

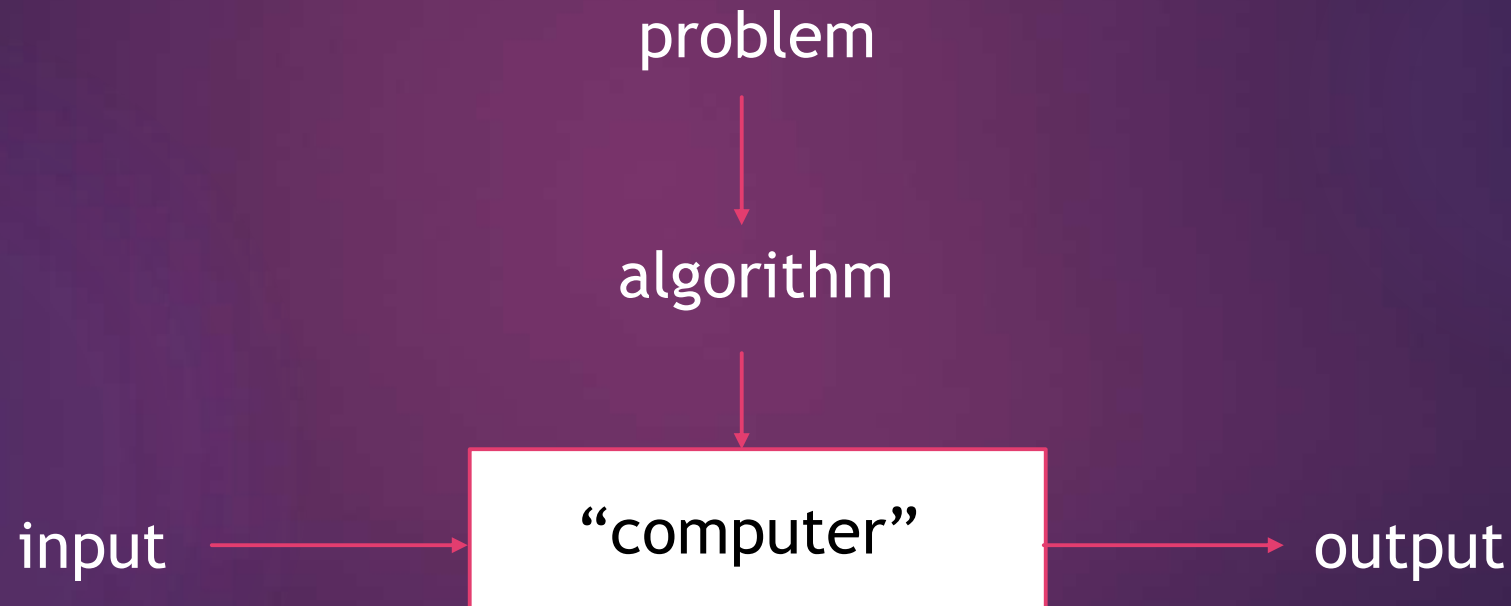


DESIGN AND ANALYSIS OF ALGORITHMS

UNIT 1

What is an algorithm?

An algorithm is a sequence of **unambiguous** instructions for solving a problem, i.e., for obtaining a required output for any **legitimate input** in a **finite amount of time**.



Euclid's Algorithm

Problem: Find $\gcd(m,n)$, the greatest common divisor of two nonnegative, not both zero integers m and n

Solution:

Euclid's algorithm is based on repeated application of equality

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example:

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

Two Descriptions of Euclid's Algorithm

Step 1 If $n = 0$, return m and stop; otherwise go 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 non - negative, both not zero integers

//Output: GCD of m and n

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Other methods for computing $\gcd(m,n)$

Consecutive Integer Checking Algorithm

- Step 1 Assign the value of $\min\{m,n\}$ to t
- Step 2 Divide m by t . If the remainder is 0, go to Step 3; otherwise, go to Step 4
- Step 3 Divide n by t . If the remainder is 0, return t and stop; otherwise, go to Step 4
- Step 4 Decrease t by 1 and go to Step 2

Other methods for $\text{gcd}(m,n)$ [cont.]

Middle-school procedure

- Step 1 Find the prime factorization of m
- Step 2 Find the prime factorization of n
- Step 3 Find all the common prime factors
- Step 4 Compute the product of all the common prime factors and return it as $\text{gcd}(m,n)$

Is this an algorithm?

Sieve of Eratosthenes

ALGORITHM Sieve (n)

//Implements the Sieve of Eratosthenes

//Input: Integer $n \geq 2$

//Output: List of primes less than or equal to n

for $p \leftarrow 2$ to n do $A[p] \leftarrow p$

for $p \leftarrow 2$ to $\lfloor n \rfloor$ do

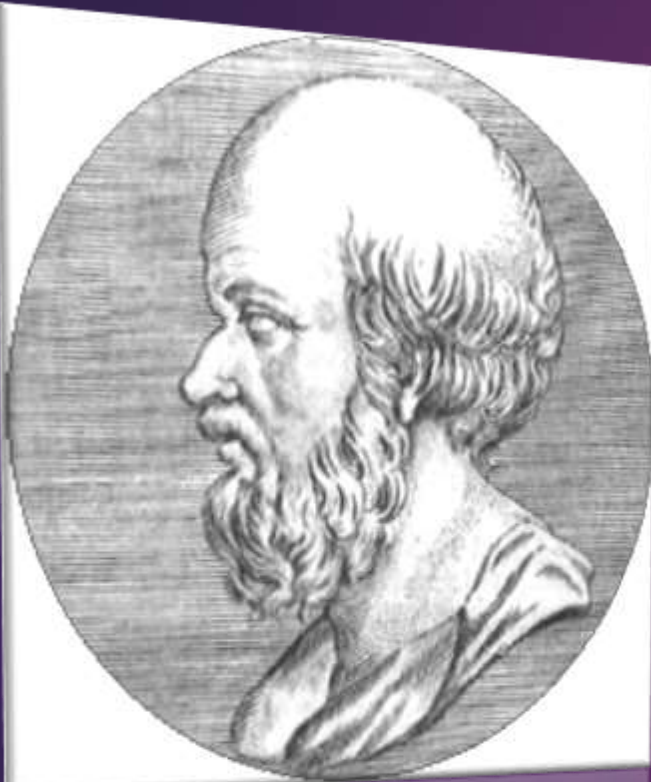
 if $A[p] \neq 0$ // p hasn't been previously eliminated from the list

$j \leftarrow p * p$

 while $j \leq n$ do


$A[j] \leftarrow 0$ //mark element as eliminated

$j \leftarrow j + p$






FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING



There are n locker doors in a hallway, numbered sequentially from 1 to n . Initially, all the locker doors are closed. You make n passes by the locker, each time starting with locker #1. On the i^{th} pass, $i = 1, 2, \dots, n$, you toggle the door of every i^{th} locker: if the door is closed, you open it; if it is open, you close it. After the last pass, which locker doors are open and which are closed? How many of them are open?



If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

IMPORTANT PROBLEM TYPES

- Sorting
- Searching
- String Processing
- Graph Problems
- Combinatorial Problems
- Geometric Problems
- Numerical Problems

BUBBLE SORT

Bubble Sort – Idea

- Compare adjacent elements of the list and exchange them if they are out of order.
- By doing it repeatedly, we end up bubbling the largest element to the last position on the list.
- The next pass bubbles up the second largest element and so on and after $n - 1$ passes, the list is sorted.
- Pass i ($0 \leq i \leq n - 2$) can be represented as follows:

$$A[0], A[1], A[2], \dots, A[j] \longleftrightarrow A[j+1], \dots, A[n-i-1] \mid A[n-i] \leq \dots \leq A[n-1]$$

5 2 4 6 1 3

Bubble Sort - Algorithm

ALGORITHM *BubbleSort*($A[0..n - 1]$)

//Sorts a given array by bubble sort

//Input: An array $A[0..n - 1]$ of orderable elements

//Output: Array $A[0..n - 1]$ sorted in ascending order

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow 0$ **to** $n - 2 - i$ **do**

if $A[j + 1] < A[j]$ swap $A[j]$ and $A[j + 1]$

SEQUENTIAL SEARCH

Sequential Search – Idea

- Compares successive elements of a given list with a given search key until:
 - A match is encountered (Successful Search)
 - List is exhausted without finding a match (Unsuccessful Search)
- An improvisation to the algorithm is to append the key to the end of the list.
- This means the search has to be successful always and we can eliminate the end of list check.

Linear Search

10

14

19

26

27

31

33

35

42

44

=
33

33

Sequential Search - Algorithm

ALGORITHM *SequentialSearch2*($A[0..n]$, K)

//Implements sequential search with a search key as a sentinel

//Input: An array A of n elements and a search key K

//Output: The index of the first element in $A[0..n - 1]$ whose value is

// equal to K or -1 if no such element is found

$A[n] \leftarrow K$

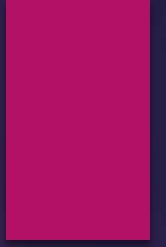
$i \leftarrow 0$

while $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1



BRUTE – FORCE STRING MATCHING

String Matching – Terms

➤ pattern:

- a string of m characters to search for

➤ text:

- a (longer) string of n characters to search in

➤ problem:

- find a substring in the text that matches the pattern

String Matching – Idea

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until:

- all characters are found to match (successful search); or
- a mismatch is detected

Step 3 While pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String Matching - Algorithm

ALGORITHM *BruteForceStringMatch*($T[0..n-1]$, $P[0..m-1]$)

//Implements brute-force string matching

//Input: An array $T[0..n-1]$ of n characters representing a text and

// an array $P[0..m-1]$ of m characters representing a pattern

//Output: The index of the first character in the text that starts a

// matching substring or -1 if the search is unsuccessful

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

$j \leftarrow 0$

while $j < m$ **and** $P[j] = T[i + j]$ **do**

$j \leftarrow j + 1$

if $j = m$ **return** i

return -1

T	H	I	S		I	S		A		S	I	M	P	L	E		E	X	A	M	P	L	E
---	---	---	---	--	---	---	--	---	--	---	---	---	---	---	---	--	---	---	---	---	---	---	---

S	I	M	P	L	E																		
	S	I	M	P	L	E																	
		S	I	M	P	L	E																
			S	I	M	P	L	E															
				S	I	M	P	L	E														
					S	I	M	P	L	E													
						S	I	M	P	L	E												
							S	I	M	P	L	E											
								S	I	M	P	L	E										
									S	I	M	P	L	E									
										S	I	M	P	L	E								

Fundamental data structures

- List
 - ✓ Array
 - ✓ Linked List
 - ✓ String
- Stack
- Queue
- Priority Queue
- Graph
- Tree
- Set and Dictionary

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- *Proving correctness*
- Efficiency
 - ✓ Theoretical analysis
 - ✓ Empirical analysis
- Optimality



FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

Algorithm design strategies

- ▶ Brute force
- ▶ Divide and conquer
- ▶ Decrease and conquer
- ▶ Transform and conquer
- ▶ Greedy Approach
- ▶ Dynamic Programming
- ▶ Backtracking and Branch and Bound
- ▶ Space and time tradeoffs

Analysis of Algorithms

- How good is the algorithm?
 - ✓ Correctness
 - ✓ Time efficiency
 - ✓ Space efficiency
- Does there exist a better algorithm?
 - ✓ Lower bounds
 - ✓ Optimality

ANALYSIS of ALGORITHMS

(What and why)

Aspects of an Algorithm:

- Correctness
- Resource Consumption
- Generality
- Simplicity
-
- Any and all of the above aspects can be analyzed to check if that aspect

is “acceptable” in the given “context”

GENERALITY, SIMPLICITY, ...

- ▶ Difficult to define precisely or quantify!
- ▶ Generality:
 - ▶ Generality of problem solved
 - ▶ Range of inputs accepted
- ▶ Simplicity:
 - ▶ Conceptual
 - ▶ Implementation
 - ▶ Aesthetics!

RESOURCE CONSUMPTION

- Resources Consumed: Time, Space, Disk, Communication Bandwidth,...
- Minimizing resource consumption is intuitively desirable!
- Pragmatic Reasons also!!
- We consider here only **TIME!!!** (well, some times, **space also!**)
- Are these issues relevant in this era of fast computers, cheap memory, broadband etc?

Theoretical Analysis vs Empirical Analysis

- ▶ **Empirical Analysis:**

Implement the algorithm in a specific language, execute it, and **measure performance** parameters of interest (generally, **time** but space and other parameters may also be of interest) aka “**PROFILING**”

- ▶ **Theoretical (a priori) Analysis:**

Analyze performance of the algorithm independent of implementation in a specific programming language / execution platform!

- ▶ **Advantages / Disadvantages ?**

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
- Code the algorithm in an actual programming language and validate it.
- Run the program with the chosen input data
- Analyze the empirical data

Not discussed any more!



Apriori Analysis ANALYSIS FRAMEWORK

Measuring an input's size

Intuition: Most of the algorithms run longer on larger inputs.

Examples:

1. Sorting takes longer time for larger inputs
2. It takes more time to multiply larger matrices

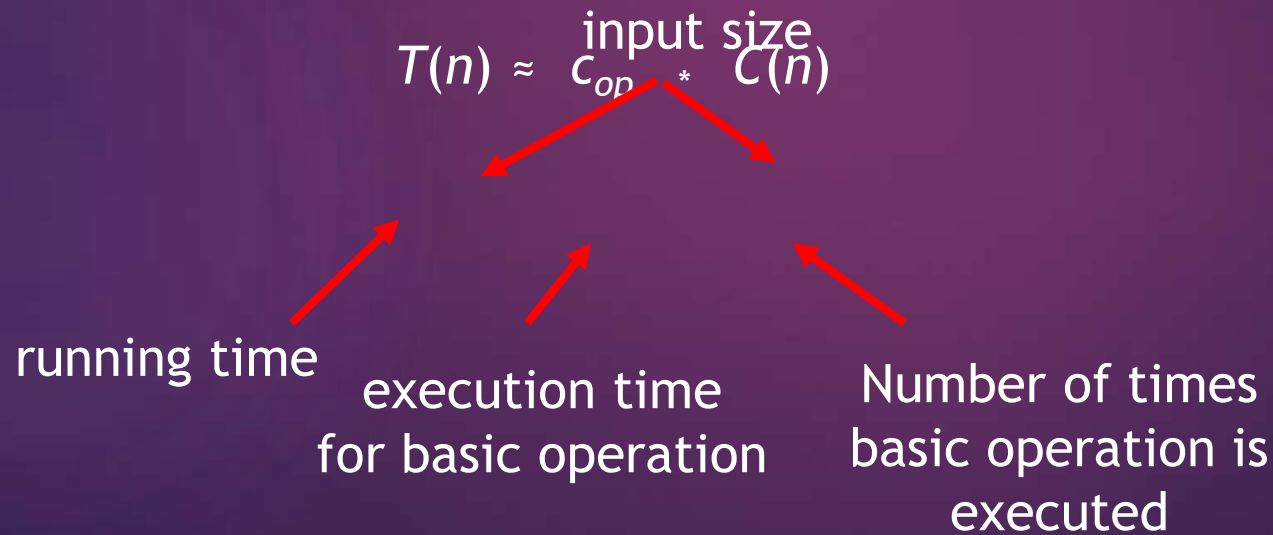
Therefore, logically, we can calculate the efficiency of an algorithm as a function of 'n' which indicates the algorithm's input size.

EXAMPLES FOR Measuring an input's size

1. Sorting
2. Primality Testing
3. Computing the average of n numbers
4. Computing $n/n!$
5. Finding the largest element in a list of n numbers
6. Spell Checking
7. String Matching
8. Evaluating a polynomial
9. Reverse display a list of n numbers
10. Reverse a list of numbers

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

<i>Problem</i>	<i>Input size measure</i>	<i>Basic operation</i>
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 's size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Values of some important functions as $n \rightarrow \infty$

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

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

Best-case, average-case, worst-case

For some algorithms, efficiency depends on form of input:

- Worst case: $C_{\text{worst}}(n)$ - maximum over inputs of size n
- Best case: $C_{\text{best}}(n)$ - minimum over inputs of size n
- Average case: $C_{\text{avg}}(n)$ - “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

$i \leftarrow 0$

while $i < n$ **and** $A[i] \neq K$ **do**

$i \leftarrow i + 1$

if $i < n$ **return** i

else return -1

Worst case	n key comparisons
Best case	1 comparison
Average case	$(n+1)/2$, assuming K is in A

Sequential Search - Average Case (more)

- ▶ Assume the probability of a successful search = p ($0 \leq p \leq 1$)
- ▶ Key match can occur at any position with same probability
- ▶ → when the search is successful:
- ▶ match occurring at any position $i = p / n$
- ▶ and number of comparisons in such a case is i
- ▶ Average Case Efficiency:

$$\begin{aligned} & [1 (p/n) + 2 (p/n) + \dots + i (p/n) + \dots + n (p/n)] + n (1-p) \\ &= (p/n) [1 + 2 + \dots + i + \dots + n] + n (1-p) \\ &= (p/n) n (n+1) / 2 + n (1-p) \\ &= [p (n+1) / 2] + [n (1-p)] \quad ; \text{ check with } p=0 \text{ and } p=1 \end{aligned}$$

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 $n \rightarrow \infty$
- ▶ 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?

Asymptotic order of growth

A way of comparing functions that ignores constant factors and small input sizes (because?)

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

Big-oh

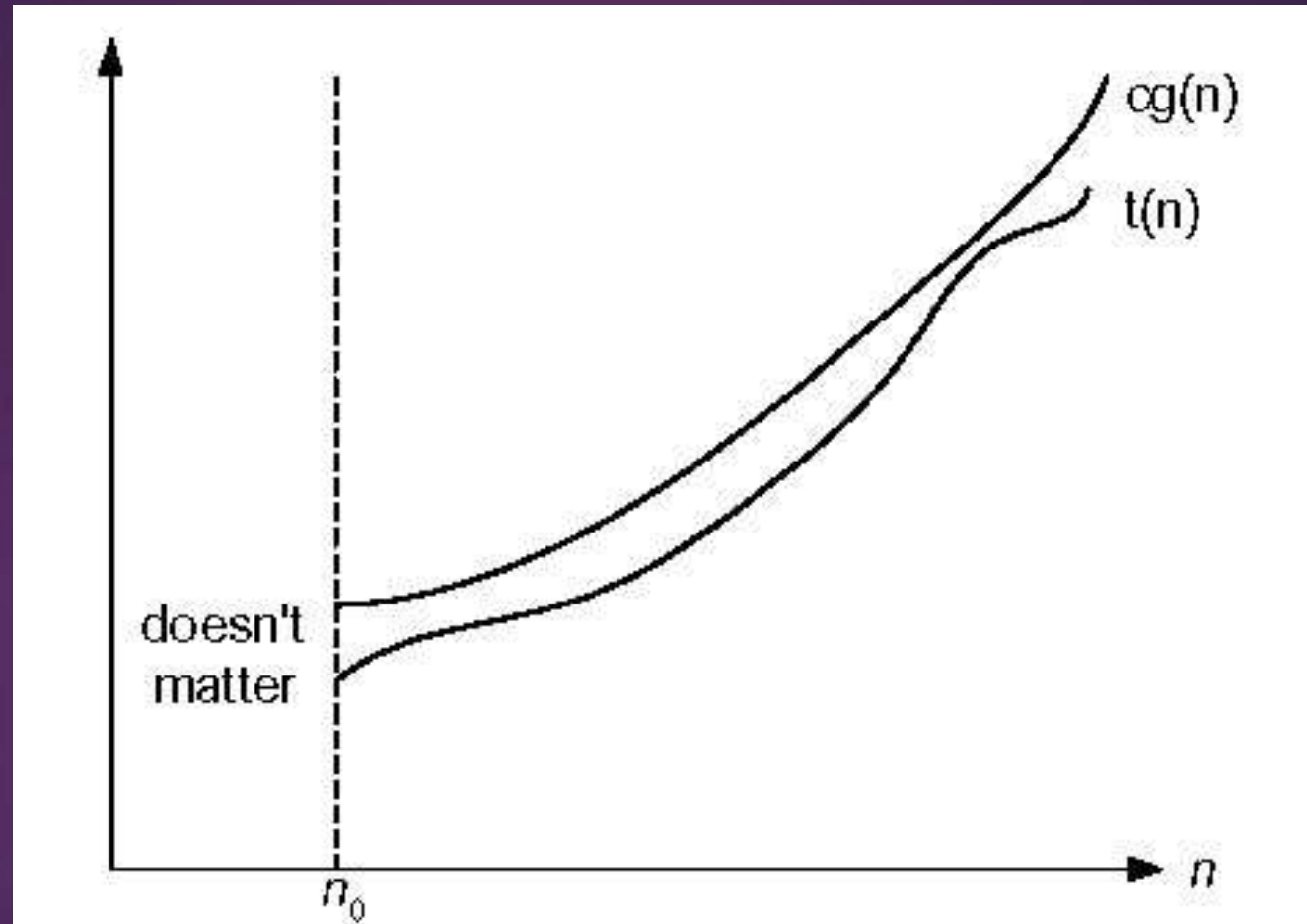
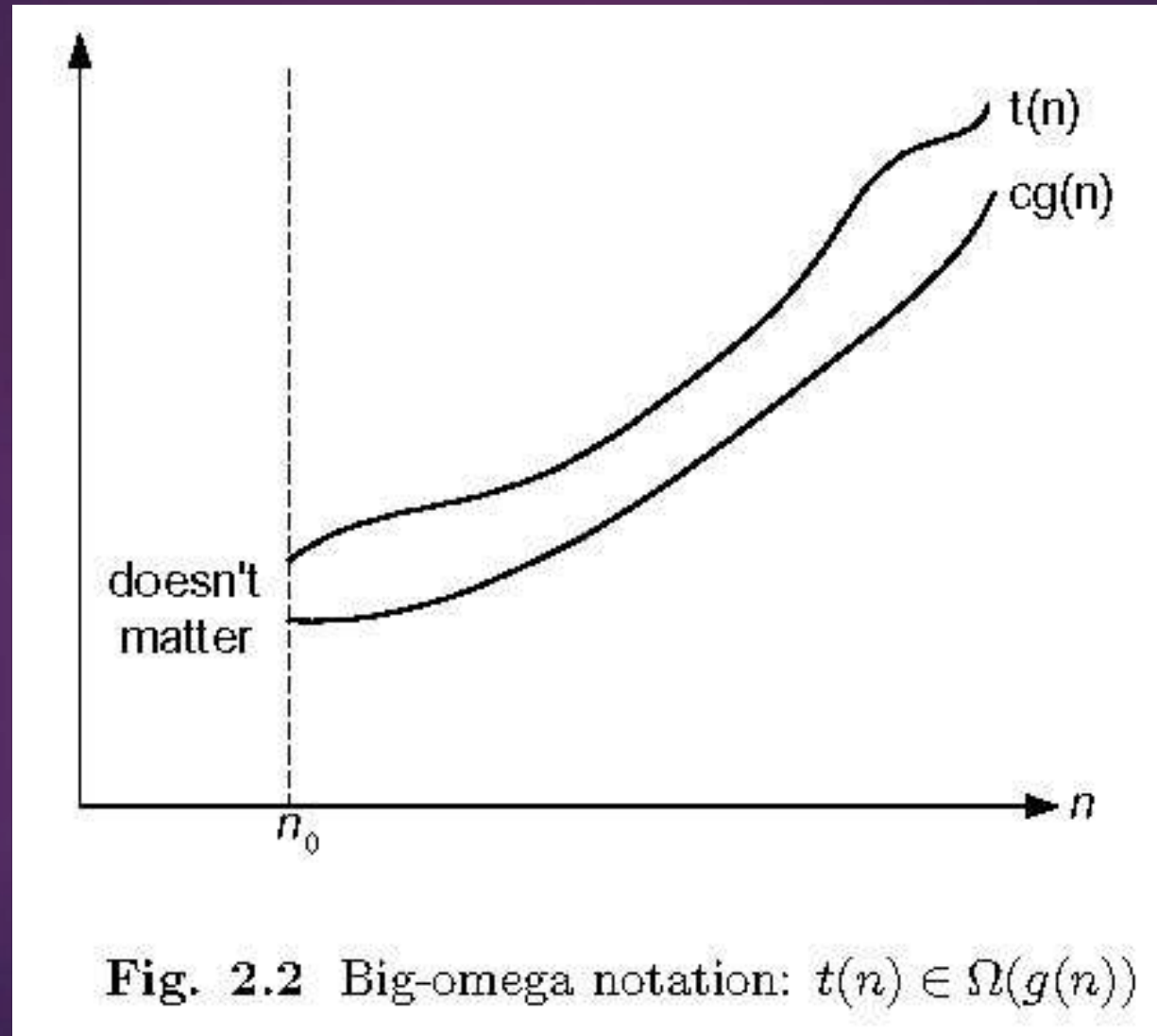


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega



Big-theta

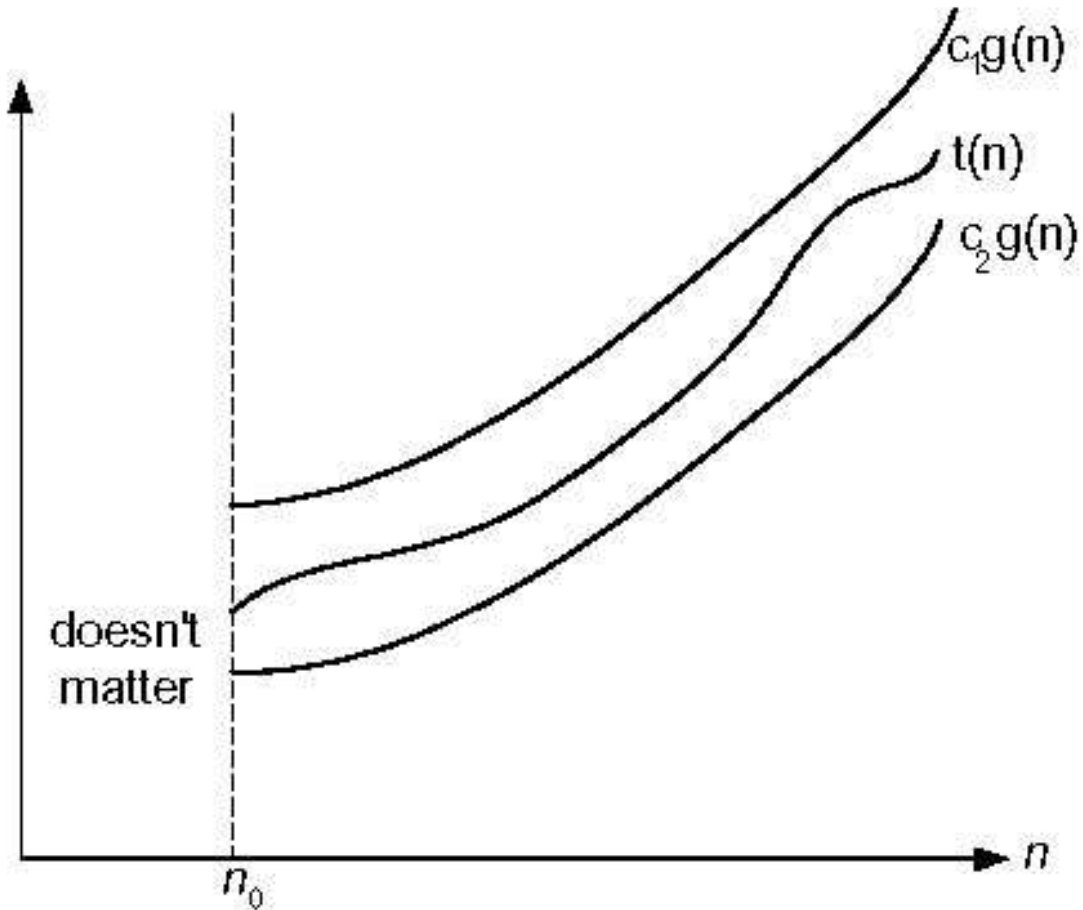


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Establishing order of growth using the definition

O - Notation

Definition:

$f(n)$ is in $O(g(n))$, denoted by $f(n) \in O(g(n))$, if the order of growth of $f(n) \leq$ order of growth of a constant multiple of $g(n)$, i.e., there exists a positive constant c and a non-negative integer n_0 such that

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

Examples:

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

Ω -notation

Definition:

A function $t(n)$ is said to be in $\Omega(g(n))$, denoted $t(n) \in \Omega(g(n))$, 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

$$t(n) \geq cg(n) \text{ for all } n \geq n_0$$

Examples:

- ▶ $10n^2 \in \Omega(n^2)$
- ▶ $0.3n^2 - 2n \in \Omega(n^2)$
- ▶ $0.1n^3 \in \Omega(n^2)$

Θ -notation

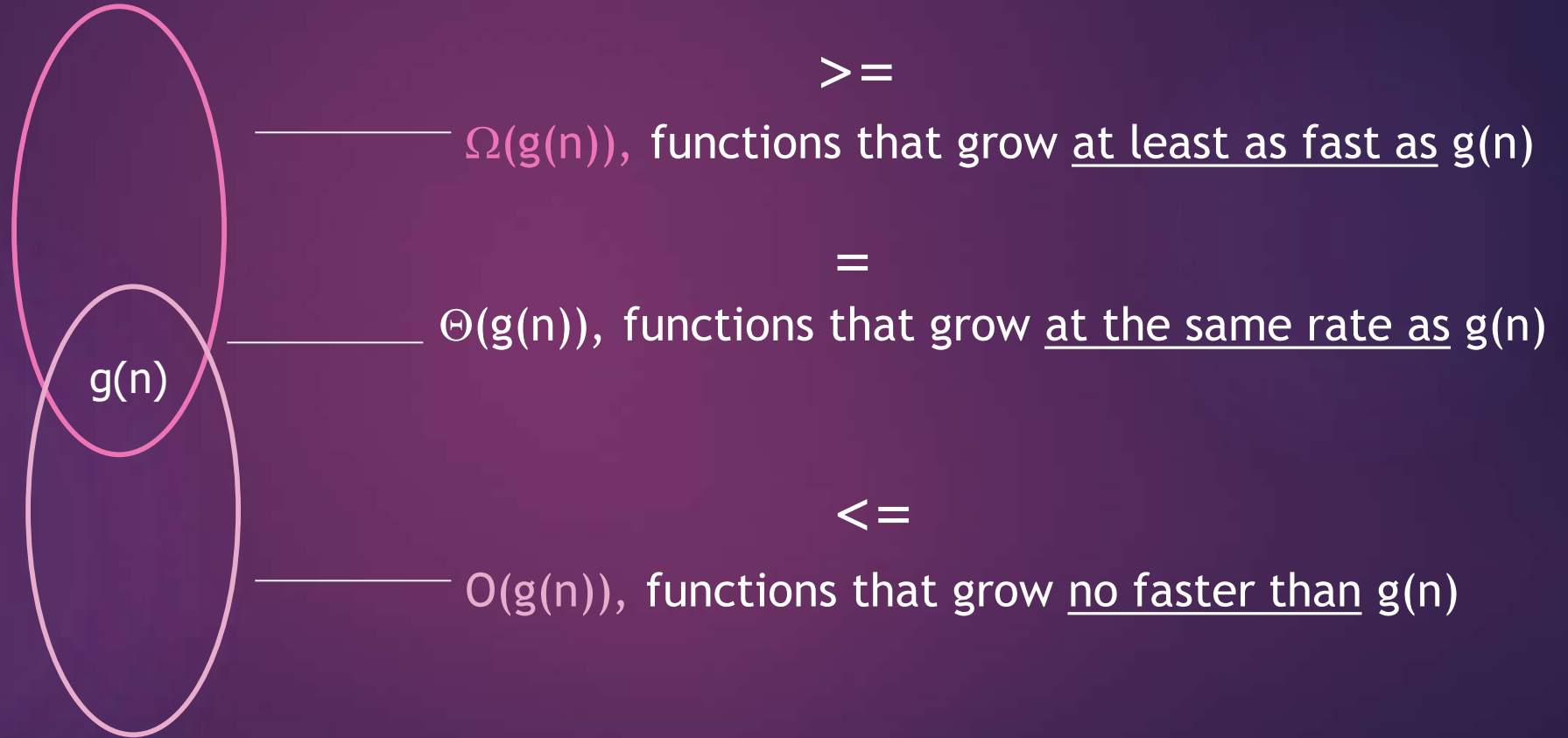
Definition

A function $t(n)$ is said to be in $\Theta(g(n))$, denoted $t(n) \in \Theta(g(n))$, 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

$$c_2 g(n) \leq t(n) \leq c_1 g(n) \text{ for all } n \geq n_0$$

Examples:

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



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 -notation and Θ -notation.

Implication: The algorithm's overall efficiency will be determined by the part with a larger order of growth, i.e., its least efficient part.

► For example, $5n^2 + 3n \log n \in O(n^2)$

Theorem

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

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

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

Define $c_3 = c_1 + c_2$ and $n_3 = \max\{n_1, n_2\}$.

Then:

$$t_1(n) + t_2(n) \leq c_3 * \max\{g_1(n), g_2(n)\}, \quad \text{for all } n \geq n_3$$

Some properties of asymptotic order of growth

- ▶ $f(n) \in O(f(n))$
- ▶ $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- ▶ If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

Note similarity with $a \leq b$

- ▶ If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then
$$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

Also, $\sum_{1 \leq i \leq n} \Theta(f(i)) = \Theta(\sum_{1 \leq i \leq n} f(i))$

Establishing order of growth using limits

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

Examples:

$$\begin{array}{ccc} 10n & \text{vs.} & n^2 \\ n(n+1)/2 & \text{vs.} & n^2 \end{array}$$

L'Hôpital's rule and Stirling's formula

L'Hôpital's rule: If $\lim_{n \rightarrow \infty} f(n) = \lim_{n \rightarrow \infty} g(n) = \infty$ and the derivatives f' , g' exist, then

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \lim_{n \rightarrow \infty} \frac{f'(n)}{g'(n)}$$

Example: $\log n$ vs. n

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

Example: 2^n vs. $n!$

Orders of growth of some important functions

- ▶ All logarithmic functions $\log_a n$ belong to the same class

$\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is

- ▶ All polynomials of the same degree k belong to the same class:

$$a_k n^k + a_{k-1} n^{k-1} + \dots + a_0 \in \Theta(n^k)$$

- ▶ Exponential functions a^n have different orders of growth for different a 's

- ▶ order $\log n < \text{order } n^\alpha \ (\alpha > 0) < \text{order } a^n < \text{order } n! < \text{order } n^n$

Basic asymptotic efficiency classes

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

Time efficiency of non - recursive 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

$$\sum_{l \leq i \leq u} 1 = 1 + 1 + \dots + 1 = u - l + 1$$

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

$$\sum_{1 \leq i \leq n} i = 1 + 2 + \dots + n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$$

$$\sum_{1 \leq i \leq n} i^2 = 1^2 + 2^2 + \dots + n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$$

$$\sum_{0 \leq i \leq n} a^i = 1 + a + \dots + a^n = (a^{n+1} - 1)/(a - 1) \text{ for any } a \neq 1$$

In particular, $\sum_{0 \leq i \leq n} 2^i = 2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$$\sum (a_i \pm b_i) = \sum a_i \pm \sum b_i \quad \sum c a_i = c \sum a_i \quad \sum_{l \leq i \leq u} a_i = \sum_{l \leq i \leq m} a_i + \sum_{m+1 \leq i \leq u} a_i$$

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 $maxval$

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

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: Gaussian elimination

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

//Implements Gaussian elimination of an n -by- $(n+1)$ matrix A

for $i \leftarrow 0$ to $n - 2$ do

 for $j \leftarrow i + 1$ to $n - 1$ do

 for $k \leftarrow i$ to n do

$$A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$$

Find the efficiency class and a constant factor improvement.

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

$count \leftarrow 1$

while $n > 1$ **do**

$count \leftarrow count + 1$

$n \leftarrow \lfloor n/2 \rfloor$

return $count$

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: $n! = 1 * 2 * \dots * (n-1) * n$ 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 $F(n - 1) * n$

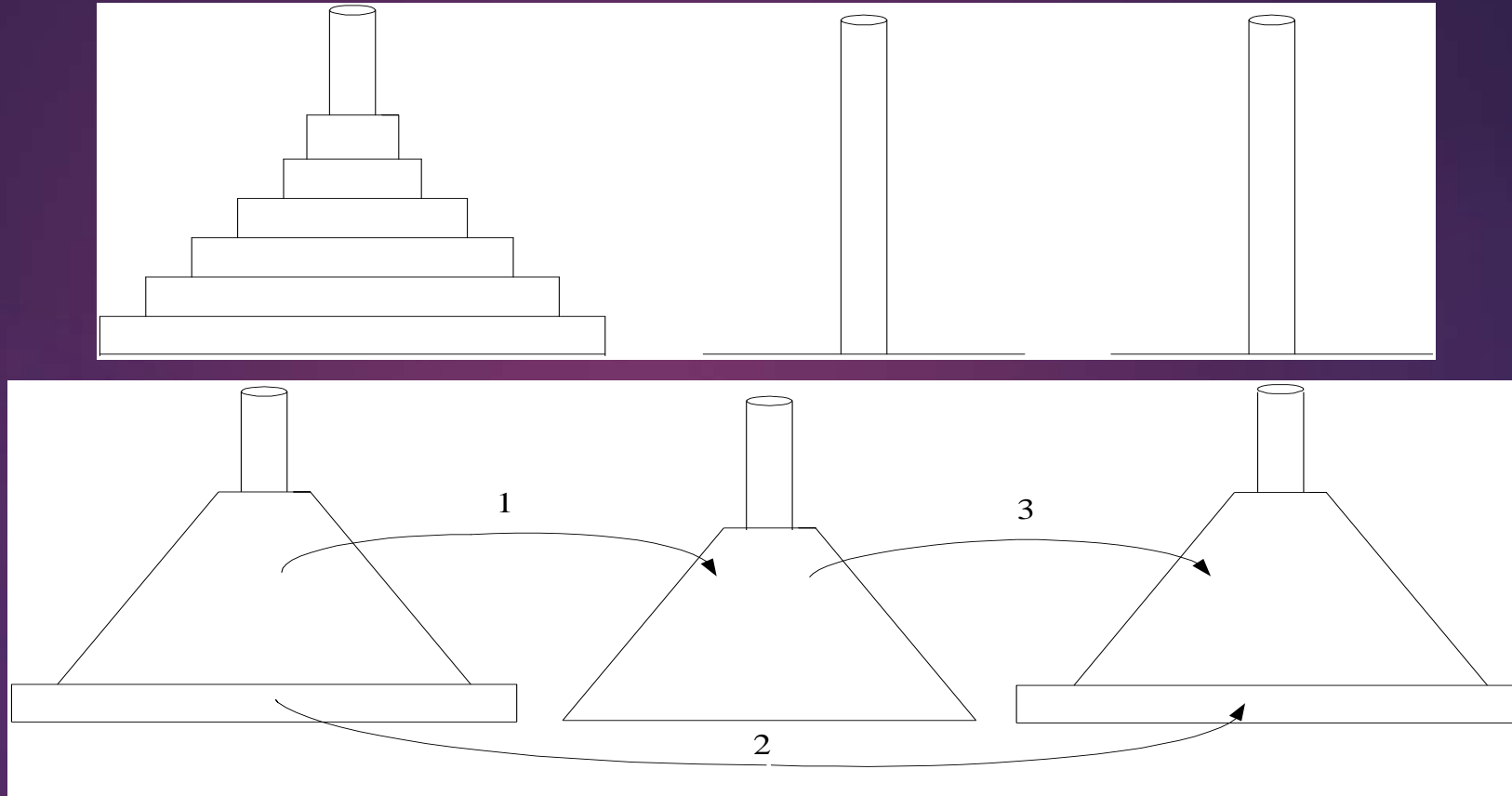
Solving the recurrence for $M(n)$

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

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

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

Example 2: The Tower of Hanoi Puzzle



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

Solving recurrence

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

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

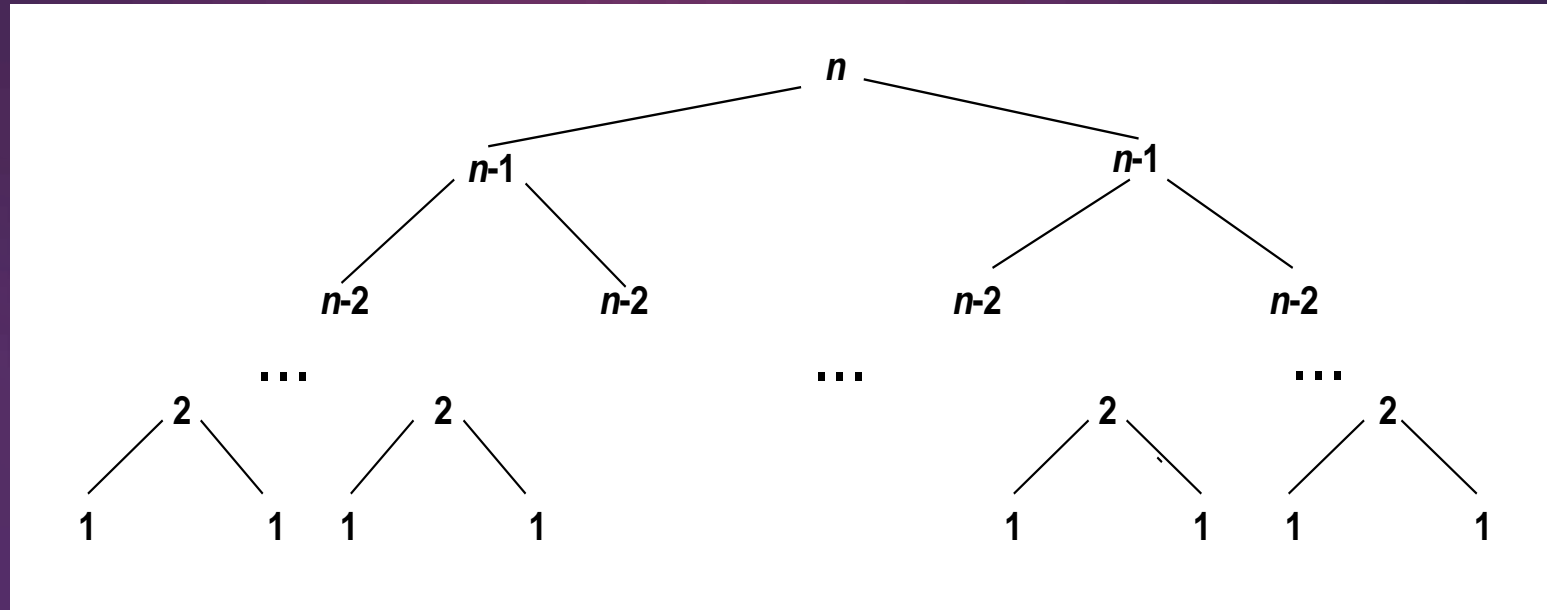
The pattern of the first three sums on the left suggests that the next one will be $2^4M(n-4) + 2^3 + 2^2 + 2 + 1$ and, generally, after i substitutions, we get

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

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 (2.3):

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

uzzle



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

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

$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

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

Thus, we end up with

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

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

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

Fibonacci numbers

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

The Fibonacci recurrence:

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

$$F(0) = 0$$

$$F(1) = 1$$

General 2nd order linear homogeneous recurrence with constant coefficients:

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

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

$$F(n) = F(n-1) + F(n-2) \quad \text{or} \quad F(n) - F(n-1) - F(n-2) = 0$$

$$F(n) - F(n-1) - F(n-2) = 0. \quad (2.10)$$

Its characteristic equation is

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

with the roots

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

Since this characteristic equation has two distinct real roots, we have to use the formula indicated in Case 1 of Theorem 1:

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

Since the characteristic equation has two distinct real roots:

$$F(0) = \alpha \left(\frac{1+\sqrt{5}}{2} \right)^0 + \beta \left(\frac{1-\sqrt{5}}{2} \right)^0 = 0$$

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

$$\begin{array}{rclcl} & \alpha & + & \beta & = & 0 \\ \left(\frac{1+\sqrt{5}}{2} \right) \alpha & + & \left(\frac{1-\sqrt{5}}{2} \right) \beta & = & 1. \end{array}$$

$$F(n) = \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n),$$

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

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.