

Chapter 2: Fundamentals of the Analysis of Algorithm Efficiency

Analysis Framework

Section 2.1. Analysis Framework

- Issues and Approaches
- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

Issues and Approaches

- **Issues**

- ❖ Correctness
- ❖ Time efficiency
- ❖ Space efficiency
- ❖ Optimality

- **Approaches**

- ❖ Theoretical analysis
- ❖ Empirical analysis

Measuring an Input's Size

- Time efficiency : the number of repetitions of the **basic operation** as a function of **input size**
- Input size is influenced by
 - the **data representation**, e.g. matrix
 - the **operations of the algorithm**, e.g. spell-checker

Examples

Problem	Size of input
Find x in an array	The number of the elements in the array
Multiply two matrices	The dimensions of the matrices
Sort an array	The number of elements in the array
Traverse a binary tree	The number of nodes
Solve a system of linear equations	The number of equations, or the number of the unknowns, or both

Units for Measuring Running Time

- Using standard time units is not appropriate
- Counting all operations in an algorithm is not appropriate
- The Approach: Identify and count the basic operation(s) in the algorithm

Basic operations

- Applied to all input items in order to carry out the algorithm
- Contribute most towards the running time of the algorithm

Examples

Problem	Operation
Find x in an array	Comparison of x with an entry in the array
Multiplying two matrices with real entries	Multiplication of two real numbers
Sort an array of numbers	Comparison of two array entries plus moving elements in the array
Traverse a tree	Traverse an edge

Work done by an algorithm

Let

$T(n)$: running time

c_{op} : execution time for basic operation

$C(n)$: number of times basic operation is executed

- Then we have: $T(n) \approx c_{\text{op}} C(n)$

Types of formulas for basic operation count

- Exact formula

$$\text{e.g., } C(n) = n(n-1)/2$$

- Formula indicating order of growth with specific multiplicative constant

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

- Formula indicating order of growth with unknown multiplicative constant

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

Example

Let $C(n) = 3n(n-1) \approx 3n^2$

Suppose we double the input size. How much longer the program will run?

Orders of Growth

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

Worst-Case, Best-Case, and Average-Case Efficiencies

- Specifics of the input

We define

- **Worst case:**

$W(n)$ – “maximum” over inputs of size n

- **Best case:**

$B(n)$ – “minimum” over inputs of size n

- **Average case:**

$A(n)$ – “average” over inputs of size n

Average-Case

- 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 repetitions considered as a random variable under some assumption about the probability distribution of all possible inputs of size n

Example

Sequential search

- **Problem:** Given a list of n elements and a search key K , find an element equal to K , if any.
- **Algorithm:** Scan the list and compare its successive elements with K until either a matching element is found (*successful search*) or the list is exhausted (*unsuccessful search*)
- Worst case: $W(n) \approx$
- Best case: $B(n) \approx$
- Average case: $A(n) \approx$

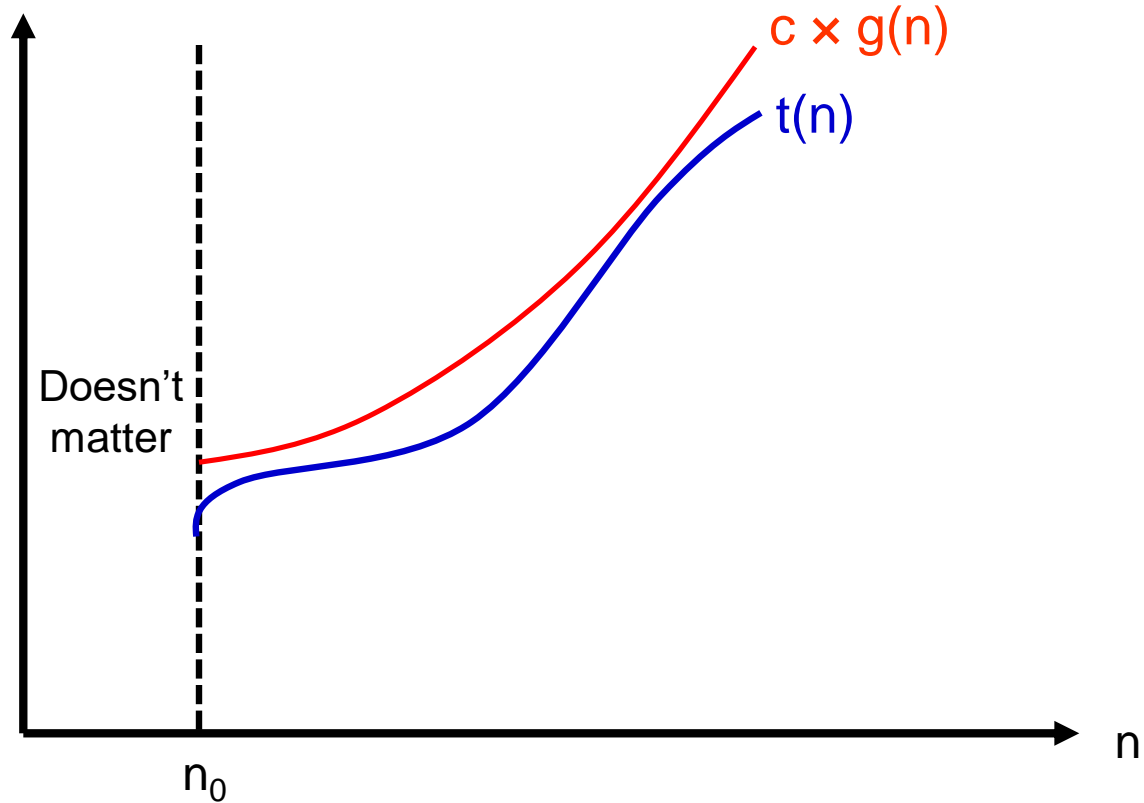
Summary of Analysis Framework

- Time and space efficiencies are functions of input size
- Time efficiency is # of times basic operation is executed
- Space efficiency is # of extra memory units consumed
- Efficiencies of some algorithms depend on type of input: requiring worst, best, average case analysis
- Focus is on order of growth of running time (or extra memory units) as input size goes to infinity

Asymptotic Growth Rate

- 3 notations used to compare orders of growth of an algorithm's basic operation count
 - $O(g(n))$: Set of functions that grow no faster than $g(n)$
 - $\Omega(g(n))$: Set of functions that grow at least as fast as $g(n)$
 - $\Theta(g(n))$: Set of functions that grow at the same rate as $g(n)$

O(big oh)-Notation



$$t(n) \in O(g(n))$$

O-Notation (contd.)

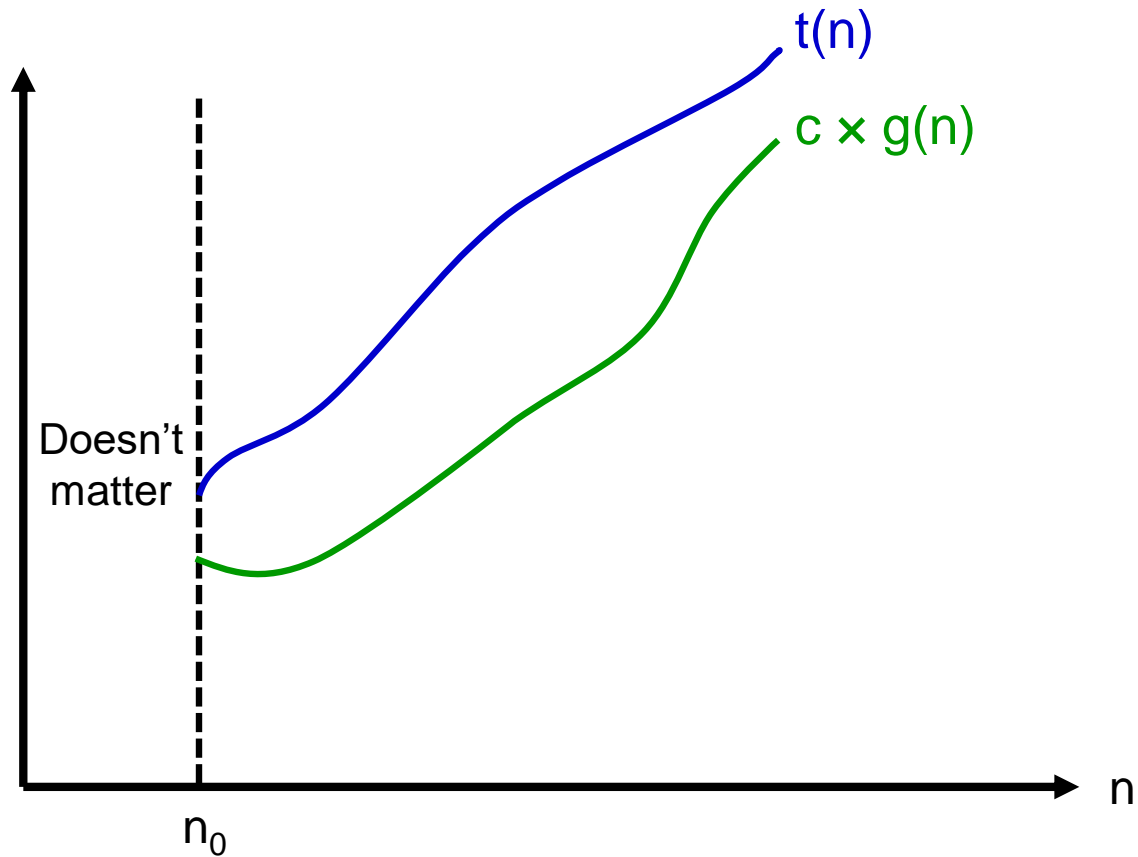
- **Definition:** A function $t(n)$ is said to be in $O(g(n))$, denoted $t(n) \in O(g(n))$, if $t(n)$ is bounded above by some positive constant multiple of $g(n)$ for sufficiently large n .
- If we can find +ve constants c and n_0 such that:

$$t(n) \leq c \times g(n) \text{ for all } n \geq n_0$$

O-Notation (contd.)

- Is $100n+5 \in O(n^2)$?
- Is $3n^2 + 4n - 2 = O(n^2)$?
- Is $n^3 \neq O(n^2)$?

Ω (big omega)-Notation



$$t(n) \in \Omega(g(n))$$

Ω -Notation (contd.)

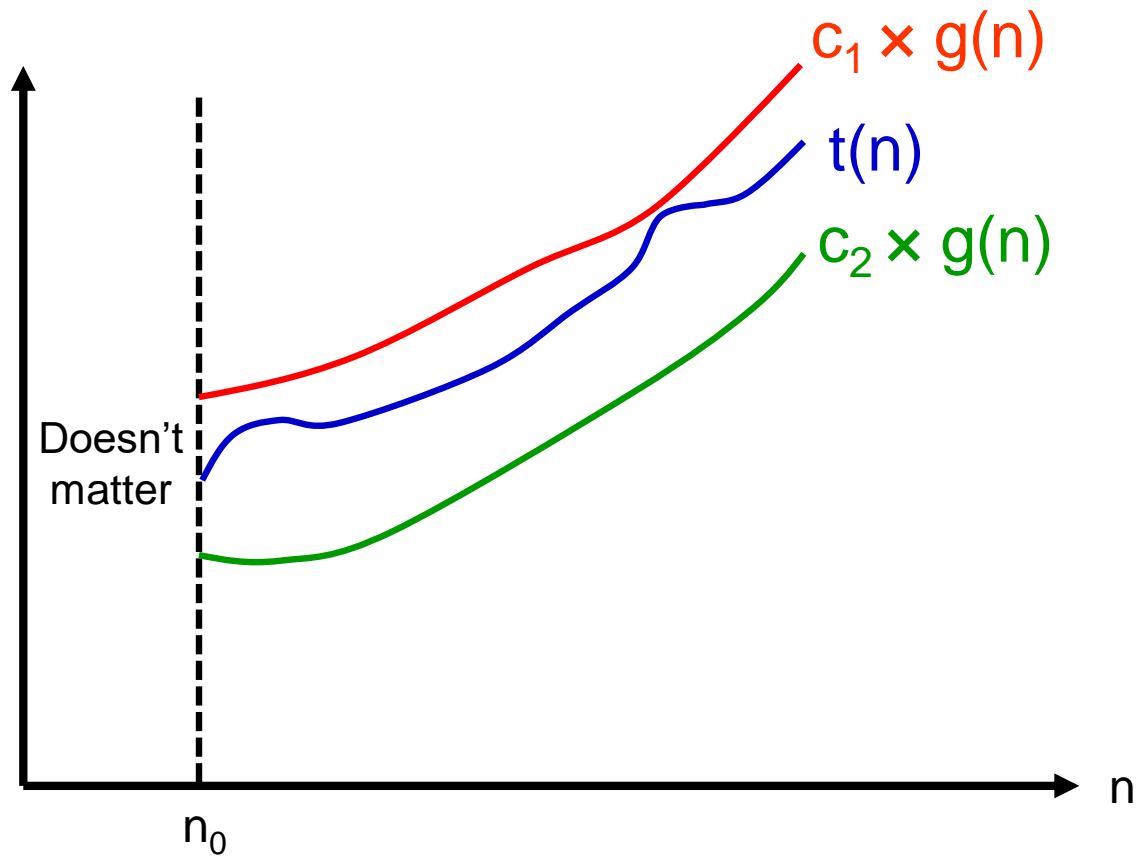
- **Definition:** A function $t(n)$ is said to be in $\Omega(g(n))$ denoted $t(n) \in \Omega(g(n))$, if $t(n)$ is bounded below by some positive constant multiple of $g(n)$ for all sufficiently large n .
- If we can find +ve constants c and n_0 such that

$$t(n) \geq c \times g(n) \text{ for all } n \geq n_0$$

Ω -Notation (contd.)

- Is $n^3 \in \Omega(n^2)$?
- Is $n^2 + 100n = \Omega(n^2)$

Θ (big theta)-Notation



$$t(n) \in \Theta(g(n))$$

Θ -Notation (contd.)

- 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 sufficiently large n .
- If we can find +ve constants c_1 , c_2 , and n_0 such that

$$c_2 \times g(n) \leq t(n) \leq c_1 \times g(n) \text{ for all } n \geq n_0$$

Θ -Notation (contd.)

- Is $n^2 + 100n = \theta(n^2)$

Theorem

- If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then
$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$$
- Analogous assertions are true for Ω and Θ notations.
- **Implication:** if sorting makes no more than n^2 comparisons and then binary search makes no more than $\log_2 n$ comparisons, then efficiency is
$$O(\max\{n^2, \log_2 n\}) = O(n^2)$$

Using Limits for Comparing Orders of Growth

• $\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} =$

0	implies that $t(n)$ grows slower than $g(n)$
$C > 0$	implies that $t(n)$ grows at the same order as $g(n)$
∞	implies that $t(n)$ grows faster than $g(n)$

1. First two cases (0 and c) means $t(n) \in O(g(n))$
2. Last two cases (c and ∞) means $t(n) \in \Omega(g(n))$
3. The second case (c) means $t(n) \in \Theta(g(n))$

Taking Limits

- **L'Hôpital's rule:** if limits of both $t(n)$ and $g(n)$ are ∞ as n goes to ∞ ,

$$\lim_{n \rightarrow \infty} \frac{t(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{t'(n)}{g'(n)} = \lim_{n \rightarrow \infty} \frac{\frac{dt(n)}{dn}}{\frac{dg(n)}{dn}}$$

- **Stirling's formula:** For large n , we have $n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$

EXAMPLE Compare the orders of growth of $n!$ and 2^n . (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

$$\lim_{n \rightarrow \infty} \frac{n!}{2^n} = \lim_{n \rightarrow \infty} \frac{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n}{2^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \frac{n^n}{2^n e^n} = \lim_{n \rightarrow \infty} \sqrt{2\pi n} \left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$;

Summary of How to Establish Order of Growth of Basic Operation Count

- **Method 1:** Using Limits
- **Method 2:** Using Theorem
- **Method 3:** Using definitions of O -, Ω -, and Θ -notations.

Basic Efficiency Classes

Class	Name	Comments
1	constant	May be in best cases
log	logarithmic	Halving problem size at each iteration
n	linear	Scan a list of size n
$n \times \log n$	"n-log-n"	Divide and conquer algorithms, e.g., merge sort
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

Important Summation Formulas

- $\sum_{i=l}^u 1 = u - l + 1$
- $\sum_{i=1}^n i = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$
- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$
- $\sum_{i=0}^n a^i = \frac{a^{n+1}-1}{a-1} \quad (a \neq 1)$
- $\sum_{i=1}^n i a^i = a \frac{d}{da} \left(\frac{a^{n+1}-1}{a-1} \right)$

$$\sum_{i=l}^u c a_i = c \sum_{i=l}^u a_i$$
$$\sum_{i=l}^u (a_i \pm b_i) = \sum_{i=l}^u a_i \pm \sum_{i=l}^u b_i$$

Analysis of Non-recursive Algorithms

ALGORITHM MaxElement(A[0..n-1])

//Determines largest element

maxval <- A[0]

for i <- 1 **to** n-1 **do**

if A[i] > maxval

 maxval <- A[i]

return maxval

Input size: n

Basic operation: > or <-

$$\begin{aligned} C(n) &= \sum_{i=1}^{n-1} 1 = n - 1 - 1 + 1 \\ &= n - 1 \in \Theta(n) \end{aligned}$$

General Plan for Analysis of Non-recursive algorithms

- Decide on a parameter indicating an **input's size**
- Identify the **basic operation**
- Does $C(n)$ depends **only on n** or does it also **depend on type** of input?
- **Set up sum** expressing the # of times basic operation is executed.
- Find closed form for the sum or at least establish its order of growth

Analysis of Non-recursive (contd.)

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

//Determines whether all elements are //distinct

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

Input size: n

Basic operation: $A[i] = A[j]$

Does $C(n)$ depend on type of input?

UniqueElements (contd.)

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

$$C_{\text{worst}}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \frac{(n-1)n}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2)$$

Why $C_{\text{worst}}(n) \in \Theta(n^2)$ is better than
saying $C_{\text{worst}}(n) \in O(n^2)$?

Analysis of Non-recursive (contd.)

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

//Output: 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

Input size: n

Basic operation: ×

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

Analysis of Non-recursive (contd.)

ALGORITHM Binary(n)

// Output: # of digits in binary

// representation of n

count <- 1

while n > 1 do

 count <- count+1

$n \leftarrow \left\lfloor \frac{n}{2} \right\rfloor$

return count

Input size: n

Basic operation: $\lfloor - \rfloor$

$C(n) = ?$

Each iteration halves n

Let m be such that $2^m \leq n < 2^{m+1}$

Take lg: $m \leq \lg(n) < m+1$, so

$m = \lfloor \lg(n) \rfloor$ and $m+1 = \lfloor \lg(n) \rfloor + 1 \in \Theta(\lg(n))$

Analysis of Recursive Algorithms

ALGORITHM F(n)

// Output: n!

if n = 0

 return 1

else

 return n × F(n-1)

Input size: n

Basic operation: ×

$M(n) = M(n-1) + 1$ for $n > 0$

to compute
F(n-1)

to multiply
n and F(n-1)



Analysis of Recursive (contd.)

$$\begin{array}{l} M(n) = M(n-1) + 1 \\ M(0) = 0 \end{array} \quad \left. \vphantom{\begin{array}{l} M(n) = M(n-1) + 1 \\ M(0) = 0 \end{array}} \right\} \text{Recurrence relation}$$

By Backward substitution

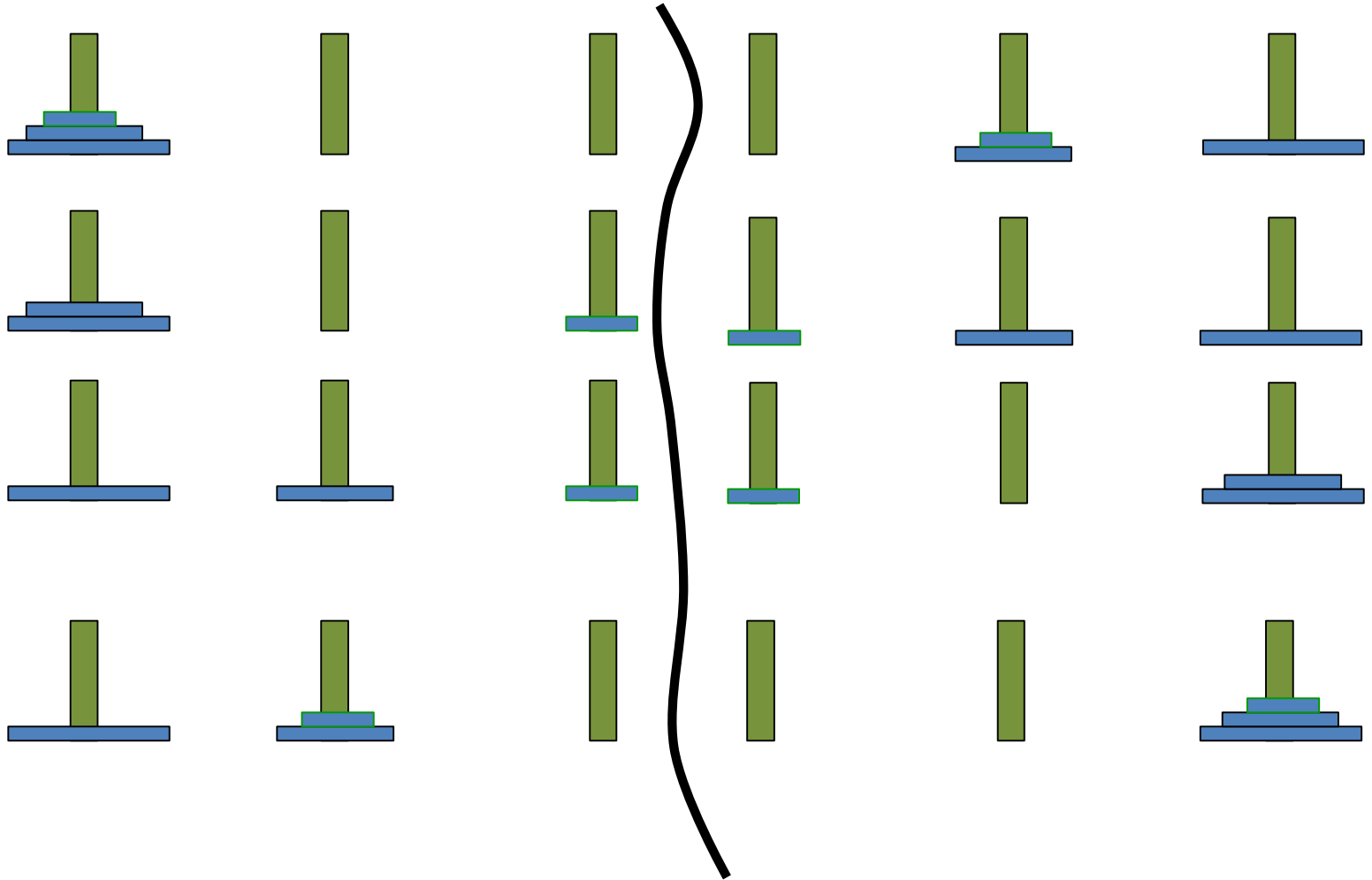
$$\begin{aligned} M(n) &= M(n-1) + 1 = M(n-2) + 1 + 1 = \dots \dots \\ &= M(n-n) + \underbrace{1 + \dots + 1}_{n \text{ 1's}} = 0 + n = n \end{aligned}$$

We could compute it non-recursively, saves function call overhead

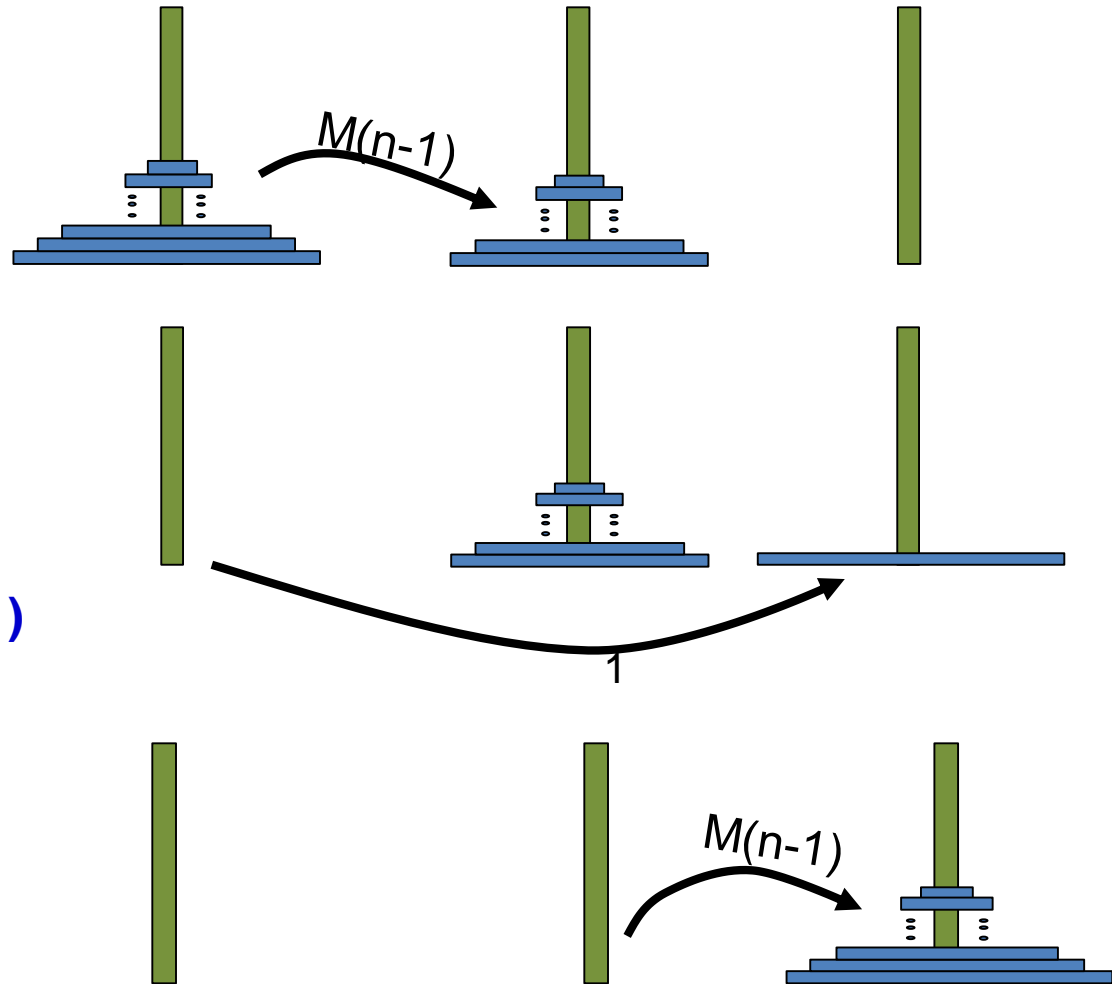
General Plan for Recursive

- Decide on **input size** parameter
- Identify the **basic operation**
- Does $C(n)$ depends also on **input type**?
- Set up a ***recurrence relation***
- Solve the recurrence or, at least establish the order of growth of its solution

Analysis of Recursive (contd.): Tower of Hanoi



Tower of Hanoi (contd.)



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

$$M(1) = 1$$


Tower of Hanoi (contd.)

ALGORITHM ToH(n, s, m, d)


if $n = 1$  $M(1) = 1$

 print “move from s to d ”

else

 ToH($n-1, s, d, m$)  $M(n-1)$

 print “move from s to d ”  $M(1)$

 ToH($n-1, m, s, d$)  $M(n-1)$

Tower of Hanoi (contd.)

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1$$

$$M(1) = 1$$

$$M(n) = 2M(n-1) + 1 \text{ [Backward substitution...]}$$

$$= 2[2M(n-2)+1] + 1 = 2^2M(n-2)+2+1$$

$$= 2^2[2M(n-3)+1]+2+1 = 2^3M(n-3)+2^2+2+1$$

... ..

$$= 2^{n-1}M(n-(n-1))+2^{n-2}+2^{n-3}+.....+2^2+2+1$$

$$= 2^{n-1}+ 2^{n-2}+2^{n-3}+.....+2^2+2+1 = \sum_{i=0}^{n-1} 2^i = 2^n-1$$

$$M(n) \in \Theta(2^n)$$

Be careful! Recursive algorithm may look simple
But might easily be exponential in complexity.