Design and Analysis of Algorithms

Lecture 1 Introductions

Algorithm definition

- An **algorithm** a computational procedure that takes an input and produces an output in order to solve a well-specified computational problem.
  - 1. The instructions must be exact enough to be implemented mechanically;
  - 2. The number of instructions must be finite.
- Input, output,
- Computational problem, instance of a problem
- Correct, efficient, precise, mechanical,

算法 ALGORITHM

是在有限步骤内求解某一问题所使用的一组定义明确的规则。

一个算法应该具有以下五个重要的特征：

1. **有穷性：** 一个算法必须保证执行有限步之后结束；
2. **确切性：** 算法的每一步骤必须有确切的定义；
3. **输　入：** 一个算法有0个或多个输入，以刻画运算对象的初始情况，所谓0个输入是指算法本身限定了初始条件；
4. **输　出：** 一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的；
5. **可行性：** 算法原则上能够精确地运行，而且人们用笔和纸做有限次运算后即可完成。

Example: Sorting problem

The sorting problem

- **Input**: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- **Output**: a permutation $\langle a_1', a_2', \ldots, a_n' \rangle$, s.t. $a_1' \leq a_2' \leq \ldots \leq a_n'$

- An **instance** of a problem:
  - the input needed to compute a solution (e.g.: $\langle 5, 3, 6, 2 \rangle$)

- An algorithm is **correct** if it ends with the correct output in a finite amount of time, on any legitimate input

Origin: al-Khwārizmī
(780-850)

- A Persian mathematician, astronomer
  and geographer;
- A scholar in the House of Wisdom in Baghdad;
- His book "Arithmetic" on the technique of performing arithmetic with Hindu-Arabic numerals was translated and introduced to the West in the twelfth century;
- Considered the founder of algebra, he presented the first systematic solution of linear and quadratic equations in his book "Algebra";
- The term "algorithm" was derived from the Latinized forms of al-Khwarizmi's name.
- Algorismi (Algoritmi) → Algorism (Algorithm)

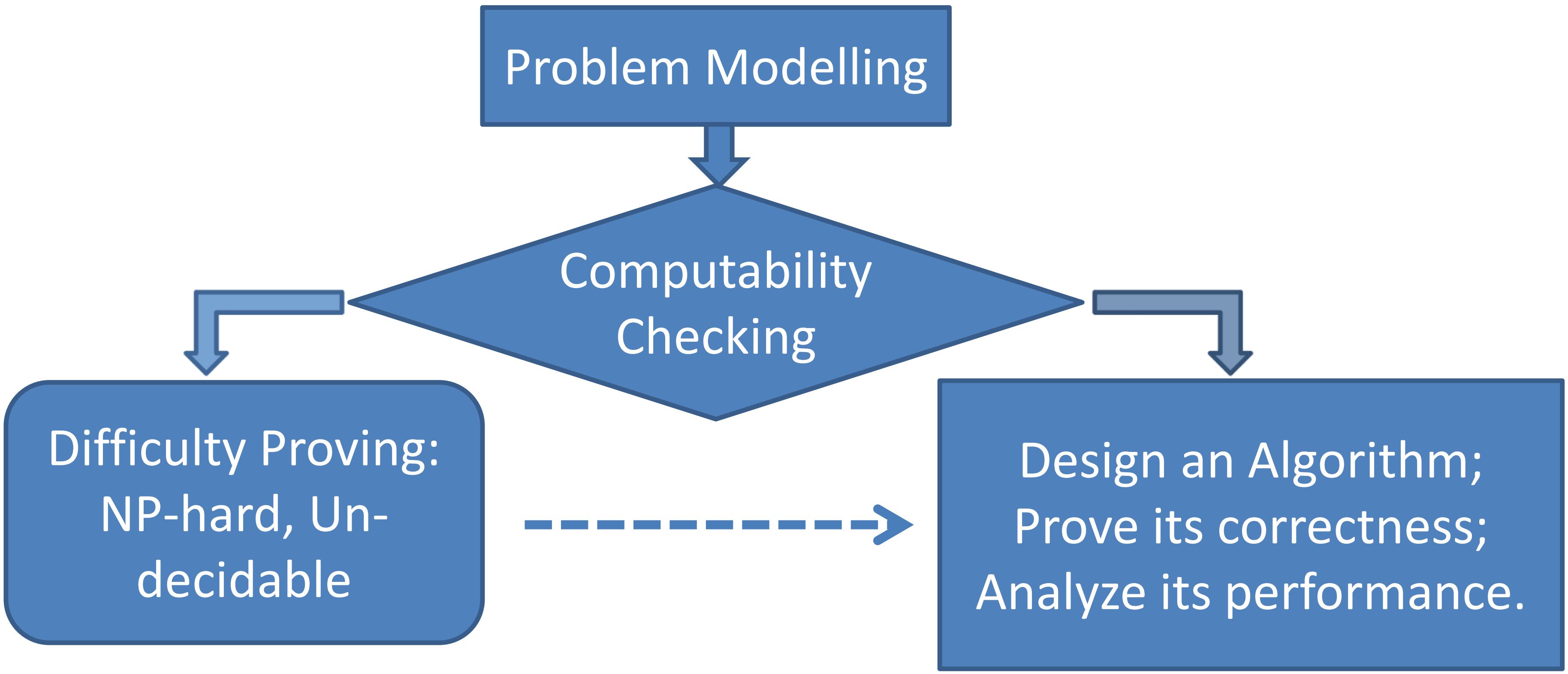
Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
  - "Everyone knows Moore's Law – a prediction made in 1965 by Intel co-founder Gordon Moore that the density of transistors in integrated circuits would continue to double every 1 to 2 years….in many areas, performance gains due to improvements in algorithms have vastly exceeded even the dramatic performance gains due to increased processor speed."
    - Excerpt from *Report to the President and Congress: Designing a Digital Future,* December 2010 (page 71).

Why Study Algorithms?

- important for all other branches of computer science
- plays a key role in modern technological innovation
- provides novel “lens” on processes outside of computer science and technology
- challenging (i.e., good for the brain!)
- fun

A general framework



Problems solved by algorithms

- Human Genome Project
- Internet search engine, routine,
- Electronic commerce, public-key cryptography and digital signature
- Oil supply, Airline scheduling, linear programming, distance in road map
- Matrices (Polynomial) Multiplication,
- Computational Geometry and Numerical algorithms

Hard Problems

- Efficient algorithms did not exist or have not found, who knows if it is existed.
- NP problems and NP-complete problems such as Traveling-salesman problem.
- Approximation algorithms and Random algorithms

Classification of Algorithms

- By problem types
  - Sorting
  - Searching
  - String processing
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems
- By design paradigms
  - Divide-and-conquer
  - Incremental
  - Dynamic programming
  - Greedy algorithms
  - Randomized/probabilistic

Steps in Algorithm Design + Analysis

1. Understand the problem
   - Specify the range of inputs the algorithm should handle
2. Learn about the model of the implementation technology
   - RAM (Random-access machine), sequential execution
3. Choosing between an exact and an approximate solution
   - Some problems cannot be solved exactly: nonlinear equations, evaluating definite integrals
   - Exact solutions may be unacceptably slow
4. Choose the appropriate data structures

5. Choose an algorithm design technique
   - General approach to solving problems algorithmically that is applicable to a variety of computational problems
   - Provide guidance for developing solutions to new problems
6. Specify the algorithm
   - Pseudocode: mixture of natural and programming language
7. Prove the algorithm's correctness
   - Algorithm yields the correct result for any legitimate input, in a finite amount of time
   - Mathematical induction, loop-invariants

8. Analyze the Algorithm

- Predicting the amount of resources required:
  - memory: how much space is needed?
  - computational time: how fast the algorithm runs?
- FACT: running time grows with the size of the input
- Input size (number of elements in the input)
  - Size of an array, polynomial degree, # of elements in a matrix, # of bits in the binary representation of the input, vertices and edges in a graph

Def: Running time = the number of primitive operations (steps) executed before termination

- Arithmetic operations (+, -, *), data movement, control, decision making (*if, while*), comparison

9. Coding the algorithm

- Verify the ranges of the input
- Efficient/inefficient implementation
- It is hard to prove the correctness of a program (typically done by testing)

Other important qualities

- Algorithms, data structures, programs
  - modularity
  - correctness
  - maintainability
  - functionality
  - robustness
  - user-friendliness
  - programmer time
  - simplicity
  - extensibility
  - reliability

Why Study the Efficiency?

- Given a problem, there are many algorithms to solve it, especially, brute-force search, a general but trivial algorithm.

- We hope the algorithm is as fast as possible. So we need to analyze and compare the algorithms.

Algorithm Efficiency vs. Speed

E.g.: sorting *n* numbers

Sort 10^6 numbers!

Friend's computer = 10^9 instructions/second

Friend's algorithm = $2n^2$ instructions

Your computer = 10^7 instructions/second

Your algorithm = $50n\lg n$ instructions

Your friend = $$\frac{2 * \left(10^6\right)^2 \text{instructions}}{10^9 \text{instructions} / \text{second}} = 2000 \text{seconds}$$

You = $$\frac{50 * \left(10^6\right) \lg 10^6 \text{instructions}}{10^7 \text{instructions} / \text{second}} \approx 100 \text{seconds}$$

20 times better!!

Types of Analysis

- Worst case
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
  - **Difficulties:** more complicated algorithms, could be slower for the type of data in encountered in practice
- Average case
  - Provides a prediction about the running time
  - Assumes that the input is random
  - **Difficulties**: complex analysis, guarantee the randomness of the input
- Best case
  - Input is the one for which the algorithm runs the fastest

How to Study the Efficiency?

How to compare the algorithms with respect to the time efficiency, the space efficiency and so on?!

The general principle of the estimation of the efficiency is the trade-off between precision and operability.

What Affects an Algorithm's Efficiency?

- Environment:
  - Pentium v.s. Core;
  - RISC (Reduced Instruction Set Computing) v.s. CISC (Complicated Instruction Set Computing);
  - 32 bits v.s. 64 bits;
  - 2M cache v.s. 8M cache.
  - ... ...
- Input:
  - Small size v.s. large size;
  - Good input v.s. bad input;

Theoretical Methodology

- We consider the arithmetic instructions (i.e., add, subtract, multiply, divide and so on), data movement instructions (i.e., load, store, copy and so on), and control instructions (i.e., subroutine call, return and so on) as elementary computer steps which is machine-independent and cost same and constant time;

- Given a particular input, the running time of an algorithm is the number of elementary computer steps.

Theoretical Methodology

- Describe the running time of the algorithm as a function of the input size.

  Conventionally:
  - For combinatorial problems such as sorting, the input size is the number of items in the input;
  - For numerical problems such as multiplication, the input size is the number of bits in binary to represent the input;
  - For graphical problems, the input size is described with the numbers of vertices and the number of edges in the graph respectively.

- Analyze the worst case (i.e., worst-case analysis).

Order of growth

- *Alg.:* MIN (a[1], …, a[n])

```
m ← a[1];
for i ← 2 to n
    if a[i] < m
      then m ← a[i];
```

- Running time:

  $T(n) = 1$ [first step] + (n) [for loop] + $(n-1)$ [if condition] + $(n-1)$ [the assignment in then] = $3n - 1$

- Order (rate) of growth:
  - The leading term of the formula
  - Gives a simple characterization of the algorithm's efficiency
  - Expresses the asymptotic behavior of the algorithm
- $T(n)$ grows like n

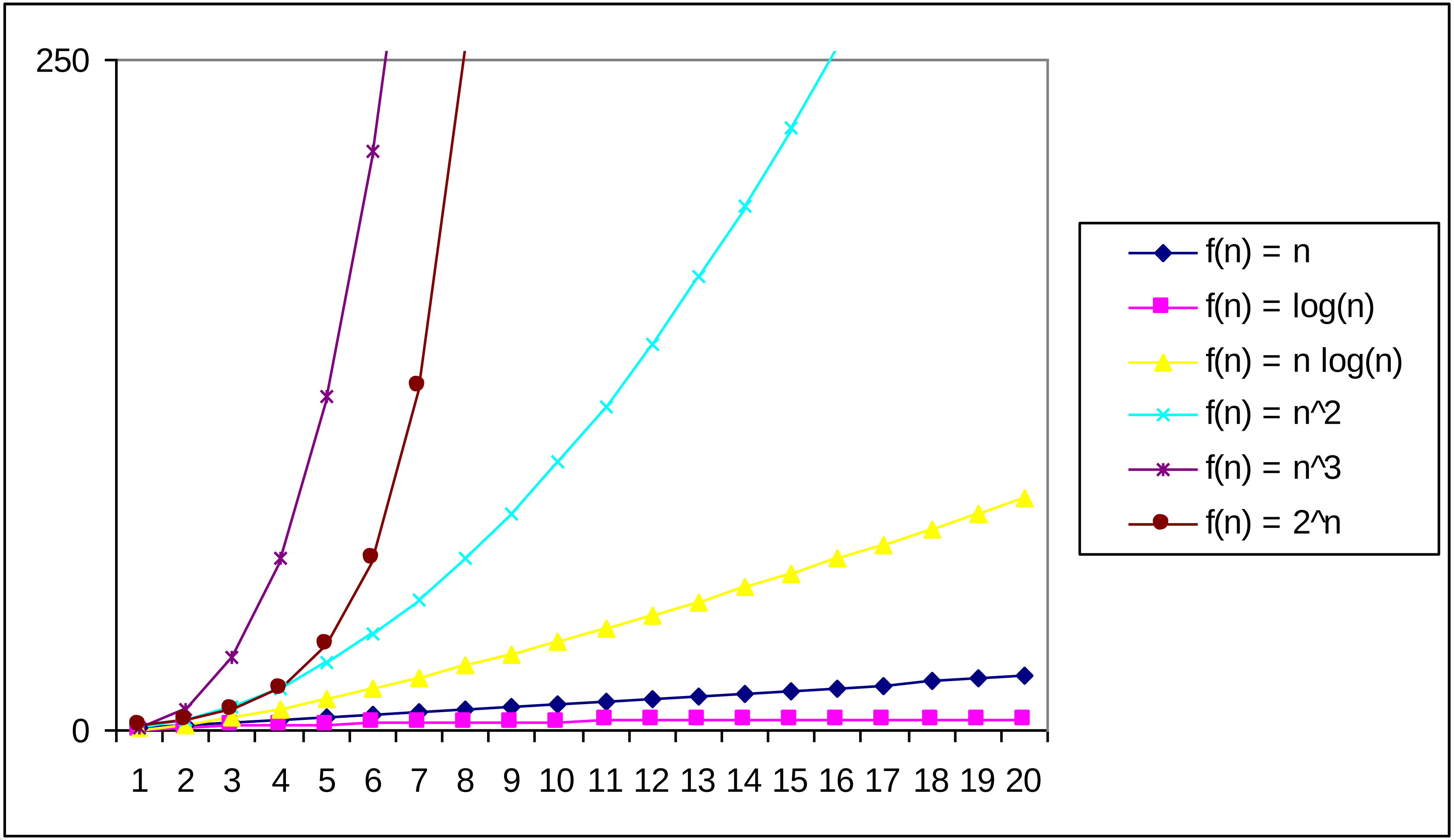
Typical Running Time Functions

- 1 (constant running time):
  - Instructions are executed once or a few times
- logN (logarithmic)
  - A big problem is solved by cutting the original problem in smaller sizes, by a constant fraction at each step
- N (linear)
  - A small amount of processing is done on each input element
- N logN
  - A problem is solved by dividing it into smaller problems, solving them independently and combining the solution

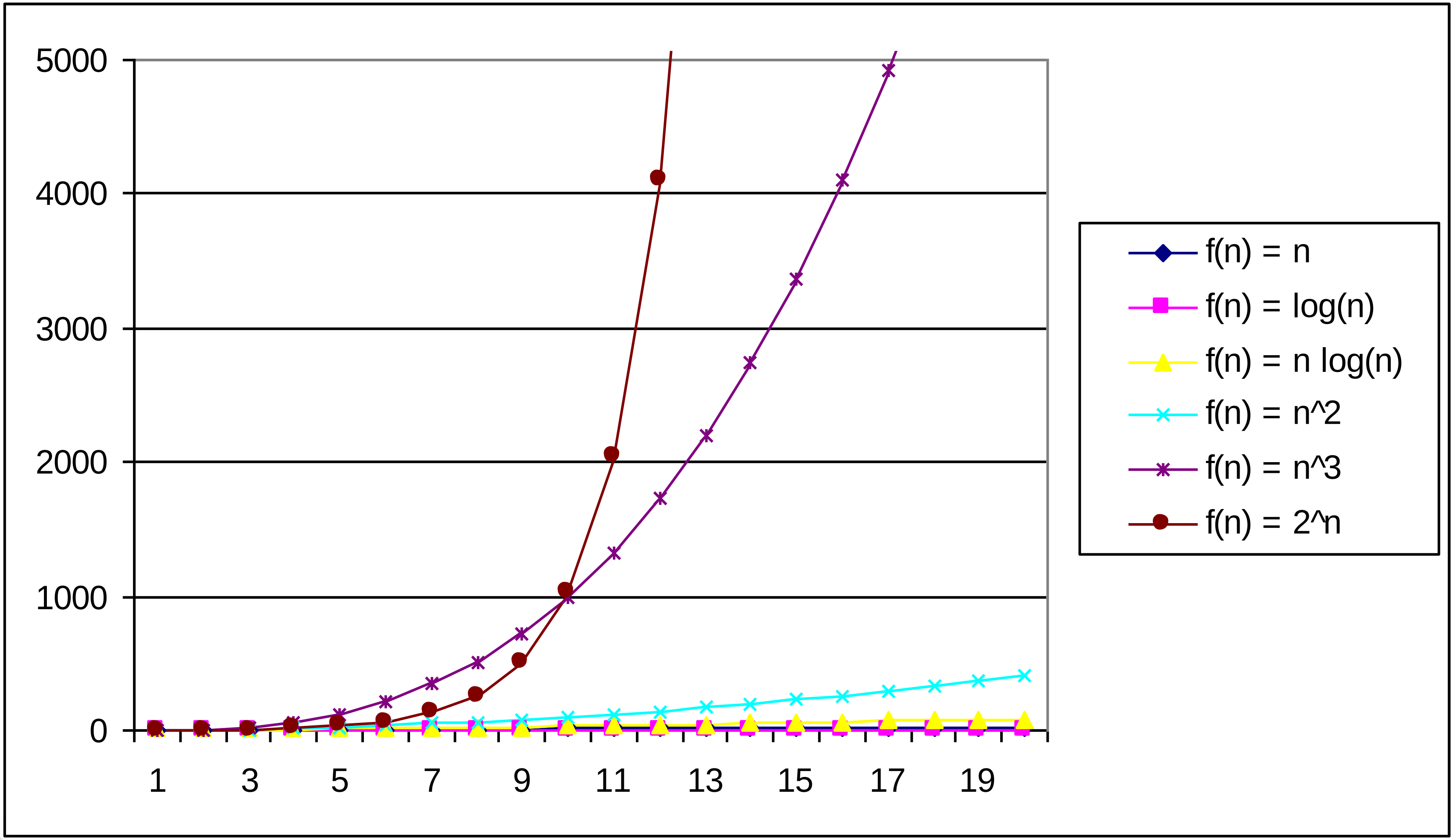
Typical Running Time Functions

- N^2 (quadratic)
  - Typical for algorithms that process all pairs of data items (double nested loops)
- N^3 (cubic)
  - Processing of triples of data (triple nested loops)
- N^K (polynomial)
- 2^N (exponential)
  - Few exponential algorithms are appropriate for practical use

Why Faster Algorithms?



Why Faster Algorithms?



Logarithms

- In algorithm analysis we often use the notation **"log n"** without specifying the base

Binary logarithm $\lg n = \log_2 n$

Natural logarithm $\ln n = \log_e n$

$$\lg^k n = (\lg n)^k$$

$$\lg \lg n = \lg(\lg n)$$

$$\log x^y = y \log x$$

$$\log xy = \log x + \log y$$

$$\log \frac{x}{y} = \log x - \log y$$

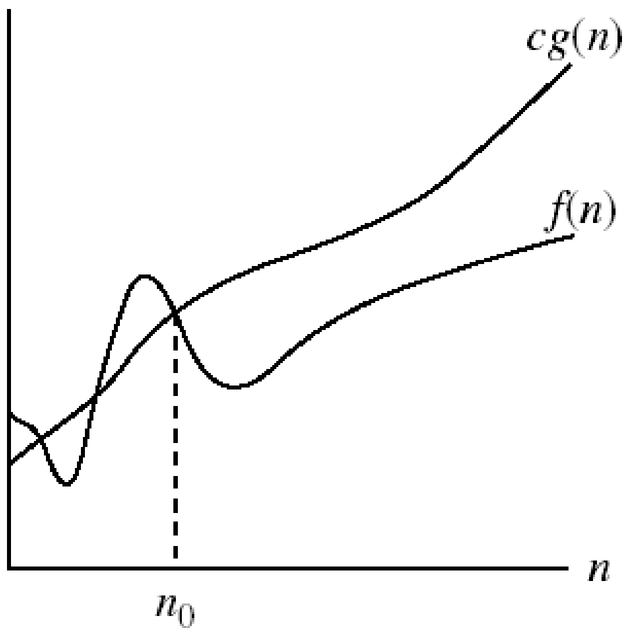
$$\log_a x = \log_a b \log_b x$$

$$a^{\log_b x} = x^{\log_b a}$$

Asymptotic notations

- *O-notation*

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



- Intuitively: $O(g(n))$ = the set of functions with a smaller or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Examples

- $2n^2 = O(n^3)$: $2n^2 \le cn^3 \Rightarrow 2 \le cn \Rightarrow c = 1$ and $n_0 = 2$

- $n^2 = O(n^2)$: $n^2 \le cn^2 \Rightarrow c \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

- $1000n^2+1000n = O(n^2)$:

$1000n^2+1000n \le cn^2 \le cn^2 + 1000n \Rightarrow c=1001$ and $n_0 = 1$

- $n = O(n^2)$: $n \le cn^2 \Rightarrow cn \ge 1 \Rightarrow c = 1$ and $n_0 = 1$

Examples

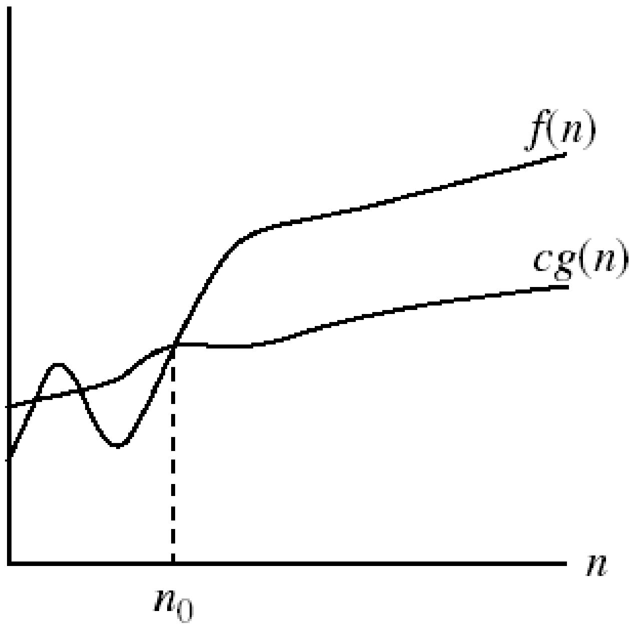
- *E.g.:* prove that $n^2 \neq O(n)$
  - Assume $\exists\ c$ & n_0 such that: $\forall\ n \geq n_0$: $n^2 \leq cn$
  - Choose $n = 2 * \max(n_0, c)$
  - $n^2 = n * n \geq n * 2c \Rightarrow n^2 \geq cn$

    contradiction!!!

Asymptotic notations (cont.)

- *Ω - notation*

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}\ .$$



- Intuitively: $\Omega(g(n))$ = the set of functions with a larger or same order of growth as $g(n)$

$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Examples

– $5n^2 = \Omega(n)$

$\exists\ c, n_0$ such that: $0 \le cn \le 5n^2 \Rightarrow cn \le 5n^2 \Rightarrow c = 1$ and $n_0 = 1$

– $100n + 5 \ne \Omega(n^2)$

$\exists\ c, n_0$ such that: $0 \le cn^2 \le 100n + 5$

$100n + 5 \le 100n + 5n\ (\forall\ n \ge 1) = 105n$

$cn^2 \le 105n \Rightarrow n(cn – 105) \le 0$

Since n is positive $\Rightarrow cn – 105 \le 0 \Rightarrow n \le 105/c$

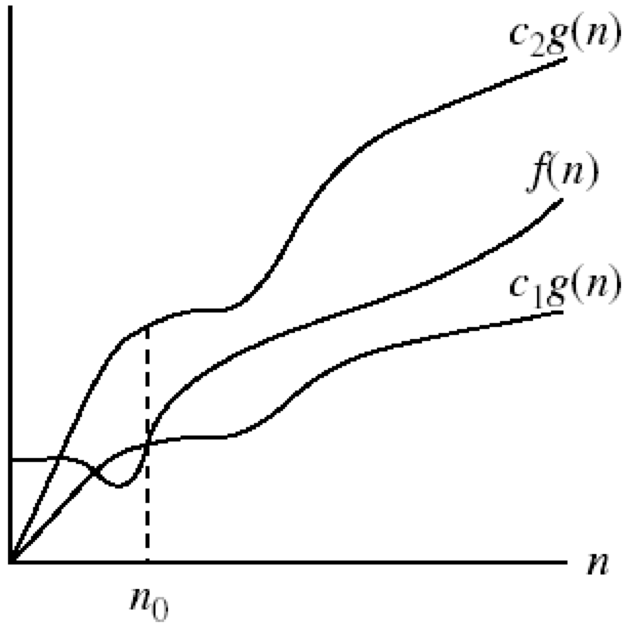
$\Rightarrow$ contradiction: n cannot be smaller than a constant

– $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(\log n)$

Asymptotic notations (cont.)

- *Θ-notation*

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \} .$$



- Intuitively $\Theta(g(n))$ = the set of functions with the same order of growth as $g(n)$

$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Examples

- $n^2/2 - n/2 = \Theta(n^2)$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \le \frac{1}{2} n^2 \ \forall n \ge 0 \quad \Rightarrow \quad c_2 = \frac{1}{2}$
  - $\frac{1}{2} n^2 - \frac{1}{2} n \ge \frac{1}{2} n^2 - \frac{1}{2} n * \frac{1}{2} n \ (\forall n \ge 2) = \frac{1}{4} n^2 \Rightarrow \quad c_1 = \frac{1}{4}$
- $n \ne \Theta(n^2)$: $c_1 n^2 \le n \le c_2 n^2 \Rightarrow$ only holds for: $n \le 1/c_1$
- $6n^3 \ne \Theta(n^2)$: $c_1 n^2 \le 6n^3 \le c_2 n^2 \Rightarrow$ only holds for: $n \le c_2/6$
- $n \ne \Theta(\log n)$: $c_1 \log n \le n \le c_2 \log n$

  $\Rightarrow c_2 \ge n/\log n, \ \forall n \ge n_0$ – impossible

Asymptotic Notations

- A way to describe behavior of functions in the limit
  - How we indicate running times of algorithms
  - Describe the running time of an algorithm as n grows to ∞
- O notation: asymptotic "less than": $f(n)$ "$\leq$" $g(n)$
- Ω notation: asymptotic "greater than": $f(n)$ "$\geq$" $g(n)$
- Θ notation: asymptotic "equality": $f(n)$ "$=$" $g(n)$

Comparisons of Functions

- *Theorem:*

  $$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

- Transitivity**:**
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Asymptotic Notations - Examples

- Θ notation
  - $n^2/2 - n/2 \quad = \Theta(n^2)$
  - $(6n^3 + 1)\lg n/(n + 1) = \Theta(n^2 \lg n)$
  - n vs. $n^2 \quad n \neq \Theta(n^2)$
- Ω notation
  - n vs. $2n \quad n = \Omega(2n)$
  - n^3 vs. $n^2 \quad n^3 = \Omega(n^2)$
  - n vs. $\log n \quad n = \Omega(\log n)$
  - n vs. $n^2 \quad n \neq \Omega(n^2)$
- O notation
  - $2n^2$ vs. $n^3 \quad 2n^2 = O(n^3)$
  - n^2 vs. $n^2 \quad n^2 = O(n^2)$
  - n^3 vs. $n\log n \quad n^3 \neq O(n\lg n)$

Transitivity:

$$
\begin{aligned}
f(n) &= \Theta(g(n)) \text{ and } g(n) = \Theta(h(n)) &&\text{imply} && f(n) = \Theta(h(n))\,, \\
f(n) &= O(g(n)) \text{ and } g(n) = O(h(n)) &&\text{imply} && f(n) = O(h(n))\,, \\
f(n) &= \Omega(g(n)) \text{ and } g(n) = \Omega(h(n)) &&\text{imply} && f(n) = \Omega(h(n))\,, \\
f(n) &= o(g(n)) \text{ and } g(n) = o(h(n)) &&\text{imply} && f(n) = o(h(n))\,, \\
f(n) &= \omega(g(n)) \text{ and } g(n) = \omega(h(n)) &&\text{imply} && f(n) = \omega(h(n))\,.
\end{aligned}
$$

Reflexivity:

$$
\begin{aligned}
f(n) &= \Theta(f(n))\,, \\
f(n) &= O(f(n))\,, \\
f(n) &= \Omega(f(n))\,.
\end{aligned}
$$

Symmetry:

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))\,.$$

Transpose symmetry:

$$
\begin{aligned}
f(n) &= O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))\,, \\
f(n) &= o(g(n)) \text{ if and only if } g(n) = \omega(f(n))\,.
\end{aligned}
$$

$$
\begin{array}{lll}
f(n) = O(g(n)) & \text{is like} & a \le b \,, \\
f(n) = \Omega(g(n)) & \text{is like} & a \ge b \,, \\
f(n) = \Theta(g(n)) & \text{is like} & a = b \,, \\
f(n) = o(g(n)) & \text{is like} & a < b \,, \\
f(n) = \omega(g(n)) & \text{is like} & a > b \,.
\end{array}
$$

Floors and ceilings

For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x"). For all real x,

$$
x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1 \,. \tag{3.3}
$$

Polynomials

Given a nonnegative integer d, a ***polynomial in n of degree d*** is a function $p(n)$ of the form

$$
p(n) = \sum_{i=0}^{d} a_i n^i \,,
$$

Exponentials

For all real $a > 0$, m, and n, we have the following identities:

$$
\begin{aligned}
a^0 &= 1 , \\
a^1 &= a , \\
a^{-1} &= 1/a , \\
(a^m)^n &= a^{mn} , \\
(a^m)^n &= (a^n)^m , \\
a^m a^n &= a^{m+n} .
\end{aligned}
$$

Logarithms

We shall use the following notations:

$$
\begin{aligned}
\lg n &= \log_2 n && \text{(binary logarithm)} , \\
\ln n &= \log_e n && \text{(natural logarithm)} , \\
\lg^k n &= (\lg n)^k && \text{(exponentiation)} , \\
\lg \lg n &= \lg(\lg n) && \text{(composition)} .
\end{aligned}
$$

For all real $a > 0$, $b > 0$, $c > 0$, and n,

$$
\begin{aligned}
a &= b^{\log_b a} , \\
\log_c(ab) &= \log_c a + \log_c b , \\
\log_b a^n &= n \log_b a , \\
\log_b a &= \frac{\log_c a}{\log_c b} , \\
\log_b(1/a) &= -\log_b a , \\
\log_b a &= \frac{1}{\log_a b} , \\
a^{\log_b c} &= c^{\log_b a} ,
\end{aligned}
$$

Functional iteration

We use the notation $f^{(i)}(n)$ to denote the function $f(n)$ iteratively applied i times to an initial value of n. Formally, let $f(n)$ be a function over the reals. For non-negative integers i, we recursively define

$$f^{(i)}(n) = \begin{cases} n & \text{if } i = 0 , \\ f(f^{(i-1)}(n)) & \text{if } i > 0 . \end{cases}$$

For example, if $f(n) = 2n$, then $f^{(i)}(n) = 2^i n$.

The iterated logarithm function

$$\lg^* n = \min \left\{ i \geq 0 : \lg^{(i)} n \leq 1 \right\} .$$

The iterated logarithm is a *very* slowly growing function:

$$\begin{aligned} \lg^* 2 &= 1 , \\ \lg^* 4 &= 2 , \\ \lg^* 16 &= 3 , \\ \lg^* 65536 &= 4 , \\ \lg^* (2^{65536}) &= 5 . \end{aligned}$$

Fibonacci numbers

We define the ***Fibonacci numbers*** by the following recurrence:

$$\begin{aligned} F_0 &= 0\,, \\ F_1 &= 1\,, \\ F_i &= F_{i-1} + F_{i-2} \qquad \text{for } i \geq 2\,. \end{aligned} \tag{3.22}$$

Thus, each Fibonacci number is the sum of the two previous ones, yielding the sequence

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ \ldots\,.$$

Fibonacci numbers are related to the ***golden ratio*** ϕ and to its conjugate $\hat{\phi}$, which are the two roots of the equation

$$x^2 = x + 1 \tag{3.23}$$

$$\begin{aligned} \phi &= \frac{1+\sqrt{5}}{2} \\ &= 1.61803\ldots\,, \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \\ &= -.61803\ldots\,. \end{aligned}$$

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}\,,$$

Polynomial-Time

Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes 2^N time or worse for inputs of size N.
- Unacceptable in practice.

n! for stable matching
with n men and n women

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor C.

> There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by $c N^d$ steps.

Def. An algorithm is poly-time if the above scaling property holds.

choose $C = 2^d$

Worst-Case Polynomial-Time

Def. An algorithm is efficient if its running time is polynomial.

Justification: It really works in practice!

- Although $6.02 \times 10^{23} \times N^{20}$ is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

Exceptions.

- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.

simplex method
Unix grep

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long