Recursive procedures

1. What is a recursive procedure?
2. Recursive versus iterative
3. An application: binary search

1. What is a recursive procedure?

A recursive procedure calls itself.

A recurrence function is defined using itself.

General form of simple recurrence function *f*():

$$f(\text{argument}) = \begin{cases} \text{base case rule, where } f \text{ does not appear} & \text{when argument is 'small'} \\ \text{recurrence case rule, where } f \text{ appears} & \text{when argument is not 'small'} \end{cases}$$

Example: computing power of a number

Recurrence function for $f(x,n) = x^n$, when n is pos integer?

Basic idea: $x^n = x\,(x^{n-1})$.

$$f(x,n) = \begin{cases} 1 & \text{when } n = 0 \\ x \cdot f(x, n-1) & \text{when } n > 0 \end{cases}$$

□

Q: When can we use a recursive procedure?

A: When the problem we want to solve can be expressed in terms of a smaller version of itself.

Example: computing power of a number (contd)

$$x^n = x \cdot (x^{n-1})$$

$$\text{problem} = x \cdot (\text{smaller version of problem})$$

$\square$

Features of recursive procedure:

- base case(s) or trivial case(s): procedure does not call itself
- recursion case(s): procedure calls itself
- recursion level
- each call has own parameters, own local variables
- call stack (what calls are still incomplete)

Example: computing power of a number (contd)

```
POWR(x, n)
input n is positive integer, x is some number output: x^n
/* We compute x^n recursively. */
if n == 0 then
    return 1
else
    z = POWR(x, n − 1)
    return x · z
end
```

Compute POWR(2,3).

step	recursion level	code line(s)	computation	stack
1	1	7	z = POWR(2, 2)	POWR(2,3)
2	2	7	z = POWR(2,1)	POWR(2,2) POWR(2,3)
3	3	7	z = POWR(2,0)	POWR(2,1) POWR(2,2) POWR(2,3)
4	4	5	return 1	POWR(2,0) POWR(2,1) POWR(2,2) POWR(2,3)
5	3	7,8	z = 1 return 2	POWR(2,1) POWR(2,2) POWR(2,3)
6	2	7,8	z = 2 return 4	POWR(2,2) POWR(2,3)
7	1	7,8	z = 4 return 8	POWR(2,3)

□

2. Recursive versus iterative

Usually a recursive procedure can be converted to an equivalent iterative procedure.

Alternative to a recursive procedure: *iterative* (*looping*) procedure: does not call itself

Correspondences:

recursive procedure	iterative procedure
recursion level	iteration
call stack	own stack (accumulator variables)
base case condition	iteration termination condition
recursion case(s)	in-loop computations

Example: computing power of a number (contd)

```
POWR(x,n)
input n is positive integer, x is some number output: x^n
/* We compute x^n recursively. */
if n == 0 then
    return 1
else
    z = POWR(x, n-1)
    return x·z
end
```

```
POWI(x,n)
input int n, some number x   output: y
/* We compute x^n iteratively for positive integer n. */
y = 1
while n > 0
  y = x·y, n = n-1
end
return y
```

Why prefer recursive procedure over iterative procedure?

- often recursive is more compact
- no need to handle stack
- often easier to write/understand

Why prefer iterative procedure over recursive procedure?

- often has better performance for some programming languages
- allows programmer better control over data structures

Ongoing debate: try searching 'why recursive is better than iterative'.

3. An application: binary search

Problem: find a value *key* in an sorted array *A*[1..*n*]

A[1]	A[2]	A[3]	. . .	A[n-1]	A[n]

Note: $A[i] \leq A[i+1]$

Idea: split array in half

A[1]	A[2]	. . .	A[mid]	A[mid+1]	. . .	A[n-1]	A[n]

left half   right half

Description

1. We start with array $A[1..n]$ of numbers in sorted order from smallest to largest. We want to find location of key in $A[1..n]$, if it occurs.

2. The range of A we are looking in is $A[L..R]$. We compute an index roughly halfway between L and R: call it mid. If $key \leq A[mid]$, then we continue searching in $A[L..mid]$. If $key > A[mid]$, then we continue searching in $A[(mid+1)..R]$.

3. We keep halving the range we are searching in until either, we find the location of key or key does not occur in array.

```
BINSEARCH(A, L, R, key)
input A[1..n] is an array containing numbers; L and R are the
leftmost and rightmost indices that concern us; key is the value
we want to find
output: the index at which key occurs A or −1
/* The numbers in A must be in order from smallest to largest.
We search in array A[L..R] for key. If key is found, we return
its location, otherwise we return −1.*/
if L == R then
   if A[L] == key then
      return L
   else
      return −1
   end
else
   mid = ⌊(L + R)/2⌋
   if key ≤ A[mid] then
      return BINSEARCH(A, L, mid, key)
   else
      return BINSEARCH(A, mid + 1, R, key)
   end
end
```

Example

Compute BINSEARCH(A,1, 7, 12) where

A =

-21	-20	-18	-6	5	12	13

step	recursion level	code line(s)	computation	array A ↑ = mid ↑ = L ↑ = R
—	—	—	L = 1 R = 7	-21 -20 -18 -6 5 12 13
1	1	16	mid = $\lfloor 8/2 \rfloor$ = 4	-21 -20 -18 -6 5 12 13
		20	return BINSEARCH(A, 5, 7, 12)	
2	2	16	mid = $\lfloor 12/2 \rfloor$ = 6	-21 -20 -18 -6 5 12 13
		18	return BINSEARCH(A, 5, 6, 12)	
3	3	16	mid = $\lfloor 11/2 \rfloor$ = 5	-21 -20 -18 -6 5 12 13
		20	return BINSEARCH(A, 6, 6, 12)	
4	4	11	return 6	-21 -20 -18 -6 5 12 13