

관계 중심의 사고법

쉽게 배우는 알고리즘

2장. 알고리즘 설계와 분석의 기초

학습목표

- 알고리즘을 설계하고 분석하는 몇 가지 기초 개념을 이해한다
- 아주 기초적인 알고리즘의 수행 시간을 분석할 수 있도록 한다
- 점근적 표기법을 이해한다

(퀴즈) 수업내용 평가

알고리즘이란 무엇인가?

- 문제 해결 절차를 체계적으로 기술한 것
- 문제의 요구조건
 - 입력과 출력으로 명시할 수 있다
 - 알고리즘은 입력으로부터 출력을 만드는 과정을 기술

Step-by-step list of instructions for solving problem

입출력의 예

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

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 차이날 수 있다

1부터 n 까지 연속한 숫자의 합

두개의 방법

```
def sum_A(n):  
    sum = 0  
    for i in range(1, n+1):  
        sum += i  
    return sum
```

```
def sum_B(n):  
    sum = n * (n + 1) // 2  
    return sum
```

필요한 계산횟수가 입력크기 *n*과 무관함

필요한 계산횟수가 입력크기 *n*과 비례함

알고리즘 분석

- 입력 크기와 계산 횟수

- 10000 까지 합

sum (A) ran: 0.8322909750004328 milliseconds

sum (B) ran: 0.00029702899882977363 milliseconds

- 1000000 까지 합

sum (A) ran: 91.97638053299852 milliseconds

sum (B) ran: 0.00033955800063267816 milliseconds

- 계산 복잡도 표현을 위해 빅오 표기를
사용함(뒤에서 설명)

두개의 방법

```
// 방법 1
int n, res = 0;
for (int i = 1; i <= n; i++) {
    res += i;
}
System.out.println(res);
```

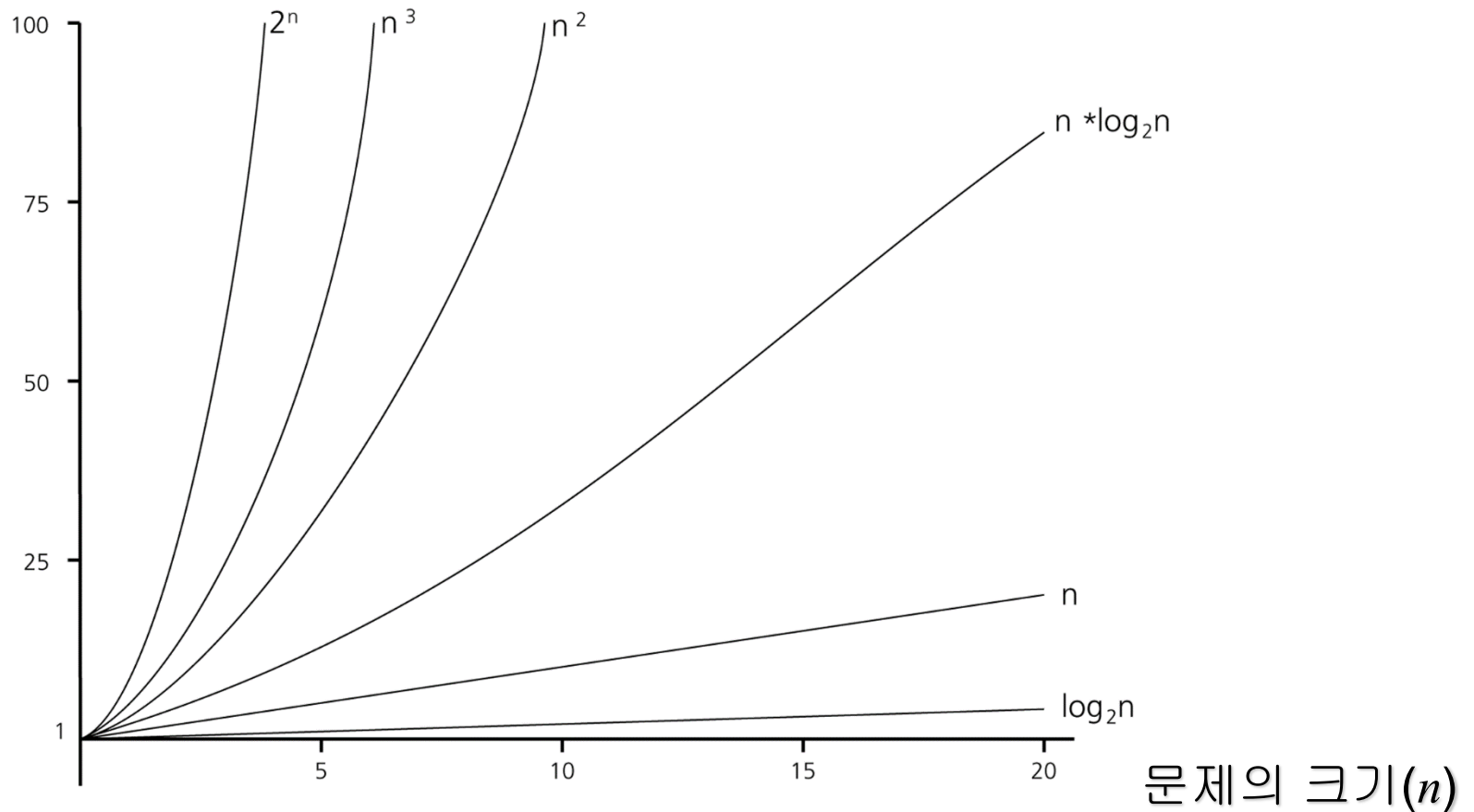
```
// 방법 2
int n, res = 0;
res = n*(n+1)/2;
System.out.println(res);
```

수행시간측정

- 수행시간 측정 방법의 문제점
 - 프로그램을 해야 함
 - 성능 측정이 어려운 경우,
블랙박스로 소스 공개가 안되는 경우 알고리즘
평가가 쉽지 않음
 - 두 개 알고리즘 비교 시 공정한 평가를 위해 동등한
조건을 만들어야 함(동일 컴퓨터, 동일 OS, 동일한
시스템 상황)

알고리즘의 수행 시간

수행 시간



알고리즘의 수행 시간

이진 로그

위키백과, 우리 모두의 백과사전.

수학에서 이진 로그 (binary logarithm)는 밑이 2인 로그이다. \log_2 또는 $\text{lb}^{[1]}$ 로 표기하며, 2의 거듭제곱의 역함수이다. 양의 실수 n 과 실수 x 에 대하여 $x = \log_2 n$ 은 $2^x = n$ 이라는 것과 같다.

$$x = \log_2 n \iff 2^x = n.$$

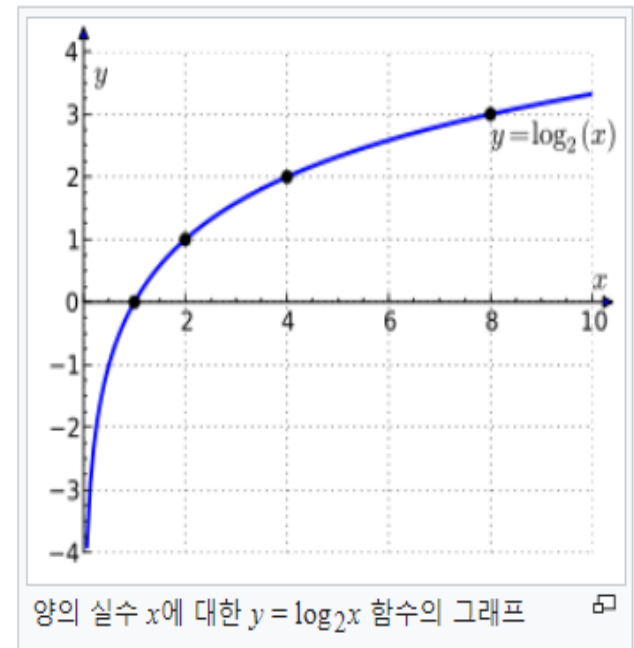
예를 들어 $\log_2 1 = 0$, $\log_2 2 = 1$, $\log_2 4 = 2$, $\log_2 32 = 5$ 이다.

각주 [편집]

- ↑ ISO 31-11과 ISO 80000-2

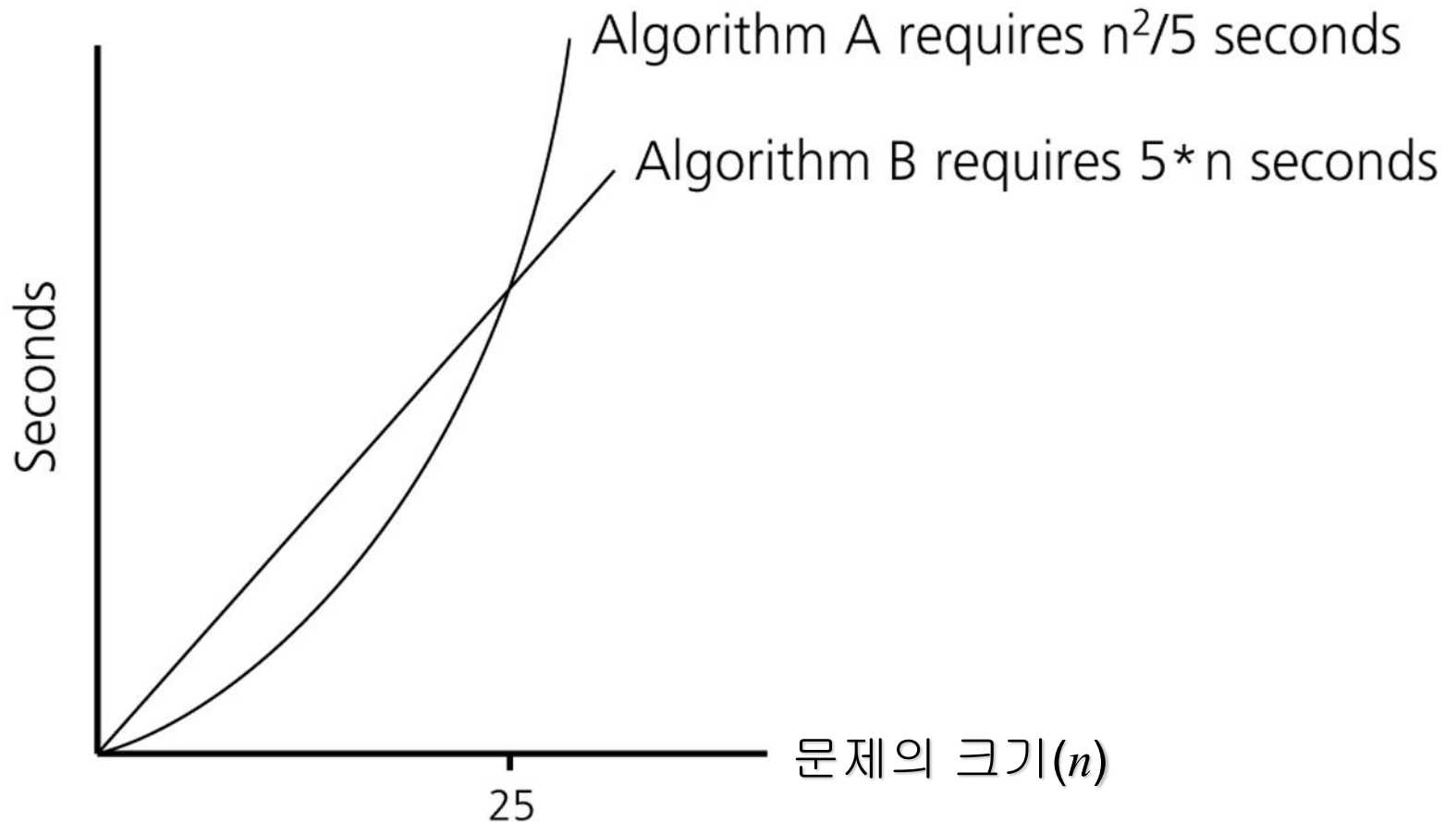
같이 보기 [편집]

- 로그
- 자연로그
- 상용로그



알고리즘의 수행 시간

수행 시간



알고리즘의 수행 시간

Function	n					
	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

알고리즘의 수행 시간

- 알고리즘의 수행 시간을 좌우하는 기준은 다양하게 잡을 수 있다
 - 예: for 루프의 반복횟수, 특정한 행이 수행되는 횟수, 함수의 호출횟수, ...
- 몇 가지 간단한 경우의 예를 통해 알고리즘의 수행 시간을 살펴본다

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 ;  
}
```

✓ n 에 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^2 에 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  $\lfloor n/2 \rfloor$ 개를 뽑을 때 이들 중 최댓값 ;
            sum ← sum + k ;
        }
    return sum ;
}
```

✓ n^3 에 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^2 에 비례하는 시간이 소요된다.

알고리즘의 수행 시간

```
factorial( $n$ )  
{  
    if ( $n=1$ ) return 1 ;  
    return  $n$ *factorial( $n-1$ ) ;  
}
```

✓ n 에 비례하는 시간이 소요된다.

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
 - $N! = N \times (N-1)!$
 - 예2: 수열의 점화식
 - $a_n = a_{n-1} + 2$

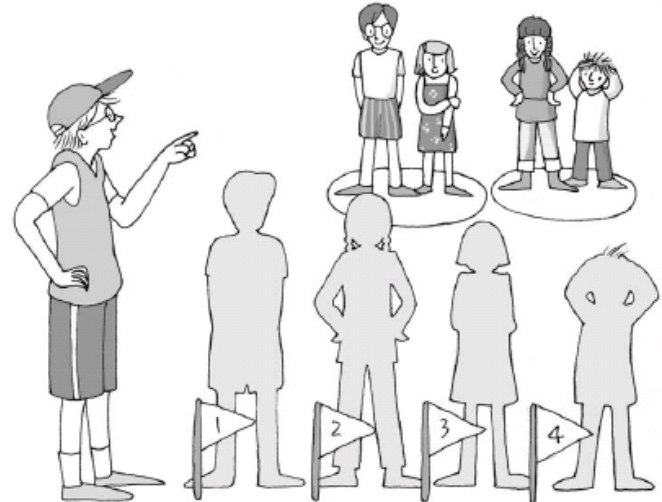


재귀와 귀납적 사고

- Divide-and-Conquer 설계 전략
 - 분할(Divide): 해결하기 쉽도록 문제를 여러 개의 작은 부분으로 나눈다
 - 각 작은 부분의 문제는 원래의 문제와 같은 속성을 지님
 - 정복(Conquer): 나눈 작은 문제를 각각 해결한다
 - 작은 부분에서 정복이 어렵거나 불가능하면 더 작은 부분으로 나눈다
 - 즉, 분할 단계로 다시 이동한다 (재귀호출 사용 가능)
 - 통합(Combine): (필요하다면) 해결된 해답을 모은다
- 이러한 문제 해결 방법을 하향식(top-down) 접근방법이라고 함

재귀의 예: Merge sort

- 문제: n 개의 정수를 (오름차순으로) 정렬하시오.
- 입력: 정수 n , 크기가 n 인 배열 $A[1..n]$
- 출력: (오름차순으로) 정렬된 배열 $A[1..n]$
- 알고리즘 개요: 각 단계별로 두 그룹의 정렬된 데이터를 서로 합쳐서 정렬된 더 큰 그룹으로 만들어 나가는 정렬

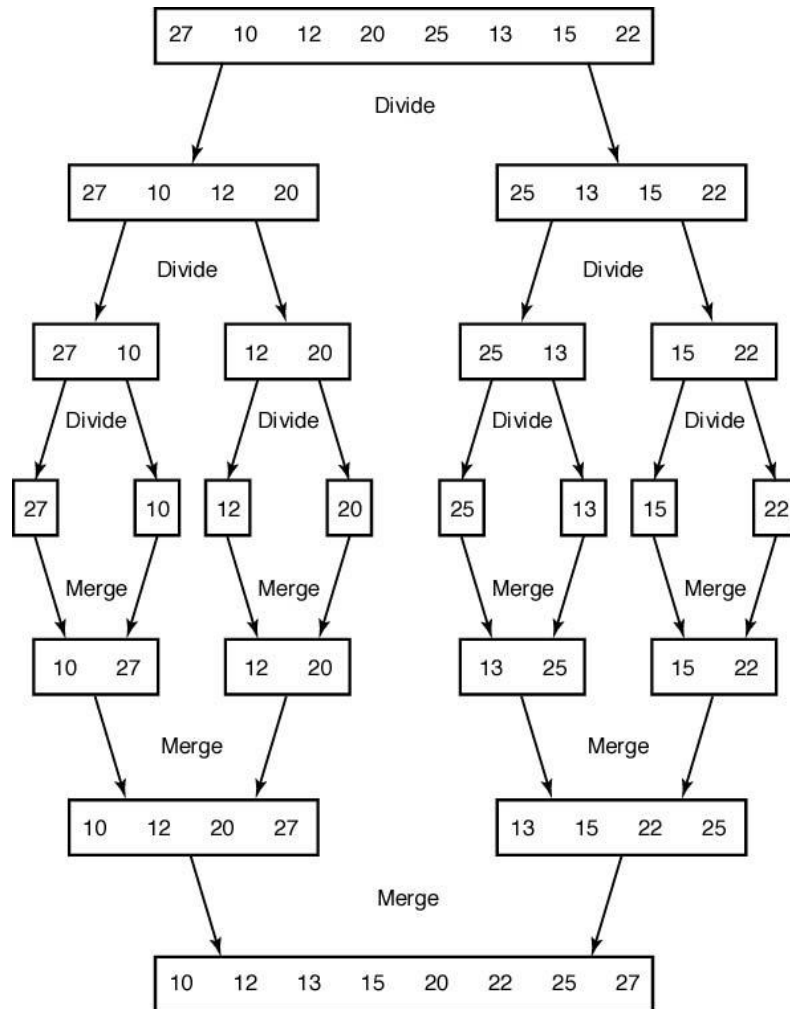


재귀의 예: 병합 정렬

mergeSort(A[], p, r) ▷ A[p ... r]을 정렬한다.

```
{  
  if (p < r) then {  
    q ← ⌊(p + r)/2⌋; ----- ①    ▷ p, q의 중간 지점 계산  
    mergeSort(A, p, q); ----- ②    ▷ 전반부 정렬  
    mergeSort(A, q+1, r); ----- ③    ▷ 후반부 정렬  
    merge(A, p, q, r); ----- ④    ▷ 병합  
  }  
}
```

```
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.  
}
```



mergeSort(A, 0, 7)

● 설계 전략

- [분할] 배열을 반으로 나누어서 2 개의 부분배열로 분할한다. 각 부분 배열의 크기가 1일 될 때까지 계속하여 분할한다.
- [정복] 가장 작은 수의 인접한 부분 배열 2개로 부터 정렬된 1개의 배열을 얻어낸다.
- [통합] 정렬된 부분 배열들을 합병하여 하나의 정렬된 배열로 만든다.

재귀의 예: 병합 정렬

mergeSort(A[], p, r) ▷ A[p ... r]을 정렬한다.

```
{  
  if (p < r) then {  
    q ← ⌊(p + r)/2⌋; ----- ①    ▷ p, r의 중간 지점 계산  
    mergeSort(A, p, q); ----- ②    ▷ 전반부 정렬  
    mergeSort(A, q+1, r); ----- ③    ▷ 후반부 정렬  
    merge(A, p, q, r); ----- ④    ▷ 병합  
  }  
}
```

✓ ②, ③은 재귀호출

✓ ①, ④는 재귀적 관계를 드러내기 위한 오버헤드

Mergesort의 수행시간

Statement	Effort
<code>MergeSort(A, left, right) {</code>	$T(n)$
<code>if (left < right) {</code>	$\Theta(1)$
<code>mid = floor((left + right) / 2);</code>	$\Theta(1)$
<code>MergeSort(A, left, mid);</code>	$T(n/2)$
<code>MergeSort(A, mid+1, right);</code>	$T(n/2)$
<code>Merge(A, left, mid, right);</code>	$\Theta(n)$
<code>}</code>	
<code>}</code>	

- So $T(n) = \Theta(1)$ when $n = 1$, and
 $2T(n/2) + \Theta(n)$ when $n > 1$
- 위의 수식은 재귀(*recurrence*) 표현임

(보충)Mergesort의 수행시간

- 성능평가
 - n 개 자료의 merge sort 계산 시간을 $T(n)$ 이라고 함
 - 정렬을 위해서 반으로 쪼개서 $n/2$ 개의 배열에 재귀함수를 호출
 - 한 배열 당 $T([n/2])$ 의 시간이 걸리므로 2개의 배열은 $T([n/2])+T([n/2])$ 의 시간이 걸림
 - 두 배열을 한번씩 비교하므로 merge의 시간은 n 임

다양한 알고리즘의 적용 주제들

- 카 네비게이션
- 스케줄링
 - TSP, 차량 라우팅, 작업공정, ...
- Human Genome Project
 - 매칭, 계통도, functional analyses, ...
- 검색
 - 데이터베이스, 웹페이지들, ...
- 자원의 배치
- 반도체 설계
 - Partitioning, placement, routing, ...
- ...

알고리즘을 왜 분석하는가

- 무결성 확인
- 자원 사용의 효율성 파악
 - 자원
 - 시간
 - 메모리, 통신대역, ...

알고리즘의 분석

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

알고리즘의 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 분석이라 한다

점근적 분석 Asymptotic Analysis

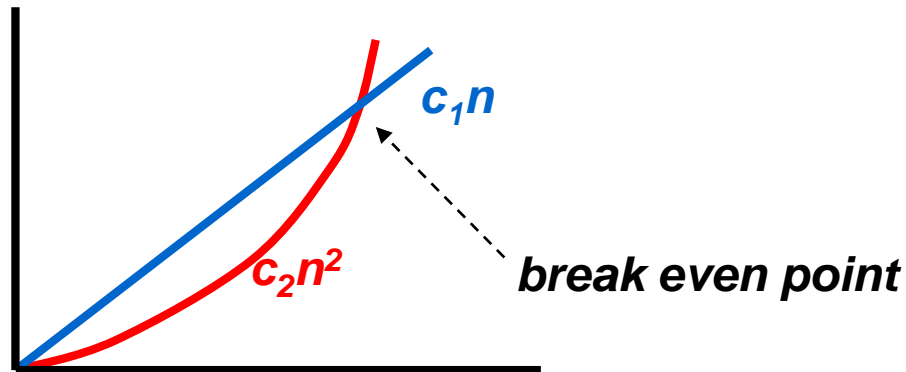
- 입력의 크기가 충분히 큰 경우에 대한 분석
- 이미 알고있는 점근적 개념의 예

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

- O , Ω , Θ , ω , o 표기법

알고리즘의 점근적 복잡도와 차수

- 차수(Order) vs. 상수(Constant)
 - c_1n 와 c_2n^2 비교(c_1 과 c_2 은 상수)



- 결과적으로...
 - 차수(Order) 중요
 - 상수 요소는 무시할 수 있음
 - 즉, N이 무한대로 갈 때를 기준으로 평가한다.
 - 입력 데이터가 최악일 때 알고리즘이 보이는 효율을 기준으로 한다.
- 낮은 차수를 제거
- 상수는 무시함

점근법 표기법 Asymptotic Notations

$O(g(n))$

- 기껏해야 $g(n)$ 의 비율로 증가하는 함수
- e.g., $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, ...

- Formal definition

- $O(g(n)) = \{ f(n) \mid \exists c > 0, n_0 \geq 0 \text{ s.t. } \forall n \geq n_0, cg(n) \geq f(n) \}$
- $f(n) \in O(g(n))$ 을 관행적으로 $f(n) = O(g(n))$ 이라고 쓴다.

- 직관적 의미

- $f(n) = O(g(n)) \Rightarrow f$ 는 g 보다 빠르게 증가하지 않는다
- 상수 비율의 차이는 무시

- 함수의 상한을 표시

점근적 표기법

- 예, $O(n^2)$
 - $3n^2 + 2n$
 - $7n^2 - 100n$
 - $n \log n + 5n$
 - $3n$
- 알 수 있는 한 최대한 tight 하게
 - $n \log n + 5n = O(n \log n)$ 인데 굳이 $O(n^2)$ 으로 쓸 필요없다
 - 엄밀하지 않은 만큼 정보의 손실이 일어난다

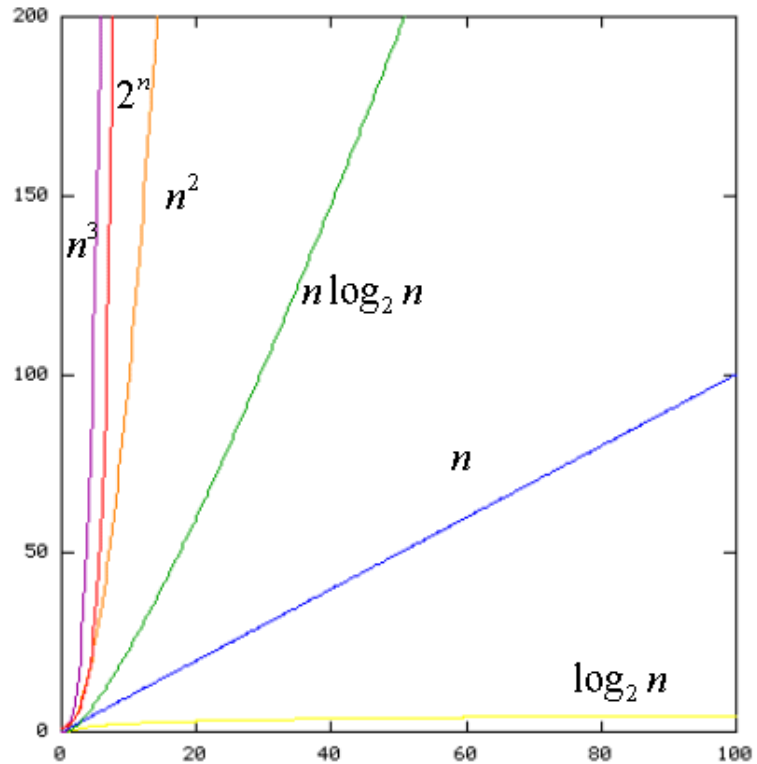
점근적 표기법

- 최고 차항의 최소 차수만을 사용하되, $\log n$ 은 버리지 말 것!

$$8n^2 \log n + 5n^2 + n = O(n^2 \log n)$$

점근적 표기법

- $O(1)$: 상수형
- $O(\log n)$: 로그형
- $O(n)$: 선형
- $O(n \log n)$: 선형로그형
- $O(n^2)$: 2차형
- $O(n^3)$: 3차형
- $O(n^k)$: k차형
- $O(2^n)$: 지수형
- $O(n!)$: 팩토리얼형



점근적 표기법

시간복잡도	n					
	1	2	4	8	16	32
1	1	1	1	1	1	1
$\log n$	0	1	2	3	4	5
n	1	2	4	8	16	32
$n \log n$	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	40320	20922789888000	26313×10^{33}

점근적 표기법

$\Omega(g(n))$

- 적어도 $g(n)$ 의 비율로 증가하는 함수
- $O(g(n))$ 과 대칭적

- Formal definition

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

- 직관적 의미

- $f(n) = \Omega(g(n)) \Rightarrow f$ 는 g 보다 느리게 증가하지 않는다

$n \geq 2$ 이면 $2n+1 \geq 2n$ 이므로 $2n+1 = \Omega(n)$

$cg(n) \leq f(n)$

점근적 표기법

$\Theta(g(n))$

– $g(n)$ 의 비율로 증가하는 함수

- Formal definition

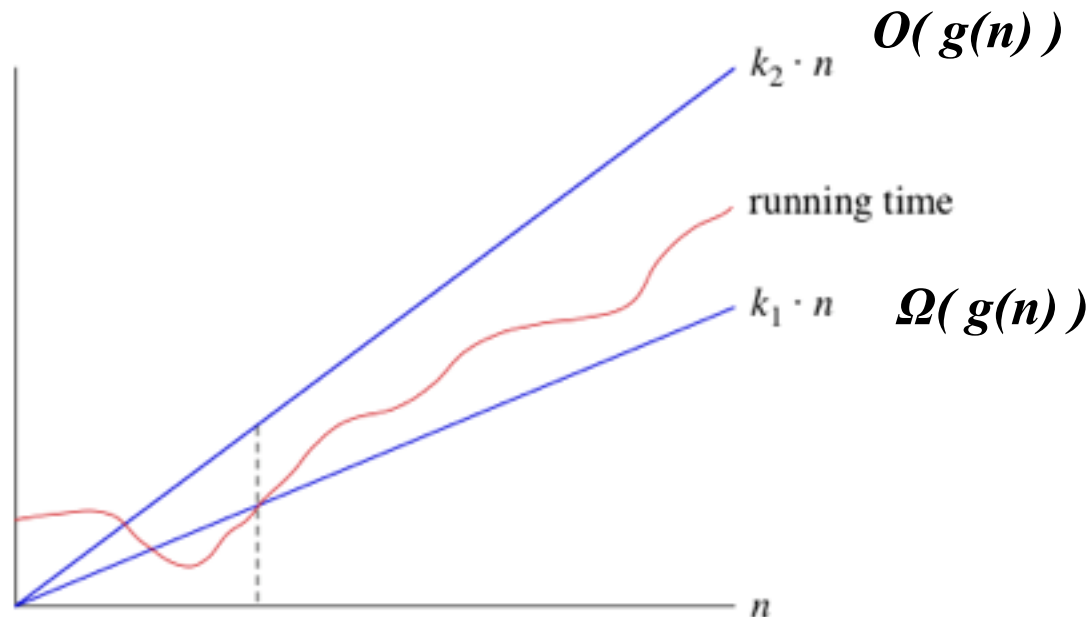
– $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

- 직관적 의미

– $f(n) = \Theta(g(n)) \Rightarrow f$ 는 g 와 같은 정도로 증가한다

점근적 표기법

- $\Theta(n)$ 은 상수 k_1 k_2 과 n 이 충분히 큰 값을 가질 때 알고리즘의 수행시간이 $k_1 \cdot n$ 보다 크고 $k_2 \cdot n$ 보다 작은 경우를 만족하는 경우로 정의할 수 있음



점근적 표기법

$f(n)=2n+3$ 에 대해 다음을 만족하는 k_1 과 k_2 가 존재한다면 $f(n) = \Theta(n)$ 임

$$k_1 * n < 2n+3 < k_2 * n$$

각 점근적 표기법의 직관적 의미

- $O(g(n))$
 - Tight or loose upper bound
- $\Omega(g(n))$
 - Tight or loose lower bound
- $\Theta(g(n))$
 - Tight bound



연습문제

- 두개의 입력에 대해 n^2 에 비례, 나머지 $n \log n$ 에 비례해서 시간 소요
 - 점근적 수행시간 $\Theta(n^2)$ 아님, 대략 $n \log n$ 에 비례 $\Theta(n \log n)$
 - 점근적 수행시간 $O(n^2)$ 임
 - 최악의 경우 $\Theta(n^2)$ 는 맞지 않음, 대략적으로 최악의 시간은 $\Theta(n \log n)$
 - 최악의 경우 $O(n^2)$ 은 맞음


점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현 예 (4장에서 공부함)
 - 선택정렬
 - $\Theta(n^2)$
 - 힙정렬
 - $O(n \log n)$
 - 퀵정렬
 - $O(n^2)$
 - 평균 $\Theta(n \log n)$

시간 복잡도 분석의 종류

- **Worst-case**
 - Analysis for the worst-case input(s)
- **Average-case**
 - Analysis for all inputs (전체에 대해 분석이 필요)
 - More difficult to analyze
- **Best-case**
 - Analysis for the best-case input(s)
 - 별로 유용하지 않음

저장/검색의 복잡도

- 
- 배열
 - $O(n)$
 - Binary search trees
 - 최악의 경우 $O(n)$
 - 평균 $\Theta(\log n)$
 - Balanced binary search trees
 - 최악의 경우 $\Theta(\log n)$
 - B-trees
 - 최악의 경우 $\Theta(\log n)$
 - Hash table
 - 평균 $\Theta(1)$

크기 n 인 배열에서 원소 찾기

- Sequential search
 - 배열이 아무렇게나 저장되어 있을 때
 - Worst case: $\Theta(n)$
 - Average case: $\Theta(n)$
- Binary search
 - 배열이 정렬되어 있을 때
 - Worst case: $\Theta(\log n)$
 - Average case: $\Theta(\log n)$

점근적 복잡도의 예

- 정렬 알고리즘들의 복잡도 표현 예 (3장에서 공부함)
 - 선택정렬
 - $\Theta(n^2)$
 - 힙정렬
 - $O(n \log n)$
 - 퀵정렬
 - $O(n^2)$
 - 평균 $\Theta(n \log n)$

과제

1. $\Theta(n)$ 시간복잡도를 갖는 검색알고리즘 (seqsearch)을 C/C++ (or Java, Python) 프로그램으로 작성하고, 자신만의 예제(**숫자 50개 이상**)를 만들어 그 실행 결과를 보이시오.
2. 피보나치 수열의 재귀 알고리즘(fib)과 반복 알고리즘(fib2)을 C/C++ (or Java, Python) 프로그램으로 작성하고, 다음 피보나치 수를 구하는데 걸리는 시간을 측정하여 그 결과를 제출하시오. (**시간이 너무 오래 걸리면 중간에 중단해도 됨**)
fib(10), fib(20), fib(50), fib(100)

 Due Date: 3/29(**금**)

 주의1: 숙제 copy는 절대 안됩니다. 저는 여러 분 양심을 믿습니다.

 주의2: **실행결과**는 **화면을 캡처** 한 후 프린트하여 제출해야 합니다.

순차검색 (Sequential Search)

알고리즘(의사코드)

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

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

 if-문: 모두 검사하였으나, x 를 찾지 못했나?

피보나찌(Fibonacci) 수열

알고리즘: 효율, 분석, 차수 - Part 1



피보나찌 수열의 정의

$$f_0 = 0$$

$$f_1 = 1$$

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

$$f_0 = 0$$

$$f_1 = 1$$

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



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



피보나치 [Leonardo Fibonacci, 1170?~1250?]

이탈리아의 수학자. 아라비아에서 발달한 수학을 섭렵하여 이를 정리·소개함으로써, 그리스도교 여러 나라의 수학을 부흥시킨 최초의 인물이 되었다. 1202년 저술한 《주판서(珠板書)》는 당시의 수학서의 결정판이다.

국적 이탈리아

활동분야 수학

출생지 이탈리아 피사

주요저서 《주판서(珠板書)》(1202) 《기하학의 실용》(1220)

피보나찌 수 구하기 – 재귀 알고리즘

 문제: n 번째 피보나찌 수를 구하라.

 입력: 양수 n

 출력: n 번째 피보나찌 수

 재귀(recursive) 알고리즘:

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

피보나찌 수 구하기 – 반복 알고리즘

문제: n 번째 피보나찌 수를 구하라.

입력: 양수 n

출력: n 번째 피보나찌 수

반복(iterative) 알고리즘:

```
int fib2 (int n)
{
    index i;
    int f[0..n];

    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];
}
```


- 참고자료

분할정복을 피해야 하는 경우

- 크기가 n 인 입력이 2개 이상의 조각으로 분할되며, 분할된 부분들의 크기가 거의 n 에 가깝게 되는 경우 \Rightarrow 시간복잡도: 지수(exponential) 시간
- 크기가 n 인 입력이 거의 n 개의 조각으로 분할되며, 분할된 부분의 크기가 n/c 인 경우. 여기서 c 는 상수이다. \Rightarrow 시간복잡도: $\Theta(n^{\lg n})$

수행시간측정(c언어로 작성)

- 컴퓨터에서 수행시간을 측정하는 방법 clock 함수가 사용
clock_t clock(void);

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void main( void )
{
    clock_t start, finish;
    double duration;
    start = clock();
    // 수행시간을 측정하고 하는 코드....
    // ....
    finish = clock();
    duration = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("%f 초입니다.\n", duration);
}
```