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 분석시 고려사항

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

2)시간 복잡도(time complexity)
알고리즘을 실행시켜 종료할 때까지의 소요시간
(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: fixed space requirements
  . 프로그램 코드(명령어)를 저장할 공간
  . 단순변수, 고장 크기의 구조화 변수, 상수등을 저장할 공간
  . 프로그램 입출력의 횟수나 크기와 관계 없는 공간

- 가변공간요구 : variable space requirements
  . 가변크기의 구조체 (예: A[]).
  . 문제의 인스턴스, I,에 의존하는 공간
  . recursion 의 경우 추가공간 소요 (지역변수, 매개변수, return address 등)

[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 3개 (a,b,c)를 위한 공간
2) 고정 공간 요구만을 가짐 $S_{abc}(I) = 0$

[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) 그리고 매개변수 n, 지역변수 tsum, I의 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) 배열주소값을 위한 data 변수 => 4bytes

2) 매개변수 n=4bytes

3) 반환주소 공간: 4bytes

4) 따라서 매회 호출시 마다 3*4=12bytes 소요됨

⇨ n번 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**

3) 정확하면 다시 컴파일하지 않음

⇨ 따라서 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: zero
Declarative Stmt: zero

Expressions and Assignment: 1, but it depends on expression
Iteration stmt (for, while, ... sum of iteration)
if (expr) then (stmt1) else (stmt2): depends on expression and statements
function invocation: one step
begin, end,{{, }} zero step
function: zero 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 실행빈도수를 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
∃c, $n_0 > 0$, such that f(n) ≤ cg(n) ∀n, $n \geq n_0$

⇨ 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) 최고 지수가 얼마나 빨리 증가하느냐가 관심사

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

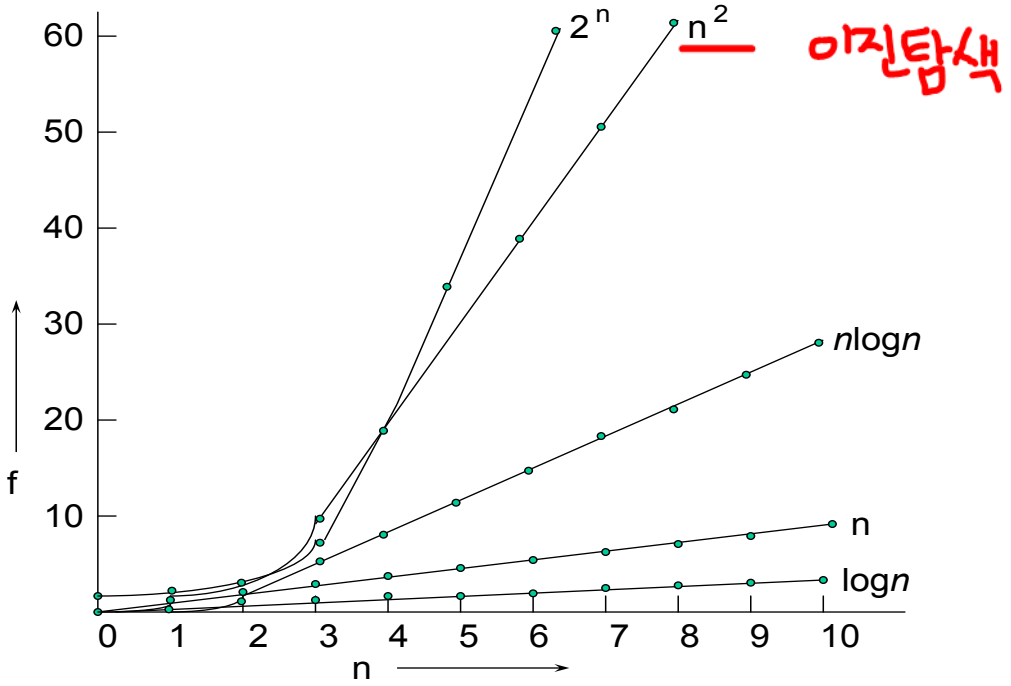
ex) 6*2^n + n^2 ≤ 7*2^n , for n ≥ 4, => 6*2^n + n^2 = O(2^n)

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



이진탐색

* 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) $O(n)*O(n-1)*O(n\log n) = O(n(n-1)n\log n) = O(n^3 \log n)$

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

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

$$f(n) = \begin{cases} n^4 & \text{if } n \text{ is even} \\ n^2 & \text{if } n \text{ is odd} \end{cases} \qquad g(n) = \begin{cases} n^2 & \text{if } n \text{ is even} \\ n^3 & \text{if } n \text{ is odd} \end{cases}$$

(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 code의 running time을 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) $O(n) + o(n^2) = O(n^2)$ ← MAX를 선택

하한선

정의 [Omega] $[f(n) = \Omega(g(n))]$

iff $\exists c, n_0 > 0$, s.t $f(n) >= c\,g(n)$ $\forall n, n \geq n_0$ ← 이어 | X

$n \geq n_0$인 모든 n 에 대하여 $f(n) \geq c * g(n)$을 만족하는 양의 상수 c 와 n_0가 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 ← $3n+2 = \Omega(n)$.

(sol) 이유: $n^3 + 2n^2 \geq n^3$ for all $n \geq 1$. (즉 1 보다 큰 모든 n 에 대하여 $n^3 + 2n^2$ 는 n^3 보다 항상 크다. ($n_0 = 1$, $c = 1$)

즉, 많은 하한 값들 중에서 제일 큰 값을 택하는 것이 타당하다.

정의 [Theta] $[f(n) = \Theta(g(n))]$

상한선 하한선 동시 만족

iff $\exists C_1, C_2, n_0 > 0$, s.t

$C_1 g(n) \le f(n) \le C_2 g(n), \quad \forall n, n \ge n_0$

⇒ $n \ge n_0$인 모든 n 에 대하여 $c_1 g(n) \le f(n) \le c_2 g(n)$을 만족하는 양의 상수 c_1, c_2와 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) $3n + 2 = \Theta(n)$ 상 → O(n) 하 → Ω(n) > 차수 같음 → Θ(n)

(sol) $3n + 2 \ge 3n$ for all $n \ge 2$, $3n + 2 \le 4n$ for all $n \ge 2$

$c_1 = 3$ and $c_2 = 4$

ex) $10n^2 + 4n + 2 = \Theta(n^2)$ 이다

(sol) $10n^2 + 4n + 2 \ge 10n^2$ for all $n \ge 1$

$10n^2 + 4n + 2 \le 11n^2$ for all $n \ge 1$, $c_1 = 10$ and $c_2 = 11$

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