

Chap 1. Algorithms:

Efficiency, Analysis, and Order

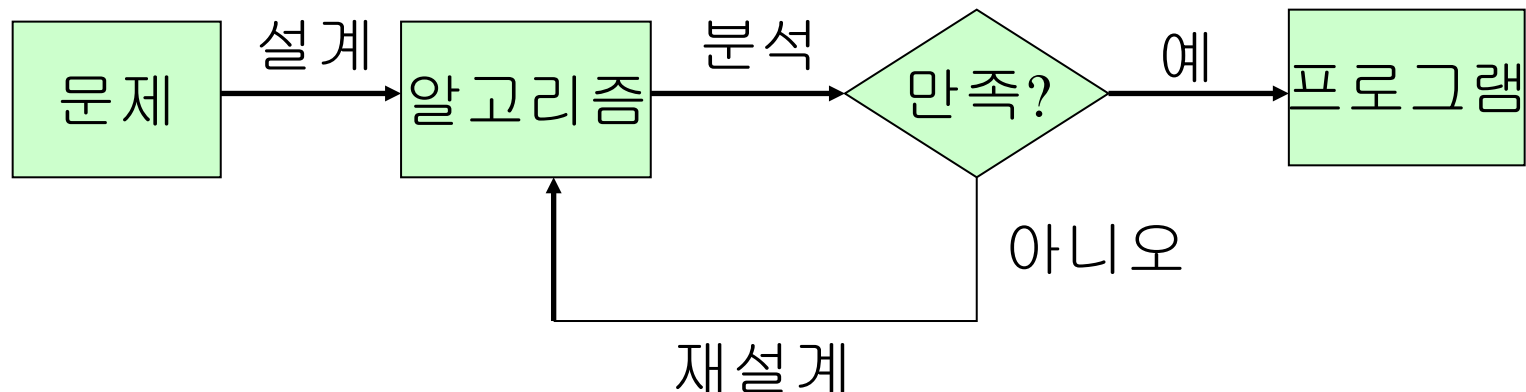
1. Algorithms
2. The Importance of Developing Efficient Algorithms
3. Analysis of Algorithms
4. Order

Overview

□ What is an algorithm ?

- Techniques for solving problems using a computer
 - Technique – not a programming style or a programming language, but the approach or methodology used to solve a problem
 - Applying a technique to a problem results in a step-by-step procedure
- Important characteristics - **efficiency**, **analysis**, and **order**

□ 프로그램 설계 과정



Overview

□ **Algorithm**

- An algorithm is a step-by-step procedure to solve a problem
- But making sure the developer is using the most efficient algorithm is very crucial no matter how fast computers become or how cheap memory gets
- How efficiently an algorithm solves a problem?
→ Need to analyze the algorithm

□ Analysis → Computational Complexity (Order)

- How efficiently an algorithm solves a problem
- Order helps group algorithms according to their eventual behavior
- Time/storage complexity
 - Time – how many times some operation(s) are performed
 - Storage – how much of memories are needed

Algorithms you'll learn in this course

Ch2. Divide-and-Conquer (분할정복법)

1. Binary Search
2. Mergesort
3. The Divide-and-Conquer Approach
4. Quicksort (Partition Exchange Sort)
5. Strassen's Matrix Multiplication Algorithm
6. When Not to Use Divide-and-Conquer

Ch3. Dynamic programming (동적계획법)

1. The Binomial Coefficient
2. Ch4.4 The Knapsack Problem
3. Chained Matrix Multiplication
4. Dynamic Programming & Optimization Problems
5. Floyd's Algorithm for Shortest Paths
6. Optimal Binary Search Trees
7. The Traveling Salesperson Problem

Algorithms you'll learn in this course

Ch4. Greedy Approach (탐욕적 방법)

1. Minimum Spanning Trees
2. Dijkstra's Algorithm for Single-Source Shortest Paths
3. Scheduling

Ch5. Backtracking (되추적)

1. The Backtracking Technique
2. The n-Queens Problems
3. Graph Coloring
4. The Hamiltonian Circuits Problem
5. The Sum-of-Subsets Problem
6. The 0-1 Knapsack Problem
7. Using a Monte Carlo Algorithm

Algorithms you'll learn in this course

Ch6. Branch-and-Bound (분기 한정법)

1. Illustrating Branch-and-Bound with the 0-1 Knapsack Problem
2. The Traveling Salesperson Problem

Ch7. The sorting problem

6. Heapsort
9. Sorting by Distribution (Radix Sort)

Algorithms you'll learn in this course

Ch8. The searching problem

1. Lower Bounds for Searching by Comparisons of Keys
2. Interpolation Search
3. Hashing

Ch9. The theory of P/NP

1. The Three General Categories of Problems
2. The Theory of NP
3. Handling NP-Hard Problems

알고리즘의 예

- 문제: 전화번호부에서 “홍길동”의 전화번호 찾기
- 알고리즘
 - **Sequential Search (순차검색)**
 - 첫 쪽부터 ‘홍길동’이라는 이름이 나올 때까지 순서대로 찾는다.
 - **Modified Binary Search (수정된 이분검색)**
 - 전화번호부는 “가나다”순으로 되어있으므로 먼저 “ㅎ”이 있을 만한 곳으로 넘겨본 후 앞뒤로 뒤적여가며 찾는다.
- 분석: 어떤 알고리즘이 더 좋은가?

알고리즘의 표기

- 자연어: 한글 또는 영어
 - Drawback 1 : difficult to write a complex problem
 - Drawback 2 : difficult to understand the algorithm
 - Drawback 3 : not clear how to create a computer language
- 프로그래밍언어: C, C++, Java 등
- 의사코드 (Pseudo-code)
 - 직접 실행할 수 있는 프로그래밍언어는 아니지만,
거의 실제 프로그램에 가깝게 계산과정을 표현할 수 있는 언어
- 알고리즘은 보통 의사코드로 표현한다.
- 이 강의/교재에서는 C++에 가까운 의사코드를 사용한다.

의사코드 유의사항(1)

- 배열 인덱스의 범위에 제한 없음
 - C++는 반드시 0부터 시작
 - 의사코드는 임의의 값 사용 가능
- 프로시저의 파라미터에 2차원 배열 크기의 가변성 허용
 - 예: `void pname(A[][]) { ... }`
- 지역배열에 변수 인덱스 허용
 - 예: `keytype S[low..high];`

의사코드 유의사항 (2)

□ 수학적 표현식 허용

- $\text{low} \leq x \ \&\& \ x \leq \text{high} \Rightarrow \text{low} \leq x \leq \text{high}$
- $\text{temp} = x; x = y; y = \text{temp} \Rightarrow \text{exchange } x \text{ and } y$

□ C++에 없는 타입 사용 가능

- **index**: 첨자로 사용되는 정수 변수
- **number**: 정수(int) 또는 실수(float) 모두 사용 가능

의사코드 유의사항(3)

- 제어 구조
 - repeat (n times) { ... }
- 프로시저와 함수
 - 프로시저: void pname(...) { ... }
 - 함수: returntype fname (...) { ... return x; }
- 참조 파라미터(reference parameter)를 사용하여 프로시저의 결과값 전달
 - 배열: 참조 파라미터로 전달
 - 기타: 데이터타입 이름 뒤에 &를 붙임
 - **const** 배열: 전달되는 배열의 값이 불변

Basic Structure

- Definition part
 - Problem, Input, Output에 관한 설명 기술:
- Procedure part

```
ReturnType procedureName(parameters) {  
  
    Body;  
  
}
```

- See Algorithm 1.1, 1.2, 1.3, 1.4

Sequential Search (순차검색)

- 문제: 크기가 n 인 배열 S 에 x 가 있는가?
 - 입력 (파라미터): (1) 양수 n , (2) 배열 $S[1..n]$, (3) 키 x
 - 출력: x 가 S 의 어디에 있는지의 위치. 만약 없으면 0.

- 알고리즘 (자연어):
 - x 와 같은 아이템을 찾을 때까지 S 에 있는 모든 아이템을 차례로 검사한다.
 - 만일 x 와 같은 아이템을 찾으면 S 안의 위치 값을 전달하고 S 를 모두 검사하고도 찾지 못하면 0을 전달한다.

Sequential Search Algorithm

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

- ✓ while-루프: 아직 검사할 항목이 있고, x 를 찾지 못했나?
- ✓ if-문: 모두 검사하였으나, x 를 찾지 못했나?

분석

- 순차검색 알고리즘으로 키를 찾기 위해서 S 에 있는 항목을 몇 개나 검색해야 하는가?
 - 키와 같은 항목의 위치에 따라 다름
 - 최악의 경우: n
- 좀 더 빨리 찾을 수는 없는가?
 - S 에 있는 항목에 대한 정보가 없는 한 더 빨리 찾을 수 없다.

Binary Search (이분검색)

- 문제: 크기가 n 인 정렬된 배열 S 에 x 가 있는가?
 - 입력: (1) 양수 n , (2) 배열 $S[1..n]$, (3) 키 x
 - 출력: x 가 S 의 어디에 있는지의 위치. 만약 없으면 0.

Binary Search Algorithm

```
void binsearch(int n,                // 입력(1)
               const keytype S[],    //      (2)
               keytype x,            //      (3)
               index& location) {    // 출력
    index low, high, mid;
    low = 1; high = n;
    location = 0;

    while (low <= high && location == 0) {
        mid = (low + high) / 2;      // 정수나눗셈
        if (x == S[mid]) location = mid;
        else if (x < S[mid]) high = mid - 1;
        else low = mid + 1;
    }
}
```

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

분석

- 이분검색 알고리즘으로 키를 찾기 위해서 S 에 있는 항목을 몇 개나 검색해야 하는가?
 - while 문을 수행할 때마다 검색 대상의 총 크기가 반씩 감소하기 때문에 최악의 경우라도 $\lg n + 1$ 개만 검사하면 된다. ($\lg = \log_2$)

순차검색 vs 이분검색

* 최악의 경우

배열의 크기	순차검색	이분검색
n	n	$\lg n + 1$
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

Applied problem

: 은행에서 고객에 관한 정보를 효과적으로 저장하고 검색하는 방법은?

Fibonacci Sequence

Definition of Fibonacci Sequence

$$f_0 = 0$$

$$f_1 = 1$$

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

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

피보나찌 수 구하기 알고리즘 (Recursive)

- 문제: Find n -th term of the Fibonacci sequence
 - 입력: nonnegative integer n
 - 출력: n -th term of the Fibonacci sequence
- 알고리즘 :

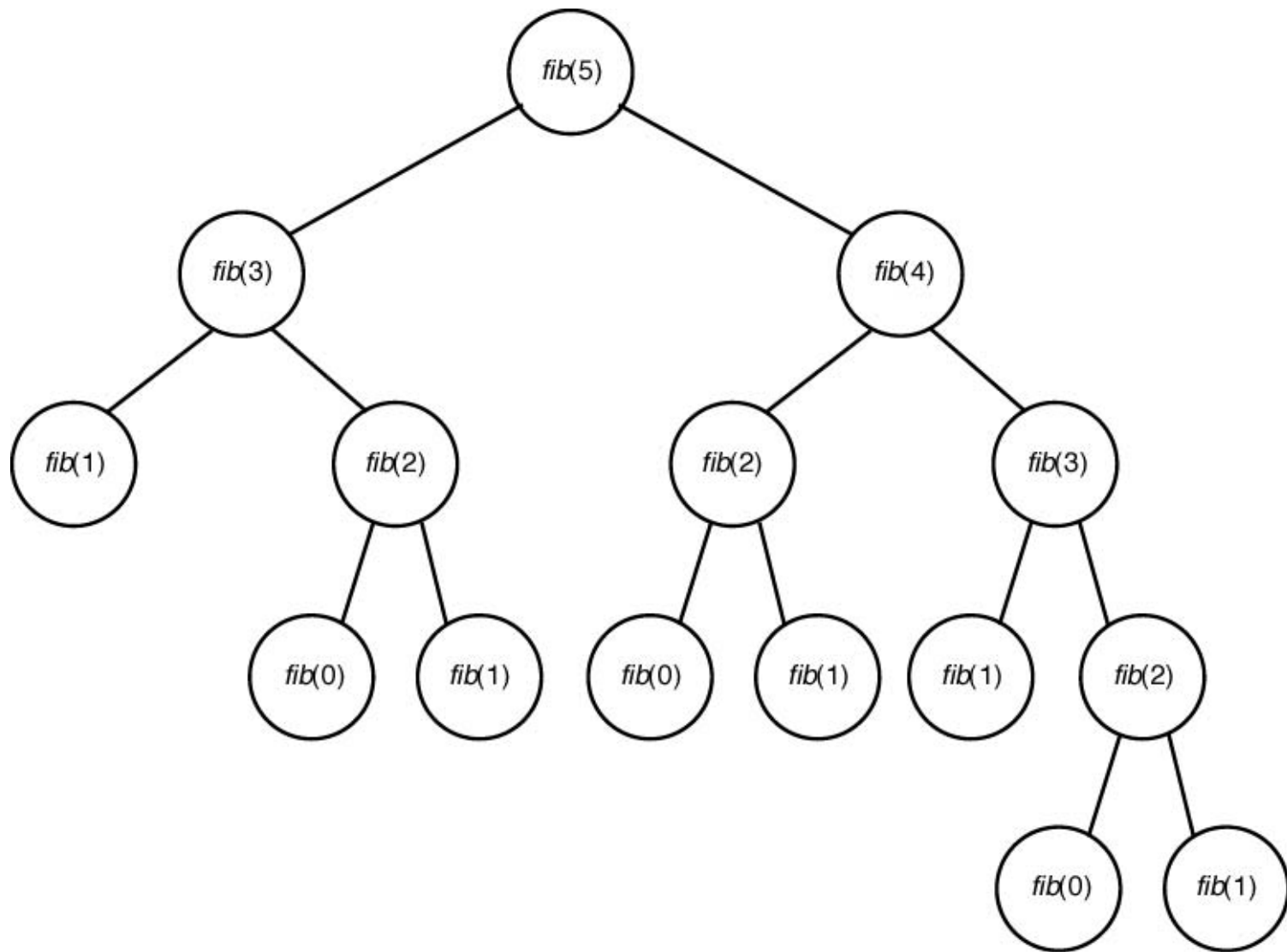
```
int fib (int n) {  
    if (n <= 1)  
        return n;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

- How does a computer to execute a recursive program?

분석

- 피보나찌 수 구하기 재귀 알고리즘
 - Easy to create and understandable, but extremely inefficient
 - 이유: values are computed over and over again
 - 예: fib(2) is computed 3 times to compute fib(5)
 - $\text{fib}(5) = \text{fib}(4) + \text{fib}(3)$
 - $\text{fib}(4) = \text{fib}(3) + \text{fib}(2)$; $\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$
 - $\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$

fib(5)의 재귀 트리



fib(n)의 함수 호출 횟수 계산

$T(n) = \text{fib}(n)$ 을 계산하기 위하여 *fib* 함수를 호출하는 횟수
즉, 재귀 트리 상의 마디(node)의 개수

$$T(0) = 1$$

$$T(1) = 1$$

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

for $n \geq 2$

$$> 2 \times T(n - 2)$$

왜냐하면 $T(n - 1) > T(n - 2)$

$$> 2^2 \times T(n - 4)$$

$$> 2^3 \times T(n - 6)$$

...

$$> 2^{n/2} \times T(0)$$

$$= 2^{n/2}$$

계산한 호출 횟수의 검증

정리: 재귀적 알고리즘으로 구성한 재귀 트리의 마디의 수를 $T(n)$ 이라고 하면, $n \geq 2$ 인 모든 n 에 대하여 $T(n) > 2^{n/2}$ 이다.

증명: (n 에 대한 수학적 귀납법으로 증명)

귀납출발점: $T(2) = T(1) + T(0) + 1 = 3 > 2 = 2^{2/2}$

$T(3) = T(2) + T(1) + 1 = 5 > 2.83 \approx 2^{3/2}$

귀납가정: $2 \leq m < n$ 인 모든 m 에 대해서 $T(m) > 2^{m/2}$ 이라 가정

귀납절차: $T(n) > 2^{n/2}$ 임을 보이면 된다.

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

$$> 2^{(n-1)/2} + 2^{(n-2)/2} + 1$$

[귀납가정에 의하여]

$$> 2^{(n-2)/2} + 2^{(n-2)/2}$$

$$= 2 \times 2^{(n/2)-1}$$

$$= 2^{n/2}$$

피보나찌 수 구하기 알고리즘 (Iterative)

```
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];  
}
```

분석

- 반복 알고리즘은 수행속도가 훨씬 더 빠르다.
 - 이유: 중복 계산이 없음
- 계산하는 항의 총 개수
 - $T(n) = n + 1$
 - 즉, $f[0]$ 부터 $f[n]$ 까지 한번씩만 계산

두 피보나찌 알고리즘의 비교

n	$n+1$	$2^{n/2}$	Iterative	Recursive (Lower bound)
40	41	1,048,576	41 <i>ns</i>	1048 μs
60	61	1.1×10^9	61 <i>ns</i>	1 <i>sec</i>
80	81	1.1×10^{12}	81 <i>ns</i>	18 <i>min</i>
100	101	1.1×10^{15}	101 <i>ns</i>	13 <i>days</i>
120	121	1.2×10^{18}	121 <i>ns</i>	36 <i>years</i>
160	161	1.2×10^{24}	161 <i>ns</i>	3.8×10^7 <i>years</i>
200	201	1.3×10^{30}	201 <i>ns</i>	4×10^{13} <i>years</i>

1 ns = 10^{-9} second; 1 μs = 10^{-6} second; 1 term can be computed in 1 nsec.

Analysis of Algorithms

- 시간복잡도(Time Complexity) 분석
 - Analyze the algorithm's efficiency by determining the number of times some basic operation is done as a function of the size of the input

- 표현 척도
 - Input size (입력크기)
 - 배열의 크기, 리스트의 길이, 행렬에서 행과 열의 크기, 트리에서 마디와 이음선의 수, 그래프에서는 정점과 간선의 수

 - Basic operation (단위연산)
 - 비교 (comparison), 지정 (assignment)

분석 방법의 종류

- Every-case time complexity analysis (모든 경우 분석)
 - $T(n)$ – the number of times the algorithm does the basic operation for an instance of size n
 - 입력크기에만 종속
 - 입력과는 무관하게 수행횟수 항상 일정
 - (Ex) Add array members, Exchange sort, Matrix multiplication

- Worst-case time complexity analysis (최악의 경우 분석)
 - $W(n)$ – the maximum number of times the algorithm will ever do its basic operation for an input size of n
 - 입력크기와 입력 값 모두에 종속
 - 단위연산이 수행되는 횟수가 최대인 경우
 - (Ex) Sequential search

분석 방법의 종류

□ Average-case time complexity analysis (평균의 경우 분석)

- $A(n)$ - the average number of times the algorithm does the basic operation for an input size of n
- 입력크기와 입력 값 모두에 종속
- 모든 입력에 대해서 단위연산이 수행되는 기대치(평균)
- 각 입력에 대해서 확률 할당 가능
- 일반적으로 최악의 경우보다 계산이 복잡
- (Ex) Sequential search

□ Best-case time complexity analysis (최선의 경우 분석)

- $B(n)$ – the minimum number of times the algorithm will ever do its basic operation for an input size of n
- 입력크기와 입력 값 모두에 종속
- 단위연산이 수행되는 횟수가 최소인 경우
- (Ex) Sequential search

알고리즘: 순차검색 시간복잡도 분석 (최악)

- 단위연산: 배열의 아이템과 키 x 와 비교연산 ($S[\text{location}] \neq x$)
- 입력크기: 배열 안에 있는 아이템의 수 n
- 최악의 경우 분석:
 - x 가 배열의 마지막 아이템이거나,
 x 가 배열에 없는 경우,
단위연산이 n 번 수행된다.
 - 따라서, $W(n) = n$

¶ 순차검색 알고리즘의 경우, 입력(배열 S 와 키 x)에 따라서 검색하는 횟수가 달라지므로, Every-case 복잡도 분석은 불가능

알고리즘: 순차검색 시간복잡도 분석 (최선)

- 단위연산: 배열의 아이템과 키 x 와 비교 연산 ($S[\text{location}] \neq x$)
- 입력크기: 배열 안에 있는 아이템의 수 n
- 최선의 경우 분석:
 - x 가 $S[1]$ 일 때, 입력의 크기에 상관없이 단위연산이 1번만 수행된다.
 - 따라서, $B(n) = 1$

알고리즘: 순차검색 시간복잡도 분석 (평균)

- 단위연산: 배열의 아이템과 키 x 와 비교 연산 ($S[\text{location}] \neq x$)
- 입력크기: 배열 안에 있는 아이템의 수 n
- 평균의 경우 분석:
 - 배열의 아이템이 모두 다르다고 가정한다.
 - 경우 1: x 가 배열 S 안에 있는 경우만 고려
 - $1 \leq k \leq n$ 에 대해서 x 가 배열의 k 번째 있을 확률 = $1/n$
 - x 가 배열의 k 번째 있다면,

S 를 찾기 위해서 수행하는 단위연산의 횟수 = k

■ 따라서, $A(n) = \sum_{k=1}^n k \times \frac{1}{n} = \frac{1}{n} \times \sum_{k=1}^n k = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$

알고리즘: 순차검색 시간복잡도 분석 (평균)

- 경우2: x 가 배열 S 안에 없는 경우도 고려

- x 가 배열 S 안에 있을 확률을 p 라고 하면,

- x 가 배열의 k 번째 있을 확률 $= p/n$

- x 가 배열에 없을 확률 $= 1-p$

- 따라서,
$$\begin{aligned} A(n) &= \sum_{k=1}^n (k \times \frac{p}{n}) + n(1-p) \\ &= \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p) \\ &= n(1 - \frac{p}{2}) + \frac{p}{2} \end{aligned}$$

- $p = 1 \Rightarrow A(n) = (n+1)/2$

- $p = 1/2 \Rightarrow A(n) = 3n/4 + 1/4$

알고리즘: 배열 덧셈

- 문제: 크기가 n 인 배열 S 의 모든 수를 더하라
 - 입력: 양수 n , 배열 $S[1..n]$
 - 출력: 배열 S 에 있는 모든 수의 합
- 알고리즘:

```
number sum (int n, const number S[]) {  
    index i;  
    number result;  
  
    result = 0;  
    for (i = 1; i <= n; i++)  
        result = result + S[i];  
    return result;  
}
```

배열 덧셈 알고리즘의 시간복잡도 분석

- 단위연산: 덧셈
- 입력크기: 배열의 크기 n
- 모든 경우 분석:
 - 배열 내용에 상관없이 for-루프가 n 번 반복된다.
 - 각 루프마다 덧셈이 1회 수행된다.
 - 따라서 n 에 대해서 덧셈이 수행되는 총 횟수는 $T(n) = n$ 이다.

알고리즘: 교환정렬

- 문제: 비내림차순(오름차순)으로 n 개의 키를 정렬
 - 입력: 양수 n , 배열 $S[1..n]$
 - 출력: 비내림차순으로 정렬된 배열
- 알고리즘:

```
void exchangesort (int n, keytype S[]) {  
    index i, j;  
    for (i = 1; i <= n-1; i++)  
        for (j = i+1; j <= n; j++)  
            if (S[j] < S[i])  
                exchange S[i] and S[j];  
}
```

알고리즘: 교환정렬 시간복잡도 분석 I

- 단위연산: 조건문 ($S[j]$ 와 $S[i]$ 의 비교)
- 입력크기: 정렬할 항목의 수 n
- 모든 경우 분석:
 - j-루프가 수행될 때마다 조건문 1번씩 수행
 - 조건문의 총 수행횟수
 - $i = 1$: j-루프 $n-1$ 번 수행
 - $i = 2$: j-루프 $n-2$ 번 수행
 - $i = 3$: j-루프 $n-3$ 번 수행
 - $i = n-1$: j-루프 1 번 수행
 - 따라서

$$T(n) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2}$$

알고리즘: 교환정렬 시간복잡도 분석 II

- 단위연산: 교환하는 연산 (exchange $S[j]$ and $S[i]$)
- 입력크기: 정렬할 항목의 수 n
- 최악의 경우 분석:
 - 조건문의 결과에 따라서 교환 연산의 수행여부가 결정된다.
 - 최악의 경우 = 조건문이 항상 참(true)이 되는 경우
= 입력 배열이 꺼꾸로 정렬되어 있는 경우

$$T(n) = \frac{(n-1)n}{2}$$

Discussion

- ❑ 최악, 평균, 최선의 경우 분석 방법 중에서 어떤 분석이 가장 정확한가?
- ❑ 최악, 평균, 최선의 경우 분석 방법 중에서 어떤 분석을 사용할 것인가?

- ❑ The best : every-case time complexity
But, all algorithms don't have the case.
- ❑ Useful case
 - Average-case : 일반적인 경우
 - Worst-case : 단 한번의 사고가 중요한 경우

Analysis of Correctness

□ Analysis of Correctness

- 알고리즘이 의도한 대로 수행되는지를 증명하는 절차
- Cf) Efficiency Analysis

□ 정확한 알고리즘이란?

- 어떠한 입력에 대해서도 답을 출력하면서 멈추는 알고리즘

□ 정확하지 않은 알고리즘이란?

- 어떤 입력에 대해서 멈추지 않거나,
- 또는 틀린 답을 출력하면서 멈추는 알고리즘

복잡도의 표기법

- O
 - Big O, asymptotic upper bound

- Ω
 - Omega, asymptotic lower bound

- Θ
 - Theta, order, asymptotic tight bound ($O \cap \Omega$)

대표적인 복잡도 카테고리

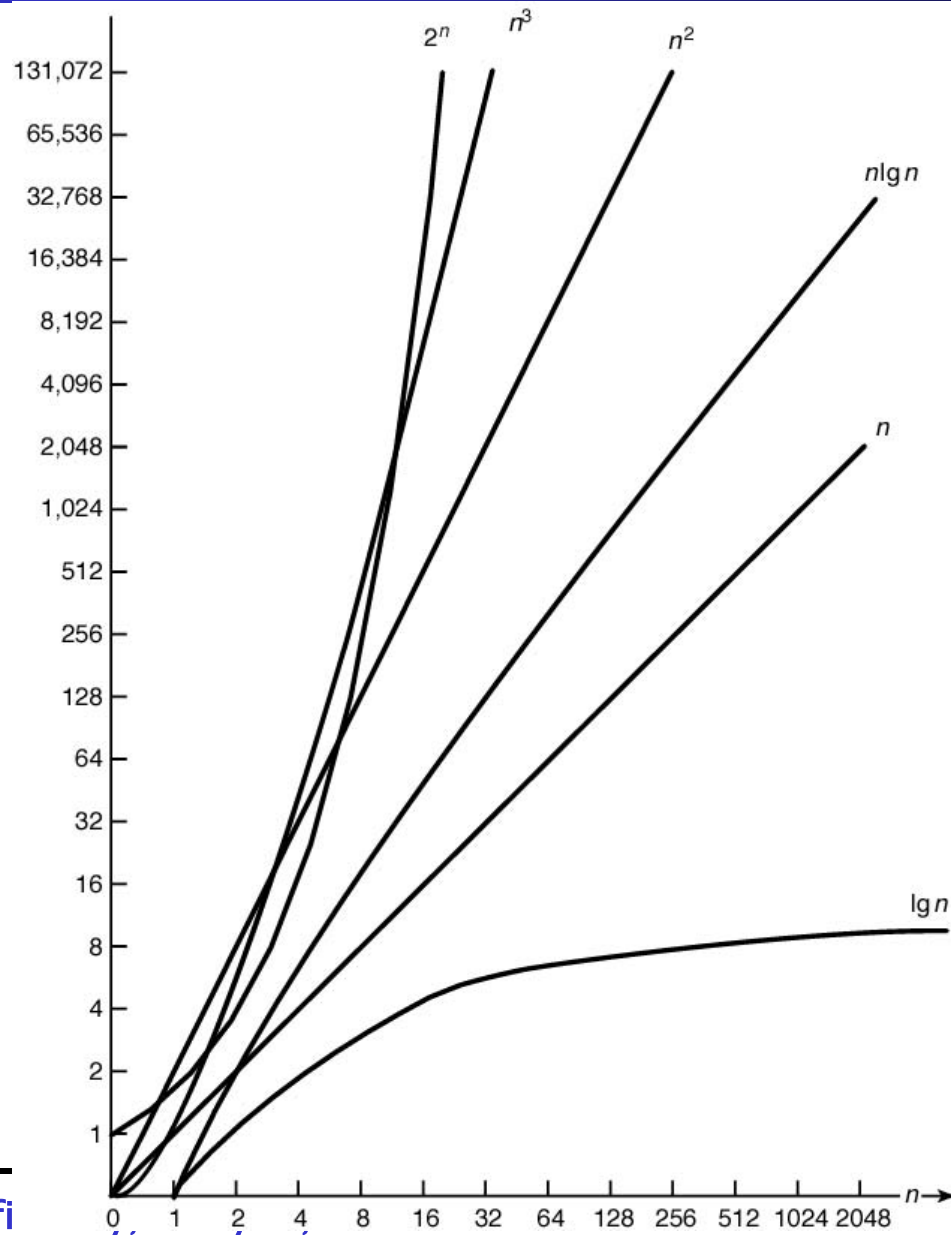
- $\Theta(\lg n)$
- $\Theta(n)$: 1차 (linear time algorithm)
- $\Theta(n \lg n)$
- $\Theta(n^2)$: 2차 (quadratic “ ”)
- $\Theta(n^3)$: 3차 (cubic “ ”)
- $\Theta(2^n)$: 지수 (exponential “ ”)
- $\Theta(n!)$

예제: 2차 항이 궁극적으로 지배한다

n	$0.1n^2$	$0.1n^2+n+100$
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1,000	100,000	101,100

$$g(n) = 5n^2 + 100n + 20 \in \theta(n^2) \equiv \text{order of } n^2$$

복잡도 함수의 증가율



시간복잡도별 실행시간 비교

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs^*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms [†]
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{13} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 days	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 days		

* 1 $\mu s = 10^{-6}$ second

† 1 ms = 10^{-3} second

Big O 표기법

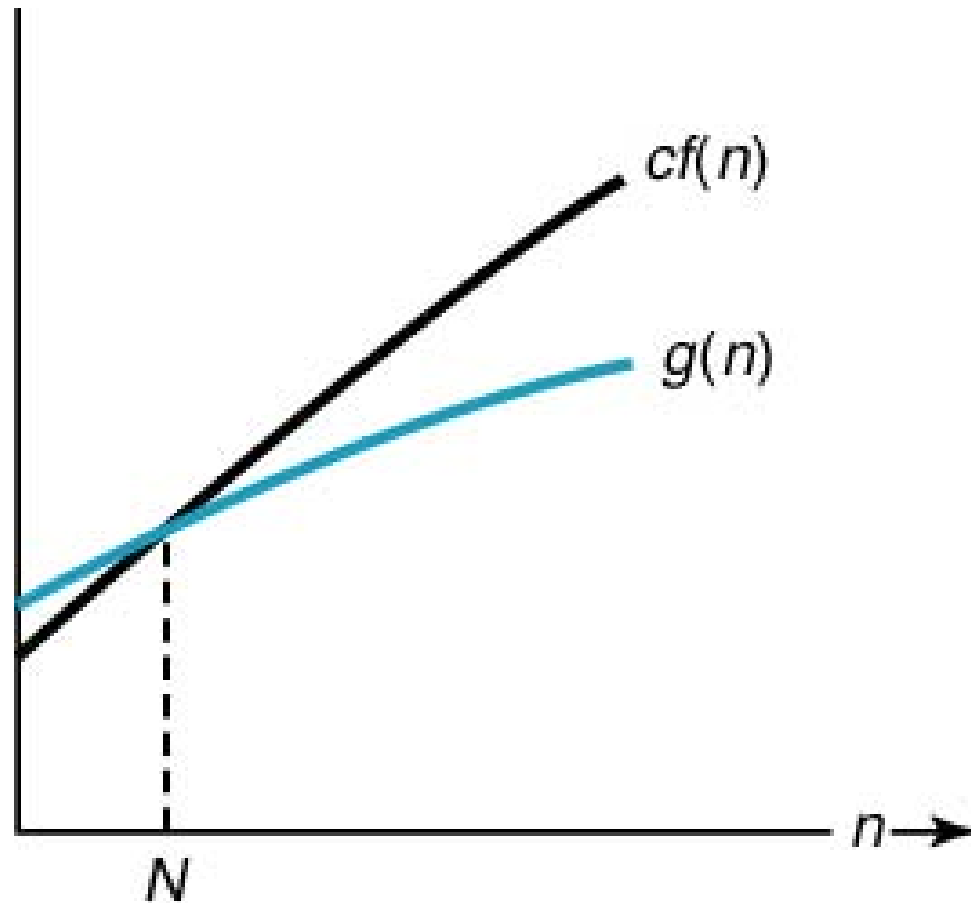
□ 정의 : 점근적 상한 (Asymptotic Upper Bound)

- 주어진 복잡도 함수 $f(n)$ 에 대해서 $g(n) \in O(f(n))$ 이면 다음을 만족한다 ($g(n)$: 분석된 결과).
- $n \geq N$ 인 모든 정수 n 에 대해서 $g(n) \leq c \times f(n)$ 이 성립하는 실수 $c > 0$ 와 음이 아닌 정수 N 이 존재한다.

□ $g(n) \in O(f(n))$ 읽는 방법:

- $g(n)$ 은 $f(n)$ 의 큰 오(big O)

Big O 표기법 (Cont)



(a) $g(n) \in O(f(n))$

Big O 표기법 (Cont)

- 어떤 함수 $g(n)$ 이 $O(n^2)$ 에 속한다는 말은
 - 그 함수($g(n)$)는 궁극에 가서는 (즉, 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차 함수 cn^2 보다는 작은 값을 가지게 된다는 것을 뜻한다. (그래프 상에서는 아래에 위치)
 - 즉 $g(n)$ 은 어떤 2차 함수 cn^2 보다는 궁극적으로 **좋다**고 (기울기가 낮다고) 말할 수 있다.
- 어떤 알고리즘의 시간복잡도가 $O(f(n))$ 이라면
 - 입력의 크기 n 에 대해서 이 알고리즘의 수행시간은 아무리 늦어도 $f(n)$ 은 된다 ($f(n)$ 이 상한이다).
 - 다시 말하면, 이 알고리즘의 수행시간은 $f(n)$ 보다 절대로 더 느릴 수는 없다는 말이다.

Big O 표기법 (예)

□ $n^2 + 10n \in O(n^2)$?

(1) $n \geq 10$ 인 모든 정수 n 에 대해서 $n^2 + 10n \leq 2n^2$ 이 성립한다.

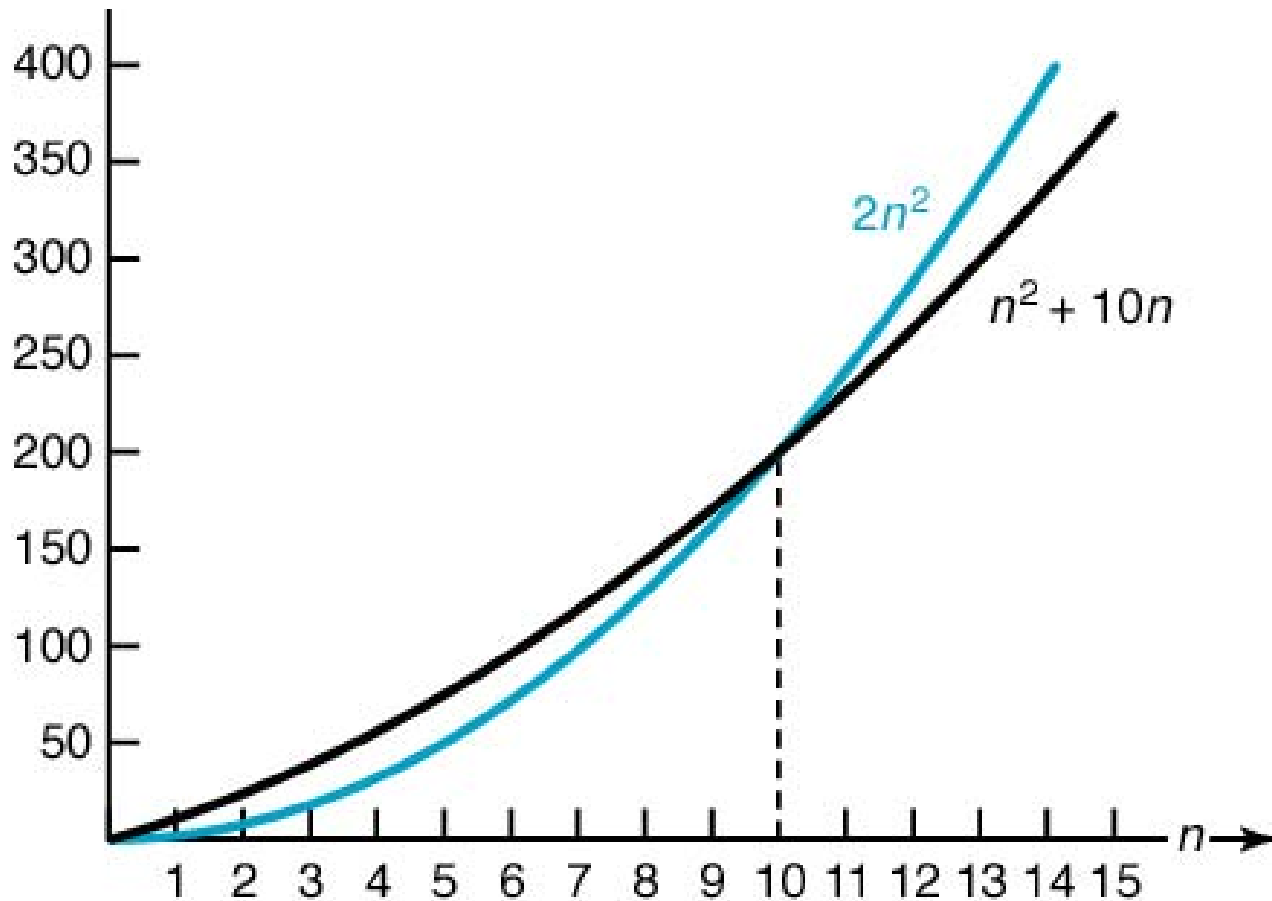
그러므로 $c = 2$ 와 $N = 10$ 을 선택하면, “Big O”의 정의에 의해서 $n^2 + 10n \in O(n^2)$ 이라고 결론지을 수 있다.

(2) $n \geq 1$ 인 모든 정수 n 에 대해서 $n^2 + 10n \leq n^2 + 10n^2 = 11n^2$ 이 성립한다.

그러므로 $c = 11$ 와 $N = 1$ 을 선택하면, “큰 O”의 정의에 의해서 $n^2 + 10n \in O(n^2)$ 이라고 결론지을 수 있다.

Big O 표기법 (예) (Cont)

$2n^2$ 과 $n^2 + 10n$ 의 비교



Big O 표기법 (예) (Cont)

□ $5n^2 \in O(n^2)$?

$c=5$ 와 $N=0$ 을 선택하면, $n \geq 0$ 인 모든 정수 n 에 대해서 $5n^2 \leq 5n^2$ 이 성립한다.

□ $T(n) = n(n-1) / 2$?

$n \geq 0$ 인 모든 정수 n 에 대해서 $n(n-1)/2 \leq n^2/2$ 이 성립한다. 그러므로 $c = 1/2$ 과 $N=0$ 을 선택하면, $T(n) \in O(n^2)$ 이라고 결론지을 수 있다.

□ $n^2 \in O(n^2+10n)$?

$n \geq 0$ 인 모든 정수 n 에 대해서, $n^2 \leq 1 \times (n^2+10n)$ 이 성립한다. 그러므로, $c=1$ 와 $N=0$ 을 선택하면, $n^2 \in O(n^2+10n)$ 이라고 결론지을 수 있다.

Big O 표기법 (예) (Cont)

□ $n \in O(n^2)$?

$n \geq 1$ 인 모든 정수 n 에 대해서, $n \leq 1 \times n^2$ 이 성립한다.

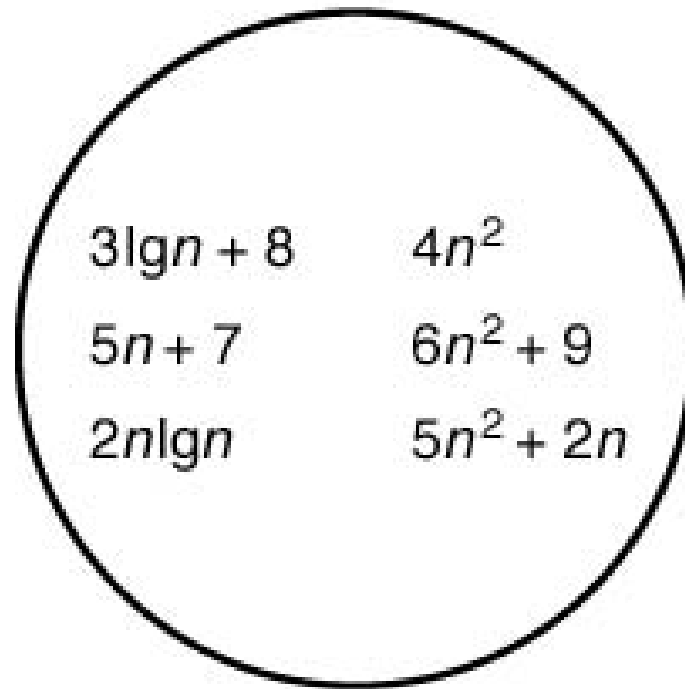
그러므로, $c=1$ 와 $N=1$ 을 선택하면, $n \in O(n^2)$ 이라고 결론지을 수 있다.

□ $n^3 \in O(n^2)$?

$n \geq N$ 인 모든 n 에 대해서 $n^3 \leq c \times n^2$ 이 성립하는 c 와 N 값은 존재하지 않는다. 즉, 양변을 n^2 으로 나누면, $n \leq c$ 가 되는데, c 를 아무리 크게 잡더라도 그 보다 더 큰 n 이 존재한다.
(성립하지 않음)

Big O 표기법 (예) (Cont)

$O(n^2)$



(a) $O(n^2)$

O : asymptotic upper bound

cn^2 보다 적은 값을 가지는 모든 함수.

Ω 표기법

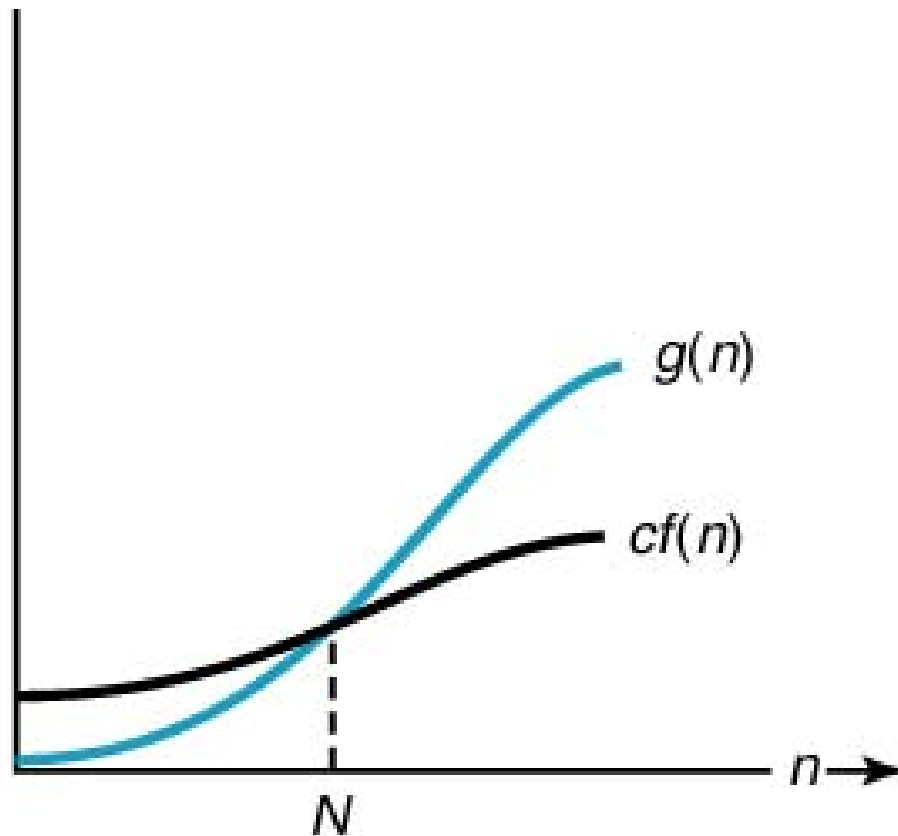
□ 정의 : 점근적 하한 (Asymptotic Lower Bound)

- 주어진 복잡도 함수 $f(n)$ 에 대해서 $g(n) \in \Omega(f(n))$ 이면 다음을 만족한다 ($g(n)$: 분석된 결과).
- $n \geq N$ 인 모든 정수 n 에 대해서 $g(n) \geq c \times f(n)$ 이 성립하는 실수 $c > 0$ 와 음이 아닌 정수 N 이 존재한다.

□ $g(n) \in \Omega(f(n))$ 읽는 방법:

- $g(n)$ 은 $f(n)$ 의 오메가(omega)

Ω 표기법



(b) $g(n) \in \Omega(f(n))$

Ω 표기법

- 어떤 함수 $g(n)$ 이 $\Omega(n^2)$ 에 속한다는 말은
 - 그 함수는 궁극에 가서는 (즉 어떤 임의의 N 값보다 큰 값에 대해서는) 어떤 2차 함수 cn^2 의 값보다는 큰 값을 가지게 된다는 것을 뜻한다 (그래프 상에서는 위에 위치).
 - 즉, 함수 $g(n)$ 은 어떤 2차 함수 cn^2 보다는 궁극적으로 나쁘다고 (기울기가 높다고) 말할 수 있다.
- 어떤 알고리즘의 시간복잡도가 $\Omega(f(n))$ 이라면,
 - 입력의 크기 n 에 대해서 이 알고리즘의 수행시간은 아무리 빨라도 $f(n)$ 밖에 되지 않는다 ($f(n)$ 이 하한이다).
 - 다시 말하면, 이 알고리즘의 수행시간은 $f(n)$ 보다 절대로 더 빠를 수는 없다는 말이다.

Ω 표기법 : 예

□ $n^2 + 10n \in \Omega(n^2)$?

$n \geq 0$ 인 모든 정수 n 에 대해서 $n^2 + 10n \geq n^2$ 이 성립한다.

그러므로 $c = 1$ 와 $N = 0$ 을 선택하면, $n^2 + 10n \in \Omega(n^2)$ 이라고 결론지을 수 있다.

□ $5n^2 \in \Omega(n^2)$?

$n \geq 0$ 인 모든 정수 n 에 대해서, $5n^2 \geq 1 \times n^2$ 이 성립한다.

그러므로, $c=1$ 와 $N=0$ 을 선택하면, $5n^2 \in \Omega(n^2)$ 이라고 결론지을 수 있다.

Ω 표기법 : 예 (계속)

□ $T(n) = n(n-1)/2$?

$n \geq 2$ 인 모든 n 에 대해서 $n-1 \geq n/2$ 이 성립한다. 그러므로,

$n \geq 2$ 인 모든 n 에 대해서 $n(n-1)/2 \geq n/2 \times n/2 = \frac{1}{4} n^2$ 이 성립한다.

따라서 $c = \frac{1}{4}$ 과 $N = 2$ 를 선택하면, $T(n) \in \Omega(n^2)$ 이라고
결론지을 수 있다.

□ $n^3 \in \Omega(n^2)$?

$n \geq 1$ 인 모든 정수 n 에 대해서, $n^3 \geq 1 \times n^2$ 이 성립한다.

그러므로, $c = 1$ 와 $N = 1$ 을 선택하면, $n^3 \in \Omega(n^2)$ 이라고
결론지을 수 있다.

Ω 표기법 : 예 (계속)

□ $n \in \Omega(n^2)$? 모순유도에 의한 증명 (Proof by contradiction)

$n \in \Omega(n^2)$ 이라고 가정.

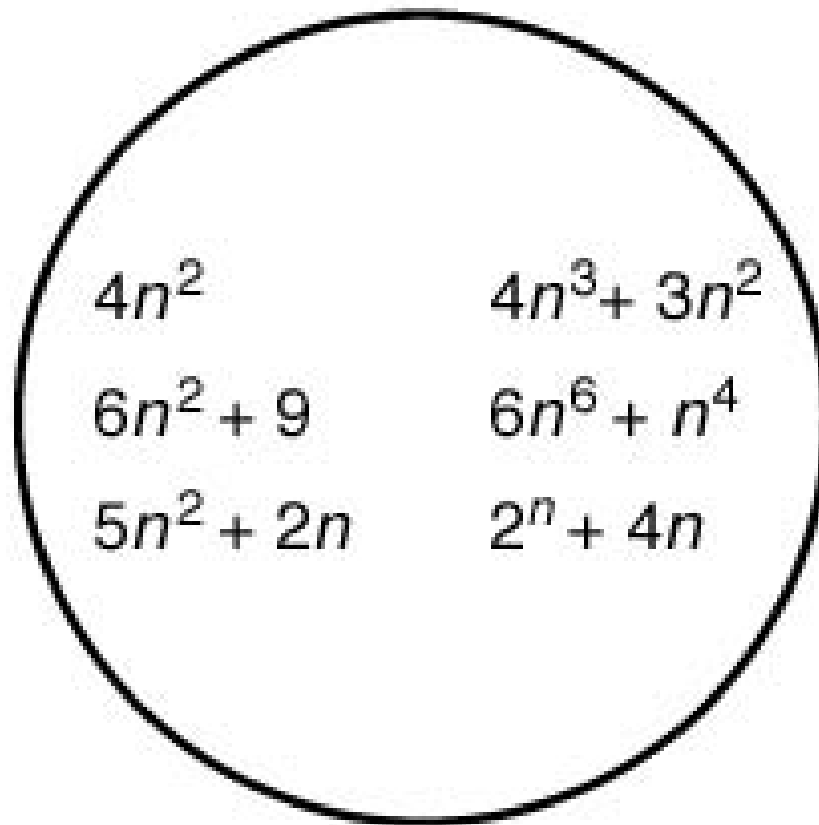
그러면 $n \geq N$ 인 모든 정수 n 에 대해서, $n \geq c \times n^2$ 이 성립하는
실수 $c > 0$, 그리고 음이 아닌 정수 N 이 존재한다.

위의 부등식의 양변을 cn 으로 나누면 $1/c \geq n$ 이 된다.

그러나 이 부등식은 절대로 성립할 수 없다.
따라서 위의 가정은 모순이다.

Ω 표기법 : 예 (계속)

$\Omega(n^2)$



(b) $\Omega(n^2)$

⊕ 표기법

□ 정의 : **Asymptotic Tight Bound**

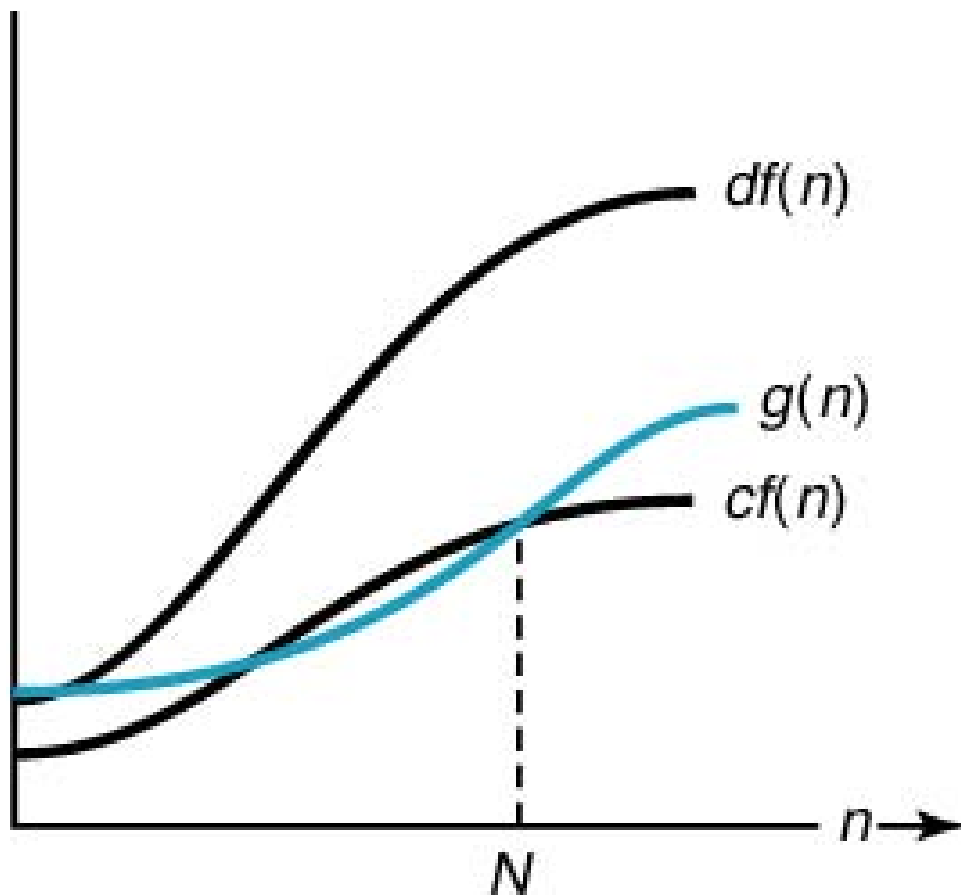
- 복잡도 함수 $f(n)$ 에 대해서 $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- 다시 말하면, $\Theta(f(n))$ 은 다음을 만족하는 복잡도 함수 $g(n)$ 의 집합이다.
- 즉, $n \geq N$ 인 모든 정수 n 에 대해서 $c \times f(n) \leq g(n) \leq d \times f(n)$ 이 성립하는 실수 $c > 0$ 와 $d > 0$, 그리고 음이 아닌 정수 N 이 존재한다.

□ 참고: $g(n) \in \Theta(f(n))$ 은 “ $g(n)$ 은 $f(n)$ 의 차수 (order)”라고 한다.

□ 예 : $T(n) = n(n-1)/2$ 은 $O(n^2)$ 이면서 $\Omega(n^2)$ 이다.

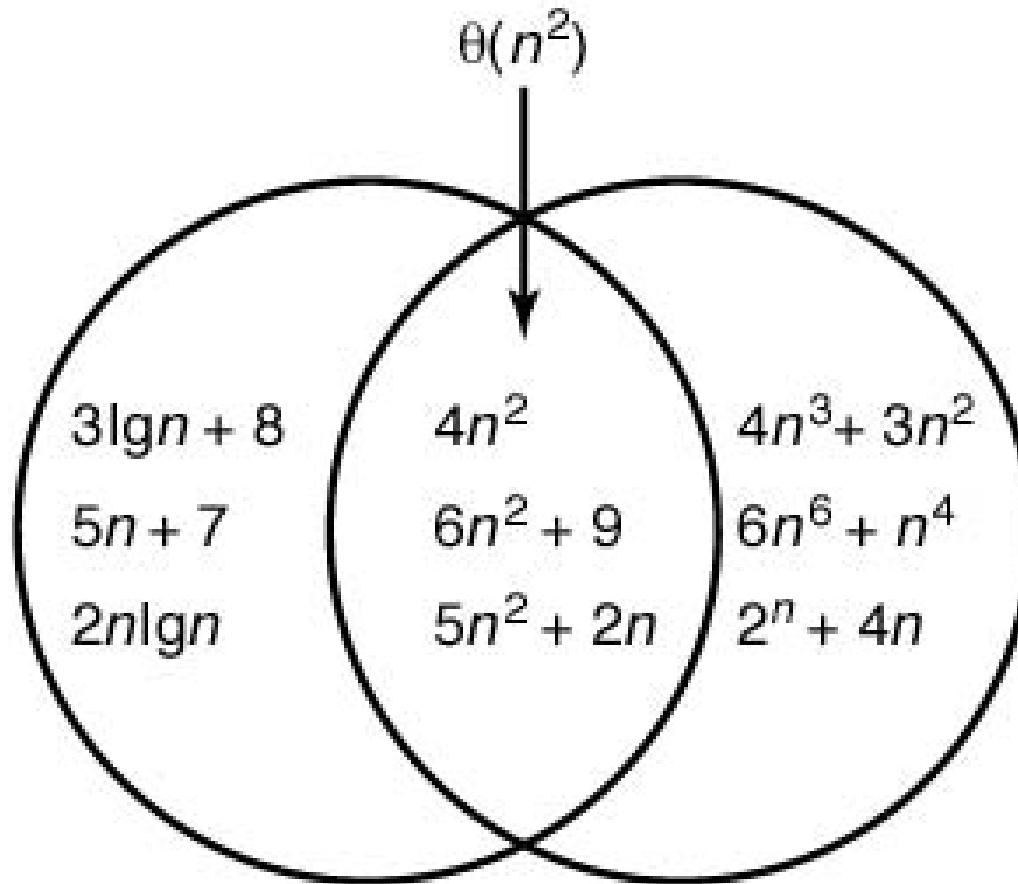
따라서 $T(n) = \Theta(n^2)$

⊕ 표기법



(c) $g(n) \in \theta(f(n))$

$\Theta(n^2)$



(c) $\theta(n^2) = O(n^2) \cap \Omega(n^2)$

작은(Small) o 표기법

- 작은 o 는 복잡도 함수 끼리의 관계를 나타내기 위한 표기법이다.
- 정의 : 작은 o
주어진 복잡도 함수 $f(n)$ 에 대해서 $o(f(n))$ 은 다음을 만족하는 모든 복잡도 함수 $g(n)$ 의 집합이다:
모든 실수 $c > 0$ 에 대해서 $g(n) \leq c \times f(n)$ (여기서 $n \geq N$ 인 모든 n 에 대해서) 이 성립하는 음이 아닌 정수 N 이 존재한다.
- 참고: $g(n) \in o(f(n))$ 은 “ $g(n)$ 은 $f(n)$ 의 작은 오(o)”라고 한다.

큰 O vs 작은 o

□ 큰 O 와의 차이점

- 큰 O - 실수 $c > 0$ 중에서 하나만 성립하여도 됨
- 작은 o - 모든 실수 $c > 0$ 에 대해서 성립하여야 함

□ $g(n) \in o(f(n))$ 은 쉽게 설명하자면

$g(n)$ 이 궁극적으로 $f(n)$ 보다 훨씬 낮다(좋다)는 의미이다.

작은 o 표기법 : 예

□ $n \in o(n^2)$?

증명:

$c > 0$ 이라고 하자. $n \geq N$ 인 모든 n 에 대해서 $n \leq c n^2$ 이 성립하는 N 을 찾아야 한다. 이 부등식의 양변을 $c n$ 으로 나누면 $1/c \leq n$ 을 얻는다. 따라서 $N \geq 1/c$ 가 되는 어떤 N 을 찾으면 된다.

여기서 N 의 값은 c 에 의해 좌우된다.

예를 들어 만약 $c=0.0001$ 이라고 하면, N 의 값은 최소한 10,000이 되어야 한다. 즉 $n \geq 10,000$ 인 모든 n 에 대해서 $n \leq 0.0001 n^2$ 이 성립한다.

작은 o 표기법 : 예 (계속)

□ $n \in o(5n)$?

모순 유도에 의한 증명: $c = 1/6$ 이라고 하자.

$n \in O(5n)$ 이라고 가정하면, $n \geq N$ 인 모든 정수 n 에 대해서,

$n \leq 1/6 \times 5n \leq 5/6 n$ 이 성립하는 음이 아닌 정수 N 이 존재해야 한다.

그러나 그런 N 은 절대로 있을 수 없다.

따라서 위의 가정은 모순이다.

차수의 주요 성질 I

1. $g(n) \in O(f(n))$ iff $f(n) \in \Omega(g(n))$
2. $g(n) \in \Theta(f(n))$ iff $f(n) \in \Theta(g(n))$
3. $b > 1$ 이고 $a > 1$ 이면, $\log_a n \in \Theta(\log_b n)$ 은 항상 성립.

다시 말하면 로그(logarithm) 복잡도 함수는 모두 같은 카테고리에 속한다. 따라서 통상 $\Theta(\lg n)$ 으로 표시한다.

4. $b > a > 0$ 이면, $a^n \in o(b^n)$. 다시 말하면, 지수(exponential) 복잡도 함수가 모두 같은 카테고리 안에 있는 것은 아니다.

차수의 주요 성질 II

5. $a > 0$ 인 모든 a 에 대해서, $a^n \in o(n!)$.

다시 말하면, $n!$ 은 어떤 지수 복잡도 함수보다도 나쁘다.

6. 복잡도 함수를 다음 순으로 나열해 보자.

$\Theta(\lg n)$, $\Theta(n)$, $\Theta(n \lg n)$, $\Theta(n^2)$, $\Theta(n^j)$, $\Theta(n^k)$, $\Theta(a^n)$, $\Theta(b^n)$,
 $\Theta(n!)$

여기서 $k > j > 2$ 이고 $b > a > 1$ 이다. 복잡도 함수 $g(n)$ 이 $f(n)$ 을 포함한 카테고리의 왼쪽에 위치한다고 하면, $g(n) \in o(f(n))$

7. $c \geq 0, d > 0$, $g(n) \in O(f(n))$, 그리고 $h(n) \in \Theta(f(n))$ 이면,

$$c \times g(n) + d \times h(n) \in \Theta(f(n))$$

$$\text{ex) } 5n + 3 \lg n + 10n \lg n + 7n^2 \in \Theta(n^2)$$

극한(limit)를 이용하여 차수를 구하는 방법

□ 정의 :

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \begin{cases} c > 0 \text{ 이면 } g(n) \in \Theta(f(n)) \\ 0 \text{ 이면 } g(n) \in o(f(n)) \\ \infty \text{ 이면 } f(n) \in o(g(n)) \end{cases}$$

□ 예 : 다음이 성립함을 보이시오.

$$\frac{n^2}{2} \in o(n^3) \quad \lim_{n \rightarrow \infty} \frac{n^2/2}{n^3} = \lim_{n \rightarrow \infty} \frac{1}{2n} = 0$$

- $b > a > 0$ 일 때, $a^n \in o(b^n)$

$$\lim_{n \rightarrow \infty} \frac{a^n}{b^n} = \lim_{n \rightarrow \infty} \left(\frac{a}{b} \right)^n = 0 \quad \text{왜냐하면, } 0 < \frac{a}{b} < 1$$

로피탈(L'Hopital)의 법칙

□ 정리 : 로피탈(L'Hopital)의 법칙

$$\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty \text{ 이면}$$

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \lim_{n \rightarrow \infty} \left(\frac{g'(n)}{f'(n)} \right) \text{ 이다.}$$

□ 예 : 다음이 성립함을 보이시오.

$$-\lg n \in o(n)$$

$$\lim_{n \rightarrow \infty} \frac{\lg n}{n} = \lim_{n \rightarrow \infty} \left(\frac{\frac{1}{n \ln 2}}{1} \right) = 0$$

$$-\log_a n \in \Theta(\log_b n)$$

$$\lim_{n \rightarrow \infty} \frac{\log_a n}{\log_b n} = \lim_{n \rightarrow \infty} \left(\frac{\frac{1}{n \ln a}}{\frac{1}{n \ln b}} \right) = \frac{\log b}{\log a} > 0$$