Fundamentals of the Analysis of Algorithm Efficiency

CS3230: Design and Analysis of Algorithms

Roger Zimmermann

National University of Singapore

Spring 2017

Ch. 2: Analysis of Algorithm Efficiency

Topics: What we will cover today

- Analysis framework
- Asymptotic notations and basic efficiency classes
- Mathematical analysis of non-recursive/recursive algorithms
- Example: Fibonacci numbers
- Empirical analysis of algorithms
- Algorithm visualization

Analysis of Algorithms

- Analysis of algorithms usually means the investigation of the efficiency of an algorithm in terms of running time and memory space.
- Efficiency can be expressed quantitatively using the big oh O, the big omega Ω, and the big theta Θ notations.
- The main mathematical techniques are sum manipulation for non-recursive algorithms and recurrence relations solving for recursive algorithms.
- Empirical analysis and algorithm visualization complement the above pure mathematical techniques.

Analysis Framework

- The main focus is on time efficiency. But the framework should also be applicable for space efficiency.
- Factors to consider are:
  1. Measuring the input size.
  2. Units for measuring running time.
  3. Orders of growth.
  4. Worst-case, best-case, average-case efficiencies.

Q: Why are all three cases (worst, best, average) of interest?

Input Size

- The running time of an algorithm depends on the amount of data it has to process.
- The efficiency of an algorithm is a function of the input size.
- One or more parameters may be needed to express the input size.

Input Size: Examples

- For sorting or searching, the input size will be the number of data items to be sorted or searched.
- To evaluate a polynomial $p(x) = a_n x^n + \ldots + a_0$, the input size is the degree n.
- To process the graph $G = (V, E)$, the input size is a combination of $|V|$ and $|E|$.
- But to multiply two square matrices of order n, the input size could be either n or the total number of matrix entries $N = 2n^2$.

  (I.e., 2-dimensional input $\Rightarrow n \times n$ or $n \times m$ input size.)

Input Size: Properties of Integers

- For algorithms dealing with properties (e.g., primeness) of integers, the input size should not be the integer n but the number of bits in the binary representation of n:

$$\lfloor \log_2 n \rfloor + 1.$$

- Note that for a non-negative integer $2^k \le n < 2^{k+1}$, there are $k + 1$ bits in the binary representation of n and $k = \lfloor \log_2 n \rfloor$.

Units for Measuring Running Time

- Since computer speed and architecture, the choice of language, the quality of the compiler, etc., vary but influence the running time, they are actually extraneous factors and so actual time units such as seconds, minutes, hours, etc., are not useful as the unit for measuring running time.

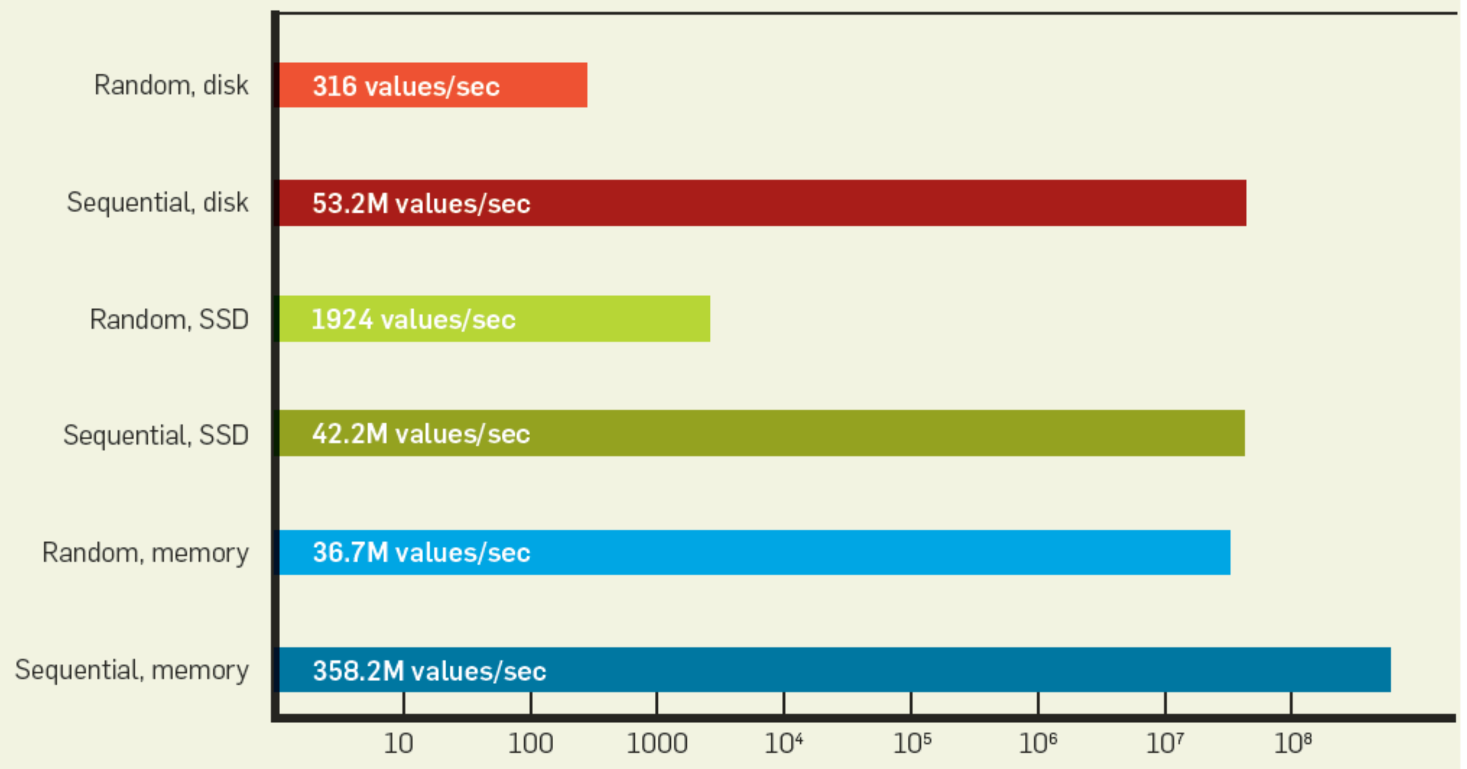
  ($\Rightarrow$ However, in practice real-world factors may need to be considered, see page 10.)
- Computer scientists count the number of basic operations that an algorithm has to execute to express its time efficiency.
- A basic operation is usually the most time-consuming operation in the algorithm's inner-most loop.

Units for Measuring Running Time (2)

- Examples
  - For sorting algorithms, key comparison is the basic operation.
  - For arithmetic algorithms, arithmetic operations are the basic operations.

Random versus Sequential Data Access

Figure 3. Comparing random and sequential access in disk and memory.



* Disk tests were carried out on a freshly booted machine (a Windows 2003 server with 64GB RAM and eight 15,000RPM SAS disks in RAID5 configuration) to eliminate the effect of operating-system disk caching. SSD test used a latest generation Intel high-performance SATA SSD.

“The Pathologies of Big Data,” by Adam Jacobs, Communications of the ACM, August 2009

The Actual Running Time

- Let $C(n)$ be the number of basic operations needed to process an input of size n.
- If for a particular implementation, the basic operations take time c_{op}, then the running time to process an input of size n is

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

$$T(n) : \text{seconds};\, c_{op} : \text{seconds per operation};\, C(n) : \text{operations}.$$

- This proportional relationship explains why $C(n)$ is a meaningful indicator of time efficiency.

Ignoring Multiplicative Constants

- Consider the question:

  If $C(n) = \frac{n(n-1)}{2}$, how much longer does the algorithm take if the input size is double?

- We have

$$\frac{T(2n)}{T(n)} \approx \frac{c_{op}C(2n)}{c_{op}C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4.$$

- The example illustrates that order of growth, which is about how much $C(n)$ increases as n increases, rather than multiplicative constants, are essential in expressing the time efficiency of an algorithm.

Order of Growth

- The most important information we want to get from the time efficiency of an algorithm is its ability to process large input.
- The most informative answer is provided by investigating the order of growth of the function $C(n)$, the count of the basic operations to process an input of size n.

Orders of Growth: Illustrations

- Consider how fast $C(n)$ is increasing as the input size n increases from 10 to 100 ($= m$) for the following functions (assuming base 2 logarithm):

$\frac{m}{n}$	$\frac{\log m}{\log n}$	$\frac{m}{n}$	$\frac{m \log m}{n \log n}$	$\frac{m^2}{n^2}$	$\frac{m^3}{n^3}$	$\frac{2^m}{2^n}$	$\frac{m!}{n!}$
10	2	10	20	10^2	10^3	1.3×10^{27}	2.6×10^{151}

- If it takes 1 second to process an input of size $n = 10$, then it takes 41 billion billion years to process an input for the exponential function $C(n) = 2^n$ and 0.8×10^{144} years for the factorial function $C(n) = n!$.

Orders of Growth: A Remark

- The base of a logarithm function is not important when order of growth is concerned.
- This is because logarithms of different bases differ by a constant multiple:

$$\log_a n = \log_a b \times \log_b n.$$

Consider both Input Size and Input Attributes

- In the preceeding discussions we gave the impression that as long as the input size is n the time efficiency is $C(n)$.
- This is actually an over-simplification because not all inputs of size n will lead to the same time efficiency $C(n)$.
- To highlight the fact that efficiency depends on both the input size n and the particular input I, we should express the efficiency as $C(n, I)$ to stress that the efficiency also depends on the specific input I whose size is n.

Best-Case Efficiency

- The best case efficiency of an algorithm for all inputs of size n is thus

$$C_{\text{best}}(n) = \min_{|I|=n} C(n, I)$$

- where $|I|$ denotes the size of input I.

- Example:
  - Sorting an already sorted array.

Average-Case Efficiency

- The average case efficiency of an algorithm for all inputs of size n is thus

$$C_{\text{average}}(n) = \sum_{|I|=n} P(I)C(n, I)$$

- where $P(I)$ is the probability that input I occurs among all inputs of size n.
- Average-case: NOT the average of worst and best case.

Worst-Case Efficiency

- The worst case efficiency of an algorithm for all inputs of size n is thus

$$C_{\text{worst}}(n) = \max_{|I|=n} C(n, I).$$

- Example: ?

Best-, Average-, Worst-Case Efficiency: An Example

- Consider the algorithm that searches for the key K in the array $A[0..n-1]$ (i.e., *SequentialSearch()*):

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

- The input to the algorithm is a key K and an array $A[0..n-1]$ of size n.
- We take $C(n, I)$ to be the number of key comparisons $A[i] \neq K$.

Best-, Worst-Case Efficiency: An Example

- Let $C(n)$ be the best-case efficiency, then

$$C_{\text{best}}(n) = 1.$$

- Let $C(n)$ be the worst-case efficiency, then

$$C_{\text{worst}}(n) = n.$$

Average-Case Efficiency: An Example

- Assumptions
  - The probability of a successful search is p, $(0 \leq p \leq 1)$.
  - The probability of the first match occuring in the i^{th} position is the same for every i, namely $\frac{p}{n}$.
- Let $C_{\text{avg}}(n)$ be the average-case efficiency, then

$$C_{\text{avg}}(n) = \left(1 \times \frac{p}{n} + 2 \times \frac{p}{n} + \cdots + n \times \frac{p}{n}\right) + (1-p) \times n$$

$$= \frac{p}{n} \times (1 + 2 + \cdots + n) + (1-p)n = \frac{p(n+1)}{2} + (1-p)n.$$

Average-Case Efficiency: An Example (2)

- Result check. Recall:

$$C_{\text{avg}}(n) = \frac{p(n+1)}{2} + (1-p)n$$

- If search must be successful, i.e., $p = 1$, then $C_{\text{avg}}(n) = \frac{n+1}{2}$.
- If search must be unsuccessful, i.e., $p = 0$, then $C_{\text{avg}}(n) = n$.

Amortized Efficiency

- Applies to a sequence of operations performed on the same data structure.
  - A single operation can be expensive but the total time for an entire sequence of n such operations is always significantly better.
  - “Amortize” the high cost of such a worst-case occurence over the entire sequence.
- Discovered by Robert Endre Tarjan, Princeton U. (ACM Turing Award winner, 1986). E.g., Splay Tree data structure.
- “Amortized Computational Complexity.” SIAM Journal on Algebraic and Discrete Methods, vol. 6, no. 2, 1985, 306-318.

Asymptotic Notations and Basic Efficiency Classes

- The asymptotic notations of interest are

$$O, \quad \Omega, \quad \Theta.$$

- The basic efficiency classes of interest are

By def is big O

$$1, \quad \log n, \quad n, \quad n \log n, \quad n^2, \quad n^3, \quad 2^n, \quad n!$$

known respectively as the constant, logarithm, linear, $n \log n$, quadratic, cubic, exponential, and factorial classes.

Asymptotic Order of Growth

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

  $f(n) \in O(g(n))$: set (or class) of functions $f(n)$ that grow no faster than $g(n)$ (*upper bound*).

  $f(n) \in \Theta(g(n))$: set (or class) of functions $f(n)$ that grow at the same rate as $g(n)$ (*tight bound*).

  $f(n) \in \Omega(g(n))$: set (or class) of functions $f(n)$ that grow at least as fast as $g(n)$ (*lower bound*).

The Need for Asymptotic Notations O, Ω, Θ

- The basic operation count $C(n)$ of an algorithm can be very complicated to be described precisely.

  e.g., $C(n) = \frac{n(n-1)}{2}$

- But we are more interested in the order of growth of $C(n)$ rather than finding its exact value.

  e.g., $C(n) \approx 0.5n^2 \approx cn^2$

- The asymptotic notations allow us to describe $C(n)$ in terms of upper bound, lower bound, and tight bound, which are sufficient for the purpose of assessing the order of growth of $C(n)$.

Eventually or Asymptotically Positive Functions

- Recall that the set of nonnegative integers is $\mathbb{N} = \{0, 1, 2, \cdots\}$.
- Let $\mathbb{R}$ be the set of real numbers.
- A function on $\mathbb{N}$, $f : \mathbb{N} \to \mathbb{R}$, is eventually positive if there is an integer n_0 such that for all integers $n \geq n_0$, $f(n) > 0$.
- For example, the function $\log_{100} \log_{100} n$ is eventually positive. Here we may choose $N = 101$.
- For some functions $f(n)$ the domain may be $\mathbb{N} \setminus \{0, \cdots, k\}$ if $f(0), \cdots, f(k)$ are not defined.
- The O, Ω, and Θ notations require that every member function is asymptotically nonnegative.

The O-Notation ("Upper Bound")

- Let $t(n)$, $g(n)$ be eventually positive functions on $\mathbb{N}$. We say $t(n)$ is in $O(g(n))$ if there is a constant $c > 0$ and an integer n_0 such that for all $n \geq n_0$:
$$t(n) \leq cg(n).$$
- For example,
$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 \in O(n^d),$$
for any $0 < a_d \in \mathbb{R}, d \in \mathbb{N}$.

The Ω-Notation ("Lower Bound")

- Let $t(n)$, $g(n)$ be eventually positive functions on $\mathbb{N}$. We say $t(n)$ is in $\Omega(g(n))$ if there is a constant $c > 0$ and an integer n_0 such that for all $n \geq n_0$:

$$t(n) \geq cg(n).$$

- For example,

$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 \in \Omega(n^{d'}),$$

for any $0 < a_d \in \mathbb{R}, d, d' \in \mathbb{N}$ and $d \geq d'$.

The Θ-Notation ("Tight Bound")

- Let $t(n)$, $g(n)$ be eventually positive functions on $\mathbb{N}$. We say $t(n)$ is in $\Theta(g(n))$ if there are constants $c_1, c_2 > 0$ and an integer n_0 such that for all $n \geq n_0$:

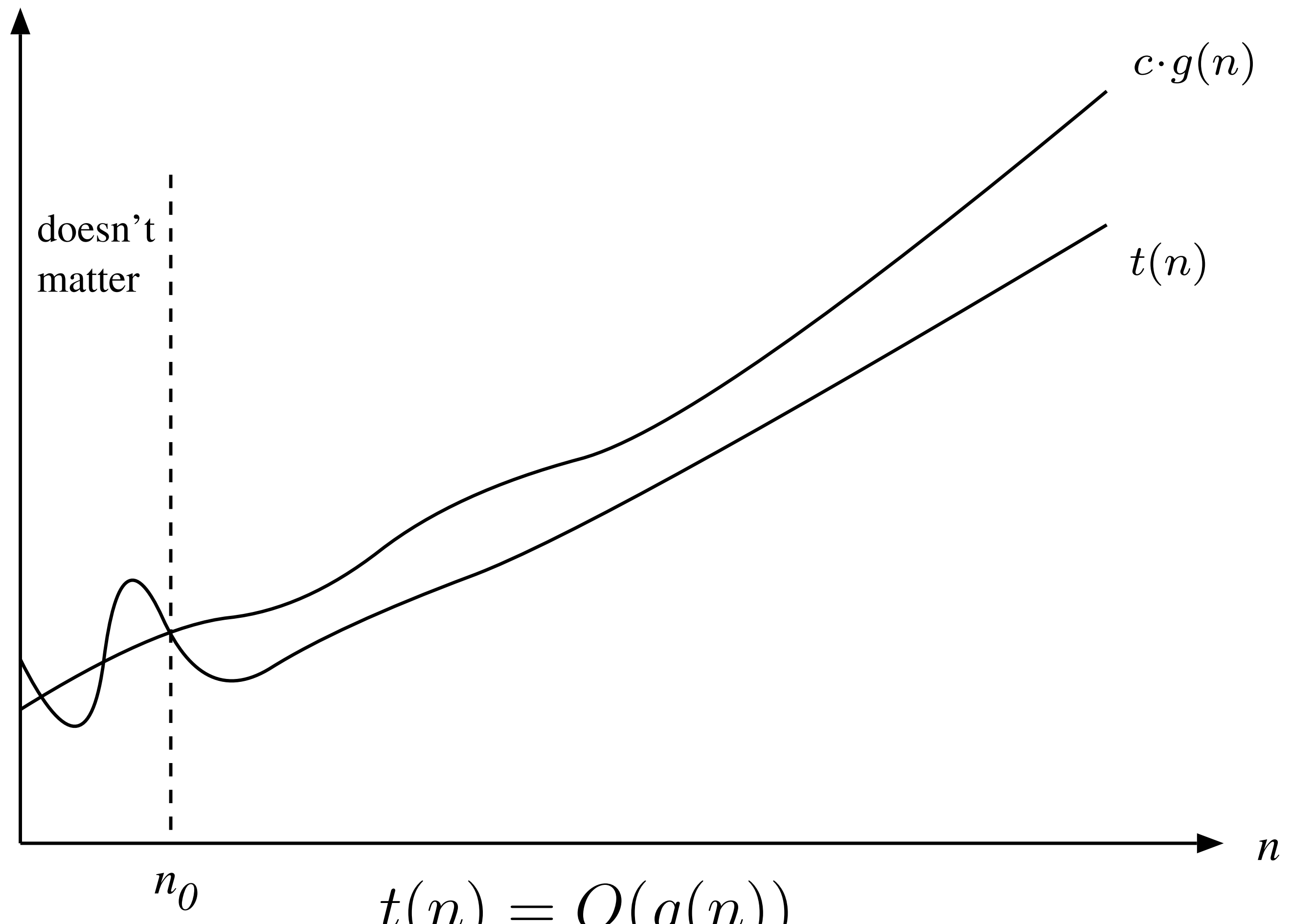
$$c_1 g(n) \leq t(n) \leq c_2 g(n).$$

- For example,

$$a_d n^d + a_{d-1} n^{d-1} + \cdots + a_0 \in \Theta(n^d),$$

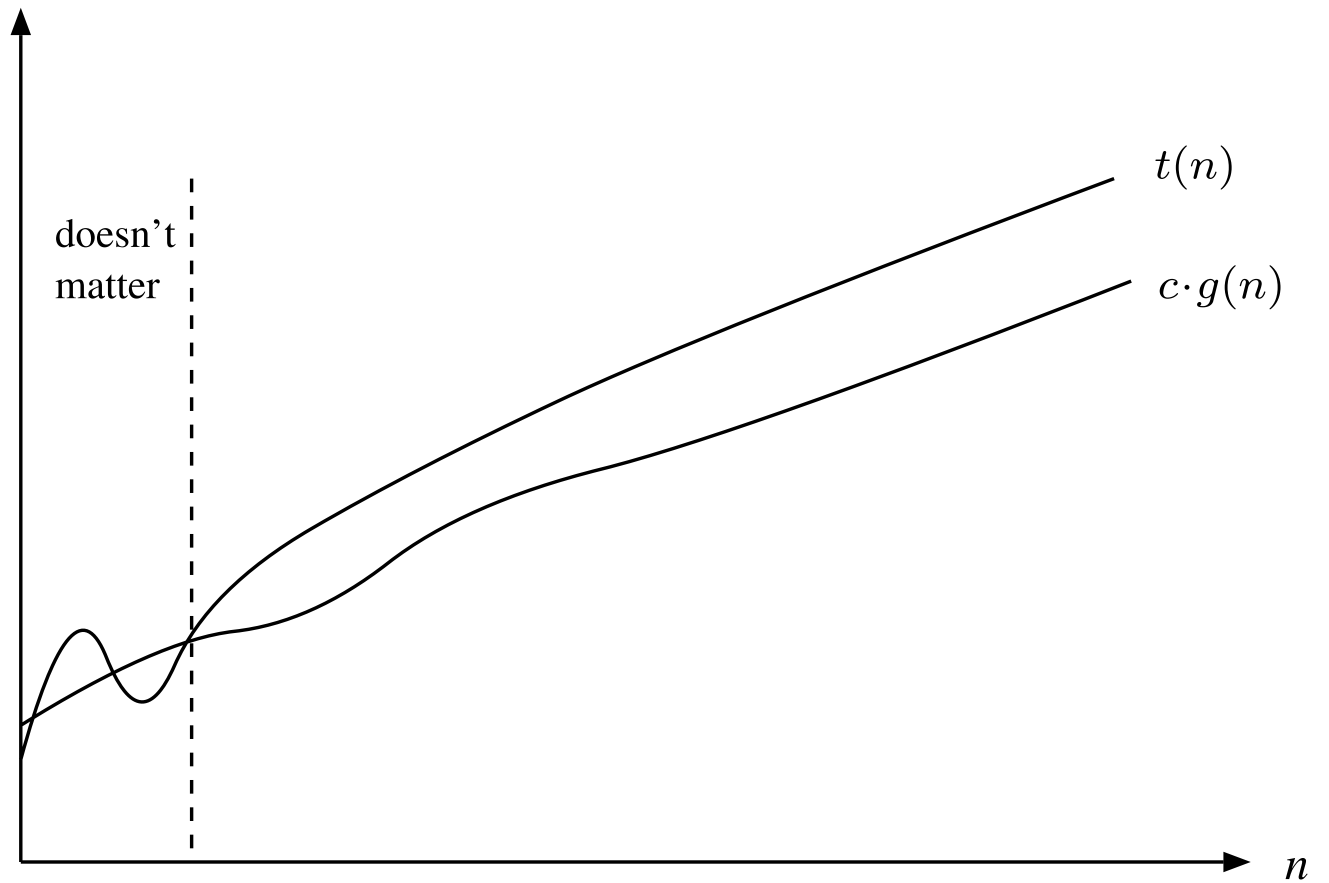
for any $0 < a_d \in \mathbb{R}, d \in \mathbb{N}$.

The O-Notation



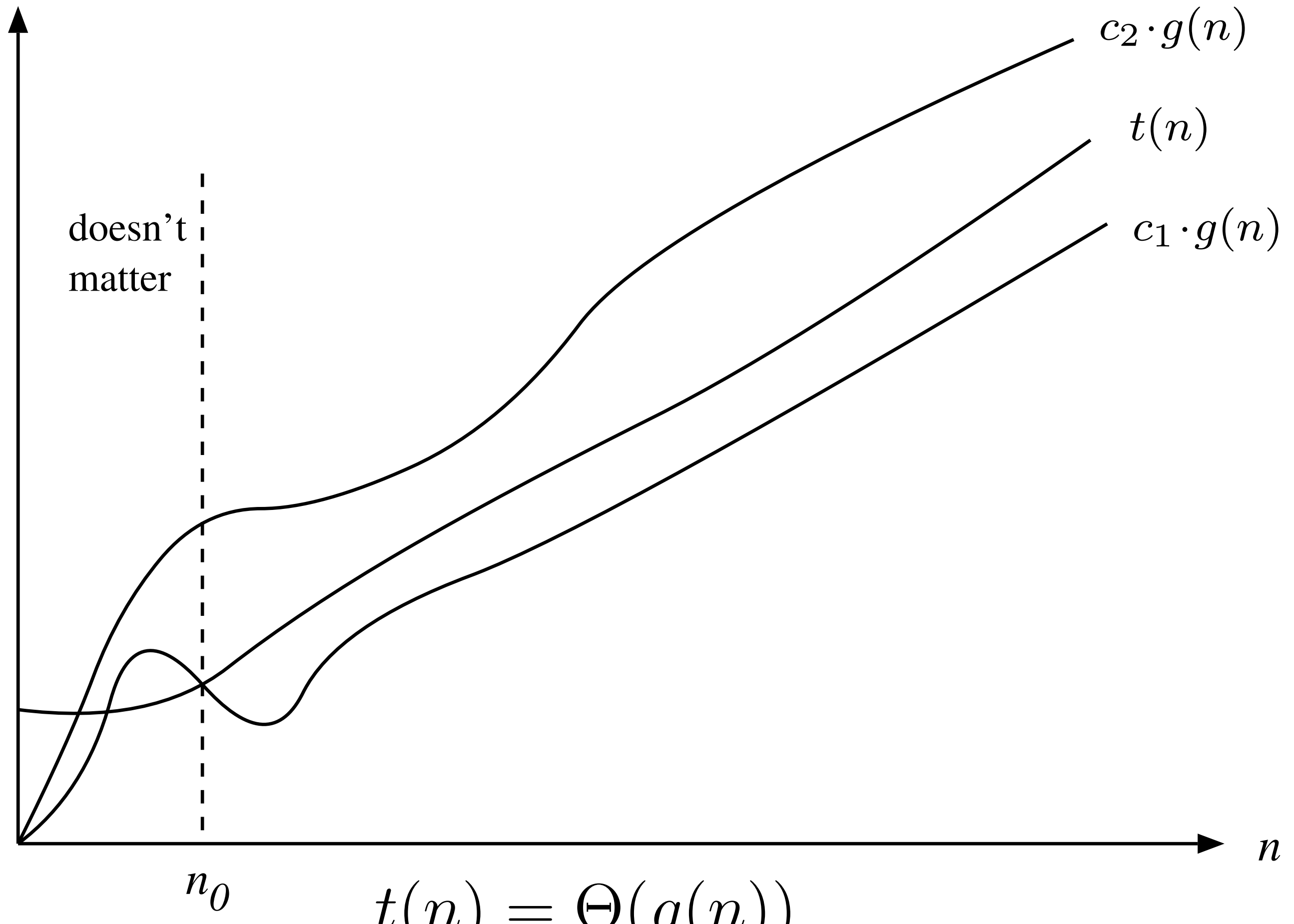
$$t(n) = O(g(n))$$

The Ω-Notation



$$t(n) = \Omega(g(n))$$

The Θ-Notation



$$t(n) = \Theta(g(n))$$

The Theorem for Asymptotic Simplification

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

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

- Proof. There are $C_1, C_2 > 0, N_1, N_2 \in \mathbb{N}$ such that
  $$\forall n \geq N_1, t_1(n) \leq C_1 g_1(n); \text{ and } \forall n \geq N_2, t_2(n) \leq C_2 g_2(n).$$

  Let $C_0 \geq \max(C_1, C_2)$ and let $N_0 \geq \max(N_1, N_2)$. Consider the sum $t_1(n) + t_2(n)$ for $n \geq N_0$:

$$\begin{aligned} t_1(n) + t_2(n) &\leq C_1 g_1(n) + C_2 g_2(n), n \geq N_0 \\ &\leq C_0(g_1(n) + g_2(n)) \\ &\leq C_0 \times 2 \times \max(g_1(n), g_2(n)). \end{aligned}$$

  Thus $t_1(n) + t_2(n) \in O(\max(g_1(n), g_2(n)))$. ∎

A Theorem for Asymptotic Simplification: Example

- The significance of the preceeding theorem is that we need only capture the dominant cost when analyzing an algorithm.

- Example:

  If the algorithm is

  $$A(n) \;\; \{B(n); C(n); \}$$

  and the costs for $B(n)$ and $C(n)$ are respectively $O(n^3)$ and $O(n^2)$; then the cost of $A(n)$ is $O(n^3)$.

Using Limits for Comparing Orders of Growth

- THEOREM Let

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = l.$$

1. If $l = 0$ then $t(n) \in O(g(n))$.

   Implies that $t(n)$ has **smaller** order of growth than $g(n)$.
2. If $l > 0$ then $t(n) \in \Theta(g(n))$
   (which also means that $t(n) \in O(g(n))$ and $t(n) \in \Omega(g(n))$).

   Implies that $t(n)$ has **the same** order of growth than $g(n)$.
3. If $l = \infty$ then $t(n) \in \Omega(g(n))$.

   Implies that $t(n)$ has **larger** order of growth than $g(n)$.

L'Hôpital's Rule

- Given a function $f : \mathbb{N} \to \mathbb{R}$ with some formula $f(n)$, usually it can be easily extended to become function $f : \mathbb{R} \to \mathbb{R}$ by reusing the formula $f(x)$.
- The formula $f(x)$ allows the evaluation of its limit $f'(x)$ if it exists.
- Briefly, L'Hôpital's rule asserts

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}.$$

The rule is named after the 17th-century French mathematician Guillaume de l'Hôpital, who published the rule in his book *l'Analyse des Infiniment Petits pour l'Intelligence des Lignes Courbes* (literal translation: Analysis of the Infinitely Small to Understand Curved Lines) (1696).

L'Hôpital and Bernoulli

- Guillaume Francois Antoine, Marquis de L'Hôpital (1661 – 2 February 1704), French mathematician.
- Johann Bernoulli (27 July 1667 – 1 January 1748), Swiss mathematician.
- It is believed that L'Hôpital's rule was in fact discovered by the Swiss mathematician Johann Bernoulli. L'Hôpital hired Bernoulli and in the contract he retained the rights to all discoveries.

Basic Efficiency Classes: Constant and Log

- Constant: 1.

  Best case efficiency for some algorithms.
- Logarithm: $\log n$.

  Typically time efficiency of algorithms that reduce the problem size by a constant factor at each iteration.
- "n-log-n": $n \log n$.

  The time efficiency of many divide-and-conquer algorithms.

Polynomial

- Linear: n.

  Algorithms that scan the input.
- Quadratic: n^2.

  Example: The time efficiency of algorithms that have two nested loops.
- Cubic: n^3.

  Example: The time efficiency of algorithms having three nested loops.

Exponential

- Exponential: 2^n.

  Typically the time efficiency of algorithms that generate all subsets of an n-element set.

- Factorial (also called exponential): $n!$.

  Typically the time efficiency of algorithms that generate all permutations of an n-element set.

Q: For which applications do we want an algorithm to have high complexity?

Mathematical Analysis of Nonrecursive Algorithms

- Decide a parameter for the input size.
- Identify the basic operation. It is usually located in the deepest nested loop.
- If in addition to n (the input size), $C(n)$ also depends on some properties of an input I of size n (so we should have used $C(n, I)$), decide which of the worst-, average, best-case analysis is needed.
- Set up a sum giving $C(n)$.
- Using standard formulas and rules of sum manipulation, either find a closed formula or establish the efficiency class.

Analysis of Nonrecursive Algorithms: Example

- An algorithm solving the element uniqueness problem:

```
// The input array is $A[0..n-1]$
for $i$ from $0$ to $n-2$ do
    for $j$ from $i+1$ to $n-1$ do
        if $A[i] = A[j]$ then return false fi
    od
od
return true
```

- Naturally n is the input size and the comparison $A[i] = A[j]$ is the basic operation.

(Continued)

- The worst case happens if all the elements are unique or all the elements are unique except $A[n-2] = A[n-1]$.
- The summation

$$C_{\text{worst}}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

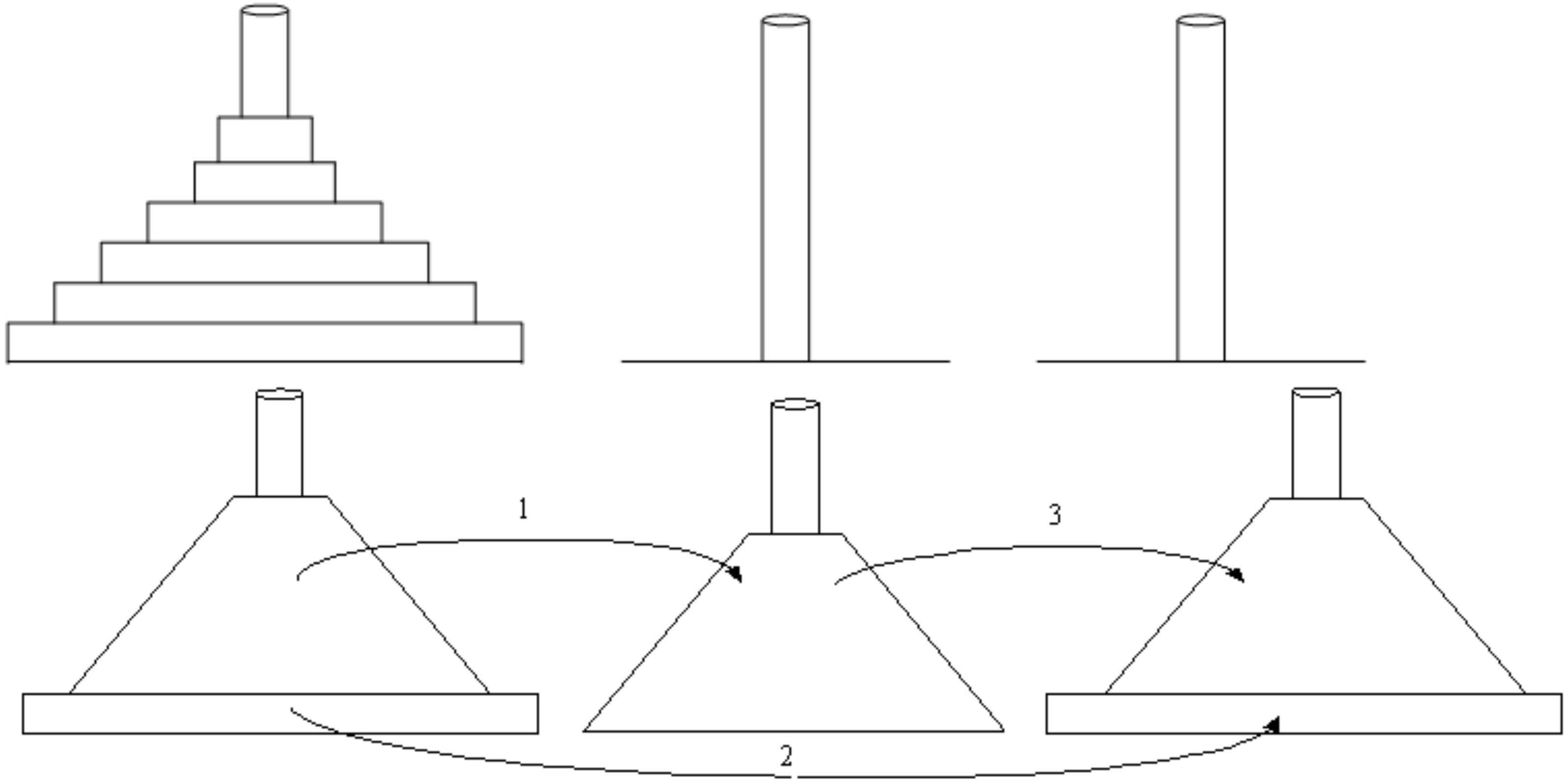
$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Mathematical Analysis of Recursive Algorithms

- Decide a parameter for the input size.
- Identify the basic operation.
- If in addition to n (the input size), $C(n)$ also depends on some properties of an input I of size n (so we should have used $C(n, I)$), decide which of the worst-, average, best-case analysis is needed.
- Set up a recurrence relation, with an appropriate initial condition, for $C(n)$.
- Either solve the recurrence or establish the efficiency class.

Tower of Hanoi



Size: n
Basic operation: move
Recurrence relation: $M(n) = 2M(n\text{-}1) + 1$ for $n>1$

Tower of Hanoi

- A recursive algorithm solving the Tower of Hanoi problem:

```
Move( $n, src, des$ ) {
    if $n = 1$ then
        move disk 1 from peg $src$ to peg $des$
    fi
    Move( $n-1, src, oth$ )
    move disk $n$ from peg $src$ to peg $des$
    Move( $n-1, oth, des$ )
}
```

- We take n, the number of disks, as the input size and “move” as the basic operation.

Analysis: Tower of Hanoi

- The recurrence is
$$C(n) = 2C(n-1) + 1$$
with the initial condition
$$C(1) = 1.$$

- Using the method of backward substitution, we have
$$\begin{aligned} C(n) &= 2C(n-1)+1 \\ &= 2(2C(n-2)+1)+1 \\ &= 2^2C(n-2)+2+1 \\ &= 2^{n-1}C(1)+2^{n-2}+\cdots+2+1 \\ &= 2^n-1. \end{aligned}$$

Example: Fibonacci Numbers

- Explicit formulas for the $F(n)$, the n^{th} Fibonacci Number.
- A recursive algorithm for Computing Fibonacci Numbers.
- An iterative algorithm for Computing Fibonacci Numbers.
- Closed form formulas for Computing Fibonacci Numbers.

- Leonardo Fibonacci, 1202, in book *Liber Abaci*.

The Fibonacci Sequence

- The Fibonacci sequence is

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \cdots$$

- The terms of the sequence can be defined by the recurrence

$$F(n) = F(n-1) + F(n-2), n \geq 2;$$

and the initial conditions

$$F(0) = 0, F(1) = 1.$$

An Explicit Term Formula

- The Fibonacci recurrence is a homogeneous second-order recurrence:

$$F(n) - F(n-1) - F(n-2) = 0.$$

- To solve the recurrence, we solve

$$x^2 - x - 1 = 0$$

for the roots

$$\Phi = \frac{1+\sqrt{5}}{2} = 1.61803, \hat{\Phi} = \frac{1-\sqrt{5}}{2} = -0.61803 = -\Phi^{-1}.$$

An Explicit Term Formula (2)

- By the theory of homogeneous recurrence, we have
$$F(n) = \alpha\Phi^n + \beta\hat{\Phi}^n.$$
- The values of α, β are solved using the initial conditions.
- Finally, we have
$$F(n) = \frac{1}{\sqrt{5}}(\Phi^n - \hat{\Phi}^n).$$
- Since for all $n \geq 0$, $-0.5 < \frac{\hat{\Phi}^n}{\sqrt{5}} < 0.5$, so
$$F(n) = \left[\frac{1}{\sqrt{5}}\Phi^n\right]$$
where $[x]$ means the integer nearest to x.

Computing $F(n)$ Recursively

- An obvious recursive algorithm to compute $F(n)$ is

```
Fib($n$) { if ($n \le 1$) then return $n$;
                else return Fib($n-1$)+Fib($n-2$); fi }
```

- Let $C(n)$ be the number of arithmetic operations to compute $F(n)$.
- Clearly,

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

  with the initial conditions $C(0) = C(1) = 0$.
- Let $D(n) = C(n) + 1$.

Computing $F(n)$ Recursively (2)

- We arrive at the recurrence
$$D(n) = D(n-1) + D(n-2)$$
and the initial conditions are $D(0) = D(1) = 1$.
- Thus we see that
$$D(n) = F(n+1) - 1 = \frac{1}{\sqrt{5}}(\Phi^{n+1} - \hat{\Phi}^{n+1}) - 1.$$
- Thus the algorithm is exponential in n.
- The algorithm is very inefficient. There are many repetitive computations. For example, $F(5)$ computes $F(2)$ three times.

Computing $F(n)$ Iteratively

- An obvious iterative algorithm to compute $F(n), n \geq 2$, is

```
Fib($n$) {
    $a \leftarrow 0$; $b \leftarrow 1$
    for $i$ from $2$ to $n$ do
        $c \leftarrow a + b$; $a \leftarrow b$; $b \leftarrow c$
    od
    return $c$
}
```

- Let $C(n)$ be the number of arithmetic operations to compute $F(n)$.
- Clearly,

$$C(n) = n - 1.$$

Computing $F(n)$ by Closed Form Formula

- Evaluating the power of a real number and rounding:

$$F(n) = \left[\frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n \right].$$

- Evaluating the power of a 2×2 matrix for $n \geq 1$:

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n.$$

Empirical Analysis of Algorithms

- Set goals for the experiment.
- Decide the efficiency metric M to be measured: basic operation counting or actual time clocking.
- Design inputs based on desired characteristics.
- Implement the algorithm as a program.
- Run the program on the inputs and record results.
- Analyze the results obtained.

Algorithm Visualization

- Algorithm visualization uses graphical images to convey useful information about an algorithm.
- How the data and operations can be represented graphically requires much design effort.

Take Away Message

- Consider input: Worst-, average-, and best-case efficiency.
- Asymtotic notations: $O, \;\; \Omega, \;\; \Theta$.
- Efficiency classes: $1, \;\; \log n, \;\; n, \;\; n \log n, \;\; n^2, \;\; n^3, \;\; 2^n, \;\; n!$.
- Mathematical analysis:
  - Non-recursive algorithms $\Rightarrow$ use sum manipulation.
  - Recursive algorithms $\Rightarrow$ use recurrence relation.