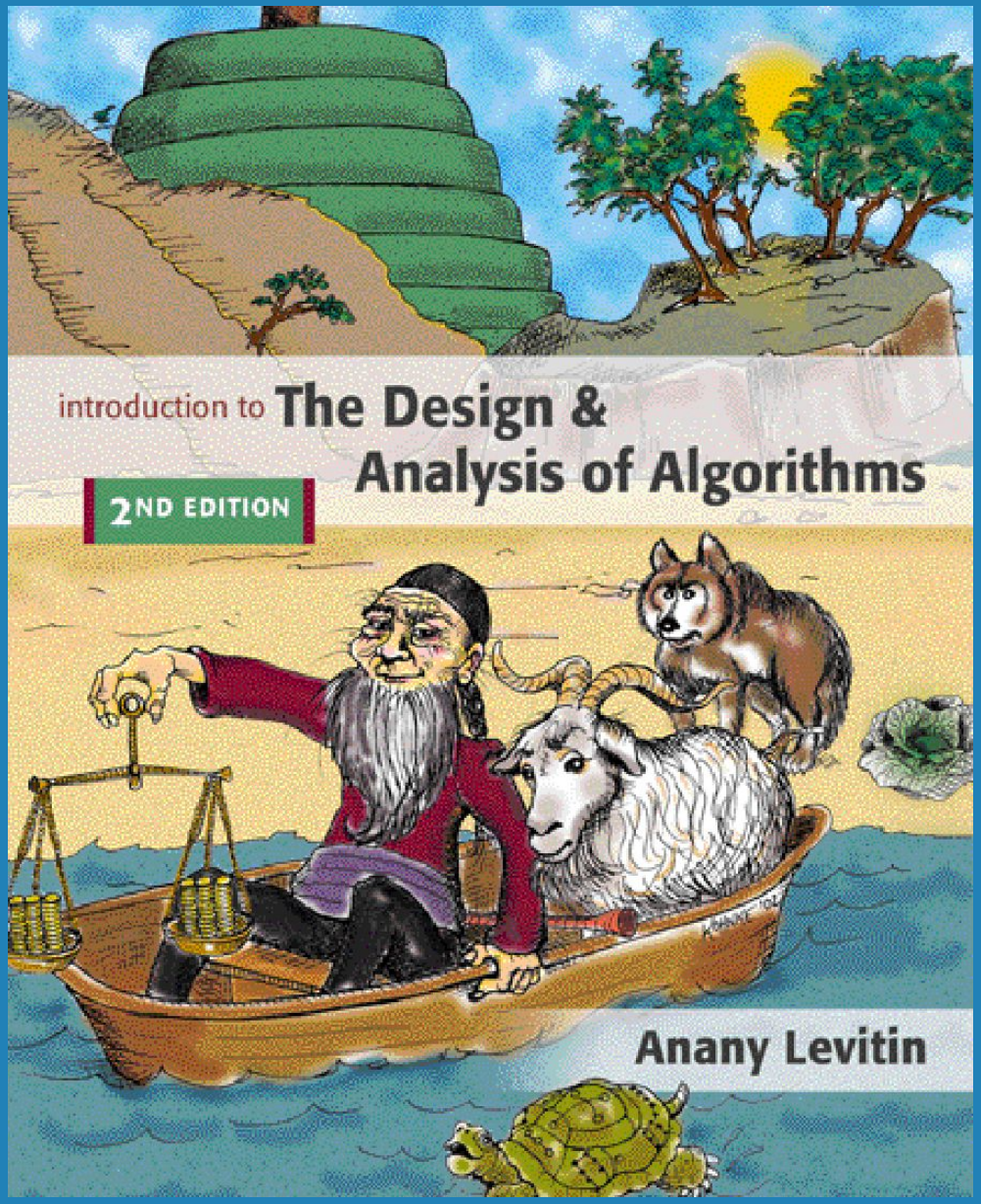
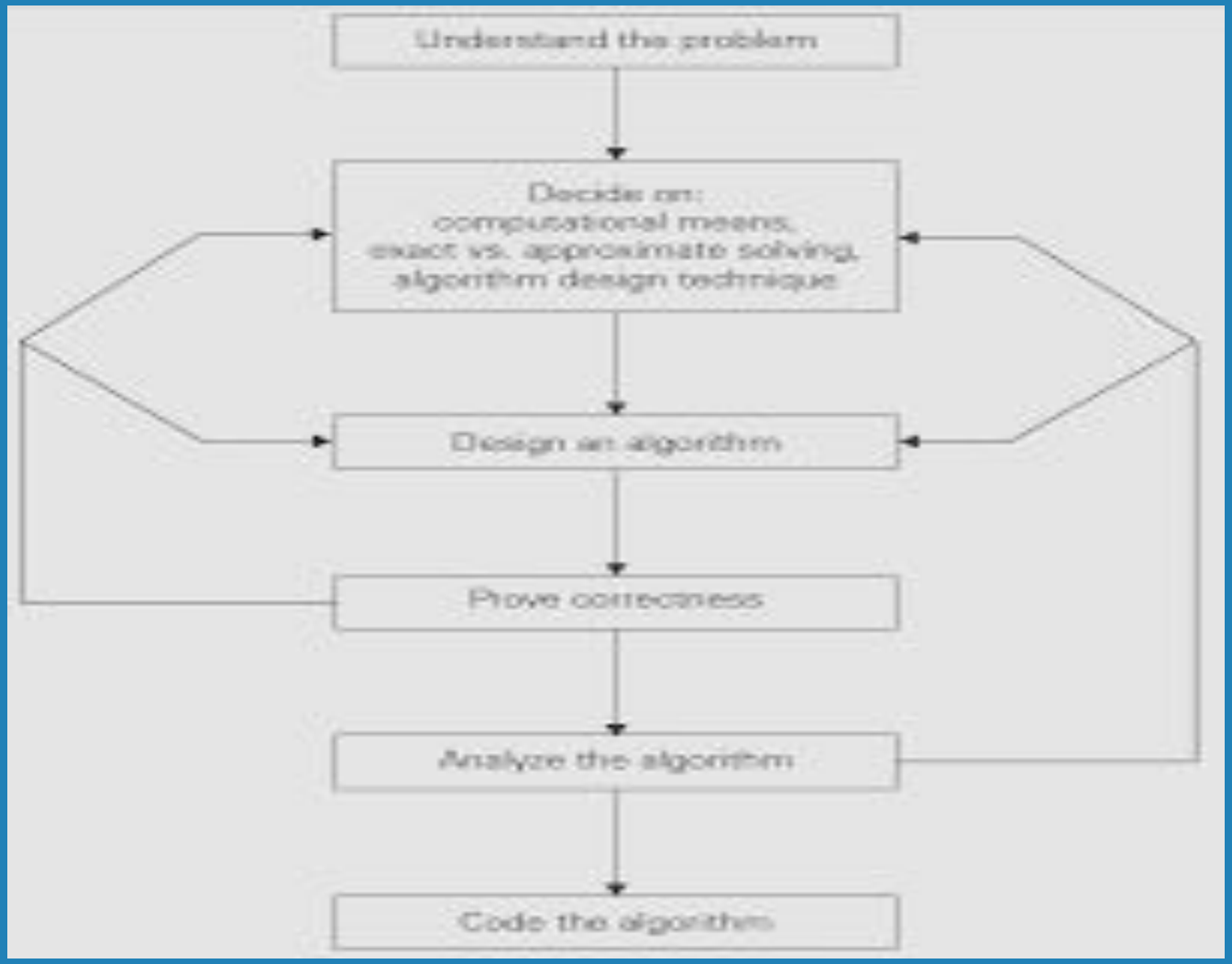
Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency



Algorithm Design and Analysis Process



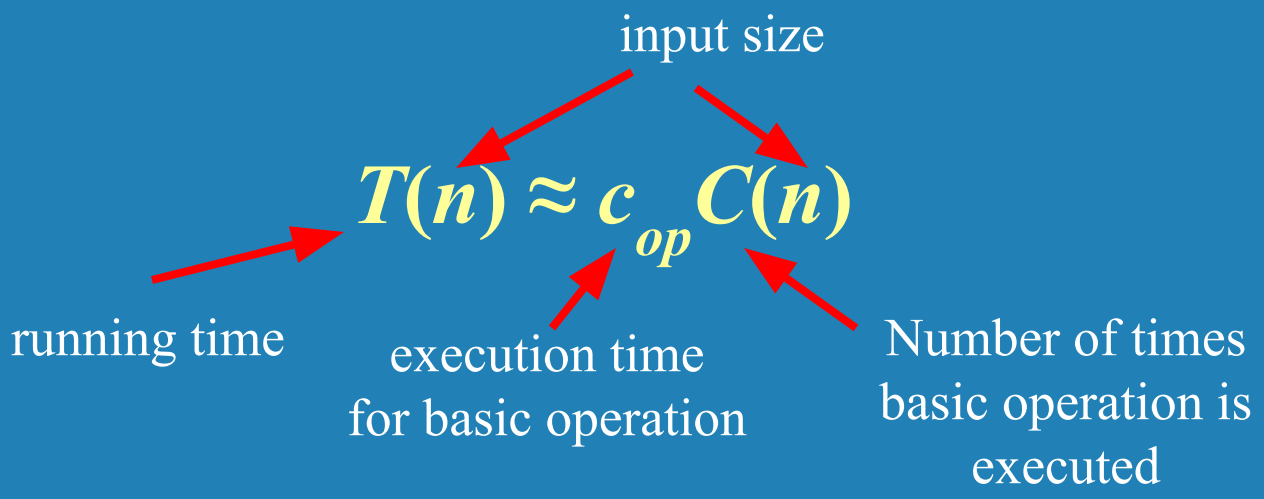
Basic Efficiency Classes

Class	Name	Comments
1	constant	May be in best cases
lgn	logarithmic	Halving problem size at each iteration
n	linear	Scan a list of size n
nlgn	linearithmic	Divide and conquer algorithms, e.g., mergesort
n^2	quadratic	Two embedded loops, e.g., selection sort
n^3	cubic	Three embedded loops, e.g., matrix multiplication
2^n	exponential	All subsets of n-elements set
n!	factorial	All permutations of an n-elements set

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*

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



Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
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

Best-case, average-case, worst-case

For some algorithms efficiency depends on form of input:

- **Worst case:** $C_{worst}(n)$ **– maximum over inputs of size** n
- **Best case:** $C_{best}(n)$ **– minimum over inputs of size** n
- **Average case:** $C_{avg}(n)$ **– "average" over inputs of size** n
  - **Number of times the basic operation will be executed on typical input**
  - **Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs**

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 ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- **Worst case :O(n)**
- **Best case: O(1)**
- **Average case:O(n/2)**

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 $n \to \infty$**

- **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 $n \to \infty$

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 ignores 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

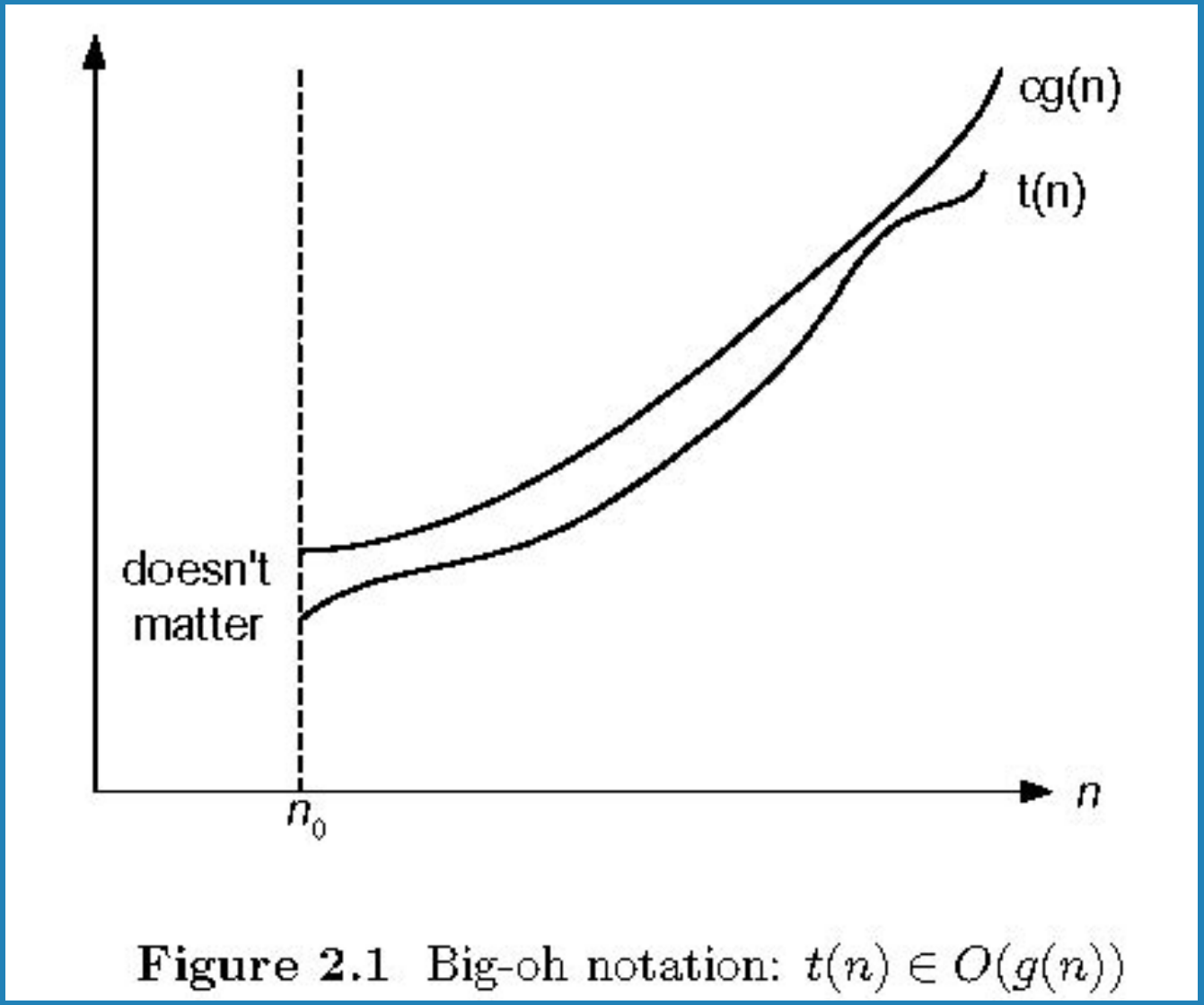


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Establishing order of growth using the definition

Definition: A function $t(n)$ is said to be in $O(g(n))$, if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., there exist positive constant c and non-negative integer n_0 such that

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

Examples:

- $100n+5 \in O(n^2)$
- 100n+5<=100n+n for all n>=n0 n0=5, c=101

Big-omega

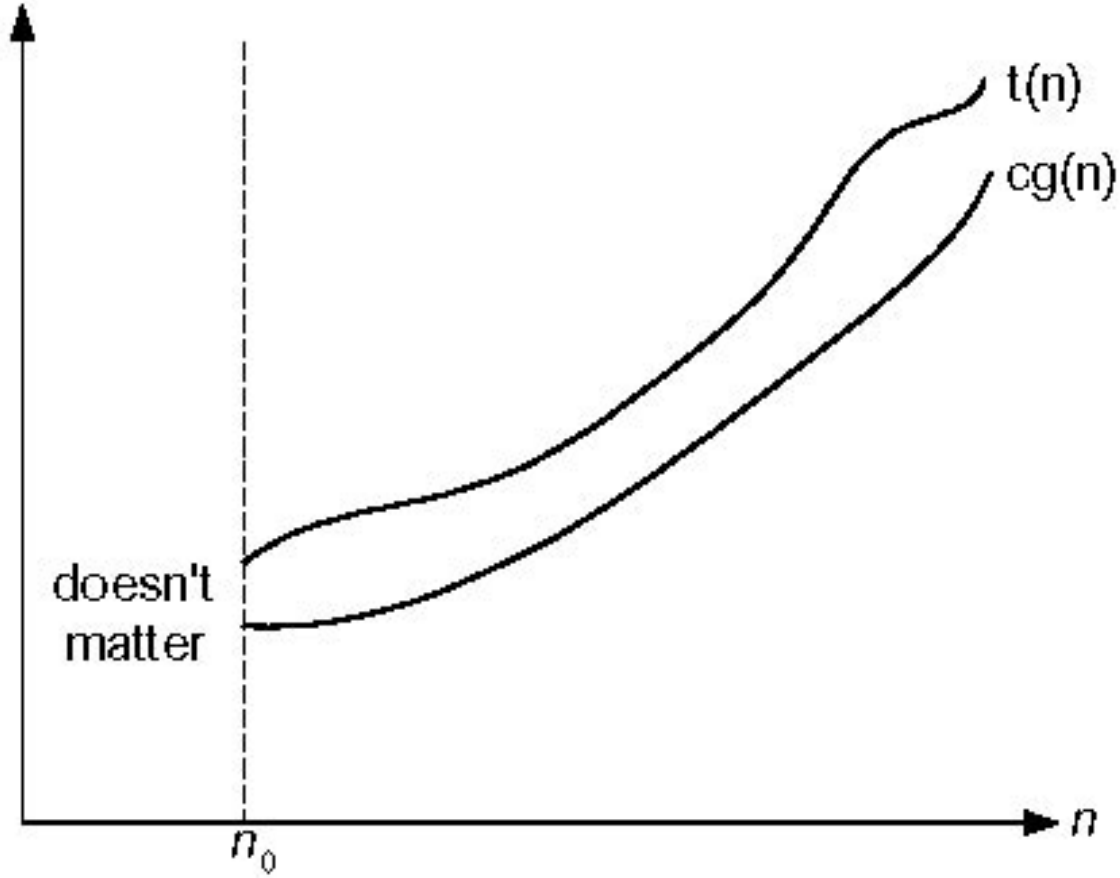


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Establishing order of growth using the definition

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

$$t(n) \geq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $n^3 \geq n^2$ *for all n>=0*

Big-theta

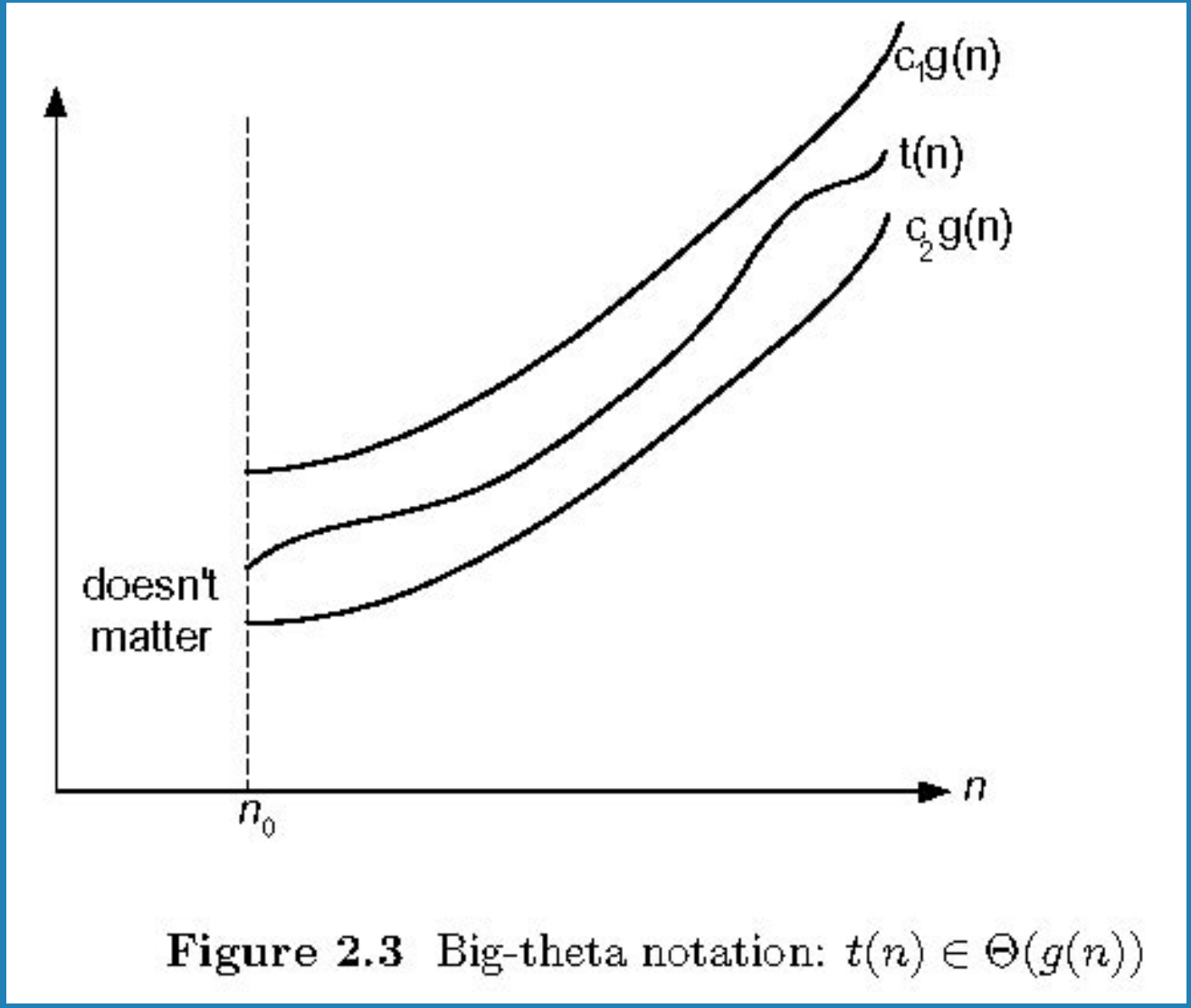


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

Definition: A function $t(n)$ is said to be in $\theta(g(n))$, denoted by $t(n) \in \theta(g(n))$ 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 positive constants c1 and c2 and non-negative integer n_0 such that

$$c2\,g(n) \leq t(n) \geq c1\,g(n) \text{ for every } n \geq n_0$$

Examples:

- $n/3 \leq n \leq 2n$ *for all n>=n0* where c1=2 ,. C2=1/3 , n0=0

Some properties of asymptotic order of growth

If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Then $t_1(n) \leq c1g1(n)$ for all $n \geq n_1$

and $t_2(n) \leq c2g2(n)$ for all $n \geq n_2$

$n \geq \max[n1,n2]$ and $c3 = \max[\, c1,c2\,]$

$$t_1(n) + t_2(n) \leq c1g1(n) + c2g2(n)$$
$$\leq c3g1(n) + c3g2(n) = c3\,[g1(n) + g2(n)\,]$$
$$\leq c3\ 2\ max\,[\, g1(n) + g2(n)\,]$$

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) =$$

- 0 order of growth of $T(n)$ < order of growth of $g(n)$
- $c > 0$ order of growth of $T(n)$ = order of growth of $g(n)$
- ∞ order of growth of $T(n)$ > order of growth of $g(n)$

Examples:

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

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
- 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 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

Basic asymptotic efficiency classes

1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-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

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

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

$\Sigma_{1 \le i \le n} i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

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

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

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

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma c a_i = c \Sigma a_i \quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

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 ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```

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 ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

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
```

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 ← ⌊n/2⌋
return count
```

It cannot be investigated the way the previous examples are.

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: $n! = 1 * 2 * \ldots * (n-1) * n$ 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 $F(n)$

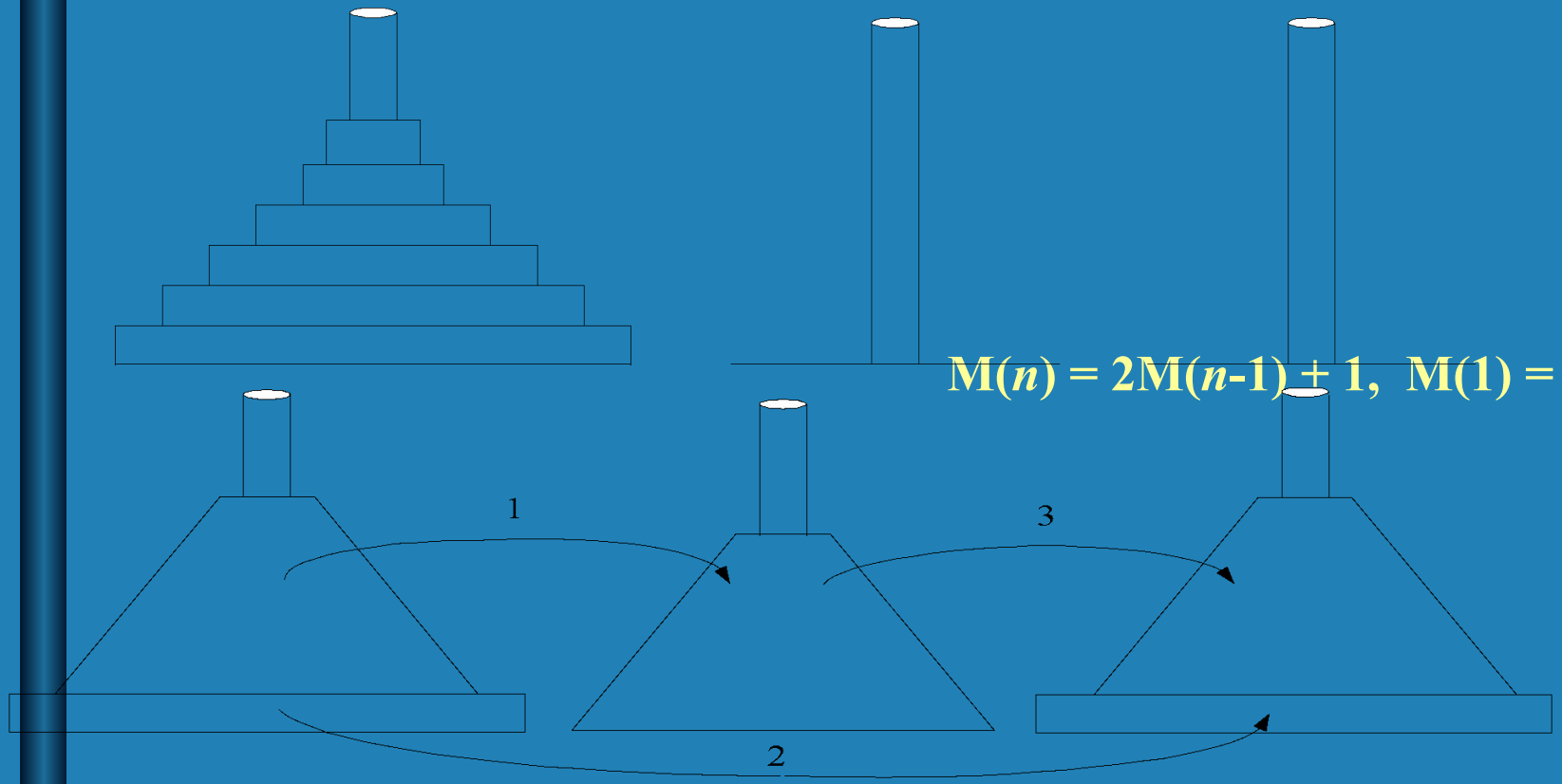
```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Size:

Basic operation: F(n-1)*n

Recurrence relation: M(*n*) = M(*n*-1) + 1, M(0) = 0

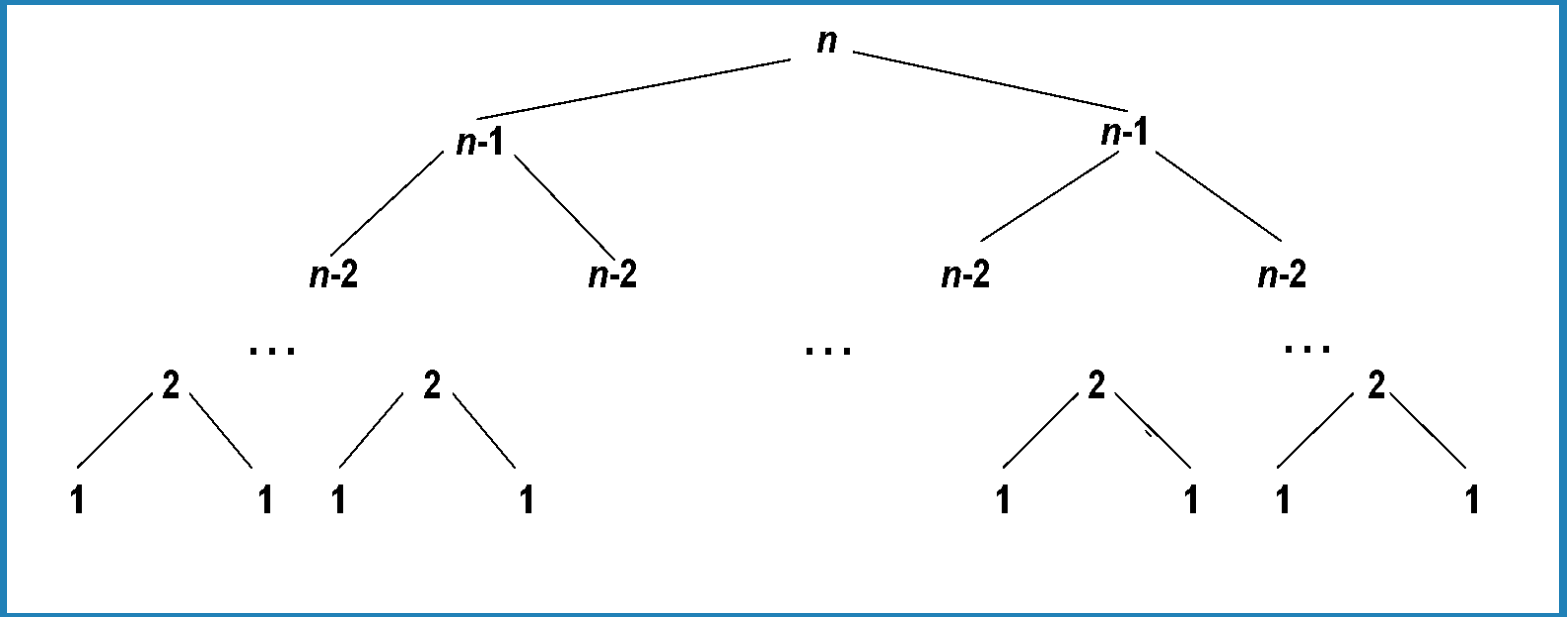
Example 2: The Tower of Hanoi Puzzle



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

Recurrence for number of moves:

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 $BinRec(\lfloor n/2 \rfloor) + 1$

Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The Fibonacci recurrence:

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

$F(0) = 0$

$F(1) = 1$

General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$$