# SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641 107

**An Autonomous Institution**

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

## DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING


## COURSE NAME : 19IT405 DESIGN AND ANALYSIS OF ALGORITHMS

II YEAR /IV SEMESTER

Unit 1- INTRODUCTION

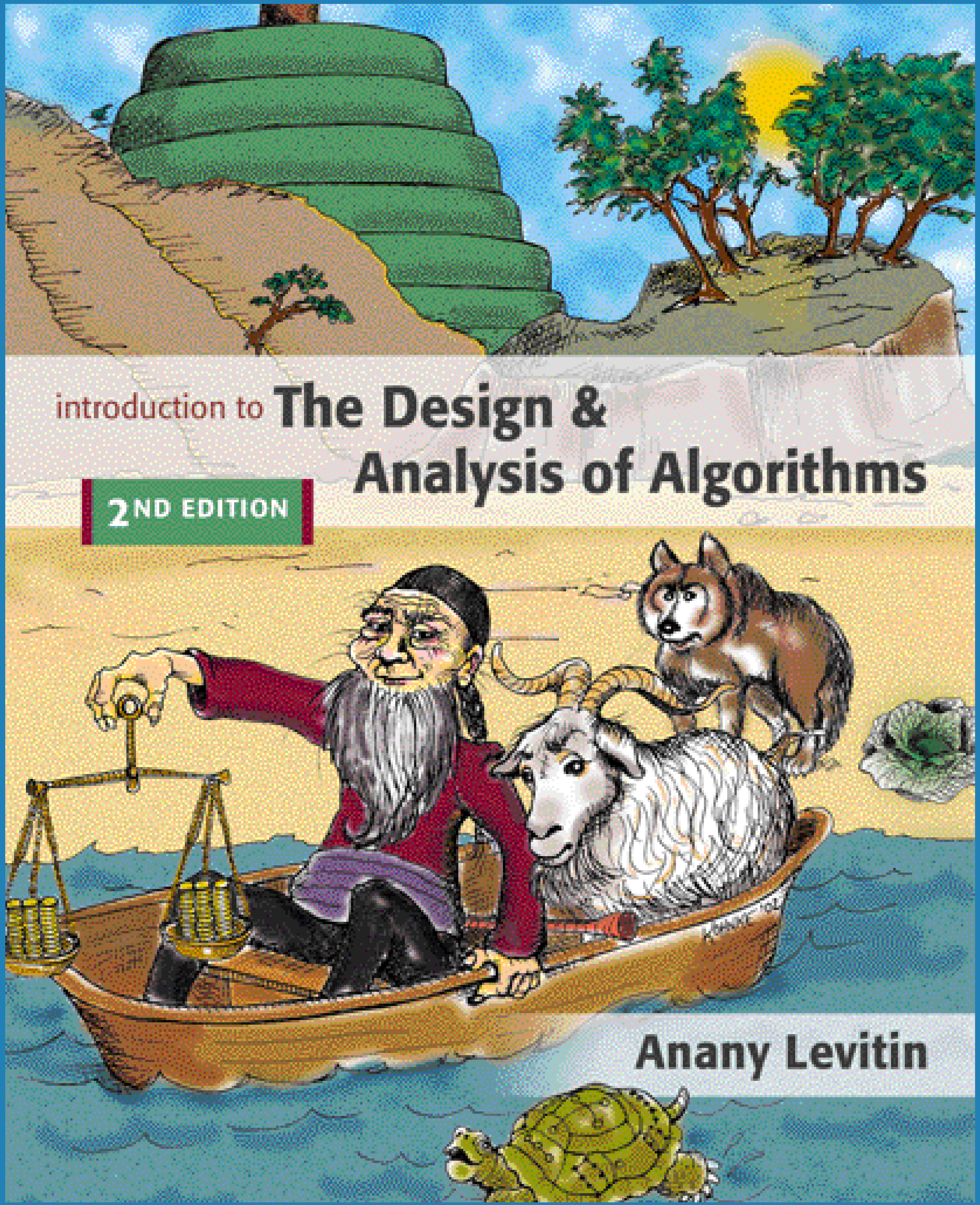
# Topic 3:Fundamentals of the Analysis of Algorithm Efficiency

# Brain Storming

1. What is Algorithm?
2. Why it is important?

# Chapter 2

## Fundamentals of the Analysis of Algorithm Efficiency



# 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***

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

input size

$$T(n) \approx c_{op}C(n)$$

running time

execution time for basic operation or cost

Number of times basic operation is executed

Note: Different basic operations may cost differently!

# 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 |

# 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:** $C_{worst}(n)$ – 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
  - 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 $n \to \infty$

- 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 | $10^1$ | $3.3\cdot10^1$ | $10^2$ | $10^3$ | $10^3$ | $3.6\cdot10^6$ |
| $10^2$ | 6.6 | $10^2$ | $6.6\cdot10^2$ | $10^4$ | $10^6$ | $1.3\cdot10^{30}$ | $9.3\cdot10^{157}$ |
| $10^3$ | 10 | $10^3$ | $1.0\cdot10^4$ | $10^6$ | $10^9$ | | |
| $10^4$ | 13 | $10^4$ | $1.3\cdot10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | 17 | $10^5$ | $1.7\cdot10^6$ | $10^{10}$ | $10^{15}$ | | |
| $10^6$ | 20 | $10^6$ | $2.0\cdot10^7$ | $10^{12}$ | $10^{18}$ | | |

**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 (because?)

- $O(g(n))$: 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

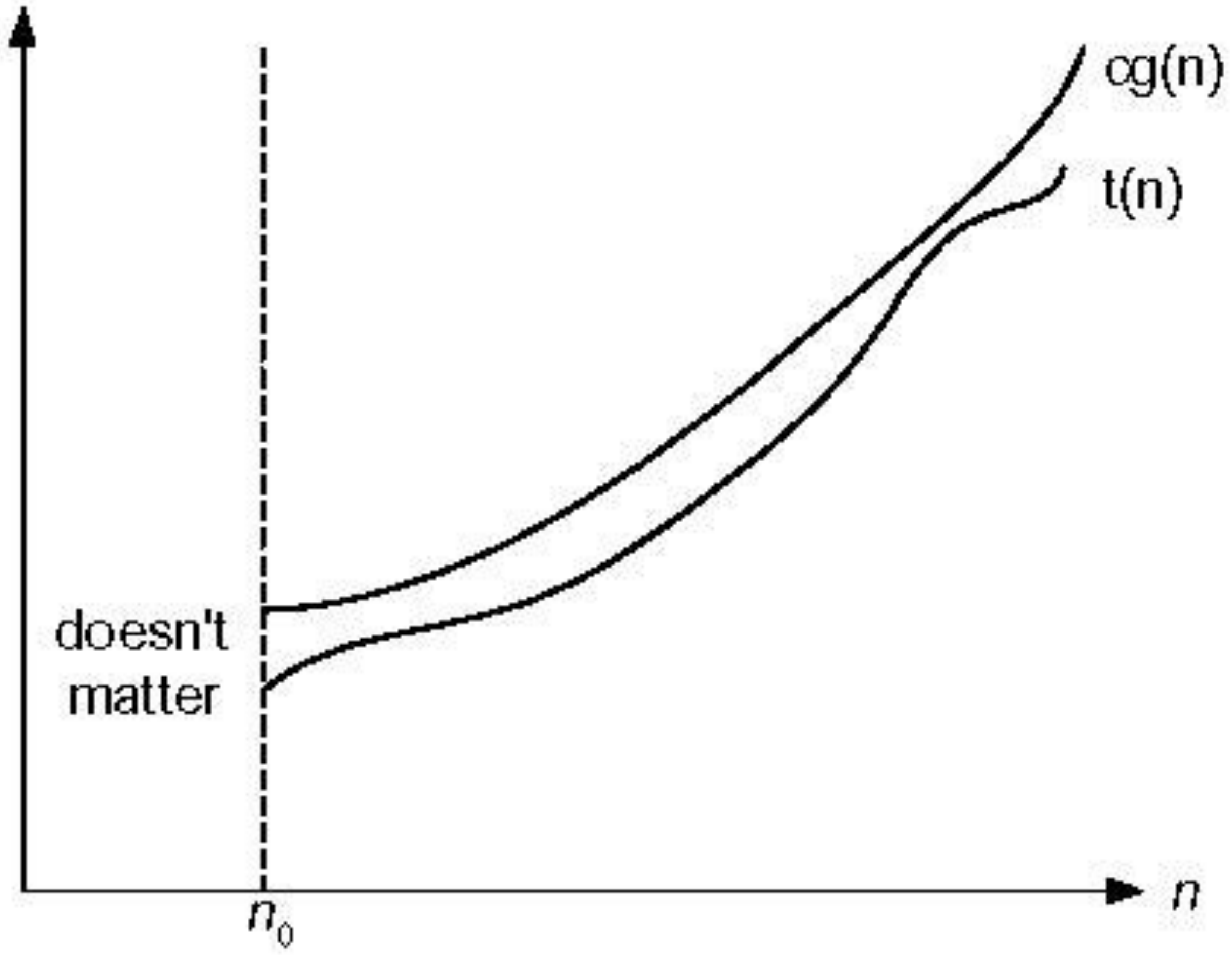



**Figure 2.1** Big-oh notation: $t(n) \in O(g(n))$

# Big-omega

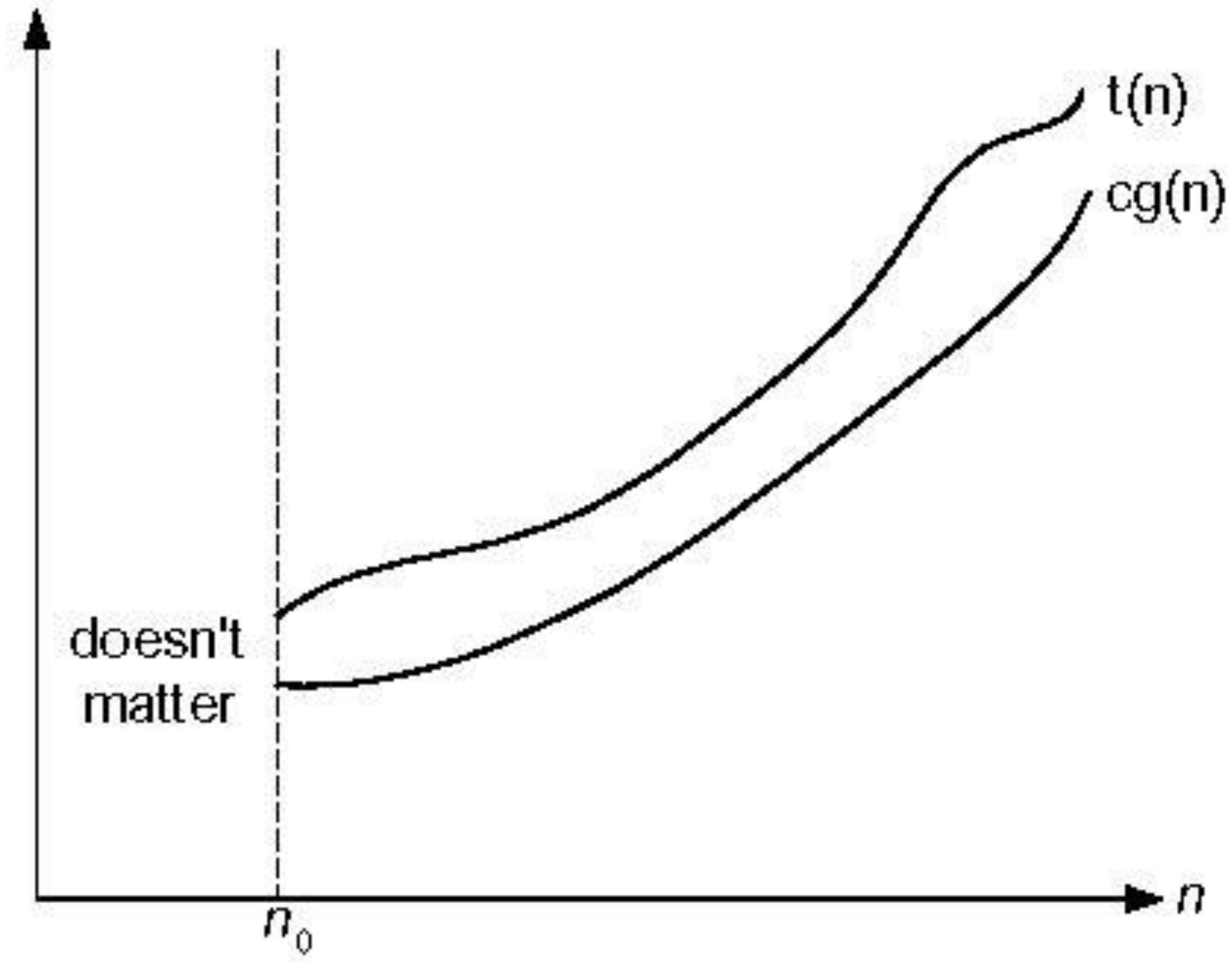



**Fig. 2.2** Big-omega notation: $t(n) \in \Omega(g(n))$

# Big-theta

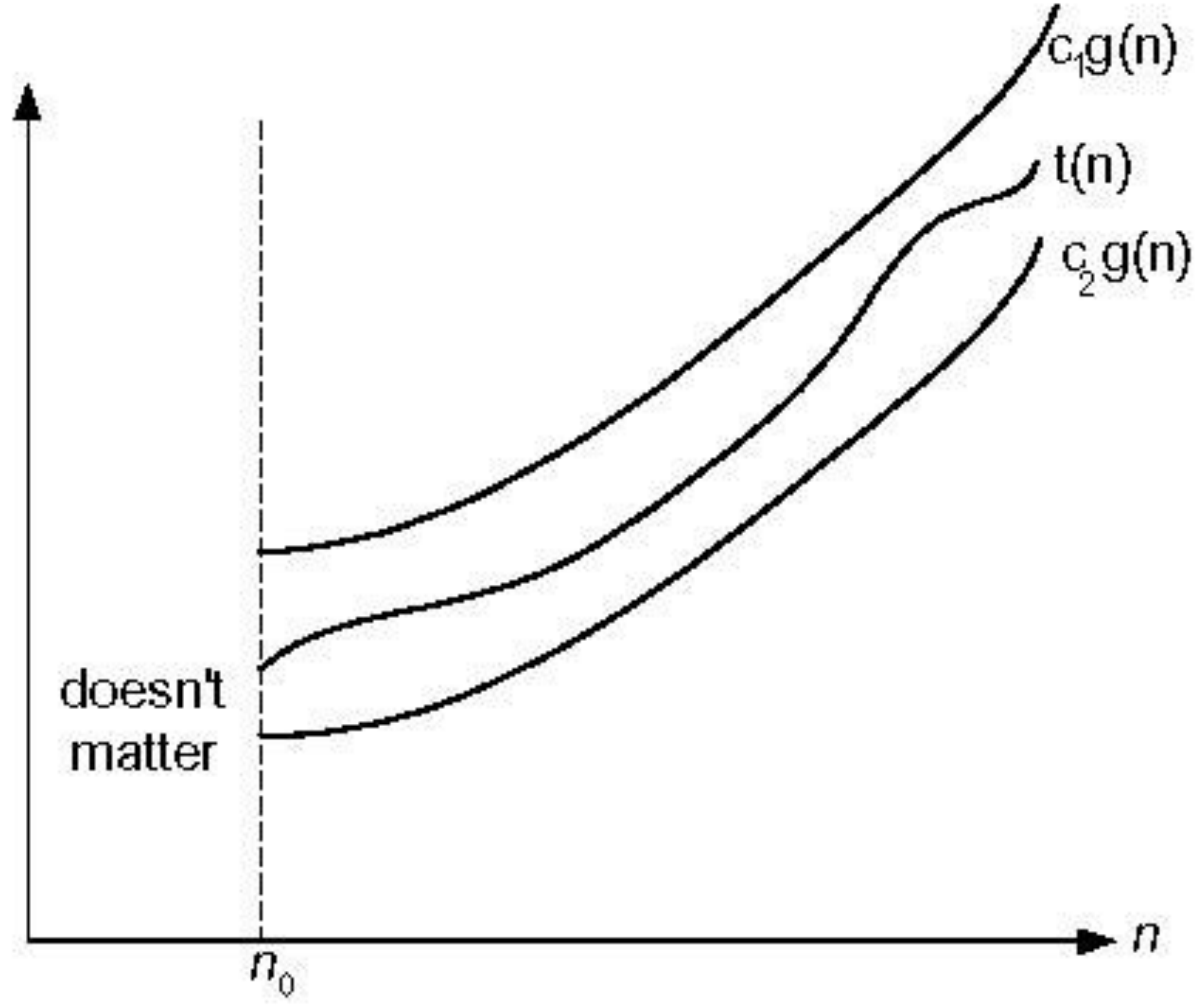



**Figure 2.3** Big-theta notation: $t(n) \in \Theta(g(n))$

# Ω-notation

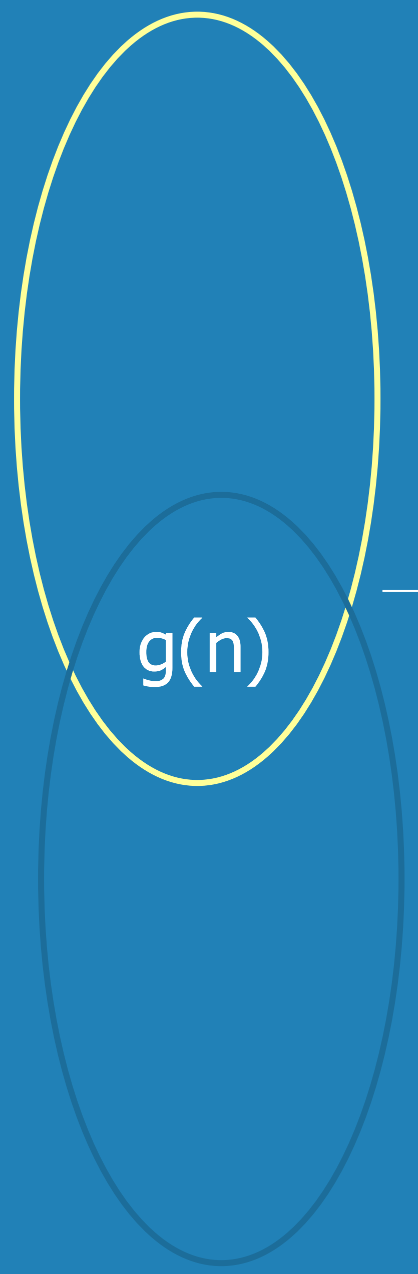
- Formal definition
  - A function *t(n)* is said to be in *Ω(g(n))*, denoted *t(n)* ∈ *Ω(g(n))*, if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer $n_0$ such that

    $t(n) \geq cg(n)$ for all $n \geq n_0$

# Θ-notation

- **Formal definition**
  - **A function *t(n)* is said to be in Θ(*g(n)*), denoted *t(n)* ∈ Θ(*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**

    $$c_2\, g(n) \le t(n) \le c_1\, g(n) \text{ for all } n \ge n_0$$




>=

Ω(g(n)), functions that grow at least as fast as g(n)

=

Θ(g(n)), functions that grow at the same rate as g(n)

<=

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

# Establishing order of growth using 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. $n^2$

# L'Hôpital's rule and Stirling's formula

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

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

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:

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

- Exponential functions $a^n$ have different orders of growth for different $a$'s

- order $\log n$ < order $n^\alpha$ ($\alpha$>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

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

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

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

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

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

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
```

T(n) = $\Sigma_{1 \le i \le n-1}$ ***1 = n-1 =*** **Θ(*n*)** 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**

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

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

$= \Theta(n^2)$ 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 ← 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
```

$T(n) = \Sigma_{0 \le i \le n-1} \Sigma_{0 \le i \le n-1}\; n$

$= \Sigma_{0 \le i \le n-1} \Theta(n^2)$

$= \Theta(n^3)$ multiplications

# 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 ← 1
    while n > 1 do
        count ← count + 1
        n ← ⌊n/2⌋
    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:** $n! = 1 * 2 * \ldots * (n-1) * n$ 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:** n

**Basic operation:** multiplication

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

$M(0) = 0$

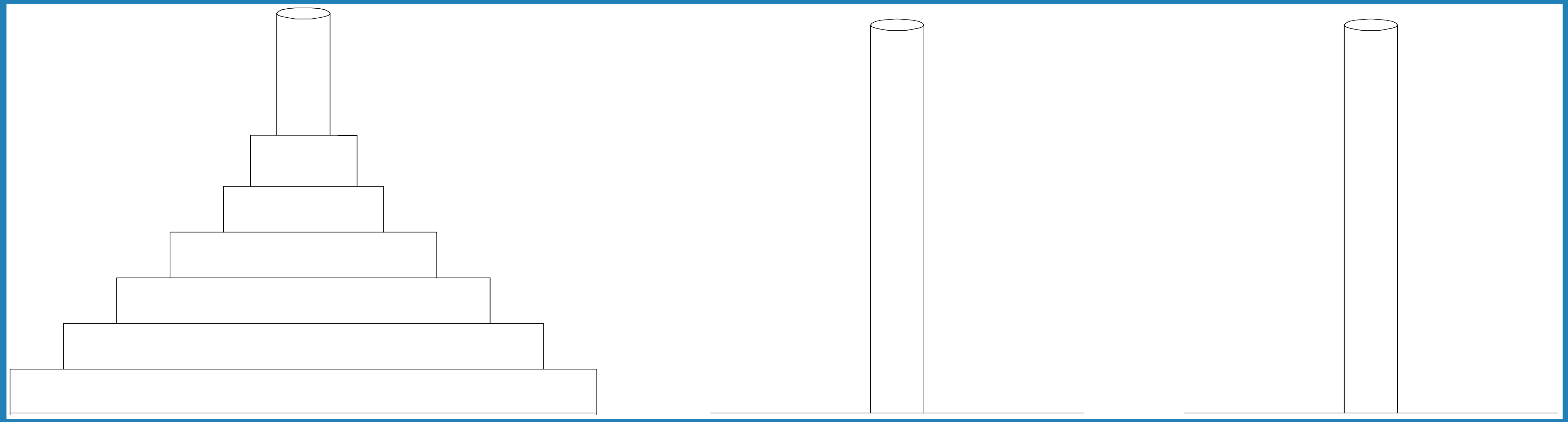
# Solving the recurrence for M($n$)

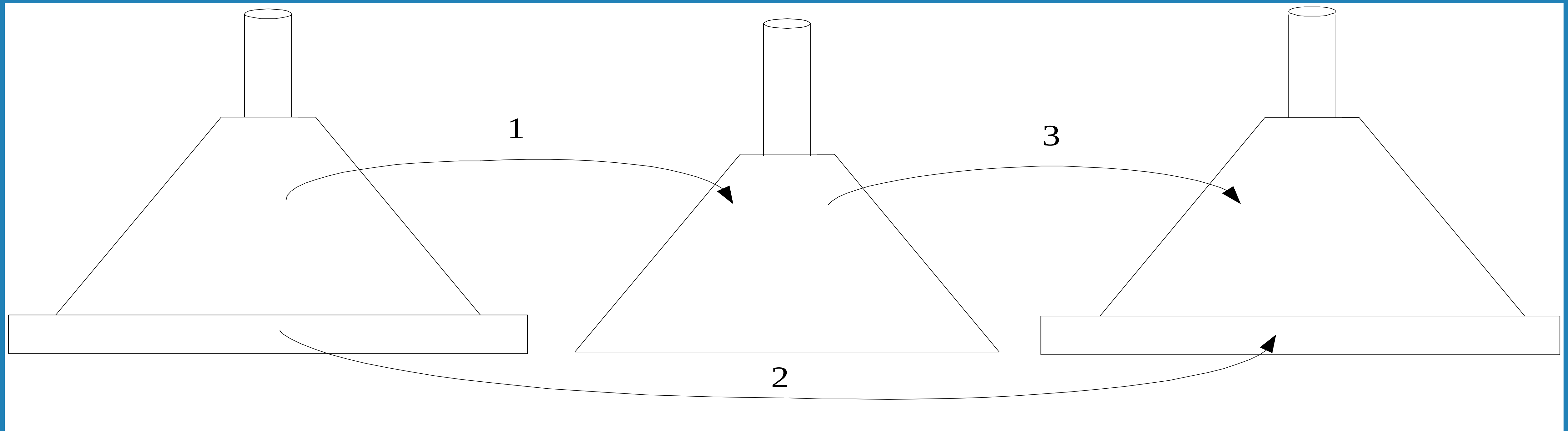
**M($n$) = M($n$-1) + 1, M(0) = 0**

$$
\begin{aligned}
M(n) &= M(n-1) + 1 \\
&= (M(n-2) + 1) + 1 = M(n-2) + 2 \\
&= (M(n-3) + 1) + 2 = M(n-3) + 3 \\
&\ldots \\
&= M(n-i) + i \\
&= M(0) + n \\
&= n
\end{aligned}
$$

The method is called backward substitution.

# Example 2: The Tower of Hanoi Puzzle






**Recurrence for number of moves:**

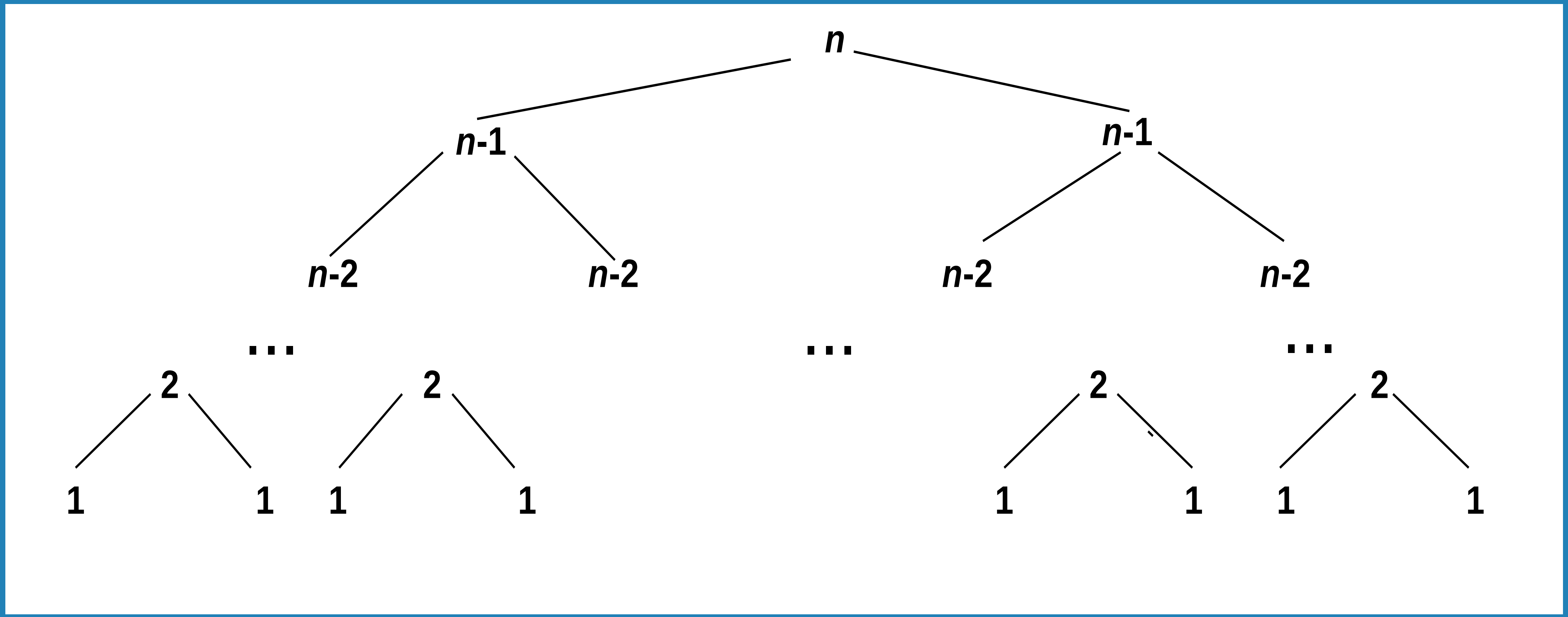
$$M(n) = 2M(n-1) + 1$$

# Solving recurrence for number of moves

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

$$
\begin{aligned}
M(n) &= 2M(n-1) + 1 \\
&= 2(2M(n-2) + 1) + 1 = 2^2*M(n-2) + 2^1 + 2^0 \\
&= 2^2*(2M(n-3) + 1) + 2^1 + 2^0 \\
&= 2^3*M(n-3) + 2^2 + 2^1 + 2^0 \\
&= \ldots \\
&= 2^{(n-1)}*M(1) + 2^{(n-2)} + \ldots + 2^1 + 2^0 \\
&= 2^{(n-1)} + 2^{(n-2)} + \ldots + 2^1 + 2^0 \\
&= 2^n - 1
\end{aligned}
$$

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

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

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

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

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

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

$$= \log_2 n$$

# 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:

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

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

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$ , $k>=0$

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

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

# Assessment 1

1. What is algorithm?

Ans : ________________________________________

2. Why algorithm effectiveness is important?

Ans : ________________________________________

# References

**TEXT BOOKS**

1. Anany Levitin, “Introduction to the Design and Analysis of Algorithms”, Third Edition, Pearson Education, 2012.

**REFERENCES**

1.Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to Algorithms”, Third Edition, PHI Learning Private Limited, 2012.

2. Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, “Data Structures and Algorithms”, Pearson Education, Reprint 2006.

3. Donald E. Knuth, “The Art of Computer Programming”, Volumes 1& 3 Pearson Education, 2009.

4. Steven S. Skiena, “The Algorithm Design Manual”, Second Edition, Springer, 2008.

## Thank You