Unit I – Basic Concepts of Algorithms

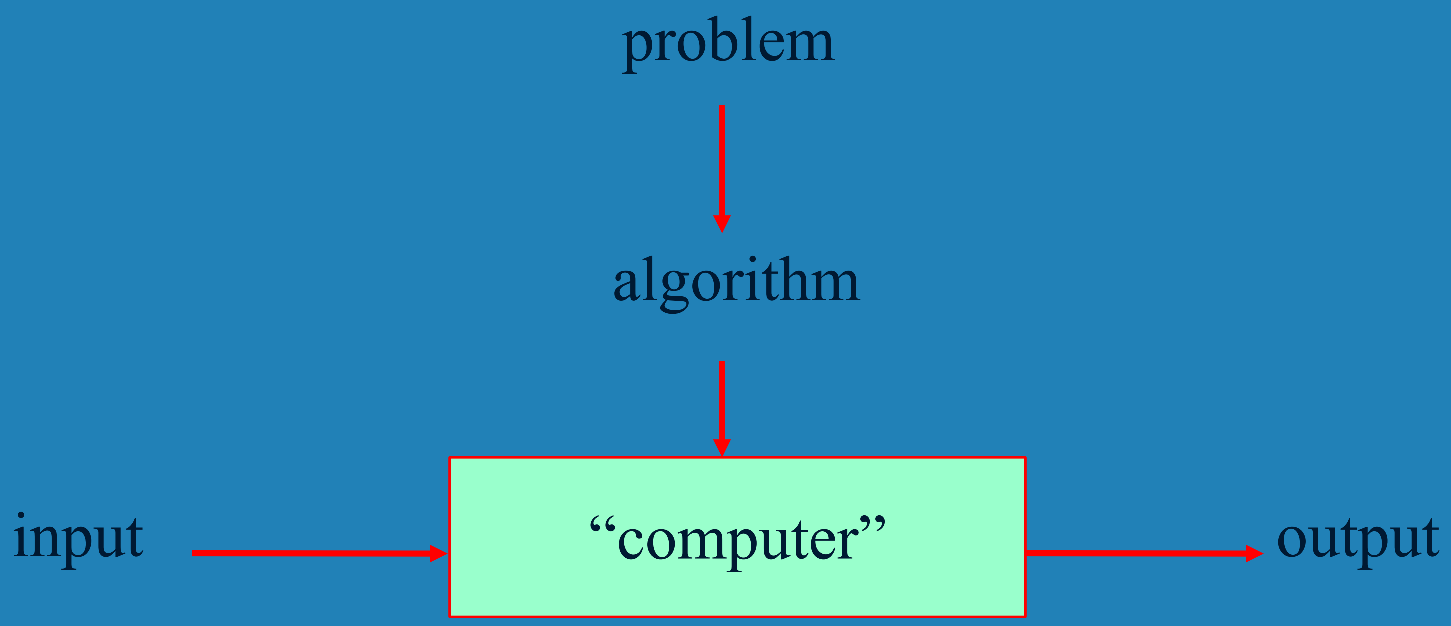
Introduction

Algorithm

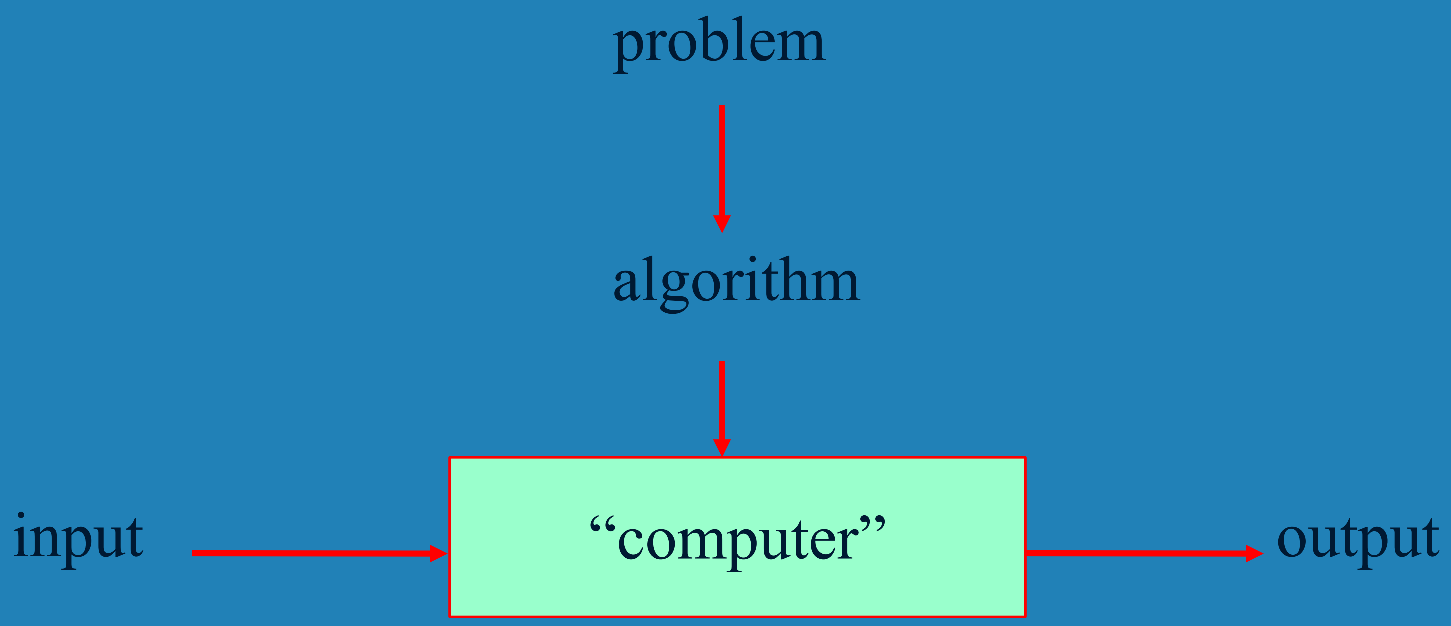
- **Abu Jafar Muhammad Ibn Musu Al-Khowarizmi [Born: about 780 in Baghdad (now in Iraq). Died: about 850]**
- **An algorithm is a set of rules for carrying out calculation either by hand or on a machine.**
- **An algorithm is a finite step-by-step procedure to achieve a required result.**
- **An algorithm is a sequence of computational steps that transform the input into the output.**
- **An algorithm is a sequence of operations performed on data that have to be organized in data structures.**
- **An algorithm is an abstraction of a program to be executed on a physical machine (model of Computation).**

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.



Notion of algorithm



Algorithmic solution

Algorithms

- **It is not depended on programming language, machine.**
- **Are mathematical entities, which can be thought of as running on some sort of *idealized computer* with an infinite random access memory**
- **Algorithm design is all about the mathematical theory behind the design of good programs.**

Contd…

- Algorithmic is a branch of computer science that consists of designing and analyzing computer algorithms
- The "design" pertain to
  - The description of algorithm at an abstract level by means of a pseudo language, and
  - Proof of correctness that is, the algorithm solves the given problem in all cases.
- The "analysis" deals with performance evaluation (complexity analysis).
- Random Access Machine (*RAM*) model

Why study algorithm design?

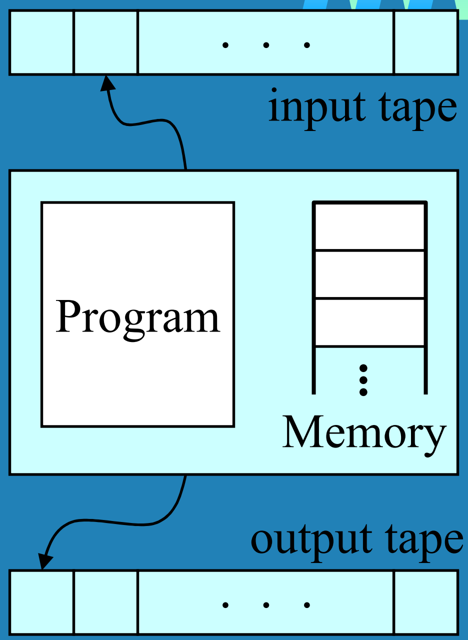
- Programming is a very complex task, and there are a number of aspects of programming hat make it so complex. The first is that most programming projects are very large, requiring the coordinated efforts of many people. (This is the topic a course like software engineering.)
- The next is that many programming projects involve storing and accessing large quantities of data efficiently. (This is the topic of courses on data structures and databases.)
- The last is that many programming projects involve solving complex computational problems, for which simplistic or naive solutions may not be efficient enough. The complex problems may involve numerical data (the subject of courses on numerical analysis), but often they involve discrete data. This is where the topic of algorithm design and analysis is important.
- The focus of this course is on how to design good algorithms, and how to analyze their efficiency. This is among the most basic aspects of good programming.

Contd…

- Algorithms help us to understand **scalability**.
- •Performance often draws the line between what is feasible and what is impossible.
- •Algorithmic mathematics provides a **language**for talking about program behavior.
- •Performance is the **currency**of computing.
- •The lessons of program performance generalize to other computing resources.
- •Speed is fun!

Random Access Machine

- A *Random Access Machine* (RAM) consists of:
  - a fixed *program*
  - an unbounded *memory*
  - a read-only *input tape*
  - a write-only *output tape*
- Each *memory register* can hold an arbitrary integer (*)
- Each *tape cell* can hold a single symbol from a finite *alphabet* Σ



- Instruction set:
  - $x \leftarrow y$, $x \leftarrow y$ {+, –, *, div, mod} z
  - goto *label*
  - if y {<, ≤, =, ≥ ,> , ≠} z goto *label*
  - $x \leftarrow$ input, output $\leftarrow y$
  - halt

- **Addressing modes:**
  - x may be direct or indirect reference
  - y and z may be constants, direct or indirect references

Contd…

- **Why analyze algorithms?**
  - **evaluate algorithm performance**
  - **compare different algorithms**
- **Analyze what about them?**
  - **running time, memory usage, solution quality**
  - **worst-case and “typical” case**
- **Computational complexity**
  - **understanding the intrinsic difficulty of computational problems - classifying problems according to difficulty**
  - **algorithms provide upper bound**
  - **to show problem is hard, must show that any algorithm to solve it requires at least a given amount of resources**
  - **transform problems to establish “equivalent” difficulty**

Example of computational problem: sorting

- **Statement of problem:**
  - ***Input:* A sequence of *n* numbers** $<a_1, a_2, \ldots, a_n>$
  - ***Output:* A reordering of the input sequence** $<a'_1, a'_2, \ldots, a'_n>$ **so that** $a'_i \le a'_j$ **whenever** $i < j$
- **Instance: The sequence <5, 3, 2, 8, 3>**
- **Algorithms:**
  - **Selection sort**
  - **Insertion sort**
  - **Merge sort**
  - **(many others)**

Selection Sort

- **Input: array a[1],…,a[n]**

- **Output: array a sorted in non-decreasing order**

- **Algorithm:**

```
for i=1 to n
    swap a[i] with smallest of a[i],…a[n]
```

• see also pseudocode, section 3.1

Insertion Sort

"pseudocode"

INSERTION-SORT (A, n) ▷ $A[1 \,.\,.\, n]$

for $j \leftarrow 2$ **to** n

  do $key \leftarrow A[j]$

    $i \leftarrow j - 1$

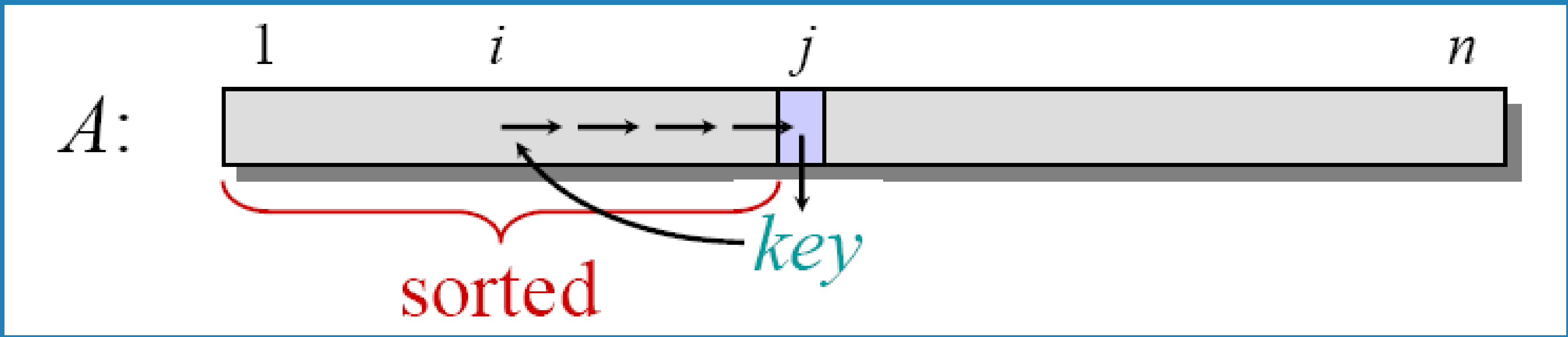
    while $i > 0$ and $A[i] > key$

      do $A[i+1] \leftarrow A[i]$

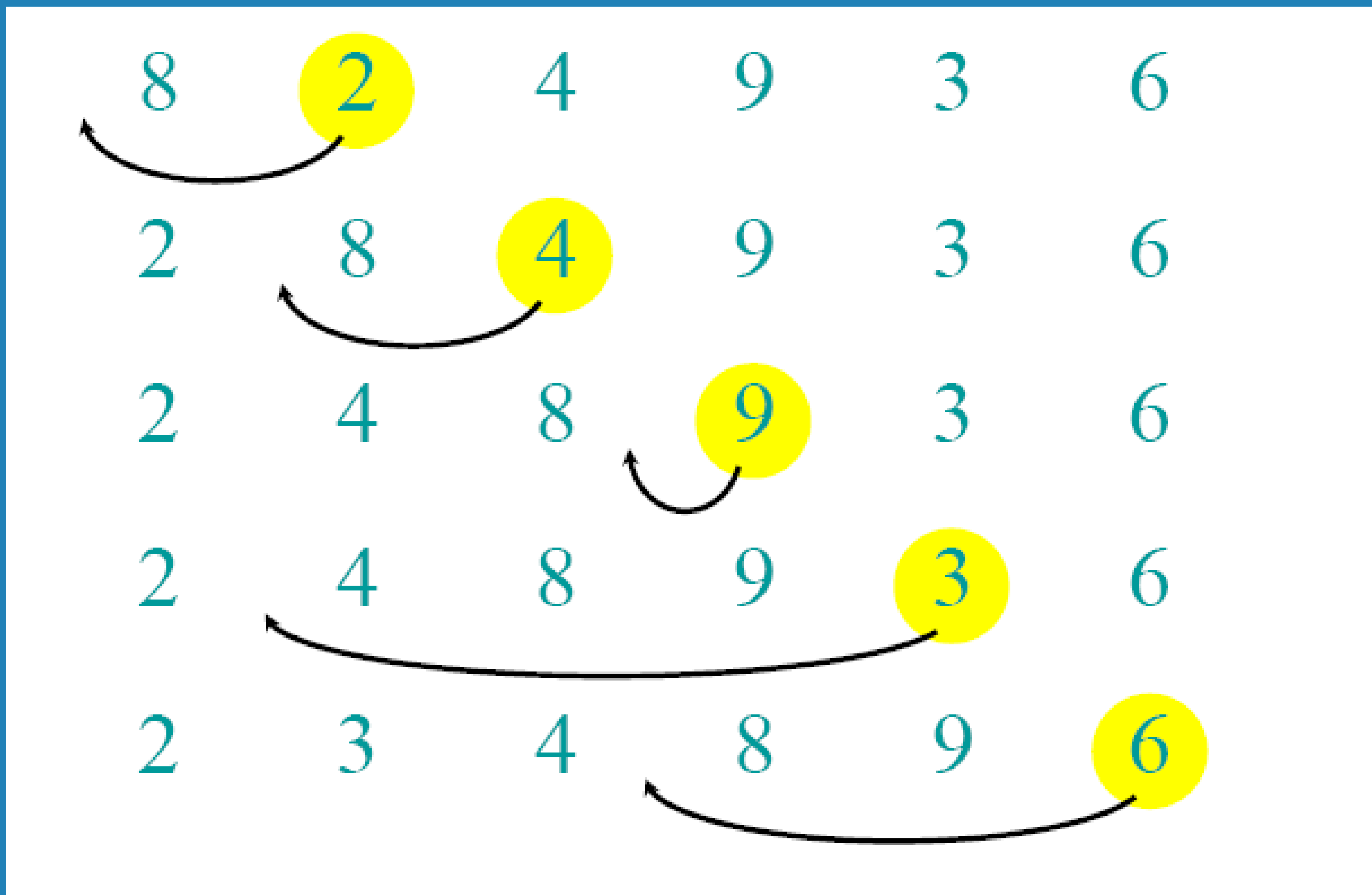
        $i \leftarrow i - 1$

    $A[i+1] = key$

Contd..



Contd…



Contd…

- **Worst-case:** (usually)
- •T(n) =maximum time of algorithm on any input of size n.
- **Average-case:** (sometimes)
- •T(n) =expected time of algorithm over all inputs of size n.
- •Need assumption of statistical distribution of inputs.
- **Best-case:** (bogus)
- •Cheat with a slow algorithm that works fast on someinput

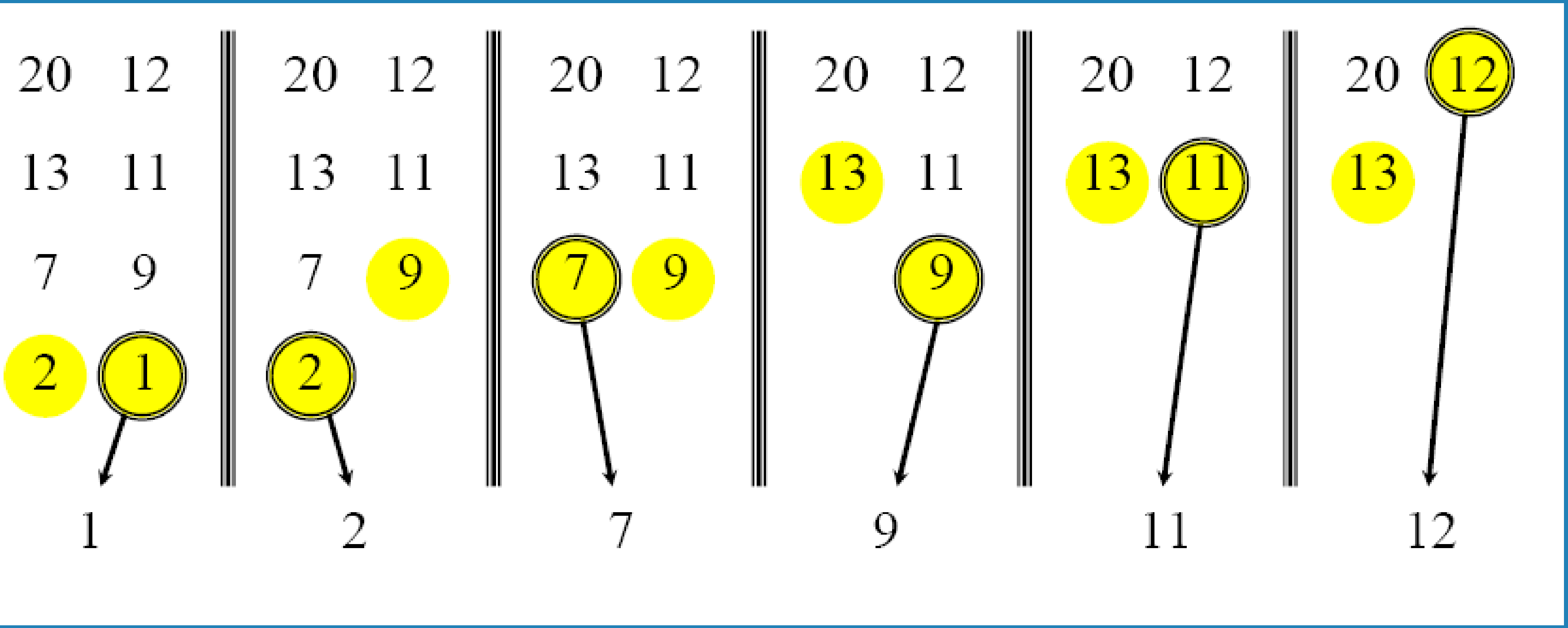
Contd…

MERGE-SORT $A[1 \,.\,.\, n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1 \,.\,.\, \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \,.\,.\, n\,]$.
3. "***Merge***" the 2 sorted lists.

Key subroutine: **MERGE**

Contd…

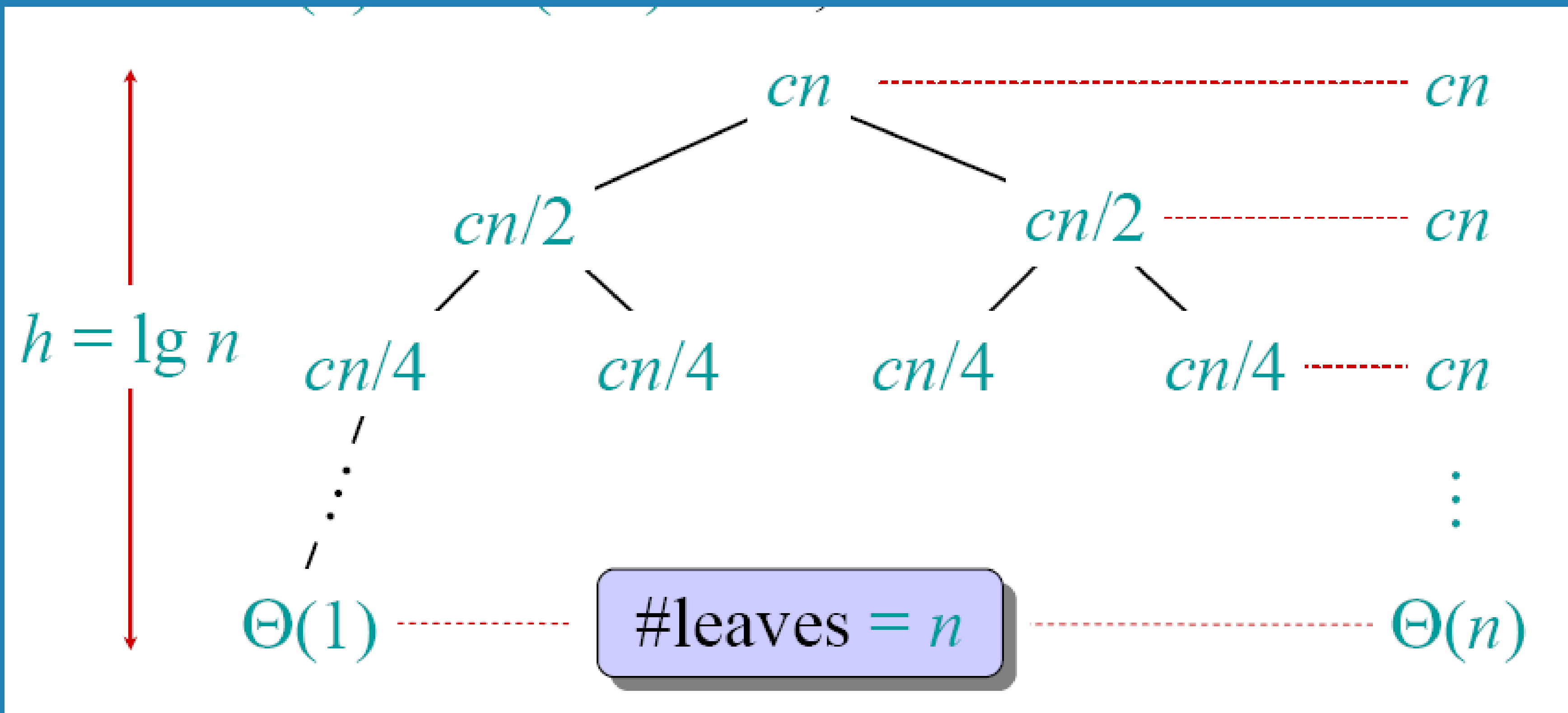


20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
20 12
13 11
9
9
20 12
13 11
11
20 12
13
12

Contd…

	$T(n)$	**MERGE-SORT** $A[1 \, . \, . \, n]$
	$\Theta(1)$	1. If $n = 1$, done.
Abuse	$2T(n/2)$	2. Recursively sort $A[\,1 \, . \, . \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \, . \, . \, n\,]$.
	$\Theta(n)$	3. ***"Merge"*** the 2 sorted lists

Contd…



Some Well-known Computational Problems

- **Sorting**
- **Searching**
- **Shortest paths in a graph**
- **Minimum spanning tree**
- **Primality testing**
- **Traveling salesman problem**
- **Knapsack problem**
- **Chess**
- **Towers of Hanoi**
- **Program termination**

What is an algorithm?

- **Recipe, process, method, technique, procedure, routine,… with following requirements:**

1. **Finiteness**
   - **terminates after a finite number of steps**
2. **Definiteness**
   - **rigorously and unambiguously specified**
3. **Input**
   - **valid inputs are clearly specified**
4. **Output**
   - **can be proved to produce the correct output given a valid input**
5. **Effectiveness**
   - **steps are sufficiently simple and basic**

Euclid's Algorithm

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

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

gcd(*m*,*n*) = gcd(*n*, *m* mod *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 fo the remainder to r**

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

while $n \neq 0$ **do**

$\quad r \leftarrow m \bmod n$

$\quad m \leftarrow n$

$\quad n \leftarrow r$

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 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 gcd(*m*,*n*)

Is this an algorithm?

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

$j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Why study algorithms?

- **Theoretical importance**
  - **the core of computer science**
- **Practical importance**
  - **A practitioner's toolkit of known algorithms**
  - **Framework for designing and analyzing algorithms for new problems**

Basic Issues Related to Algorithms

- **How to design algorithms**
- **How to analyze algorithms**

Algorithm Problem solving

- Understand the problem
- Decide on:
  - Computational Means
  - Exact vs Approximate solving
  - Data structures
  - Algorithm design techniques
- Design an algorithm
- Prove correctness
- Analyse an algorithm
- Code an algorithm

Understand the problem

- First important step
- What must be done rather than how to do it
- Input-*instance* of the problem
- Ascertaining the capability of computational device
- Generic one-processor, random-access-machine (RAM)- instructions are executed one by one.
- computer resources- memory, BW, CPU measures efficiency of algorithm.

Choosing between exact and approximate problem solving

- **An algorithm that can solve the problem exactly is called an exact algorithm. The algorithm that can solve the problem approximately is called an approximation algorithm.**
- **The problems that can be approximately are**
  - **extracting square roots**
  - **solving non-linear equations**
  - **evaluate definite integrals**
  - **for some, algorithm for solving a problem exactly is not acceptable because it can be slow due to its intrinsic complexity of that problem. For ex, like traveling salesman problem which finds shortest tour through n cities.**

Deciding on appropriate data structures

- **Algorithm + Data Structures = Program**

Algorithmic design technique

- An algorithm design technique is a general approach to solving problems mathematically that is applicable to a variety of problems from different areas of computing.
- Algorithm design technique provides guidance for designing algorithms for new problems or problems for which there is no satisfactory algorithm. It makes it possible to classify algorithm according to underlying design idea.

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Analysis of algorithms

- **How good is the algorithm?**
  - **time efficiency**
  - **space efficiency**

- **Does there exist a better algorithm?**
  - **lower bounds**
  - **optimality**

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary

Analysis of algorithms

- **Issues:**
  - **correctness**
  - **time efficiency**
  - **space efficiency**
  - **optimality**

- **Approaches:**
  - **theoretical analysis**
  - **empirical analysis**

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**

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

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

$i \leftarrow 0$
while $i < n$ **and** $A[i] \neq K$ **do**
　　$i \leftarrow i + 1$
if $i < n$ **return** i
else return -1

- Worst case

- Best 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

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 $g(n)$ (upper bound)**
- **Θ($g(n)$): class of functions $f(n)$ that grow <u>at same rate</u> as $g(n)$ (Same bound)**
- **Ω($g(n)$): class of functions $f(n)$ that grow <u>at least as fast</u> as $g(n)$ (Lower bound)**

Big-oh

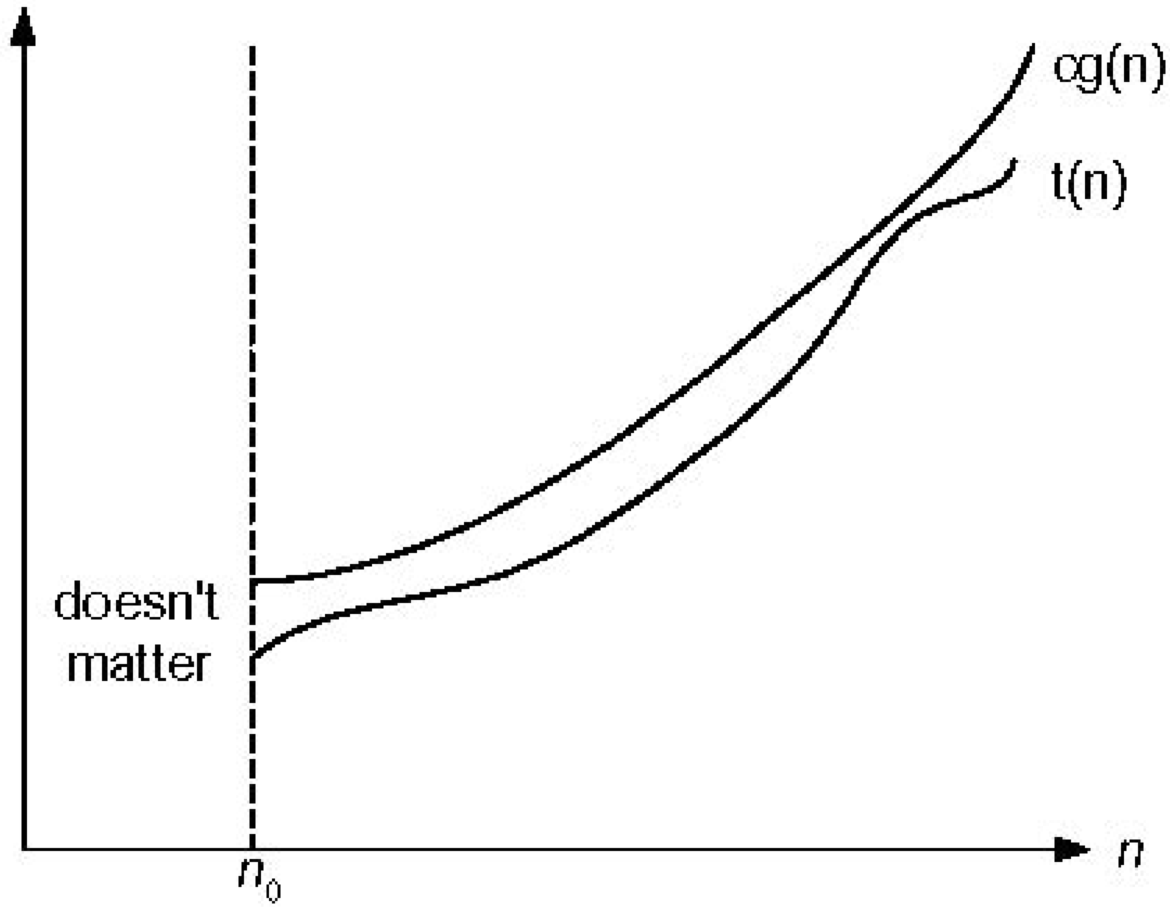


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-omega

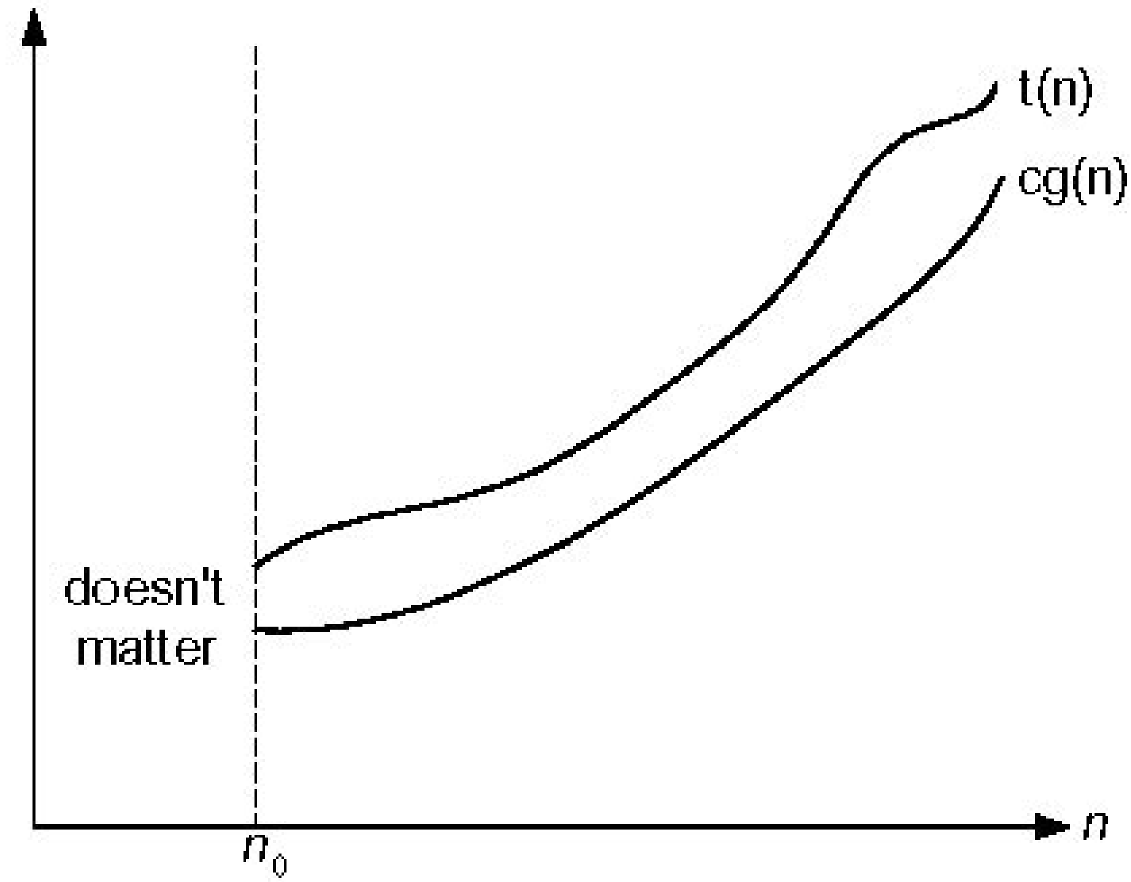


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-theta

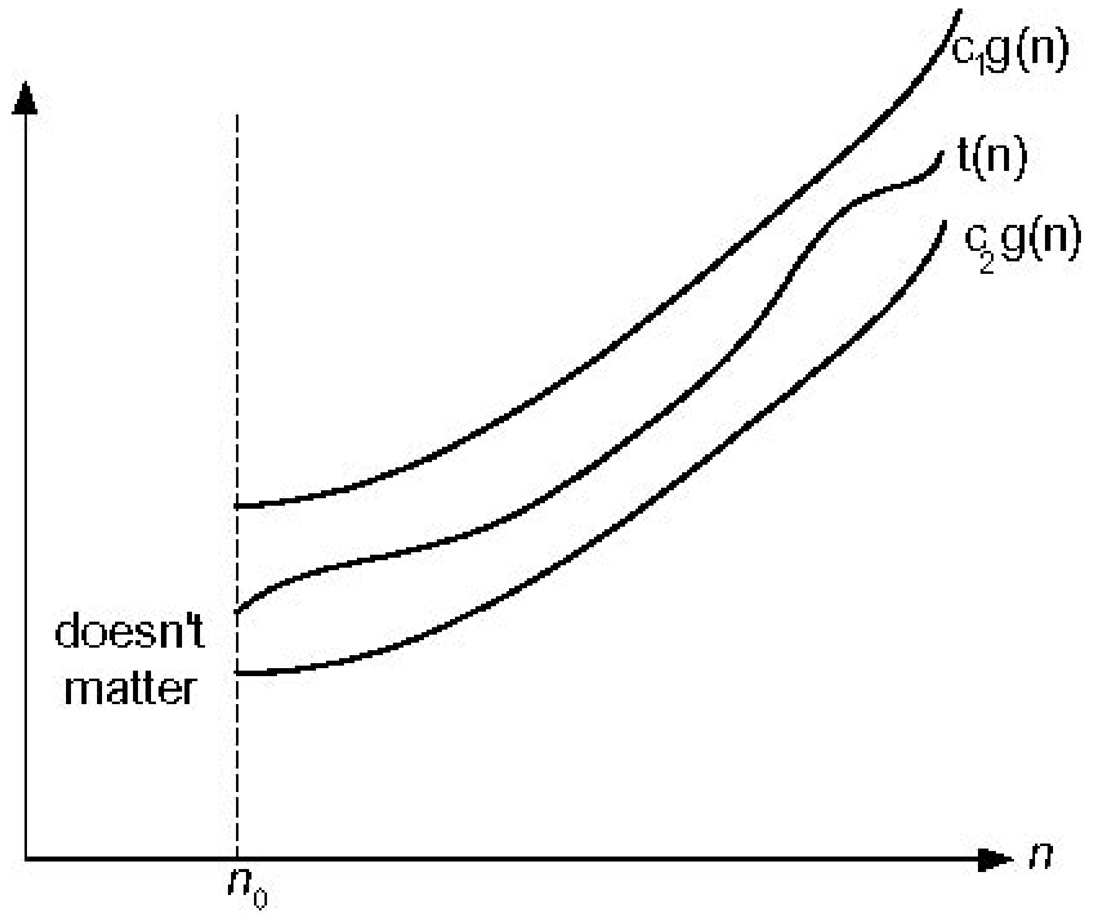


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

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** $O(n)$

growth

- $f(n) \in O(f(n))$

- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(\text{n}))$

- 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

  $$f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$$

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

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

- **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**

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

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

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

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 $a \ne 1$

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

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

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

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.

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 $count$

It cannot be investigated the way the

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

ALGORITHM $F(n)$

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

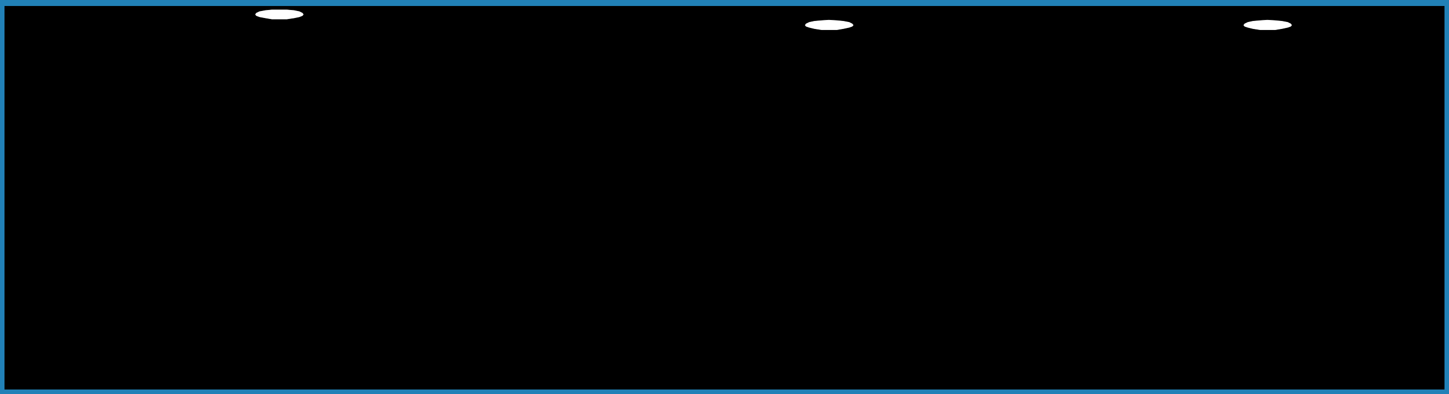
$n!$: $F(n) = F(n-1) * n$

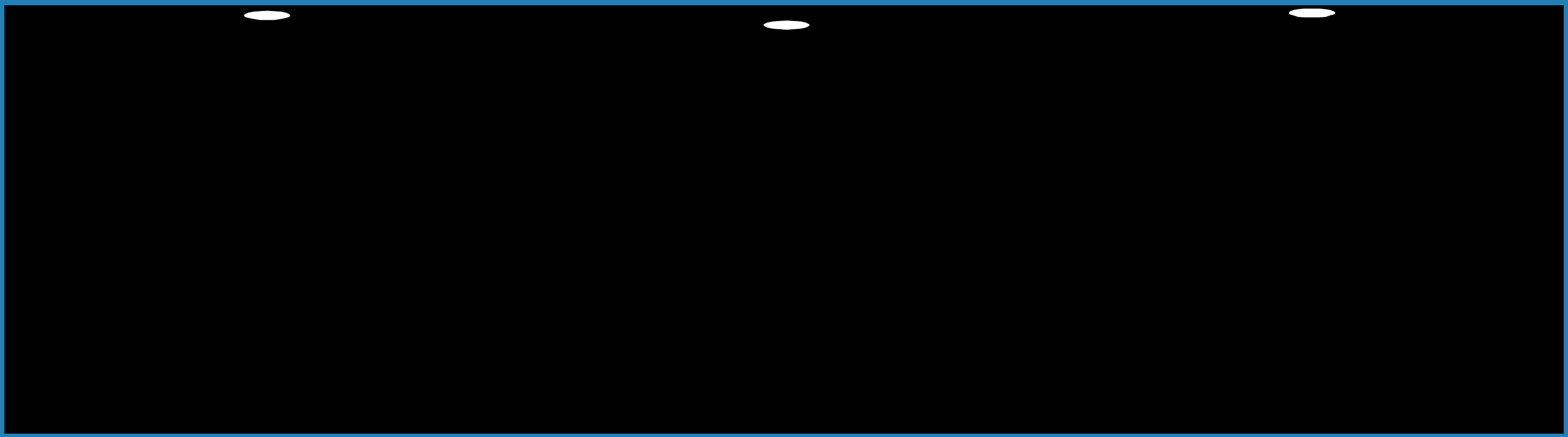
$F(0) = 1$

Solving the recurrence for M(n)

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

Puzzle





Recurrence for number of moves:

moves

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

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

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:

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

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

numbers

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

Characteristic equation:

Roots of the characteristic equation:

General solution to the recurrence:

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

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.