

1. 자료구조 기본 개념

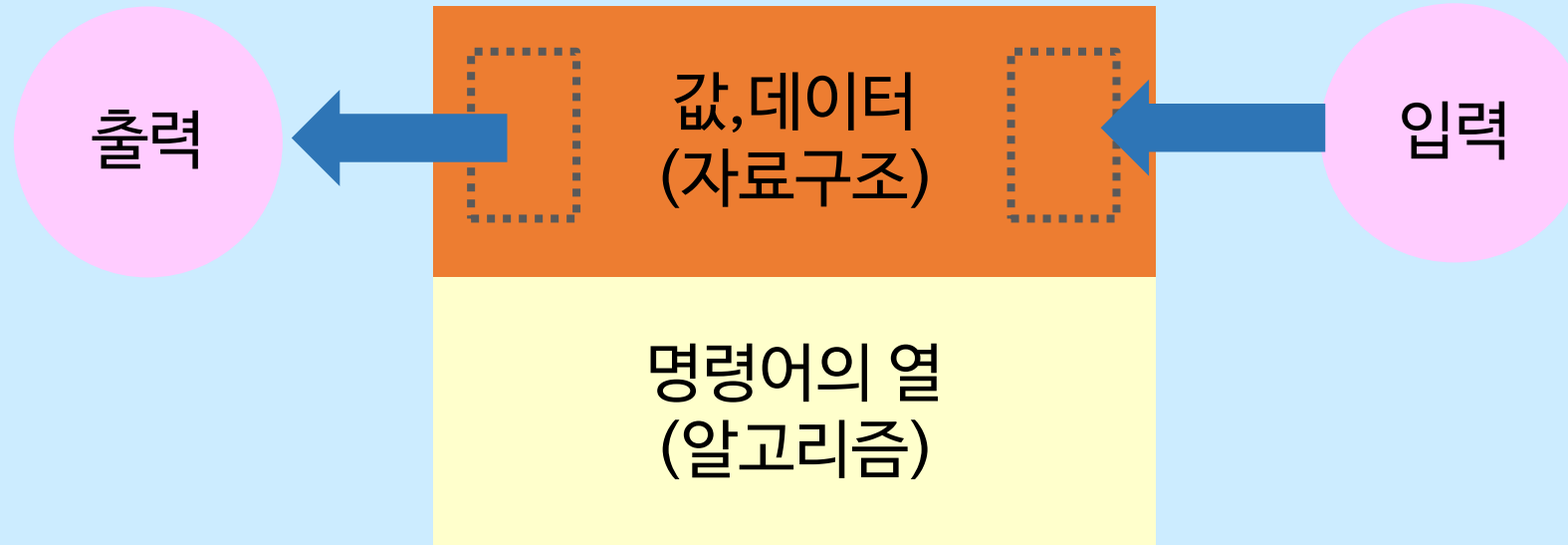
한국항공대학교 안준선

주요 내용

- 자료구조와 알고리즘의 개념
- 알고리즘의 분석 : 정확성과 성능
- 알고리즘의 복잡도
 - 점근적 복잡도
- 프로그램 수행 시간의 측정

컴퓨터 프로그램의 동작

- 입력으로부터 출력(결과)을 생성



- 기본 명령어의 종류
 - 지정문 : $A \leftarrow B$
 - 계산 : $A \leftarrow B \text{ op } C$ (산술, 비트, 비교, 논리 연산)
 - 제어 : jump I (분기, 반복)

컴퓨터 프로그램의 동작

■ 알고리즘의 설계

- 원하는 작업이 이루어 지도록, 절차를 기본 명령어들을 표현한 것
ex) 학생들의 이름, 과목별 점수를 받아 각 과목 평균과 학생 별 등수 출력
- 효율적인 알고리즘의 사용이 중요

■ 자료구조

- 알고리즘은 데이터에 대하여 작업을 수행
- 데이터의 개수가 많아지면 원하는 작업에 적합하도록 배치/구성하는 것이 필요
- 자료구조의 종류: 변수, 배열, 리스트, 스택, 큐, 트리, 그래프 ...

자료구조

■ 자료 구조 (Data Structure)

- 자료값의 집합에 특정 규칙(구조)를 부여하여 저장한 것
- **자료값 집합**과 일정시간에 수행 가능한 기본 연산을 가짐
- 기본연산은 언어에서 제공하거나 따로 알고리즘으로 작성됨
- 기본 연산의 구성 : 생성자(createor, constructor), 변환자, 관찰자

예1) 정수 배열

➤ 자료집합 : 정수집합

➤ 연산 :

Create : $\text{int} \rightarrow \text{intArray}$

// int x[10];

read : $\text{intArray} * \text{int} \rightarrow \text{int}$

// y = x[2];

write : $\text{intArray} * \text{int} * \text{int} \rightarrow \text{intArray}$

// x[5] = 2;

자료구조

예2) 실수 스택(Stack) : Last In First Out

➤ 자료 집합 : 실수집합

➤ 연산 :

Create : $() \rightarrow \text{realStack}$

Push : $\text{realStack} * \text{real} \rightarrow \text{realStack}$

Pop : $\text{realStack} \rightarrow \text{real} (* \text{realStack})$

IsEmpty : $\text{realStack} \rightarrow \text{boolean}$

추상적 자료 구조 (Abstract Data Structure)

- 추상적 자료 구조 (Abstract Data Structure)
 - 자료구조의 사용(명세, specification)과 내부 구현이 분리
 - 구현 방법을 변경해도 자료구조를 사용하는 프로그램은 그대로 유지
 - 자료구조를 사용하기 위해 구현을 신경쓸 필요가 없음
 - 예: 스택의 사용

```
intStack s = createIntStack();  
push(s, 1);      // 구현이 없어도 이해 가능  
push(s, 2);      // ← 구현과 명세의 분리  
x = pop(s);  
push(s, 3);  
y = pop(s);  
z = pop(s);
```

알고리즘

- 주어진 문제를 해결하기 위한 방법을 명령어로 유한하게 기술한 것
- 알고리즘의 필요조건
 - 입력 : 0개 이상의 입력이 존재
 - 출력 : 한 개 이상의 출력이 존재
 - 명확성 : 명령어는 수행 환경에서의 의미(동작)가 명확해야 함
 - 유효성 : 명령어는 주어진 수행 환경에서 일정한 시간 내에 실행 가능
 - 유한성 : 유한 시간 내에 (한정된 단계 내에) 종료
- 알고리즘 ≠ 프로그램
- 하나의 문제 해결을 위해서는 다양한 알고리즘의 사용이 가능
 - 알고리즘의 선택이 프로그램의 효율성을 위해서 가장 중요
 - 하나의 알고리즘은 여러 가지의 프로그래밍언어로 구현 가능

알고리즘의 분석: 정확성과 성능

■ 정확성

- 항상 **입력**에 대하여 **정확한 결과값**을 항상 **유한한 시간 내에** 계산해 내는가?
 - 정확한 결과값의 **정의** 필요
 - 입력값의 **범위**가 명확해야 함 (일반적인 프로그래밍 언어의 타입으로는 부족)
ex) fibo 함수는 **0보다 큰 정수**에 n 에 대하여 n 번째 **피보나치 수**를 계산한다.
- 유용한 도구: 수학적 귀납법 (mathematical induction)
 - $(P(0) \text{ 증명}) + (P(n) \text{이면 } P(n+1) \text{도 성립 증명}) \rightarrow \forall n, P(n) \text{ 증명}$
 - Strong Induction : $(P(0) \text{ 증명}) + (\forall i < n, P(i) \text{이면 } P(n) \text{ 성립 증명})$

■ 성능

- 결과를 계산하는데 요구되는 **시간** 및 **기억 공간 크기**는?
- 일반적으로 입력값 또는 대상 자료구조의 크기에 따라 달라진다.

피보나치 수열

- 피보나치 수열 : $a_1, a_2, a_3, a_4, \dots$

- 정의: 정확성 분석의 기준

$$\begin{aligned} a_1 &= 1, a_2 = 1 \\ a_n &= a_{n-1} + a_{n-2} \quad (n \geq 2) \end{aligned}$$



재귀적인 정의


피보나치 수열

■ 알고리즘 1

```
int fibo(int n) {  
    int fib1=1, fib2=1;  
    int j, fib;  
    if (n==1 || n==2)  
        return 1;  
    for (j=3; j++; j<=n) {  
        fib = fib1+fib2;  
        fib1 = fib2;  
        fib2 = fib;  
    }  
    return fib;  
}
```

■ 알고리즘 2

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

- 
- 정확한 알고리즘인가?
 - 성능은?

피보나치 수열

- fibonacci 함수의 수행 시간의 예

n	알고리즘1	알고리즘2
10	0 sec. (8)	0 sec. (54)
20	0 (18)	0.01 (6764)
30	0 (28)	0.05 (832039)
40	0 (38)	5.16 (102334154)
45	0.01 (43)	57.14 (1134903169)

알고리즘의 분석

- 알고리즘과 자료구조
 - 자료들의 집합을 사용하는 경우 저장 형태에 따라 적절한 알고리즘 결정
 - 알고리즘과 자료구조는 상호 보완적인 특성을 가진다.
- 적절한 자료구조의 선택
 - 속도의 중요성, 메모리의 제약, 자료의 성격, 프로그램 사용 형태 등 고려
- 알고리즘의 성능 분석
 - 경험적 분석(Performance measurement) : 실행 상황에 의존적
 - 수학적 분석(Performance analysis) : 복잡도
 - 실제 컴퓨터가 아닌 기본적인 명령어를 수행하는 가상의 컴퓨터를 기준으로 함(machine Independent)
 - 점근적 복잡도 (Asymptotic Complexity)

복잡도 (Complexity)

- 입력의 크기 (n)에 대한 자원(메모리, 수행시간)의 사용량을 식으로 표현
- 공간 복잡도(Space Complexity), 시간 복잡도(Time Complexity)
- 예

```
int sum(int n) {
    int *b = malloc(n*sizeof(int));
    int i, s = 0;
    for (i=0; i<n; i++) {
        b[i] = i;
        s += b[i];
    }
    return s;
}
```

$$- S_{\text{sum}}(n) = c + n * (\text{sizeof}(\text{int}))$$

$$- T_{\text{sum}}(n) = T_{\text{malloc}} + T_{=} + T_{=} + n * (T_{b[i]=} + T_{+=} + T_{<} + T_{++}) + T_{<} + T_{\text{return}}$$

복잡도 (계속)

- 복잡도는 입력에 따라 달라질 수 있다.

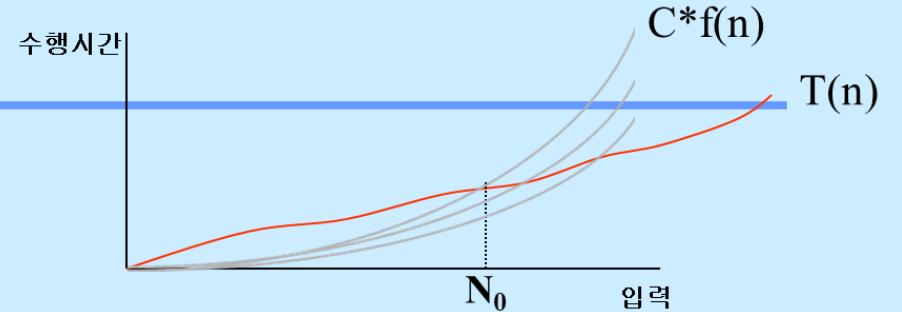
```
int get(int a[], int n, int v) {  
    int i;  
    for (i=0; i<n; i++)  
        if (a[i] == v) return i;  
    return -1;  
}
```

- 모든 경우 복잡도 (every-case complexity : $T(n)$)
 - 최악의 경우 복잡도 (worst case complexity : $W(n)$)
 - 평균의 경우 복잡도 (average-case complexity : $A(n)$)
- 복잡도를 정확한 식으로 표현하는 것이 합당한가?
 - 너무 복잡함 (명령어 단위의 기준은?)
 - 컴퓨터에 따라 실제적인 자원 요구량은 달라질 수 있음
 - 필요 없는 정보를 모두 포함하여 사용성이 떨어짐

점근적 복잡도 (Asymptotic Complexity)

- 복잡도 식의 차수를 복잡도 비교의 기준으로 사용
 - $T(n) = 1000*n + 2000$ 인 알고리즘과 $T(n) = n^2$ 인 알고리즘 중에서 어느 것이 더 효율적인 알고리즘인가?
 - 가장 높은 차수항이 의미를 가진다.
 - 일반적으로 정확한 $T(n)$, $S(n)$ 은 계산이 불가능
 - 어떤 경우에는 정확한 차수를 계산할 수 없는 경우도 있다.
- 상한 성능
 - 알고리즘의 복잡도 식의 차수의 상한값을 가지고 해당 차수의 기본항으로 복잡도 표시
 - O 표기법을 사용한다.
 - ex) 알고리즘 A는 입력 데이터의 크기 n 에 대하여 $O(n^2)$ 복잡도를 가진다.
 - 가능한 한 낮은 차수의 식을 선택

점근적 복잡도



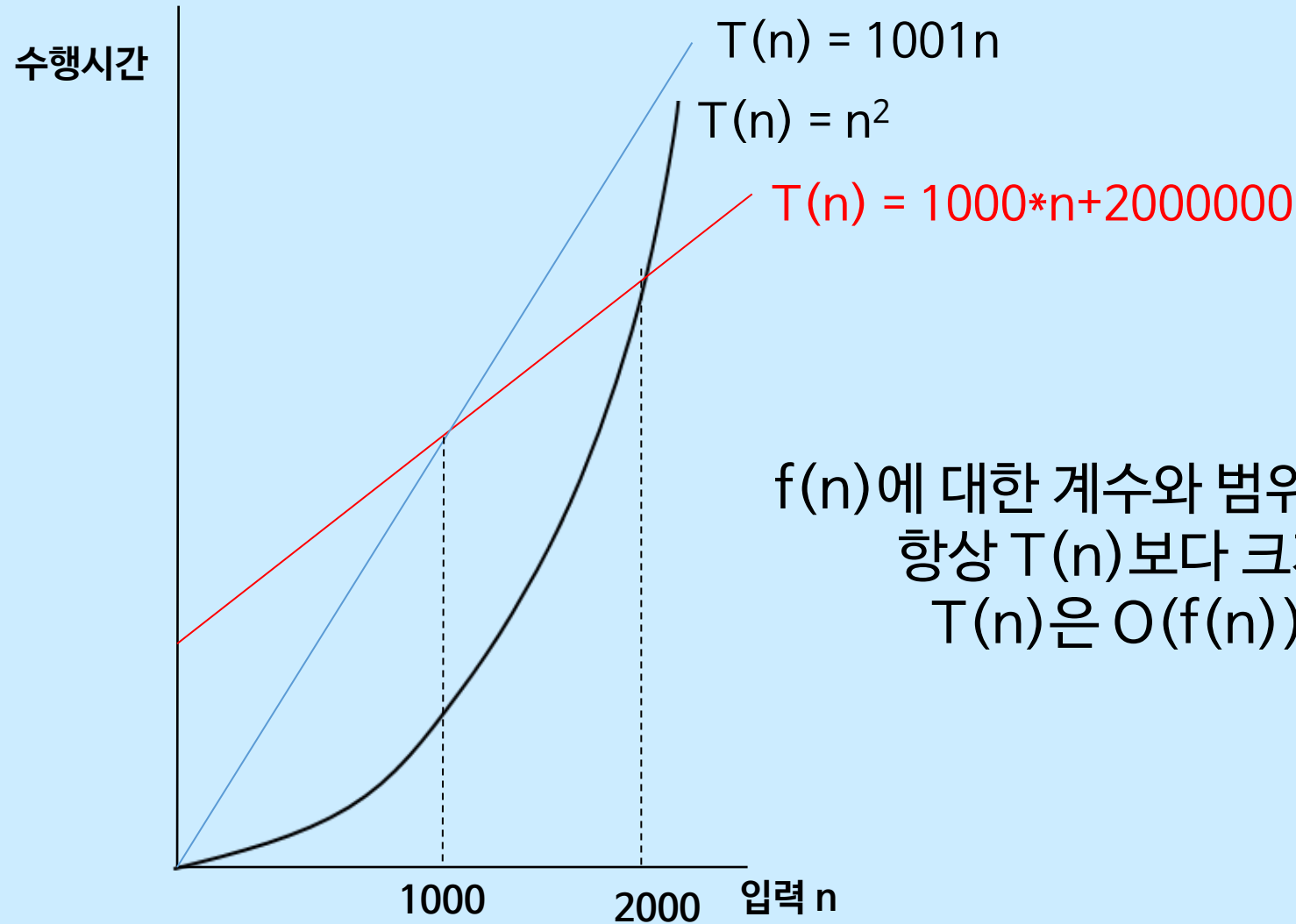
- O 표기법의 엄밀한 정의

어떤 알고리즘의 복잡도가 $T(n)$ 이라고 할 때,
 $n \geq N_0$ 인 모든 정수 n 에 대하여 $C * f(n) \geq T(n)$ 인 상수 C 와 N_0 가 존재하면
 해당 알고리즘의 점근적 상한 복잡도를 $O(f(n))$ 으로 표시한다.
- 예: 알고리즘 A의 $T(n)$ 이 $T(n) = 1000 * n + 2000000$ 일 때

 - $f(n) = n$, $C = 1001$, $N_0 = 2000000$ 이면
 $n \geq 2000000$ 인 모든 정수 n 에 대하여 $1001 * n \geq 1000 * n + 2000000$ 이 성립
 \Rightarrow 따라서, 알고리즘 A는 $O(n)$ 이다
 - $f(n) = n^2$, $C = 1$, $N_0 = 2000$ 이면
 $m \geq 2000$ 인 모든 정수 m 에 대하여 $1 * m^2 \geq 1000 * m + 2000000$ 이 성립
 \Rightarrow 따라서, 알고리즘 A는 $O(n^2)$ 이다

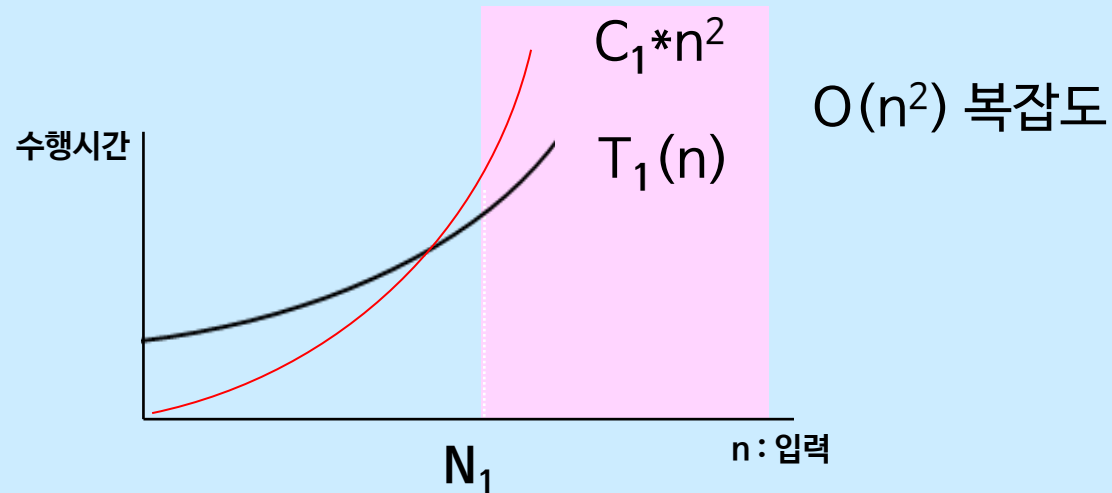
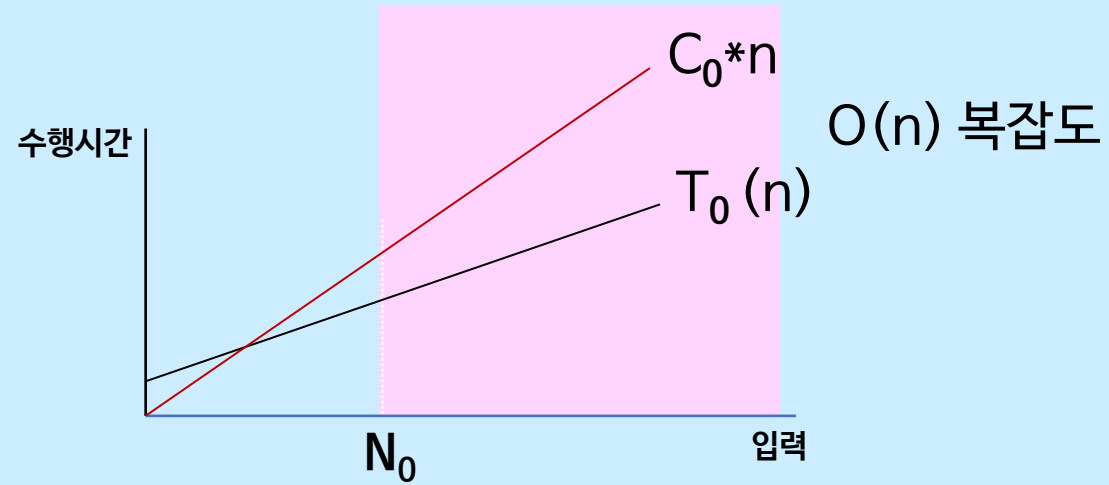
- 1), 2) 모두 성립하지만 우리는 1)을 선호한다.

점근적 복잡도



$f(n)$ 에 대한 계수와 범위의 시작을 크게 잡아서
항상 $T(n)$ 보다 크게 만들 수 있으면
 $T(n)$ 은 $O(f(n))$ 으로 인정한다!

점근적 복잡도



점근적 복잡도

■ Complexity 의 종류 (Complexity Categories)

$O(1)$: constant

$O(\log_2 n)$: logarithmic

$O(n)$: linear

$O(n \cdot \log_2 n)$: log-linear

$O(n^2)$: quadratic

$O(n^3)$: cubic

polynomial
time

 $O(2^n)$: exponential

$O(n!)$: factorial

exponential
time

점근적 복잡도

- Ω 표기법 : 하한 성능

$n \geq N_0$ 인 모든 정수 n 에 대하여 $T(n) \geq C * (f(n))$ 를 만족하게 하는 상수 N_0 와 C 가 존재하면 해당 알고리즘의 복잡도를 $\Omega(f(N))$ 이라고 한다

- θ 표기법

$n \geq N_0$ 인 모든 정수 n 에 대하여 $C_0 * (f(n)) \leq T(n) \leq C_1 * (f(n))$ 을 만족하게 하는 상수 N_0 와 C_0, C_1 이 존재하면 $T(N) = \theta(f(N))$ 이라고 한다

점근적 복잡도

■ 예 : Time Complexity : 순차검색

```
int find(int a[], int size, int key) {  
    int i = 0;  
    for (i=0; i++; i<size)  
        if (a[i] == key) return i;  
    return -1;  
}
```

- 속도에 영향을 끼치는 요인은 ?
- 실제 수행 시간은 ???
- 복잡도 (점근적 복잡도)
 - 모든 경우 : 적용되지 않음.
 - 최악의 경우 복잡도 : $\Theta(n)$ ($O(n)$ and $\Omega(n)$)

점근적 복잡도

- 예 : Time Complexity : 같은 원소 제거

```
void replace_dup_with_0 (int *a, int n) {  
    int i=0, j=0;  
    for ( i = 0; i<n-1; i++)  
        for (j = i+1; j<n; j++)  
            if (a[i] == a[j]) a[j] = 0;  
}
```

- 가장 많이 반복되는 문장의 반복 횟수 :
 $(n-1) + (n-2) + \dots + 1 = n(n-1)/2$
- 모든 경우 복잡도 적용(최악의 경우 == 평균의 경우)

Asymptotic Time Complexity : $\Theta(n^2) \leftrightarrow (O(n^2) \text{ and } \Omega(n^2))$

점근적 복잡도

■ 예 : Time Complexity : 이분검색

```
int find(int a[], int size, int key) {  
    int low = 0, high = size-1;  
    int mid;  
    while (low <= high) {  
        mid = (low + high)/2;  
        if (a[mid] < key)  
            low = mid+1;  
        else if (a[mid] > key)  
            high = mid-1;  
        else  
            return mid;  
    }  
    return -1;  
}
```

→ Asymptotic Time Complexity : $\Theta(\log n)$ ($O(\log n)$ and $\Omega(\log n)$)

점근적 복잡도

- 예 : Time Complexity : 피보나치 수열

```
int fibo(int n) {  
    int fib1=1;  
    int fib2=1;  
    int j, fib;  
    if (n<1) return -1;  
    if (n==1 || n==2) return 1;  
    for (j=3; j++; j<=n) {  
        fib = fib1+fib2;  
        fib1 = fib2;  
        fib2 = fib;  
    }  
    return fib;  
}
```

→ Asymptotic Time Complexity : $\Theta(n)$ ($O(n)$ and $\Omega(n)$)

점근적 복잡도

- 예 : Time Complexity : 피보나치 수열

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

: 수행시간 $T(n) = T(n-1) + T(n-2) + c$

→ Asymptotic Time Complexity : $O(3^{n/2})$, $\Omega(2^{n/2})$

점근적 복잡도

■ 예 : Space Complexity

```
float sum(float list[], int n) {  
    float tempsum = 0;  
    int i;  
    for(i = 0; i < n; i++)  
        tempsum += list[i];  
    return tempsum;  
}
```

- C passes arrays by pointer → **O(1) Space Complexity**
- Pascal passes arrays by value → **O(n) Space Complexity !**

수행 시간의 측정

■ 알고리즘의 실행 시간 측정

- sys/time.h (linux)

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
struct timeval {
    time_t      tv_sec;        /* seconds */
    suseconds_t tv_usec;      /* microseconds */
};
```

- 사용 예

```
#include <stdio.h>
#include <sys/time.h>
main() {
    struct timeval tstart, tfinish;
    double tsecs;
    gettimeofday(&tstart, NULL);
    foo();
    gettimeofday(&tfinish, NULL);
    tsecs = (tfinish.tv_sec-tstart.tv_sec) +
            1e-6 * (tfinish.tv_usec-tstart.tv_usec);
    printf("%f\n", tsecs);
}
```

프로그램의 개발

- 프로그램의 개발 단계
 1. 요구 사항 파악
 2. 알고리즘과 자료구조의 설계
 3. 알고리즘 분석
 4. 프로그램의 작성
 5. 검증 : 증명, 테스트, 검토(review), 오류 수정
- 효율적인 프로그램 작성을 위한 Tips
 - 좋은 알고리즘과 자료구조의 선택이 가장 중요
 - 반복되는 부분(=가장 많이 수행되는 부분)을 최대한 간단히 하라.
 - 함수 사용을 적절히 한다. (속도/코드크기/ 가독성)
 - 실수형의 사용을 줄인다.

프로그램의 개발

- 좋은 프로그램의 작성
 - 정확한 프로그램을 작성하라
 - 나 자신에게 엄밀하게 설득시켜라
 - 모든 가능한 입력에 대하여 고려하라.
 - divide and conquer
 - 작업이 충분히 간단할 때까지 나누어서 생각하라
 - 분할의 단위 : 함수, 클래스(객체)
 - 추상 자료 구조
 - 명세와 구현을 분리
 - 재사용 가능한 함수를 작성
 - 전역 변수의 사용을 피한다.
 - 함수의 타입을 구성하는 요소만으로 함수의 동작을 정확히 설명할 수 있도록 !

정리

- 자료구조와 알고리즘의 개념
- 알고리즘의 분석 : 정확성과 성능
- 알고리즘의 복잡도
 - 점근적 복잡도
- 프로그램 수행 시간의 측정
- 안전하고 효율적인 프로그램 작성