Chapter 3. Algorithm

- 1. Algorithm definition

  - pseudocode

  - examples

  - Recursive algorithm

- 2. Analysis of Algorithm

  - Time Complexity

  - Order notation

1. Introduction

- An *algorithm* is a step-by-step method for solving some problem

- a finite set of instructions with the following characteristics:

  - **Precision**: steps are precisely stated
  - **Uniqueness**: The intermediate results of each step of execution are uniquely defined. They depend only on inputs and results of preceding steps.
  - **Finiteness**: the algorithm stops after finitely many steps
  - **Input/Output** : the algorithm receives input and produces output
  - **Generality**: the algorithm applies to various sets of inputs

Notation for algorithms

- methods: programming language (too detail), flowchart, pseudocode,…

- Pseudocode: English like + PL style (looks computer language such as C++ or Pascal), free of details easy to translate to PL.

- General form of PSEUDOCODE

```
BEGIN
    { statements}
END.
```

1.1 Pseudocode

<u>1)Assignment statement</u>: assigns value to a variable.

ex) Compute sum of two numbers, 1st and 2nd, And assigns result to SUM

BEGIN

INPUT first and second

sum ← first + second

END.

ex) Algorithm to find the largest of three numbers a, b, c:

1. input a, b, c
2. large= a ;; "copy the value of a into large"
3. If b > large then large= b
4. If c > large then large= c
5. return large.

Ex) find smallest of three numbers a,b,c,?

Pseudocode

2) **Control statements**: flow of control through the algorithm

a) sequence : List of statements to be executed as a single unit

Ex) procedure max(a,b,c)

```
begin
  x:=a;
  If b>x then  x:= b
  If c>x then  x:= c
  Return(x)
 End max
```

b) conditional: if-then or if-then-else

```
- if p then  action1  else  action2
- if p then begin
     action1, action2, … actionN
  endif
```

c) iterative (loop)

- FOR variable := initial value TO final value DO {Statements}
- WHILE (expression) DO { statements }
- REPEAT {Statements} UNTIL (condition)

1.2 Algorithm examples

1) Find sum of first n odd numbers

```
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
```

2) test a positive integer is prime

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

3) find a prime larger than a given integer

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

More examples

* Euclidean algorithm

finding the greatest common divisor between two integers.

Ex) GCD of 4 and 6 = 2

if a,b,q are integers, $b \neq 0$, satisfying $a = bq$, then

- b DIVIDES a,
- writes $b \mid a$ (q = quotient, b=divisor of a)

  (if does not divide, then we write $b \nmid a$)

- Theorem:

  If a is a nonnegative integer, b is a integer, and

  $a = bq + r$, $0 \le r < b$ then $\mathrm{Gcd}(a,b) = \gcd(b,r)$

GCD procedure

- Procedure gcd(a,b)

```
If a<b  then  swap(a,b)
While b NEQ 0 do
 begin
   divide a by b to obtain a = bq+r,  (0<= r <b)
   a:= b
   b:=r
 end
 return(a)
End GCD
```

1.3 Recursive algorithms

- A *recursive procedure* is a procedure that invokes itself (divide-conquer)

- A *recursive algorithm* is an algorithm that contains a recursive procedure

(Ex: **factorial of n** is defined as, n! = n(n-1)(n-2)…3.2.1)

- Ex)

```
Procedure fact(n)
  If n=0 then  Return (1)
  Return (n*fact(n-1))
End
```

```
Procedure  iterative-fact (n)
  X=1;
  For I = 1 to n    { x= i * x}
End
```

- Fibonacci sequence f_1, f_2,… defined recursively as follows:

$$f_1 = 1, \quad f_2 = 2$$

$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 3$$

- First terms of the sequence are: 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,…

Recursive algorithm example

- A Robot can take steps of 1meter or 2meters. An algorithm to calculate the number of ways the robot can walk n meters

Distance	sequence of steps	ways to walk
1	1	1
2	1,1 or 2	2
3	1,1,1 or 1,2 or 2,1	3
4	1,1,1,1 or 1,1,2 or 1,2,1 or 2,1,1 or 2,2	5

if n>2, if robot takes 1meter step, then n-1 meters remain
if robot takes 2meter step, then n-2 meters remain

Algorithm:

```
 Walk(n) {
    If (n==1 ∨n==2) return n
    Return walk(n-1) + walk(n-2)
}  * Fibonacci sequence
```

Recursive algorithm example (이진탐색)

- **<u>예제2 [이진탐색] : 정수 searchnum이 배열 list에 있는지 검사</u>**

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

=> list[i] = searchnum인 경우 인덱스 i를 반환, 없는 경우는 -1 반환

(초기값: left = 0, right = n-1; list의 중간 위치: middle=(left + right) / 2)

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

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

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

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

이진탐색 알고리즘

```
int binarysearch(int list[], int num, int left, int right) {
while (left <= right) {
    middle = (left + right) / 2;
    if (num < list[middle])      right = middle - 1;
    else if (num == list[middle])     return middle;
    else left = middle + 1;
}
int binsearch(int list[], int num, int left, int right) {
if (left <= right)  {
    middle = (left + right) / 2;
    if (num < list[middle]) binsearch(list, num, middle-1, right);
    else if (num == list[middle])  return middle;
    else binsearch(list, num, left, middle+1);
 }
 return -1;
}
```

Algorithm 4.4.2: Computing n Factorial

```
factorial(n) {
   if (n == 0)
      return 1
   return n * factorial(n - 1)
}
```

Algorithm 4.4.6: Robot Walking

Input: n

Output: $walk(n)$

```
walk(n) {
   if (n == 1 ∨ n == 2)
      return n
   return walk(n − 1) + walk(n − 2)
}
```

Algorithm 4.1.1: Finding the Maximum of Three Numbers

Input: a, b, c
Output: *large* (the largest of a, b, and c)

```
max3(a, b, c) {
   large = a
   // if b is larger than large, update large
   if (b > large)
      large = b
   // if c is larger than large, update large
   if (c > large)
      large = c
   return large
}
```

Algorithm 4.1.2: Finding the Maximum Value in a Sequence

Input: s, n

Output: *large* (the largest value in the sequence s)

```
max(s, n) {
  large = s_1
  for i = 2 to n
    if (s_i > large)
      large = s_i
  return large
}
```

Algorithm 4.2.1: Text Search

Input: p (indexed from 1 to m), m, t (indexed from 1 to n), n

Output: i

```
text_search(p, m, t, n) {
  for i = 1 to n − m + 1 {
    j = 1

    // i is the index in t of the first character of the
    // substring to compare with p, and j is the index in p

    // the while loop compares t_i ··· t_{i+m−1} and p_1 ··· p_m
    while (t_{i+j−1} == p_j) {
      j = j + 1
      if (j > m)
        return i
    }
  }
  return 0
}
```

2. 알고리즘분석 (Analysis of algorithms)

- *Complexity*: Amount of time and/or space needed to execute algorithm.
- Complexity depends on many factors: data representation type, kind of computer, computer language used, etc.

- **Alternative method: - count basic operation/program steps**

ex) function sum(a: Elementlist; n: integer): real;

1	var s: real; I: integer;	0
2	s:= 0	1
3.	for i:= 1 to n do	n+1
4.	s:= s+ a[i]	n
5.	sum:= s;	1

total steps: 2n+3

ex)
```
for i=1 to n do
    for j=1 to n do
        x=x+1
```

ex)
```
while (n≥1) do {
    for i=1 to n do
        x=x+1
    n= n/2 }
```

2.1 시간 복잡도 (Time complexity)

- Time complexity function, T(n): when n is size of problem, T(n) is the estimation of running time for the algorithm.

ex) If t(n) is, $60\,n^2+5n+1$

n	$t(n)=60\,n^2+5n+1$	$60\,n^2$
10	6051	6,000
1000	60,005,001	60,000,000
10000	6,000,050,001	6,000,000,000

- for large $60n^2$ is approx. equal to t(n)

- ignore constants, so t(n) grows like n^2 as n increases

- idea is to replace $t(n)=60n^2+5n+1$ with simpler expression as n^2

* 알고리즘의 정확한 수행시간보다도, 알고리즘의 수행시간이 input N 에 따라 얼마나 증가하는가에 관심이 있음.

2.2 Order notation (차수표기법)

Compare the time complexities of two programs that computes same function and to predict the growth in run time as the instance characteristics changes.

1) Definition: (Big-Oh), $f(n) = O(g(n))$; f(n) is AT MOST g(n)

- f of n is big oh of g of n, iff there exist positive constants C and n_0

$$\therefore\ |f(n)| \le C1|g(n)|,\ \forall n,\ n \ge n_0$$

g(n) 은 f(n) 의 상한선(upper bound).

ex) $3n+2 =$ ex) $100n+6=$ ex) $1000\, n^2+100n-6 =$

- THM: If f1 is O(g1) and f2 is O(g2) then

1) f1f2 is O(g1g2) 2) f1+f2 is O(max{g1,g2})

알고리즘의 난이도/복잡도 (Complexity Classes)

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

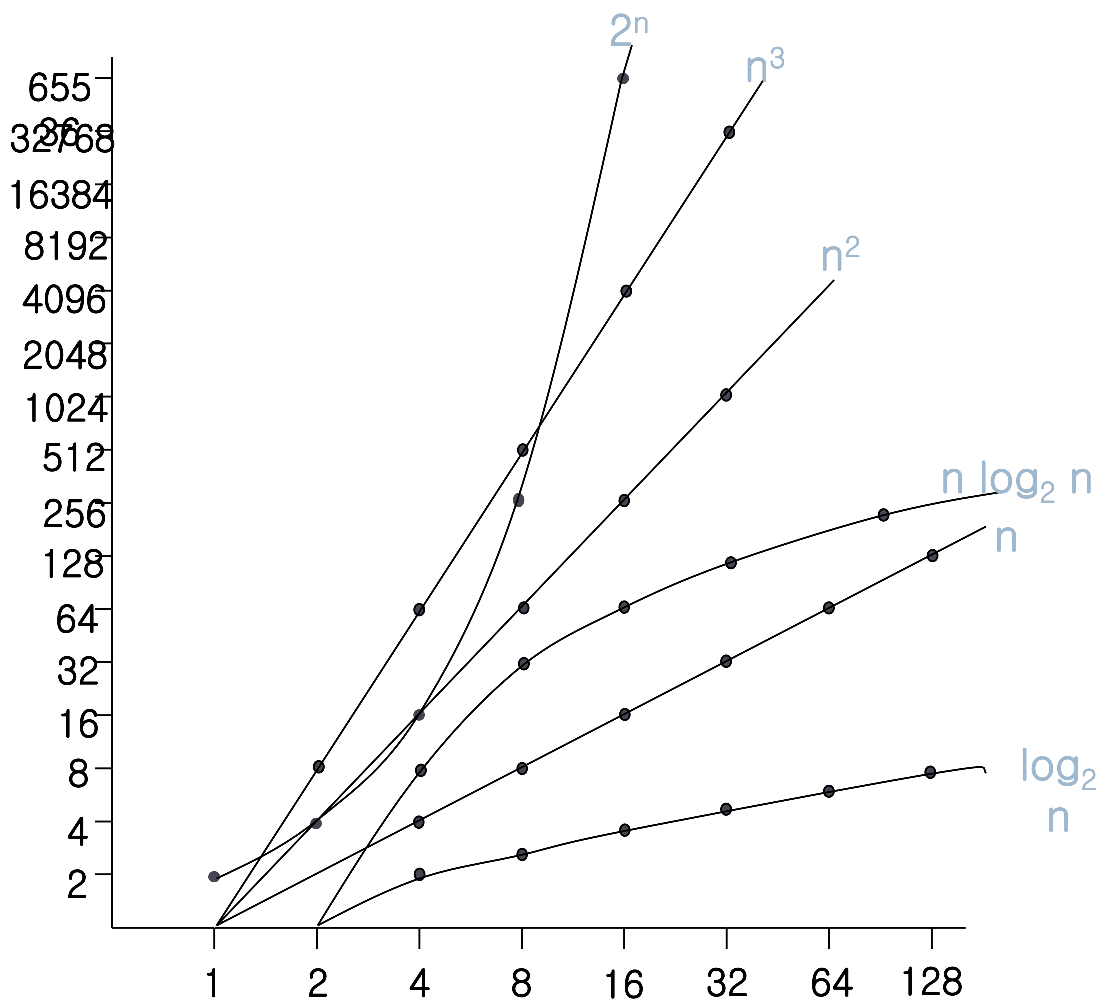

그림 10-1 입력 크기에 따른 각 함수의 증가 비율

Order notation (차수표기법) – con't

2) Definition: (OMEGA), $f(n) = \Omega(g(n))$; f(n) is AT LEAST g(n)

- f of n is omega of g of n, iff $\exists$ positive constants C and n_0

$\therefore$ **$|f(n)| \geq C1|g(n)|$, $\forall n, \; n \geq n_0$,** g(n) 은 f(n) 의 upper bound.

ex) $3n+2 =$ ex) $100n+6=$ ex) $1000\,n^2+100n-6 =$

3) **Definition: (THETA),** $f(n) = \Theta(g(n))$;

iff there exist positive constants C1,C2, and n_0 such that

$C_1 g(n) \leq f(n) \leq C_2 g(n)$, for all n, $n \geq n_0$

ex) $f(n) = 2n+ 3 \log n$ ex) $f(n) = 60n^2+5n+1$

ex) $f(n) = a_m n^m + \ldots\ldots + a_1 n + a_0$, and $a_m > 0$, then $f(n) = \Theta(n^m)$

Examples

ex) $\underline{\mathbf{1+2+\ldots+n}} \le n+n+\ldots n = n.n = n^2$ for all $n \ge 1$ (1)

=> for upper bound: $1+2+\ldots+n = O(n^2)$

=> for lower bound, $1+2+\ldots n \ge 1+1+,,,+1 = n \Rightarrow 1+2+\ldots n = \Omega(n)$

but we cannot deduce Θ-estimate since, the lower bound and upper bound are not equal.

$1+2+\ldots n \ge \lceil n/2 \rceil + \lceil n/2 \rceil + \ldots \lceil n/2 \rceil =$

$\lceil (n+1)/2 \rceil \lceil n/2 \rceil \ge (n/2)(n/2) = n^2/4$; half of (1)

- So we conclude that $1+2+..n= \Omega(n^2)$ and $1+2+..n=\Theta(n^2)$.

ex) j:= n

while j>=1 do

for I:= 1 to j do { x:= x+1; j:= $\lfloor j/2 \rfloor$}

end