3.5 RECURSIVE ALGORITHMS

REVIEW : An ***algorithm*** is a computational representation of a function.

Remark: Although it is often easier to write a correct recursive algorithm for a function, iterative implementations typically run faster, because they avoid calling the stack.

RECURSIVELY DEFINED ARITHMETIC

Example 3.5.1: recursive addition of natural numbers: $succ$ = successor, $pred$ = predecessor.

Algorithm 3.5.1: recursive addition

recursive function: $\mathrm{sum}(m, n)$
Input: integers $m \geq 0, n \geq 0$
Output: $m + n$

If $n = 0$ **then return** (m)
    else return $\big(\mathrm{sum}(succ(m), pred(n))\big)$

Example 3.5.2: iterative addition of natural numbers

Algorithm 3.5.2: iterative addition

function: sum(m, n)
Input: integers $m \geq 0, n \geq 0$
Output: $m + n$

While $n > 0$ **do**
    $m := succ(m)$;
    $n := pred(n)$;
    endwhile

Return (m)

Example 3.5.3: proper subtraction of natural numbers: $succ =$ successor, $pred =$ predecessor.

Algorithm 3.5.3: proper subtraction

recursive function: $\mathrm{diff}(m, n)$
Input: integers $0 \leq n \leq m$
Output: $m - n$

If $n = 0$ **then return** (m)
else return $\big(\mathrm{diff}(pred(m), pred(n))\big)$

Example 3.5.4: natural multiplication:

Algorithm 3.5.4: natural multiplication

recursive function: $\mathrm{prod}(m, n)$
Input: integers $0 \leq n \leq m$
Output: $m \times n$

If $n = 0$ **then return** (0)
else return $\big(\mathrm{prod}(m, pred(n)) + m\big)$

Example 3.5.5: factorial function:

Algorithm 3.5.5: factorial

recursive function: factorial(n)
Input: integer $n \geq 0$
Output: $n!$

If $n = 0$ **then return** (1)
  else return $\big(\text{prod}(n, \text{factorial}(n-1))\big)$

NOTATION: Hereafter, we mostly use the infix notations +, -, *, and ! to mean the functions sum, diff, prod, and factorial, respectively.

RECURSIVELY DEFINED RELATIONS

DEF: The ***(Iverson) truth function*** **true** assigns to an assertion the boolean value TRUE if true and FALSE otherwise.

Example 3.5.6: order relation:

Algorithm 3.5.6: order relation

recursive function: $\text{ge}(m, n)$
Input: integers $m, n \geq 0$
Output: **true** $(m \geq n)$

If $n = 0$ **then return** (TRUE)
  elseif $m = 0$ **then return** (FALSE)
  else return $\big(\text{ge}(m-1, n-1)\big)$

Time-Complexity: $\Theta(\min(m, n))$.

OTHER RECURSIVELY DEFINED FUNCTIONS

REVIEW EUCLIDEAN ALGORITHM:

Algorithm 3.5.7: Euclidean algorithm

recursive function: $\gcd(m, n)$
Input: integers $m > 0, n \geq 0$
Output: $\gcd\ (m, n)$

If $n = 0$ **then return** (m)
else return $\big(\gcd(n, m \bmod n)\big)$

Time-Complexity: $\mathcal{O}(\ln n)$.

Example 3.5.7: Iterative calc of gcd (289, 255)

$m_1 = 289 \quad n_1 = 255 \quad r_1 = 34$
$m_2 = 255 \quad n_2 = 34 \quad r_2 = 17$
$m_3 = 34 \quad n_3 = 17 \quad r_3 = 0$

DEF: The execution of a function exhibits ***exponential recursive descent*** if a call at one level can generate multiple calls at the next level.

Example 3.5.8: Fibonacci function
$f_0 = 1, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$
$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$

Time-Complexity: $\Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$.

Algorithm 3.5.8: Fibonacci function

iterative speedup function: fibo(n)
Input: integer $n \geq 0$
Output: fibo (n)

If $n = 0 \vee n = 1$ **then return** (1)
else $f_{n-2} := 1; f_{n-1} := 1;$
  for $j := 2$ **to** n **step** 1
    $f_n := f_{n-1} + f_{n-2};$
    $f_{n-2} := f_{n-1};$
    $f_{n-1} := f_n;$ **endfor**
return (f_n)

Time-Complexity: $\Theta(n)$.

RECURSIVE STRING OPERATIONS

$\Sigma =$ set, $c \in \Sigma$ (object), $s \in \Sigma^*$ (string)

$\Sigma^0 = \Lambda = \{\lambda\}$ strings of length 0

$\Sigma^n = \Sigma^{n-1} \times \Sigma$ strings of length n

$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots$ all finite non-empty strings

$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$ all finite strings

These three ***primitive string functions*** are all defined and implemented nonrecursively for arbitrary sequences, not just strings of characters.

DEF: ***appending a character*** to a string

append : $\Sigma^* \times \Sigma \rightarrow \Sigma^*$ non-recursive

$(a_1 a_2 \cdots a_n, c) \mapsto a_1 a_2 \cdots a_n c$

DEF: ***first character*** of a non-empty string

first : $\Sigma^+ \rightarrow \Sigma$ non-recursive

$a_1 a_2 \cdots a_n \mapsto a_1$

DEF: ***trailer*** of a non-empty string

trailer : $\Sigma^n \rightarrow \Sigma^{n-1}$

$a_1 a_2 \cdots a_n \mapsto a_2 \cdots a_n$

These four ***secondary string functions*** are all defined and implemented recursively.

DEF: ***length*** of a string

length : $\Sigma^* \rightarrow \mathcal{N}$

$$\begin{aligned} \text{length}(s) &= 0 \quad \text{if } s = \lambda \\ &= 1 + \text{length}(\text{trailer}(s)) \text{ if } s \neq \lambda \end{aligned}$$

DEF: ***concatenation*** of two strings

concat : $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

$$\begin{aligned} (s \circ t) &= s \quad \text{if } t = \lambda \\ &= \text{append}(s, \text{first}(t)) \circ \text{trailer}(t) \text{ if } s \neq \lambda \end{aligned}$$

NOTATION: It is customary to **overload** the concatenation operator $\circ$ so that it also appends.

DEF: ***reversing*** a string

reverse : $\Sigma^* \rightarrow \Sigma^*$

$$\begin{aligned} s^{-1} &= s \quad \text{if } s = \lambda \\ &= \text{trailer}(s)^{-1} \circ \text{first}(s) \text{ if } s \neq \lambda \end{aligned}$$

DEF: ***last character*** of a non-empty string

last: $\Sigma^+ \rightarrow \Sigma$

$$\text{last}(s) = \text{first}(s^{-1})$$

RECURSIVE ARRAY OPERATIONS

Algorithm 3.5.9: location

recursive function: location$(x, A[\ ])$
Input: target value x, sorted array $A[\ ]$
Output: 0 if $x \notin A$; $\min\{j \mid x = A[j]\}$ if $x \in A$

If length $(A) = 1$ **then**
  return (**true** $(x = A[1])$)
elseif $x \leq \text{midval}(A)$
  return location$(x, \text{fronthalf}(A))$
else
  return location$(x, \text{backhalf}(A))$

function: midindex(A)
Input: array $A[\ ]$
Output: middle location of array A
$\text{midindex}(A) = \lfloor \textbf{length}(A)/2 \rfloor$

function: midval(A)
Input: array $A[\ ]$
Output: value at middle location of array A
$\text{midval}(A) = A[\text{midindex}(A)]$

continued on next page

Algorithm 3.5.9: location, continuation

function: fronthalf(A)
Input: array $A[\;]$
Output: front half-array of array A
$\text{fronthalf}(A) = A[1 \ldots \text{midindex}(A)]$

function: backhalf(A)
Input: array $A[\;]$
Output: back half-array of array A
$\text{fronthalf}(A) = A[\text{midindex}(A) + 1 \ldots \text{length}(A)]$

Time-Complexity: $\Theta(\log n)$.

Algorithm 3.5.10: verify ascending order

recursive function: ascending$(A[\;])$
Input: array $A[\;]$
Output: TRUE if ascending; FALSE if not

if length $(A[\;]) \leq 1$ **then**
  return $(TRUE)$
else
  return $(a_1 \leq a_2 \wedge \text{ascending}(\text{trailer}(A[\;])))$

Time-Complexity: $\Theta(n)$.

Algorithm 3.5.11: merge sequences

recursive function: $\mathrm{merge}(s, t)$
Input: ascending sequences s, t
Output: merged ascending sequence

If length $(s) = 0$ **then**
  return t
elseif $s_1 \leq t_1$
  return $\left(\mathrm{first}(s) \circ \mathrm{merge}(\mathrm{trailer}(s), t)\right)$
else
  return $\left(\mathrm{first}(t) \circ \mathrm{merge}(s, \mathrm{trailer}(t))\right)$

Algorithm 3.5.12: mergesort

recursive function: $\mathrm{msort}(A)$
Input: ascending array $A[\;]$
Output: merged ascending array

If length $(A[\;]) \leq 1$ **then** **return** $(A[\;])$
else **return**

$$\Big(\mathrm{merge}(\mathrm{msort}(\mathrm{fronthalf}(A)), \mathrm{msort}(\mathrm{backhalf}(A))\Big)$$