$
덧셈 연산		$n$	$n * n$
곱셈 연산	1		
나눗셈 연산			
전체 연산수	2	$2n + 1$	$2n^2 + 1$

# 알고리즘의 수행 시간

```
sample1(A[ ], n)
```

```
{
```

```
     $k = \lfloor n/2 \rfloor$  ;
```

```
    return A[k];
```

```
}
```



# 알고리즘의 수행 시간

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

✓  $n$ 에 관계없이 상수 시간이 소요된다.

# 알고리즘의 수행 시간

```
sample2(A[ ], n)
```

```
{
```

```
    sum  $\leftarrow$  0 ;
```

```
    for  $i \leftarrow 1$  to  $n$ 
```

```
        sum  $\leftarrow$  sum +  $A[i]$  ;
```

```
    return sum ;
```

```
}
```



## 알고리즘의 수행 시간

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

✓  $n$ 에 비례하는 시간이 소요된다.



## 알고리즘의 수행 시간

```
sample3(A[ ], n)
```

```
{
```

```
    sum  $\leftarrow$  0 ;
```

```
    for  $i \leftarrow 1$  to  $n$ 
```

```
        for  $j \leftarrow 1$  to  $n$ 
```

```
            sum  $\leftarrow$  sum +  $A[i] * A[j]$  ;
```

```
    return sum ;
```

```
}
```



## 알고리즘의 수행 시간

```
sample3(A[ ], n)
```

```
{
```

```
    sum  $\leftarrow$  0 ;
```

```
    for  $i \leftarrow 1$  to  $n$ 
```

```
        for  $j \leftarrow 1$  to  $n$ 
```

```
            sum  $\leftarrow$  sum +  $A[i] * A[j]$  ;
```

```
    return sum ;
```

```
}
```

✓  $n^2$ 에 비례하는 시간이 소요된다.

## 알고리즘의 수행 시간

sample4(A[ ],  $n$ )

{

$\text{sum} \leftarrow 0$  ;

**for**  $i \leftarrow 1$  **to**  $n-1$

**for**  $j \leftarrow i+1$  **to**  $n$

$\text{sum} \leftarrow \text{sum} + A[i] * A[j]$  ;

**return**  $\text{sum}$  ;

}



## 알고리즘의 수행 시간

```
sample4(A[ ], n)
```

```
{
```

```
    sum  $\leftarrow$  0 ;
```

```
    for  $i \leftarrow 1$  to  $n-1$ 
```

```
        for  $j \leftarrow i+1$  to  $n$ 
```

```
            sum  $\leftarrow$  sum +  $A[i]*A[j]$  ;
```

```
    return sum ;
```

```
}
```

✓  $n^2$ 에 비례하는 시간이 소요된다.

# 알고리즘의 수행 시간

sample5(A[ ], n)

{

sum  $\leftarrow$  0 ;

**for**  $i \leftarrow 1$  **to**  $n$

**for**  $j \leftarrow 1$  **to**  $n$  {

$k \leftarrow A[1 \dots n]$ 에서 임의로  $\lfloor n/2 \rfloor$  개를 뽑을 때 이들 중 최댓값 ;

sum  $\leftarrow$  sum +  $k$  ;

}

**return** sum ;

}



# 알고리즘의 수행 시간

```
sample5(A[ ], n)
```

```
{
```

```
    sum ← 0 ;
```

```
    for  $i \leftarrow 1$  to  $n$ 
```

```
        for  $j \leftarrow 1$  to  $n$  {
```

```
             $k \leftarrow A[1 \dots n]$ 에서 임의로  $\lfloor n/2 \rfloor$  개를 뽑을 때 이들 중 최댓값 ;
```

```
            sum ← sum +  $k$  ;
```

```
        }
```

```
    return sum ;
```

```
}
```

✓  $n^3$ 에 비례하는 시간이 소요된다.

# 점근적 분석 Asymptotic Analysis

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

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

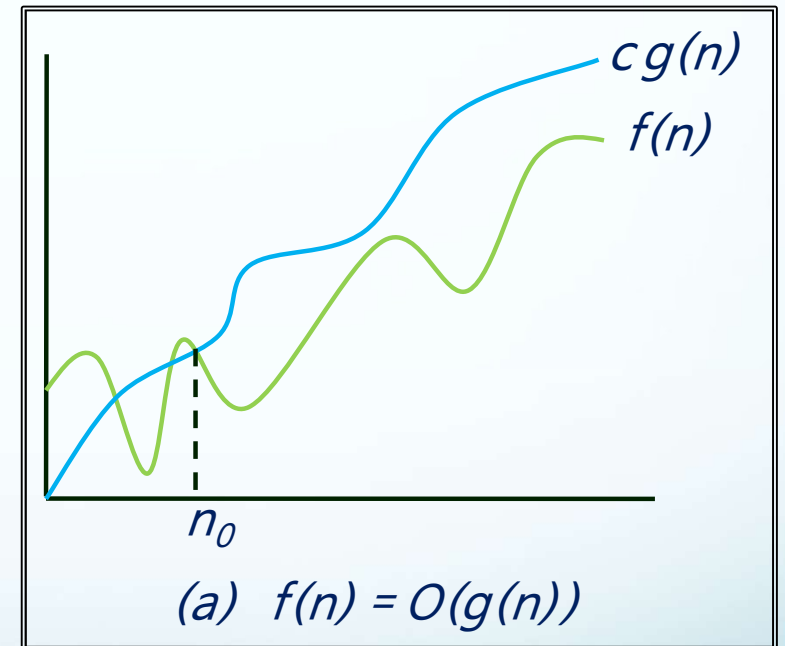
- 점근법 표기법(Asymptotic Notations)

$O, \Omega, \Theta, \omega, o$  표기법

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- 연산 횟수를 대략적(점근적)으로 표기한 것으로 함수의 상한을 의미함.
- 두 개의 함수  $f(n)$ 과  $g(n)$ 이 주어졌을 때, 모든  $n \geq n_0$ 에 대하여  $|f(n)| \leq c|g(n)|$ 을 만족하는 2개의 상수  $c$ 와  $n_0$ 가 존재하면  $f(n) = O(g(n))$  이다.



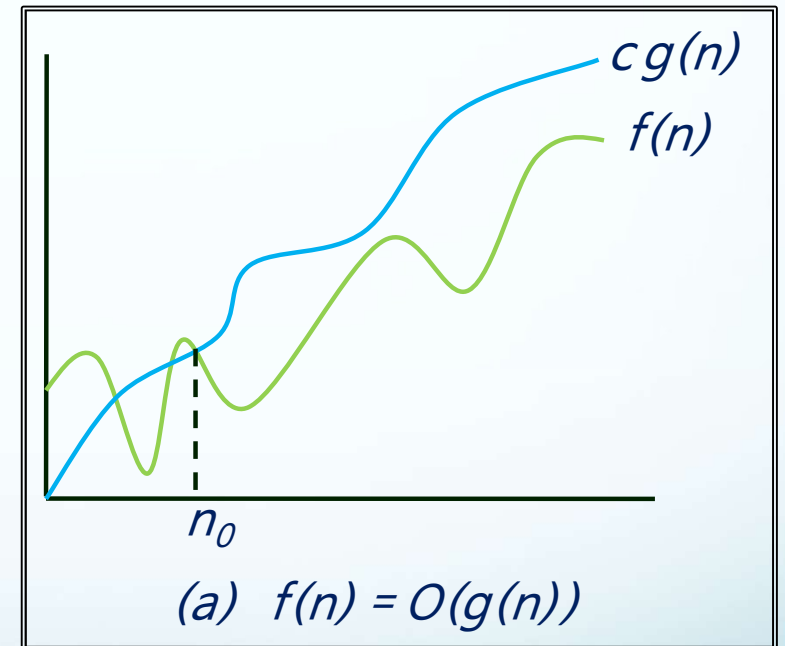


- **Big O notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Big O is a member of a family of notations invented by German mathematicians Paul Bachmann,<sup>[1]</sup> Edmund Landau,<sup>[2]</sup> and others, collectively called **Bachmann–Landau notation** or **asymptotic notation**. The letter O was chosen by Bachmann to stand for Ordnung, meaning the order of approximation.
- In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.<sup>[3]</sup> In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem. Big O notation is also used in many other fields to provide similar estimates.

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- 연산 횟수를 대략적(점근적)으로 표기한 것으로 함수의 상한을 의미함.
- 두 개의 함수  $f(n)$ 과  $g(n)$ 이 주어졌을 때, 모든  $n \geq n_0$ 에 대하여  $|f(n)| \leq c|g(n)|$ 을 만족하는 2개의 상수  $c$ 와  $n_0$ 가 존재하면  $f(n) = O(g(n))$  이다.



# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- 성능의 대략적 표기
  - 예  $n \geq 2$  이면  $n^2 + n + 1 < 2n^2$  이므로  $n^2 + n + 1 = O(n^2)$
  - $n=1000$  인 경우,

$$T(n) = n^2 + n + 1$$

99%

1%

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- Example

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

$$f_1(n) = O(g_1(n)), \quad f_2(n) = O(g_2(n))$$

$$(1) f_1(n) + f_2(n) =$$

$$(2) f_1(n) \cdot f_2(n) =$$



# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- Example

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

$$f_1(n) = O(g_1(n)), \quad f_2(n) = O(g_2(n))$$

$$(1) f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(\max(g_1(n), g_2(n)))$$

$$(2) f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- Example

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

$$f_1(n) = O(g_1(n)), \quad f_2(n) = O(g_2(n))$$

$$(1) f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(\max(g_1(n), g_2(n)))$$

$$(2) f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$g_1 < g_2,$$

$$n > n_0, f_1(n) \leq c_1 g_1(n) < c_1 g_2(n)$$

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

- Example

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

$$f_1(n) = O(g_1(n)), \quad f_2(n) = O(g_2(n))$$

$$(1) f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) = O(\max(g_1(n), g_2(n)))$$

$$(2) f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$$

$$g_1 < g_2,$$

$$n > n_0, f_1(n) \leq c_1 g_1(n) < c_1 g_2(n)$$

$$n > n'_0, f_2(n) \leq c_2 g_2(n)$$

$$n > n^*, f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n) < c_1 g_2(n) + c_2 g_2(n) = (c_1 + c_2) g_2(n)$$

# 시간 효율성 평가

## 빅오 표기법 (Big-O Notation)

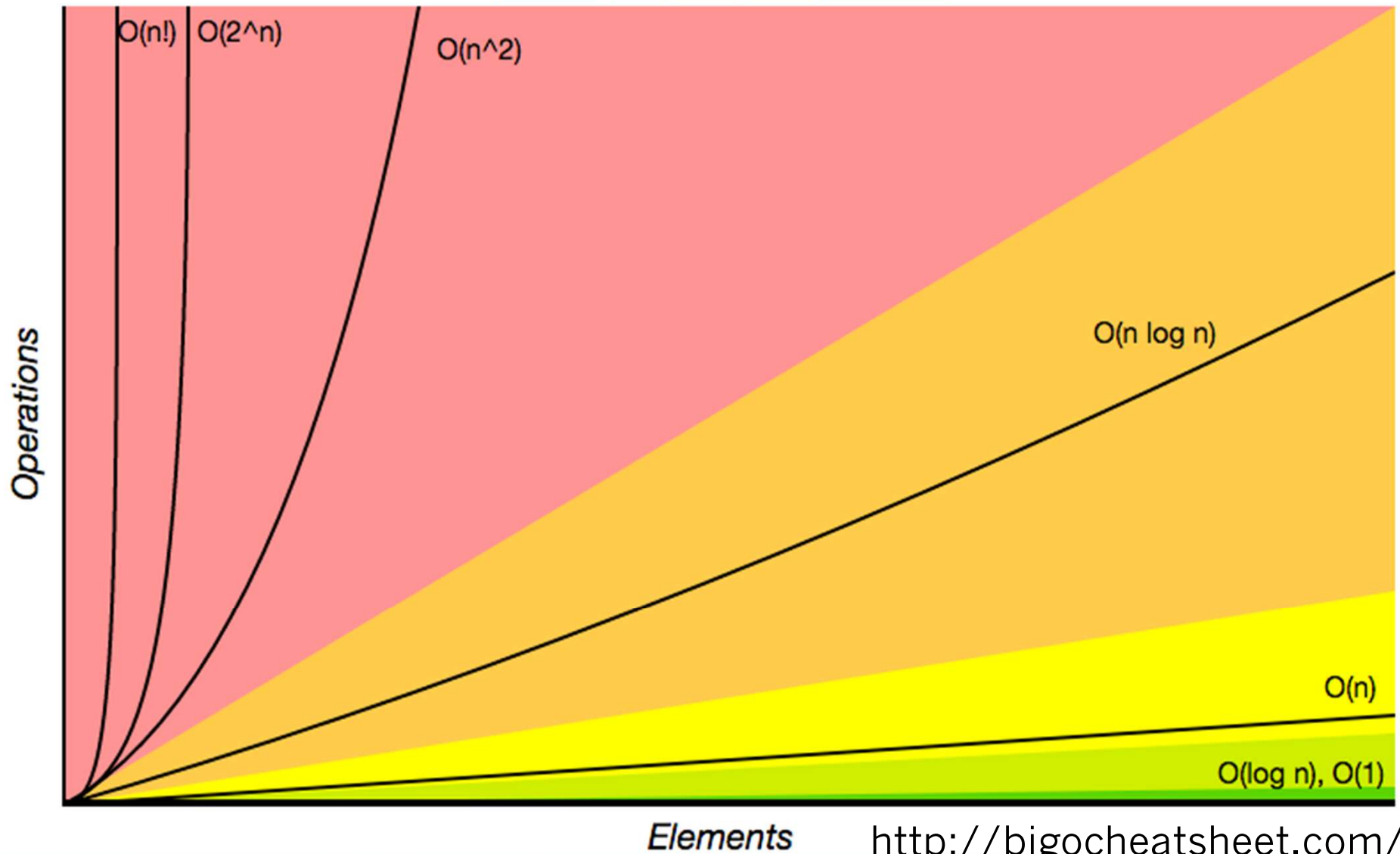
- $O(1)$  : 상수형, constant
- $O(\log n)$  : 로그형, logarithmic
- $O(n)$  : 선형, linear
- $O(n \log n)$  : 로그선형
- $O(n^2)$  : 2차형, quadratic
- $O(n^3)$  : 3차형, cubic
- $O(n^k)$  : k차형, polynomial
- $O(2^n)$  : 지수형, exponential
- $O(n!)$  : 팩토리얼형

시간복잡도	$n$					
	1	2	4	8	16	32
$1$	1	1	1	1	1	1
$\text{Log } n$	0	1	2	3	4	5
$n$	1	2	4	8	16	32
$n \text{ Log } n$	0	2	8	24	64	160
$n^2$	1	4	16	64	256	1024
$n^3$	1	8	64	512	4096	32768
$2^n$	2	4	16	256	65536	4294967296
$n!$	1	2	24	40326	20922789888000	$26313 \times 10^{33}$



# Big-O Complexity Chart

Horrible Bad Fair Good Excellent

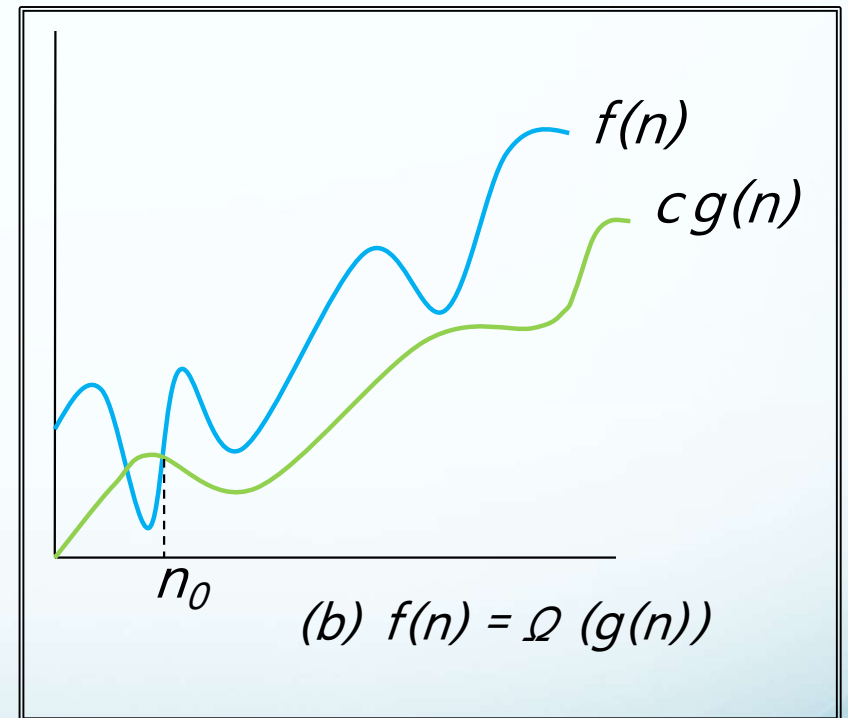


$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < \dots < O(2^n)$$

# 시간 효율성 평가

## 빅오메가 표기법 (Big- $\Omega$ Notation)

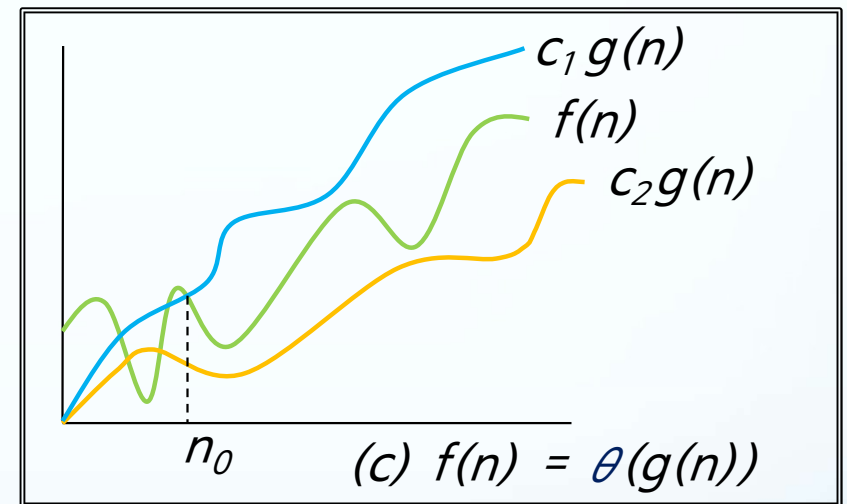
- 함수의 하한을 의미함
- 모든  $n \geq n_0$ 에 대하여  $|f(n)| \geq c|g(n)|$ 을 만족하는 2개의 상수  $c$ 와  $n_0$ 가 존재하면  $f(n) = \Omega(g(n))$ 이다.
- (예)  $n \geq 0$  이면  $2n+1 > n$  이므로  
$$2n+1 = \Omega(n)$$



# 시간 효율성 평가

## 빅세타 표기법 (Big- $\theta$ Notation)

- 빅세타는 함수의 하한인 동시에 상한을 표시한다.
- 모든  $n \geq n_0$ 에 대하여  $c_1 |g(n)| \leq |f(n)| \leq c_2 |g(n)|$ 을 만족하는 3개의 상수  $c_1$ ,  $c_2$ 와  $n_0$ 가 존재하면  $f(n) = \theta(g(n))$ 이다.
- (예)  $n \geq 10$ 이면  $n \leq 2n+1 \leq 3n$ 이므로  $2n+1 = \theta(n)$



# 점근적 표기법

- $O( g(n) )$

- Tight or loose upper bound

- 

- $\Omega( g(n) )$

- Tight or loose lower bound

- 

- $\Theta( g(n) )$

- Tight bound

- 

- $o( g(n) )$

- Loose upper bound

- 

- $\omega( g(n) )$

- Loose lower bound

-

# 점근적 표기법

- $O(g(n))$  빅오
  - Tight or loose upper bound
  - $O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, f(n) \leq c g(n) \}$
- $\Omega(g(n))$  빅오메가
  - Tight or loose lower bound
  - $\Omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, c g(n) \leq f(n) \}$
- $\Theta(g(n))$  빅세타
  - Tight bound
  - $\Theta(g(n)) = \{ f(n) \mid \exists c_1, c_2 > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \}$
- $o(g(n))$  스몰오
  - Loose upper bound
  - $o(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, f(n) < c g(n) \}$
- $\omega(g(n))$  스몰오메가
  - Loose lower bound
  - $\omega(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, c g(n) < f(n) \}$

# 최선, 평균, 최악의 경우의 성능표기

- 알고리즘의 수행시간은 입력 자료 집합에 따라 다를 수 있다.

- 예 순차탐색

- 최선의 경우(best case)



- 최악의 경우(worst case)



- 평균의 경우(average case)



# 최선, 평균, 최악의 경우의 성능표기

- 알고리즘의 수행시간은 입력 자료 집합에 따라 다를 수 있다.
- 예 순차탐색
  - 최선의 경우(best case)
    - > 의미 없는 경우가 많다.
  - 평균의 경우(average case)
    - > 계산하기가 상당히 어렵다.
  - 최악의 경우(worst case)
    - > 가장 널리 사용된다. 계산하기 쉽고 응용에 따라 중요한 의미를 가질 수도 있다.

# 점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현 예 (정렬에서 공부함)
  - 선택정렬
    - $\theta(n^2)$
  - 힙정렬
    - $O(n \log n)$
  - 퀵정렬
    - $O(n^2)$
    - 평균  $\theta(n \log n)$



## Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
<u>Array</u>	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
<u>Stack</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Queue</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Singly-Linked List</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Doubly-Linked List</u>	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$\theta(n)$
<u>Skip List</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n \log(n))$
<u>Hash Table</u>	N/A	$\theta(1)$	$\theta(1)$	$\theta(1)$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
<u>Binary Search Tree</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
<u>Cartesian Tree</u>	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$
<u>B-Tree</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
<u>Red-Black Tree</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
<u>Splay Tree</u>	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	N/A	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
<u>AVL Tree</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$
<u>KD Tree</u>	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(\log(n))$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$	$\theta(n)$

# Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
<u>Quicksort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(\log(n))$
<u>Mergesort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Timsort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$
<u>Heapsort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(1)$
<u>Bubble Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Insertion Sort</u>	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Selection Sort</u>	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$	$O(1)$
<u>Tree Sort</u>	$\Omega(n \log(n))$	$\Theta(n \log(n))$	$O(n^2)$	$O(n)$
<u>Shell Sort</u>	$\Omega(n \log(n))$	$\Theta(n(\log(n))^2)$	$O(n(\log(n))^2)$	$O(1)$
<u>Bucket Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n^2)$	$O(n)$
<u>Radix Sort</u>	$\Omega(nk)$	$\Theta(nk)$	$O(nk)$	$O(n+k)$
<u>Counting Sort</u>	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$	$O(k)$
<u>Cubesort</u>	$\Omega(n)$	$\Theta(n \log(n))$	$O(n \log(n))$	$O(n)$

<http://bigocheatsheet.com/>