

Analysis and Design of Algorithms

CMP 2202
Lecture 2

Analysis of algorithms

- ✓ **Issues:**

- ✓ correctness
- ✓ time efficiency
- ✓ space efficiency
- ✓ optimality

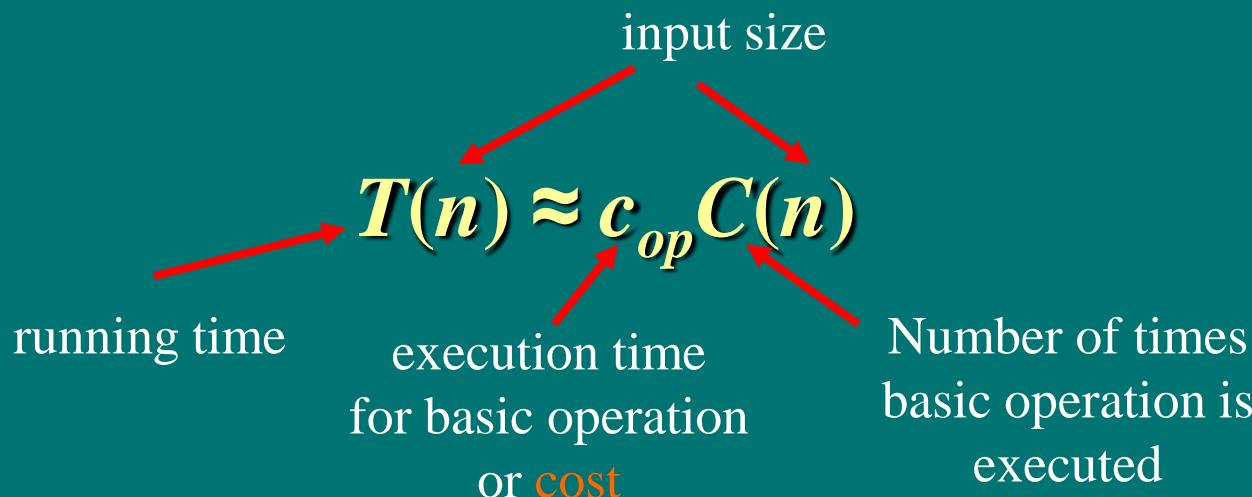
- ✓ **Approaches:**

- ✓ theoretical analysis
- ✓ empirical analysis

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size

- Q Basic operation: the operation that contributes the most towards the running time of the algorithm



Note: Different basic operations may cost differently!

Input size and basic operation examples

<i>Problem</i>	<i>Input size measure</i>	<i>Basic operation</i>
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n 'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Empirical analysis of time efficiency

- ❑ Select a specific (typical) sample of inputs
- ❑ Use physical unit of time (e.g., milliseconds)
 - or
 - Count actual number of basic operation's executions
- ❑ Analyze the empirical data

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- ❑ Worst case: $C_{\text{worst}}(n)$ – maximum over inputs of size n
- ❑ Best case: $C_{\text{best}}(n)$ – minimum over inputs of size n
- ❑ Average case: $C_{\text{avg}}(n)$ – “average” over inputs of size n
 - Number of times the basic operation will be executed on typical input
 - NOT the average of worst and best case
 - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs. So, avg = expected under uniform distribution.

Example: Sequential search

ALGORITHM *SequentialSearch($A[0..n - 1]$, K)*

//Searches for a given value in a given array by sequential search
//Input: An array $A[0..n - 1]$ and a search key K

//Output: The index of the first element of A that matches K

// or -1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

❑ **Worst case**

n key comparisons

❑ **Best case**

1 comparisons

❑ **Average case**

$(n+1)/2$, assuming K is in A

Types of formulas for basic operation's count

❑ Exact formula

e.g., $C(n) = n(n-1)/2$

❑ Formula indicating order of growth with specific multiplicative constant

e.g., $C(n) \approx 0.5 n^2$

❑ Formula indicating order of growth with unknown multiplicative constant

e.g., $C(n) \approx cn^2$

Order of growth

- ❑ **Most important: Order of growth within a constant multiple as $n \rightarrow \infty$**
- ❑ **Example:**
 - How much faster will algorithm run on computer that is twice as fast?
 - How much longer does it take to solve problem of double input size?

Values of some important functions as $n \rightarrow \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

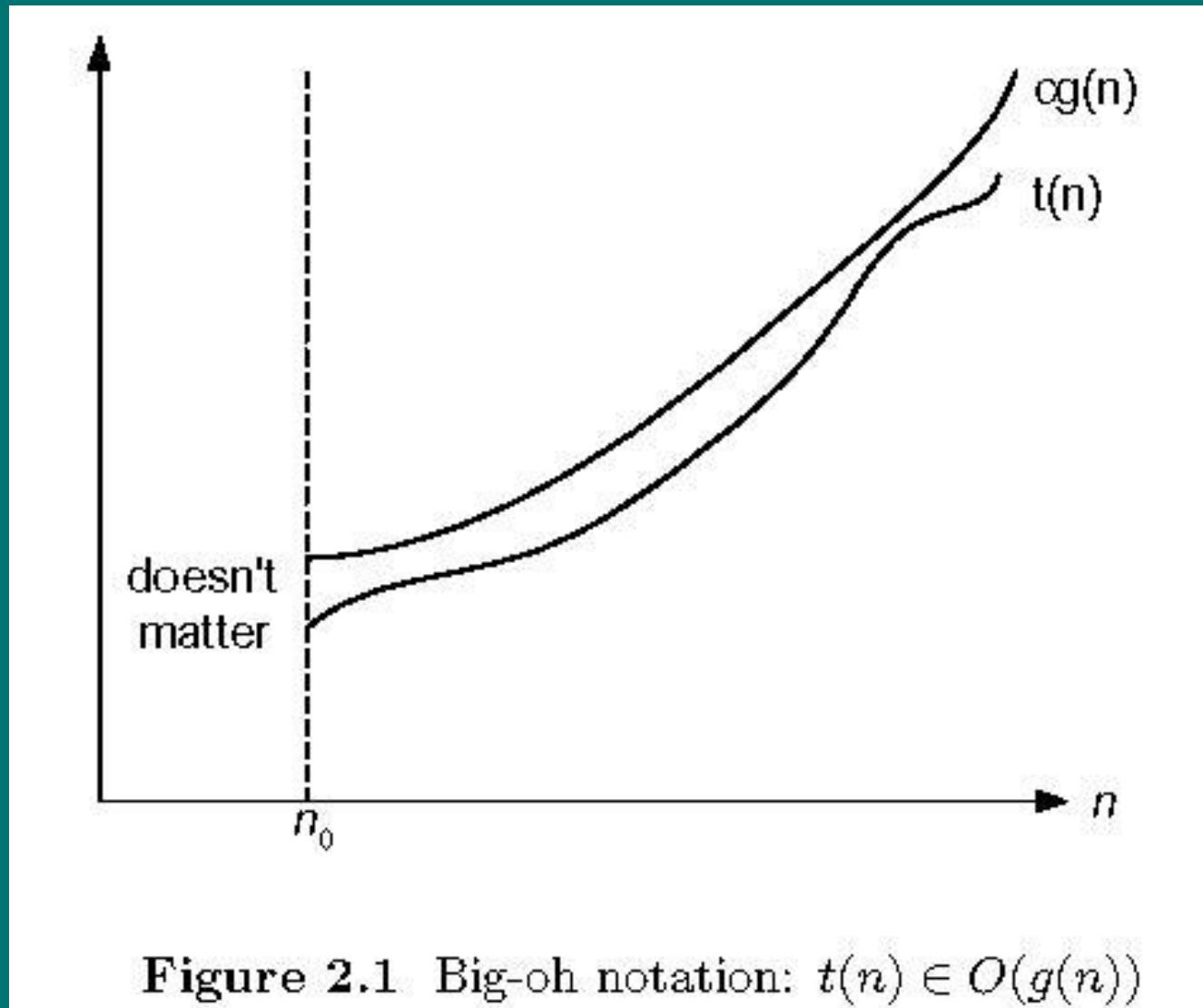
Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Asymptotic order of growth

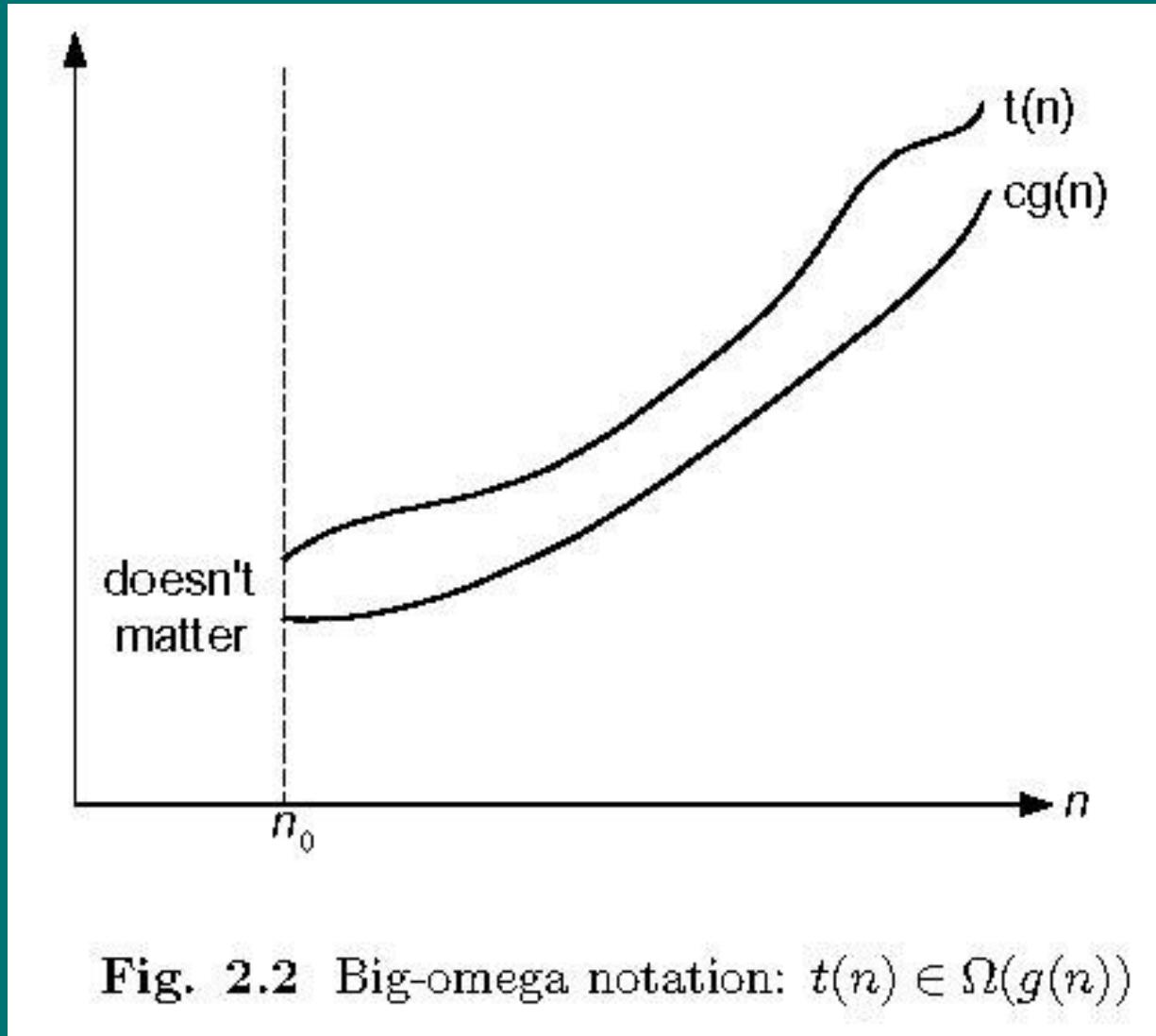
A way of comparing functions that ignore constant factors and small input sizes

- ❑ $O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$
- ❑ $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$
- ❑ $\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$

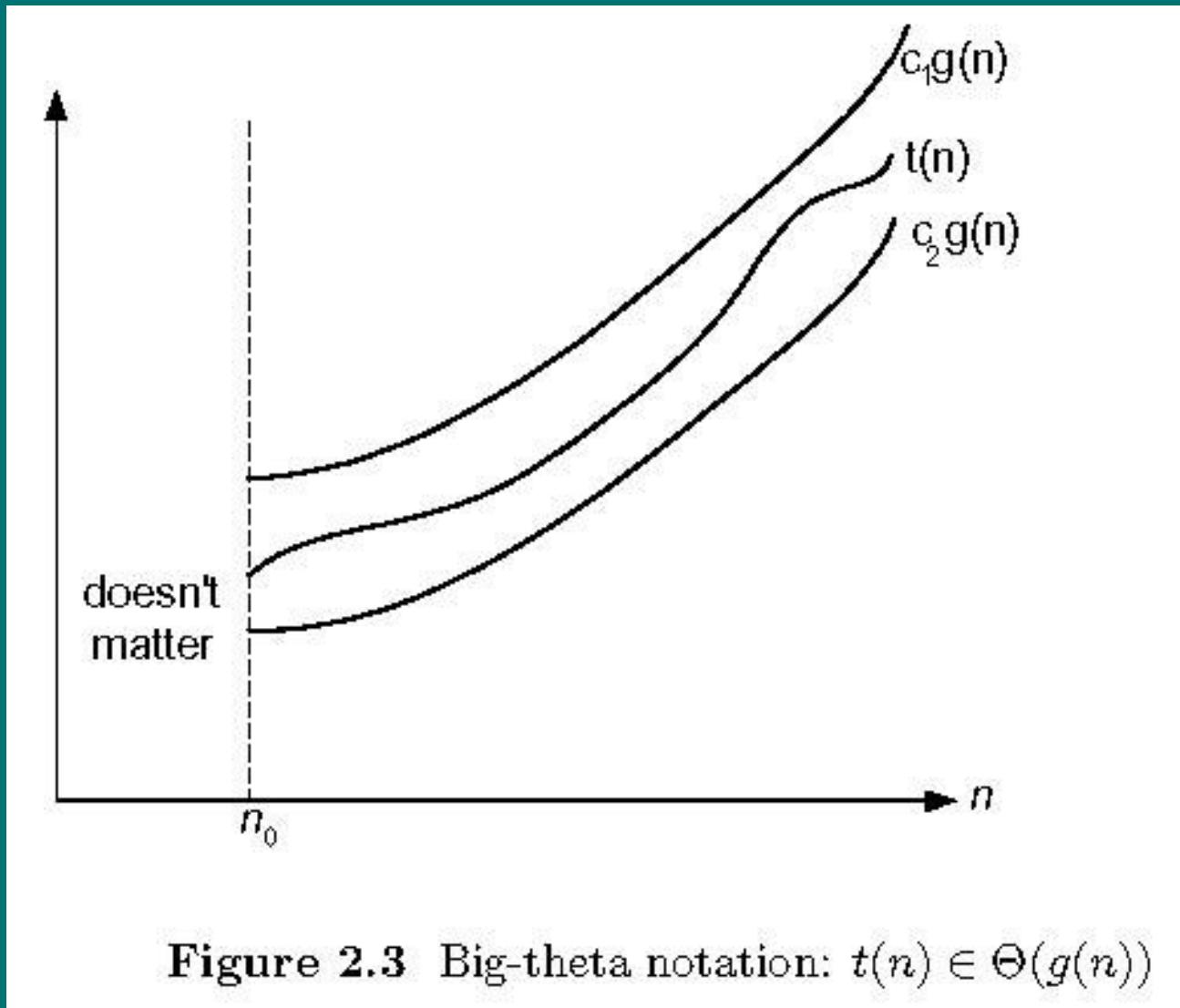
Big-oh



Big-omega



Big-theta



Establishing order of growth using the definition

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

$$f(n) \leq c g(n) \text{ for every } n \geq n_0$$

Examples:

❑ $10n$ is in $O(n^2)$

❑ $5n+20$ is in $O(n)$

Ω -notation

§ Formal definition

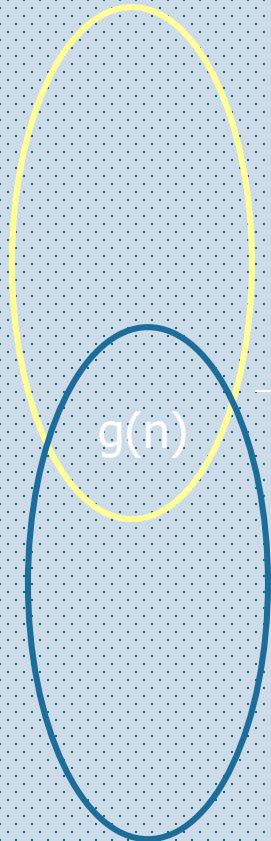
- A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some constant multiple of $g(n)$ for all large n , i.e., if there exist some positive constant c and some nonnegative integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

Θ -notation

Q Formal definition

- A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, if $t(n)$ is bounded both above and below by some positive constant multiples of $g(n)$ for all large n , i.e., if there exist some positive constant c_1 and c_2 and some nonnegative integer n_0 such that
 $c_2 g(n) \leq t(n) \leq c_1 g(n)$ for all $n \geq n_0$



\geq

$\Omega(g(n))$, functions that grow at least as fast as $g(n)$

$=$

$\Theta(g(n))$, functions that grow at the same rate as $g(n)$

\leq

$O(g(n))$, functions that grow no faster than $g(n)$

Theorem

- ∅ If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then
 $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$.
 - The analogous assertions are true for the Ω -notation and Θ -notation.
- ∅ Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.
 - For example, $5n^2 + 3n\log n \in O(n^2)$

Proof. There exist constants c_1, c_2, n_1, n_2 such that

$$t_1(n) \leq c_1 * g_1(n), \text{ for all } n \geq n_1$$

$$t_2(n) \leq c_2 * g_2(n), \text{ for all } n \geq n_2$$

Define $c_3 = c_1 + c_2$ and $n_3 = \max\{n_1, n_2\}$. Then

$$t_1(n) + t_2(n) \leq c_3 * \max\{g_1(n), g_2(n)\}, \text{ for all } n \geq n_3$$

Some properties of asymptotic order of growth

② $f(n) \in O(f(n))$

③ $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$

④ If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

Note similarity with $a \leq b$

⑤ If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

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

Also, $\sum_{1 \leq i \leq n} \Theta(f(i)) = \Theta(\sum_{1 \leq i \leq n} f(i))$

Establishing order of growth using limits

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

Examples:

• $10n$ vs. n^2

• $n(n+1)/2$ vs. n^2

L'Hôpital's rule and Stirling's formula

L'Hôpital's rule: If $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$ and the derivatives f' , g' exist, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

Example: $\log n$ vs. n

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Example: 2^n vs. $n!$

Orders of growth of some important 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

because

$$\log_a n = \log_b n / \log_b a$$

- All polynomials of the same degree k belong to the same class:

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 \in \Theta(n^k)$$

- Exponential functions a^n have different orders of growth for different a 's
- order $\log n < \text{order } n^\alpha (\alpha > 0) < \text{order } a^n < \text{order } n! < \text{order } n^n$

Basic asymptotic efficiency classes

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	$n\text{-log-}n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Time efficiency of nonrecursive algorithms

General Plan for Analysis

- ✓ Decide on parameter n indicating input size
- ✓ Identify algorithm's basic operation
- ✓ Determine worst, average, and best cases for input of size n
- ✓ Set up a sum for the number of times the basic operation is executed
- ✓ Simplify the sum using standard formulas and rules (see Appendix A)

Useful summation formulas and rules

$$\sum_{l \leq i \leq n} 1 = 1+1+\dots+1 = n - l + 1$$

In particular, $\sum_{l \leq i \leq n} 1 = n - 1 + 1 = n \in \Theta(n)$

$$\sum_{1 \leq i \leq n} i = 1+2+\dots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$$

$$\sum_{1 \leq i \leq n} i^2 = 1^2+2^2+\dots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$$

$$\sum_{0 \leq i \leq n} a^i = 1 + a + \dots + a^n = (a^{n+1} - 1)/(a - 1) \text{ for any } a \neq 1$$

In particular, $\sum_{0 \leq i \leq n} 2^i = 2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

Example 1: Maximum element

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
```

$$T(n) = \sum_{1 \leq i \leq n-1} 1 = n-1 = \Theta(n) \text{ comparisons}$$

Example 2: Element uniqueness problem

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  $i \leftarrow 0$  to  $n - 2$  do  
    for  $j \leftarrow i + 1$  to  $n - 1$  do  
        if  $A[i] = A[j]$  return false  
return true
```

$$T(n) = \sum_{0 \leq i \leq n-2} (\sum_{i+1 \leq j \leq n-1} 1)$$

$$= \sum_{0 \leq i \leq n-2} n-i-1 = (n-1+1)(n-1)/2$$

$= \Theta(n^2)$ comparisons

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  $i \leftarrow 0$  to  $n - 1$  do
        for  $j \leftarrow 0$  to  $n - 1$  do
             $C[i, j] \leftarrow 0.0$ 
            for  $k \leftarrow 0$  to  $n - 1$  do
                 $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$ 
    return  $C$ 
```

$$\begin{aligned} T(n) &= \sum_{0 \leq i \leq n-1} \sum_{0 \leq j \leq n-1} n \\ &= \sum_{0 \leq i \leq n-1} \Theta(n^2) \\ &= \Theta(n^3) \text{ multiplications} \end{aligned}$$

Example 4: Gaussian elimination

Algorithm *GaussianElimination(A[0..n-1,0..n])*

//Implements Gaussian elimination on an n -by- $(n+1)$ matrix A

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

for $k \leftarrow i$ **to** n **do**

$A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement.

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

$B \leftarrow A[j,i] / A[i,i]$

for $k \leftarrow i$ **to** n **do**

$A[j,k] \leftarrow A[j,k] - A[i,k] * B$

Example 5: Counting binary digits

ALGORITHM *Binary(n)*

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

It cannot be investigated the way the previous examples are.

The halving game: Find integer i such that $n/2^i \leq 1$.

Answer: $i \leq \log n$. So, $T(n) = \Theta(\log n)$ divisions.

Another solution: Using recurrence relations.

Plan for Analysis of 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 op. is executed may vary on different inputs of the same size. (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 op. is executed.
- ❑ Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 * 2 * \dots * (n-1) * n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n-1) * n$ 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

Recurrence relation:

$M(n) = M(n-1) + 1$

$M(0) = 0$

Solving the recurrence for $M(n)$

$$M(n) = M(n-1) + 1, \quad M(0) = 0$$

$$M(n) = M(n-1) + 1$$

$$= (M(n-2) + 1) + 1 = M(n-2) + 2$$

$$= (M(n-3) + 1) + 2 = M(n-3) + 3$$

...

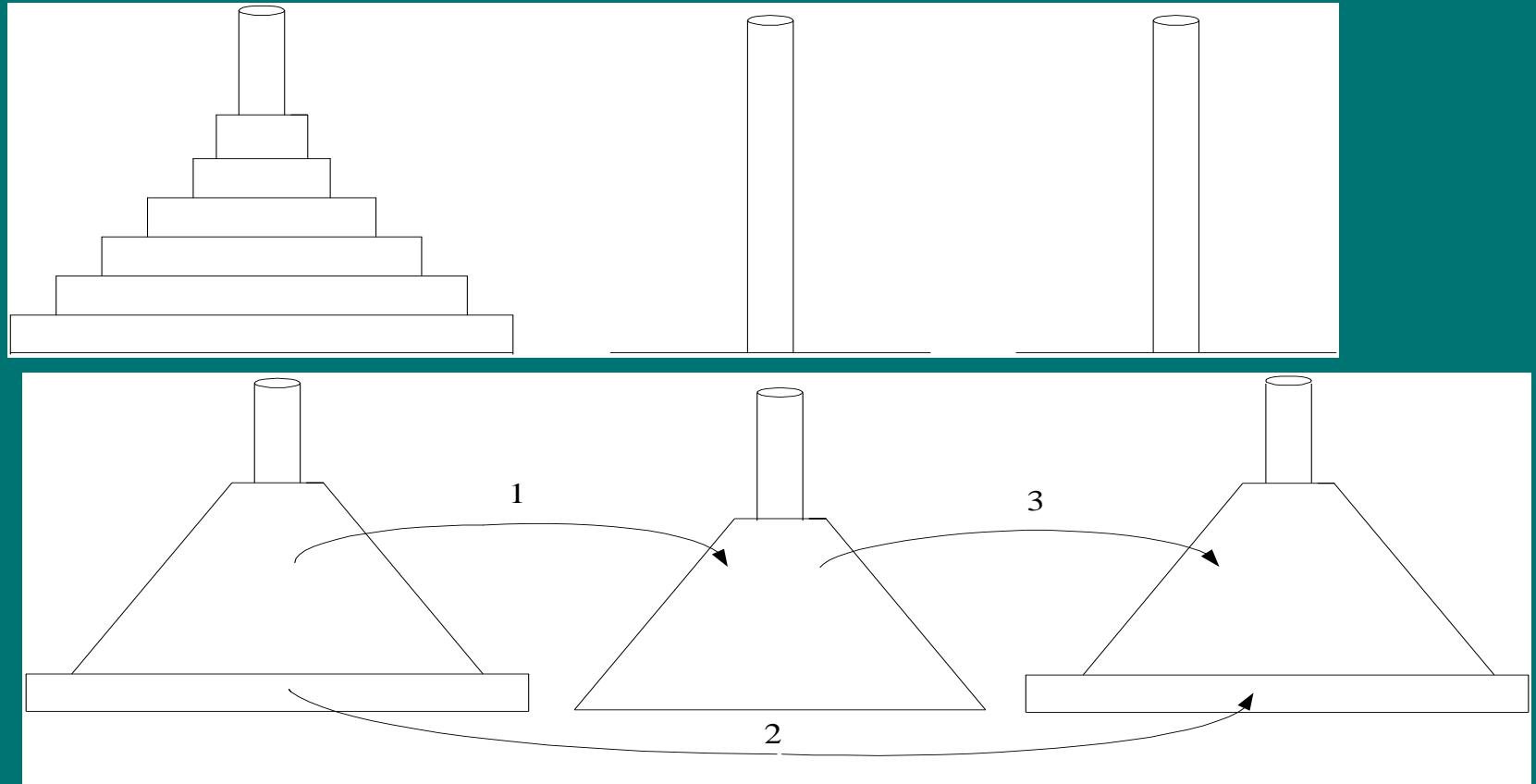
$$= M(n-i) + i$$

$$= M(0) + n$$

$$= n$$

The method is called **backward substitution**.

Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

$$M(n) = 2M(n-1) + 1$$

Solving recurrence for number of moves

$$M(n) = 2M(n-1) + 1, \quad M(1) = 1$$

$$M(n) = 2M(n-1) + 1$$

$$= 2(2M(n-2) + 1) + 1 = 2^2 * M(n-2) + 2^1 + 2^0$$

$$= 2^2 * (2M(n-3) + 1) + 2^1 + 2^0$$

$$= 2^3 * M(n-3) + 2^2 + 2^1 + 2^0$$

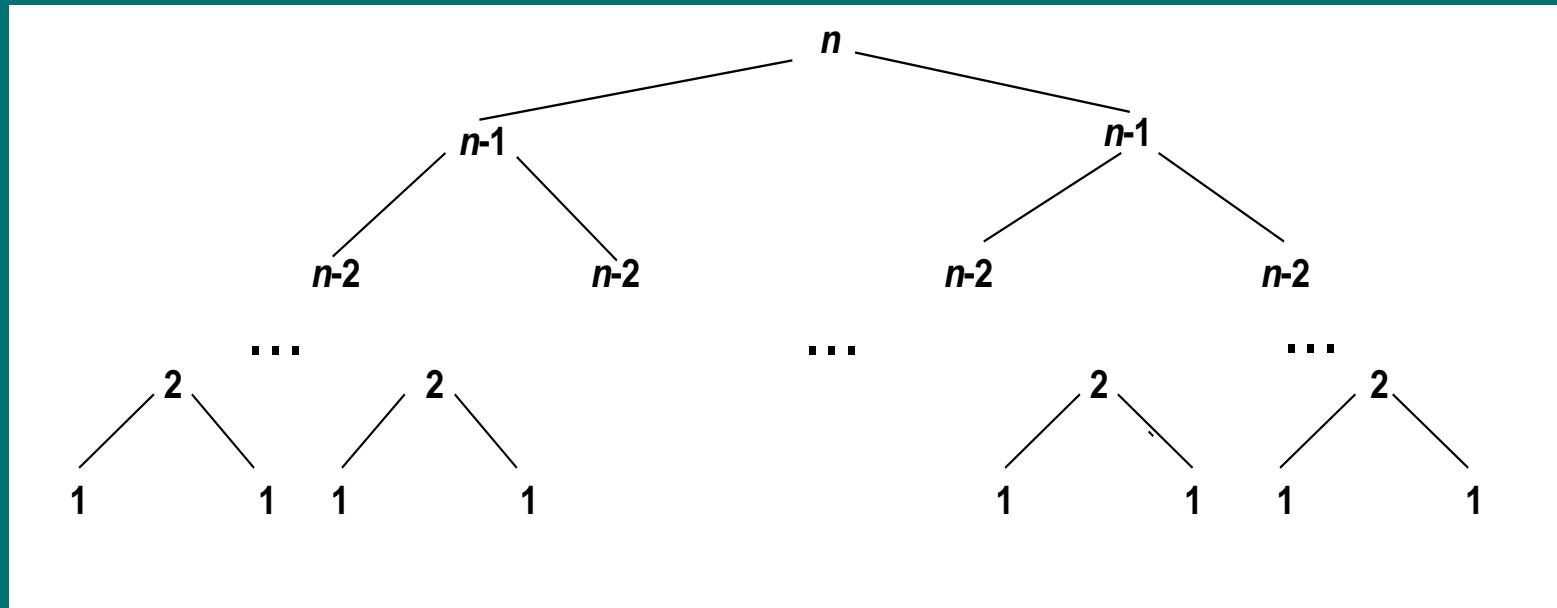
= ...

$$= 2^{n-1} * M(1) + 2^{n-2} + \dots + 2^1 + 2^0$$

$$= 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0$$

$$= 2^n - 1$$

Tree of calls for the Tower of Hanoi Puzzle



Example 3: Counting #bits

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
```

$$A(n) = A(\lfloor n/2 \rfloor) + 1, \quad A(1) = 0$$

$$A(2^k) = A(2^{k-1}) + 1, \quad A(2^0) = 1 \quad (\text{using the Smoothness Rule})$$

$$= (A(2^{k-2}) + 1) + 1 = A(2^{k-2}) + 2$$

$$= A(2^{k-i}) + i$$

$$= A(2^{k-k}) + k = k + 0$$

$$= \log_2 n$$

Smoothness Rule

Q Let $f(n)$ be a nonnegative function defined on the set of natural numbers. $f(n)$ is call smooth if it is eventually nondecreasing and

$$f(2n) \in \Theta(f(n))$$

- Functions that do not grow too fast, including $\log n$, n , $n\log n$, and n^α where $\alpha >= 0$ are smooth.

Q Smoothness rule

Let $T(n)$ be an eventually nondecreasing function and $f(n)$ be a smooth function. If

$T(n) \in \Theta(f(n))$ for values of n that are powers of b , where $b >= 2$, then

$$T(n) \in \Theta(f(n)) \text{ for any } n.$$

Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

The Fibonacci recurrence:

$$F(n) = F(n-1) + F(n-2)$$

$$F(0) = 0$$

$$F(1) = 1$$

General 2nd order linear homogeneous recurrence with constant coefficients:

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

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

❑ Set up the characteristic equation (quadratic)

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

❑ 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: $X(n) = \alpha r_1^n + \beta r_2^n$

if $r_1 = r_2 = r$ are two equal real roots: $X(n) = \alpha r^n + \beta n r^n$

❑ Particular solution can be found by using initial conditions

Application to the Fibonacci numbers

$$F(n) = F(n-1) + F(n-2) \text{ or } F(n) - F(n-1) - F(n-2) = 0$$

Characteristic equation:

$$r^2 - r - 1 = 0$$

Roots of the characteristic equation:

$$r_{1,2} = (1 \pm \sqrt{5})/2$$

General solution to the recurrence:

$$\alpha \cdot r_1^n + \beta \cdot r_2^n$$

Particular solution for $F(0)=0, F(1)=1$:

$$\alpha + \beta = 0$$

$$\alpha \cdot r_1 + \beta \cdot r_2 = 1$$

Computing Fibonacci numbers

1. Definition-based recursive algorithm
2. Nonrecursive definition-based algorithm
3. Explicit formula algorithm
4. Logarithmic algorithm based on formula:

$$\begin{pmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

for $n \geq 1$, assuming an efficient way of computing matrix powers.

Important Recurrence Types

Decrease-by-one recurrences

- A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size n and a smaller size $n - 1$.
- Example: $n!$
- The recurrence equation for investigating the time efficiency of such algorithms typically has the form

$$T(n) = T(n-1) + f(n)$$

Decrease-by-a-constant-factor recurrences

- A decrease-by-a-constant-factor algorithm solves a problem by dividing its given instance of size n into several smaller instances of size n/b , solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.
- Example: binary search.
- The recurrence equation for investigating the time efficiency of such algorithms typically has the form

$$T(n) = aT(n/b) + f(n)$$

Decrease-by-one Recurrences

Q One (constant) operation reduces problem size by one.

$$T(n) = T(n-1) + c \quad T(1) = d$$

Solution:

$$T(n) = (n-1)c + d \quad \text{linear}$$

Q A pass through input reduces problem size by one.

$$T(n) = T(n-1) + c \quad n \quad T(1) = d$$

Solution:

$$T(n) = [n(n+1)/2 - 1] c + d \quad \text{quadratic}$$

Decrease-by-a-constant-factor recurrences – The Master Theorem

$$T(n) = aT(n/b) + f(n), \quad \text{where } f(n) \in \Theta(n^k), k \geq 0$$

1. $a < b^k$ $T(n) \in \Theta(n^k)$
2. $a = b^k$ $T(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $T(n) \in \Theta(n^{\log_b a})$

Ω Examples:

- $T(n) = T(n/2) + 1$ $\Theta(\log n)$
- $T(n) = 2T(n/2) + n$ $\Theta(n \log n)$
- $T(n) = 3T(n/2) + n$ $\Theta(n^{\log_2 3})$
- $T(n) = T(n/2) + n$ $\Theta(n)$