1장 기본개념

1.1 개요: 시스템 생명주기 (시스템: large-scale program)

⇨ 시스템: Input + Processor + Output
⇨ 시스템 생명주기 (system life cycle): 프로그램 개발단계
(시스템 개발을 위한 일련의 단계)

(1) 요구사항 (requirements) 분석
- 문제에 대한 적절한 해를 구하기 위한 요구조건을 정의
- 프로젝트들의 목적을 정의한 명세 (specification) 들의 집합

(2) 명세 (specification)
- 기능적 명세 (functional specification)
  . 입력/출력 명세 – 구문적 제약조건 (비 절차적 명세)
  . 기능적 명세 – 의미적 제약조건 (절차적 또는 비절차적)

(3) 설계(design)
- 명세된 기능을 어떻게 달성하는가를 기술 (추상적 언어: Pseudo code)
- 전체 시스템 설계 : top-down, bottom-up
- 추상 자료형 (abstract data type) – 자료객체와 연산
- 자료 객체 (object)들과 수행될 연산 (operation) 의 정의
  (예: 수강등록 시스템 – student, course, 등을 object 라고 하고,
  이 object 에 적용할 insert, remove 등이 operation 이다)

(4) 구현(implementation) – coding
- 구조화 프로그래밍 (assignment, conditional, loop)
- modular 프로그래밍 (function, entry/exit point)

(5) 검증(Verification)

- 테스트(testing): 프로그램의 수행 검증, 프로그램 성능 검사
  (모듈검사, 통합검사, 시스템 검사..)
- 정확성 증명(correctness proofs):
  수학적 기법들을 사용하여 프로그램의 정확성 증명
- debugging - 오류제거

(6) 운영 및 유지보수 (operation and maintenance)

- 시스템 설치(installation), 운영, 유지보수
- 새로운 요구조건 – 변경
- 수정 내용 기록 유지

1.2 소프트웨어 설계 방법론

(1) 하향식(top-down): 크고 복잡한 시스템 개발시 사용

- 문제들을 실제 다룰 수 있을 정도의 작은 단위들로 나눔
- 프로그램을 독립된 기능을 수행하는 작은 세그먼트로 분리

abstract program (initial solution)
↓ decomposition (refinement)
concrete program (executable by a target machine)

(2) 상향식(bottom-up): 하위레벨 부터 상세히 프로그램을 작성

Module 개발(하나의 기능수행)
↓ . 작은 문제를 해결, 프로그래밍언어로 직접 개발, 하나의 기능을 수행

module 통합(시스템 통합)
- 전체 시스템: 하나의 완전한 프로그램

1.3 데이터 추상화 와 캡슐화

- 대표적 data type:

기본 데이터 타입(int, float,..), 파생 데이터 타입(포인터, 참조), 데이터 집단화(array, struct, class), 사용자 정의..

⇨자료가 얼마나 잘 구조화 되어 있는가에 따라 프로그램의 속도, 개발시간 유지보수의 비용이 결정됨

⇨자료구조는 데이터를 유용하게 구조화 할 수 있는 다양한 방법론을 study한다. (data encapsulation, data abstraction,..)

• 데이타 캡슐화(Encapsulation)

- 선언된 Class 의 정보은폐 기능지원(ex. C++의 Private, public,..)
- 관련있는 데이터 및 함수를 하나로 캡슐화하여 정보은닉.
- 간접적 접근경로는 제공해야함.

Ex)캡슐화 실패 (Class 추가시..)

```
Class Point {
      Int x;     int y;
   Public:
      Int GetX() {return x;}
      Int GetY() {return y;}
      Void SetX(int _x);
      Void SetY(int _y);
   }
Int main() {
     …Point p;
      p.SetX(x);   p.SetY(y); PointShow show;   show.Showdata(p);
…
```

```
Class PointShow {
   Void Showdata(point p) {
       cout <<…<<p.GetX()<<endl;
             ….
```

Ex) 캡슐화 성공

```
Class Point {
        Int x;     int y;
   Public:
      Int GetX() {return x;}
      Int GetY() {return y;}
      void SetX(int _x);
      void SetY(int _y);
      void Showdata();
   }
Int main() {
     …    Point p;
  p.SetX(x);    p.SetY(y); p.Showdata();
```

```
void Point::Showdata() {
    ….
```

- **데이타 추상화 (data abstraction)**

- 무엇과 어떻게를 독립=> 데이터 객체의 명세와 구현을 분리
- 여러 자료형을 하나로 묶어 단순화(추상화)하는것 (ex. 구조체)

⇨ **추상자료형 (Abstract Data Type: ADT)**

. 자료 및 연산을 하나의 단위로 묶어, 외부로부터 내부자료를 접근 못하게 함.（사용자 정의 자료형, 사용자 정의연산）

Ex)Dragon 을 추상화 하면,

```
     Class Dragon {
        Private:
           double x,y;      ➔     데이터
           int    color;
           …
        Public:
           void fly();      ➔   기능
           void walk();
  };
 int main(void) {
     Dragon d; // Dragon  생성..
     d.walk();
     …      //walk()문제 발생시, walk()함수만 점검하면 된다.
```

*** 추상화 와 캡슐화의 장점**

⑴ 소프트웨어 개발의 간소화: 복잡한 작업을 부분작업들로 분해 (분업 가능)

⑵ 검사와 디버깅의 단순화: 부분 작업

⑶ 재 사용성: 자료 구조가 시스템에서 별개의 개체로 구현

1.4 알고리즘 명세

알고리즘(Algorithm)의 정의: (An algorithm is a finite set of instructions that accomplishes a particular task)

. 특정한 일을 수행하기 위한 명령어의 유한 집합
. 동일한 문제에 여러 개의 알고리즘이 존재함
(예: 전화번호부-> 무순서(순차적방법), 사전식배열(이진탐색)

알고리즘은 다음조건 (criteria)을 만족해야함

i. 입력(input): 0 or more are externally provided

ii. 출력 (Output) : 적어도 한 개 이상의 결과가 생성됨

iii. 명확성(definiteness) : 모호하지 않은 명확한 명령

iv. 유한성(finiteness) : 종료

v. 유효성(effectiveness): 기본적, 실행가능 명령

ex. program ≒ algorithm
(알고리즘은 유한 단계를 거친 후 반드시 종료, 프로그램은 반드시 종료는 아님, 예: 운영체제는 실행할 job 이 없으면 대기상태로 감)

-Algorithm 기술방법

. 자연어 : (정확성 결여) . flowchart : (명백성과 모호성의 결여)

. 프로그래밍 언어 : (문법상의 복잡성)

. 의사코드(pseudocode) : best way

● **Pseudocode 작성**

1)Assignment ; causes a value to be assigned to a variable

variable <- expression

ex) {Compute sum of two numbers, first and second}

{And the result is assigned to SUM}

```
BEGIN
    INPUT first and second
    sum ← first + second
 END.
```

Ex) {find maximum of three numbers a,b,c}

```
Input a,b,c
large:= a
 if b>large    then large:= b
 if c>large    then large:= c
 return large;
```

Ex) Find minimum of the three numbers a,b,c ➔

2)Control statements ; flow of control through algorithm

- Sequence, **Condition, Iteration**

- **Sequence**: list of statements to be executed as a single unit (above examples)
- **Conditional:** if-then or if-then-else

- if P then action, if P then action1 else action2
- if P then begin action1; action2; …. action N end

- **Iteration (loops) For, while, repeat,..**

Ex) Find sum of first n odd numbers (n개 홀수의 합)

```
Procedure find_odd
 begin
   sum <- 0
   I ← 1
   input n
   while I ≤ n do
      begin
        sum ← sum + I ;
            I ← I+2
      end
   output sum
end
```

ex) Testing whether a positive integer is prime

```
procedure is_prime(m)
   for i=2 to m-1 do
      if m MOD i=0 then return false
      else return(true)
end is_prime.
```

Ex) finding a prime larger then a given integer

```
Procedure large_prime(n)
   m=n+1
   while not(is_prime(m)) do
         m=m+1
   return m
end large_prime
```

Ex) Find largest in a finite sequence (while loop)

```
Ex) Procedure find_large (s,n)
     begin
        large := s1
        i:=2
        while i  ≤  n do
          begin
            if si >large then large := si
            i:= i+1
          end
        return(large)
     end find_large
```

<u>예제 1 [이진탐색] :   정수 searchnum 이 배열 list 에 있는지 검사</u>

. list[0]   <= list[1] <= ...   <= list[n-1]   /* 미리 정렬되어 있음 */

. list[i] = searchnum 인 경우 인덱스 i 를 반환, 없는 경우는 -1 반환

(초기 값 :   left = 0, right = n-1;

list 의 중간 위치 :   middle = (left + right) / 2)

* list[middle] 과 searchnum 비교 시 다음 3 가지중 하나를 선택

1) searchnum < list[middle]: /* search again between left and moddile-1 */

2) searchnum = list[middle]: /* middle 을 반환 */

3) searchnum > list[middle]: /* search again between middle+1 and right*/

- **Version 1**

```
while (left <= right) {
      middle = (left + right) / 2;
      if (searchnum < list[middle])              right = middle - 1;
      else if (searchnum == list[middle])        return middle;
      else left = middle + 1;
}
```

- **Version 2**

```
   while (left<=right)    {
      middle = (left + right)/2;
      switch (COMPARE(list[middle], searchnum)) {
         case   -1: left = middle + 1;           break;
         case     0: return middle;
         case     1: right = middle - 1;       break; }
   }
   return -1;      }
```

```
char compare (int x, int y)
{
   if (x > y) return 1;
   else if (x < y) return -1;
   else return 0;
}
```

[문제 #1] 다음 코드를 완성하고 실행결과를 보이시오

```
#include <iostream.h>

int binarySearch(int data[], int num, int left, int right);

void main()
{
    int data[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int num, found;

    cout << "Enter an integer to search : ";
    cin >> num;
    found = binarySearch(data, num, 0, 9);

    if (found == -1)
            cout << "Not in the list" << endl;
    else
            cout << "Found at position " << found << endl;
}
```

* 순환 알고리즘 (Recursive Algorithm)

- 수행이 완료되기 전에 자기 자신을 다시 호출
- 순환 알고리즘의 장단점.
  . 장점 : 대단히 강력한 알고리즘 표현 방법일 뿐 아니라 복잡한 알고리즘의 과정을 명료하게 표현 가능
  . 단점 : 비 순환 알고리즘보다 시간(time)과 공간(space)면에 있어 비효율적이다.

-순환 알고리즘의 구성
  . 반드시 순환호출을 끝내는 **종료 조건**이 있어야 한다.
  . 종료 조건에 접근하는 다음 단계의 **순환호출**이 있다.

* Factorial function
- 반복적 정의: $n! = n*(n-1)*(n-2) \ldots\ldots *2*1$
- 순환적 정의: $n! = 1 \quad \text{if } n=1$
  $n*n(-1)! \quad \text{if } n>1$

ex) recursive factorial

```
int factorial (int n)
{
    if (n == 0)   return 1     /* anchor (종료조건)*/
    else return
       (n*factorial(n-1));     /* recursive step(순환호출) */
}
```

3! = 3*2! = 3*2*1! = 3*2*1*0! (0! =1)

***   Factorial 의 반복 프로그램 (non-recursive factorial)**

```
 int factorial(int n)
 {
    int fact = 1;
    for (int i=0;   i<n;   i++)
         fact = fact*i;
    return fact;
  }
```

ex) 이진 탐색

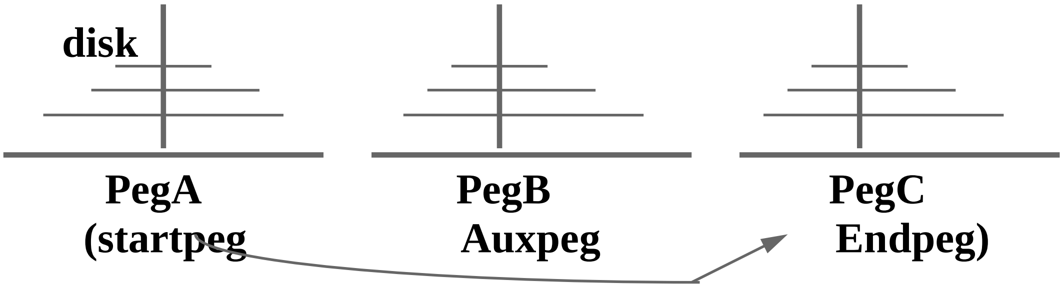
```
int binsearch(int list[], int searchnum, int left, int right)
{
   /* search list[0]     list[1]   ...     list[n-1] for searchnum*/
  int middle;

   if (left <= right)   {
      middle = (left + right) / 2;
      switch (COMPARE(list[middle], searchnum)) {
         case   -1: return
                      binsearch(list, searchnum, middle+1, right);
         case    0: return middle;
         case    1: return
                      binsearch(list, searchnum, left, middle-1);
      }
   }
   return -1;
}
```

ex) Towers of Hanoi

$m_k = 2m_{k-1} + 1$ $(k>= 2)$, $m_1 = 1$ $(k= 1)$

then $m_n = 2^n - 1$ for $n>= 1$

Requirement: 1) move one at a time 2) smaller disk on top of larger disk



- Algorithm: move n disk from startpeg to endpeg using auxpeg)

1) if only one disk, output "move disk from startpeg to endpeg"
2) else
    move (n-1 disk from startpeg to auxpeg using endpeg)
    output "Move disk from startpeg to endpeg"
    move (n-1 disk from auxpeg to endpeg using startpeg)

```
void towerHanoi (char from, char to, char aux, int n)
{
  if (n==1)
       printf("move disk 1 from peg%c to peg%c\n", from , to);

  else {
      towerHanoi(from, aux, to , n-1);
      printf("move disk %d from peg %c to peg %c\n ", n, from, to);
      towerHanoi(aux, to, from, n-1);
      }
  }
  ................
    towerHanoi('A', 'C', 'B', num);
```

성능 분석 (performance Analysis)

1. Program 을 평가하는 요소

(1) 본래의 개발요구사항의 충족여부
(2) 정확성 (works correctly?)
(3) 충분한 documentation 의 여부 (how to use it and how it works)
(4) Logical unit 별로 module 화 여부 (5) Readability
(6) Space complexity (memory utilization, 효율성)
(7) Time complexity (efficiency of running time)

2. 알고리즘 분석 시 고려사항

1) 공간 복잡도(space complexity)
프로그램을 실행시켜 완료하는데 필요한 공간의 양
(amount of memory required by storage structure)

2)시간 복잡도(time complexity)
프로그램을 실행시켜 완료하는데 필요한 컴퓨터 시간의 양
(amount of time required to execute the algorithm)

3. **Space Complexity (공간 복잡도)**

● 프로그램의 실행에 필요한 공간 S(p)

*공간 요구량 : $S(P) = c + S_P(I)$

($S(P)$: 전체공간, c: 고정공간, $S_P(I)$: 가변공간)

. 고정기억공간 요구: 프로그램 입출력 횟수나 크기와 관계없는 공간 요구
(fixed space requirements)
ex) 프로그램 코드(명령어) 저장 공간, 단순변수, 고정 크기의 구조화 변수,
상수 등 저장할 공간

[예제 1]:

```
float abc(float a, float b, float c)
{
    return a+b+b*c + (a+b-c) / (a+b) + 4.00;
}
```

소요공간: 1) 변수 3 개 (a,b,c)를 위한 공간

2) 고정 공간 요구만을 가짐 $S_{abc}(I) = 0$

. 가변공간요구: variable space requirements

. 가변크기의 구조체 (예: A[]).

. 문제의 instance i 에 의존하는 공간

. recursion 의 경우 추가공간 소요 (지역변수, 매개변수, return address 등)

[예제 2]

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

소요공간:

1) C/C++배열을 전달할 때, 배열의 주소를 전달함 (call by ref)

⇨ 따라서 소요공간은 배열의 주소 값 => 4bytes

2) 매개변수 n, 지역변수 tsum, I 의 공간 (4*3=12 bytes)

3) 가변 공간 없음… 모두 16bytes

[예제 3]   순환 알리즘

```
float rsum(float list[], int n)
{
    if (n) return   rsum(list, n-1) + list[n-1];
    return 0;
}
```

소요공간:

1) 배열주소 list 값을 위한 data 변수 => 4bytes
2) 매개변수 n=4bytes
3) 반환주소 공간: 4bytes
4) 따라서 매회 호출시 마다 3*4=12bytes 소요됨
   ⇨ n 번 호출시, 총 소요공간은 12*n

4. 시간 복잡도 (Time complexity)

* 프로그램의 실행에 필요한 **시간**
  (프로그램 P 에 의해 소요되는 시간 : T(P))

  1) 컴파일 시간 + 실행 시간 (T_p) 으로 구성됨
  2) 컴파일 시간은 프로그램의 특성에 영향없음
  ⇨ run time 만 고려

* Estimating run time is not easy
  - 가장 좋은 방법: 시스템 clock 사용
  - 다른 방법: 프로그램이 수행하는 연산의 횟수계산
    . 알고리즘의 기본연산의 수(number of basic operations in algorithm)
    . This gives us Machine-independent estimate

정의: 프로그램 단계**(program step)** 실행 시간이 인스턴스 특성에 상관없이 구문적으로 또는 의미적으로 독립성을 갖는 프로그램의 단위

- **Program step:** Meaningful segment of a program that has an execution time that is independent of the instance characteristics. (number of data,..)

Comments: 0 step
Declarative Statement: 0 step

Expressions and Assignment: 1 step, but it depends on expression

Iteration statement (for, while, ... sum of iteration)
If (expr) then (statement1) else (statement2): depends on expression and statements
Function invocation: 1 step
Begin, end,{{, }} 0 step
Function: 0 step

```
ex) 1.  float sum (float list[], int n);    0
    2.  {                                    0
    3.    float tempsum=0;                   1
    4.    int i;                             0
    5.      for (I =0; I<n ; I ++)          n+1
    6.          tempsum += list[I];          n
    7.    return tempsum;                    1
    8.  }                                    0
```

=> Total Steps: 2n+3

```
ex)  1.  void  add (int a[][max_size...]        0
     2.  {                                      0
     3.      int I,j;                           0
     4.      for (I=0; I<m; I++)                m+1
     5.          for ( j= 0; j<n; j++)          m(n+1)
     6.              c[I][j] = a[I][j] + b[I][j];   mn
     7.  }                                      0
         => total steps: 2mn+2m+1
```

```
ex)  sum = 0;                     1
     i = 0;                       1
     while (i < n) {              n+1
          cin >> num              n
          sum = sum + num;        n
          i++                     n
     }                            0
     mean = sum / n               1

 =>  total steps:   4n+4
```

ex) [수치 값 리스트의 합산을 위한 반복 호출]

```
 float sum(float list[], int n)          0
{                                        0
    float tempsum = 0;                   1
    int i;                               0
    for (i = 0; i < n; i++)              n+1
        count += 2;                      n
     count += 3;                         1
    return 0;                            1
}        =>  total steps in counts:    2n + 4 steps
```

< 점근 표기법 (O, Ω, Θ) > (Asymptotic/Order Notation)

* Step count 는 (either best or worst) difficult task, not precise
=> 정확한 단계의 계산: 무의미

* Order notation: 상수 인자나 적은수의 자료무시하고, 함수를 정의하거나, 비교하는 방법:

1) 알고리즘의 시간 복잡도를 표기하기 위한 방법
2) 알고리즘의 실제 수행시간이 아니라 명령어의 실행 빈도수를 함수로 표현한 것
3) 동일한 일을 수행하는 2 개의 서로 다른 알고리즘의 time complexity 를 비교할 수 있다.
4) 어떤 알고리즘의 특성변화에 따른 실행시간의 증가 추이를 예측할 수 있다.
   (Ex. 처리해야 할 자료의 개수 변화에 따른 실행시간 증가 추이 예측)

정의 [Big "oh"] [$f(n) = O(g(n))$]
iff $\exists c, n_0 > 0$, such that $f(n) \le cg(n)$ $\forall n$, $n \ge n_0$

⇨ f is of order AT MOST g(n) , if there exists positive constants C, such that $|f(n)| <= C|g(n)|$,

⇨ (g(n) 은 f(n) 의 상한선(upper bound)이 된다)

⇨ f(n) 의 수행시간이 g(n) 보다는 덜 걸린다

. $f(n) = O(g(n))$ 은 그 알고리즘이 n 개의 입력자료가 수행 될 때, 걸리는 시간이 $|g(n)|$ 에 상수 C 를 곱한 것보다 항상 같거나 작아진다는 의미.

ex) $3n+2 = O(n)$,

(sol) $3n+2 <= 4n$ for all $n >= 2$ 즉, 2 보다 큰 모든 n 에 대하여 $3n+2$ 는 $4n$ 보다 항상 작다. $n0=2, c=4$

ex) $1000 n^2+100n - 6 = O(n^2)$

(sol) $1000 n^2 +100n -6 <= 1001 n^2$, for $n>= 100$

ex) $6*2^n + n^2 \le 7*2^n$, for $n \ge 4$, => $6*2^n + n^2 = O(2^n)$

ex) $3n+3= O(n^2)$ is correct, but not this way.

ex) $f(x) = 100x^2-50x+2$

$f(n) = a_m n^m + ... + a_1 n + a_0$, then $f(n) = O(n^m)$

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

$O(1)$: computing time is constant

$O(n)$ is linear, $O(n^2)$ is quadratic,

$O(2^n)$ is exponential

*** Some Rule**

1) 상수는 무시한다.

$O(cf(n)) = O(f(n))$, ex) $O(3n^2) = O(n^2)$

2) 더 할 때는 max 를 택한다.

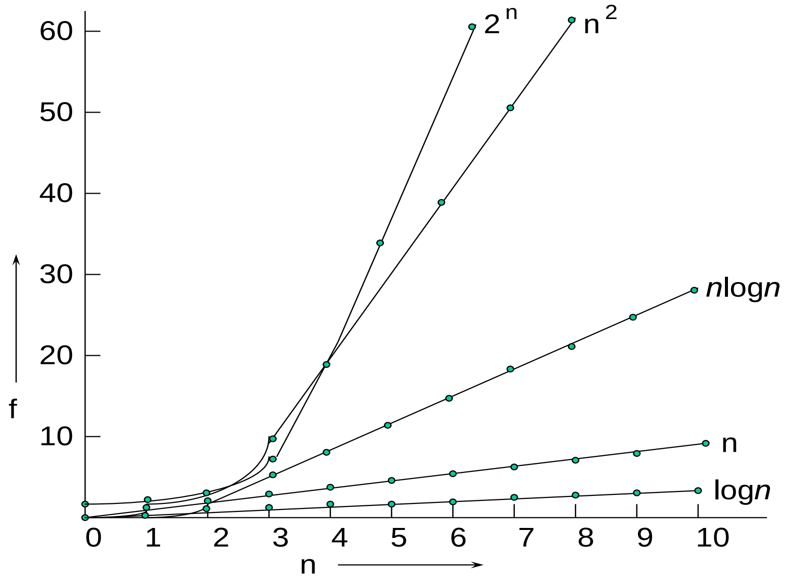
$O(f(n) + g(n)) = O(max(f(n), g(n))$

ex) $O(2n^3 + 108n) = O(n^3)$

3) $O(f(n))$ * $O(g(n)) = O(f(n)*g(n))$

ex) $O(n)*O(n-1)*O(nlogn) = O(n(n-1)nlogn) = O(n^3 logn)$

4) Assignment, read/write instruction = O(1)



ex) 어떤 프로그램이 다음과 같은 성질이 있는 f(n)과 g(n)으로 구성되었을 때 running time 을 계산하시오

f(n) = n^4 if n is even　　　g(n) = n^2 if n is even
n^2 if n is odd　　　　　　n^3 if n is odd

(sol) running time = O(f(n) + g(n))이며, O(max(f(n), g(n))이므로 O(n^4) if n is even, and O(n^3) if n is odd 이다.

ex) 다음 segment code 의 running time 을 구하시오

```
for i = 2 to n do    a[i] = 0
for i = 1 to n do
    for j = 1 to n do
        a[i] := a[i] + a[j]
```

=> (Sol) $O(n) + o(n^2) = O(n^2)$

정의 [Omega]　[$f(n) = \Omega(g(n))$] **iff $\exists c, n_0 > 0$, such that $f(n) >= c.g(n)$　$\forall n, n \ge n_0$**

$n \ge n_0$ 인 모든 n 에 대하여 $f(n) \ge c * g(n)$을 만족하는 양의 상수 c 와 n_0 가 존재한다면 $f(n) = \Omega(g(n))$ 이다.

- $g(n)$ 은 $f(n)$ 의 하한(Lower bound) 이다.
- $f(n)$이 $g(n)$ 이상의 시간이 걸린다

Ex) $n^3 + 2n^2 = \Omega(n^3)$ 이다

(sol) 이유: $n^3 + 2n^2 \geq n^3$ for all $n \geq 1$.
(즉 1 보다 큰 모든 n 에대하여 $n^3 + 2n^2$ 는 n^3 보다 항상 크다. $(n_0 = 1, c = 1)$
즉, 많은 하한 값들 중에서 제일 큰 값을 택하는 것이 타당하다.

정의 [Theta] $[f(n) = \Theta(g(n))]$
iff $\exists C_1, C_2, n_0 > 0,$ s.t $C_1 g(n) \leq f(n) \leq C_2 g(n), \quad \forall n, n \geq n_0$

=> $n \geq n_0$ 인 모든 n 에 대하여 $c_1 g(n) \leq f(n) \leq c_2 g(n)$을 만족하는
양의 상수 c_1, c_2 와 n_0 가 존재한다면 $f(n) = \theta(g(n))$ 이다.

($g(n)$이 $f(n)$에 대해 상한 값과 하한 값을 모두 가지는 경우)

⇨ $g(n)$ 은 $f(n)$ 의 상한(Upper bound) 인 동시에 하한 (Lower bound) 이다.
⇨ $f(n)$ is "theta of $g(n)$" 이라고 읽는다.

ex) $3n + 2 = \theta(n)$

(sol) $3n + 2 \geq 3n$ for all $n \geq 2$, $3n + 2 \leq 4n$ for all $n \geq 2$
$c_1 = 3$ and $c_2 = 4$

ex) $10n^2 + 4n + 2 = \theta(n^2)$ 이다

(sol) $10n^2 + 4n + 2 \geq 10n^2$ for all $n \geq 1$
$10n^2 + 4n + 2 \leq 11n^2$ for all $n \geq 1$, $c_1 = 10$ and $c_2 = 11$

* 결론: 점근적 복잡도(asymptotic complexity:O,Ω,Θ)는
정확한 단계수의 계산 없이 쉽게 구함