Algorithms and Their Applications
- Asymptotic Notations -

Won-Yong Shin

March 23rd, 2020

연세대학교
YONSEI UNIVERSITY

- Order of growth
  - A way of comparing functions that **ignores** *constant factors and small input sizes*
  - **$O(g(n))$**: class of functions $f(n)$ that grow <u>no faster</u> than $g(n)$
  - **$\Theta(g(n))$**: class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$
  - **$\Omega(g(n))$**: class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

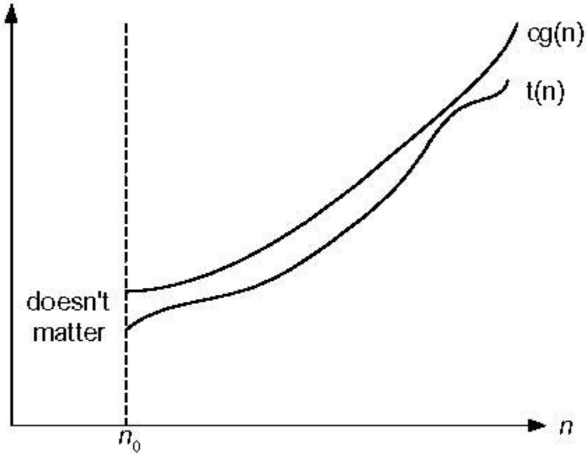


Figure 2.1 Big-oh notation: $\boxed{t(n) \in O(g(n))}$

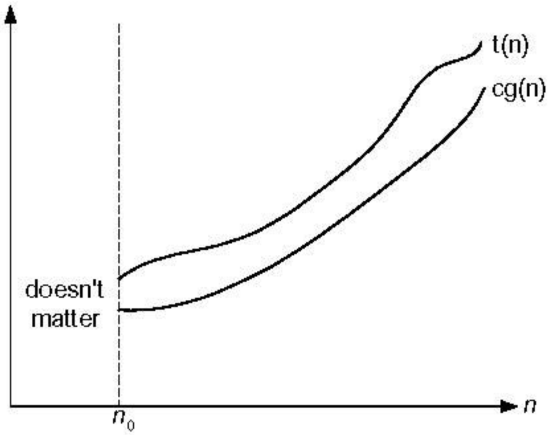


Fig. 2.2 Big-omega notation: $\boxed{t(n) \in \Omega(g(n))}$

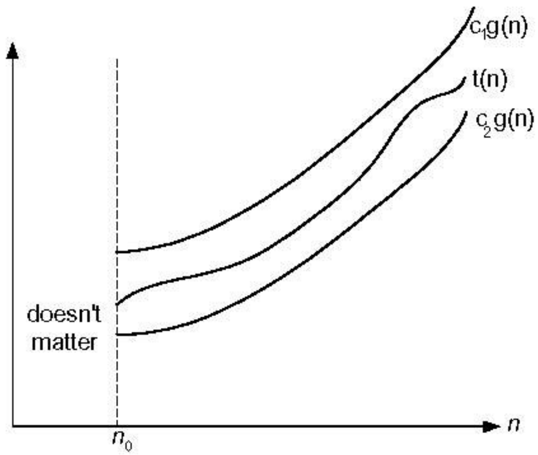


Figure 2.3 Big-theta notation: $\boxed{t(n) \in \Theta(g(n))}$

Establishing Order of Growth Using the Definition

- Definition
  - **$f(n)$ is in $O(g(n))$** if order of growth of $f(n) \le$ order of growth of $g(n)$ (within constant multiple), i.e., there exist positive constant c and non-negative integer n_0 such that

$$f(n) \le c\, g(n) \text{ for every } n \ge n_0$$

  - **$f(n)$ is in $\Omega(g(n))$** if $f(n) \ge c\, g(n)$ for every $n \ge n_0$
  - **$f(n)$ is in $\Theta(g(n))$** if $c_2\, g(n) \le f(n) \le c_1\, g(n)$ for every $n \ge n_0$

- Examples:
  - $10n$ is $O(n^2)$
  - $5n+20$ is $O(n)$
  - $2n^3$ is $\Omega(n^2)$
  - $0.5n(n-1)$ is $\Theta(n^2)$

Some Properties of Asymptotic Order of Growth

- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$

- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Establishing Order of Growth Using Limits

- Limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

- Examples
  - $10n$ vs. n^2
  - $n(n+1)/2$ vs. n^2
  - $\log_2 n$ vs. n

Order of Growth of Some Important Functions

- **Logarithmic functions**
  - All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- **Polynomials**
  - All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$
- **Exponential functions**
  - Exponential functions a^n have *different* orders of growth for *different* a's
- Order
  - $\log n$ < order n^{α} $(\alpha>0)$ < order a^n < order $n!$ < order n^n

Nonrecursive Algorithms

- General plan for analysis
  - 1. Decide on parameter *n* indicating *input size*
  - 2. Identify algorithm's *basic operation*
  - 3. Determine *worst*, *average*, and *best* cases for input of size *n*
  - 4. Set up a sum for the number of times the basic operation is executed
  - 5. Simplify the sum using standard formulas and rules

- $\Sigma_{l\le i\le n} 1 = 1+1+\cdots+1 = n - l + 1$

  In particular, $\Sigma_{1\le i\le n} 1 = n - 1 + 1 = n \in \boldsymbol{\Theta(n)}$

- $\Sigma_{1\le i\le n}\, i = 1+2+\cdots+n = n(n+1)/2 \approx n^2/2 \in \boldsymbol{\Theta(n^2)}$

- $\Sigma_{1\le i\le n}\, i^2 = 1^2+2^2+\cdots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \boldsymbol{\Theta(n^3)}$

- $\Sigma_{0\le i\le n}\, a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

  In particular, $\Sigma_{0\le i\le n}\, 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \boldsymbol{\Theta(2^n)}$

ALGORITHM *MaxElement*$(A[0..n-1])$
//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return *maxval*

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \boxed{\Theta(n)}$$

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//        and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$

$$= \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \boxed{\Theta(n^2)}$$

Example 3: Matrix Multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1$$

$$= \boxed{n^3}$$

Example 4: Counting Binary Digits

ALGORITHM *Binary*(n)
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
  $count \leftarrow count + 1$
  $n \leftarrow \lfloor n/2 \rfloor$
return $count$

$$\lfloor \log_2 n \rfloor + 1 \approx \boxed{\log_2 n}$$

Recursive Algorithms

Plan for Analysis of Recursive Algorithms

- Recursive algorithms
  - Decide on a parameter indicating an *input's size*
  - Identify the algorithm's basic operation
  - Check whether the number of times the basic operation is executed may <u>vary on different inputs of the same size</u> (If it may, the worst, average, and best cases must be investigated separately)
  - Set up a *recurrence* relation with an appropriate initial condition expressing the number of times the basic operation is executed
  - Solve the recurrence (or, at the very least, establish its solution's order of growth) by *backward substitutions* or another method

- Definition
  - $n! = 1 \cdot 2 \cdot \ldots \cdot (n\text{-}1) \cdot n$ for $n \geq 1$ and $0! = 1$

- Recursive definition of n!
  - **$F(n) = F(n\text{-}1) \cdot n$** for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

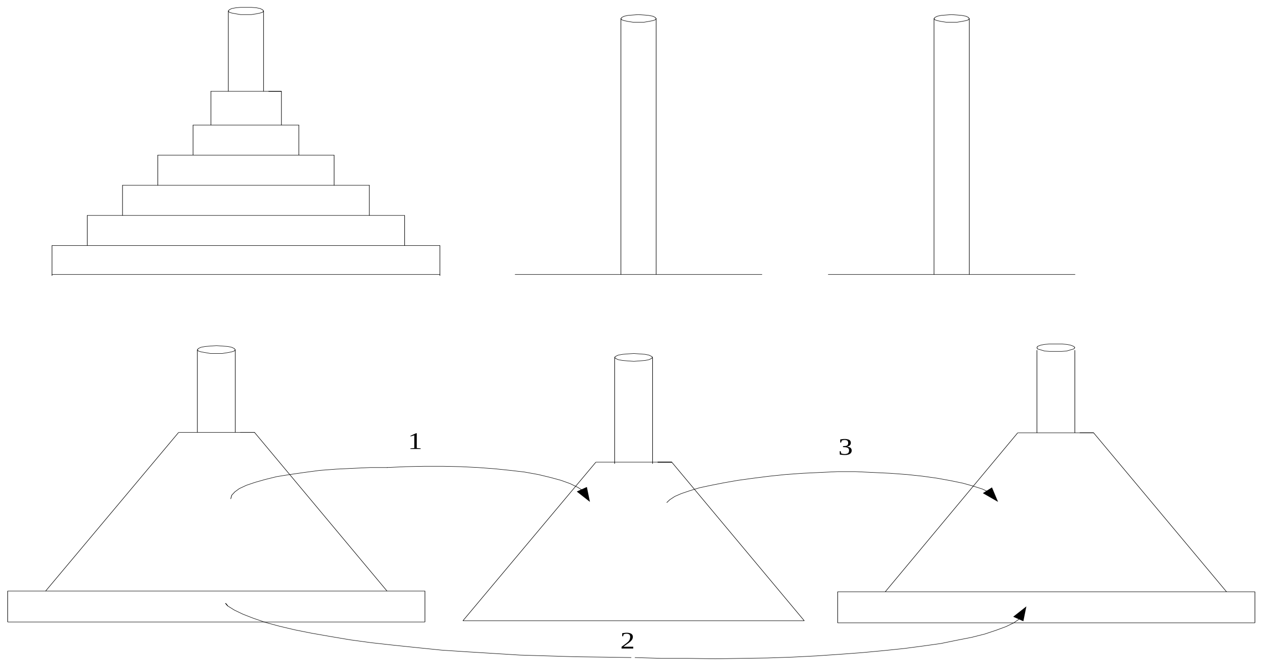
```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) * n
```

- Size: n
- Basic operation: multiplication
- Recursive relation:

$M(n) = M(n\text{-}1) + 1, \ M(0) = 0$

$M(n)$: # of multiplications

Example 2: The Tower of Hanoi Puzzle

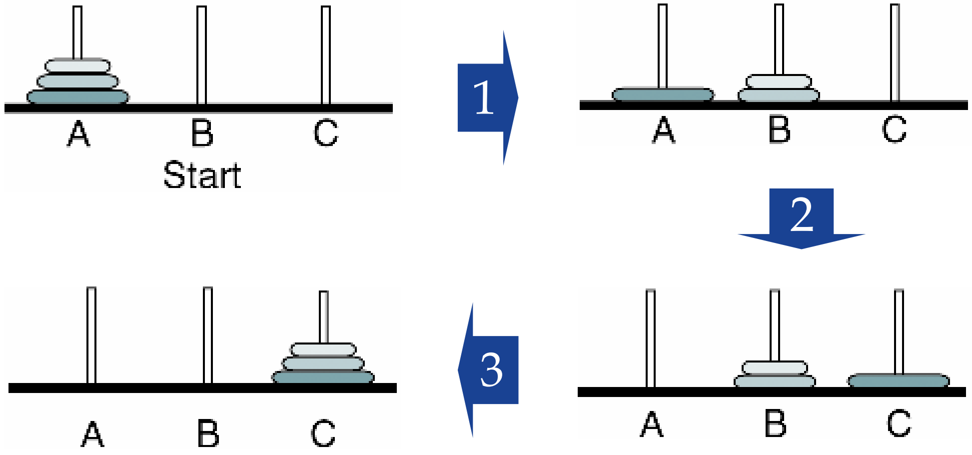


- Recurrence for number of moves:

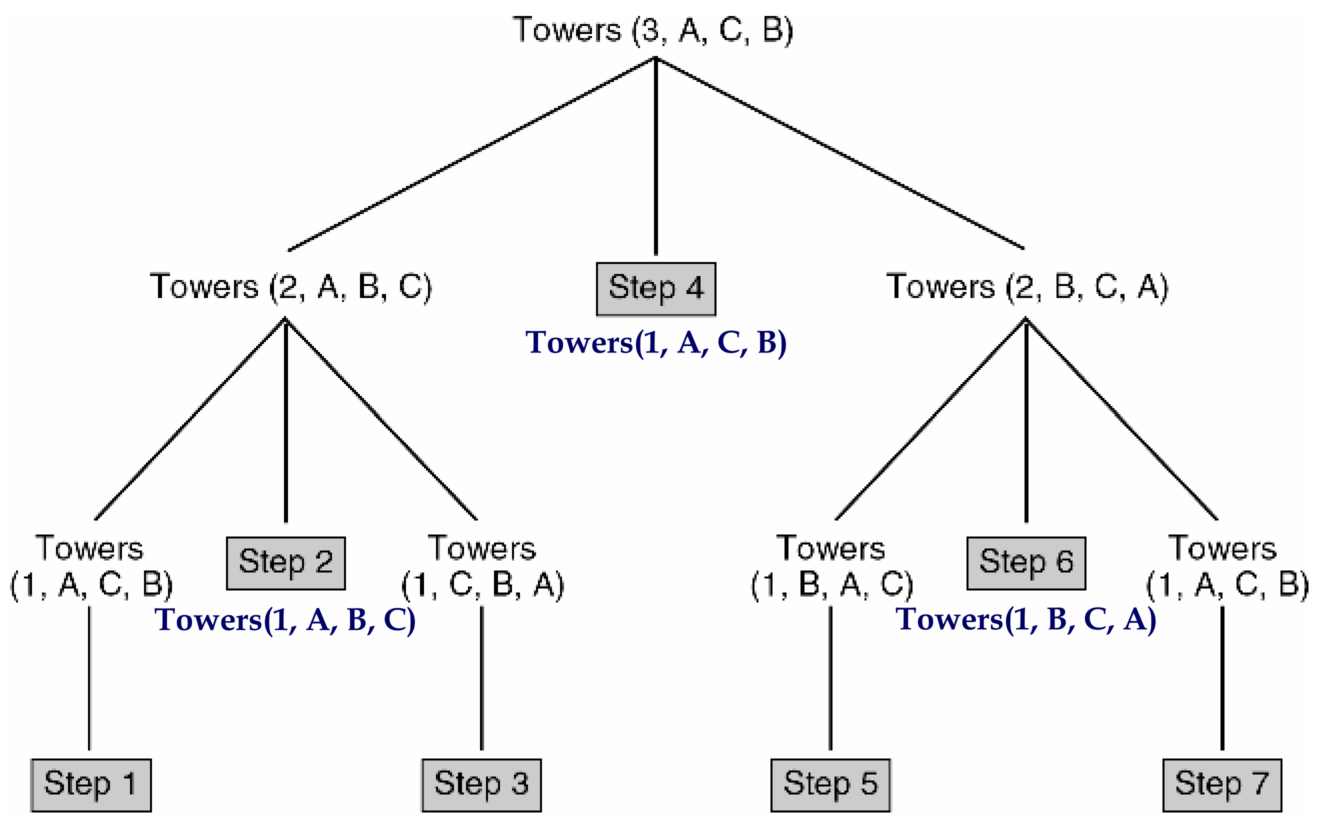
$$\boxed{M(n) = 2M(n\text{-}1) + 1, \;\; M(1) = 1}$$

- **Moving n disks**
  1. move n -1 disks from source to auxiliary
  2. move 1 disk from source to destination → **Base case**
  3. move n -1 disks from auxiliary to destination

Ex) n = 3



The Towers of Hanoi – Moving 3 Disks



ALGORITHM *BinRec(n)*
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

- Basic operation: **additions**
- Recursive relation:

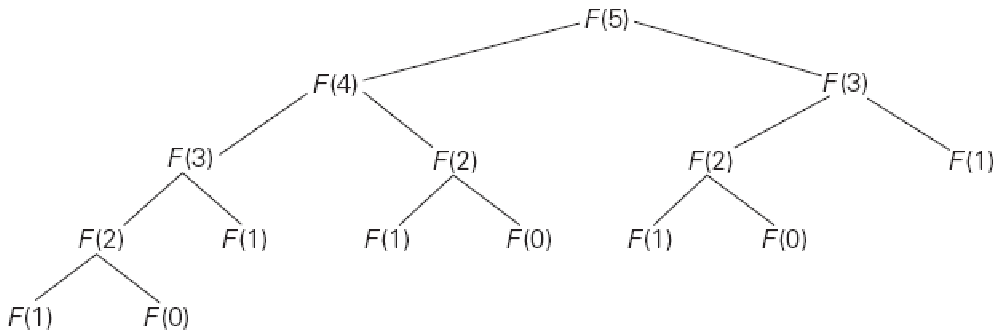
$$\boxed{\begin{array}{l} A(n) = A(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1. \\ A(1) = 0. \end{array}}$$

- Solving steps
  - **Step 1**: $n = 2^k$
  - **Step 2**: $A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$
    $A(2^0) = 0.$
  - **Step 3**: $A(2^k) = A(2^{k-1}) + 1 \quad \cdots \quad = A(2^{k-k}) + k$

$\Rightarrow$ $A(n) = \log_2 n \in \Theta(\log n).$

- The Fibonacci numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
- The Fibonacci recurrence:

$$\boxed{\begin{array}{l} F(n) = F(n-1) + F(n-2) \\ F(0) = 0,\ \ F(1) = 1 \end{array}}$$



- General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

Solving $aX(n) + bX(n-1) + cX(n-2) = 0$

- Solve to obtain roots r_1 and r_2

- General solution to the recurrence
  - If r_1 and r_2 are two distinct real roots: $\mathbf{X(n) = \alpha r_1^{\,n} + \beta r_2^{\,n}}$
  - If $r_1 = r_2 = r$ are two equal real roots: $\mathbf{X(n) = \alpha r^n + \beta n r^n}$

- Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

- *Particular* solution can be found by using <u>initial conditions</u>

- Application to the Fibonacci numbers
  - $\mathbf{F(n) = F(n-1) + F(n-2)}$ or $\mathbf{F(n) - F(n-1) - F(n-2) = 0}$
  - Characteristic equation
  - Roots of the characteristic equation
  - General solution to the recurrence
  - Particular solution for $F(0) = 0$, $F(1) = 1$

- Basic operation
  - Additions performed by the algorithm in computing $F(n)$
- Recursive relation:

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

- Solving steps
  - **Step 1**: substituting $B(n) = A(n) + 1$
  - **Step 2**: $B(n) - B(n-1) - B(n-2) = 0,$

    $B(0) = 1, \qquad B(1) = 1.$

$$\Rightarrow \quad A(n) = F(n+1) - 1$$