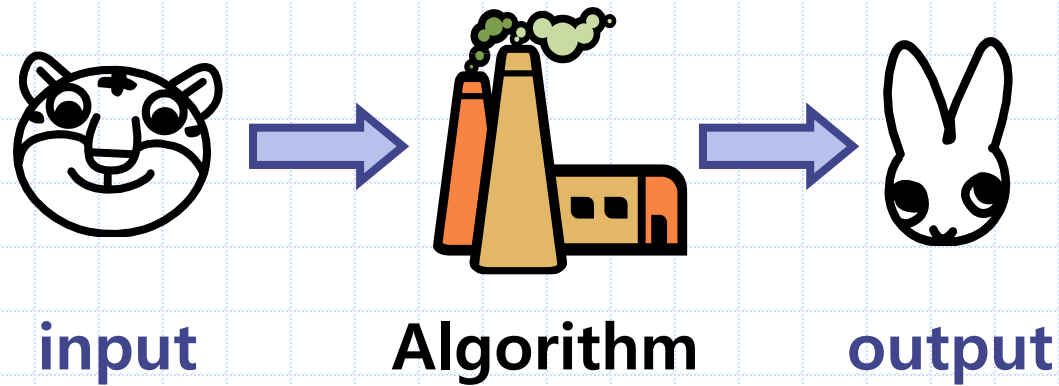


알고리즘 분석



Outline

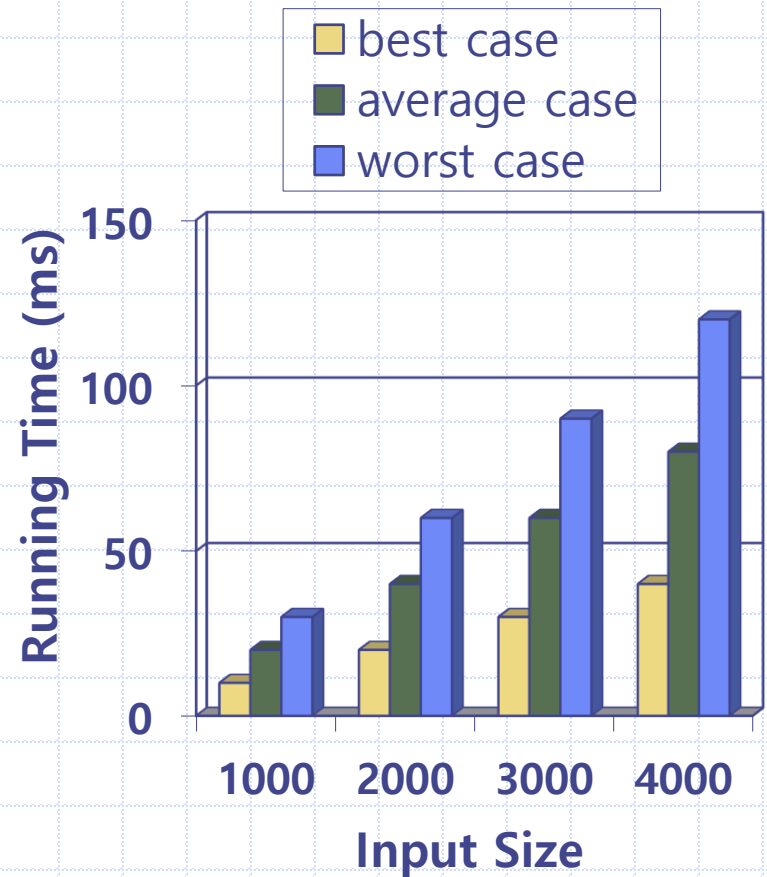
- ◆ 1.1 실행시간
- ◆ 1.2 의사코드
- ◆ 1.3 실행시간 측정과 표기
- ◆ 1.4 전형적인 함수들의 증가율
- ◆ 1.5 알아야 할 수학적 배경
- ◆ 1.6 응용문제

알고리즘 분석

- ◆ **알고리즘(algorithm)**: 주어진 문제를 유한한 시간 내에 해결하는 단계적 절차
- ◆ **데이터구조(data structure)**: 데이터를 조직하고 접근하는 체계적 방식
- ◆ **관심사: "좋은" 알고리즘과 데이터구조를 설계**
- ◆ **"좋은"의 척도**
 - 알고리즘과 데이터구조 작업에 소요되는 실행시간
 - 기억장소 사용량
- ◆ 어떤 알고리즘과 데이터구조를 **"좋다"**고 분류하기 위해서는, 이를 분석하기 위한 정밀한 수단을 필요로 한다

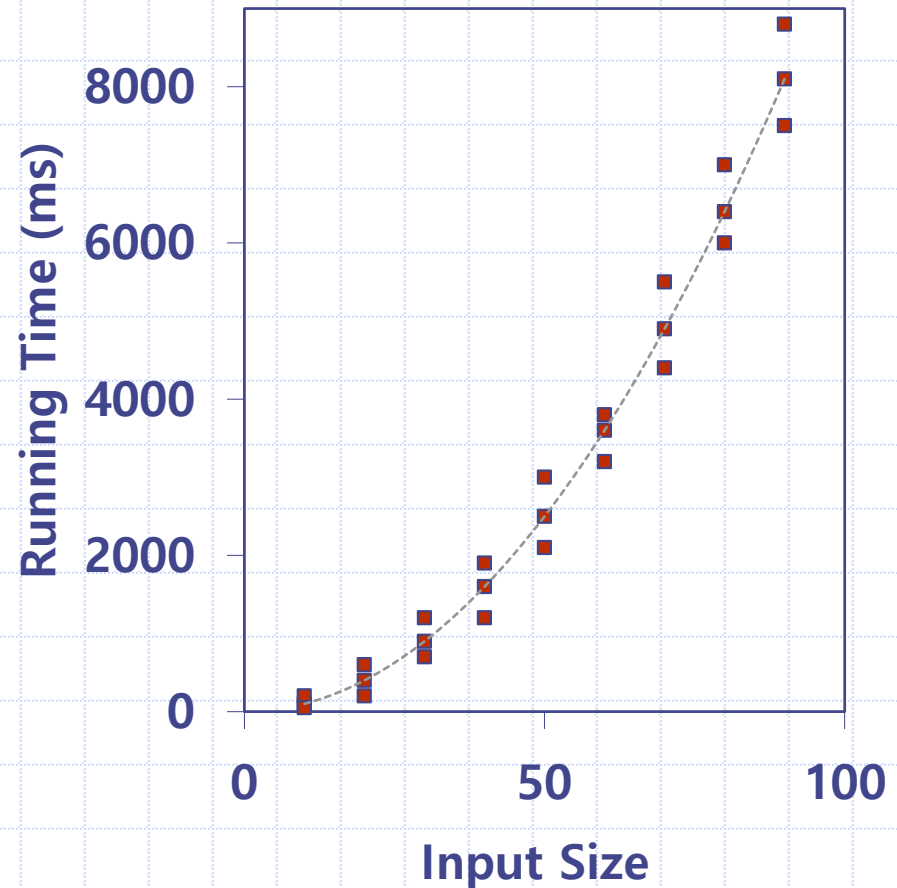
실행시간

- ◆ 대부분의 알고리즘은 입력을 출력으로 변환한다
- ◆ 알고리즘의 **실행시간**(running time)은 대체로 **입력의 크기**(input size)와 함께 성장한다
- ◆ **평균실행시간**(average case running time)은 종종 결정하기 어렵다
- ◆ **최악실행시간**(worst case running time)에 집중
 - 분석이 비교적 용이
 - 게임, 재정, 로봇 등의 응용에서 결정적 요소



실행시간 구하기: 실험적 방법

- ◆ 알고리즘을 구현하는 프로그램을 작성
- ◆ 프로그램을 다양한 크기와 요소로 구성된 입력을 사용하여 실행
- ◆ 시스템콜을 사용하여 실제 실행시간을 정확히 측정
- ◆ 결과를 도표로 작성



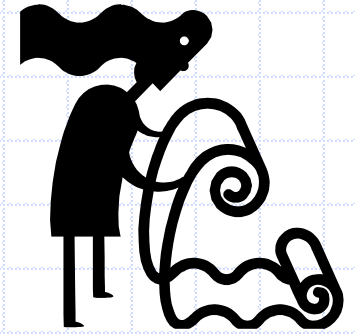
실험의 한계

- ◆ 실험 결과는 실험에 포함되지 않은 입력에 대한 실행시간을 제대로 반영하지 않을 수도 있다
- ◆ 두 개의 알고리즘을 비교하기 위해서는, 반드시 동일한 하드웨어와 소프트웨어 환경이 사용되어야 한다
 - HW: processor, clock rate, memory, disk 등
 - SW: OS, programming language, compiler 등
- ◆ 알고리즘을 완전한 프로그램으로 구현해야 하는데, 이것이 매우 어려울 수가 있다

실행시간 구하기: 이론적 방법



- ◆ 모든 입력 가능성을 고려한다
- ◆ 하드웨어나 소프트웨어와 무관하게 알고리즘의 속도 평가 가능
 - 실행시간을 입력 크기, n 의 함수로 규정
- ◆ 알고리즘을 구현한 프로그램 대신, 고급언어, 구체적으로는, 의사코드로 표현된 알고리즘을 사용한다



의사코드

- ◆ **의사코드** (pseudo-code): 알고리즘을 설명하기 위한 고급언어
 - 컴퓨터가 아닌, 인간에게 읽히기 위해 작성됨
 - 저급의 상세 구현내용이 아닌, 고급 개념을 소통하기 위해 작성됨
- ◆ 자연어 문장보다 더 구조적이지만, 프로그래밍 언어보다 덜 상세함
- ◆ 알고리즘을 설명하는데 선호되는 표기법

- ◆ 예: 배열의 최대값 원소 찾기

Alg *arrayMax*(A, n)

input array A of n integers

output maximum element of A

1. $currentMax \leftarrow A[0]$
2. **for** $i \leftarrow 1$ **to** $n - 1$
 - if** ($A[i] > currentMax$)
 $currentMax \leftarrow A[i]$
3. **return** $currentMax$

의사코드 문법

◆ 제어(control flow)

- **if** (*exp*) ...
 [elseif (*exp*) ...]*****
 [else ...]
- **for** *var* ← *exp*₁ **to** *exp*₂
 ...
 for each *var* ∈ *exp*
 ...
 while (*exp*)
 ...
 do
 ...
 while (*exp*)

◆ 주의: 들여쓰기(indentation)로 범위(scope)를 정의

◆ 연산(arithmetic)

←	치환(assignment)
=, <, ≤, >, ≥	관계 연산자
&, , !	논리 연산자
$s_1 \leq n^2$	첨자 등 수학적 표현 허용

◆ 메소드(method) 정의, 반환, 호출

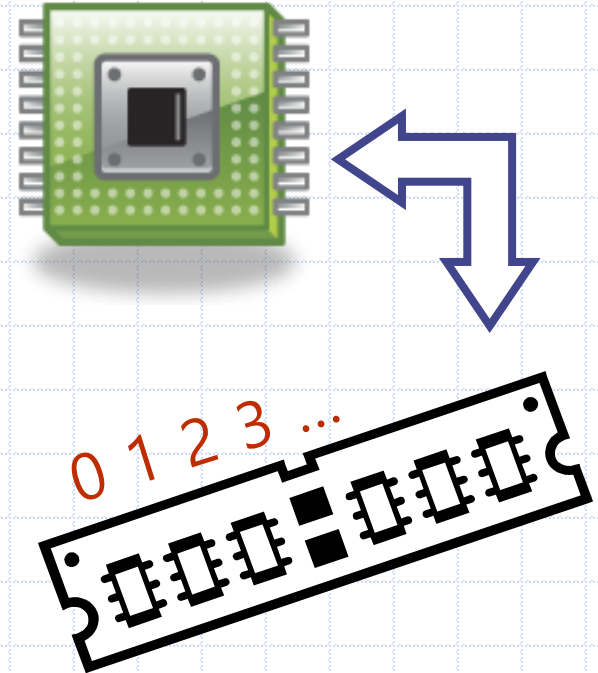
- **Alg** *method* (**[arg** [, *arg*]*****)
 ...
 ■ **return** [*exp* [, *exp*]*****]
 ■ *method* (**[arg** [, *arg*]*****)

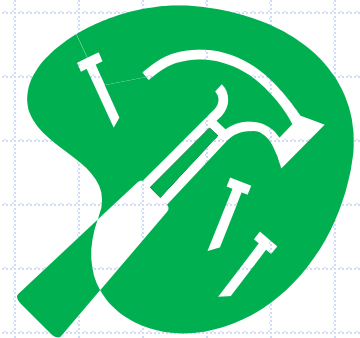
◆ 주석(comments)

input ...
output ...
{This is a comment}

임의접근기계 모델

- ◆ 임의접근기계(random access machine, RAM) 모델
 - 하나의 중앙처리장치(CPU)
 - 무제한의 메모리셀(memory cell), 각각의 셀은 임의의 수나 문자 데이터를 저장함
- ◆ 메모리셀들은 순번으로 나열되며, 어떤 셀에 대한 접근이라도 동일한(즉, 상수) 시간단위가 소요됨





원시작업

◆ 원시작업(primitive operation)이란:

- 알고리즘에 의해 수행되는 기본적인 계산들
- 의사코드로 표현 가능
- 프로그래밍 언어와는 대체로 무관
- 정밀한 정의는 중요하지 않음
- 임의접근기계 모델에서 수행시 상수시간이 소요된다고 가정

◆ 예

- 산술식/논리식의 평가(EXP)
- 변수에 특정값을 치환(ASS)
- 배열원소 접근(IND)
- 메소드 호출(CAL)
- 메소드로부터 반환(RET)

원시작업 수 세기

- ◆ 의사코드를 조사함으로써, 알고리즘에 의해 실행되는 원시작업의 최대 개수를 **입력크기의 함수** 형태로 결정할 수 있다

Alg *arrayMax*(*A*, *n*)

input array *A* of *n* integers

output maximum element of *A*

1. *currentMax* $\leftarrow A[0]$

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

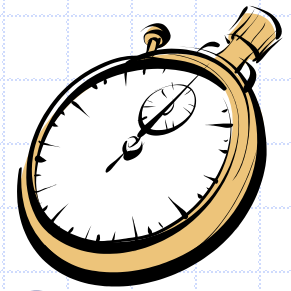
if (*A*[*i*] > *currentMax*)

currentMax $\leftarrow A[i]$

 {increment counter *i*}

3. **return** *currentMax*

{operations	count}
{IND, ASS	2}
{ASS, EXP	1 + <i>n</i> }
{IND, EXP	2(<i>n</i> - 1)}
{IND, ASS	2(<i>n</i> - 1)}
{EXP, ASS	2(<i>n</i> - 1)}
{RET	1}
{Total	7 <i>n</i> - 2}



실행시간 추정

- ◆ **arrayMax**는 최악의 경우 $7n - 2$ 개의 원시작업을 실행한다
- ◆ 다음과 같이 정의하자
 - a = 가장 빠른 원시작업 실행에 걸리는 시간
 - b = 가장 느린 원시작업 실행에 걸리는 시간
- ◆ 그리고 $T(n)$ 을 **arrayMax**의 최악인 경우의 시간이라 놓으면, 다음이 성립
$$a(7n - 2) \leq T(n) \leq b(7n - 2)$$
- ◆ 즉, 실행시간 $T(n)$ 은 두 개의 선형함수 사이에 놓이게 된다

실행시간의 증가율

- ◆ 하드웨어나 소프트웨어 환경을 변경하면:
 - $T(n)$ 에 상수 배수 만큼의 영향을 주지만,
 - $T(n)$ 의 증가율을 변경하지는 않는다
- ◆ 따라서 선형의 **증가율**(growth rate)을 나타내는 실행시간 $T(n)$ 은 **arrayMax**의 고유한 속성이다



Big-Oh와 증가율

- ◆ Big-Oh 표기법은 함수의 증가율의 상한(upper bound)을 나타낸다
- ◆ " $f(n) = O(g(n))$ "이라 함은 " $f(n)$ 의 증가율은 $g(n)$ 의 증가율을 넘지 않음"을 말한다
- ◆ Big-Oh 표기법을 사용함으로써, 증가율에 따라 함수들을 서열화할 수 있다

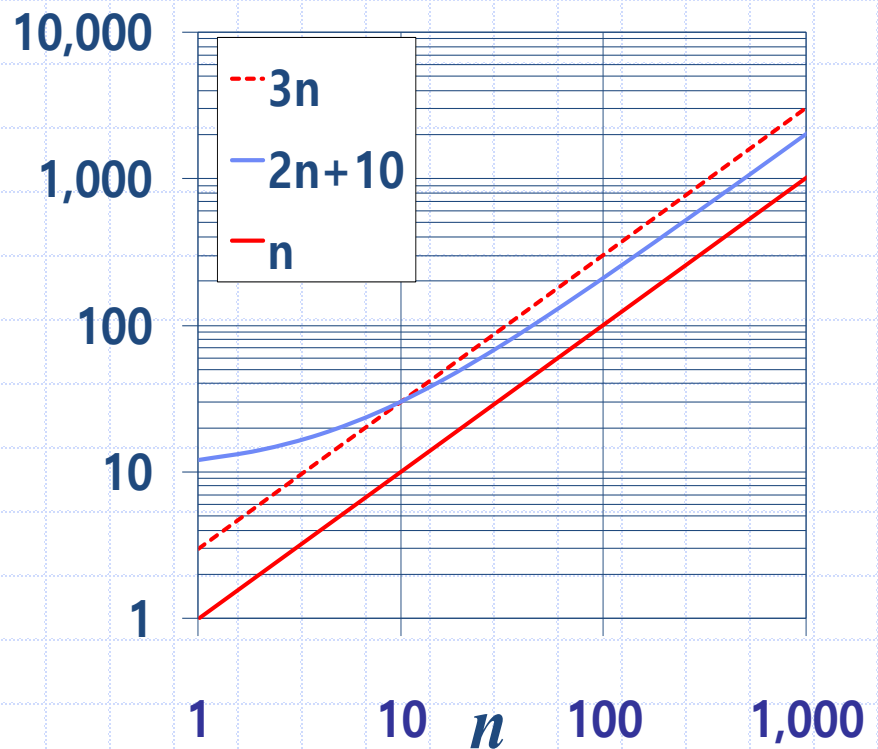
	$f(n) = O(g(n))$	$g(n) = O(f(n))$
$g(n)$ 의 증가율이 더 빠르면	yes	no
$f(n)$ 의 증가율이 더 빠르면	no	yes
둘이 같으면	yes	yes

Big-Oh 표기법

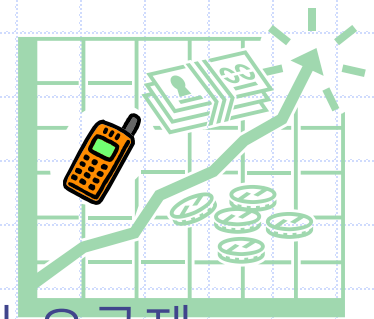
◆ 주어진 두 개의 함수 $f(n)$ 과 $g(n)$ 에 관해, 만약 모든 정수 $n \geq n_0$ 에 대해 $f(n) \leq c \cdot g(n)$ 가 성립하는 실수의 상수 $c > 0$ 및 정수의 상수 $n_0 \geq 1$ 가 존재하면 " $f(n) = O(g(n))$ " 이라고 말한다

◆ 예: $2n + 10 = O(n)$

- $2n + 10 \leq c \cdot n$
- $(c - 2)n \geq 10$
- $n \geq 10/(c - 2)$
- $c = 3$ 및 $n_0 = 10$ 으로 성립



통신요금제 비유



◆ **전제 1:** 총 통화도수 n 에 대한 두 요금제의 요금 계산방식

- A 요금제: $2n + 10$
- B 요금제: $c \cdot n$

◆ **관찰:** $c = 3$ 에 대해 10도수 이상 통화할 경우, A 요금은 B 요금을 넘지 않는다

◆ 즉, $n \geq n_0$ 에 대해 $2n + 10 \leq c \cdot n$ 이 성립하는 실수의 상수 $c = 3 > 0$ 및 정수의 상수 $n_0 = 10 \geq 1$ 가 존재하므로, " $2n + 10 = O(n)$ "

◆ **전제 2:** 또 하나의 요금제

- C 요금제: $c \cdot n^2$

◆ **관찰:** $c = 1$ 에 대해 5도수 이상 통화할 경우, A 요금은 C 요금을 넘지 않는다

◆ 즉, $n \geq n_0$ 에 대해 $2n + 10 \leq c \cdot n$ 가 성립하는 실수의 상수 $c = 1 > 0$ 및 정수의 상수 $n_0 = 5 \geq 1$ 가 존재하므로, " $2n + 10$ is $O(n^2)$ "

◆ 따라서, " $2n + 10 = O(n)$ ", " $2n + 10 = O(n^2)$ " 모두 성립

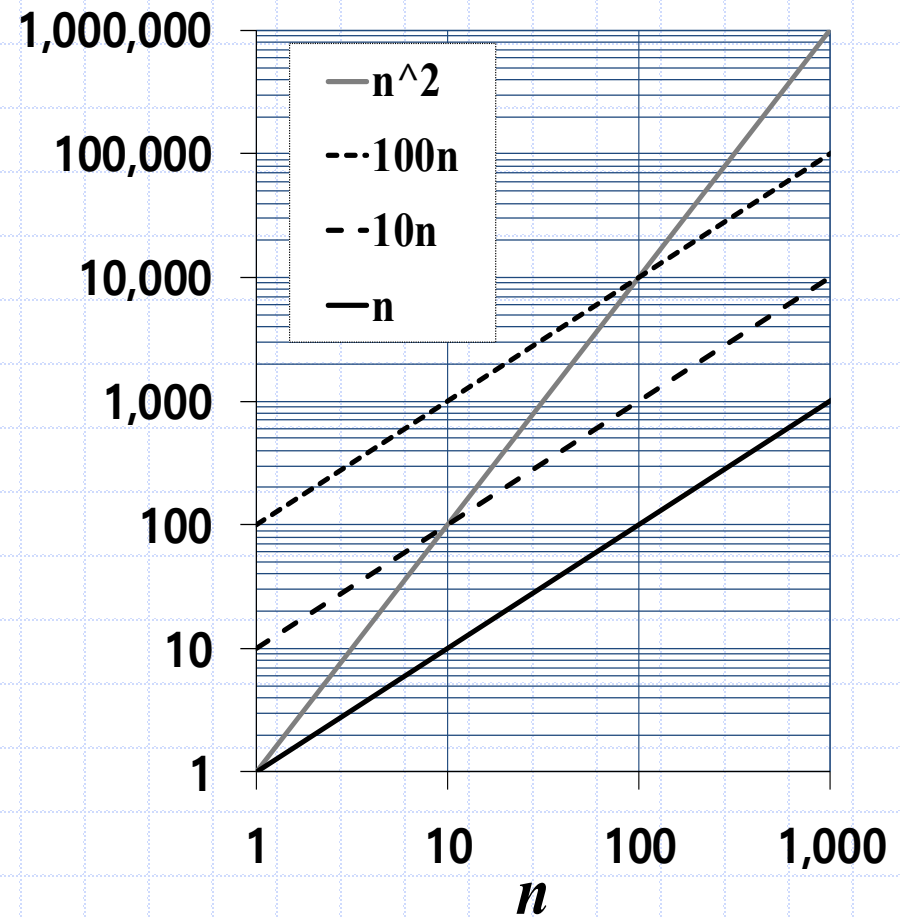
Big-Oh 예

◆ 예: $n^2 \neq O(n)$

- $n^2 \leq c \cdot n$
- $n \leq c$
- 관찰: c 가 상수여야
하므로, 위의 부등식은
성립시킬 수 없다

◆ 통신요금제 비유

- A: n^2
- B: $c \cdot n$
- 관찰: 어떤 c 에
대해서도 A 요금이 B
요금을 넘지 않게 되는
최저통화도수는 없다



Big-Oh 예

◆ $7n - 2$

- $7n - 2 = O(n)$
- $n \geq n_0$ 에 대해 $7n - 2 \leq c \cdot n$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 7$ 및 $n_0 = 1$ 로 성립

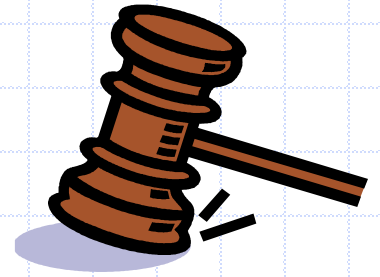
◆ $3n^3 + 20n^2 + 5$

- $3n^3 + 20n^2 + 5 = O(n^3)$
- $n \geq n_0$ 에 대해 $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 4$ 및 $n_0 = 21$ 로 성립

◆ $3 \log n + \log \log n$

- $3 \log n + \log \log n = O(\log n)$
- $n \geq n_0$ 에 대해 $3 \log n + \log \log n \leq c \cdot \log n$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 4$ 및 $n_0 = 2$ 로 성립

Big-Oh 계산 요령



- ◆ $f(n)$ 이 상수이면, $f(n) = O(c)$, 또는 $O(1)$
- ◆ $f(n)$ 이 차수 d 의 다항식이면, $f(n) = O(n^d)$, 즉,
 1. 낮은 차수의 항들을 탈락시킨 후
 2. 상수계수를 탈락시킨다
- ◆ **최소한의** 함수계열을 사용
 - " $2n = O(n^2)$ " 대신 " $2n = O(n)$ " 라고 말할 것
- ◆ 해당 함수계열 중 가장 **단순한** 표현을 사용
 - " $3n + 5 = O(3n)$ " 대신 " $3n + 5 = O(n)$ "이라고 말할 것

점근분석

◆ 알고리즘을 점근분석(asymptotic analysis) 함으로써 big-Oh 표기법에 의한 실행시간을 구할 수 있다

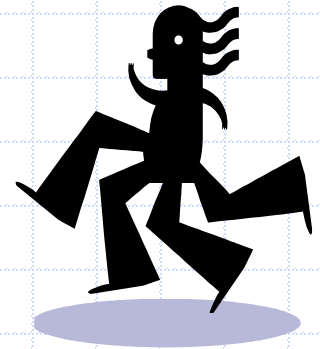
◆ 점근분석을 수행하기 위해서는,

1. 최악의 원시작업 실행회수를 입력크기의 함수로서 구한다
2. 이 함수를 big-Oh 표기법으로 나타낸다

◆ 예

1. 알고리즘 `arrayMax`가 최대 $7n - 2$ 개의 원시작업을 실행한다는 것을 구한다
2. "알고리즘 `arrayMax`는 $O(n)$ 시간에 수행된다"고 말한다

◆ 상수계수와 낮은 차수의 항들은 결국 탈락되므로, 원시작업 수를 계산할 때부터 이들을 무시할 수 있다



분석의 지름길

◆ 다중의 원시작업

- 하나의 식에 나타나는 여러 개의 원시작업을 하나로 계산
- 예: $O(c)$

$sum \leftarrow sum + (salary + bonus) \times (1 - tax)$

◆ 반복문

- 반복문의 실행시간 \times 반복회수
- 예: $O(n)$

for $i \leftarrow 1$ to n

$k \leftarrow k + 1$

$sum \leftarrow sum + i$

◆ 중첩 반복문

- 반복문의 실행시간 $\times \Pi$ 각 반복문의 크기
- 예: $O(n^2)$

for $i \leftarrow 1$ to n

for $j \leftarrow 1$ to n

$k \leftarrow k + 1$

분석의 지름길 (conti.)

◆ 연속문

- 각 문의 실행시간을 합산, 즉, 이들 중 최대값을 선택

- 예: $O(n^2)$

for $i \leftarrow 0$ to $n - 1$ $\{O(n)\}$

$A[i] \leftarrow 0$

for $i \leftarrow 0$ to $n - 1$ $\{O(n)\}$

 for $j \leftarrow 0$ to $n - 1$ $\{O(n^2)\}$

$A[i] \leftarrow A[i] + A[j]$

◆ 조건문

- 조건검사의 실행시간에 if-else 절의 실행시간 중 큰 것을 합산

- 예: $O(n)$

if ($k = 0$) $\{O(c)\}$

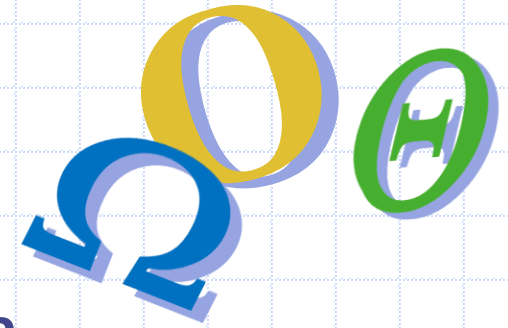
 return $\{O(c)\}$

else

 for $i \leftarrow 1$ to n $\{O(n)\}$

$j \leftarrow j + 1$

Big-Oh의 친척들

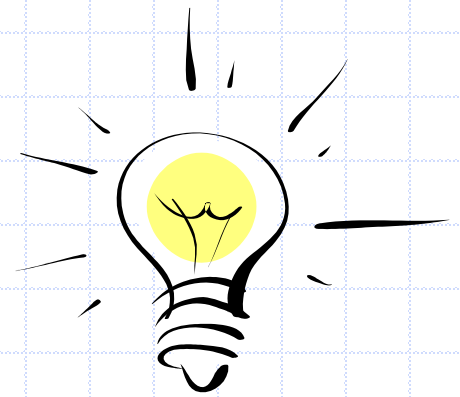


◆ Big-Omega

- $n \geq n_0$ 에 대해 $f(n) \geq c \cdot g(n)$ 이 성립하는 상수 $c > 0$ 및 정수의 상수 $n_0 \geq 1$ 가 존재하면 " $f(n) = \Omega(g(n))$ "
- Big-Oh가 함수의 증가율의 **상한**(upper bound)을 나타내는데 반해, big-Omega는 함수의 증가율의 **하한**(lower bound)을 나타낸다

◆ Big-Theta

- $n \geq n_0$ 에 대해 $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ 이 성립하는 상수 $c' > 0$, $c'' > 0$ 및 정수의 상수 $n_0 \geq 1$ 가 존재하면 " $f(n) = \Theta(g(n))$ "
- 다시 말해, " $f(n) = O(g(n))$ "인 동시에 " $f(n) = \Omega(g(n))$ "이면, " $f(n) = \Theta(g(n))$ "
- Big-Theta는 함수의 증가율의 **상한**과 **하한**을 모두 나타내므로 **동일함수**를 나타낸다



점근표기에 관한 직관

◆ Big-Oh

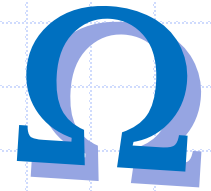
- 점근적으로 $f(n) \leq g(n)$ 이면, " $f(n) = O(g(n))$ "

◆ Big-Omega

- 점근적으로 $f(n) \geq g(n)$ 이면, " $f(n) = \Omega(g(n))$ "

◆ Big-Theta

- 점근적으로 $f(n) = g(n)$ 이면, " $f(n) = \Theta(g(n))$ "



Big-Omega 예

◆ $5n^2$

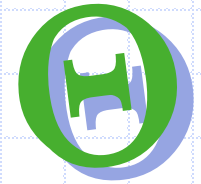
- $5n^2 = \Omega(n^2)$
- $n \geq n_0$ 에 대해 $5n^2 \geq c \cdot n^2$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 5$ 및 $n_0 = 1$ 로 성립

◆ $5n^2$

- $5n^2 = \Omega(n)$
- $n \geq n_0$ 에 대해 $5n^2 \geq c \cdot n$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 1$ 및 $n_0 = 1$ 로 성립

◆ $3 \log n + \log \log n$

- $3 \log n + \log \log n = \Omega(\log n)$
- $n \geq n_0$ 에 대해 $3 \log n + \log \log n \geq c \cdot \log n$ 이 성립하는 $c > 0$ 및 $n_0 \geq 1$ 가 필요
- $c = 3$ 및 $n_0 = 2$ 로 성립



Big-Theta 예

◆ $5n^2$

- $5n^2 = \Theta(n^2)$
- $n \geq n_0$ 에 대해 $c' \cdot n^2 \leq 5n^2 \leq c'' \cdot n^2$ 이 성립하는 $c', c'' > 0$ 및 $n_0 \geq 1$ 가 필요
- $c' = c'' = 5$ 및 $n_0 = 1$ 로 성립

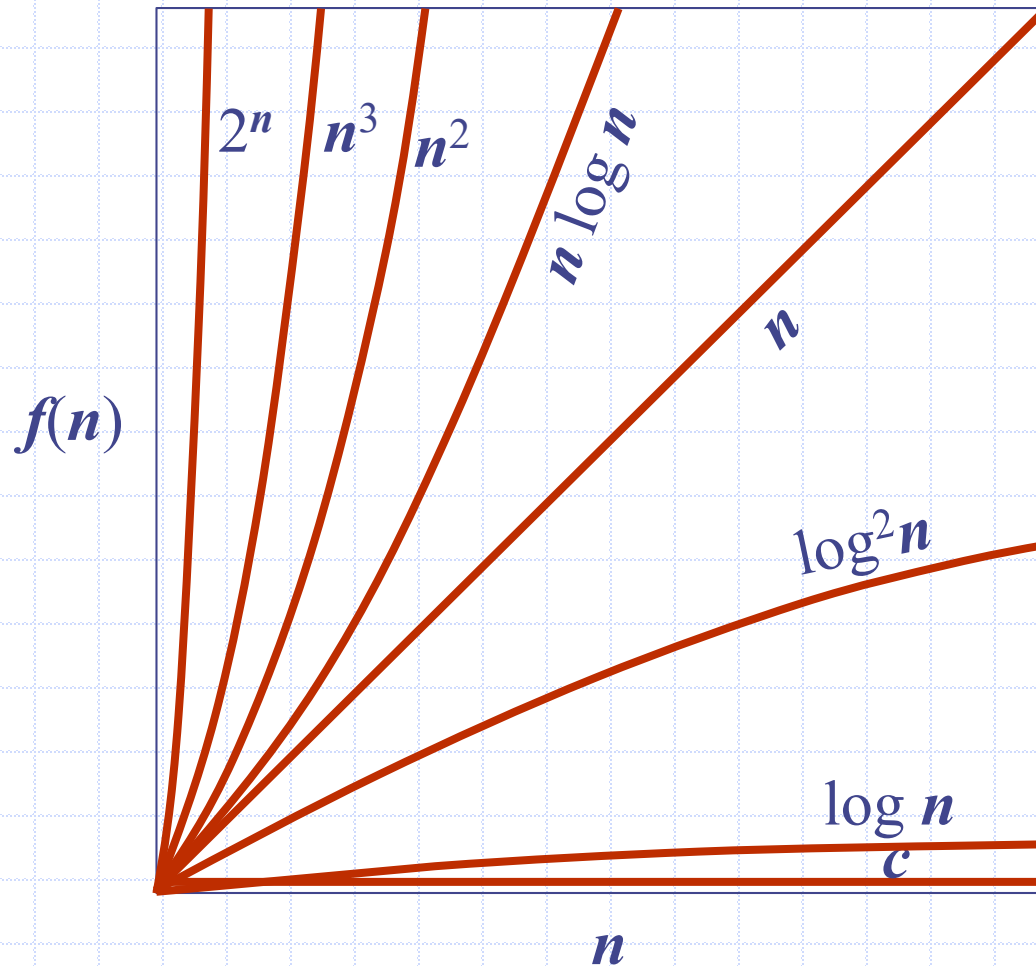
◆ $3 \log n + \log \log n$

- $3 \log n + \log \log n = \Theta(\log n)$
- $n \geq n_0$ 에 대해 $c' \cdot \log n \leq 3 \log n + \log \log n \leq c'' \cdot \log n$ 이 성립하는 $c', c'' > 0$ 및 $n_0 \geq 1$ 가 필요
- $c' = 3, c'' = 4$ 및 $n_0 = 2$ 로 성립

전형적인 증가율

함수	이름	$f(10^2)$	$f(10^3)$	$f(10^4)$	$f(10^5)$
c	상수(constant)	1	1	1	1
$\log n$	로그(logarithmic)	7	10	14	18
$\log^2 n$	로그제곱(log-squared)	49	100	200	330
n	선형(linear)	100	1,000	10,000	100,000
$n \log n$	로그선형(log-linear)	700	10,000	140,000	1.8×10^6
n^2	2차(quadratic)	10,000	10^6	10^8	10^{10}
n^3	3차(cubic)	10^6	10^9	10^{12}	10^{15}
2^n	지수(exponential)	10^{30}	10^{300}	10^{3000}	10^{30000}

전형적인 증가 함수들의 플롯





알아야 할 수학적 배경

◆ 합계(summations)

- $\sum_{i=0}^n a^i = 1 + a + a^2 + \dots + a^n = (1 - a^{n+1})/(1 - a) \quad \{\text{for } a > 0\}$
- $\sum_{i=1}^n i = n(n+1)/2$
- $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$

◆ 로그(logarithms)

- $\log_b(xy) = \log_b x + \log_b y$
- $\log_b(x/y) = \log_b x - \log_b y$
- $\log_b x^a = a \log_b x$
- $\log_b a = (\log_x a) / \log_x b$

◆ 지수(exponentials)

- $a^{b+c} = a^b a^c$
- $a^{b-c} = a^b / a^c$
- $a^{bc} = (a^b)^c$
- $a^{\log_a b} = b$

응용문제: 행렬에서 특정 원소 찾기

- ◆ $n \times n$ 배열 A 의 원소들 중 특정 원소 x 를 찾는 알고리즘 **findMatrix**를 작성하고자 한다
 - ◆ 알고리즘 **findMatrix**는 A 의 행들을 반복하며, x 를 찾거나 또는 A 의 모든 행들에 대한 탐색을 마칠 때까지, 각 행에 대해 알고리즘 **findRow**를 호출한다
- A. 위에 의도한 바와 일치하도록 알고리즘 **findMatrix**를 의사코드로 작성하라
- B. **findMatrix**의 최악실행시간을 n 에 관해 구하라
- C. 이는 선형시간 알고리즘인가? 왜 그런지 또는 왜 아닌지 설명하라

Alg *findRow*(A, x)

input array A of n elements,
element x

output the index i such that $x = A[i]$ or -1 if no element of A is equal to x

1. $i \leftarrow 0$
2. **while** ($i < n$)
 if ($x = A[i]$)
 return i
 else
 $i \leftarrow i + 1$
3. **return** -1

해결

A. 오른 편에 **findMatrix** 보임

B. 최악의 실행시간은 $O(n^2)$

- 최악의 경우, 원소 x 는 검사 대상인 $n \times n$ 배열의 맨 마지막 원소다
- 이 경우, **findMatrix**는 **findRow**를 n 번 호출하게 된다 – **findRow**는 행마다 n 개의 원소 모두를 탐색해야 하며 마지막 호출에 가서야 x 를 찾는다
- 그러므로, **findRow**를 호출할 때마다 n 회의 비교가 수행된다
- **findRow**가 n 회 호출되므로 총 $n \times n$ 회의 작업을 수행하는 것이 되며, 이는 $O(n^2)$ 실행시간이 된다

C. 선형시간 알고리즘이 아니다

- n^2 은 2차시간(quadratic-time)이다 – 즉, 실행시간이 입력 크기에 **제곱비례**한다. 선형시간이 되려면, 실행시간이 입력 크기에 **정비례**해야 해야 할 것이다

Alg **findMatrix**(A, x)

input array A of $n \times n$ elements, element x

output the location of x in A or a failure message if no element of A is equal to x

1. $r \leftarrow 0$
2. **while** ($r < n$)
 - $i \leftarrow \text{findRow}(A[r], x)$
 - if** ($i \geq 0$)
 - write**(“found at”, r, i)
 - return**
 - else**
 - $r \leftarrow r + 1$
3. **write**(“not found”)
4. **return**

응용문제: 비트행렬에서 최대 1행 찾기

- ◆ $n \times n$ 배열 A 의 각 행은 1과 0으로만 구성되며, A 의 어느 행에서나 1들은 해당 행의 0들보다 앞서 나온다고 가정하자
- ◆ A 가 이미 주기억장치에 존재한다고 가정하고, 가장 많은 1을 포함하는 행을 $O(n)$ 시간에 찾는 알고리즘 `mostOnes(A, n)`를 의사코드로 작성하라
- ◆ 예: 8×8 배열 A
 - 6행이 가장 많은 1을 포함

	0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	1	1	1	1	1	1	0	0
4	1	1	1	1	0	0	0	0
5	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	0
7	1	1	1	1	1	0	0	0

A

응용문제: 비트행렬에서 최대 1 행 찾기 (conti.)

Alg **mostOnesButSlow**(A, n)
input bit matrix $A[n \times n]$
output the row of A with most 1's

```
1. row  $\leftarrow jmax \leftarrow 0$ 
2. for  $i \leftarrow 0$  to  $n - 1$ 
     $j \leftarrow 0$ 
    while (( $j < n$ ) & ( $A[i, j] = 1$ ))
         $j \leftarrow j + 1$ 
    if ( $j > jmax$ )
        row  $\leftarrow i$ 
         $jmax \leftarrow j$ 
3. return row
```

◆ 주의: 알고리즘 **mostOnesButSlow**는 1이 가장 많은 행을 찾기는 하지만, 실행시간이 $O(n)$ 이 아니라 $O(n^2)$ 이다

	0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	1	1	1	1	1	1	0	0
4	1	1	1	1	0	0	0	0
5	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	0
7	1	1	1	1	1	0	0	0

A

해결

◆ 해결 방법은 다음과 같다

1. 행렬의 좌상 셀에서 출발
2. 0이 발견될 때까지 행렬을 가로질러 간다
3. 1이 발견될 때까지 행렬을 내려간다
4. 마지막 행 또는 열을 만날 때까지 위 2, 3 단계를 반복
5. 1을 가장 많이 가진 행은 가로지른 마지막 행이다

◆ 최대 $2n$ 회의 비교를 수행하므로, 명백히 $O(n)$ -시간 알고리즘!

◆ 두 가지 버전으로 작성 가능

	0	1	2	3	4	5	6	7
0	1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	0	0
2	1	0	0	0	0	0	0	0
3	1	1	1	1	1	1	0	0
4	1	1	1	1	0	0	0	0
5	0	0	0	0	0	0	0	0
6	1	1	1	1	1	1	1	0
7	1	1	1	1	1	0	0	0

A

해결 (conti.)

Alg *mostOnes*(A, n) {ver.1}
input bit matrix $A[n \times n]$
output the row of A with most 1's

```
1.  $i \leftarrow j \leftarrow 0$ 
2. while (1)
    while ( $A[i, j] = 1$ )
         $j \leftarrow j + 1$ 
        if ( $j = n$ )
            return  $i$ 
     $row \leftarrow i$ 
    while ( $A[i, j] = 0$ )
         $i \leftarrow i + 1$ 
        if ( $i = n$ )
            return  $row$ 
```

{Total $O(n)$ }

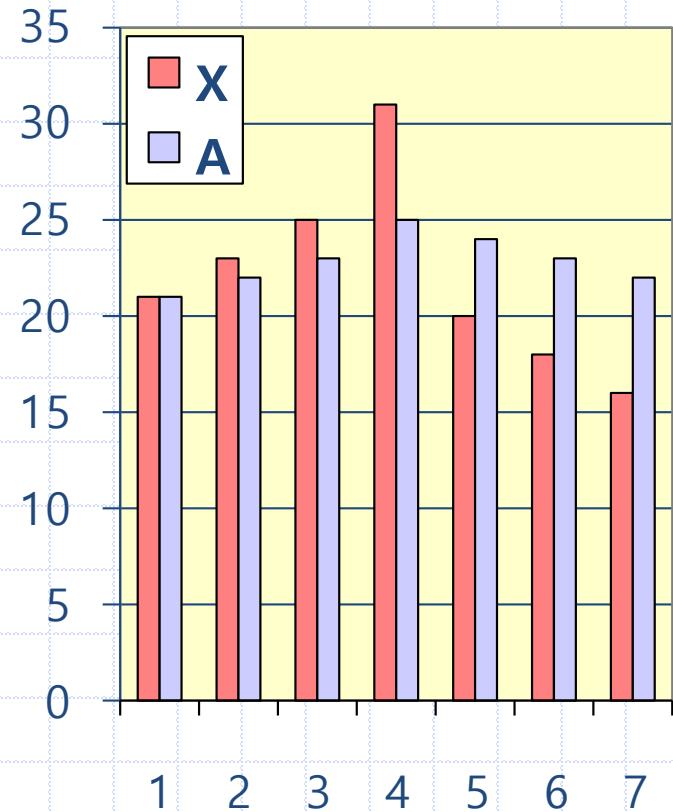
Alg *mostOnes*(A, n) {ver.2}
input bit matrix $A[n \times n]$
output the row of A with most 1's

```
1.  $i \leftarrow j \leftarrow row \leftarrow 0$ 
2. while (( $i < n$ ) & ( $j < n$ ))
    if ( $A[i, j] = 0$ )
         $i \leftarrow i + 1$ 
    else
         $row \leftarrow i$ 
         $j \leftarrow j + 1$ 
3. return  $row$ 
```

{Total $O(n)$ }

응용문제: 누적평균

- ◆ 배열 X 의 i -번째
누적평균(prefix average)이란
 X 의 i -번째에 이르기까지의 $(i + 1)$ 개 원소들의 평균이다 - 즉,
 $A[i] = (X[0] + X[1] + \dots + X[i]) / (i + 1)$
- ◆ 배열 X 의 누적평균(prefix average) 배열 A 를 구하는 알고리즘을 의사코드로 작성하라
- ◆ 응용: 경제, 통계 분야
 - 오르내림 변동을 순화시킴으로써 대략적 추세를 얻어내기 위해 사용
 - 부동산, 주식, 펀드 등에 활용



해결: 누적평균 (Ver. 1)

- ◆ 아래 알고리즘은 정의를 이용하여 누적평균값들을 2차 시간(quadratic time)에 구한다

Alg <i>prefixAverages1</i> (X, n)	{ver.1}
input array X, A of n integers	
output array A of prefix averages of X	
1. for $i \leftarrow 0$ to $n - 1$	{ n }
$sum \leftarrow 0$	{ n }
for $j \leftarrow 0$ to i	{ $1 + 2 + \dots + n$ }
$sum \leftarrow sum + X[j]$	{ $1 + 2 + \dots + n$ }
$A[i] \leftarrow sum / (i + 1)$	{ n }
2. return	{1}
	{Total $O(n^2)$ }

해결: 누적평균 (Ver. 2)

- ◆ 아래 알고리즘은 **중간 합**을 보관함으로써
누적평균값들을 **선형시간**(linear time)에 구한다

```
Alg prefixAverages2( $X, n$ )           {ver.2}  
  input array  $X, A$  of  $n$  integers  
  output array  $A$  of prefix averages of  $X$   
  
1.  $sum \leftarrow 0$                         {1}  
2. for  $i \leftarrow 0$  to  $n - 1$              { $n$ }  
     $sum \leftarrow sum + X[i]$              { $n$ }  
     $A[i] \leftarrow sum / (i + 1)$          { $n$ }  
3. return                               {1}  
                                         {Total  $O(n)$ }
```