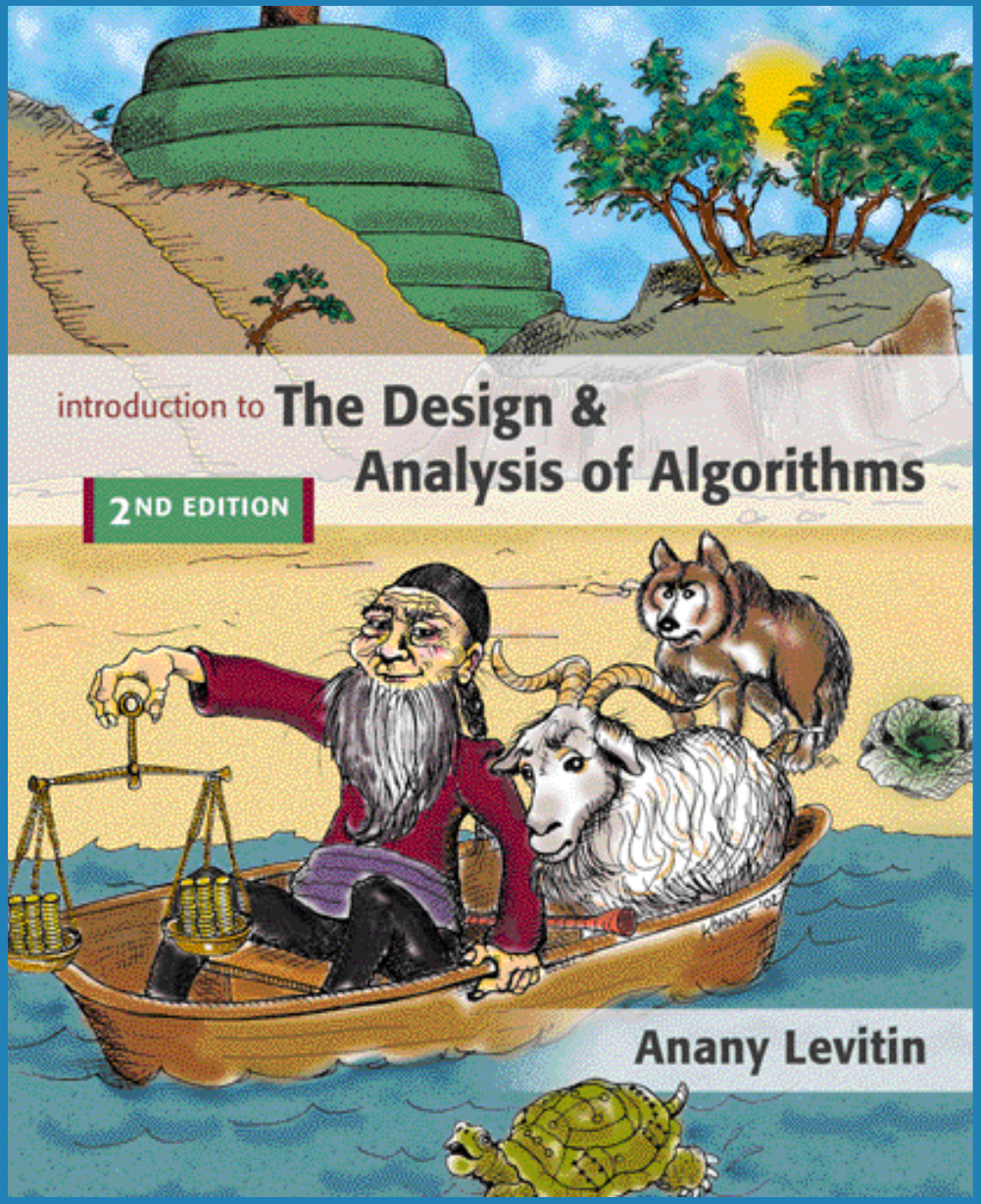
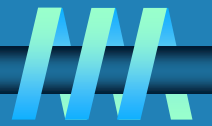
Chapter 2

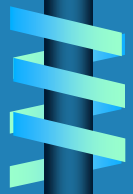
Fundamentals of the Analysis of Algorithm Efficiency



Analysis of algorithms



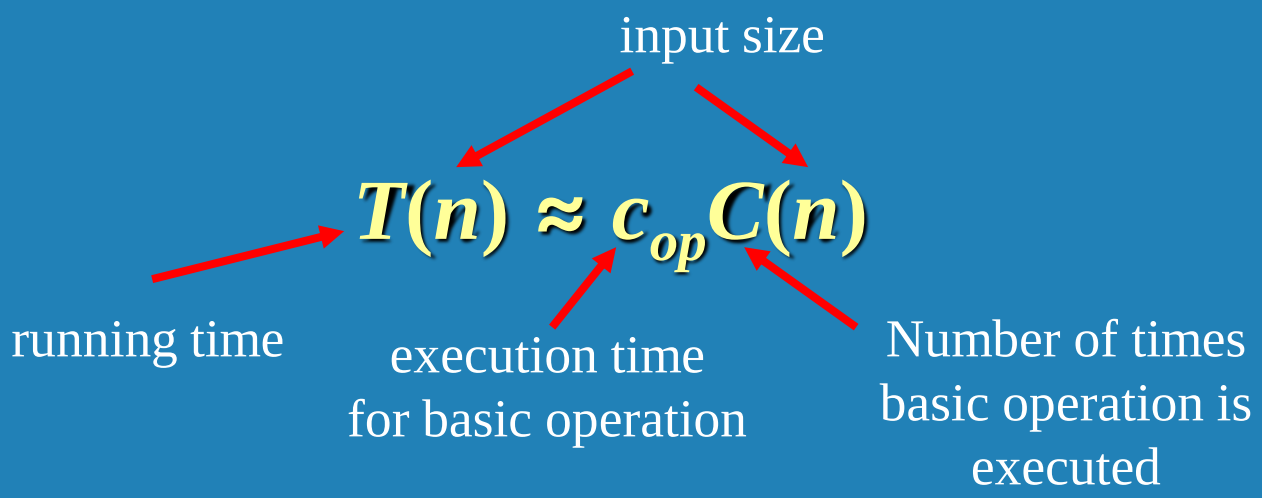
- **Issues:**
  - correctness
  - time efficiency
  - space efficiency
  - optimality
- **Approaches:**
  - theoretical analysis
  - empirical analysis



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*

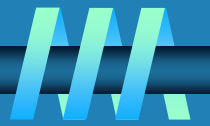
- ***Basic operation*: the operation that contributes most towards the running time of the algorithm**



Input size 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

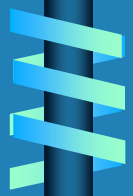
Empirical analysis of time efficiency



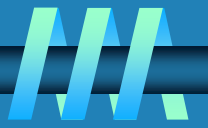
- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)

  or

  Count actual number of basic operation's executions
- Analyze the empirical data

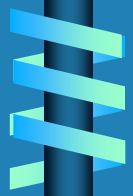


Best-case, average-case, worst-case

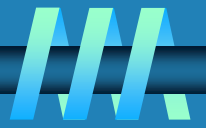


For some algorithms efficiency depends on form of input:

- **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
```

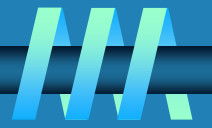
- **Worst case**
- **Best case**
- **Average case**

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

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

Big-oh

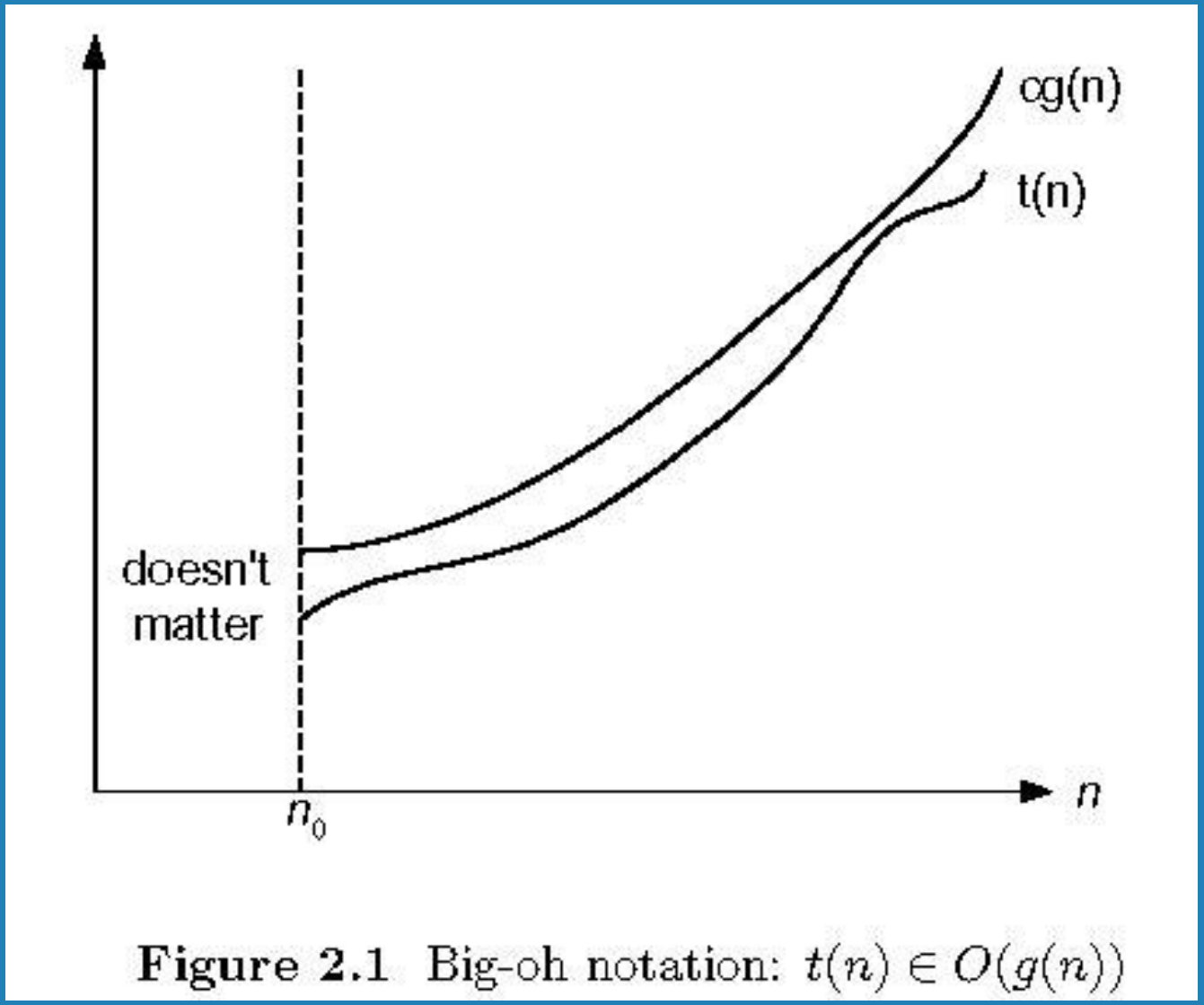


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

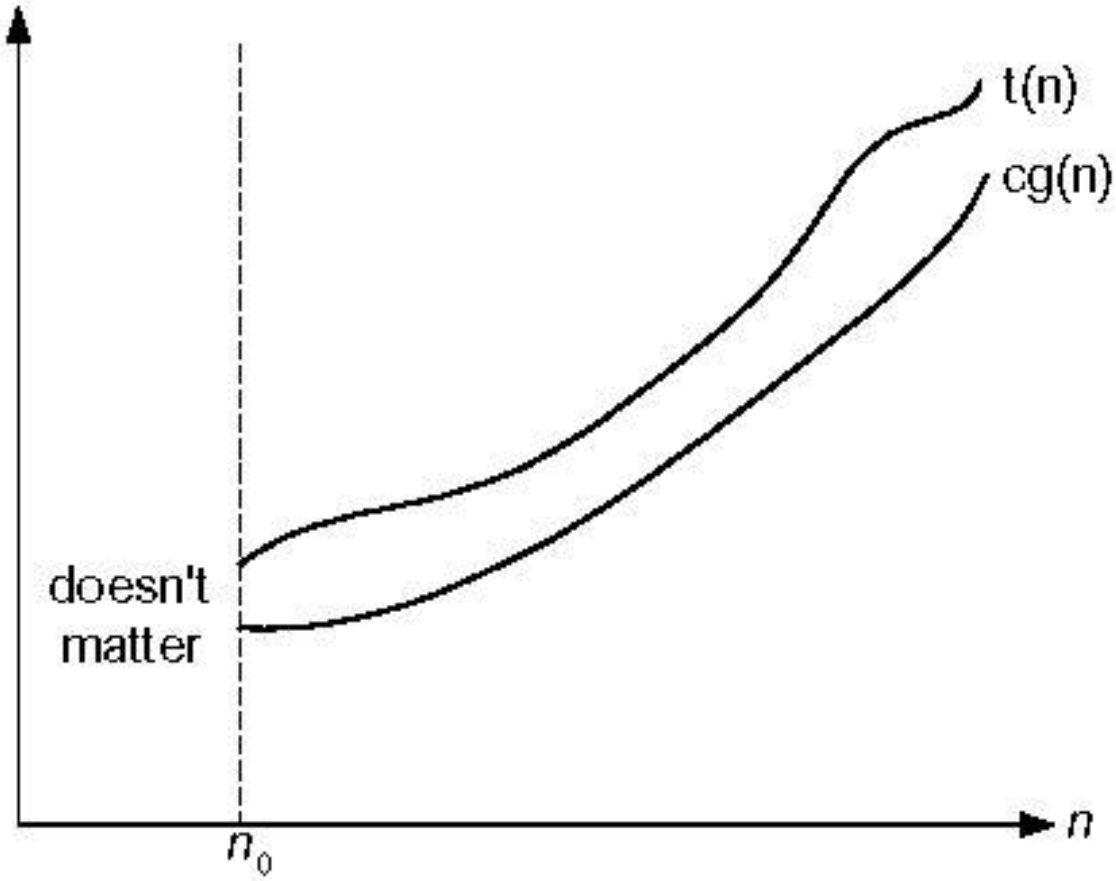


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

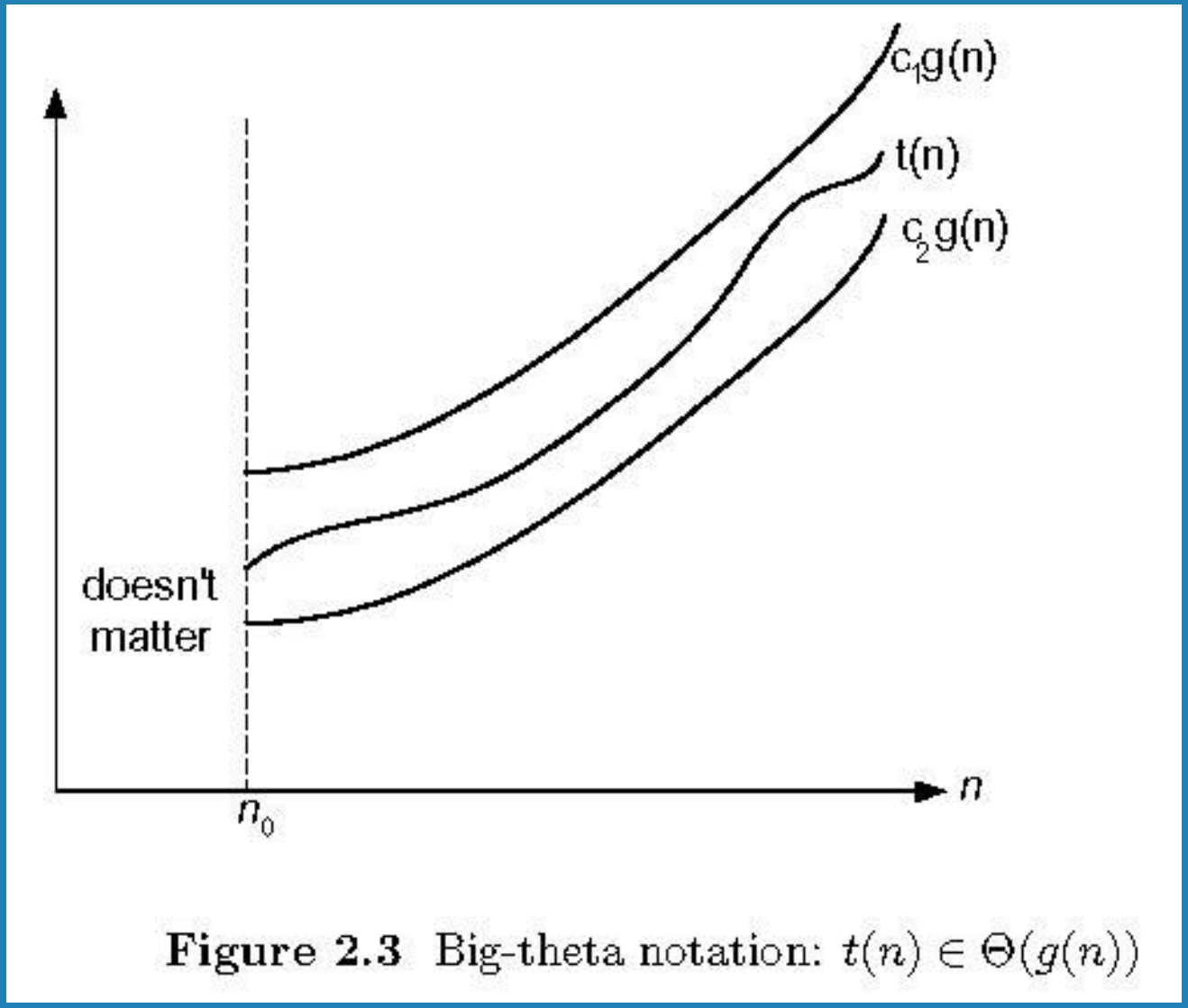
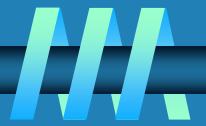


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition



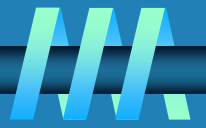
Definition: $f(n)$ **is in** $O(g(n))$ **if order of growth of** $f(n) \leq$ **order of growth of** $g(n)$ **(within constant multiple), i.e., there exist positive constant** c **and non-negative integer** n_0 **such that**

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ **is** $O(n^2)$
- $5n+20$ **is** $O(n)$

Establishing order of growth using limits

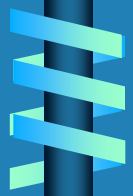


$$\lim_{n\to\infty} T(n)/g(n) =$$

- 0 order of growth of $T(n)$ < order of growth of $g(n)$
- $c > 0$ order of growth of $T(n)$ = order of growth of $g(n)$
- ∞ order of growth of $T(n)$ > order of growth of $g(n)$

Examples:

- $10n$ vs. n^2
- $n(n+1)/2$ vs. n^2



THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some non-negative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \geq \max\{n_1, n_2\}$ so that we can use both inequalities. Adding them yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\leq c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

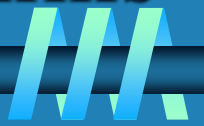
So what does this property imply for an algorithm that comprises two consecutively executed parts? It implies that the algorithm's overall efficiency is determined by the part with a higher order of growth, i.e., its least efficient part:

$$\left.\begin{array}{|l} t_1(n) \in O(g_1(n)) \\ \hline t_2(n) \in O(g_2(n)) \end{array}\right\} \quad t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

Basic asymptotic efficiency classes

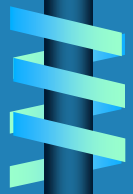
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

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.



Useful summation formulas and rules

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

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

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

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

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

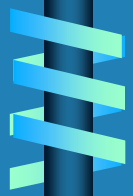
In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

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

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 ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```



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

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

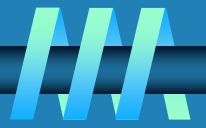
Example 5: Counting binary digits

ALGORITHM *Binary*(n)

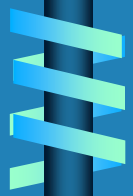
```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ← 1
while n > 1 do
    count ← count + 1
    n ← ⌊n/2⌋
return count
```

It cannot be investigated the way the previous examples are.

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.
- Set up a recurrence relation with an appropriate initial condition expressing the number of times the basic op. is executed.
- Solve the recurrence by backward substitutions or another method.



Example 1: Recursive evaluation of $n!$

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

Recursive definition of $n!$: $F(n) = F(n-1) * n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

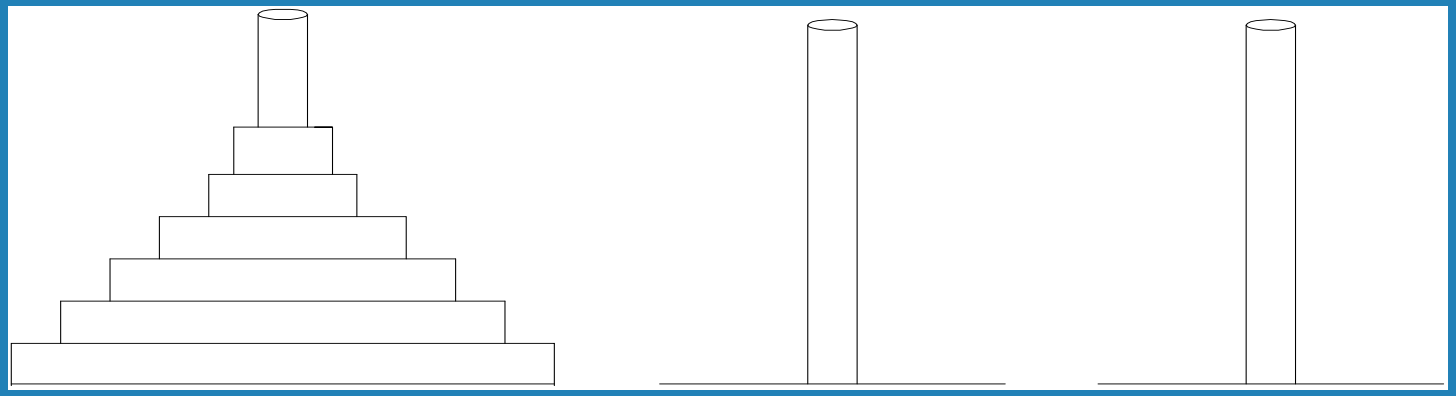
```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) * n
```

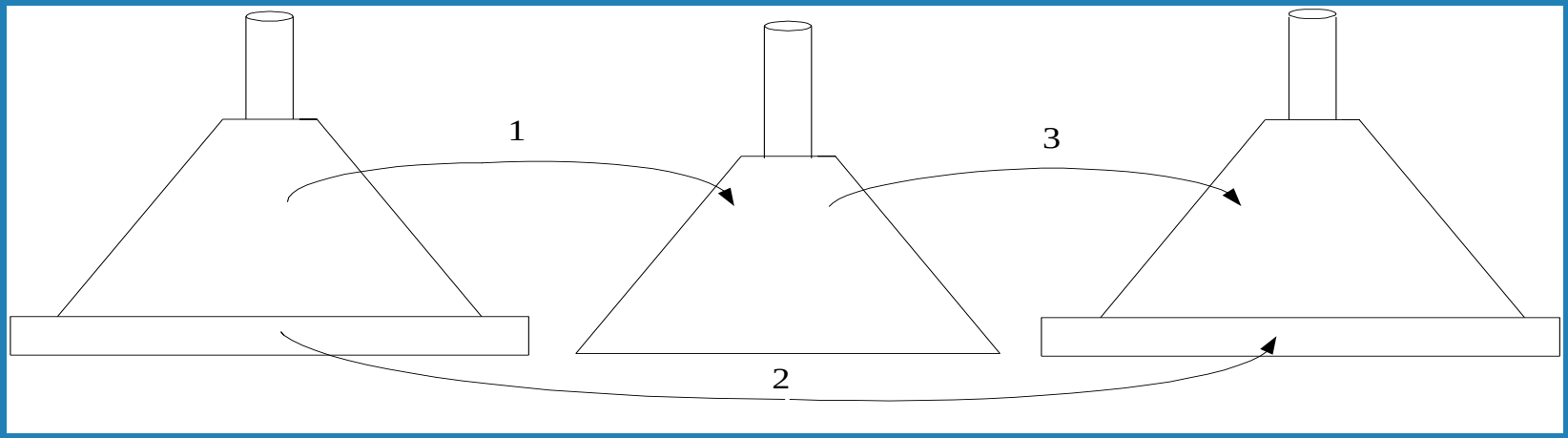
Size:

Basic operation:

Recurrence relation:

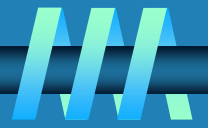
Example 2: The Tower of Hanoi Puzzle





Recurrence for number of moves:

Solving recurrence for number of moves



M(*n*) = 2M(*n*-1) + 1, M(1) = 1

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

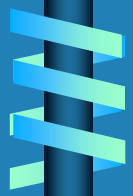
$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1,$$
$$M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

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

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

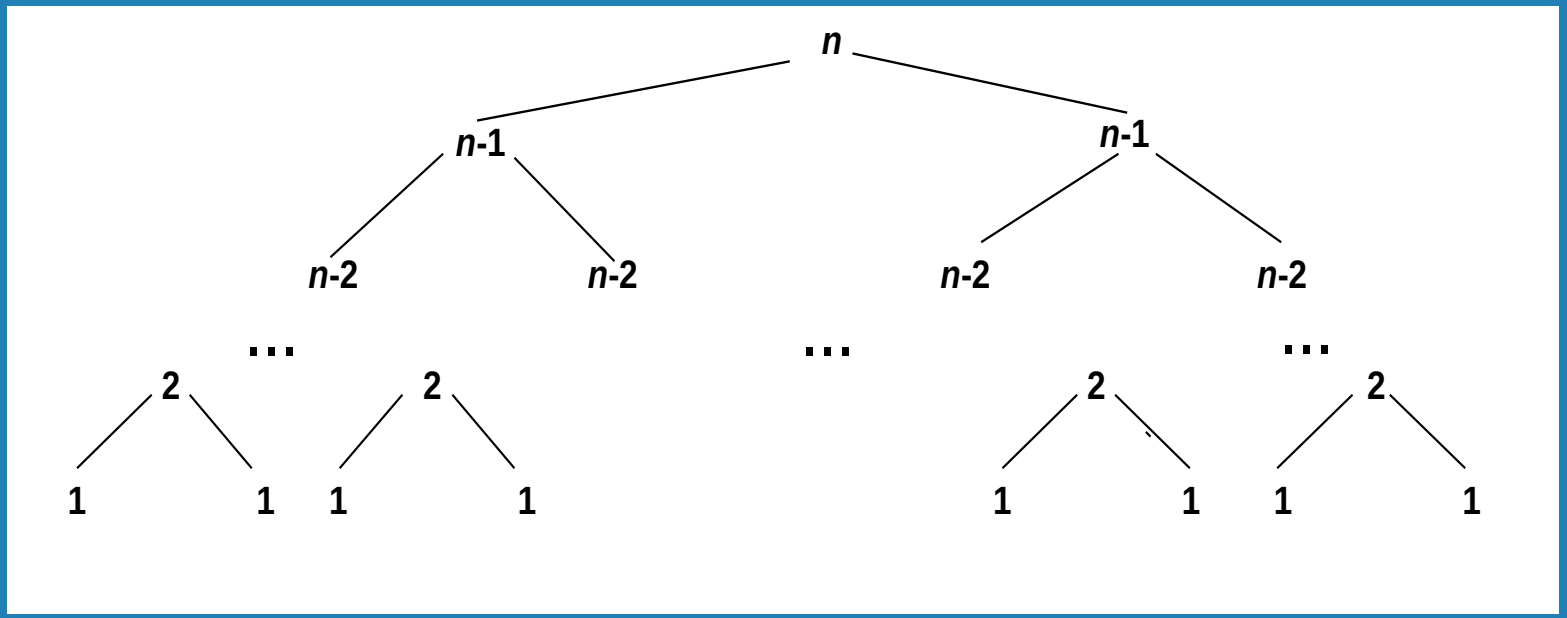
$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$



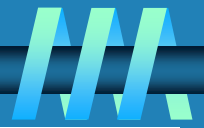
Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence

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

Tree of calls for the Tower of Hanoi Puzzle



Example 3: Counting #bits



ALGORITHM *BinRec*(n)

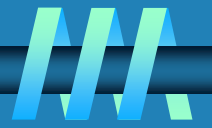
//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

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

Exercise



1. For each of the following algorithms, indicate (i) a natural size metric for its inputs, (ii) its basic operation, and (iii) whether the basic operation count can be different for inputs of the same size:

a. computing the sum of n numbers

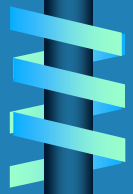
b. computing $n!$

c. finding the largest element in a list of n numbers

d. Euclid's algorithm

e. sieve of Eratosthenes

f. pen-and-pencil algorithm for multiplying two n-digit decimal integers



• • •

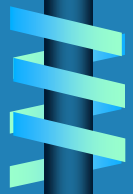
Use the informal definitions of O, Θ, and Ω to determine whether the following assertions are true or false.

a. $n(n+1)/2 \in O(n^3)$

b. $n(n+1)/2 \in O(n^2)$

c. $n(n+1)/2 \in \Theta(n^3)$

d. $n(n+1)/2 \in \Omega(n)$



Indicate whether the first function of each of the following pairs has a smaller, same, or larger order of growth (to within a constant multiple) than the second function.

a. $n(n+1)$ and $2000n^2$

b. $100n^2$ and $0.01n^3$

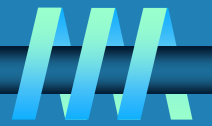
c. $\log_2 n$ and $\ln n$

d. $\log_2^2 n$ and $\log_2 n^2$

e. 2^{n-1} and 2^n

f. $(n-1)!$ and $n!$

....



- **ALGORITHM** *S(n)*

//Input: A positive integer *n*

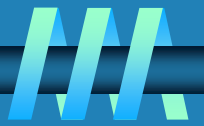
//Output: The sum of the first *n* cubes

if $n = 1$ **return** 1

else return $S(n-1) + n * n * n$

Set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed.

.....



1. **x(n)=x(n-1)+n for n>0 , x(0)=0**
2. $x(n) = x(n/2) + n$ for $n > 1$, $x(1) = 1$
3. $x(n) = 3x(n-1)$ for $n > 1$, $x(1) = 4$