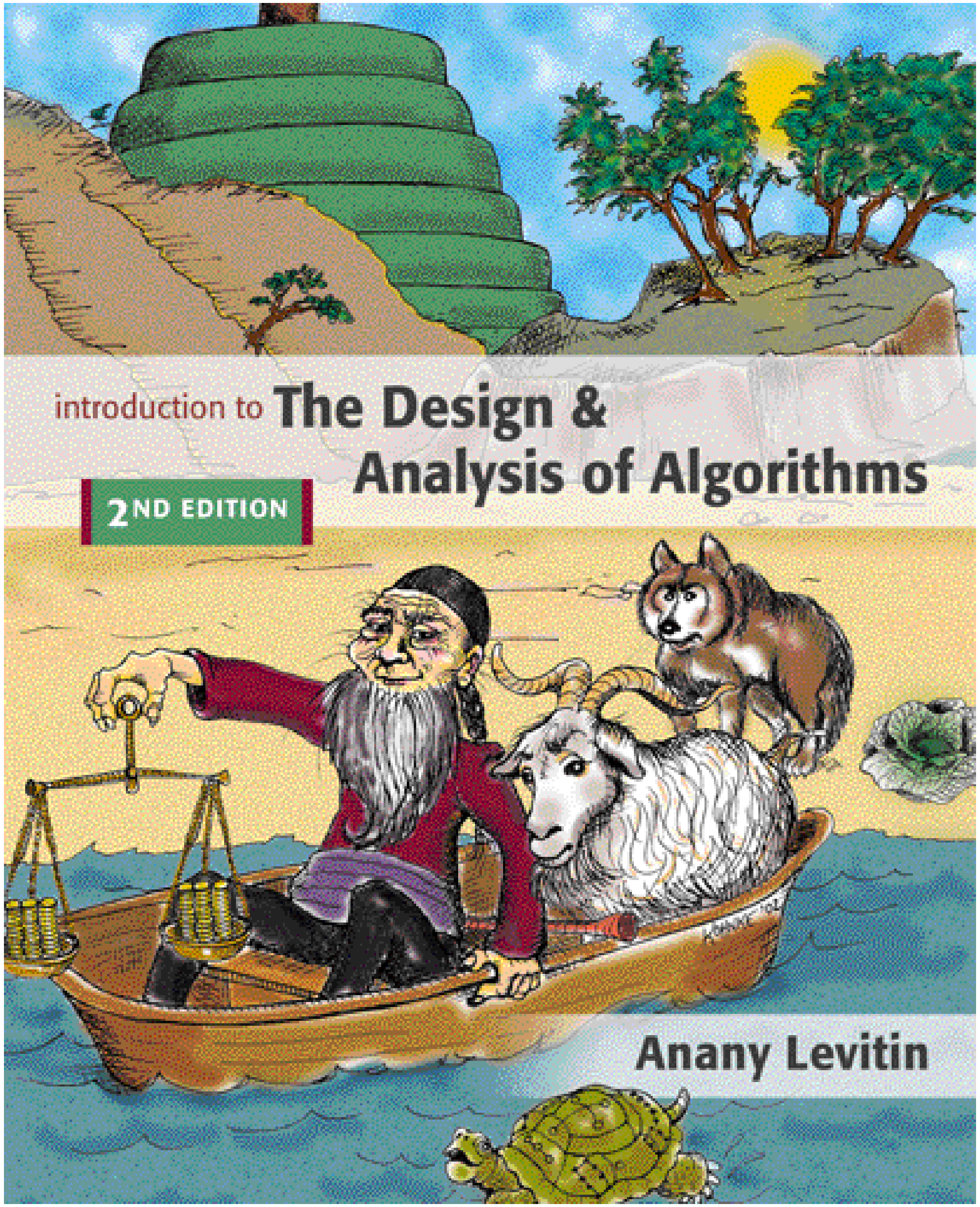
Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency



Analysis of algorithms

- Issues:
  - Correctness – hard even for simple algorithms
  - Time efficiency – how fast (for all inputs)
  - Space efficiency – how much extra memory is necessary
  - Optimality – error rate for approximate algorithms

- 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 most towards the running time of the algorithm

input size

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

running time

execution time
for basic operation

Number of times
basic operation is
executed

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: $T_{worst}(n)$ – maximum over inputs of size n
- Best case: $T_{best}(n)$ – minimum over inputs of size n
- Average case: $T_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on <u>typical</u> 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

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
- Best case
- Average case

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

$$C_{worst}(n) = n$$

C= # comparisons

- Best case

$$C_{best}(n) = 1$$

Example: Sequential search

- Average case

Assumptions:

(a) The probability or a successful search is equal to p $(0 \le p \le 1)$;

(b) The probability of the first match occurring in the *i*th position of the list is the same for every *i*.

- Successful case: probability of occurring in position *i* is *p/n*, for every *i*, and the number of comparisons made is *i*;
- Unsuccessful case: probability is *(1-p)* with *n* comparisons.

Therefore:

$$C_{avg}(n) = [1.p/n + 2.p/n + \ldots + i.p/n + \ldots n.p/n] + n.(1-p)$$
$$= p/n[1+2+\ldots+i+\ldots+n] + n(1-p)$$
$$= p/n.\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

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*→∞**

- 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?

Theoretical analysis of time efficiency

- *How much faster would this algoritm run on a machine that is ten times faster than the one I have?*

  Resp: 10

- Assuming that

$$C(n) = \tfrac{1}{2}\, n\,(n-1)$$

How much longer will the algoritm run if we double its input size?

For all but very small values of n:

$$C(n) = \tfrac{1}{2}\, n\,(n-1) = \tfrac{1}{2}\, n^2 - \tfrac{1}{2}\, n \approx \tfrac{1}{2}\, n^2$$

$$\Rightarrow \quad \frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\tfrac{1}{2}(2n)^2}{\tfrac{1}{2}n^2} = 4$$

- unknown c_{op};
- constant ½ cancelled out

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

$$\boxed{\log_a n = \log_a b \, \log_b n}$$

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 <u>no faster</u> than $g(n)$

  $n \in O(n^2)$ $100n + 5 \in O(n^2)$ $1/2n(n-1) \in O(n^2)$

- $\Omega(g(n))$: class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$

  $n^3 \in \Omega(n^2)$ $100n + 5 \notin \Omega(n^2)$ $1/2n(n-1) \in \Omega(n^2)$

- $\Theta(g(n))$: class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$

  $5n \in \Theta(n)$ $100n + 5 \notin \Theta(n^2)$ $1/2n(n-1) \in \Theta(n^2)$

Big-oh

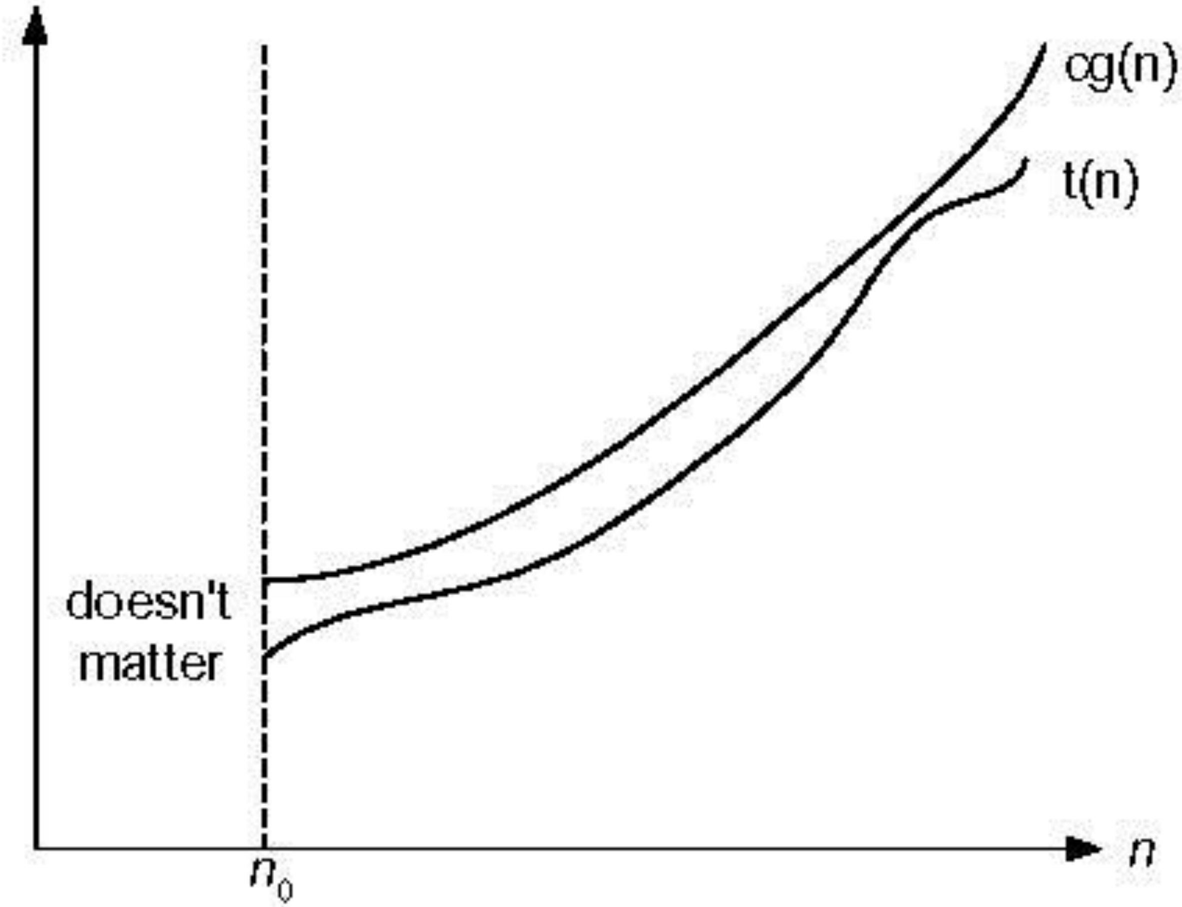


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

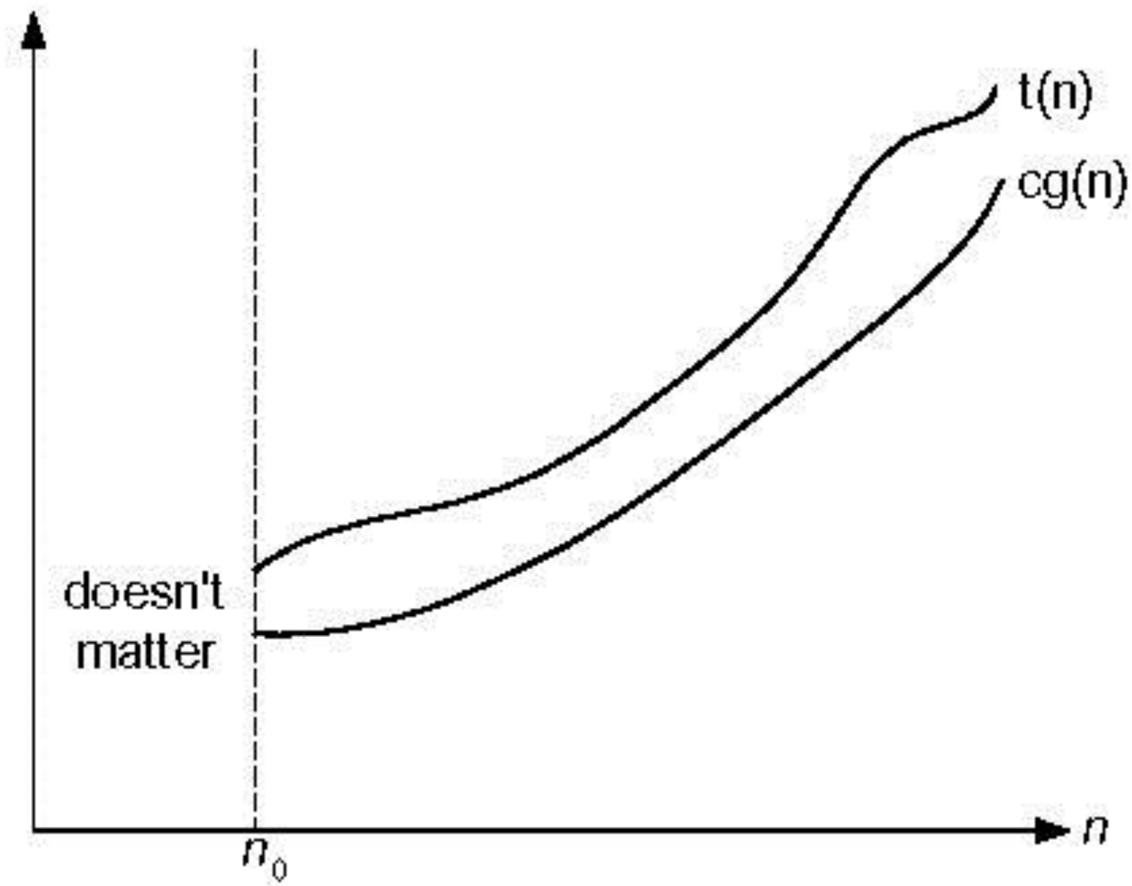


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

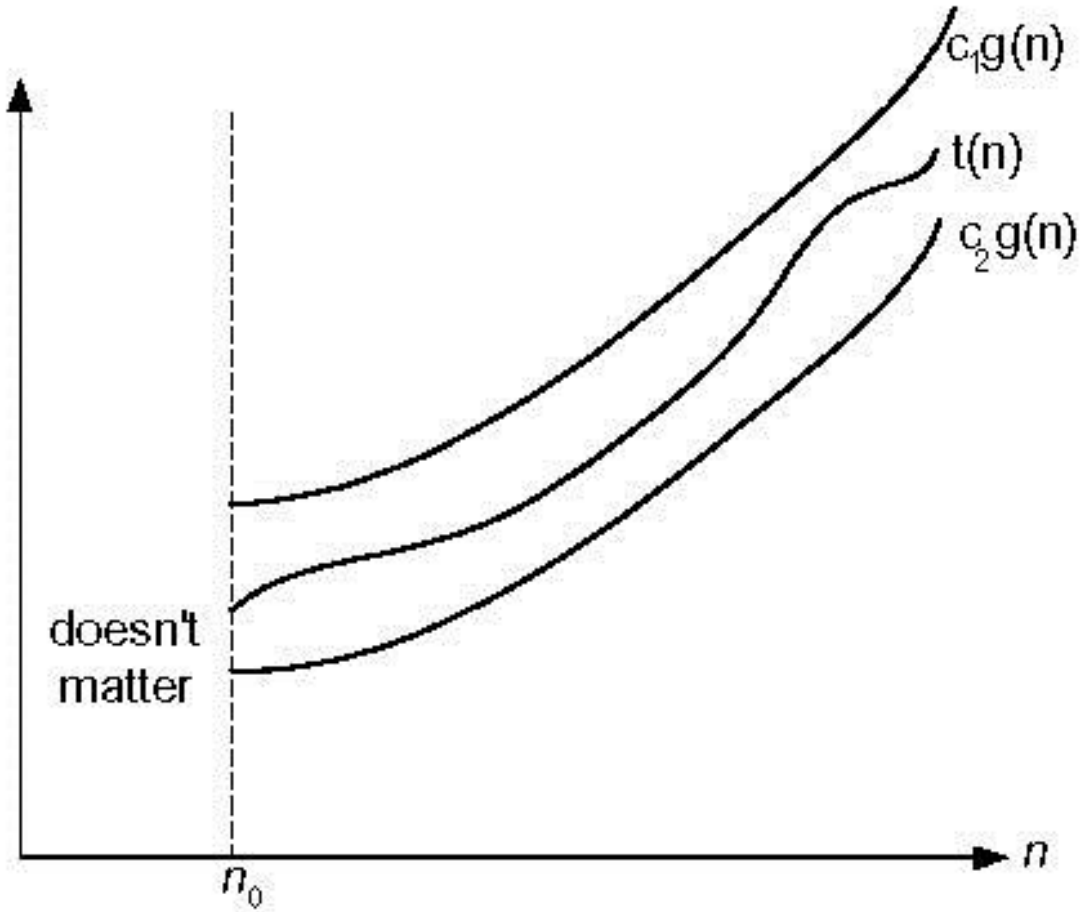


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

Definition: $f(n)$ is 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 $O(n^2)$

- $5n+20$ is $O(n)$

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$

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

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

  Note similarity with $a \leq \mathrm{b}$

- 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

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

Examples:

- $10n$ **vs.** n^2

- $n(n-1)/2$ **vs.** n^2

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^k (k>0) < order a^n < order $n!$ < order n^n

Basic asymptotic efficiency classes

1	constant	Usually best-case efficiencies
$\log n$	logarithmic	Typically a result of cutting a problem's size by a constant factor on each interaction of the algorithm.
n	linear	Scan a list of size n
$n \log n$	*n*-log-*n*	Divide-and-conquer algorithms
n^2	quadratic	Two embedded loops
n^3	cubic	Three embedded loops
2^n	exponential	Typically generates all subsets of an n-element set (candidates of solution)
$n!$	factorial	Typical for algorithms that generate all permutations of an n-element set

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

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

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

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

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

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i$

$\Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

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
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

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 $maxval$

$$C(n) = \Sigma_{1 \le i \le n-1} 1 = n - 1 \in \Theta(n)$$

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

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

$C_{worst}(n)$: when there is no equal elements **or** when the last two elements are the only pair of equal elements

$C_{worst}(n)$ = $\Sigma_{0\le i\le n-2}\ \Sigma_{i+1\le j\le n-1}\ 1 = \Sigma_{0\le i\le n-2}\ (n-1-i) = (n-1)n/2$ $\in \Theta(n^2)$

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 $i \leftarrow 0$ **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

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
```

$$\Sigma_{0\le i\le n-1} \Sigma_{0\le j\le n-1} \Sigma_{0\le k\le n-1} 1 = \Sigma_{0\le i\le n-1} \Sigma_{0\le j\le n-1}\ n = \Sigma_{0\le i\le n-1}\ n^2$$
$$= n^3$$
$$\in \Theta(n^3)$$

Example 4: Gaussian elimination

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

//Implements Gaussian elimination of 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 time efficiency class of this algorithm

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
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

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
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

$$C(n) = \lfloor \log_2 n \rfloor + 1$$

Time efficiency of **recursive** algorithms

Example 1: Recursive evaluation of n!

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

Recursive def. of n!: $\begin{cases} F(n) = F(n\text{-}1) * n \text{ for } n \geq 1 \text{ and} \\ F(0) = 1 \end{cases}$

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:
Recurrence relation:

Solving the **recurrence** for T(n) by backward substitutions

$$\begin{cases} T(n) = T(n\text{-}1) + 1; \; n \geq 1 \\ T(0) = 0 \end{cases}$$

T(n) = T(n-1) + 1 substitute T(n-1) = T(n-2)+1

= [T(n-2)+1]+1 = T(n-2)+2 substitute T(n-2)

= [T(n-3)+1]+2 = T(n-3)+3

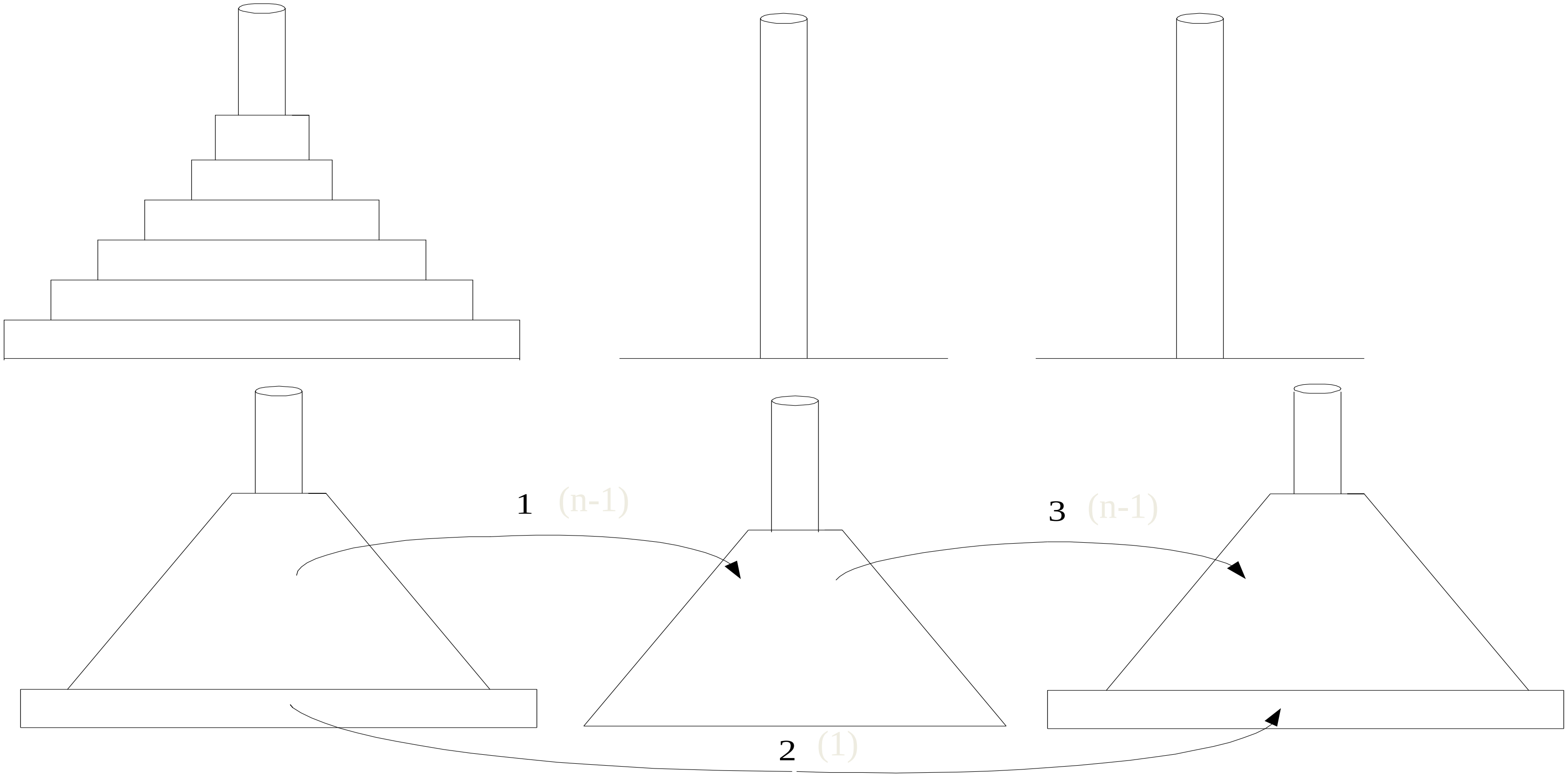
…

= T(n-i)+i = ….T(n-n)+n **= n**

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 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

Solving recurrence for number of moves

$$\begin{cases} M(n) = 2M(n\text{-}1) + 1 \; ; \; n>1 \\ M(1) = 1 \end{cases}$$

By backward substitutions:

Solving recurrence for number of moves

$$\begin{cases} M(n) = 2M(n-1) + 1 \; ; n>1 \\ M(1) = 1 \end{cases}$$

By backward substitutions:

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

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

…

$M(n) = 2^i\, M(n-i)+2^{i-1}+2^{i-2}+\ldots+2+1 = 2^i\, M(n-i)+2^i-1$

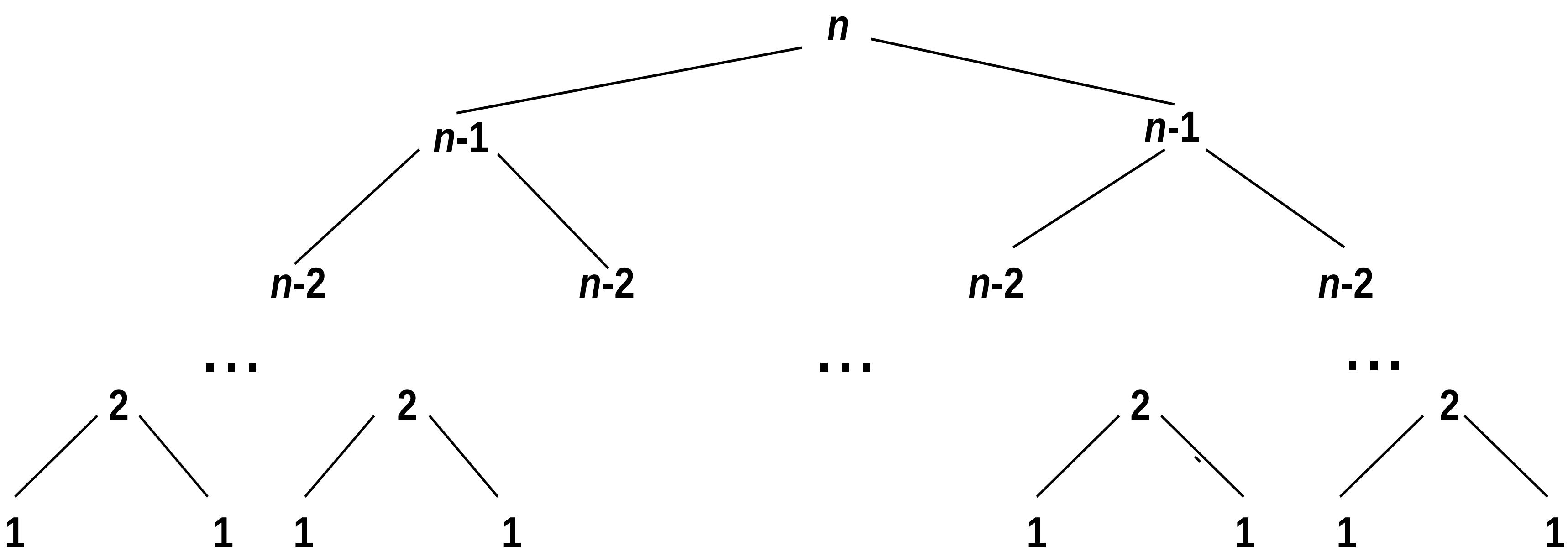
When $i= n-1 \Rightarrow n=1$*, initial condition:*

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

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

$= 2^{n-1}+2^{n-1} -1 = 2.2^{n-1} - 1= 2^n -1 \in \Theta(2^n)$

Tree of calls for the Tower of Hanoi Puzzle



Number of calls made = number of nodes of the tree

$$\mathbf{C(n)} = \sum_{0 \le k \le n-1} 2^k = 2^n - 1 \in \Theta(2^n)$$

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