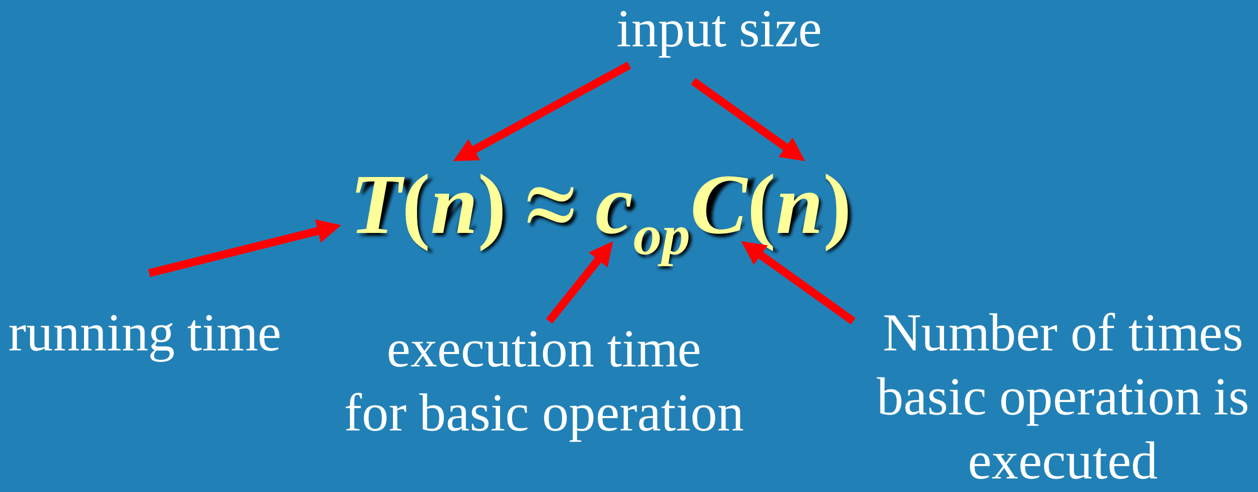
Chapter 2: Analysis of algorithms

- Issues:
  - correctness
  - time efficiency
  - space efficiency
  - optimality
- Approaches:
  - theoretical analysis
  - empirical analysis

2.1 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

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

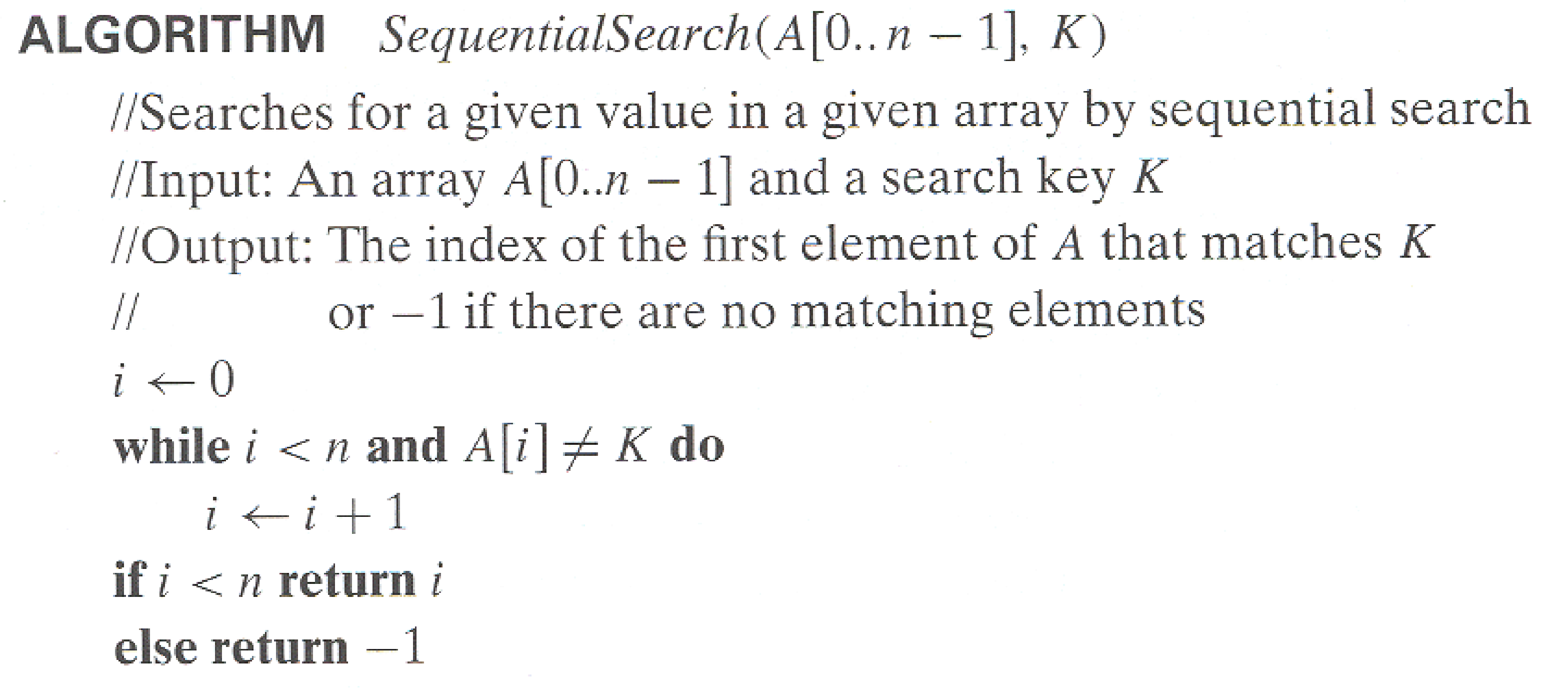
- Analyze the empirical data

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

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

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

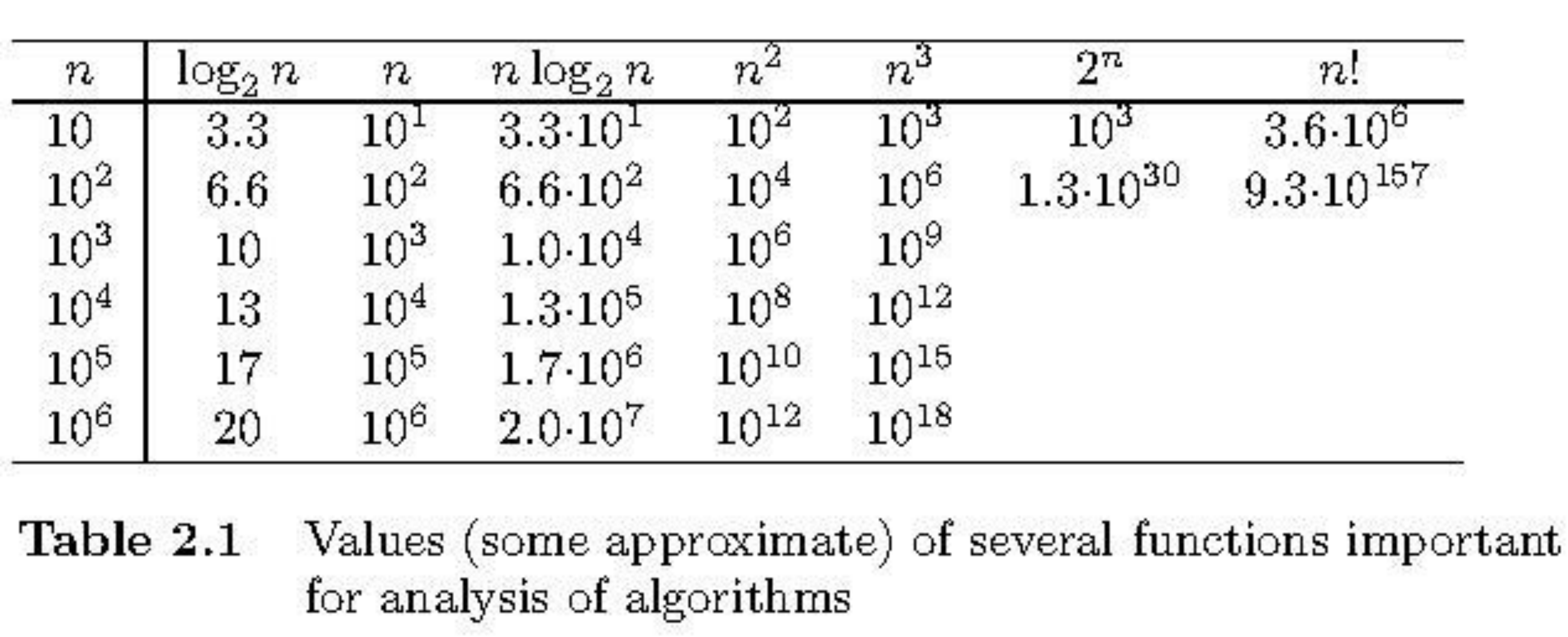
  e.g., $C(n) \approx cn^2$

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

Recapitulation of the Analysis Framework

- Both time and space efficiencies are measured as functions of the algorithm's input size.
- Time efficiency is measured by counting the number of times the algorithm's basic operation is executed. Space efficiency is measured by counting the number of extra memory units consumed by the algorithm.
- The efficiencies of some algorithms may differ significantly for inputs of the same size. For such algorithms, we need to distinguish between the worst-case, average-case, and best-case efficiencies.
- The framework's primary interest lies in the order of growth of the algorithm's running time (extra memory units consumed) as its input size goes to infinity.

1.2 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 no faster than g(n)
- Θ(g(n)): class of functions f(n) that grow at same rate as g(n)
- Ω(g(n)): class of functions f(n) that grow at least as fast as g(n)

Big-oh

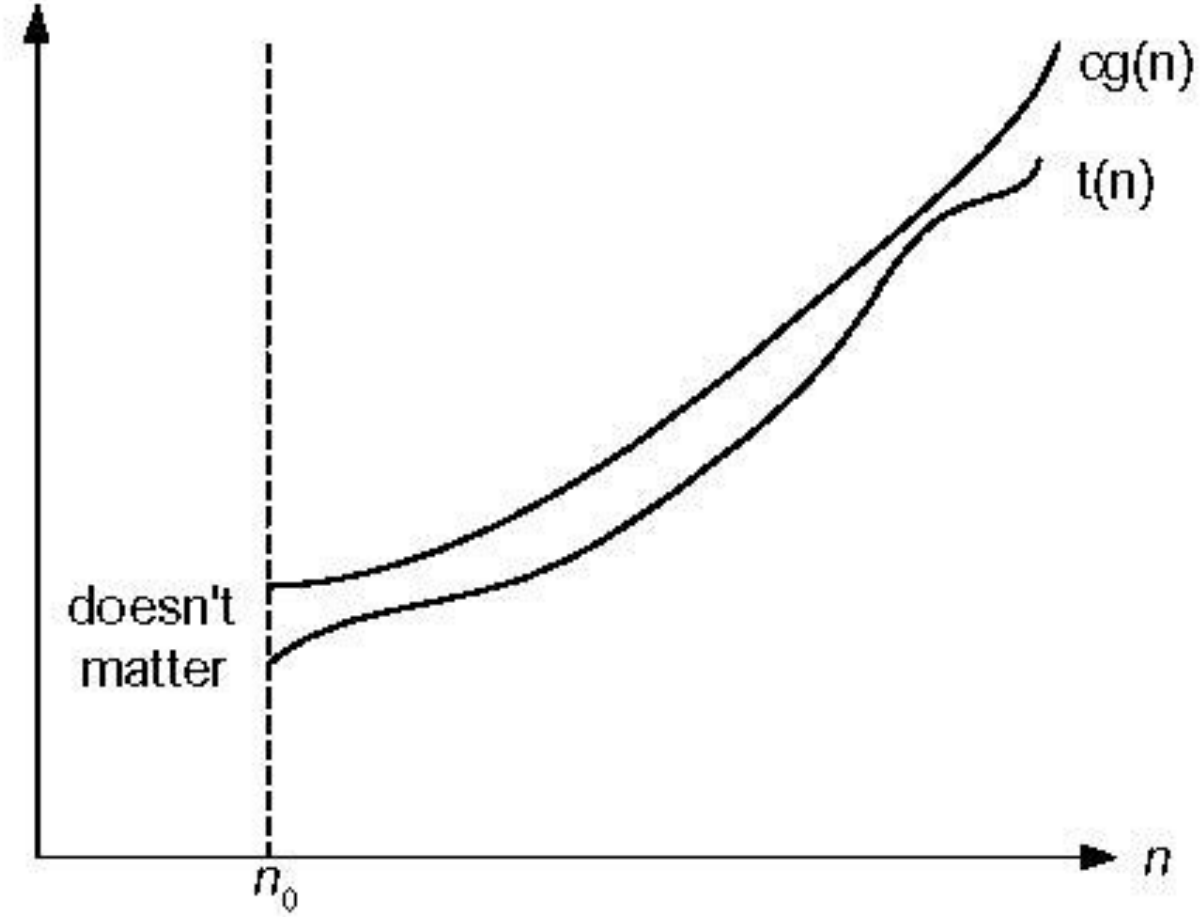


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

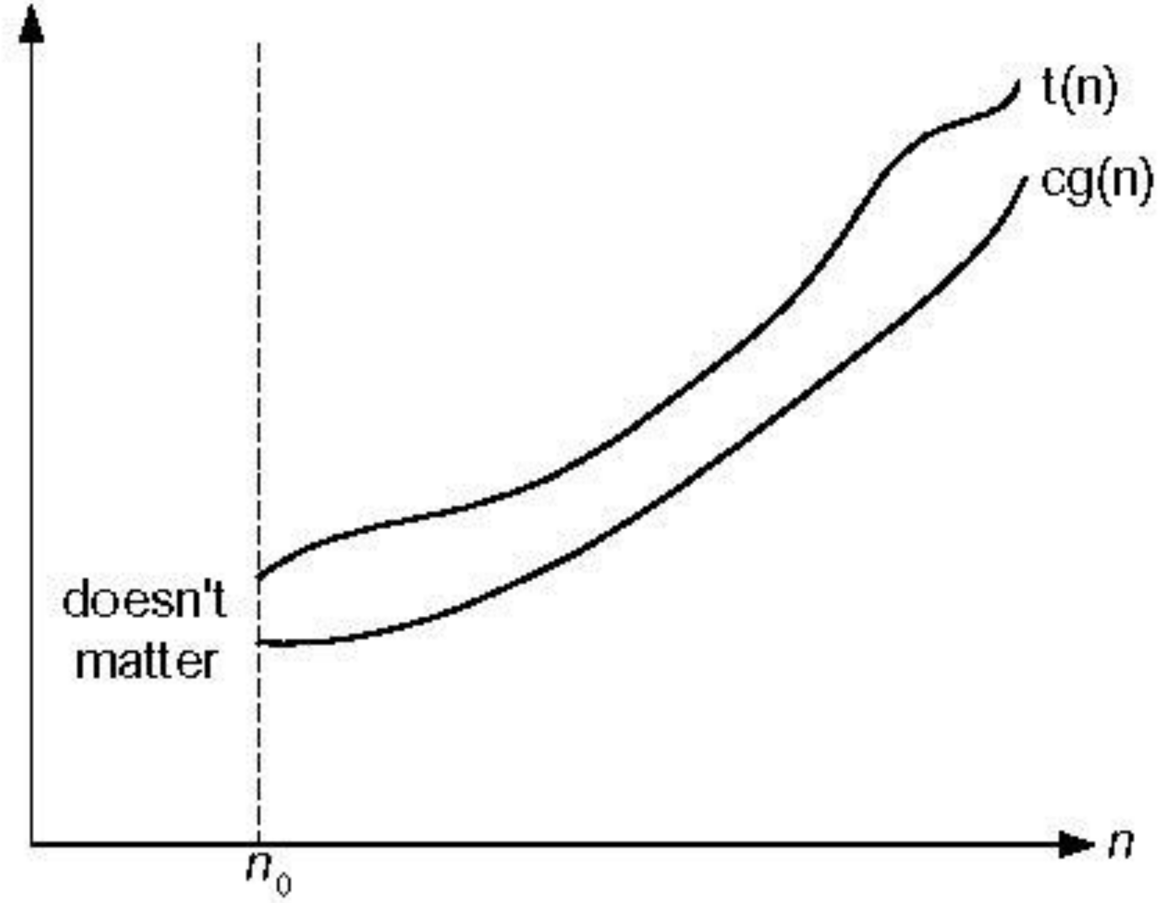


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

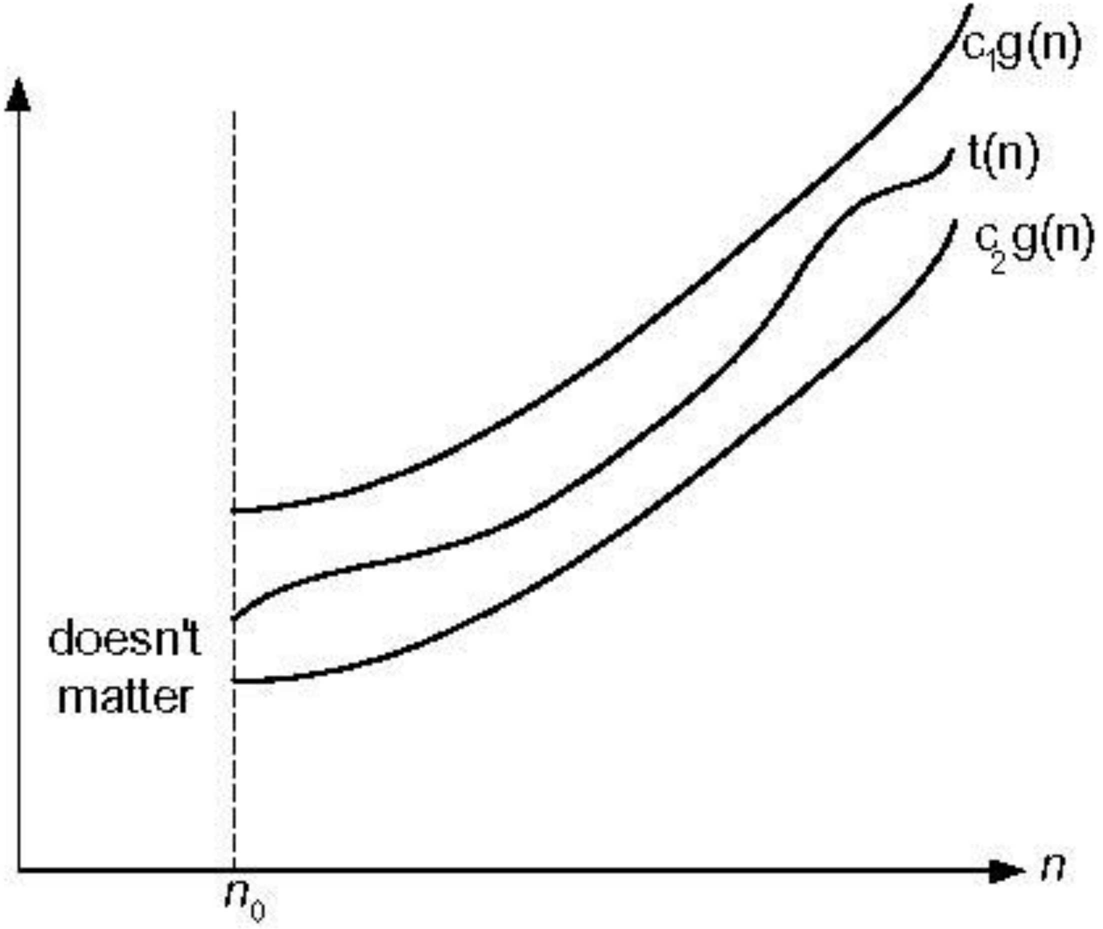


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

Definition: $f(n)$ **is in** $O(g(n))$ **if order of growth of** $f(n) \le$ **order of growth of** $g(n)$ **(within constant multiple), i.e., there exist positive constant** c **and non-negative integer** n_0 **such that**

$$f(n) \le c\, g(n) \text{ for every } n \ge n_0$$

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

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

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: $\log_2 n$ vs. n

Example: 2^n vs. $n!$

Stirling’s formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

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* or linearithmic
n^2	quadratic
n^3	cubic
2^n	exponential
n!	factorial

2.3 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

$\Sigma_{l \le i \le u} 1 = 1 + 1 + \cdots + 1 = u - l + 1$

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

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

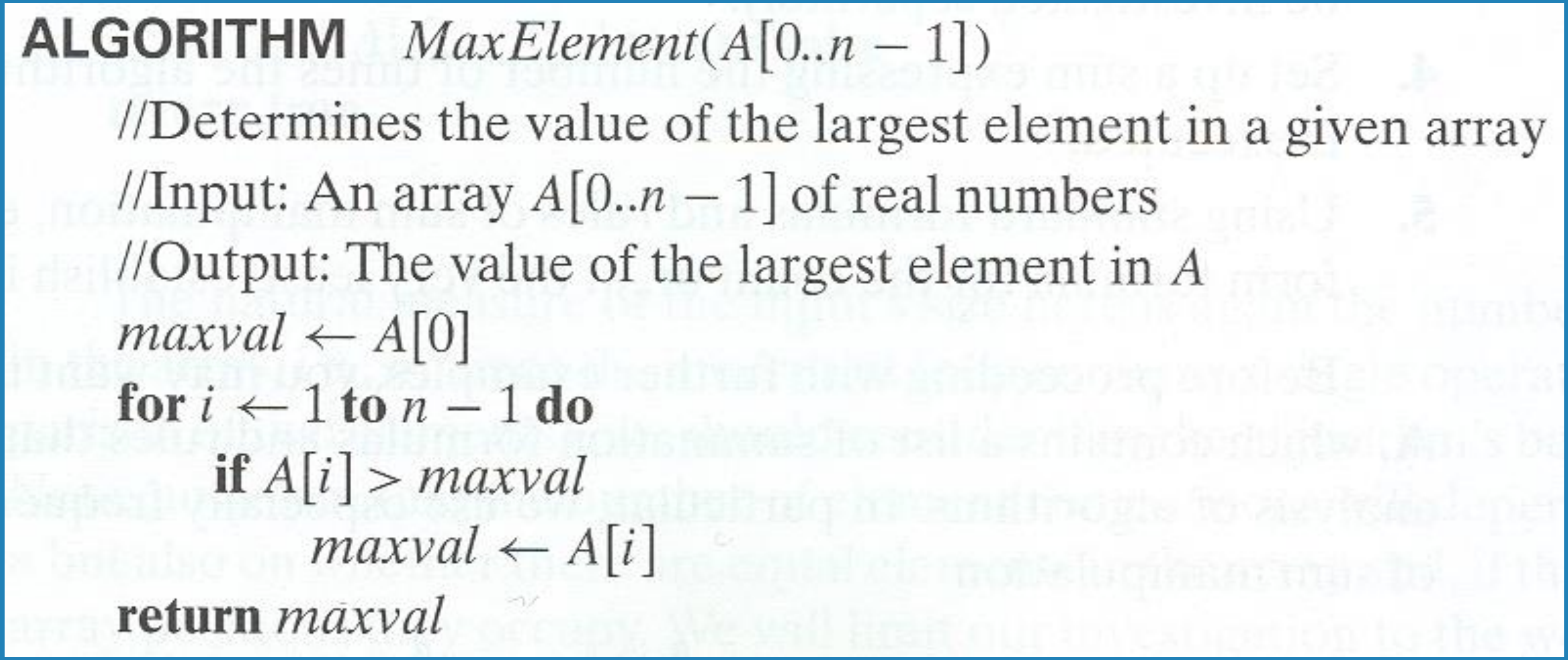
$\Sigma_{1 \le i \le n}\, i^2 = 1^2 + 2^2 + \cdots + n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

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

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

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i$ $\quad \Sigma c a_i = c \Sigma a_i$ $\quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

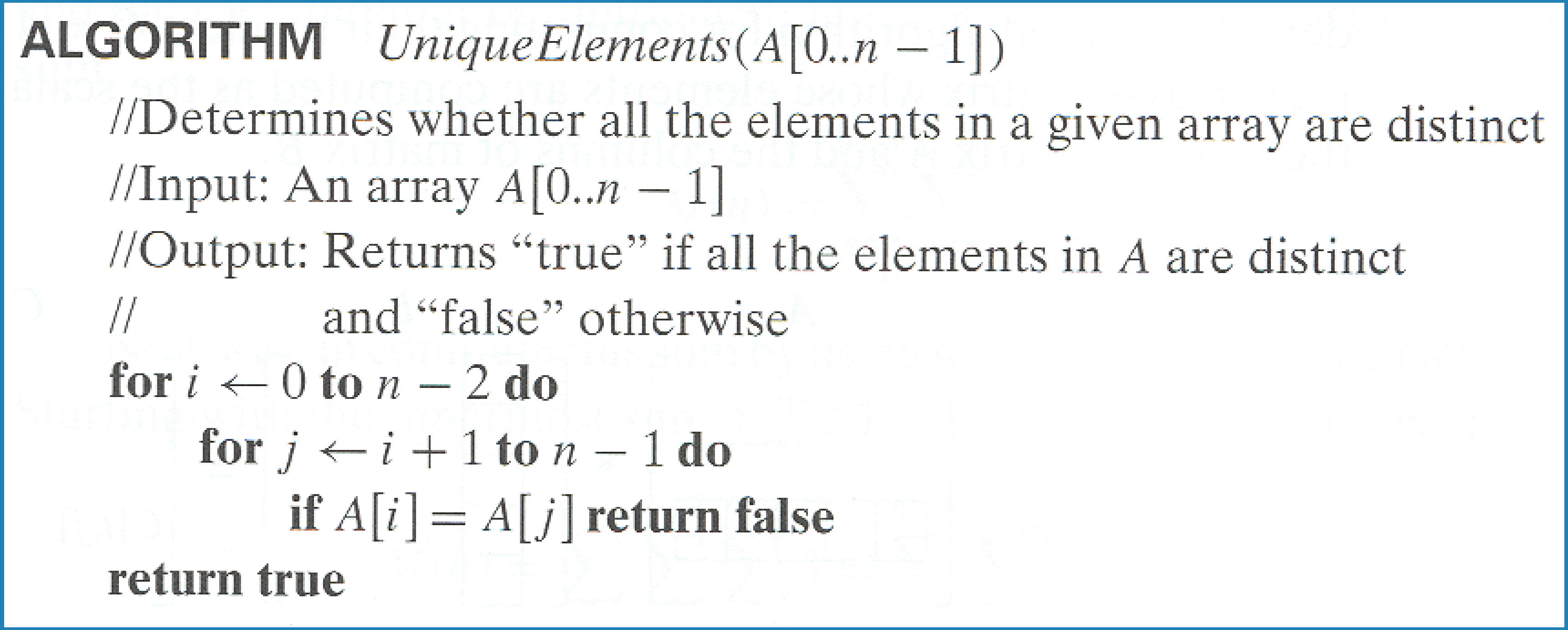
Example 1: Maximum element



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

No need to distinguish the best, worst, and average cases here!

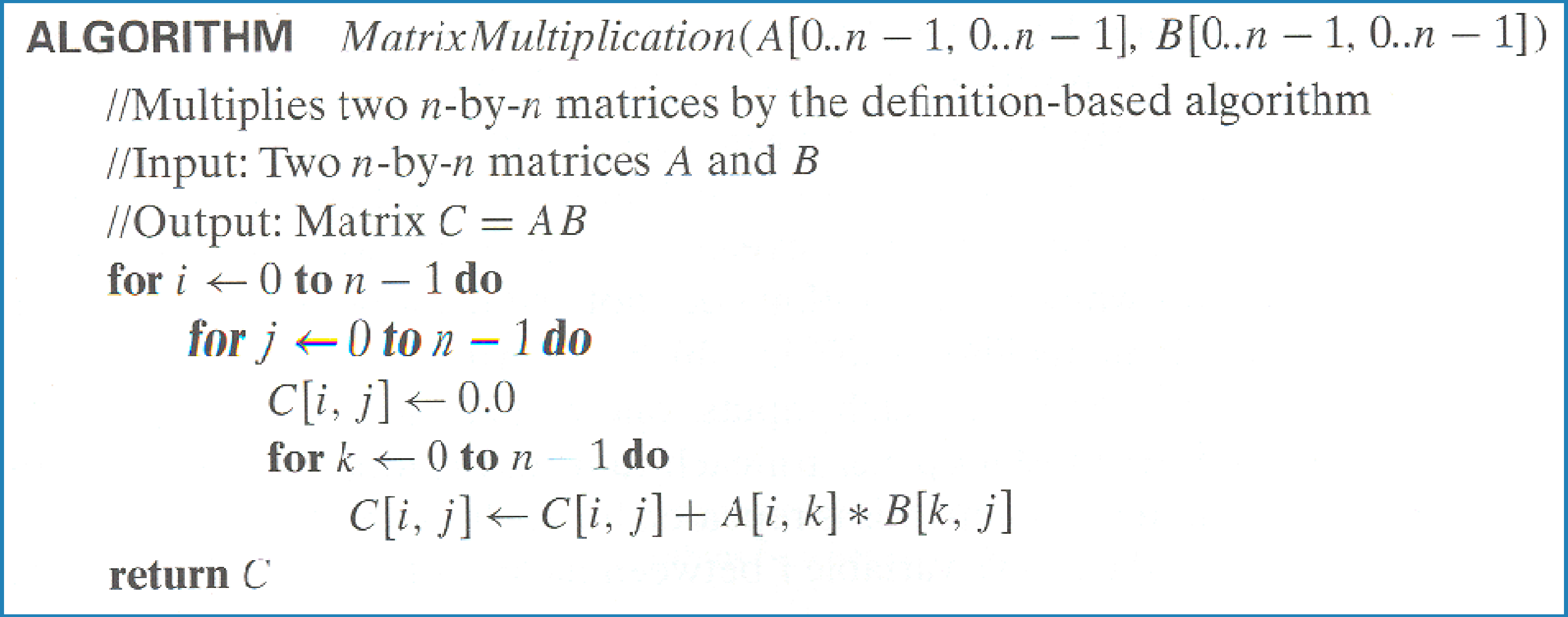
Example 2: Element uniqueness problem



```
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
```

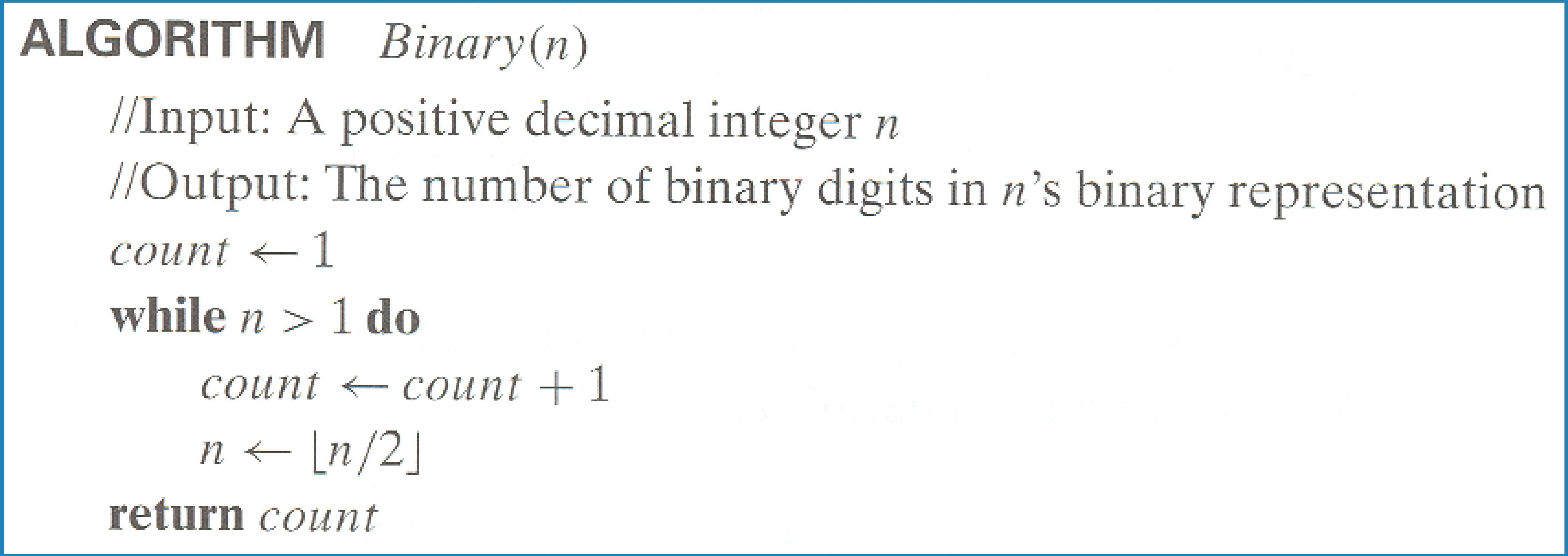
Consider worst case only!

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 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
        $n \leftarrow \lfloor n/2 \rfloor$
    return count
```

It cannot be investigated the way the previous examples are.

$$count = \lfloor \log_2 n \rfloor + 1$$

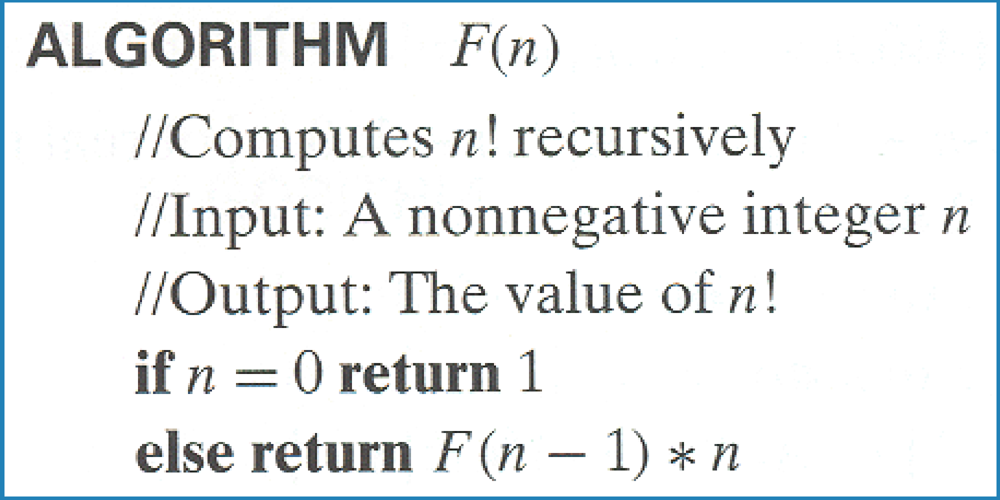
2. 4 Plan for Analysis of Recursive Algorithms

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

Recursive definition of *n*!: $F(n) = F(n\text{-}1) \cdot n$ **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:

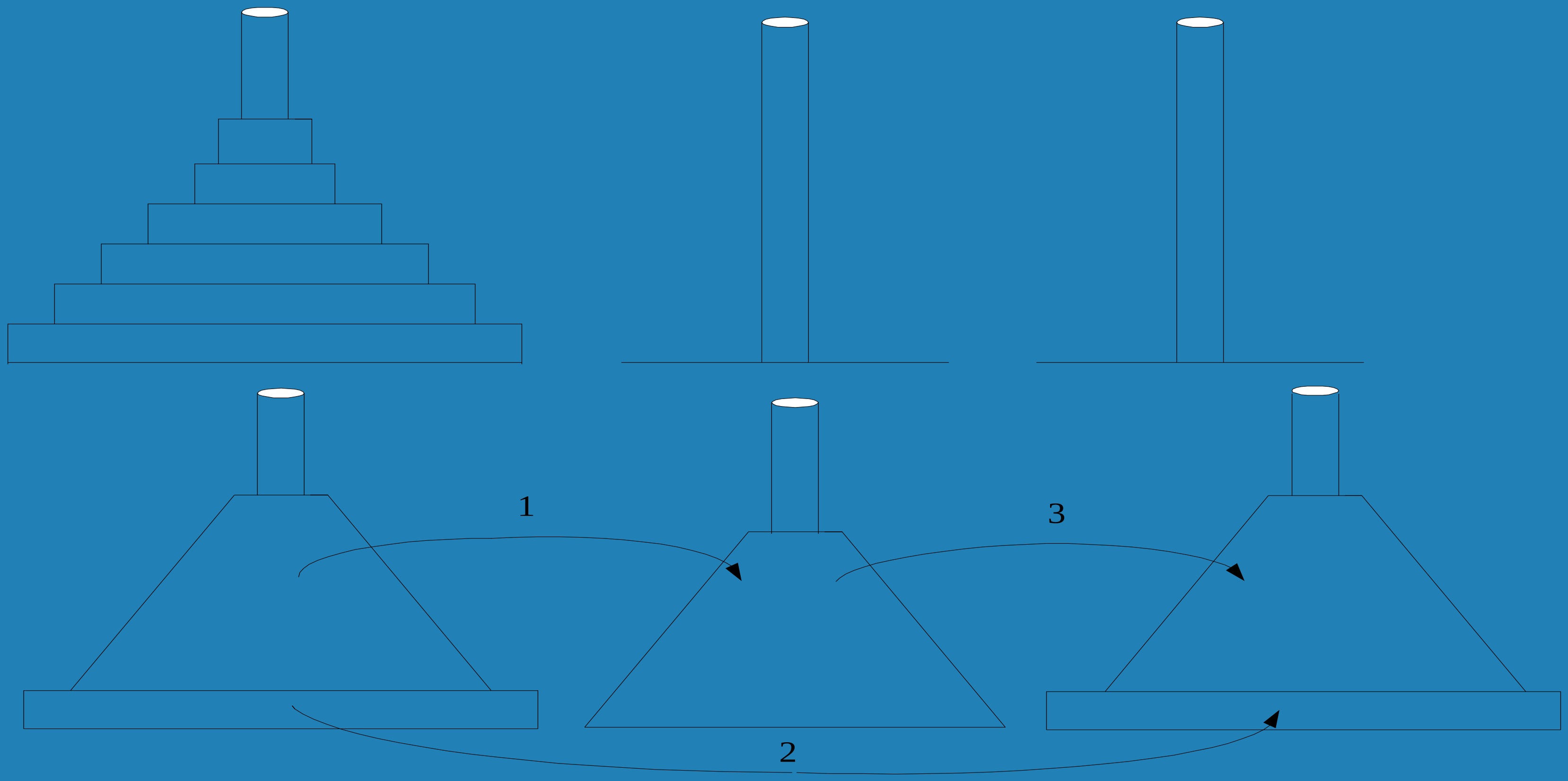
Basic operation:

Recurrence relation:

Solving the recurrence for M(*n*)

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

Example 2: The Tower of Hanoi Puzzle

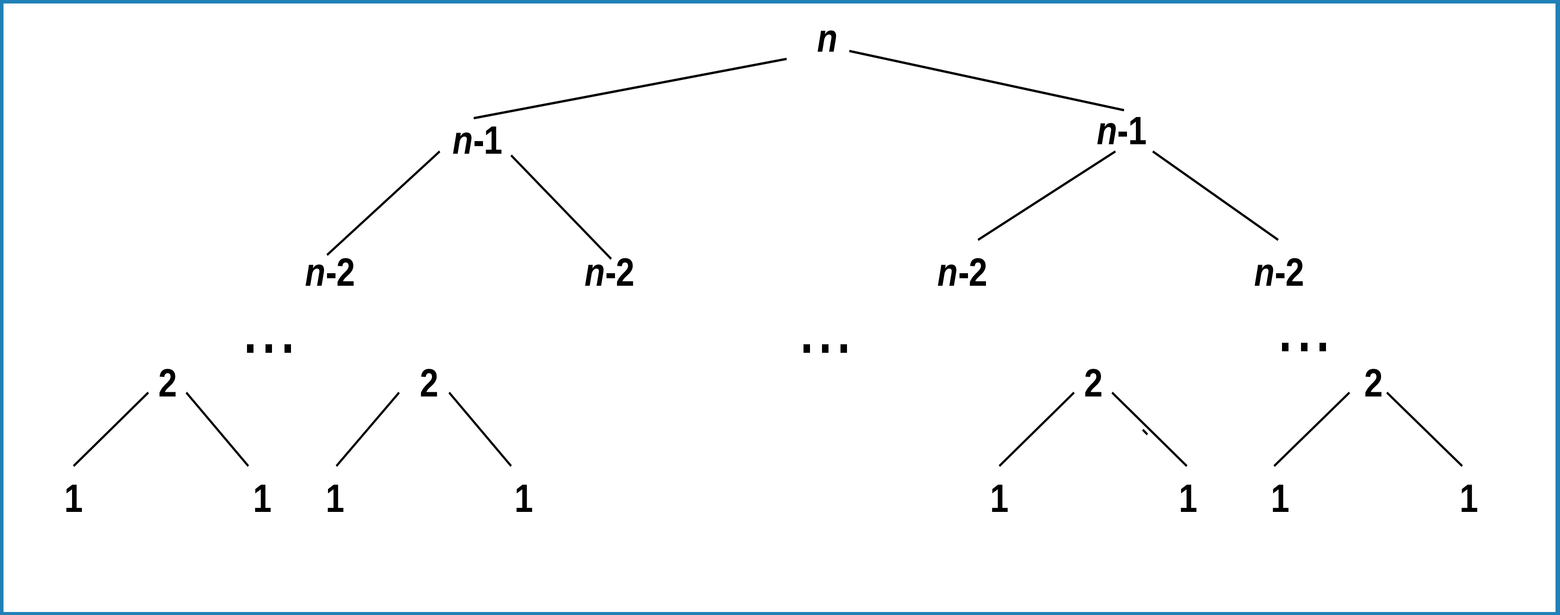


Recurrence for number of moves:

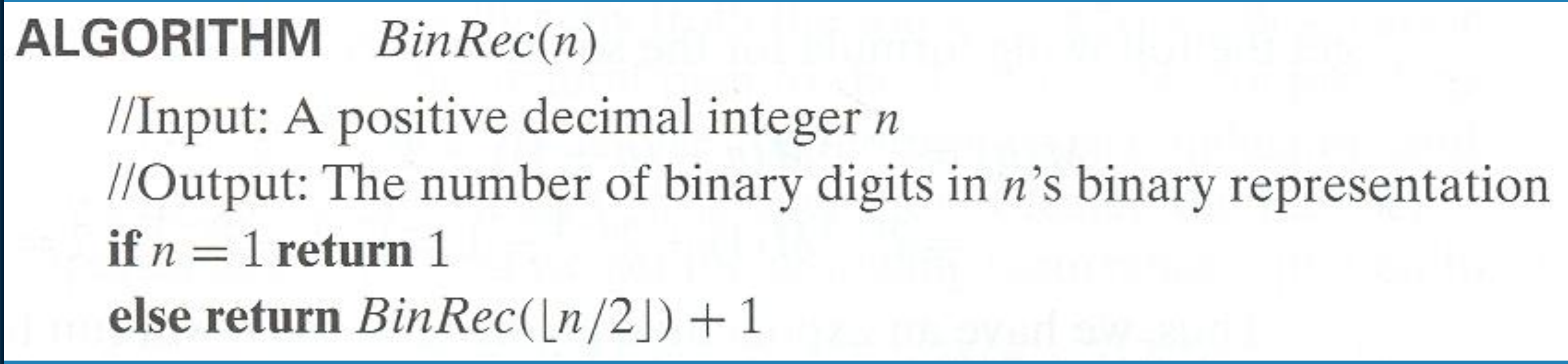
Solving recurrence for number of moves

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

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

The number of additions A(n) made by the algorithm

$$A(n) = A(n/2) + 1 \text{ for } n > 1$$

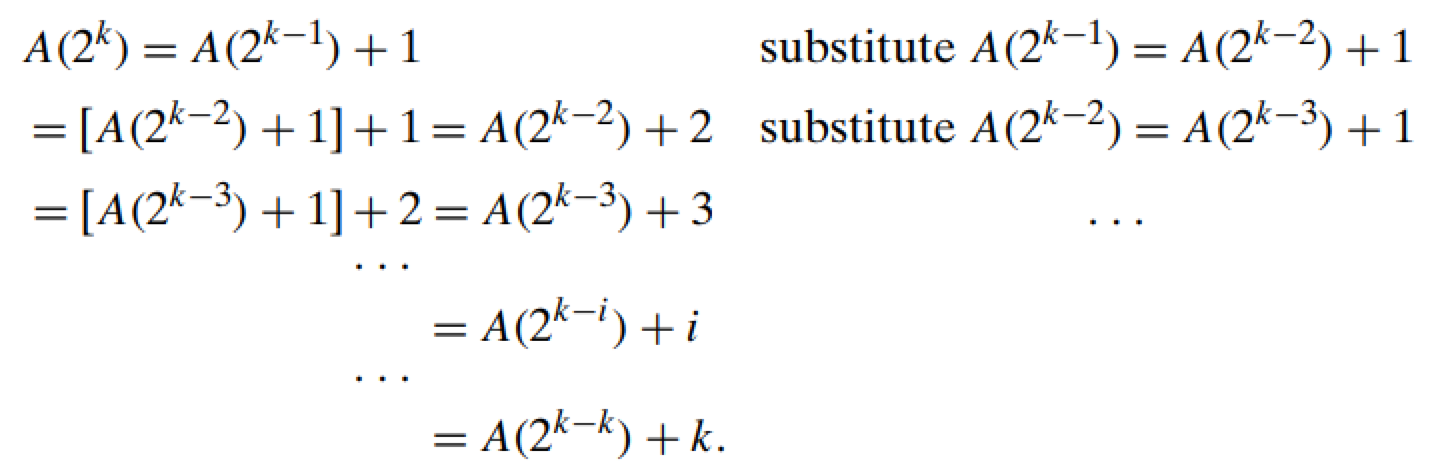
$$A(1) = 0$$

Let $n = 2^k$, So we have $A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,
$A(2^0) = 0$.

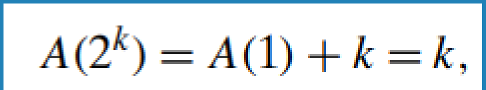
Example 3: Counting #bits (Cont.)

Using backward substitutions to solve

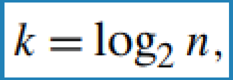
$$A(2^k) = A(2^{k-1}) + 1 \text{ for } k > 0,$$
$$A(2^0) = 0.$$



$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

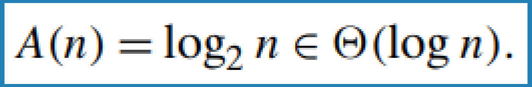


$$A(2^k) = A(1) + k = k,$$



$$k = \log_2 n,$$

$$\Rightarrow$$



$$A(n) = \log_2 n \in \Theta(\log n).$$

2.5 Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

The Fibonacci recurrence:

$F(n) = F(n\text{-}1) + F(n\text{-}2)$

$F(0) = 0$

$F(1) = 1$

General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$$

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

- Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

- Solve to obtain roots r_1 and r_2

- General solution to the recurrence

  if r_1 and r_2 are two distinct real roots: $X(n) = \alpha r_1^{\,n} + \beta r_2^{\,n}$

  if $r_1 = r_2 = r$ are two equal real roots: $X(n) = \alpha r^n + \beta n r^{\,n}$

- Particular solution can be found by using initial conditions

Application to the Fibonacci numbers

F(*n*) = F(*n*-1) + F(*n*-2) or F(*n*) - F(*n*-1) - F(*n*-2) = 0

Characteristic equation:

$$r^2 - r - 1 = 0,$$

Roots of the characteristic equation:

$$r_{1,2} = \frac{1 \pm \sqrt{1-4(-1)}}{2} = \frac{1 \pm \sqrt{5}}{2}.$$

General solution to the recurrence:

$$F(n) = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^n + \beta \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

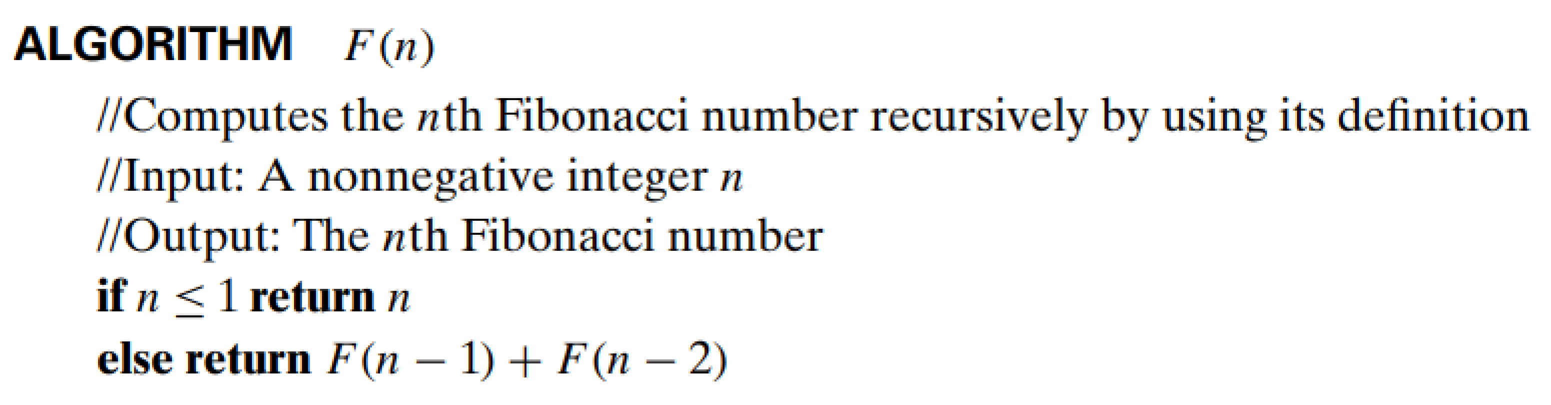
Particular solution for F(0) =0, F(1)=1:

$$F(n) = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n),$$

where $\phi = (1+\sqrt{5})/2 \approx 1.61803$ and $\hat{\phi} = -1/\phi \approx -0.61803$.[6]

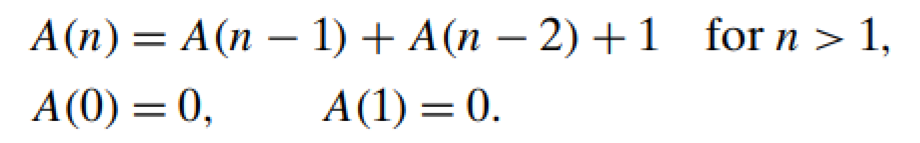
Computing Fibonacci numbers

1. **Definition-based recursive algorithm**



```
ALGORITHM  $F(n)$
    //Computes the $n$th Fibonacci number recursively by using its definition
    //Input: A nonnegative integer $n$
    //Output: The $n$th Fibonacci number
    if $n \le 1$ return $n$
    else return $F(n-1) + F(n-2)$
```

The numbers of additions satisfies the following recurrence equation.



$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

Computing Fibonacci numbers

Solve the following recurrence equation.

$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

Transform it into homogenous recurrence equation to solve

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

substituting $B(n) = A(n) + 1$:

$$\Rightarrow \quad B(n) - B(n-1) - B(n-2) = 0, \qquad B(0) = 1, \qquad B(1) = 1.$$

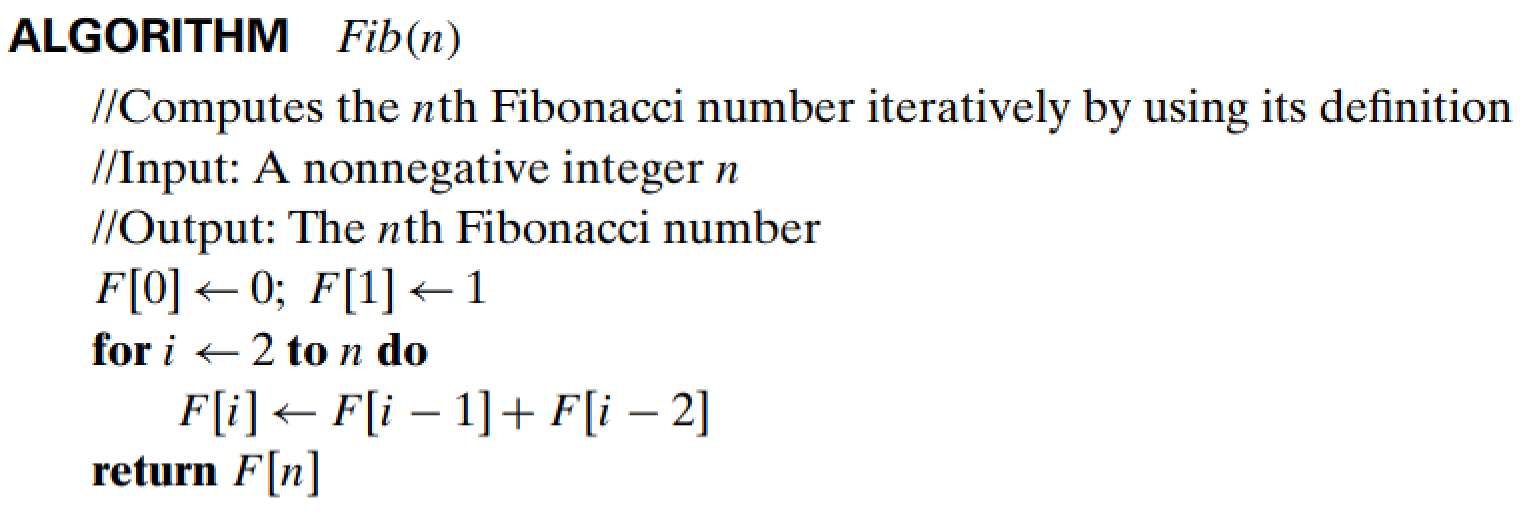
$\Rightarrow$ So $B(n) = F(n+1)$, and

$$A(n) = B(n) - 1 = F(n+1) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1.$$

$$\Rightarrow \quad A(n) \in \Theta(\phi^n)$$

Computing Fibonacci numbers

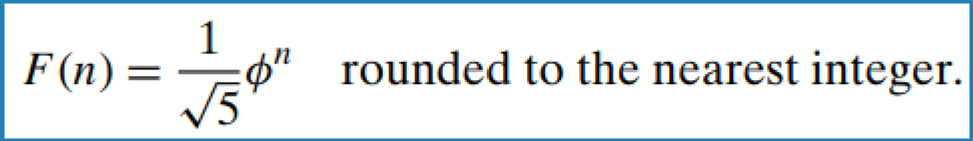
2. **Nonrecursive definition-based algorithm**



```
ALGORITHM  Fib(n)
    //Computes the nth Fibonacci number iteratively by using its definition
    //Input: A nonnegative integer n
    //Output: The nth Fibonacci number
    $F[0] \leftarrow 0;\ F[1] \leftarrow 1$
    for $i \leftarrow 2$ to $n$ do
        $F[i] \leftarrow F[i-1] + F[i-2]$
    return $F[n]$
```

Only n-1 additions!

3. **Explicit formula algorithm**



$F(n) = \frac{1}{\sqrt{5}}\phi^n$ rounded to the nearest integer.

Computing Fibonacci numbers

1. Definition-based recursive algorithm

2. Nonrecursive definition-based algorithm

3. Explicit formula algorithm

4. Logarithmic algorithm based on formula:

$$\begin{pmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n$$

for $n \geq 1$, assuming an efficient way of computing matrix powers.

Empirical Analysis of Algorithms

General Plan for the Empirical Analysis

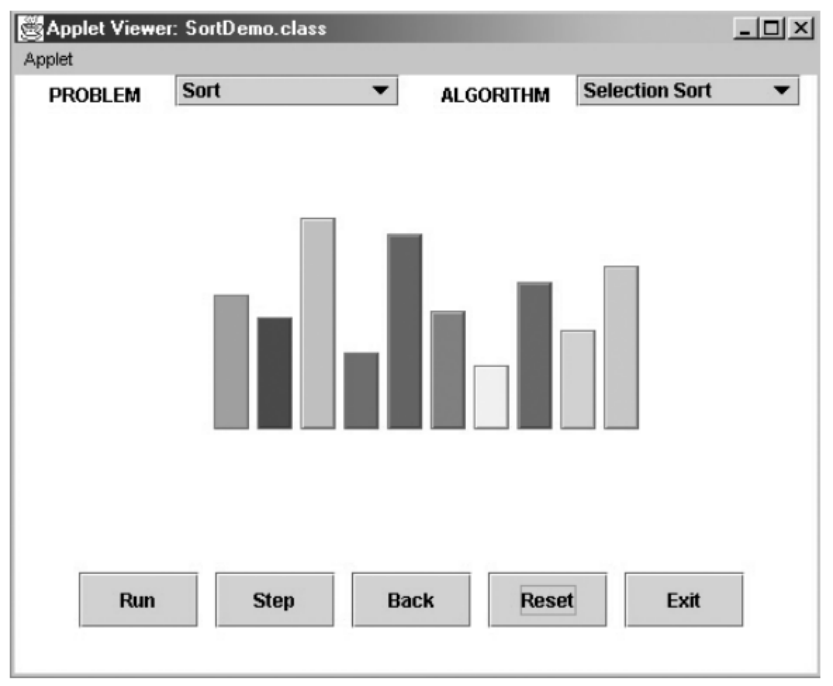
1. Understand the experiment's purpose.
2. Decide on the efficiency metric M to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the experimentation
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

Algorithm Visualization

- Algorithm visualization can be defined as the use of images to convey some useful information about algorithms.
- There are two principal variations of algorithm visualization
  - Static algorithm visualization, which shows an algorithm’s progress through a series of still images
  - Dynamic algorithm visualization, also called algorithm animation, which shows a continuous, movie-like presentation of an algorithm’s operations.

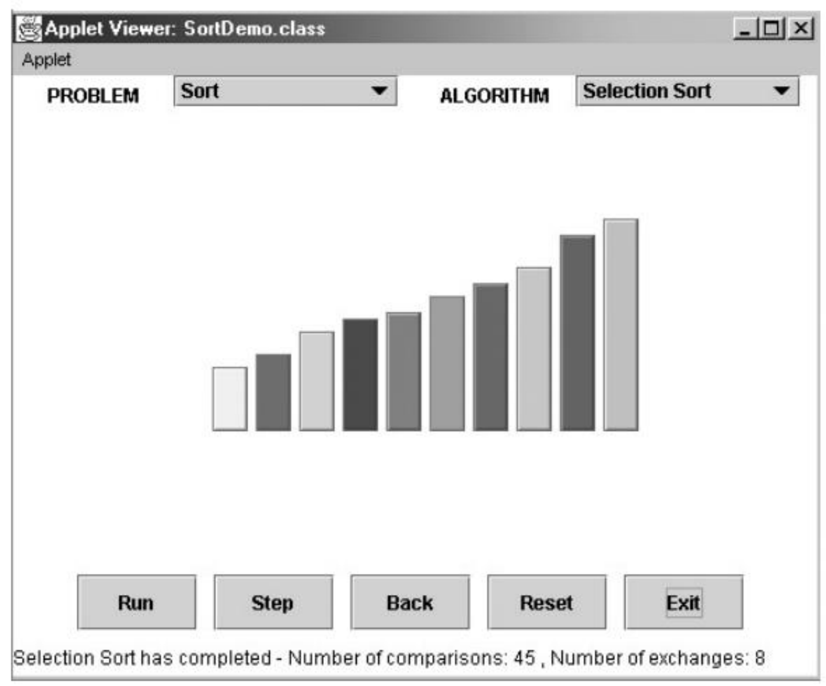
Algorithm Visualization: Example 1

Initial and final screens of a typical visualization of a sorting algorithm using the bar representation



PROBLEM Sort

ALGORITHM Selection Sort



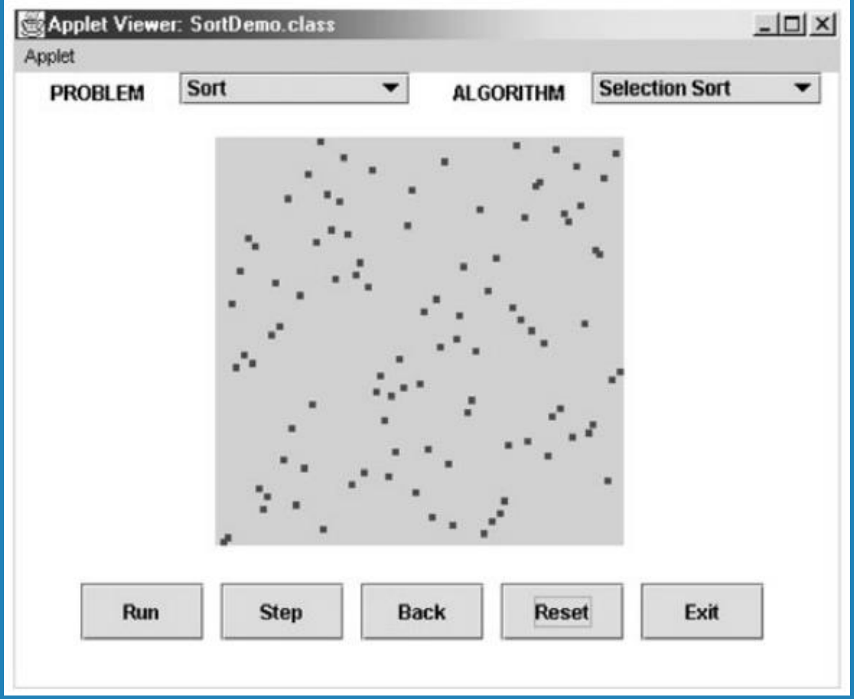
PROBLEM Sort

ALGORITHM Selection Sort

Selection Sort has completed - Number of comparisons: 45 , Number of exchanges: 8

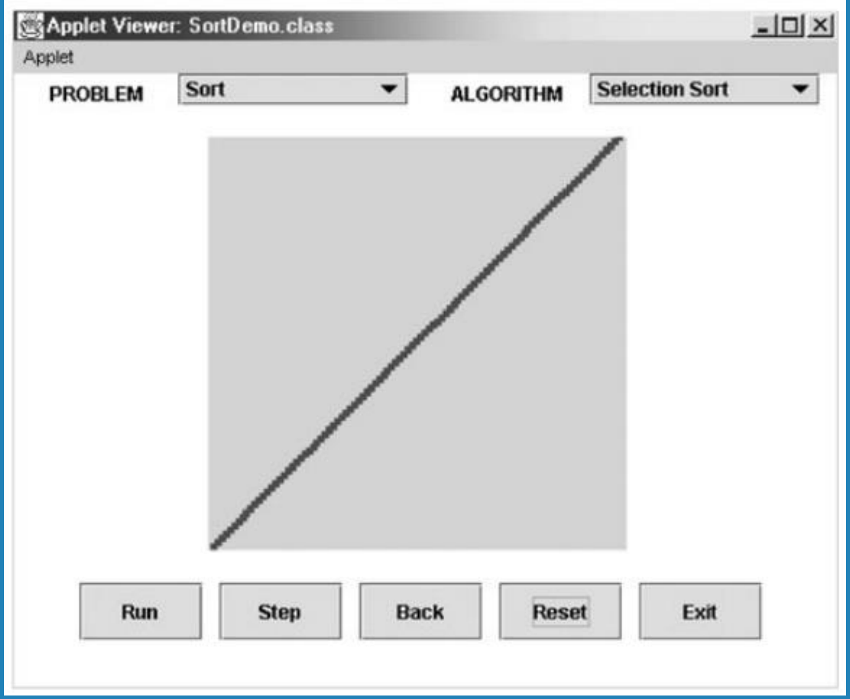
Algorithm Visualization: Example 2

Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation



PROBLEM Sort

ALGORITHM Selection Sort



PROBLEM Sort

ALGORITHM Selection Sort