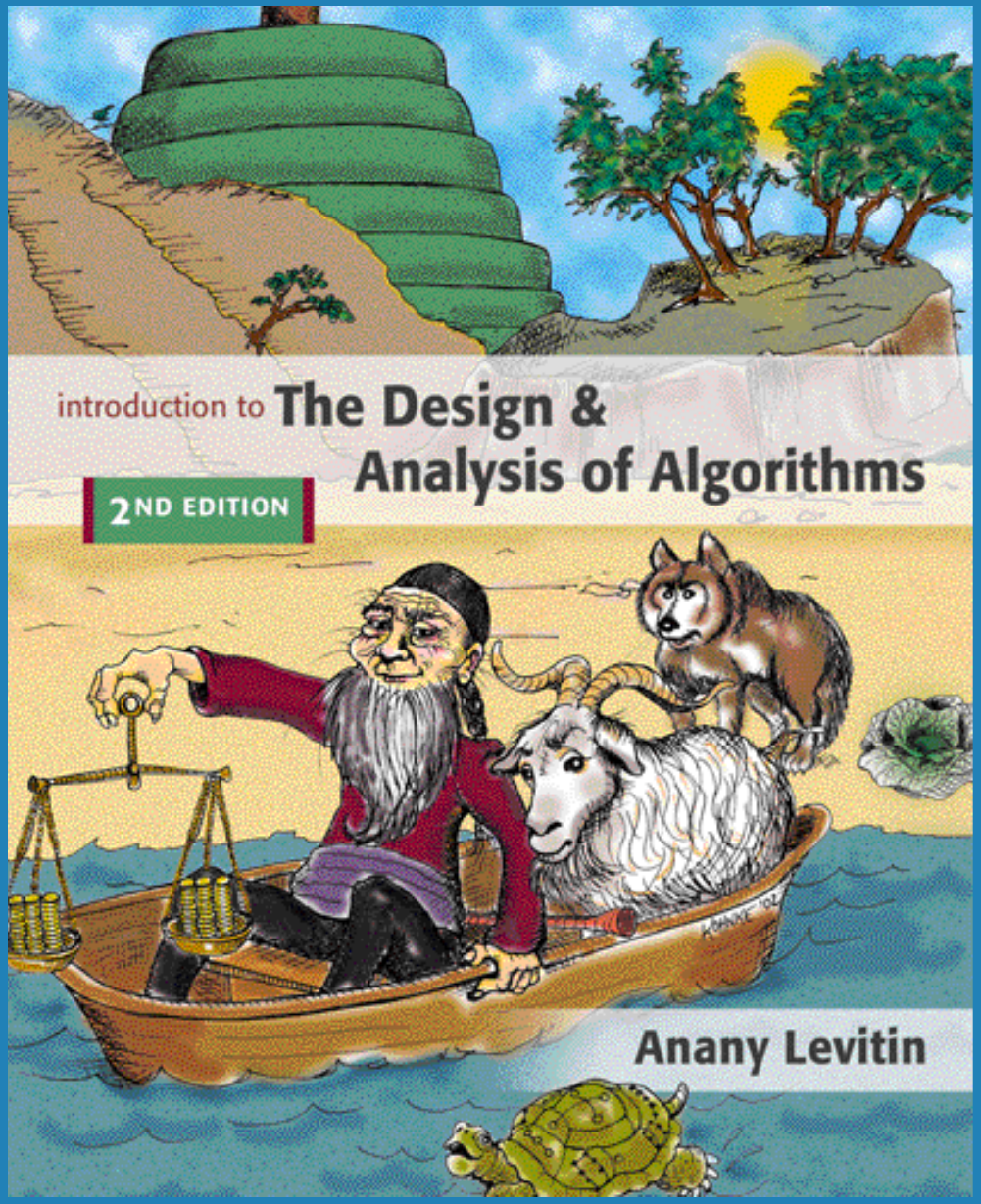
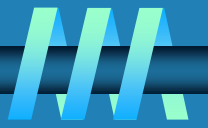
Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency

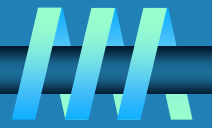


Analysis Framework



- In what ways can we compare algorithms?
  - Time & Space efficiency are most common
  - Others: Cost, Power, …
- Characteristics of a good framework
  - Independence from
    - Programmer implementation
    - Language / compiler optimizations
  - Dependence on
    - Input
    - Critical operation of algorithm
    - Frequency of critical operation execution
    - Number of “things” stored in memory relative to input

Input Size



- **Goal:**
  - **express efficiency of an algorithm relative to the size of its encoded input.**
  - **We shall assume that all input to our algorithms is encoded efficiently.**
  - **Example, integers will typically be encoded in binary, which implies the following:**

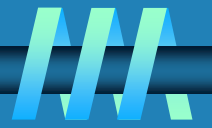
Let b represent number of bits for the encoding of n in binary:

$$b = \lfloor \log_2 n \rfloor + 1 \qquad (1)$$

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of n items	Number of 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

Measuring Running Time



- **Basic Operation**
  - The operation contributing the most to total runtime
  - Frequency of execution depends on input

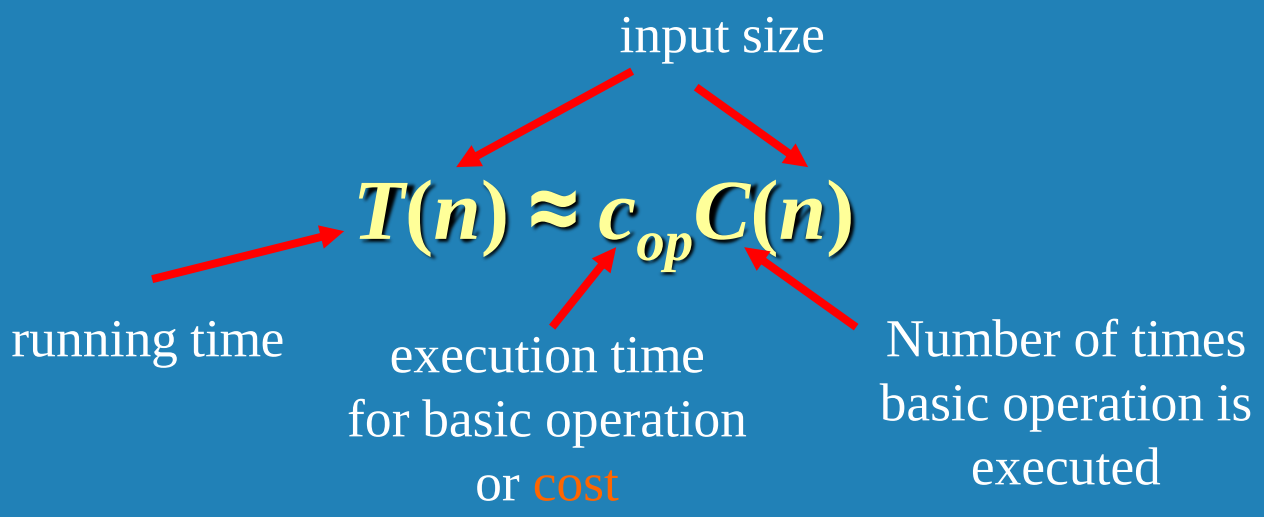
Let $c_{\_op}$ be the execution time of an algorithms basic operation on a particular computer, and let *C(n)* be the number of times this operation is run on input n, then we can estimate the running time *T(n)* of a program:

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

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 the most towards the running time of the algorithm**



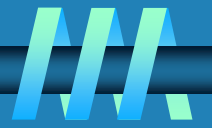
Note: Different basic operations may cost differently!

Orders of Growth

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

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}		

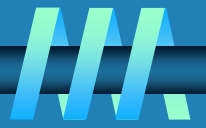
Best-case, average-case, worst-case



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. So, avg = expected under uniform distribution.**

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** — n key comparisons
- **Best case** — 1 comparisons
- **Average case** — (n+1)/2

$$C_{avg}(n) = [\,1 \cdot p/n + 2 \cdot p/n + 3 \cdot p/n + \ldots + i \cdot p/n + \ldots + n \cdot p/n\,] + n \cdot (1 - p)$$

$$= p/n\,[\,1 + 2 + 3 + \ldots + n\,] + n(1 - p)$$

$$= p/n \cdot n(n+1)/2 + n(1 - p)$$

$$= p\,(n+1)/2 + n(1 - p)$$

Asymptotic Notations and Basic efficiency classes

- **Informal introduction**

O(g(n)) is the set of all functions with a smaller or same order of growth as g(n).

$n \in O(n^2)$

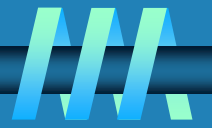
Ω(g(n)), stands for the set of all functions with larger or same order of growth as g(n).

$n^3 \in \Omega(n^2)$

Θ(g(n)) is the set of all functions that have same order of growth as g(n).

$7n^2 \in \Theta(n^2)$

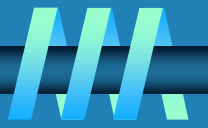
Asymptotic order of growth



A way of comparing functions that ignores constant factors and small input sizes .

- $O(g(n))$: class of functions $t(n)$ that grow no faster than $g(n)$
- $\Theta(g(n))$: class of functions $t(n)$ that grow at same rate as $g(n)$
- $\Omega(g(n))$: class of functions $t(n)$ that grow at least as fast as $g(n)$

Establishing order of growth using the definition



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

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

Examples:

- $10n \in O(n^2)$

- $5n+20 \in O(n)$

Big-oh

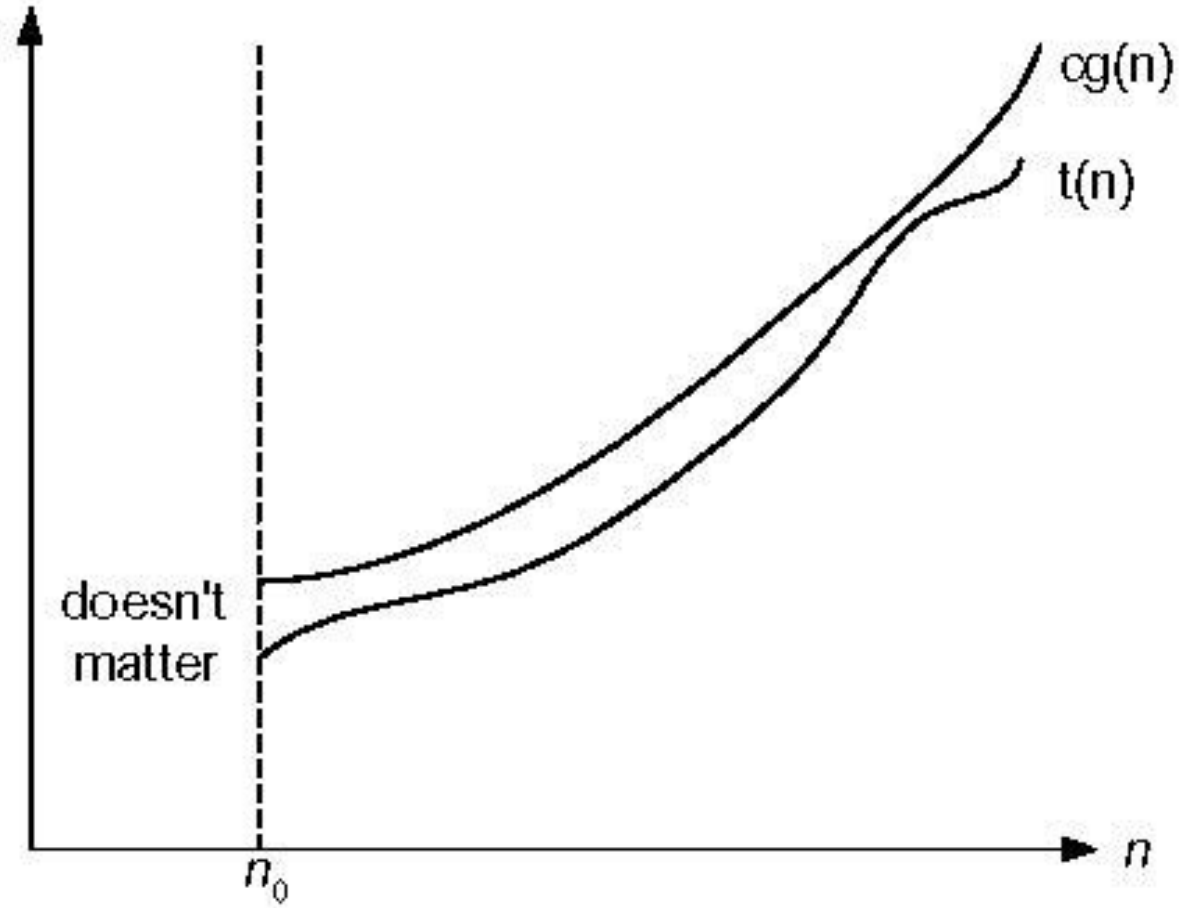
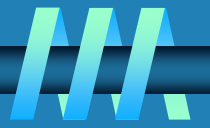


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

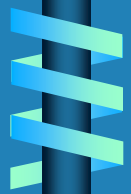
Ω-notation



- **Formal definition**
  - **A function *t(n)* is said to be in Ω*(g(n))*, denoted *t(n)* ∈ Ω*(g(n))*, if *t(n)* is bounded below by some constant multiple of *g(n)* for all large *n*, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that**

    $t(n) \geq cg(n)$ for all $n \geq n_0$

- **Exercises: prove the following using the above definition**
  - $10n^2 \in \Omega(n^2)$
  - $0.3n^2 - 2n \in \Omega(n^2)$
  - $0.1n^3 \in \Omega(n^2)$



Big-omega

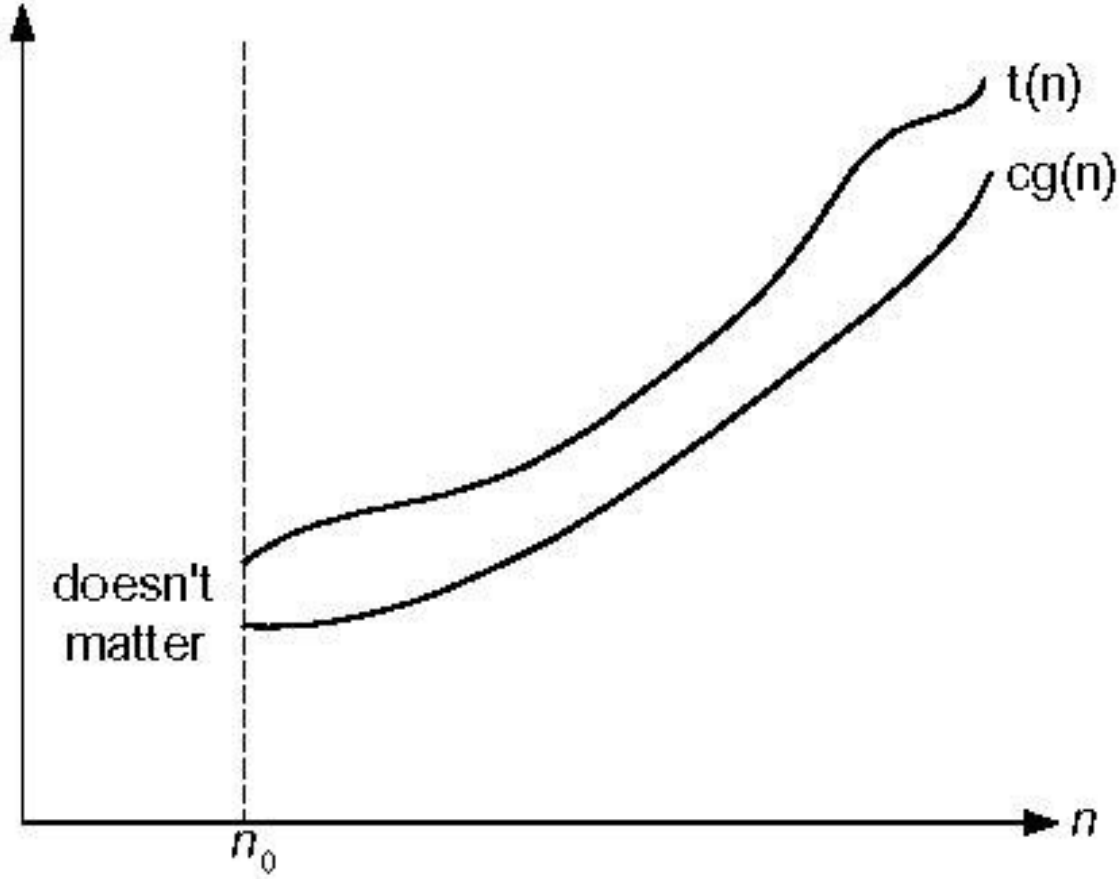


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Θ-notation

- **Formal definition**
  - **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 nonnegative integer n_0 such that**

$$c_2\, g(n) \le t(n) \le c_1\, g(n) \text{ for all } n \ge n_0$$

  - $10n^2 \in \Theta(n^2)$
  - $0.3n^2 - 2n \in \Theta(n^2)$
  - $(1/2)n(n+1) \in \Theta(n^2)$

Big-theta

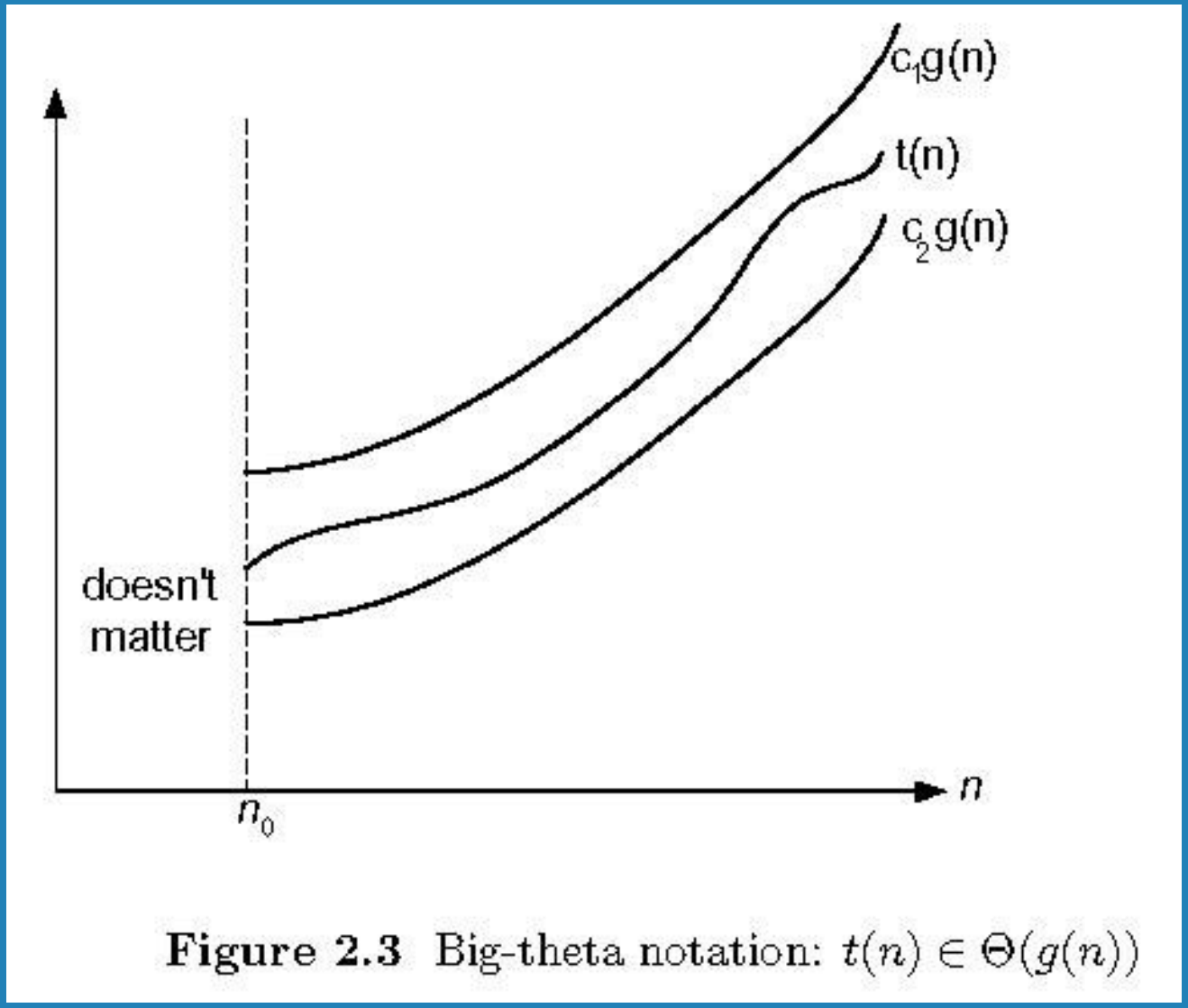
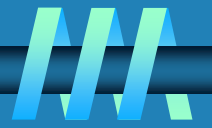


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Exercises

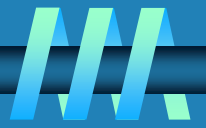


1. **Use the informal definition of O, Θ and Ω to determine whether the following assertions are true or false.**

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

Theorem



- **If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\})$.**

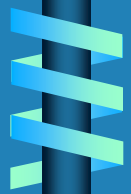
($a_1 <= b_1$ and $a_2 <= b_2$ then $a_1 + a_2 <= 2 \max\{b_1, b_2\}$

Proof: There exist constants c_1, c_2, n_1, n_2 such that

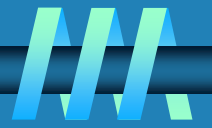
$$t_1(n) \leq c_1 * g_1(n), \quad \text{for all } n \geq n_1$$
$$t_2(n) \leq c_2 * g_2(n), \quad \text{for all } n \geq n_2$$

Let us denote $c_3 = max\{c_1, c_2\}$ and $n >= max\{n_1, n_2\}$. Then
Adding the two inequalities above yields the following:

$$t_1(n) + t_2(n) <= c_1 g_1(n) + c_2 g_2(n)$$



Continued….



$<= c_3 g_1(n) + c_3\, g_2(n) = c_3\, [g_1(n) + g_2(n)]$

$<= c_3\ 2 \max\{g1(n)\ ,\ g_2(n)\ \}$

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.

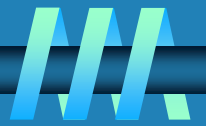
Establishing order of growth using limits

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

Examples:

- $10n$ vs. n^2

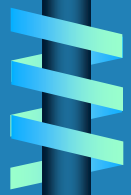
- $n(n+1)/2$ vs. n^2

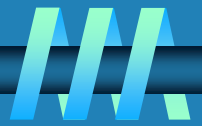


EXAMPLE 1 Compare the orders of growth of $\frac{1}{2}n(n-1)$ and n^2. (This is one of the examples we used at the beginning of this section to illustrate the definitions.)

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty} (1-\frac{1}{n}) = \frac{1}{2}.$$

Since the limit is equal to a positive constant, the functions have the same order of growth or, symbolically, $\frac{1}{2}n(n-1) \in \Theta(n^2)$. ■

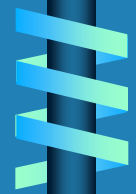


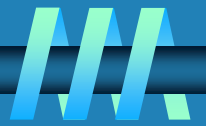


EXAMPLE 2 Compare the orders of growth of $\log_2 n$ and $\sqrt{n}$. (Unlike Example 1, the answer here is not immediately obvious.)

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0.$$

Since the limit is equal to zero, $\log_2 n$ has a smaller order of growth than $\sqrt{n}$. (Since $\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = 0$, we can use the so-called ***little-oh notation***: $\log_2 n \in o(\sqrt{n})$. Unlike the big-Oh, the little-oh notation is rarely used in analysis of algorithms.)





EXAMPLE 3 Compare the orders of growth of $n!$ and 2^n. (We discussed this informally in Section 2.1.) Taking advantage of Stirling's formula, we get

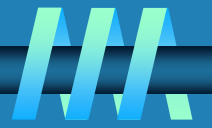
$$\lim_{n\to\infty} \frac{n!}{2^n} = \lim_{n\to\infty} \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n} = \lim_{n\to\infty} \sqrt{2\pi n}\frac{n^n}{2^n e^n} = \lim_{n\to\infty} \sqrt{2\pi n}\left(\frac{n}{2e}\right)^n = \infty.$$

Thus, though 2^n grows very fast, $n!$ grows still faster. We can write symbolically that $n! \in \Omega(2^n)$; note, however, that while the big-Omega notation does not preclude the possibility that $n!$ and 2^n have the same order of growth, the limit computed here certainly does. ■

Basic asymptotic efficiency classes

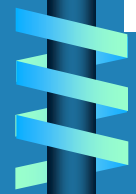
1	constant
$\log n$	logarithmic
n	linear
$n \log n$	n-log-n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Mathematical analysis of Nonrecursive algorithms



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 inner-most 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.



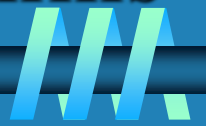
Mathematical analysis of Nonrecursive algorithms

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
```

T(n) = $\Sigma_{1 \le i \le n-1} 1 = n-1 = \Theta(n)$ comparisons

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

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

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

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

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

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

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

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

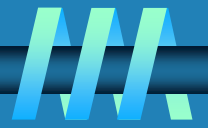
Example 2: Element uniqueness problem

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
```

n^2

Example 3: Matrix multiplication



ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm

//Input: Two n-by-n matrices A and B

//Output: Matrix $C = AB$

for $i \leftarrow 0$ **to** $n-1$ **do**

  for $j \leftarrow 0$ **to** $n-1$ **do**

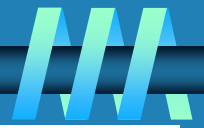
    $C[i, j] \leftarrow 0.0$

    for $k \leftarrow 0$ **to** $n-1$ **do**

      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

return C

Example 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.

The halving game: Find integer i such that $n/2^i \leq 1$.

Answer: $i \leq \log n$. ***So,*** $T(n) = \Theta(\log n)$ divisions.

Mathematical analysis of recursive algorithms

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
```

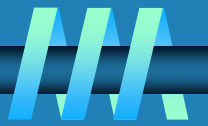
Size: n

Basic operation: multiplication

Recurrence relation: $M(n) = M(n-1) + 1$

$M(0) = 0$

Solving the recurrence for M(*n*)



M(*n*) = M(*n*-1) + 1, M(0) = 0

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

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

= (M(n-3) + 1) + 2 = M(n-3) + 3

…

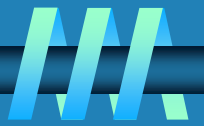
= M(n-i) + i

= M(0) + n

= n

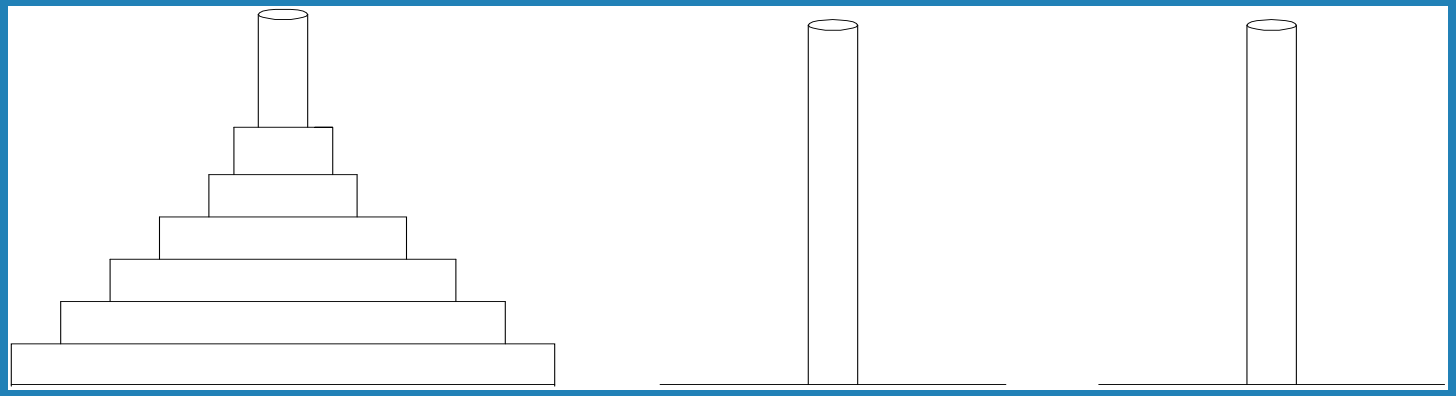
The method is called backward substitution.

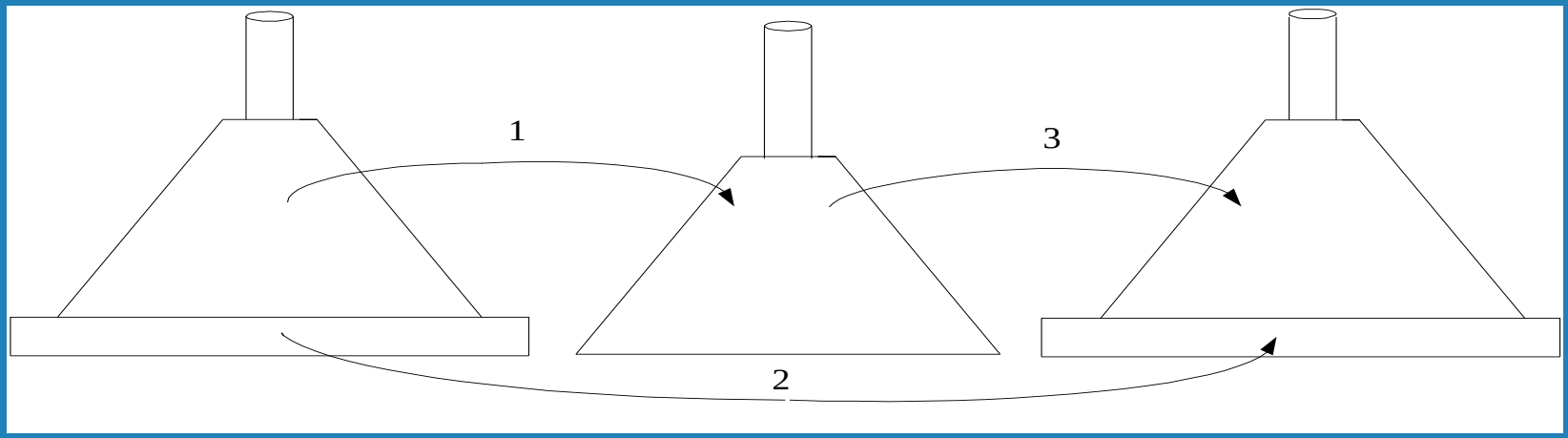
Plan for Analysis of Recursive Algorithms



- 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 2: The Tower of Hanoi Puzzle

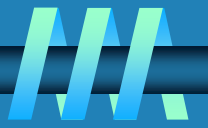




Recurrence for number of moves:

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

Solving recurrence for number of moves



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

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

= 2(2M(n-2) + 1) + 1 = 2^2*M(n-2) + 2^1 + 2^0

= 2^2*(2M(n-3) + 1) + 2^1 + 2^0

= 2^3*M(n-3) + 2^2 + 2^1 + 2^0

= …

= 2^(n-1)*M(1) + 2^(n-2) + … + 2^1 + 2^0

= 2^(n-1) + 2^(n-2) + … + 2^1 + 2^0

= 2^n - 1

Tree of calls for the Tower of Hanoi Puzzle