Analysis of Algorithms

What to analyze

[Lewis/Denenberg 2.1, Goodrich/Tamassia 3.5]

- correctness
- termination
- efficiency

Time efficiency

[Lewis/Denenberg 2.2, Goodrich/Tamassia 3.6+3.7]

- growth rate
- worst case, expected case, amortized
- analysis techniques for iterative algorithms
- analysis techniques for recursive algorithms

Mathematical background

[Lewis/Denenberg 1.3 (except of pp. 26-32); Goodrich/Tamassia 3.3]

Correctness

"An algorithm must not give the ***wrong*** *answer."* [Lewis/Denenberg]

A function *fact* for computing *factorial* must not return 6 for the call $fact(2)$.

Which answers are wrong?

- the user knows that, or
- a *specification* of legal inputs and corresponding correct answers is needed.

An algorithm is *correct* iff for any legal input

- the computation *terminates*, and
- the answer is *as specified*.

Termination (1)

An algorithm should

- produce an answer in a finite number of steps
- for any legal input

Example:

Algorithm for squaring an integer using $n^2 = (n-1)^2 + 2n - 1 \ \forall n \in \mathbb{N}$

$$
\begin{array}{l}
\textbf{function } \mathit{Square}(\textbf{ integer } n) : \textbf{integer} \\
\quad \textbf{if } n = 0 \textbf{ return } 0 \\
\quad \textbf{if } n \neq 0 \textbf{ return } \mathit{Square}(n-1) + 2 \cdot (n-1) + 1
\end{array}
$$

does not terminate for $n < 0$.

Termination (2)

Termination is a difficult problem:

function $OddEven($ **integer** $m)$: **integer** [Lewis/Denenberg, Algorithm 2.1]

$n \leftarrow m$

while $n > 1$ **do**

  if n is even **then**

    $n \leftarrow n/2$

  else

    $n \leftarrow 3n+1$

return m

Does this algorithm compute the identity function for all $m \geq 1$?

Efficiency

Different algorithms may solve the same problem.

How to compare them?

- Resources used by an algorithm:
  - memory
  - **time**
- Analysis of time efficiency should be:
  - machine-independent
  - valid for all legal data
- We compare:
  - time **growth-rate** for growing size of (input) data (scalability)
  - mostly for **worst-case** problem instances

Efficiency (2)

function *TableSearch*(**table**<**key**> $T[0..n-1]$, **key** K) : **integer**

(1) **for** i **from** 0 **to** $n-1$ **do**

(2) **if** $T[i] = K$ **then return** i

(3) **if** $T[i] > K$ **then return** -1

(4) **return** -1

What is the worst-case problem instance?

Worst case time:

$$n \cdot (t_1 + t_2 + t_3) + t_4$$

Efficiency (3)

function $BinSearch($ **table** $T[0..n-1]$, **key** $K)$: **integer**

(0) **if** $n \leq 0$ **then return** -1

(1) $l \leftarrow 0; \quad u \leftarrow n-1$

(2) **while** $l < u$ **do**

(3) $\quad mid \leftarrow \lfloor (l+u)/2 \rfloor$

(4) $\quad$ **if** $K = T[mid]$ **then return** mid

(5) $\quad$ **if** $K < T[mid]$ **then** $u \leftarrow mid - 1$ **else** $l \leftarrow mid + 1$

(6) **if** $K = T[l]$ **then return** l **else return** -1

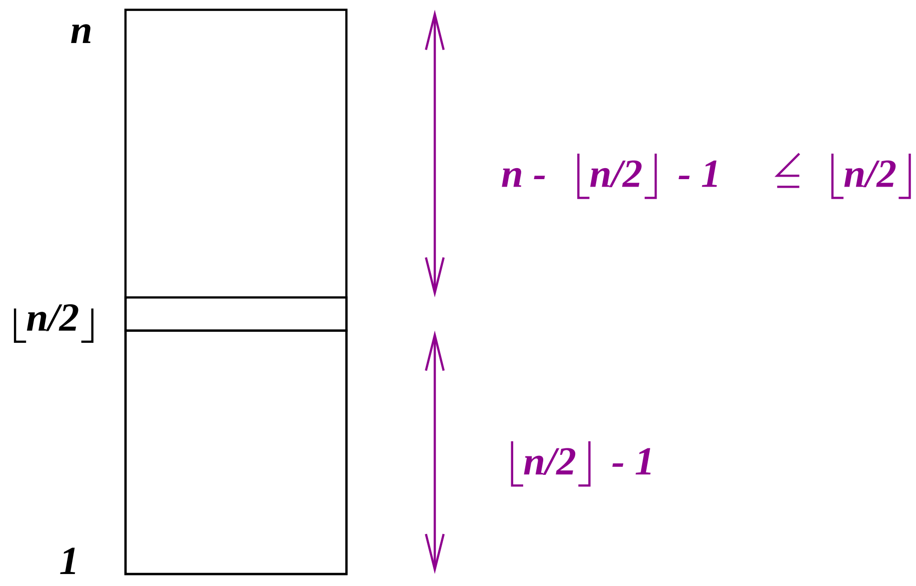
Worst case time: $t_0 + t_1 + maxit \cdot (t_2 + t_3 + t_4 + t_5) + t_6$

where $maxit$ = maximal number of iterations of the **while** loop

Efficiency (4)

How to compute *maxit* for $n = 1, 2, ...$?

$$maxit(1) = 0,$$
$$maxit(2) = 1,$$
$$maxit(3) = 1,$$
$$maxit(4) = 2,$$
$$maxit(5) = 2,$$
$$maxit(6) = 2,$$
$$\ldots$$



$$maxit(n) = 1 + maxit(\lfloor n/2 \rfloor)$$

$$maxit(n) = \lfloor \log_2 n \rfloor$$

Estimating execution time for iterative programs

Elementary operation

takes / can be bound by a constant time

Sequence of operations

takes the sum of the times of its components

Loop (**for...** and **while...**)

the time of the body multiplied by number of repetitions (in the worst case)

Conditional statement (**if...then...else...**)

the time for evaluating and checking the condition

plus maximum of the times for **then** and **else** parts.

Example: Independent Nested Loops

Matrix-vector product (here, for a quadratic matrix)

given: vector $\vec{x} \in \mathbb{R}^n$, matrix $A \in \mathbb{R}^{n,n}$, with $n > 0$

compute: vector $\vec{y} \in \mathbb{R}^n$ with

$$\vec{y} = A \cdot \vec{x}, \qquad \text{that is,} \qquad y_i = \sum_{j=1}^{n} a_{ij} x_j, \qquad i = 1, ..., n$$

procedure $matvec($**array**<**real**> $x[1..n], A[1..n, 1..n]$) : **array**<**real**> $y[1:n]$

(1) **for** i **from** 1 **to** n **do**

(2) $\quad y[i] \leftarrow 0.0$

(3) $\quad$ **for** j **from** 1 **to** n **do**

(4) $\qquad y[i] \leftarrow y[i] + A[i,j] * x[j]$

return y

Time: $n(t_1 + t_2) + n^2(t_3 + t_4)$

Example: Dependent Nested Loops

Prefix-Sums

given: Vector $\vec{x} \in \mathbb{N}^n$,

compute: “Prefix-sums” vector $\vec{y} \in \mathbb{N}^n$ with $$y_i = \sum_{j=1}^{i} x_j, \qquad i = 1, \dots, n$$

A straightforward algorithm follows directly from the definition:

procedure $prefixsum($**array**<**integer**> $x[1..n]$ $)$: **array**<**integer**> $y[1:n]$

(1) **for** i **from** 1 **to** n **do**

(2) $\quad y[i] \leftarrow 0.0$

(3) $\quad$ **for** j **from** 1 **to** i **do**

(4) $\qquad y[i] \leftarrow y[i] + x[j]$

$\quad$ **return** y

Total time: $$t(n) = n(t_1 + t_2) + (1 + 2 + \dots + (n-1) + n)(t_3 + t_4)$$ $$= n(t_1 + t_2) + \frac{n(n+1)}{2}(t_3 + t_4)$$

Remark: There exists a better, linear-time algorithm!

Principles of Algorithm Analysis

An algorithm should work for (input) data of any size.

(Example *TableSearch*: input size is the size of the table.)

Show the resource (time/memory) used as

an *increasing function* of *input size*.

Focus on the *worst case* performance.

Ignore *constant factors*

analysis should be machine-independent;

more powerful computers introduce speed-up by constant factors.

Study *scalability / asymptotic behaviour* for large problem sizes:

ignore lower-order terms, focus on dominating terms.

Commonly used increasing functions

Let x, y, a, b, α be real numbers.

Logarithm to the base $b > 0$ of $x > 0$

$y = \log_b x$ iff $b^y = x$

We consider only cases where $a, b > 1$.

Changing base – multiplication by a *constant* factor:

$\log_b x = \log_b (a^{\log_a x}) = \log_a x \ \log_b a$

Power function of x

x^α where $\alpha > 0$, such as x, $x^{1/2}$, x^2, ...

Exponential function of x

c^x for some $c > 1$

Combinations of these, e.g. $x \log_2 x$

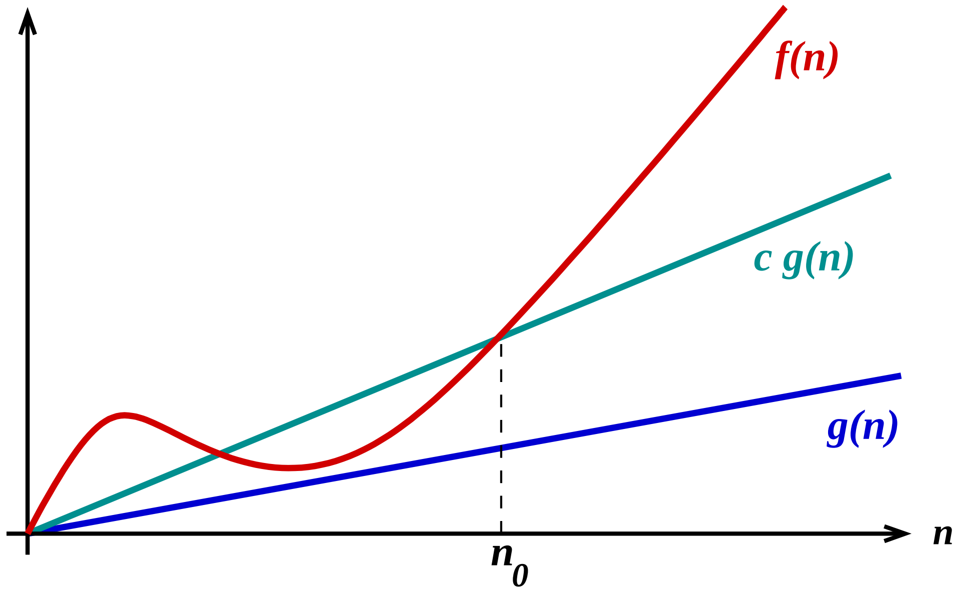
How functions grow

n	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
2	1	2	2	4	4
16	4	16	64	256	$6.5 \cdot 10^4$
64	6	64	384	4096	$1.84 \cdot 10^{19}$

$1.84 \cdot 10^{19} \mu$sec = $2.14 \cdot 10^8$ days = 5845 centuries

Asymptotic analysis: Dominance relation

Consider two growing functions f, g from natural numbers to positive real numbers:



f dominates g iff $f(n)/g(n)$ increases without bounds for $n \to \infty$

that is, for a given constant factor $c > 0$,

there is some threshold value $n_0 \in \mathbb{N}$

such that $f(n) > c \cdot g(n)$ for all $n > n_0$. (Ex.: $f(n) = n^2$ dominates $g(n) = 7n$.)

Asymptotic analysis: Order Notation (1)

Motivation:

+ comparing growth rates of increasing functions

+ estimating efficiency of algorithms by reference to simple functions

+ abstraction from constant factors $\rightarrow$ classes of functions

Asymptotic analysis: Order Notation (2)

f, g growing functions from natural numbers to positive real numbers

f **is (in)** $O(g)$ iff there exist $c > 0$, $n_0 \geq 0$ such that

$$f(n) \leq c\, g(n) \quad \text{for all } n > n_0$$

Intuition: Apart from constant factors, f grows at most as quickly as g

f **is (in)** $\Omega(g)$ iff there exist $c > 0$, $n_0 \geq 0$ such that

$$f(n) \geq c\, g(n) \quad \text{for all } n > n_0$$

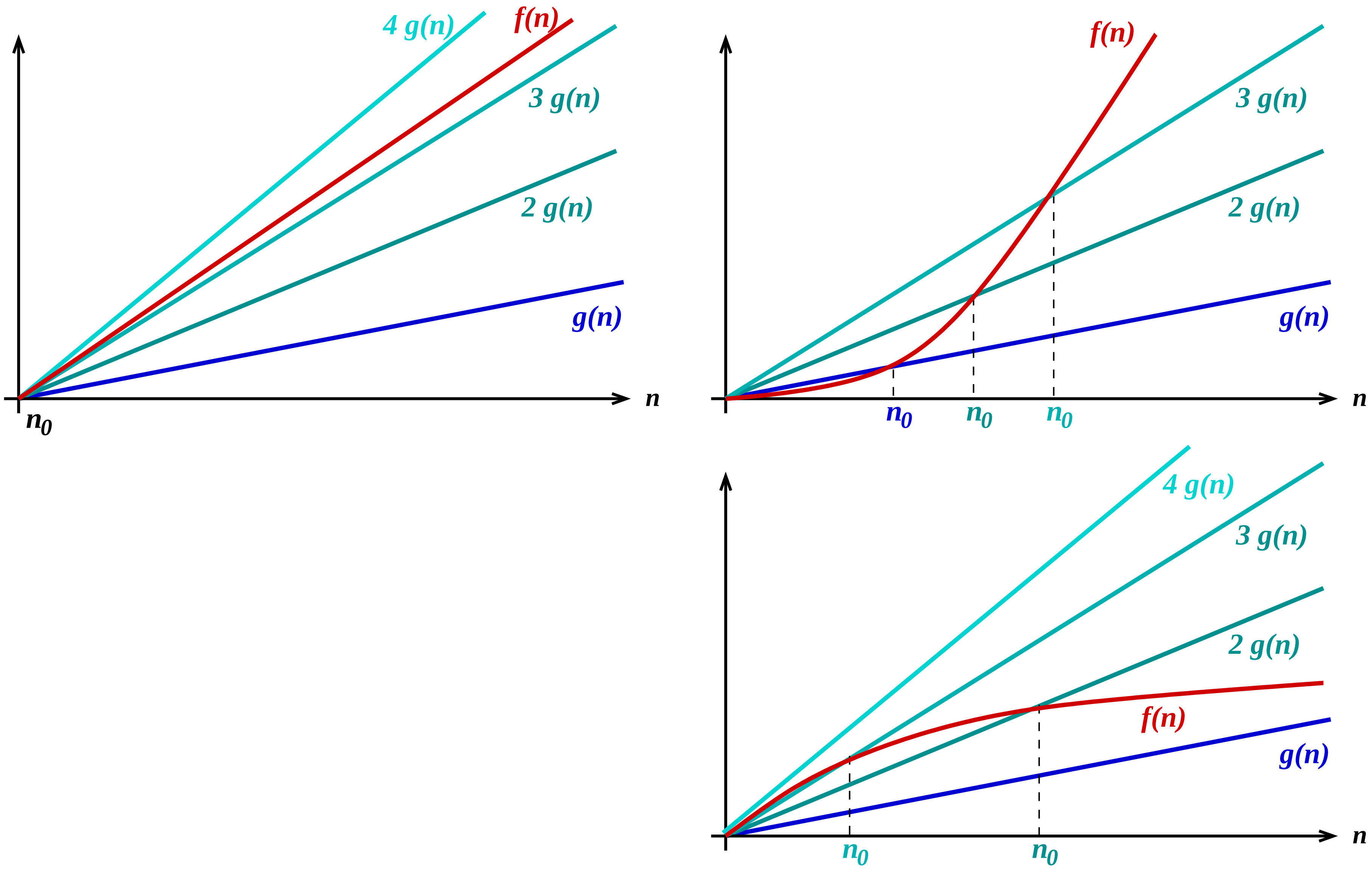
Intuition: Apart from constant factors, f grows at least as quickly as g

$\Omega()$ is the converse of O, i.e. f **is in** $\Omega(g)$ iff g **is in** $O(f)$

f **is (in)** $\Theta(g)$ iff $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$

Intuition: Apart from constant factors, f grows exactly as quickly as g

Asymptotic analysis: Order Notation (3)



Asymptotic analysis: Examples of Order Notation

$$t_{TableSearch}(n) = n * (t_1 + t_2 + t_3) + t_4 = k_1 \cdot n + k_2$$

hence:

$t_{TableSearch}(n) \in O(n)$ (Why?)

$t_{TableSearch}(n) \in \Omega(n)$ (Why?)

$t_{TableSearch}(n) \notin O(\log\ n)$ (Why?)

$t_{TableSearch}(n) \in \Theta(n)$ (Why?)

$$t_{BinSearch}(n) = c_1 \cdot (\lfloor \log_2(n) \rfloor) + c_2$$

hence:

$t_{BinSearch}(n) \in O(\log\ n)$

$t_{BinSearch}(n) \in O(n)$

Asymptotic analysis: Dominance Relation Revisited

Growing functions on natural numbers: f and g

f **is (in) o(g)**, i.e., f *is dominated by* g
iff for **any** $c > 0$ there is an $n_0 > 0$ such that $g(n) > cf(n)$ for all $n > n_0$

Intuition: g grows *more quickly* than f.

- If $f \in o(g)$ then $f \in O(g)$ but not vice versa.

Example: $n \in o(n^2)$

Asymptotic analysis: Comparing Growth Rates of Simple Functions

Some simple facts:

- $n^{\alpha} \in O(n^{\beta})$ iff $\alpha \leq \beta$ $(\alpha, \beta > 0)$ $[n^{\alpha} \in o(n^{\beta})$ iff $\alpha < \beta]$
  growth rate of power function is determined by the value of power
- $\log_b n \in o(n^{\alpha})$ for any $b, \alpha > 0$
  power functions grow more quickly than logarithms
- $n^{\alpha} \in o(c^n)$ for any $\alpha > 0$, $c > 1$
  exponential functions grow more quickly than power functions
- $\log_a n \in O(\log_b n)$ for any a and b
  growth rate of logarithms of various bases is equal
- $c^n \in O(d^n)$ iff $c \leq d$, $[c^n \in o(d^n)$ iff $c < d]$
- Any constant function $f(n) = c$ is in $O(1)$
  there is no difference in growth rate of constant functions

Asymptotic analysis: Checking Growth Rates

If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$

transitivity

If $f \in O(g)$ then also $f + g \in O(g)$

growth rate depends only on fastest growing components

If $f \in O(f')$ and $g \in O(g')$ then $f \cdot g \in O(f' \cdot g')$

If there are $d, n_0 > 0$ such that $f(n) \geq d$ for all $n \geq n_0$
then $k \cdot f(n) + c \in O(f)$ for all constants k, c.

Prove or disprove

$$(n+1)^2 \in O(n^3)$$

$$(n-1)^3 \in O(n^2)$$

$$3^{n-1} \in O(2^n)$$

$$\sqrt{n^5} \in O(n^2)$$

more...

Asymptotic analysis: Comparing functions

To check $f \in O(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$, analyze

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

- $f \in O(g)$ iff $l < \infty$
- $f \in \Omega(g)$ iff $l > 0$
- $f \in \Theta(g)$ iff $0 < l < \infty$

Analysis of Recursive Programs (1)

function $fact($ **integer** $n)$: **integer**

  if $n = 0$ **then return** 1

  else return $n \cdot fact(n-1)$

Execution time:

time for comparison: t_c
time for multiplication: t_m
time for call and return neglected

Total execution time $T(n)$

$$T(0) = t_c$$

$$T(n) = t_c + t_m + T(n-1), \text{ if } n > 0$$

(T is defined by a *recurrence relation*)

Hence for $n > 0$:

$$\begin{aligned} T(n) &= (t_c + t_m) + (t_c + t_m) + T(n-2) \\ &= (t_c + t_m) + (t_c + t_m) + (t_c + t_m) + T(n-3) = \ldots \\ &= \underbrace{(t_c + t_m) + \ldots + (t_c + t_m)}_{n \text{ times}} + t_c \\ &= n \cdot (t_c + t_m) + t_c \in O(n) \end{aligned}$$

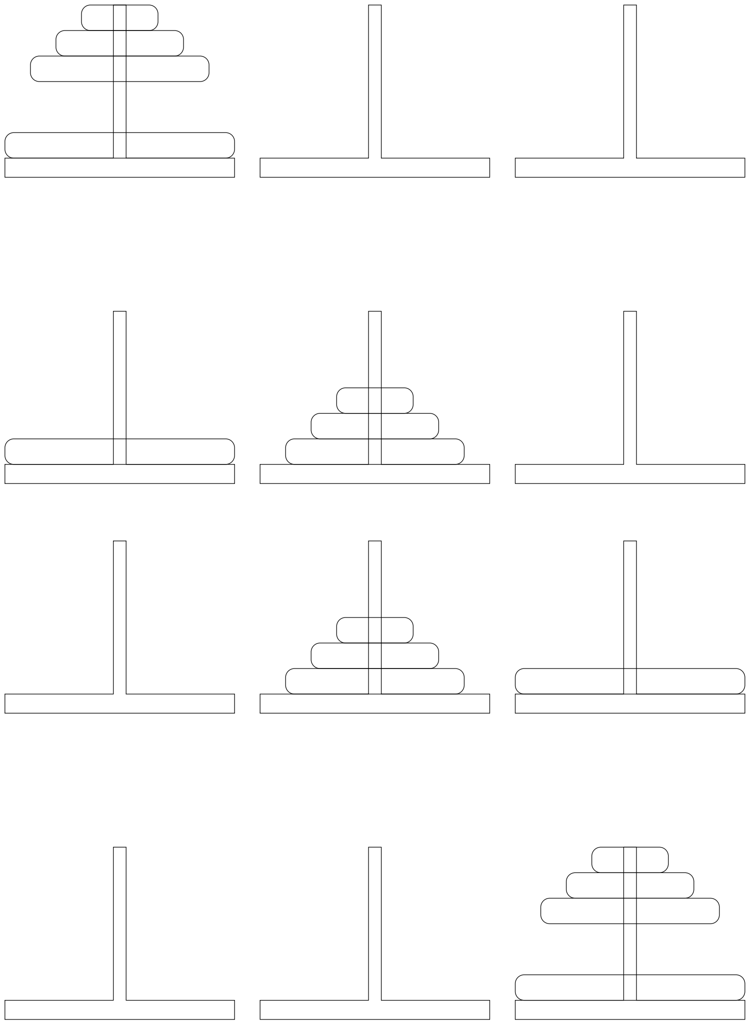
Analysis of Recursive Programs (2)

- Characterize execution time by a recurrence relation
- Find solution (closed form, non-recursive) of the recurrence relation

  If not listed in a textbook, you may:

  1. Unroll the recurrence relation a few times to get a hypothesis for a possible solution: $T(n) = \ldots$
  2. Prove the hypothesis for $T(n)$ by induction. If that fails, modify the hypothesis and try again ...

Towers of Hanoi



Hanoi

procedure $Hanoi($**integer** n, **char** $X, Y, Z)$:

{ move n topmost slices from tower X to tower Z, using Y as temporary }

if $n = 1$ **then** $output$(“move X to Z”)

else

$Hanoi(n-1, X, Z, Y)$

$output$(“move X to Z”)

$Hanoi(n-1, Y, X, Z)$

return

$T(1) = t_o$

$T(n) = 2T(n-1) + t_o$

$T(n) = 4T(n-2) + 3t_o = 8T(n-3) + 7t_o = 2^n T(1) + (2^n - 1)t_o \in O(2^n)$

Average Case Analysis (1)

Reconsider *TableSearch*(): sequential search through a table

Input argument:

one of the table elements,

assume it is chosen with *equal probability* for all elements.

function $TableSearch($ **table**<**key**> $T[0..n-1]$, **key** K) : **integer**

for i **from** 0 **to** $n-1$ **do**

if $T[i] = K$ **then return** i

Expected search time:

$$\frac{1+2+3+\ldots+n}{n}\, t_c \;=\; \frac{n(n+1)}{2n}\, t_c \;\in\; O(n)$$

Average Case Analysis (2)

- We *have to* know the probability distribution for the input data
- Gives no information about the worst cases
- Often difficult to analyze

Amortized Analysis

Done for *sequences* of operations and input data.

Example:

Given: a sorted table $T[0..n-1]$

Input: a permutation $e_1, ..., e_n$ of all elements of T

The total time for linear search of *all* elements is

$$\frac{n(n+1)}{2} t_c$$

Guaranteed!