Module 1
Introduction

Course: Analysis and Design of Algorithms

In-charge: Sudeep Manohar

What is an algorithm?

- An Algorithm is a finite set of instructions that, if followed accomplishes a particular task
- An algorithm is a sequence of unambiguous instructions for solving a problem

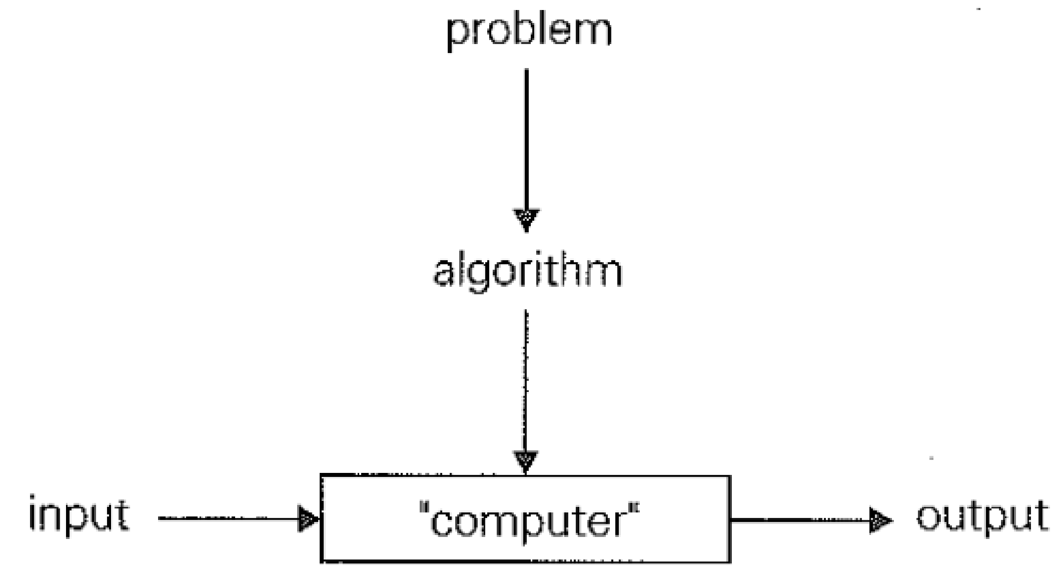


FIGURE 1.1 Notion of algorithm

Properties of an algorithm (or criteria)

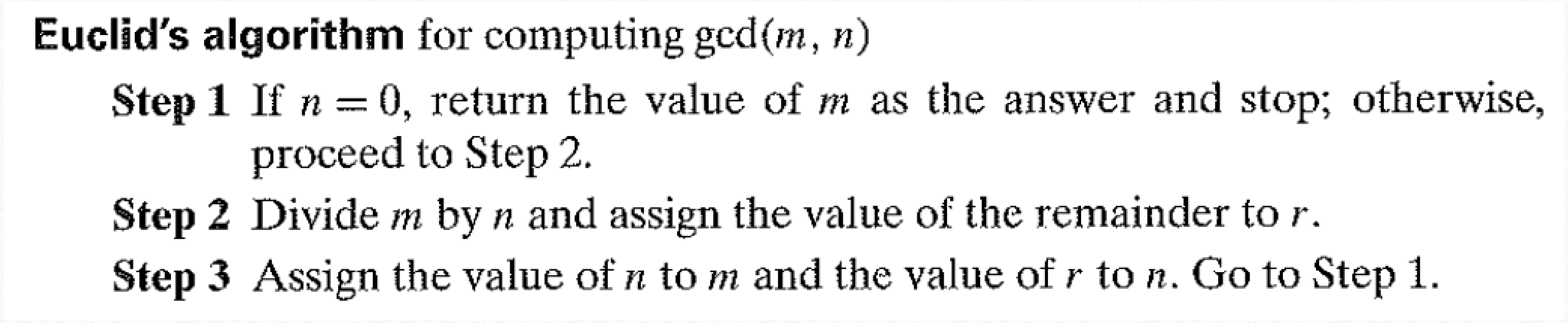
- All algorithms must satisfy the following criteria
  - Input – 0 or more quantities are externally supplied
  - Output – At least one quantity is produced
  - Definiteness – Each instruction is clear and unambiguous
  - Finiteness – The algorithm should terminate after a finite number of steps
  - Effectiveness – Every instruction must be very basic

Example: Finding the GCD of two numbers using Euclid’s algorithm

gcd(m, n) = gcd(n, m mod n)

gcd(m, 0) = m

gcd(60, 24) = gcd(24, 12) = gcd(12, 0) = 12

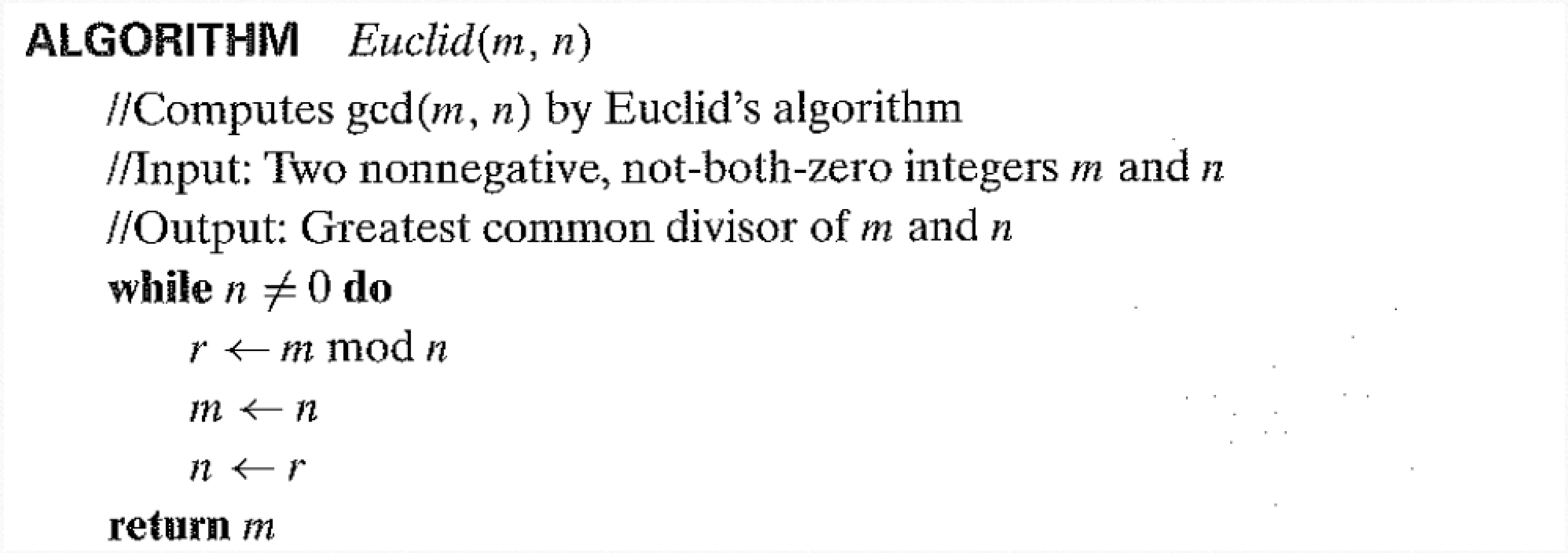


Euclid’s algorithm for computing gcd(m, n)

Step 1 If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 Divide m by n and assign the value of the remainder to r.

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.



```
ALGORITHM  Euclid(m, n)
    //Computes gcd(m, n) by Euclid's algorithm
    //Input: Two nonnegative, not-both-zero integers m and n
    //Output: Greatest common divisor of m and n
    while $n \neq 0$ do
        $r \leftarrow m \bmod n$
        $m \leftarrow n$
        $n \leftarrow r$
    return m
```

Active areas of research in the study of algorithms

- **How to device algorithms?**
  - Design strategies or techniques: Greedy method, Divide and conquer, Dynamic programming, Linear, Non-linear, Backtracking, Branch and Bound etc.
- **How to validate algorithms?**
  - Algorithm validation - Check whether the algorithm computes the correct answer for all possible legal inputs
  - Program verification – Write the program and check the program output

- **How to analyze algorithms?**
  - Algorithm uses CPU and memory
  - Algorithm analysis or performance analysis refers to the task of determining how much computing time and storage an algorithm requires
- **How to test a program?**
  - Debugging – Process of executing programs on sample data sets to determine faulty results and correct them
  - Profiling – Process of executing a correct program on data sets and measuring the time and space its takes to compute the results

Algorithm specification

- Natural language – English
- Graphic representation – Flowcharts
- Pseudocode – Mixture of programming language and English

Example

Definition of Selection sort

- From those elements that are currently unsorted, find the smallest and place it next in the sorted list

Pseudocode

```
for ( i = 1; i <= n; i++ ) {
        Examine a[ i ]  to a[ n ] and suppose the smallest element is at a[ j ];
        Interchange a[ i ] and a[ j ];
    }
```

```
void SelectionSort ( Type a[ ], int n )
//Sort the array a[1:n] into non decreasing order
{
    for ( int i = 1; i <= n; i++ ) {
        int j = i;
        for ( int k = i+1; k <=n; k++ )
            if  ( a[ k ] < a[ j ] ) j = k;
        Type t = a[ i ]; a[ i ] = a[ j ]; a[ j ] = t;
    }
}
```

Recursive algorithms

- Direct recursion
- Indirect recursion

- Ex: Tower of Hanoi and Permutation

Fundamentals of Algorithmic problem solving

- Algorithms are procedural solutions to problems
- These solutions are not answers but specific instructions for getting answers

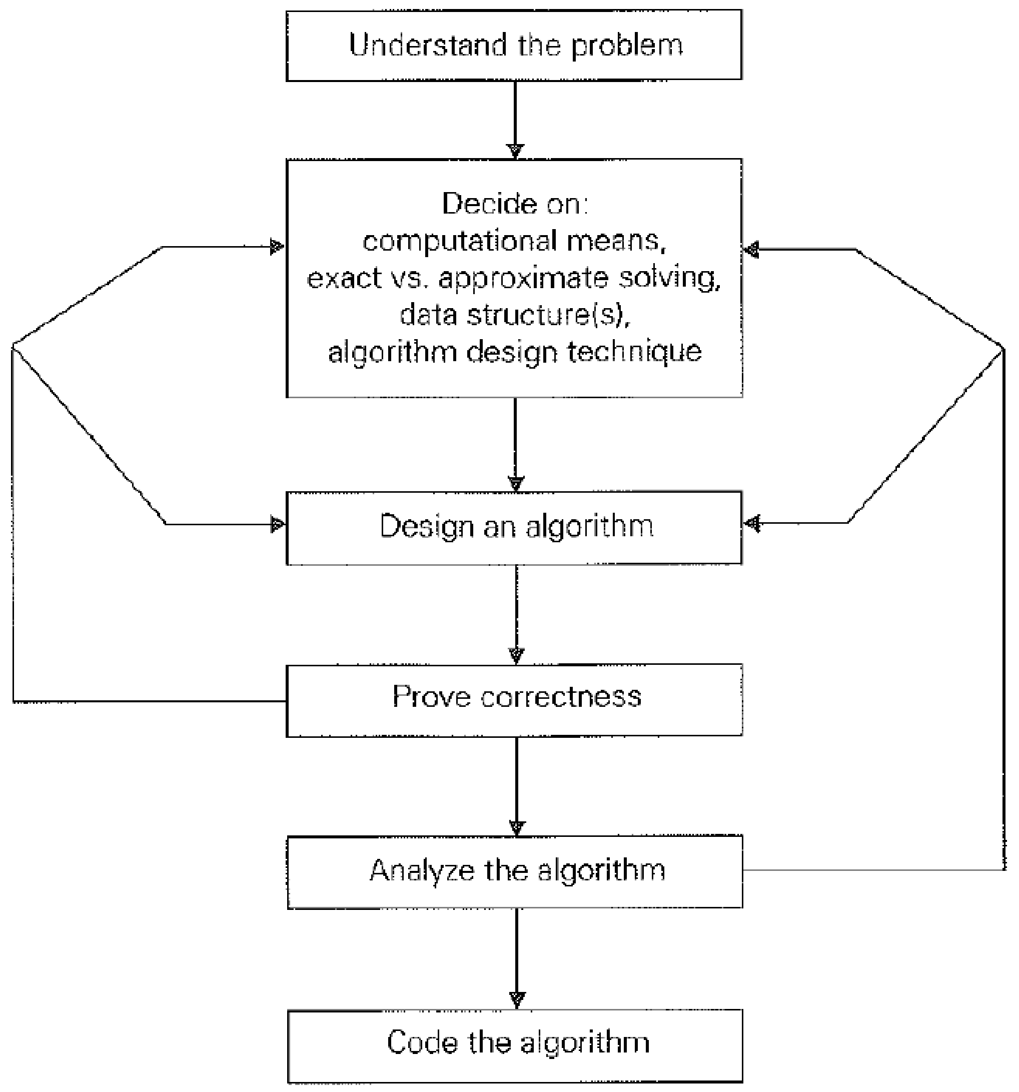

Understand the problem
Decide on: computational means, exact vs. approximate solving, data structure(s), algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm

FIGURE 1.2 Algorithm design and analysis process

Algorithm design and analysis process

1. Understanding the problem
2. Ascertaining the capabilities of a computational device
3. Choosing between exact and approximate problem solving
4. Deciding on appropriate data structures
5. Algorithm design techniques
6. Methods of specifying an algorithm
7. Proving an algorithm's correctness
8. Analyzing an algorithm
9. Coding an algorithm

Analysis Framework

- Time efficiency – indicates how fast an algorithm runs
- Space efficiency – deals with minimum memory required by an algorithm

Measuring an Input’s size

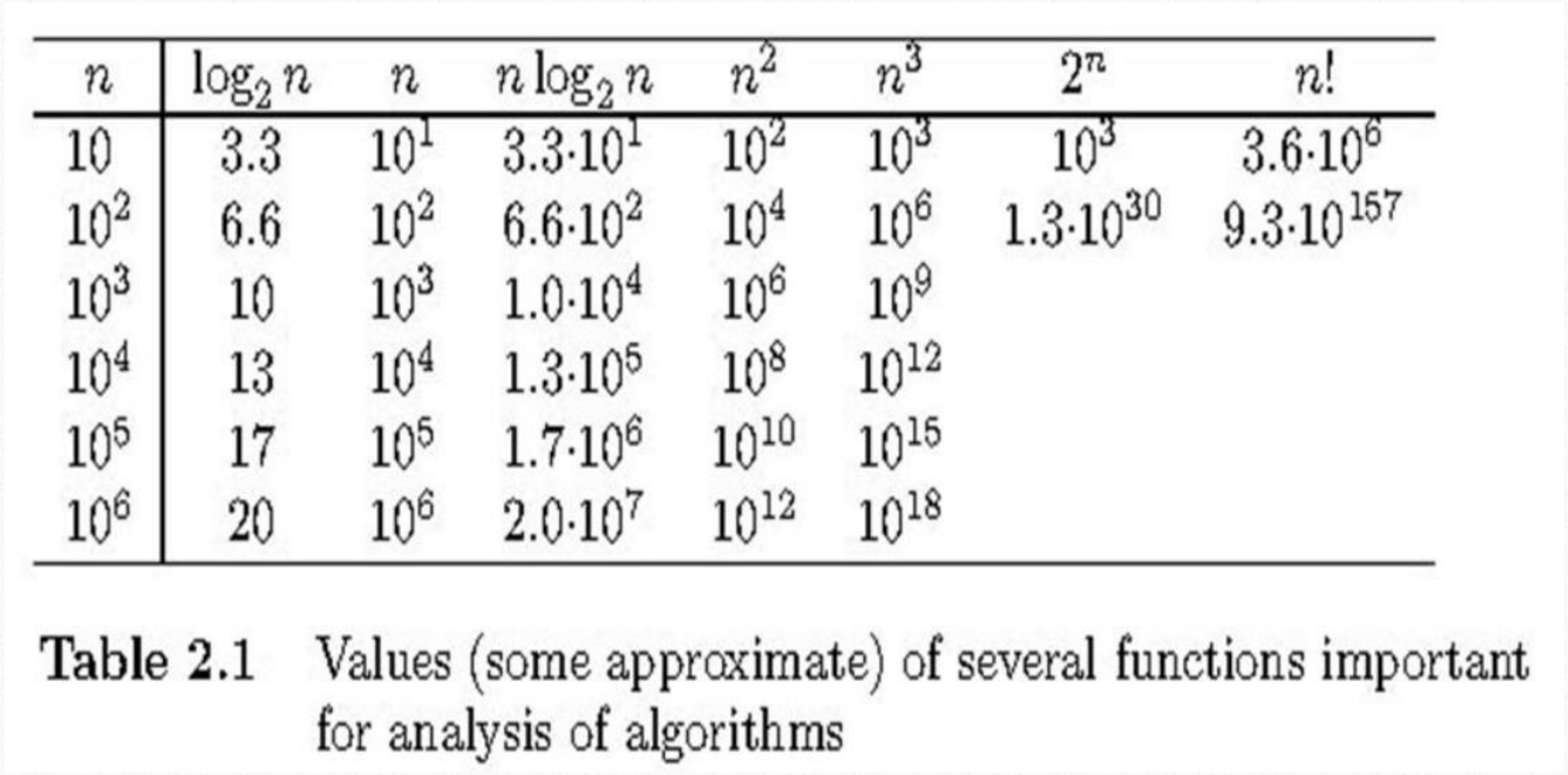
- Algorithm’s efficiency is a function of some parameter n indicating the algorithm’s input size
- n may be
  - ✓ Size of the list for sorting and searching problems
  - ✓ Polynomial’s degree
  - ✓ Order of the matrix
  - ✓ Number of characters
  - ✓ Number of bits in n’s binary representation

Units for measuring running time

- Standard unit of time – Seconds, milliseconds etc.
    - ✓ Depends on the speed of a particular computer
    - ✓ Quality of a program and compiler
- Count the number of times algorithm's operations are executed
    - ✓ Identity the basic operations and count the number of times it is executed
    - ✓ c_{op} – execution time of an algorithm's basic operation
    - ✓ $C(n)$ – number of times the basic operation needs to be executed
    - ✓ $T(n)$ – running time of a program

$$T(n) = c_{op}\, C(n)$$

Orders of growth



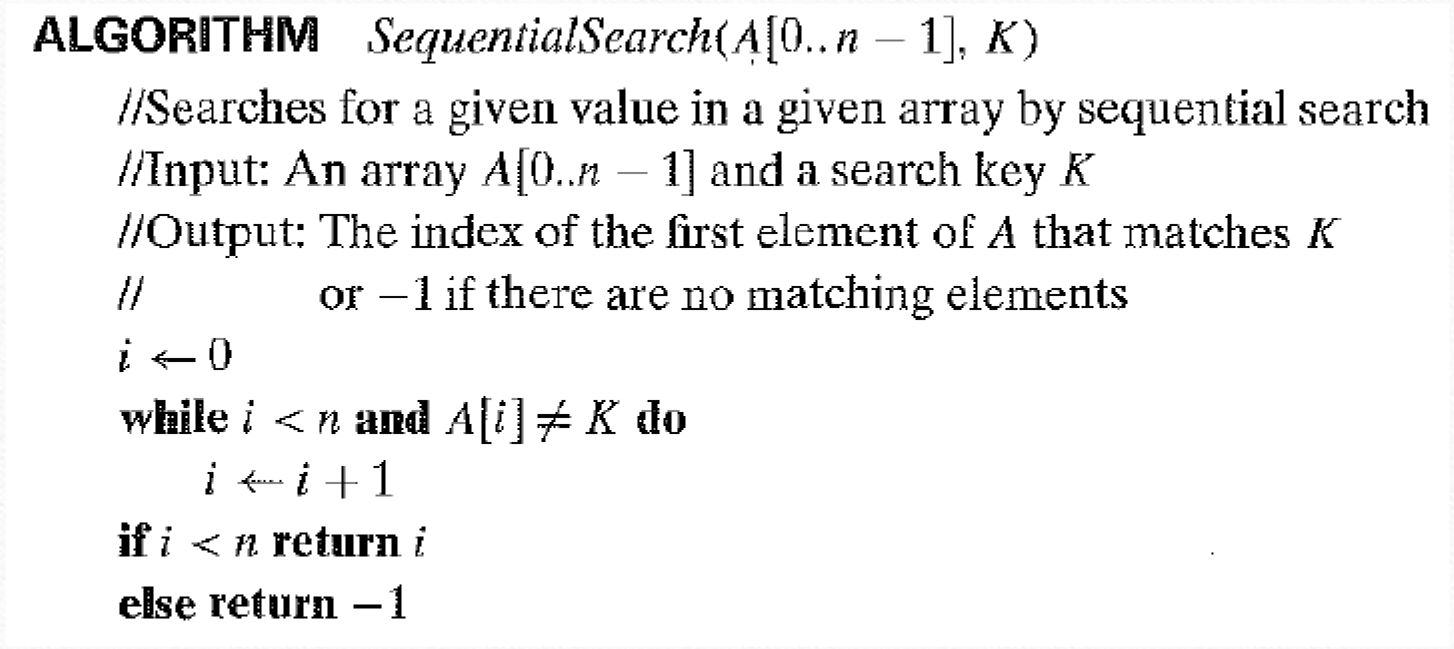
n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	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

- Worst case efficiency – efficiency for the worst-case input of size n for which the algorithm runs the longest indicated by $C_{worst}(n)$
- Best case efficiency – efficiency for the best-case input of size n for which the algorithm runs the fastest, indicated by $C_{best}(n)$
- Average case efficiency – efficiency for the average-case input of size n for which the algorithm runs for average time, indicated by $C_{avg}(n)$
- Sequential search algorithm example
- $C_{worst}(n) = n$
- $C_{best}(n) = 1$
- $C_{avg}(n) = (n+1)/2$

Sequential search algorithm example



```
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
```

- $C_{worst}(n) = n$
- $C_{best}(n) = 1$
- $C_{avg}(n) = (n+1)/2$

Performance analysis

Space complexity

- The space complexity of an algorithm is the amount of memory it needs to run to completion
- The space needed by the programs is the sum of the following components
  - A fixed part that is independent of the characteristics of the inputs and outputs. This part typically includes the instruction space, space for simple variables and fixed size component variables, space for constants etc.
  - A variable part that consists of the space needed by the component variables whose size is dependent of the particular problem instance being solved, space needed by referenced variables and the recursion stack space

- The space requirement S(P) of any algorithm P is

$$S(P) = c + S_P \text{ (instance characteristics)}$$

- Where c is a constant and S_P is the instance characteristics

Example 1

```
float abc(float a, float b, float c)
{    return (a + b + b * c + (a + b - c) / (a + b) + 4.0);   }
```

- Space needed by abc is independent of the instance characteristics
- Hence S_P (instance characteristics) = 0

Example 2

```
float sum(float a[ ], int n)
{     float s = 0.0;
      for (int i=1; i<=n; i++)
            s += a[i];
      return s;
}
```

- The program is characterized by n, the number of elements to be summed
- The space needed by n is one word
- The space needed by the array a is the space needed by n float values, i.e., n words
- Hence S_{SUM} >= n+3 (n for a[ ], one for each n, i and s)

Example 3

```
float RSum(float a[ ], int n)
{     if(n<=0) return (0.0);
      else return (RSum(a, n-1) + a[n]);
}
```

- Instances are characterized by n
- Recursion stack includes space for formal parameters, local variables and return address
- Assume return address requires one word
- Each call to Rsum requires 3 words (for n, return address and pointer to a[ ])
- The depth of the recursion is n+1
- Hence $S_{RSum} >= 3(n+1)$

Time complexity

Based on execution time

- The time complexity of an algorithm is the amount of computer time it needs to run to completion
- The time T(P) taken by a program P is the sum of the compile time and the run time
- Compile time doesn't depend on the instance characteristics
- Compiled program may be run several times without recompilation
- Hence only run time is concerned which is denoted by

  t_P (instance characteristics)

$$t_P(n) = c_a ADD(n) + c_s SUB(n) + c_m MUL(n) + c_d DIV(n) + \ldots$$

- ca, cs, cm and cd are time needed for addition, subtraction, multiplication and division
- ADD, SUB, MUL and DIV are the number of additions, subtraction, multiplication and division

Based on number of program steps

Method 1

- A program step is defined as meaningful segment of a program that has an execution time that is independent of the instance characteristics
- Ex: return (a + b + b*c + (a+b-c)/(a+b) +4.0); is considered one step
- Comments and declarative statements count as zero steps
- Assignment statement is considered one step
- For iterative statement, a step count is considered for each iteration
- Consider a global variable count and increment whenever a program step is encountered

```
float sum(float a[ ], int n)
{     float s = 0.0;
      count++;
      for (int i=1; i<=n; i++)
      {
            count++;     //For 'for'
            s += a[i];
            count++;     //For assignment
      }
      count++;           //For the last iteration of 'for'
      count++;           //For return
      return s;
}
```

Each invocation of sum executes a total of 2n+3 steps

```
float RSum(float a[ ], int n)
{
      count++;         //for if  condition
      if(n<=0)
      {
            count++;   //for return
            return (0.0);
      }
      else
      {
            count++;   //for addition, function invocation and return;
            return (RSum(a, n-1) + a[n]);
      }
}
```

- If n=0, $t_{RSum}(0) = 2$
- If n>0, total steps is $2 + t_{RSum}(n-1)$

- Hence the recursive formula for step count is
- $t_{RSum}(n) = \begin{cases} 2 & n = 0 \\ 2 + tRSum(n-1) & n > 0 \end{cases}$
- Recursive formulas are also called recurrence relations

$$\begin{aligned} t_{RSum}(n) &= 2 + t_{RSum}(n-1) \\ &= 2 + 2 + t_{RSum}(n-2) \\ &= 2(2) + t_{RSum}(n-2) \\ &\dots \\ &= 3(2) + t_{RSum}(n-3) \\ &= n(2) + t_{RSum}(n-n) \\ &= n(2) + t_{RSum}(0) \\ &= 2n + 2 \end{aligned}$$

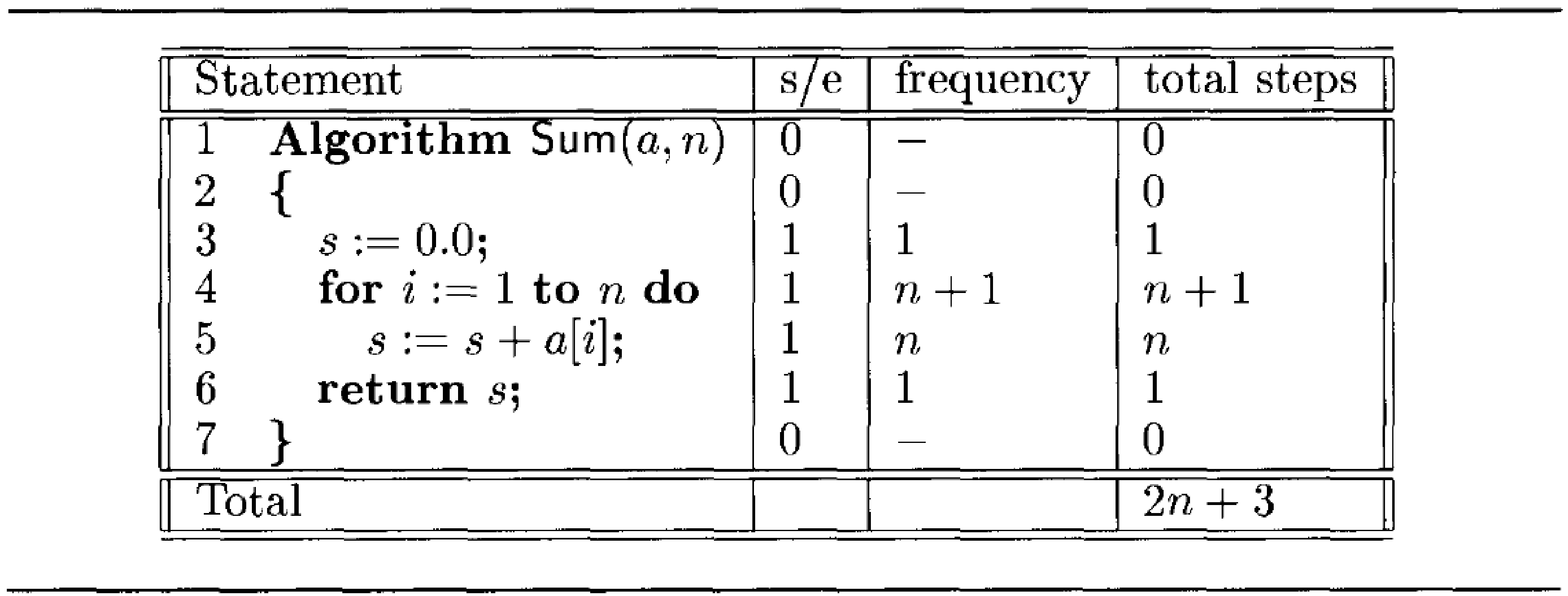
Step count for RSum is 2n+2

```
void Add (Type a[ ][size], Type b[ ][size], Type c[ ][size], int m, int n)
{    for (int i=1; i<=m; i++)
     {
          count++;
          for(int j=1; j<=n; j++)
          {
               count++;
               c[i][j] = a[i][j] + b[i][j];
               count++;
          }
          count++;
     }
     count++;
}
```

Step count $R_{Add} = 2mn + 2m + 1$

Method 2

- Step count is calculated based on a table in which total number of steps contributed by each statement is recorded
- s/e – steps per execution



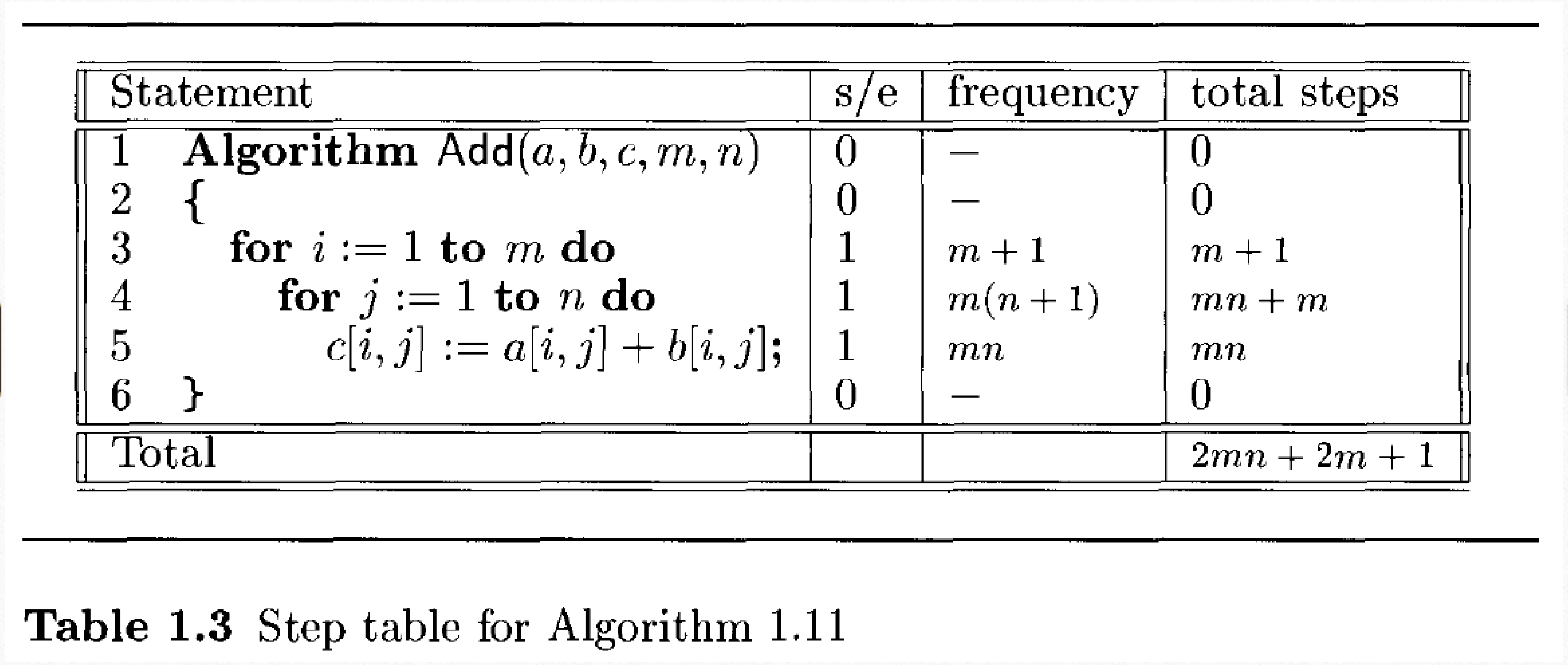
Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0;$	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i];$	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

Table 1.1 Step table for Algorithm 1.6

		frequency		total steps	
Statement	s/e	$n = 0$	$n > 0$	$n = 0$	$n > 0$
1 **Algorithm** $\mathsf{RSum}(a, n)$	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 $\mathsf{RSum}(a, n-1) + a[n]$;	$1 + x$	0	1	0	$1 + x$
7 }	0	–	–	0	0
Total				2	$2 + x$

$$x = t_{\mathsf{RSum}}(n-1)$$

Table 1.2 Step table for Algorithm 1.7



Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	—	0
2 {	0	—	0
3 **for** $i := 1$ **to** m **do**	1	$m + 1$	$m + 1$
4 **for** $j := 1$ **to** n **do**	1	$m(n + 1)$	$mn + m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	—	0
Total			$2mn + 2m + 1$

Table 1.3 Step table for Algorithm 1.11

Asymptotic notations and Basic efficiency classes

- Efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as principal indicator of the algorithm's efficiency
- To compare the orders of growth the three notations, O(big oh), Ω(big omega) and Θ(big theta) are used
- Let t(n) and g(n) be any non negative functions defined on a set of natural numbers
- We are interested in t(n), the algorithm's running time
- g(n) be some simple function to the compare the count with
- C(n) be the basic operation count

- $O(g(n))$ is the set of all functions with a smaller or same order of growth as $g(n)$
- Let $g(n) = n^2$
- $n \in O(n^2)$ $100n + 5 \in O(n^2)$ $\frac{1}{2} n(n-1) \in O(n^2)$
- $n^3 \notin O(n^2)$ $0.00001n^3 \notin O(n^2)$ $n^4 + n + 1 \notin O(n^2)$

- $\Omega(g(n))$ is the set of all functions with a larger or same order of growth as $g(n)$
- Let $g(n) = n^2$
- $n^3 \in \Omega(n^2)$ $\frac{1}{2} n(n-1) \in \Omega(n^2)$ $100n + 5 \notin \Omega(n^2)$

O-notation

- A function t(n) is said to be in O(g(n)) denoted t(n) ∈ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, there exist some positive constant c and some non negative integer n_0 such that
- $t(n) \leq c\, g(n) \quad \text{for all } n \geq n_0$
- Ex: $100n + 5 \in O(n^2)$
- $100n + 5 \leq 100n + n$ (for all $n \geq 5$)
- $= 101n \leq 101n^2$
- Constant c is 101 and n_0 is 5

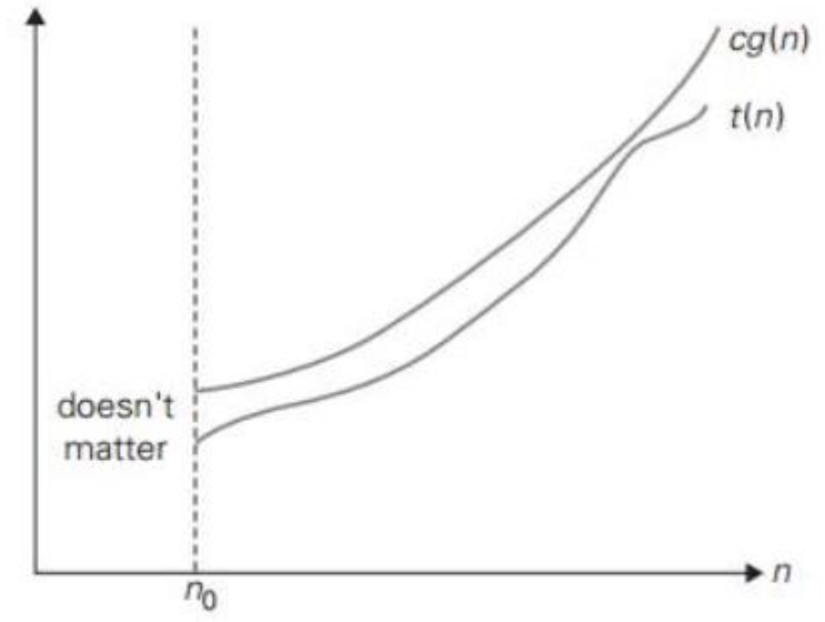


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

Ω-notation

- A function t(n) is said to be in Ω(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 large n, i.e., if there exist some positive constant c and some non negative integer n0 such that
- $t(n) \geq c\, g(n)$ for all $n \geq n_0$
- Ex: $n^3 \in \Omega(n^2)$
- $n^3 \geq n^2$ for all $n \geq 0$
- Constant $c = 1$ and $n_0 = 0$

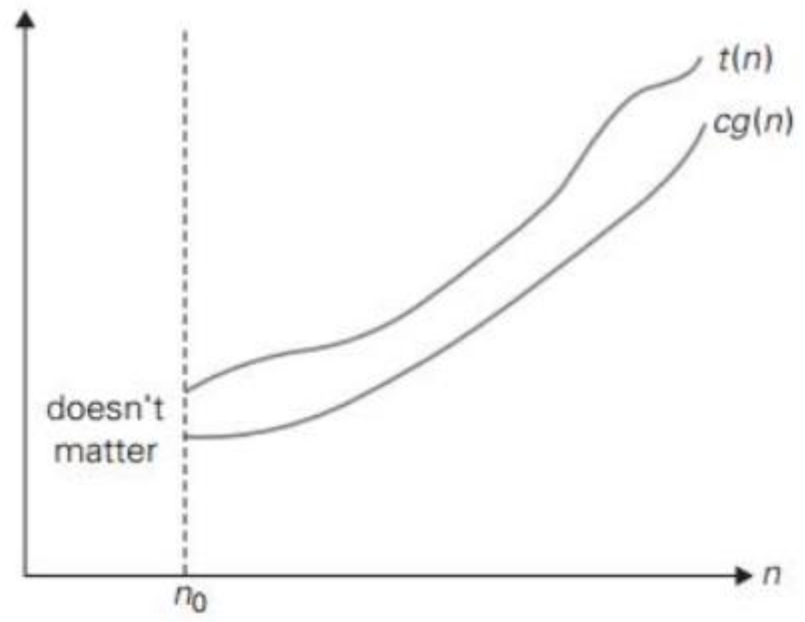


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

Θ-notation

- 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 non negative integer n_0 such that
- $c_2\, g(n) \le t(n) \le c_1\, g(n)$ for all $n \ge n_0$

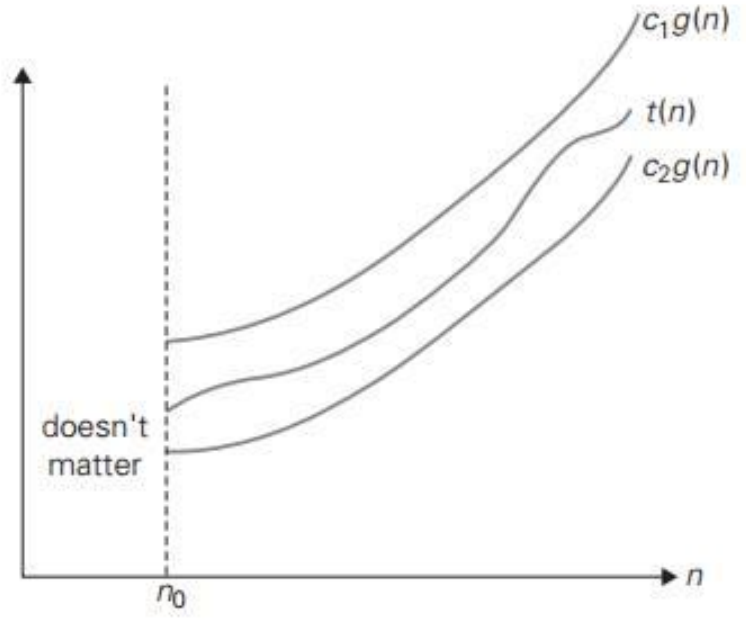
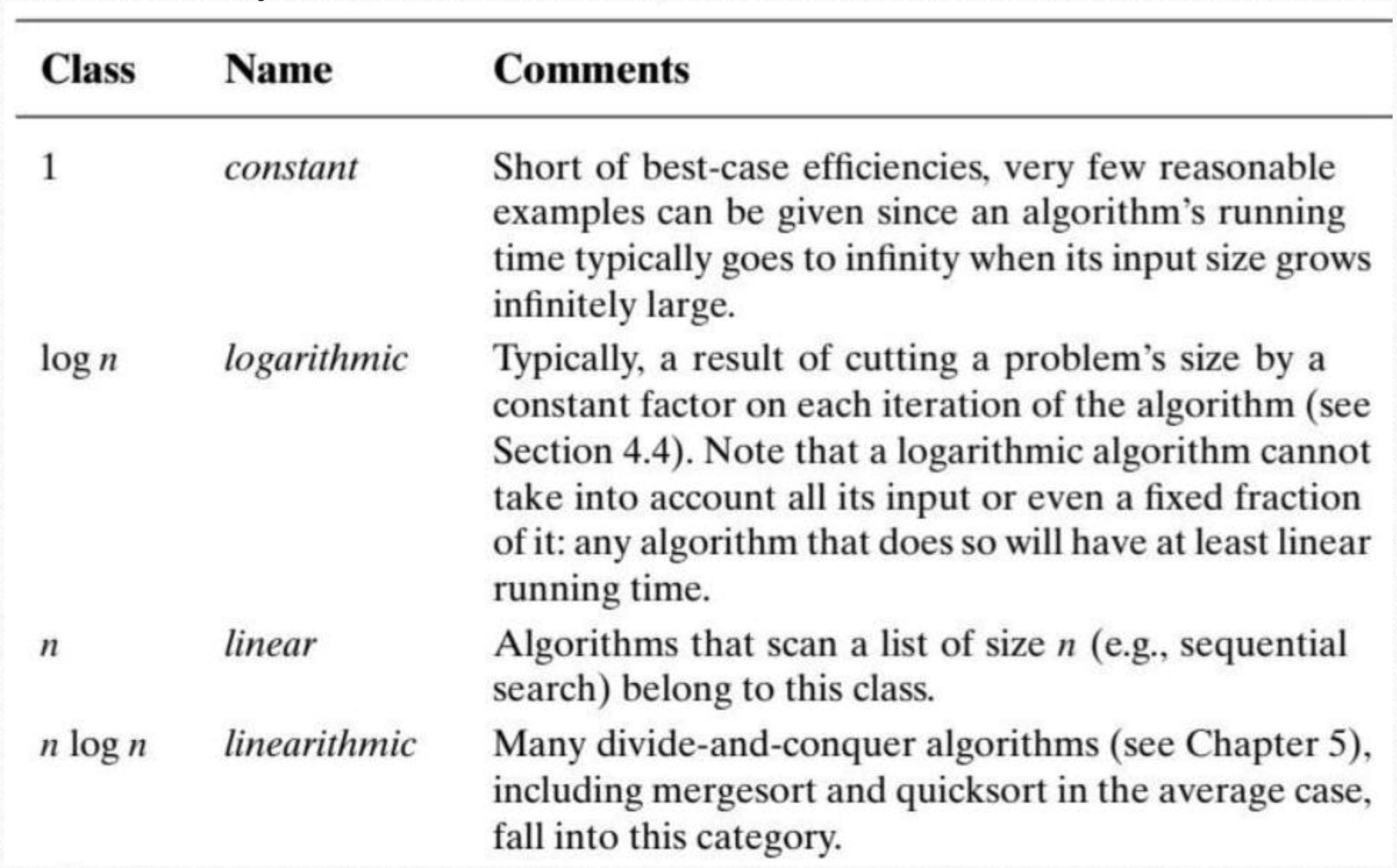


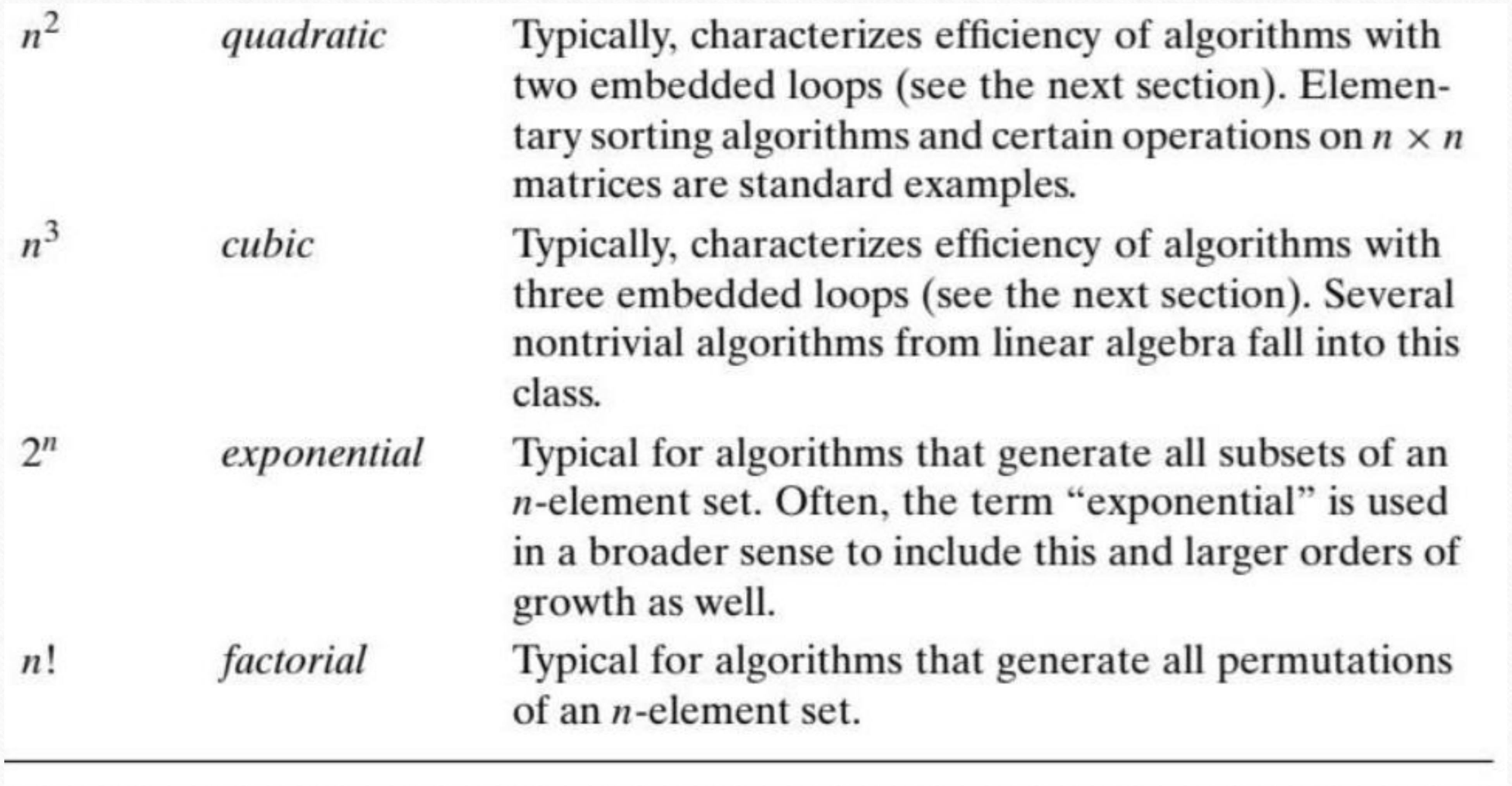
FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

Θ-notation

Basic efficiency classes



Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
log n	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 4.4). Note that a logarithmic algorithm cannot take into account all its input or even a fixed fraction of it: any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
n log n	*linearithmic*	Many divide-and-conquer algorithms (see Chapter 5), including mergesort and quicksort in the average case, fall into this category.



n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on $n \times n$ matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term “exponential” is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Ex: Linear search

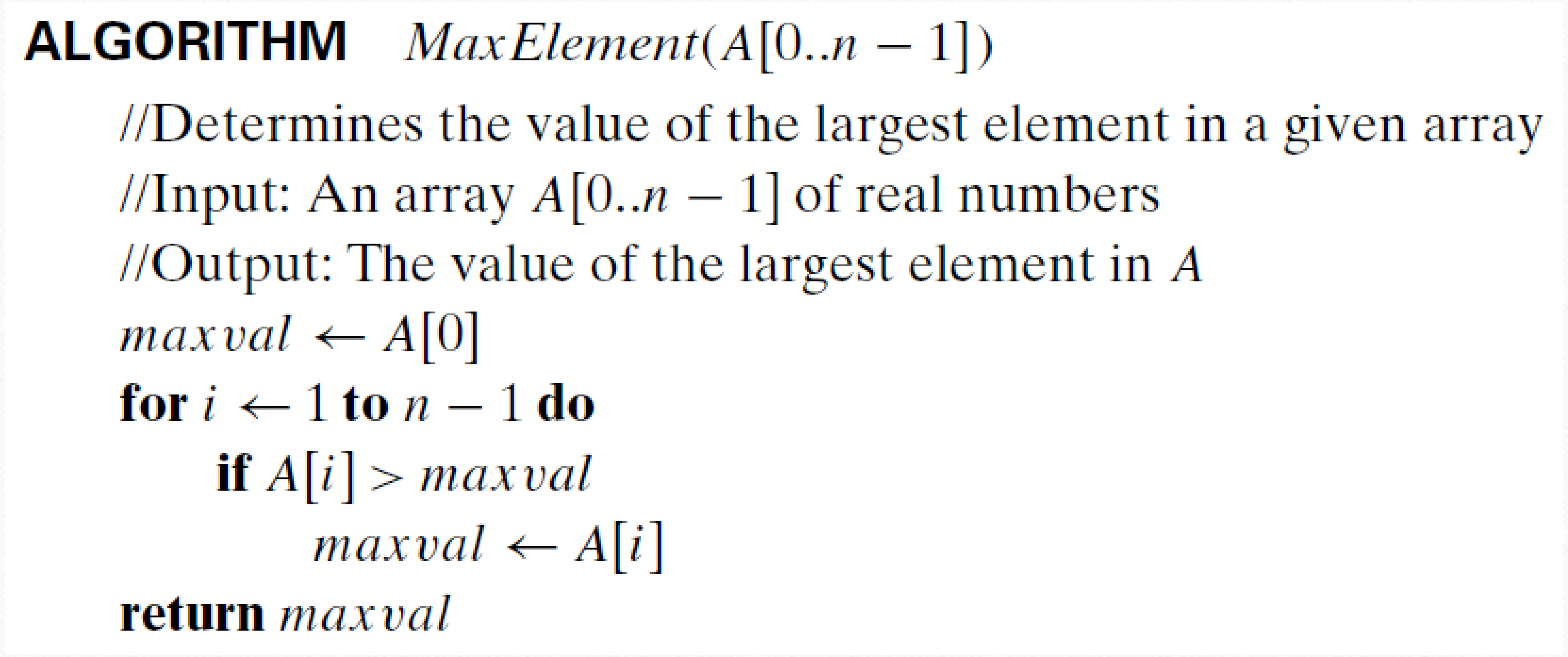
- n =100
- a[100] = {1, 2, 3, ... 100}
- Key = 1, 100, 50
- Best case = Ω (1)
- Worst case = O(n)
- Average case = n/2 = Θ (n)

Other examples

- $O(n^3)$
- $\Omega(\log n)$

Mathematical analysis of non recursive algorithms

- Example 1: Finding the largest element in an array



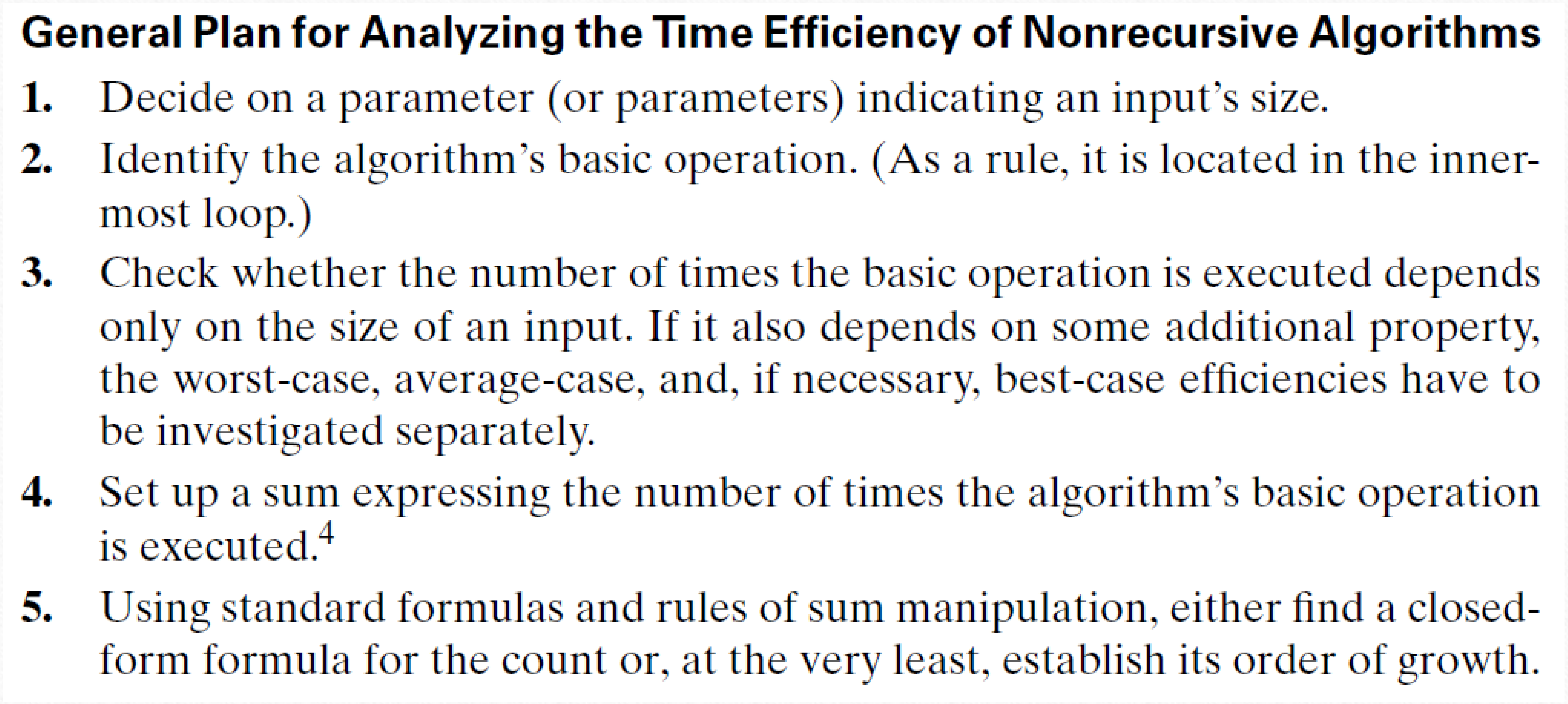
```
ALGORITHM  MaxElement(A[0..n − 1])
    //Determines the value of the largest element in a given array
    //Input: An array A[0..n − 1] of real numbers
    //Output: The value of the largest element in A
    maxval ← A[0]
    for i ← 1 to n − 1 do
        if A[i] > maxval
            maxval ← A[i]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

- The measure of the input's size is the number of elements n in the array
- Algorithm's for loop executes for the greatest number of times
- The comparison operation within the loop is considered basic than the assignment operation
- Since the comparison has to run for all the iteration for any value of n, there is nothing distinguish among worst, best and average cases
- Let C(n) be the number of times this comparison operation is executed
- The comparison operation repeats for each iteration from i=1 to n-1

$$\text{Hence } C(n) = \sum_{i=1}^{n-1} 1$$

- This is nothing but 1 repeated n-1 times

$$C(n) = n-1 \in \Theta(n)$$



General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation. (As a rule, it is located in the innermost loop.)
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.[4]
5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c \sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

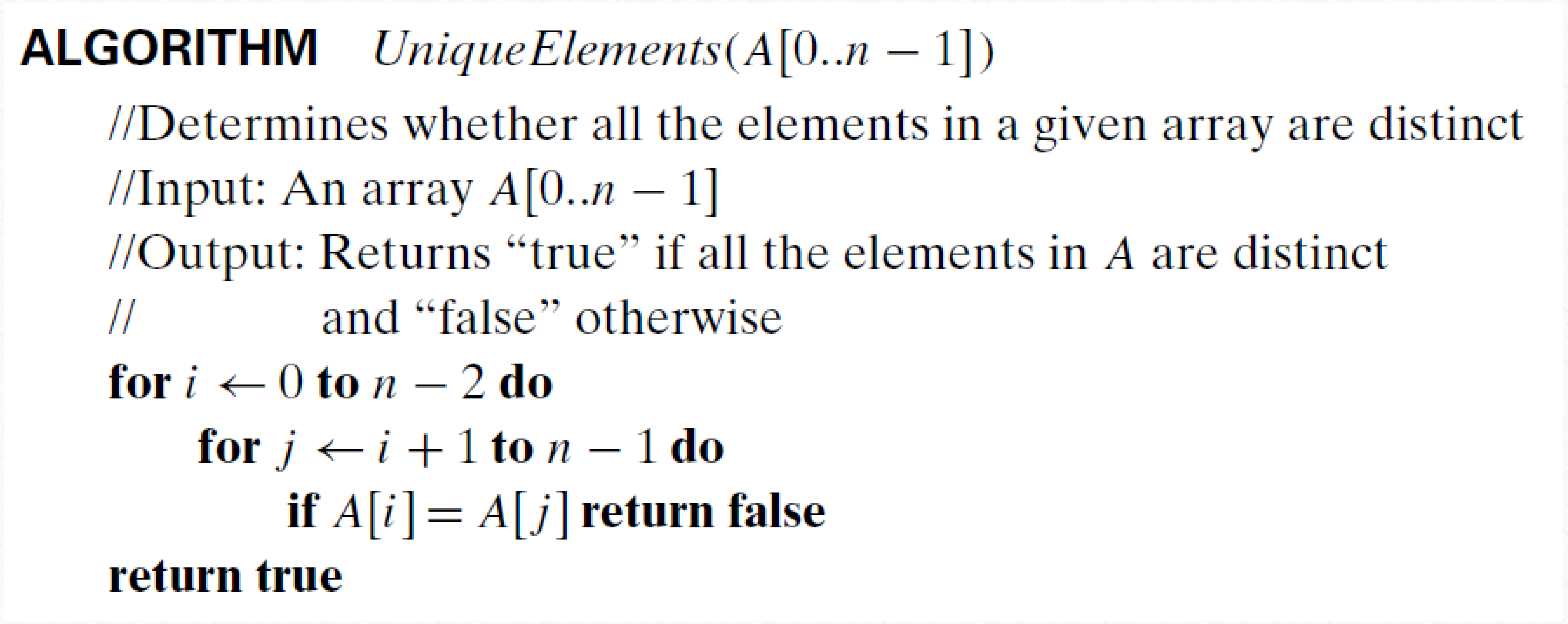
Summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

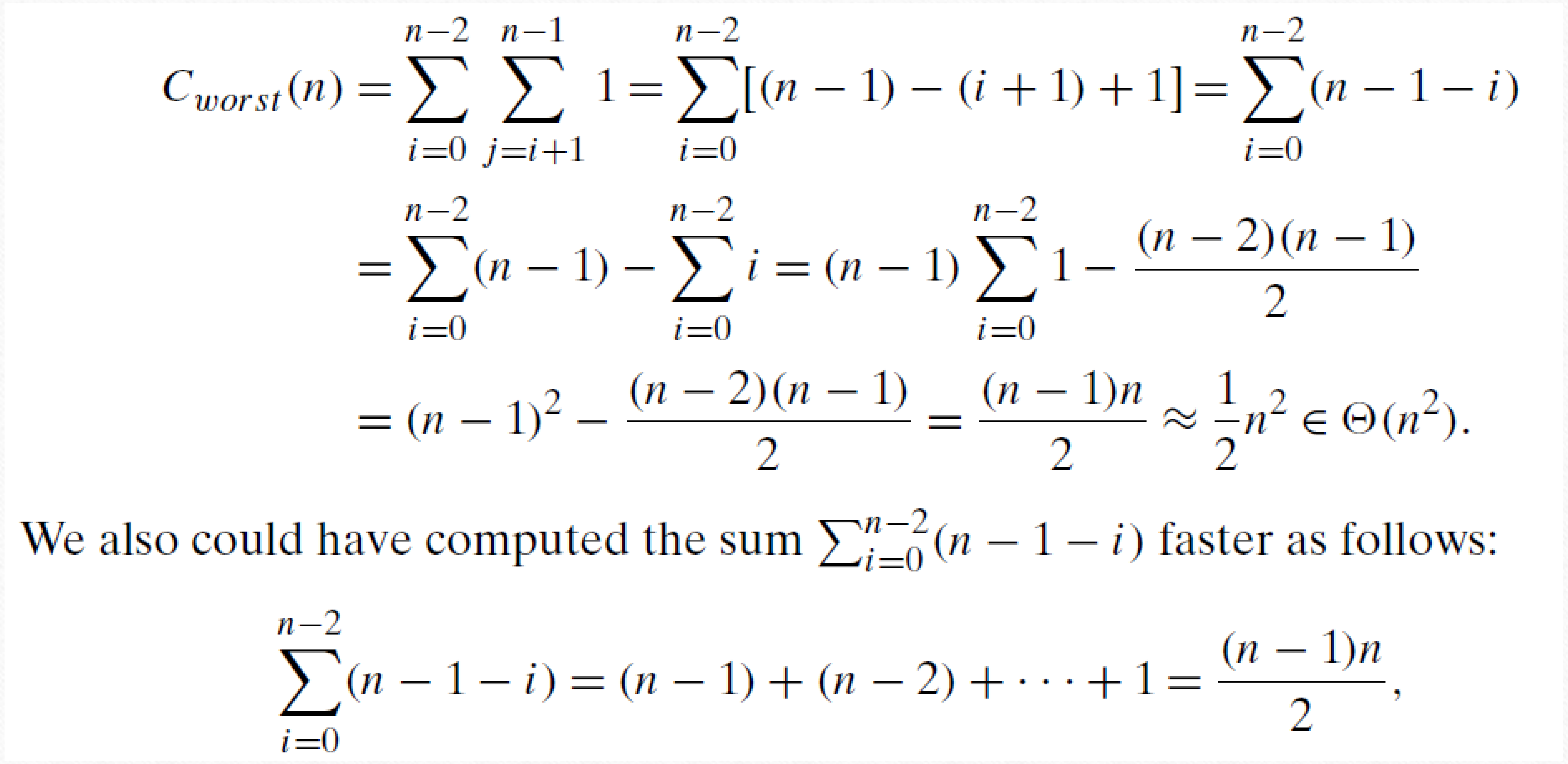
Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.

Example 2: Element uniqueness problem: Check whether all elements in a given array are distinct



```
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
```

- Measure of input’s size is n, number of elements in the array
- The comparison operation in the innermost loop is the basic operation
- The number of comparisons depend on the position of equal elements
- Hence, we can distinguish among best, worst and average cases
- Worst case happens for two cases:
- If the array doesn’t contain equal elements
- If the only pair of equal elements occupies the last two positions



$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

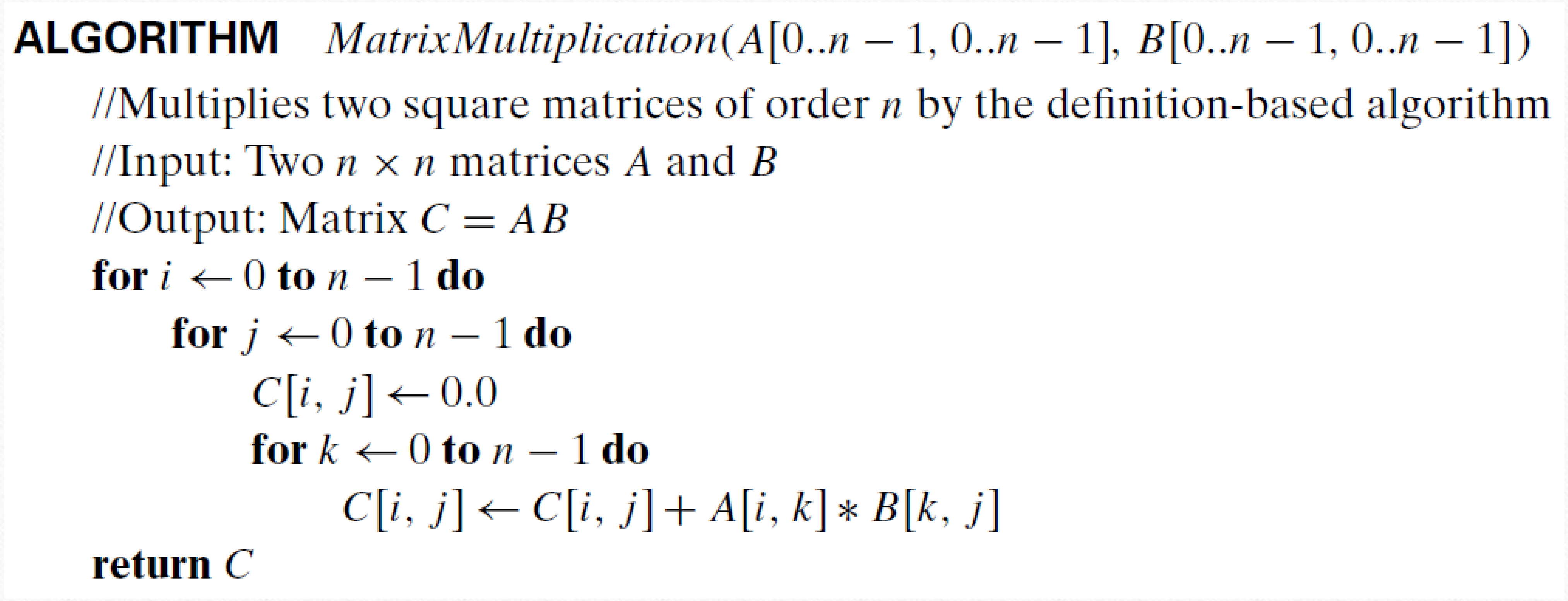
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2} (n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

Example 3: Finding the product of two matrices



```
ALGORITHM  MatrixMultiplication($A[0..n-1, 0..n-1], B[0..n-1, 0..n-1]$)
    //Multiplies two square matrices of order $n$ by the definition-based algorithm
    //Input: Two $n \times n$ 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 $C$
```

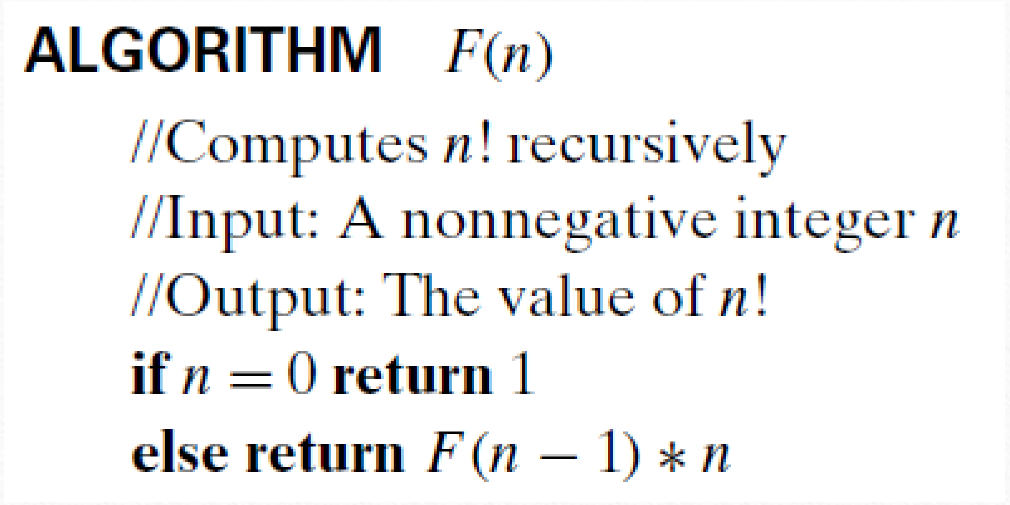
- Input's size is measured by the matrix order n
- The innermost loop statement contains multiplication which is considered the basic operation
- Since the loop must execute for all the iterations there is no distinguish among best, worst and average case
- Total number of multiplications M(n) is expressed as

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

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

Mathematical analysis of recursive algorithms

- Example 1: Compute the factorial function F(n) = n!
- n! = 1 * . . . . . . (n-1) * n = (n-1)! * n   for n>=1
- We know that 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 $F(n - 1) * n$
```

- Measure of input's size is n
- Basic operation is multiplication denoted by M(n)
- F(n) is computed according to the formula

$$F(n) = F(n-1)\ .\ n \qquad \text{for } n > 0$$

- The number of multiplications needed is

$$M(n) = \underset{\substack{\text{to compute}\\ F(n-1)}}{M(n-1)} + \underset{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}}{1} \qquad \text{for } n > 0.$$

- The initial condition is at n = 0, because it makes the algorithm to stop recursive calls

  If n = 0 return 1

- When n = 0 multiplication operation doesn’t execute
- The recurrence relation for the factorial algorithm is

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

- Therefore the number of multiplications M(n) is

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0,$$
$$M(0) = 0.$$

- Apply the method of backward substitution

$$\begin{aligned} M(n) &= M(n-1)+1 && \text{substitute } M(n-1)=M(n-2)+1 \\ &= [M(n-2)+1]+1 = M(n-2)+2 && \text{substitute } M(n-2)=M(n-3)+1 \\ &= [M(n-3)+1]+2 = M(n-3)+3. \end{aligned}$$

- In general

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

- Since n = 0 is the initial condition, i can have n values. Substituting i = n

$$M(n) = M(n-1)+1 = \cdots = M(n-i)+i = \cdots = M(n-n)+n = n.$$

- The order of growth of factorial algorithm is linear, i.e, n

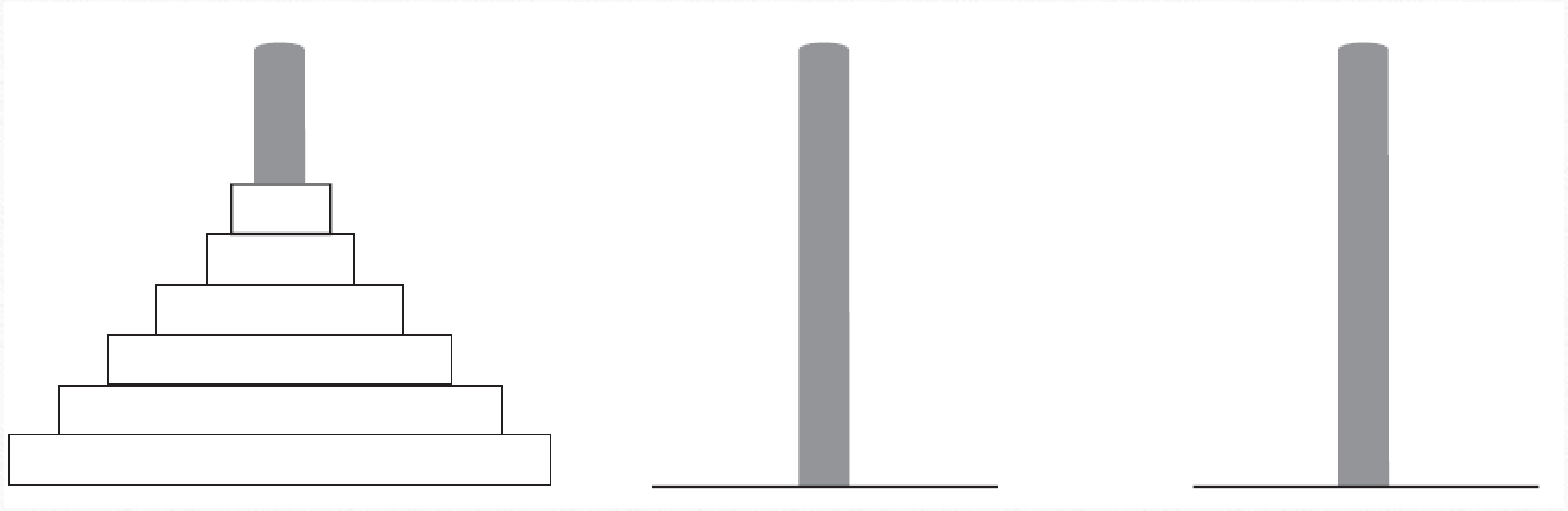
General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case efficiencies must be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the order of growth of its solution.

Example 2: Tower of Hanoi

Steps

1. Recursively move n-1 disks from Peg 1 to Peg 2
2. Move n^{th} disk from Peg 1 to Peg 3
3. Recursively move n-1 disks from Peg 2 to Peg 3



- Measure of input's size is number of disks n
- Moving a disk is the algorithm's basic operation
- Number of moves M(n) depends only on n
- The recurrence relation is

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

- The initial condition is when n = 1 the number of move is 1
- Hence the recurrence relation with the initial condition is

$$\begin{aligned} &M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \\ &M(1) = 1. \end{aligned}$$

- Applying the method of backward substitution

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2M(n-2) + 2 + 1 \qquad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3M(n-3) + 2^2 + 2 + 1.$$

- The next pattern will be

$$2^4M(n-4) + 2^3 + 2^2 + 2 + 1$$

- In general after i substitutions

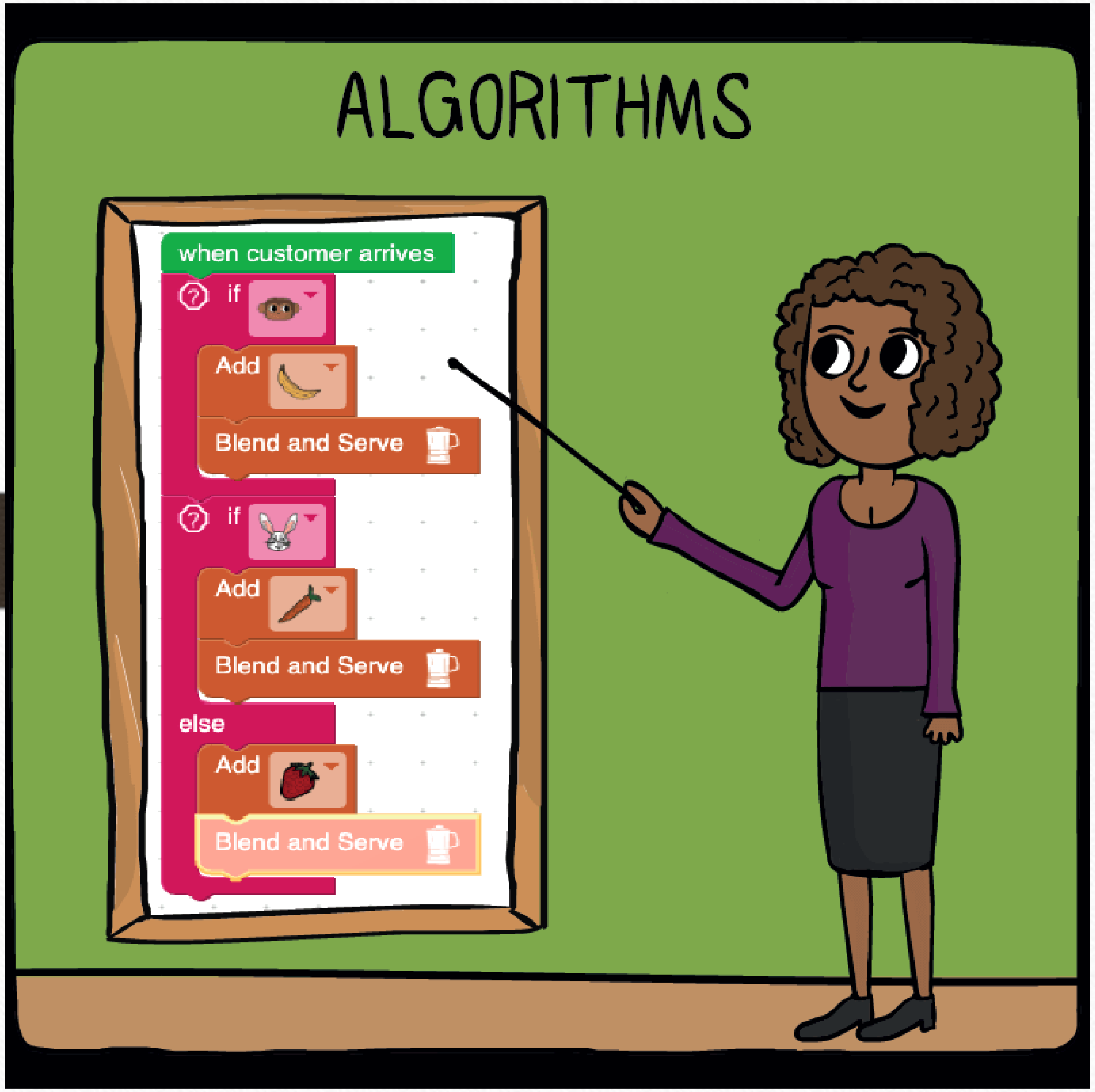
$$M(n) = 2^iM(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^iM(n-i) + 2^i - 1$$

- The initial condition n = 1 is achieved for i = n-1. Substituting i

$$M(n) = 2^i M(n - i) + 2^i - 1.$$

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

- The order of growth of tower of Hanoi is exponential, i.e., 2^n


ALGORITHMS
when customer arrives
if
Add
Blend and Serve
if
Add
Blend and Serve
else
Add
Blend and Serve