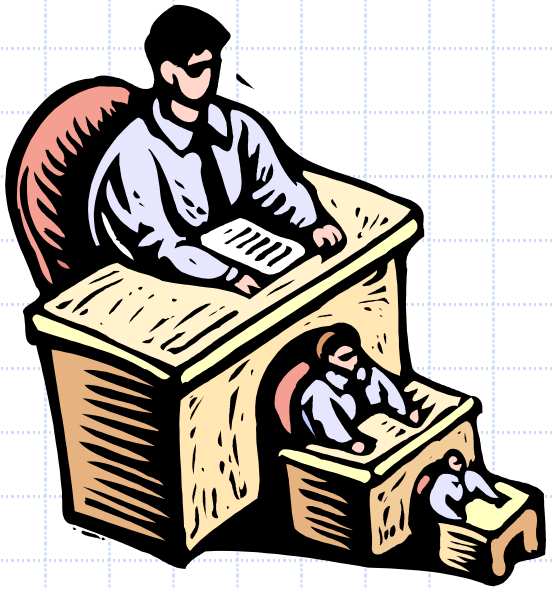
Recursion



Intro to Recursion

- Recursion: when a function calls itself
- Types of recursion
  - Linear recursion: one recursive call
    - Tail recursion: One recursive call at the very end
  - Binary recursion: Two recursive calls
  - Multiple recursion: More than two recursive calls

Easier to convert to iterative version

Example: Factorial Function

- Classic example--the factorial function:
  - n! = 1· 2· 3· … · (n-1)· n
- Recursive definition:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & else \end{cases}$$

- As a C++ method:

```
// recursive factorial function
int recursiveFactorial(int n) {
    if (n==0) return 1;                    // base case
    return n*recursiveFactorial(n- 1); // recursive case
}
```

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- One recursive call
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

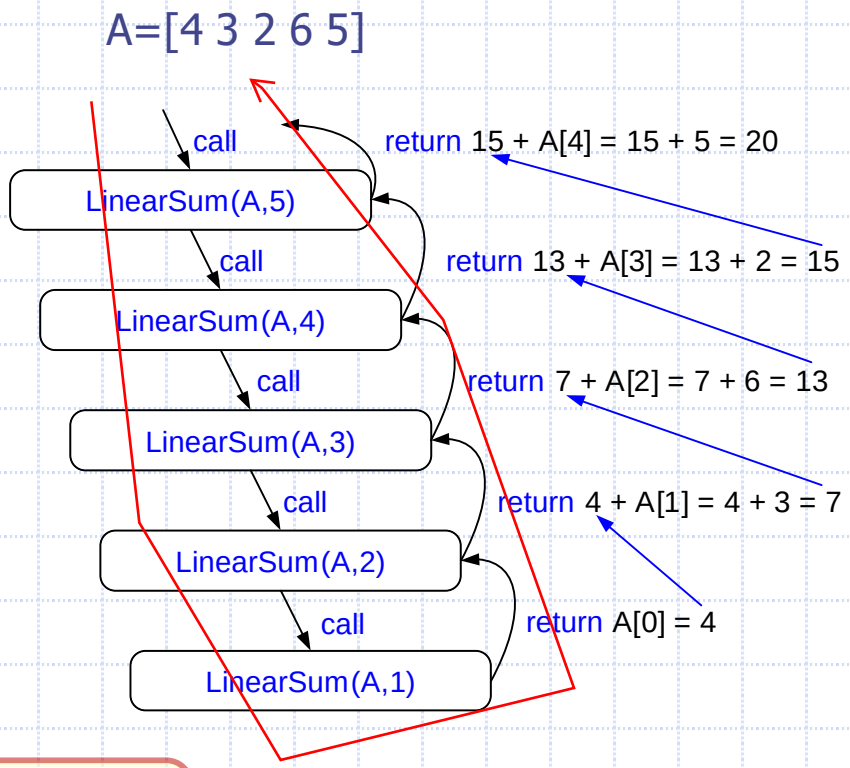
Example of Linear Recursion

Algorithm LinearSum(*A*, *n*):
Input:
A integer array *A* and an integer *n* = 1, such that *A* has at least *n* elements
Output:
The sum of the first *n* integers in *A*

```
if n = 1 then
  return A[0]
else
  return LinearSum(A, n - 1) +
    A[n - 1]
```

Alternative:
A[0]+LinearSum(A+1, n-1)

Example recursion trace:



Reversing an Array

Initial invocation:
ReverseArray(A, 0, n-1)

Algorithm ReverseArray(*A, i, j*):

Input: An array *A* and nonnegative integer indices *i* and *j*

Output: The reversal of the elements in *A* starting at index *i* and ending at *j*

if *i*<*j* **then**

Swap *A*[*i*] and *A*[*j*]

ReverseArray(*A, i* + 1, *j* - 1)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(*A, i, j*), not ReverseArray(*A*).

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n)=\begin{cases}1 & \text{if } n=0\\ x\cdot p(x,n-1) & \text{else}\end{cases}$$

- This leads to an power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- For example,

$$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$
$$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$
$$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$
$$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$

Recursive Squaring Method

Algorithm Power(x, n):
Input: A number x and integer $n = 0$
Output: The value x^n
if $n = 0$ **then**
return 1
if n is odd **then**
$y = \text{Power}(x, (n-1)/2)$
return $x \cdot y \cdot y$
else
$y = \text{Power}(x, n/2)$
return $y \cdot y$

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

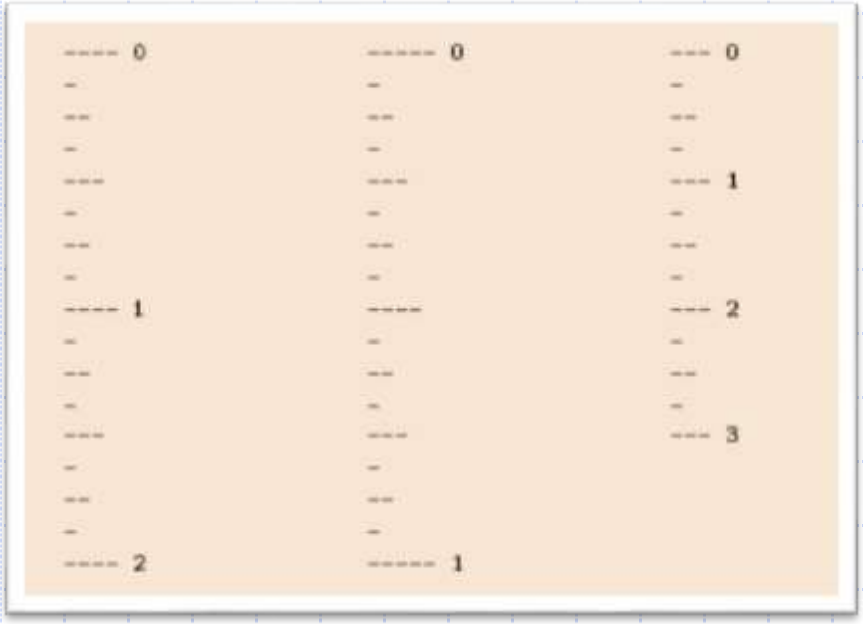
Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(*A, i, j*):
Input: An array *A* and nonnegative integer indices *i* and *j*
Output: The reversal of the elements in *A* starting at index *i* and ending at *j*
while *i* < *j* **do**
Swap *A*[*i*] and *A*[*j*]
i = *i* + 1
j = *j* - 1
return

Binary Recursion

- Binary recursion occurs whenever there are **two** recursive calls for each non-base case.
- Example: the DrawTicks method for drawing ticks on an English ruler.



A Binary Recursive Method for Drawing Ticks

```
                                                    // one tick with optional label
void drawOneTick(int tickLength, int tickLabel = -1) {
  for (int i = 0; i < tickLength; i++)
    cout << "-";
  if (tickLabel >= 0) cout << " " << tickLabel;
  cout << "\n";
}
void drawTicks(int tickLength) {                    // draw ticks of given length
  if (tickLength > 0) {                             // stop when length drops to 0
    drawTicks(tickLength-1);                        // recursively draw left ticks
    drawOneTick(tickLength);                        // draw center tick
    drawTicks(tickLength-1);                        // recursively draw right ticks
  }
}
void drawRuler(int nInches, int majorLength) {// draw the entire ruler
  drawOneTick(majorLength, 0);                      // draw tick 0 and its label
  for (int i = 1; i <= nInches; i++) {
    drawTicks(majorLength-1);                       // draw ticks for this inch
    drawOneTick(majorLength, i);                    // draw tick i and its label
  }
}
```

Note the two recursive calls

Draw inch labels

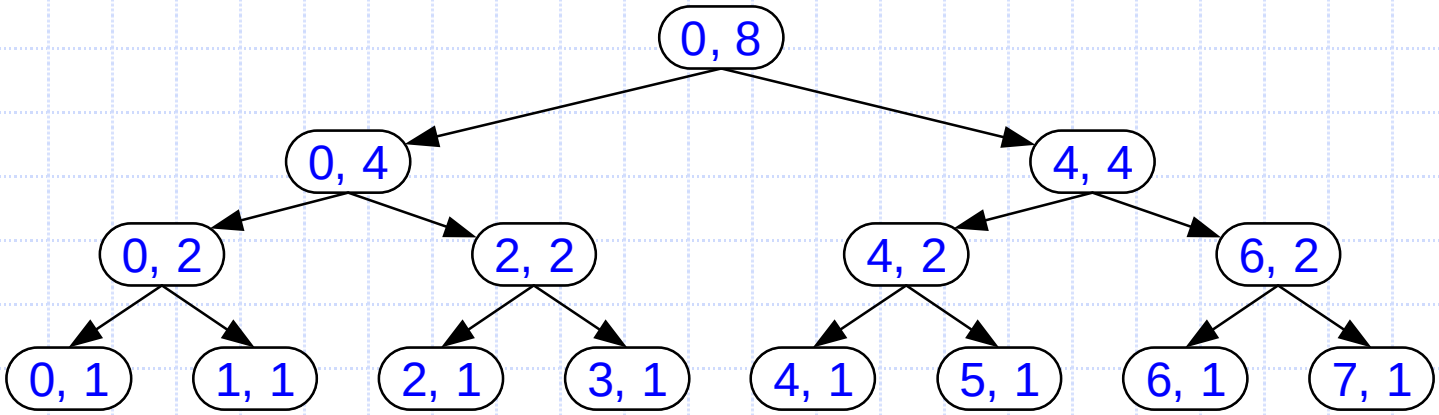
Code Fragment 3.37: A recursive implementation of a function that draws a ruler.

Another Binary Recusive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(*A, i, n*):
Input: An array *A* and integers *i* and *n*
Output: The sum of the *n* integers in *A* starting at index *i*
if $n = 1$ **then**
return $A[i]$
return BinarySum(*A, i, n*/2) + BinarySum(*A, i* + *n*/2, *n*/2)

- **Example trace:**



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

$F_0 = 0$

$F_1 = 1$

$F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if $k <= 1$ **then**

return k

return BinaryFib(k - 1) + BinaryFib(k - 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67$.
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

A Better Fibonacci Algorithm

- Use linear recursion instead

Algorithm LinearFibonacci(k):
Input: A nonnegative integer k
Output: Pair of Fibonacci numbers (F_k, F_{k-1})
if $k = 1$ **then**
return (k, 0)
else
(i, j) = LinearFibonacci($k - 1$)
return (i +j, i)

- LinearFibonacci makes k–1 recursive calls

Multiple Recursion

- Motivating example:
  - summation puzzles
    - *pot* + *pan* = *bib*
    - *dog* + *cat* = *pig*
    - *boy* + *girl* = *baby*
- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

Algorithm for Multiple Recursion

Algorithm PuzzleSolve(k, S, U):
Input: Integer k, sequence S, and set U (universe of elements to test)
Output: Enumeration of all k-length extensions to S using elements in U without repetitions
for all e in U **do**
if k = 0 **then**
if S solves the puzzle **then**
return “Solution found: ” S
else
PuzzleSolve(k-1, S+e, U-{e})

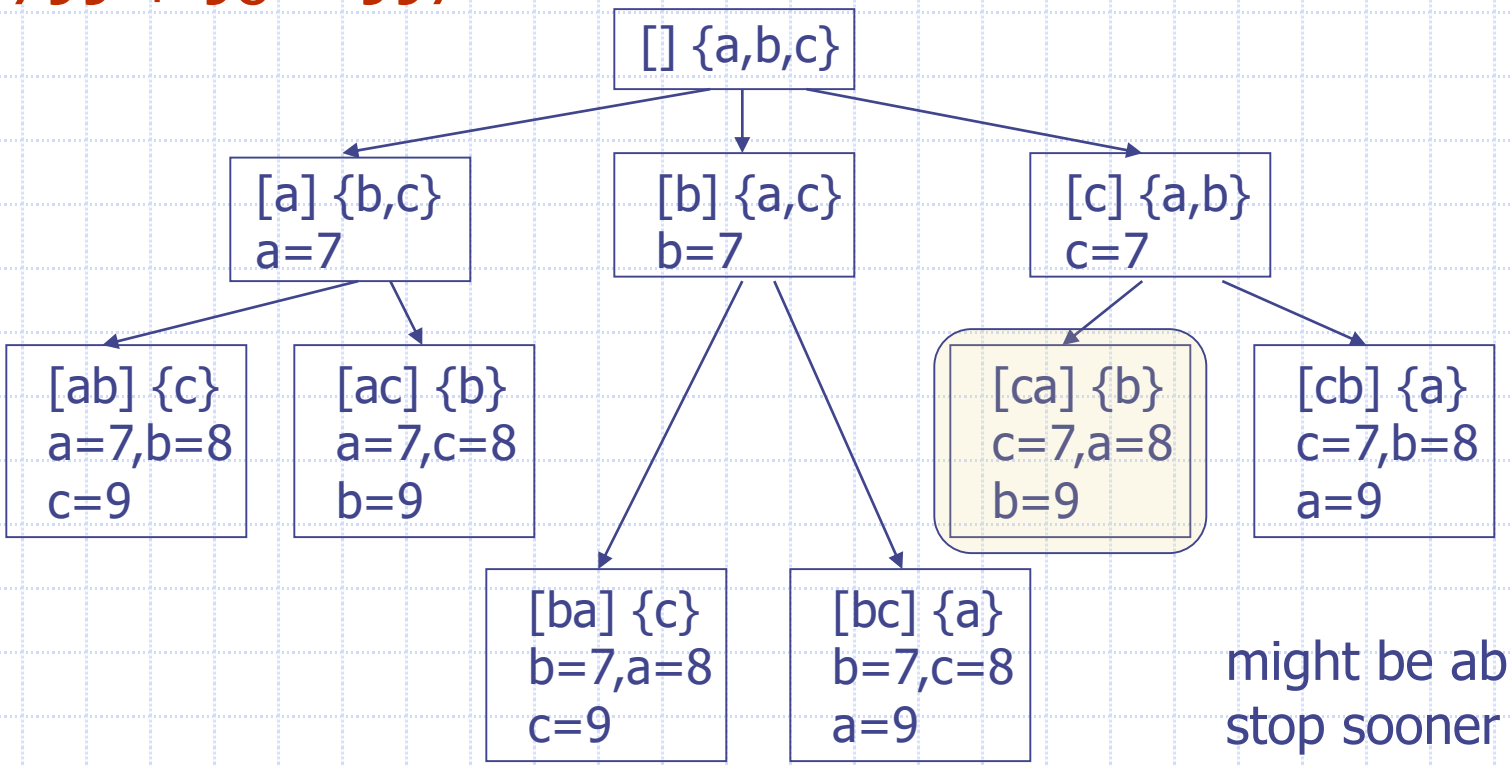
Slide by Matt Stallmann included with permission.

Example

cbb + ba = abc

799 + 98 = 997

a,b,c stand for 7,8,9; not necessarily in that order



might be able to stop sooner

Visualizing PuzzleSolve