Algorithms

Chapter 2.1, 2.2, 2.3, 2.4, Appendix B

ROAD MAP

- **Analysis of algorithms**
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality

- Approaches:
  - theoretical analysis
  - empirical analysis

Analysis of Algorithms

- Study complexity of an algorithm
  - How good is the algorithm?
  - How is it when compared with other algorithms?
  - Is it the best that can be done?

Analysis of Algorithms

- Complexities
  - Space
    - Number of bits
    - Number of elements
  - Time
    - Number of operations
      - Depends on model
      - RAM

Run-Time Analysis of Algorithms

- Algorithm complexity is investigated as a function of some parameter *n* indicating problem's size

- Time complexity, T(*n*), is can be computed as the number of times the algorithm's most important operation -- called its basic operation -- is executed

- Space complexity, S(*n*), is usually computed as the size of memory space used during an execution of the algorithm

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

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

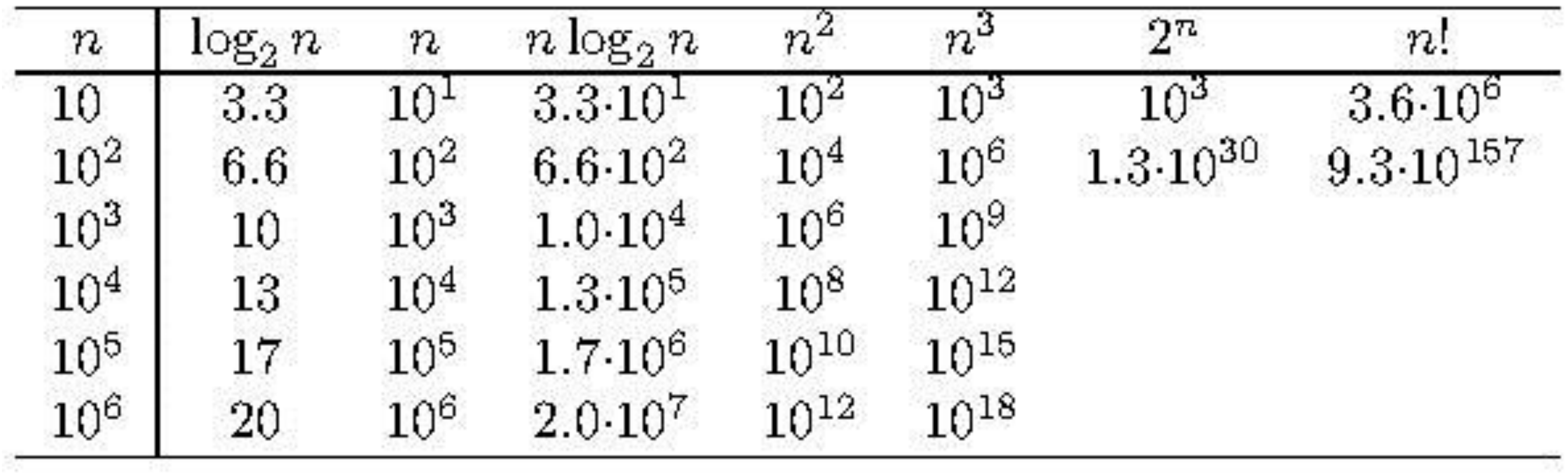
  e.g., $C(n) \approx cn^2$

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

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

Types of Complexities

- Worst case

$$T(n) = max_{|I|=n}\{T(I)\}$$

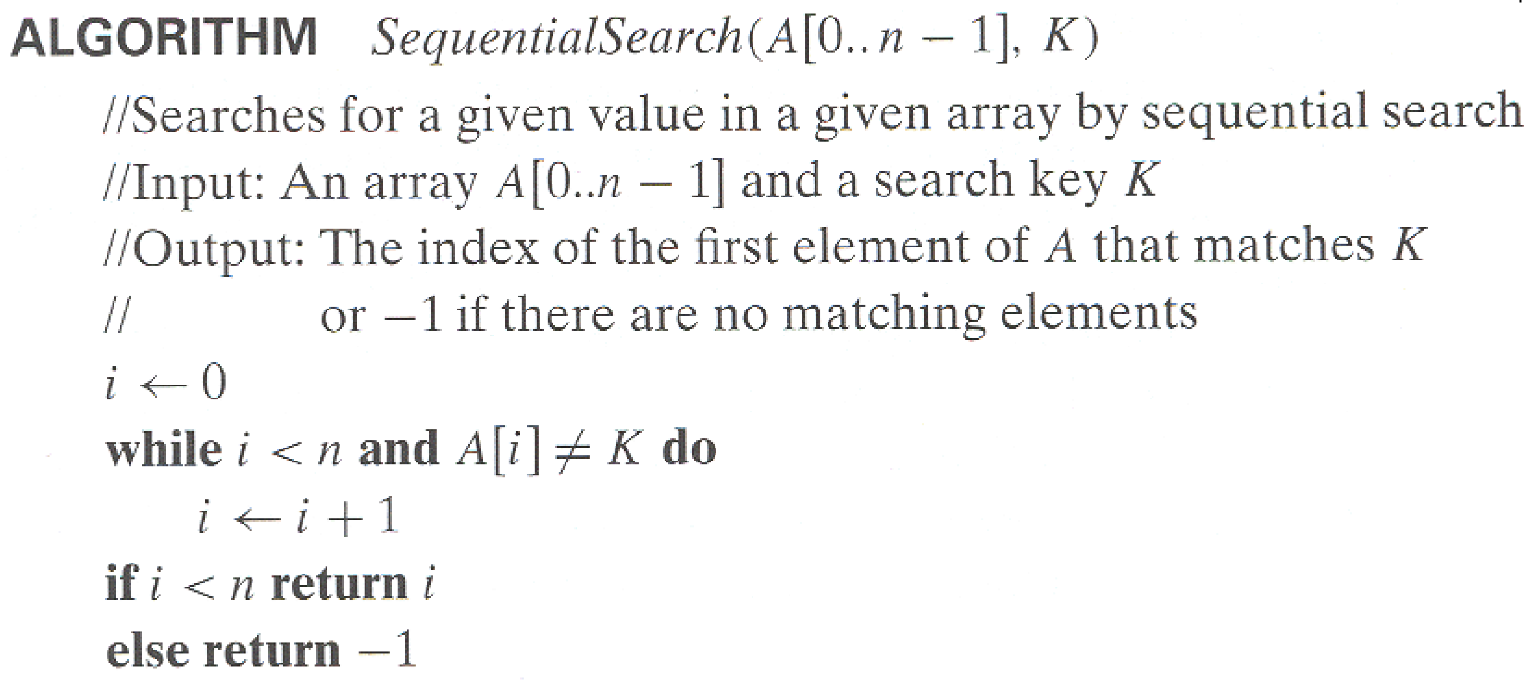
- Average case

$$T(n) = \sum_{|I|=n} T(I).\Pr ob(I)$$

- Best case

$$T(n) = min_{|I|=n}\{T(I)\}$$

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
- Best case
- Average case

Sequential search

Algorithm Complexity:

- Best case

  A[1] = key
- Worst case

  A[i] ≠ key   for any key
  - time is proportional to the number of elements
  - time complexity of linear search is O(n)
- Average case ?
  - if any key is equally likely ~ n/2

ROAD MAP

- Analysis of algorithms
- **Running time functions**
- Mathematical Analysis of Nonrecursive Algorithms
- Mathematical Analysis of Recursive Algorithms
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

Running Time Functions

- Definition

  A nondecreasing function is called ***running time function*** if

  $f: Z^+ \rightarrow R$ such that $f(n) > 0$ for all $n \geq m$ where m is some positive integer

  $Z^+ = \{ 1, 2, 3, \ldots \}$

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*)

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

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

Asymptotic notations

O notation

- Definition

  Let f and g are running time functions. We denote ***f(n) = O(g(n))*** if there exists a real constant c and integer m such that

  $$f(n) \le c\,(g(n)) \quad \text{for all } n \ge m$$

Big-oh

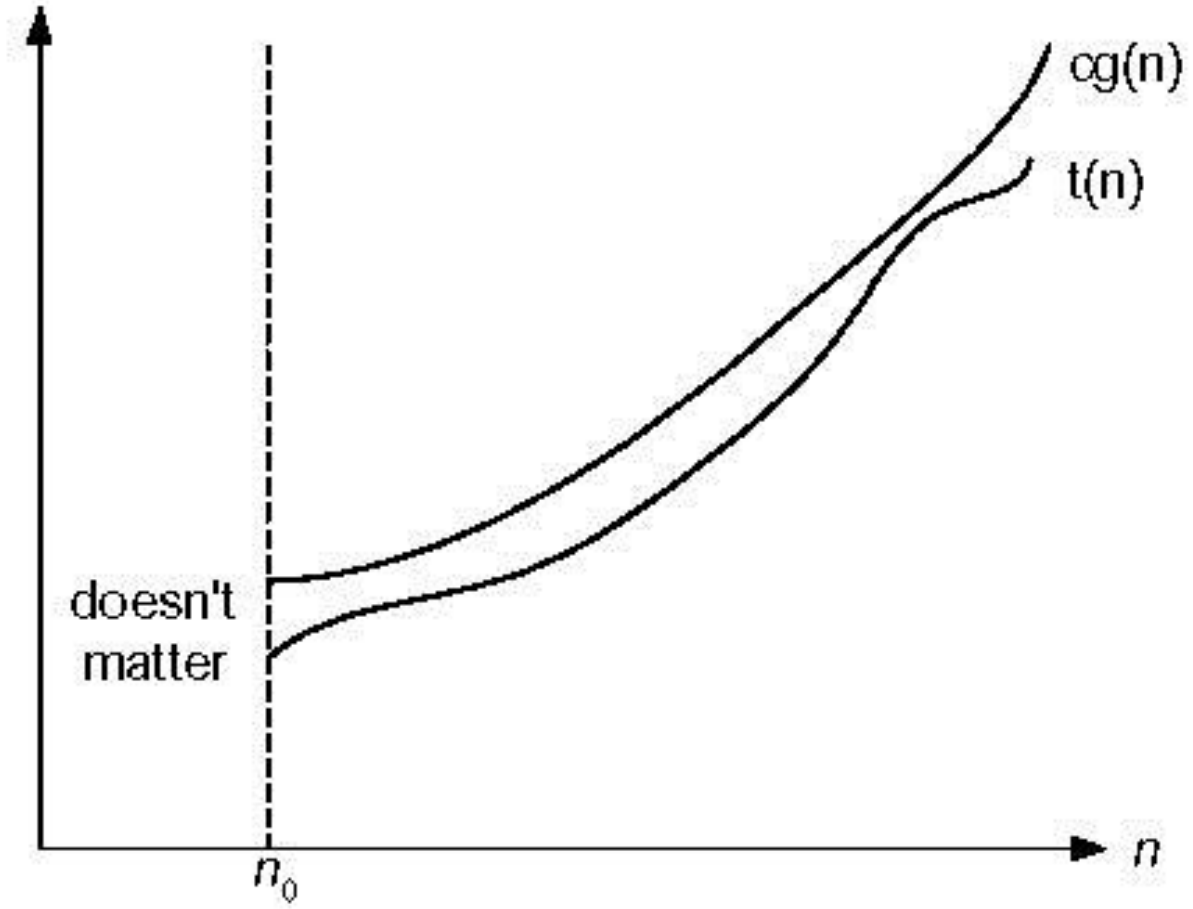


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

O notation

- Ex:
  $7n + 5 = O(n)$

- Ex:
  $10n^2 + 4n + 2 = O(n^2)$

- Ex:
  $7n + 5 = O(n^2)$

- Ex
  $7n + 5 \neq O(1)$

Asymptotic notations

Ω notation

- Definition

  Let f and g are running time functions. We denote ***f(n) = Ω(g(n))*** if there exists a real constant c and integer m such that

  f(n) ≥ c (g(n)) for all *n≥m*

Big-omega

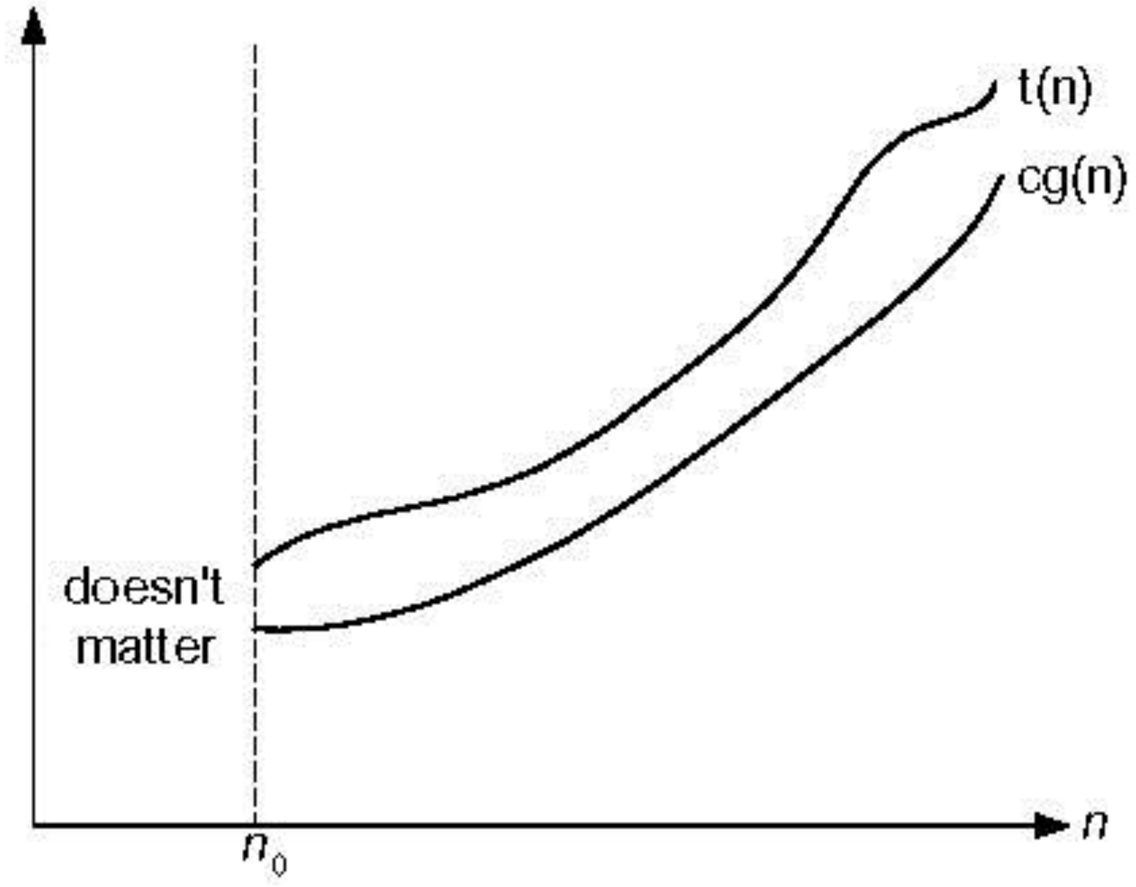


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Ω notation

- Ex:

  $3n + 2 = \Omega(n)$

- Ex:

  $6.2^n + n^2 = \Omega(2^n)$

- Ex:

  $3n - 7 = \Omega(1)$

Asymptotic notations

θ notation

- Definition

Let f and g are running time functions. We denote ***f(n) = θ(g(n))*** if there exists real constants c_1 and c_2 and integer m such that

$$c_2\, g(n) \le f(n) \le c_1\, g(n) \quad \text{for all } n \ge m$$

Big-theta

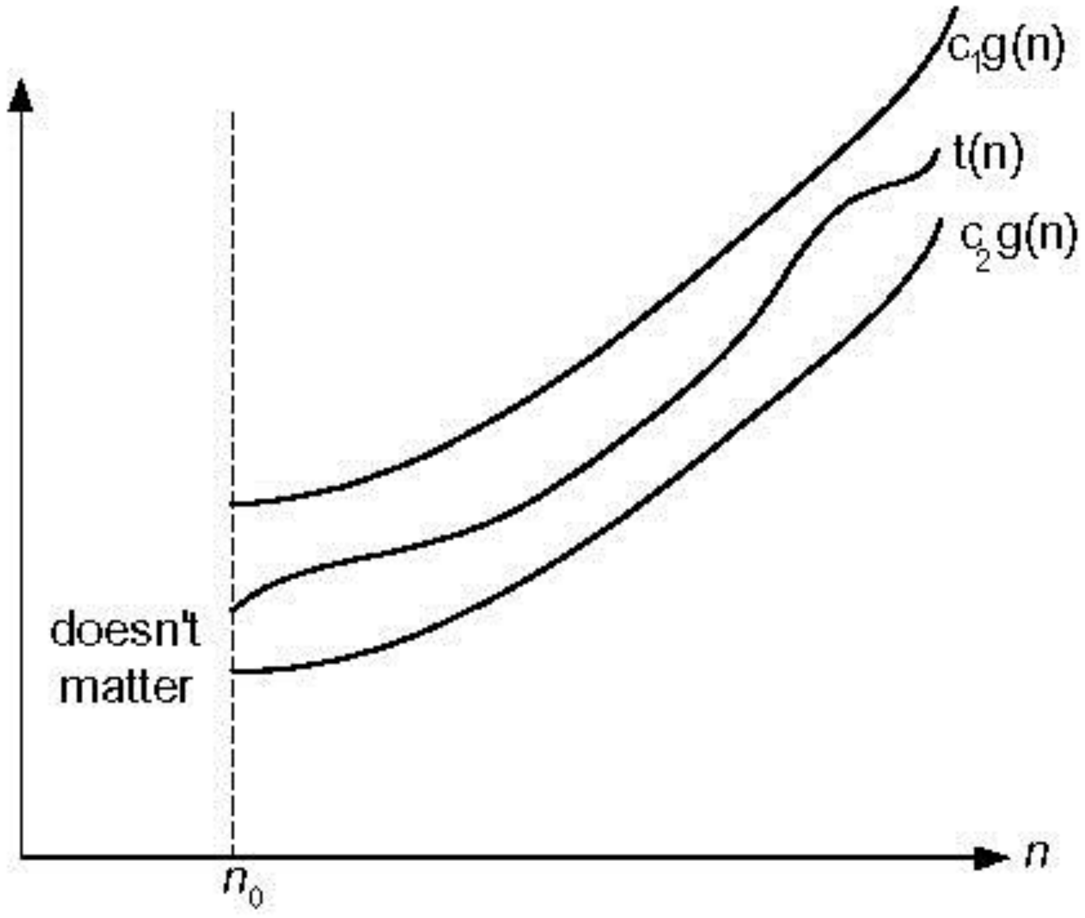


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

θ notation

- Ex:

  $3n + 2 = \theta(n)$

- Ex:

  $10 \log n + 4 = \theta(\log n)$

- Ex:

  $3n + 2 \neq \theta(1)$

Asymptotic notations

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$
- $f(n) = O(g(n))$
  - *g* is an upper bound of *f*
  - *f* grows no faster than *g*
- How tight is this bound ?
  - $n=O(n^2)$
  - $n=O(2^n)$
- $f(n) = O(g(n)) \rightarrow g(n) = O(f(n))$?

Some Rules

- Transitivity

$$f(n) = O(g(n)) \quad \& \quad g(n) = O(h(n)) \quad \Rightarrow \quad f(n) = O(h(n))$$

- Addition

$$f(n) + (g(n)) = O(\max\{f(n), g(n)\})$$

- Polynomials

$$a_0 + a_1 n + \ldots + a_d n^d = O(n^d)$$

Some Rules

- θ is equivalence notation

$$f(n) = \theta(f(n))$$

$$f(n) = \theta(g(n)) \quad \Rightarrow \quad g(n) = \theta(f(n))$$

$$f(n) = \theta(g(n))\ \&\ g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$$

Some Rules

$$f_1(n) = \theta(g(n)) \,\&\, f_2(n) = \theta(g(n))$$
$$\Rightarrow f_1(n) + f_2(n) = \theta(g(n))$$

$$f_1(n) = \theta(g_1(n)) \,\&\, f_2(n) = \theta(g_2(n))$$
$$\Rightarrow f_1(n) * f_2(n) = \theta(g_1(n) * g_2(n))$$

Establishing order of growth using limits

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

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
- 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

ROAD MAP

- Analysis of algorithms
- Running time functions
- **Mathematical Analysis of Nonrecursive Algorithms**
- Mathematical Analysis of Recursive Algorithms
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

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)

Properties of Logarithms

1. $\log_a 1 = 0$
2. $\log_a a = 1$
3. $\log_a x^y = y \log_a x$
4. $\log_a xy = \log_a x + \log_a y$
5. $\log_a \dfrac{x}{y} = \log_a x - \log_a y$
6. $a^{\log_b x} = x^{\log_b a}$
7. $\log_a x = \dfrac{\log_b x}{\log_b a} = \log_a b \log_b x$

Important Summation Formulas

1. $\sum_{i=l}^{u} 1 = \underbrace{1 + 1 + \cdots + 1}_{u-l+1 \text{ times}} = u - l + 1$ (l, u are integer limits, $l \leq u$); $\sum_{i=1}^{n} 1 = n$

2. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2$

3. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{1}{3}n^3$

4. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + n^k \approx \frac{1}{k+1}n^{k+1}$

Important Summation Formulas

5. $\sum_{i=0}^{n} a^i = 1 + a + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1} \; (a \neq 1); \quad \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

6. $\sum_{i=1}^{n} i2^i = 1 \cdot 2 + 2 \cdot 2^2 + \cdots + n2^n = (n-1)2^{n+1} + 2$

7. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \ln n + \gamma$, where $\gamma \approx 0.5772 \ldots$ (Euler's constant)

8. $\sum_{i=1}^{n} \lg i \approx n \lg n$

Sum Manipulation Rules

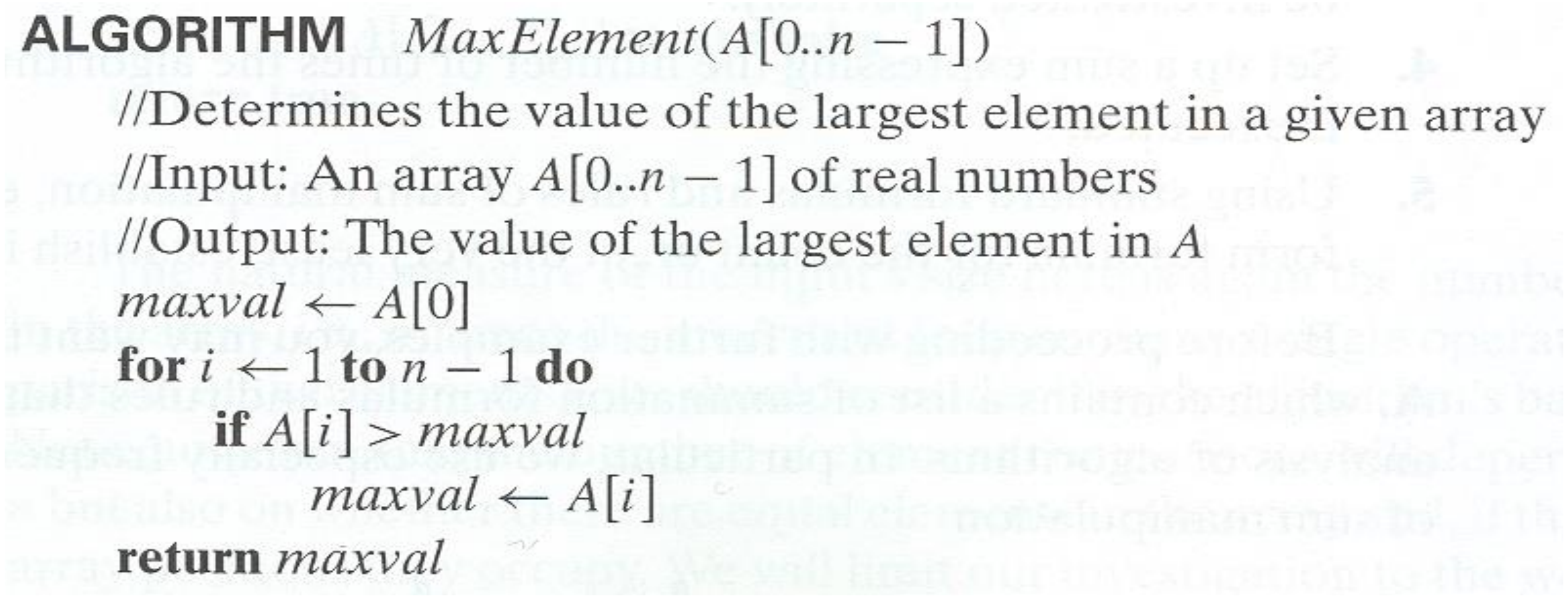
1. $$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$$

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

3. $$\sum_{i=l}^{u} a_i = \sum_{i=l}^{m} a_i + \sum_{i=m+1}^{u} a_i, \text{ where } l \le m < u$$

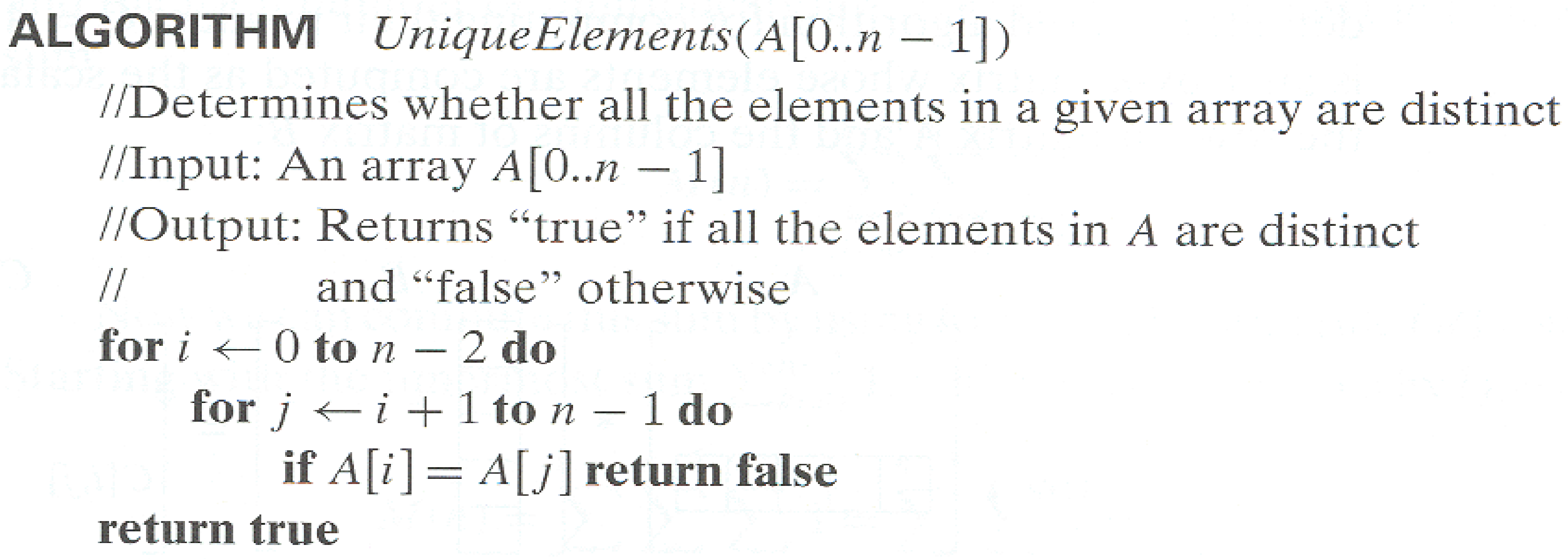
4. $$\sum_{i=l}^{u} (a_i - a_{i-1}) = a_u - a_{l-1}$$

Example : Maximum element



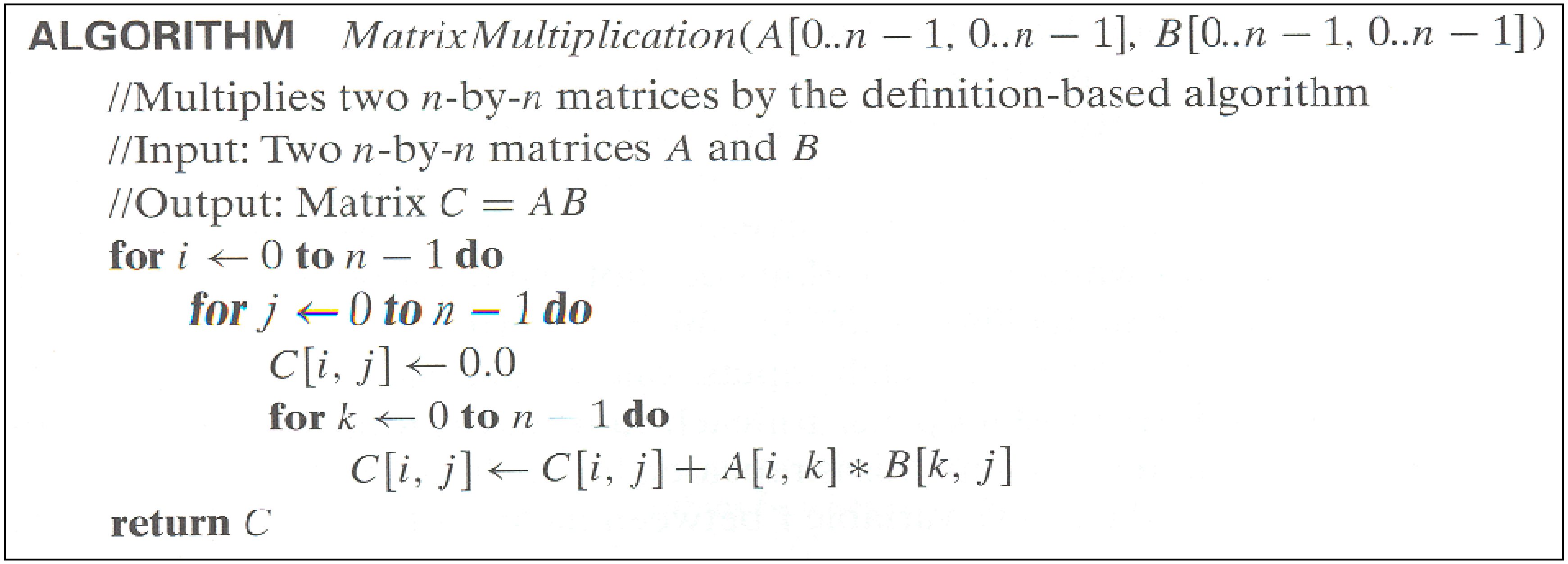
```
ALGORITHM  $MaxElement(A[0..n-1])$
    //Determines the value of the largest element in a given array
    //Input: An array $A[0..n-1]$ of real numbers
    //Output: The value of the largest element in $A$
    $maxval \leftarrow A[0]$
    for $i \leftarrow 1$ to $n-1$ do
        if $A[i] > maxval$
            $maxval \leftarrow A[i]$
    return $maxval$
```

Example : Element uniqueness problem



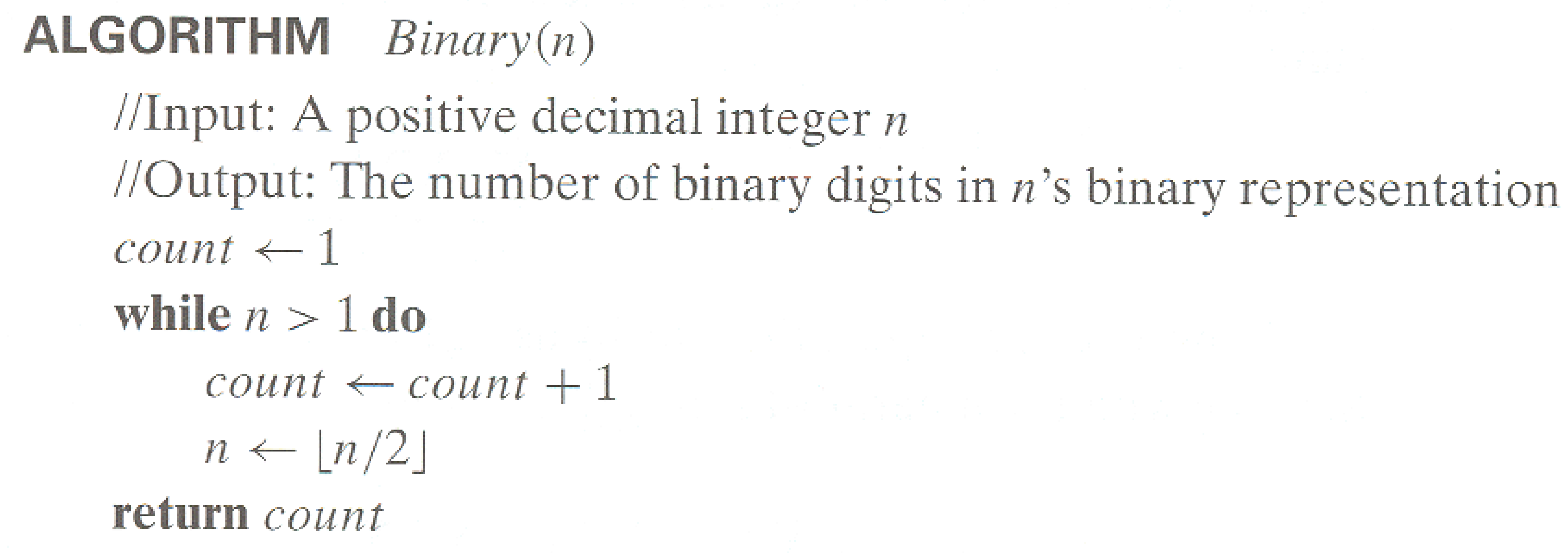
```
ALGORITHM  $UniqueElements(A[0..n-1])$
    //Determines whether all the elements in a given array are distinct
    //Input: An array $A[0..n-1]$
    //Output: Returns “true” if all the elements in $A$ are distinct
    //             and “false” otherwise
    for $i \leftarrow 0$ to $n-2$ do
        for $j \leftarrow i+1$ to $n-1$ do
            if $A[i] = A[j]$ return false
    return true
```

Example : 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 : 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.

ROAD MAP

- Analysis of algorithms
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- **Mathematical Analysis of Recursive Algorithms**
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

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.

Recurrence Relations

- **Solution Methods**
  - **Exact**
    - Forward substitution
    - Backward substitution
  - **Asymptotic**
    - Master theorem

Example 1 – Factorial Calculation

- Goal:

  Computing the factoral function **`F(n)=n!`** for an arbitrary non-negative integer **`n`**.

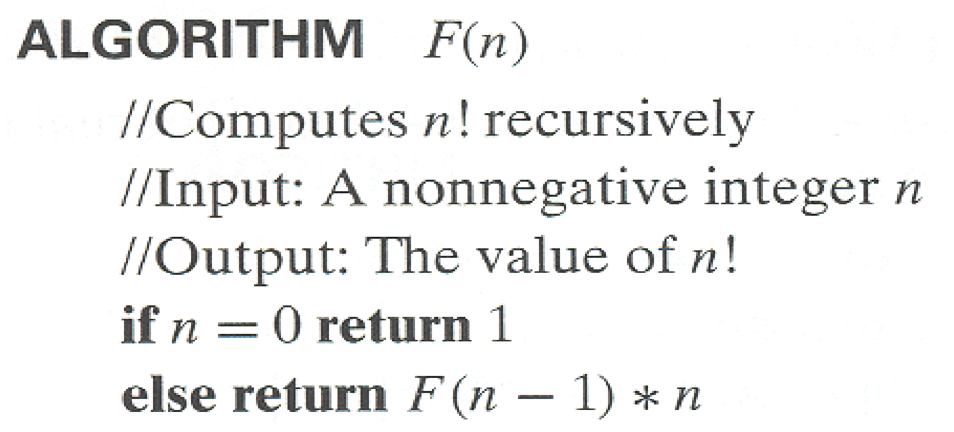
  Since

$$n! = n\times(n-1)\times\ldots\times 1 = n\times(n-1)! \quad \text{for} \quad n\geq 1 \quad \text{and} \quad 0! = 1$$

  by definition.

$$F(n) = F(n-1)\times n \quad \text{for} \quad n>0$$
$$F(0) = 1$$

Example : Recursive evaluation of *n*!



```
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:

Factorial Calculation

<u>Algorithm :</u>

```
if n=0 return 1
else return F(n-1)*n
```

<u>Analysis:</u>

$$M(n) = \underbrace{M(n-1)}_{\text{to calculate } F(n-1)} + \underbrace{1}_{\text{to multiply } F(n-1) \text{ by } n}$$

$$M(0) = 0$$

ROAD MAP

- Analysis of algorithms
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- **Mathematical Analysis of Recursive Algorithms**
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

Forward Substitution

- Start with the initial condition
- Generate first few terms of the solution
  - Using the recurrence equation and values of previous terms
- Try to guess a pattern
- Form a closed-form formula
- Check the validity of the formula
  - Using induction or
  - Substituting in the recurrence equation and initial condition

Factorial Function

Using forward substitution:

$$M(0) = 0$$

$$M(1) = M(0) + 1 = 1$$

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

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

$$\mathrm{M}$$

$$M(n) = n$$

Need to prove the resulting formula

ROAD MAP

- Analysis of algorithms
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- **Mathematical Analysis of Recursive Algorithms**
    - Exact Solution
        - Forward substitution
        - Backward substitution
    - Asymptotic Solution
        - Master theorem

Backward Substitution

- Start with the recurrence equation
- Substitute *f(n-1)*
  - with its value using recurrence equation
- Perform similar substitution few more times
  - for *f(n-2)*, *f(n-3)* etc..
- Try to guess a pattern for *f(n)* in terms of *f(n-i)*
- Check the validity of the pattern
  - usually using induction
- Pick an *i* which makes *n-i* to reach the initial condition
- Obtain a closed-form formula

Factorial Function

Using back substitution:

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

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

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

$$\mathrm{M}$$

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

$$\text{for } n = i$$

$$M(n) = M(0)+n = n$$

Example 2- Towers of Hanoi

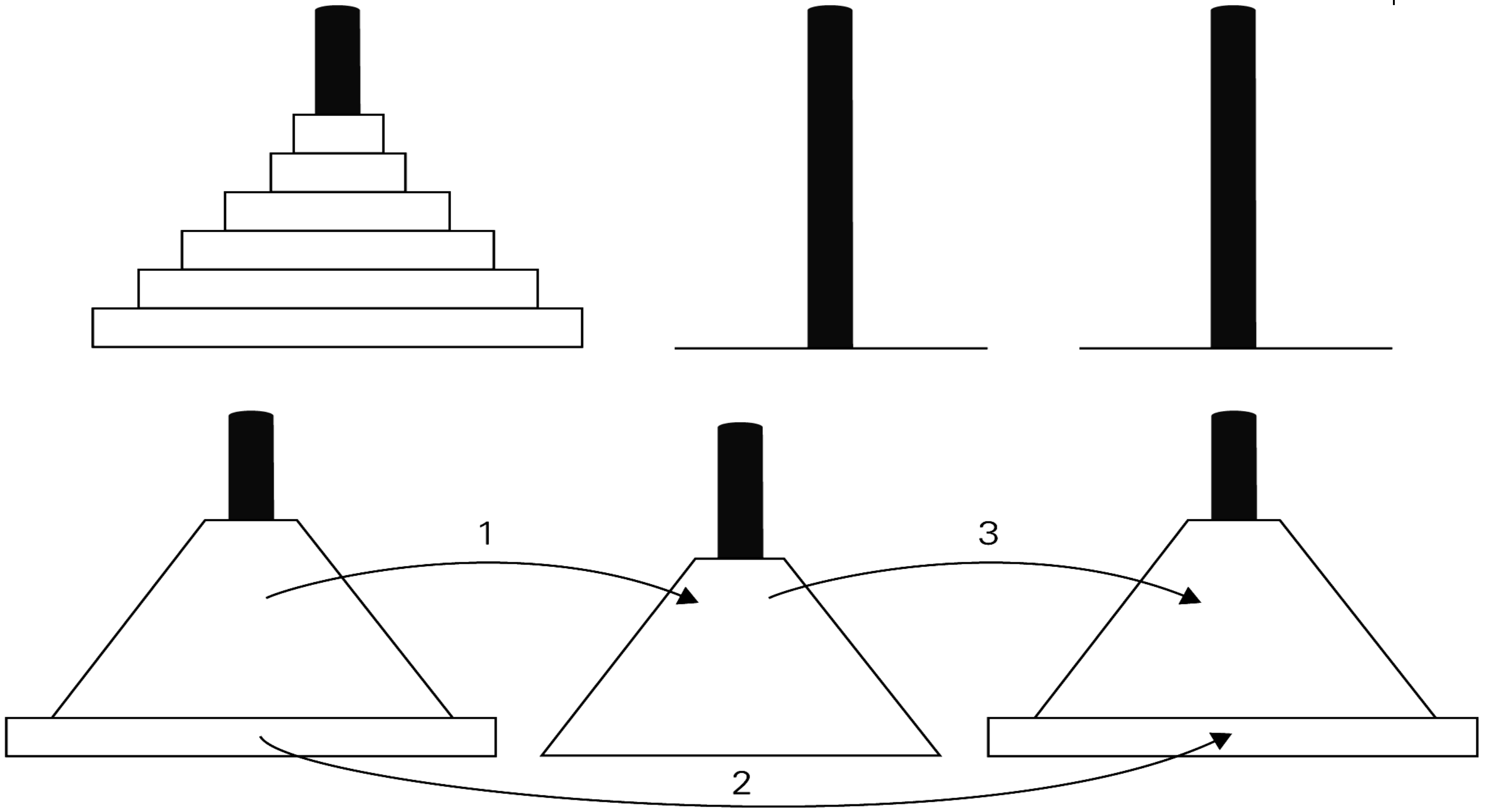


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle

Example 2- Towers of Hanoi

<u>Goal:</u>

Transfer *n* disks from peg *A* to peg C using peg B

<u>Approach:</u> (recursive)

- transfer n-1 disks from *A* to *B* using *C*
- move largest disk from *A* to *C*
- transfer n-1 disks from *B* to *C* using *A*

Total number of moves

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

`T(1) = 1`

Tree of calls for the Tower of Hanoi Puzzle

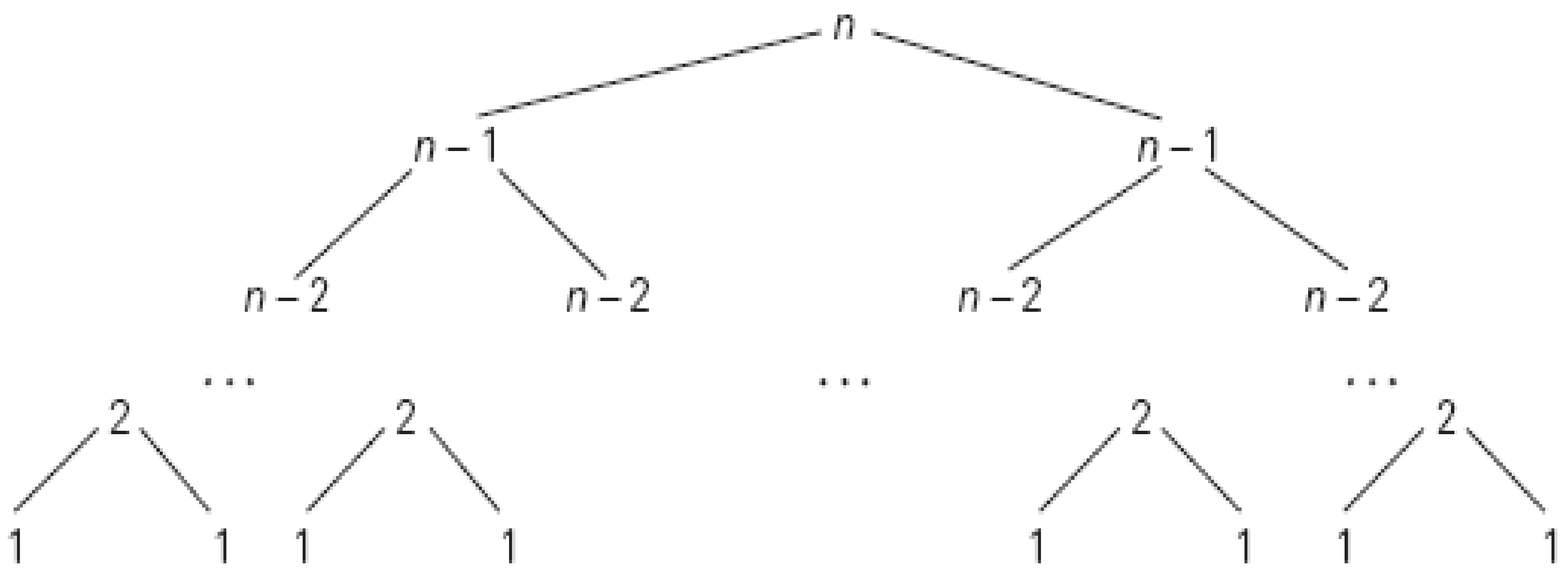


FIGURE 2.5 Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.

Towers of Hanoi

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

$$T(1) = 1$$

Solution by backward substitution

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

$$T(n) = 4\left(T(n-2)\right) + 2 + 1$$

$$T(n) = 4\left(2T(n-3) + 1\right) + 2 + 1$$

$$T(n) = 8T(n-3) + 4 + 2 + 1$$

M

$T(n) = 2^i T(n-i) + 2^{i-1} + 2^{i-2} + \ldots + 2^1 + 1$ **← needs proving**

$$when \qquad i = n-1$$

$$T(n) = 2^{n-1} T(1) + 2^{n-2} + \ldots + 2^1 + 1$$

$$T(n) = 2^n - 1 = \theta(2^n)$$

ROAD MAP

- Analysis of algorithms
- Running time functions
- Mathematical Analysis of Nonrecursive Algorithms
- **Mathematical Analysis of Recursive Algorithms**
  - Exact Solution
    - Forward substitution
    - Backward substitution
  - Asymptotic Solution
    - Master theorem

Master Theorem

Let T(n) be an eventually nondecreasing function that satisfies the recurrence

$$T(n) = a\,T(n/b) + f(n) \qquad \text{for } n = b^k, k = 1, 2, \dots$$
$$T(1) = c,$$
$$where\ a \geq 1,\ b \geq 2, c > 0.$$
$$If\ f(n) \in \Theta(n^d)\ \ where\ d \geq 0, then$$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Examples

$$T(n) = T(n/2) + c$$

$$T(n) = 2\,T(n/2) + c\,n$$

$$T(n) = 7\,T(n/2) + 18\,n^2$$

$$T(n) = 9\,T(n/3) + n$$

$$T(1) = \varepsilon$$

Common Recurrence Types in Algorithm Analysis

- Decrease-by-One

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

- Decrease-by-a-Constant-Factor

$$T(n) = T\left({}^{n}/_{b}\right) + f(n)$$

- Divide-and-Conquer

$$T(n) = a\,T\left({}^{n}/_{b}\right) + f(n)$$