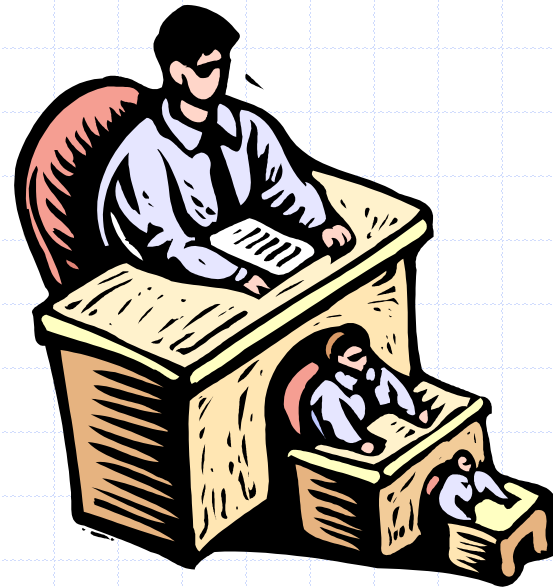


Recursion



The Recursion Pattern

- ❑ **Recursion:** when a method calls itself
- ❑ Classic example--the factorial function:
 - $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n-1) \cdot n$
- ❑ Recursive definition:

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

- ❑ As a Python method:

```
1 def factorial(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * factorial(n-1)
```

Content of a Recursive Method

□ Base case(s)

- Values of the input variables for which we perform no recursive calls are called **base cases** (there should be at least one base case).
- Every possible chain of recursive calls **must** eventually reach a base case.

□ Recursive calls

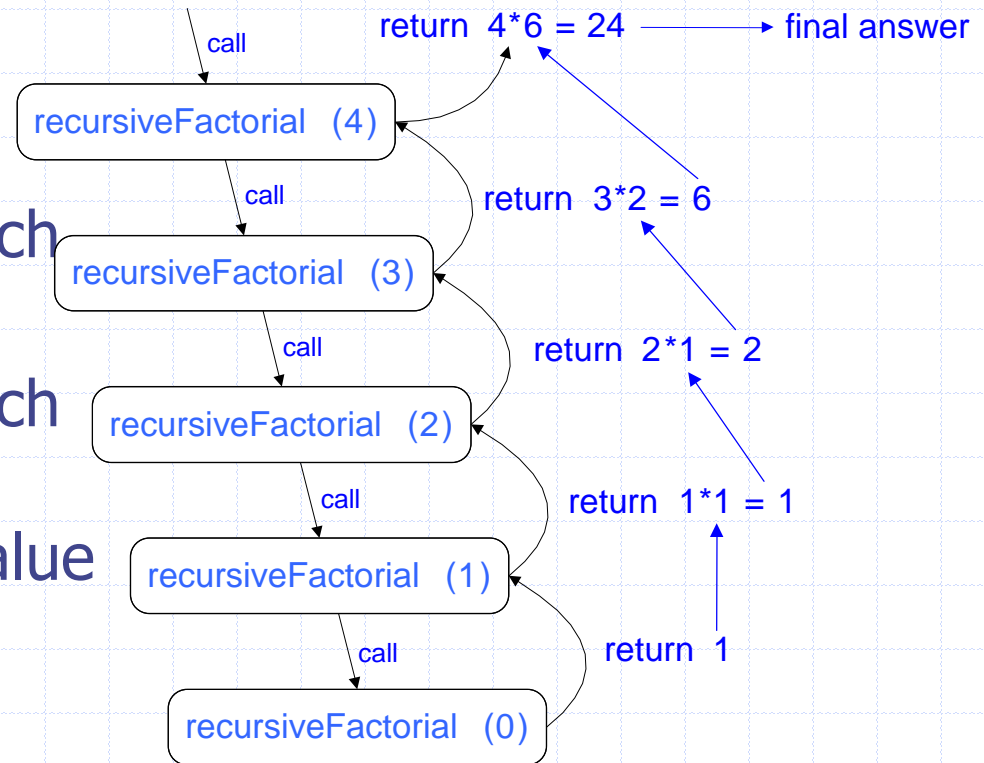
- Calls to the current method.
- Each recursive call should be defined so that it makes progress towards a base case.

Visualizing Recursion

□ Recursion trace

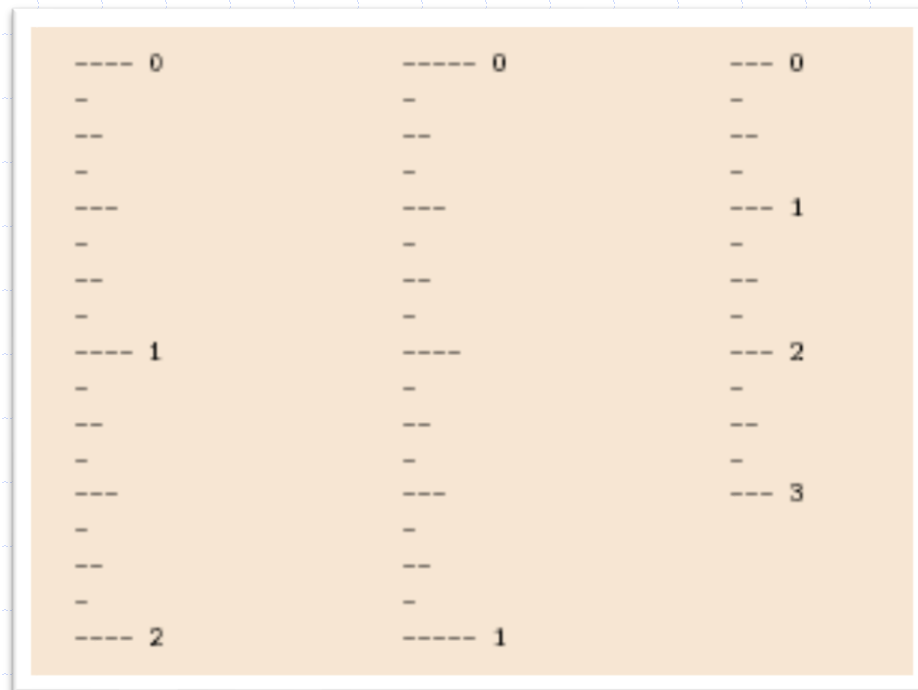
- A box for each recursive call
- An arrow from each caller to callee
- An arrow from each callee to caller showing return value

□ Example



Example: English Ruler

- Print the ticks and numbers like an English ruler:

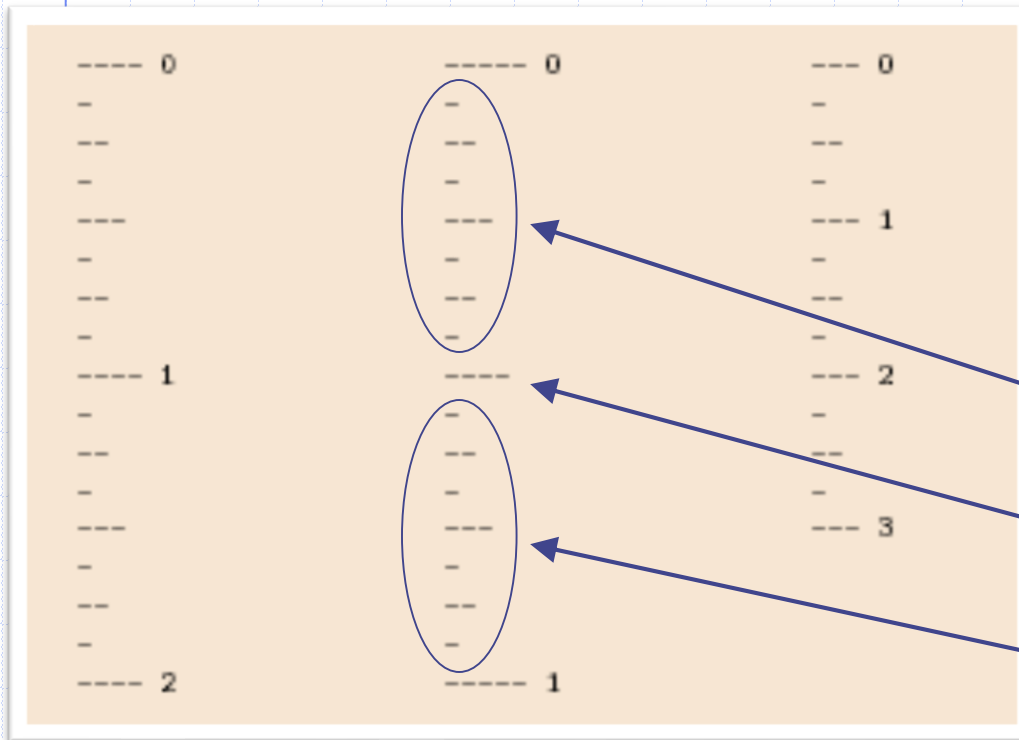


Using Recursion

`drawTicks(length)`

Input: length of a 'tick'

Output: ruler with tick of the given length in the middle and smaller rulers on either side



`drawTicks(length)`

if(length > 0) then

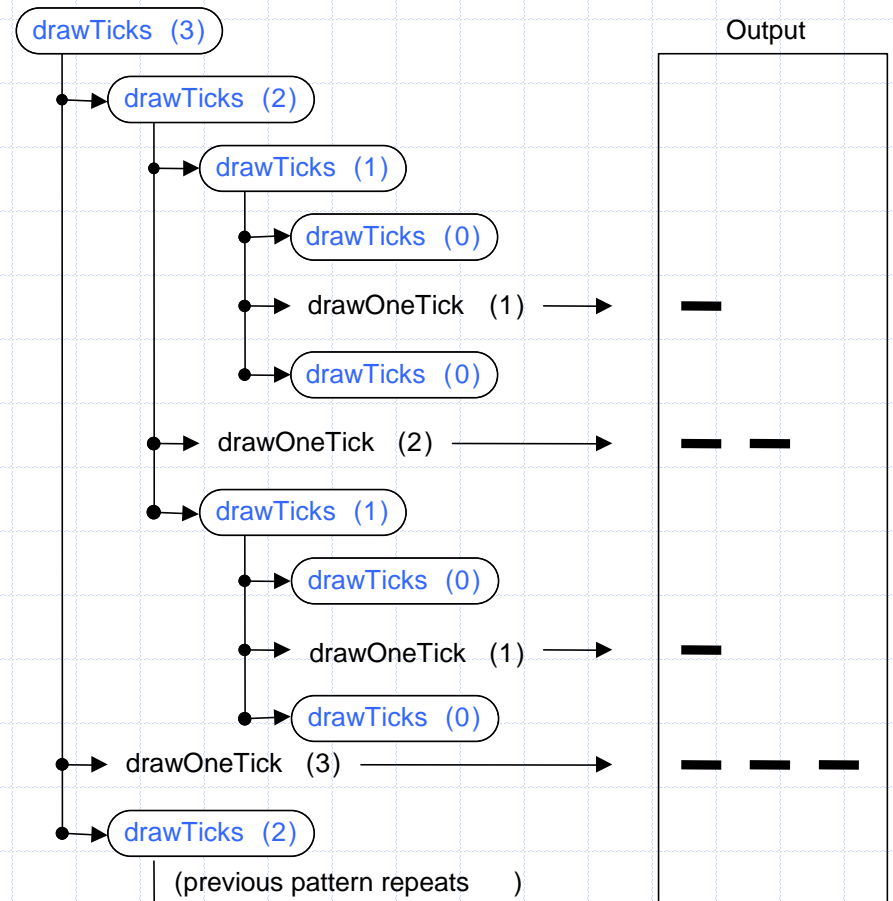
`drawTicks(length - 1)`

draw tick of the given length

`drawTicks(length - 1)`

Recursive Drawing Method

- The drawing method is based on the following recursive definition
- An interval with a central tick length $L \geq 1$ consists of:
 - An interval with a central tick length $L-1$
 - An single tick of length L
 - An interval with a central tick length $L-1$



A Recursive Method for Drawing Ticks on an English Ruler

```
1 def draw_line(tick_length, tick_label=' '):
2     """Draw one line with given tick length (followed by optional label)."""
3     line = '-' * tick_length
4     if tick_label:
5         line += ' ' + tick_label
6     print(line)
7
8 def draw_interval(center_length):
9     """Draw tick interval based upon a central tick length."""
10    if center_length > 0:
11        draw_interval(center_length - 1)
12        draw_line(center_length)
13        draw_interval(center_length - 1)
14
15 def draw_ruler(num_inches, major_length):
16     """Draw English ruler with given number of inches, major tick length."""
17     draw_line(major_length, '0')
18     for j in range(1, 1 + num_inches):
19         draw_interval(major_length - 1)
20         draw_line(major_length, str(j))
```

Note the two
recursive calls

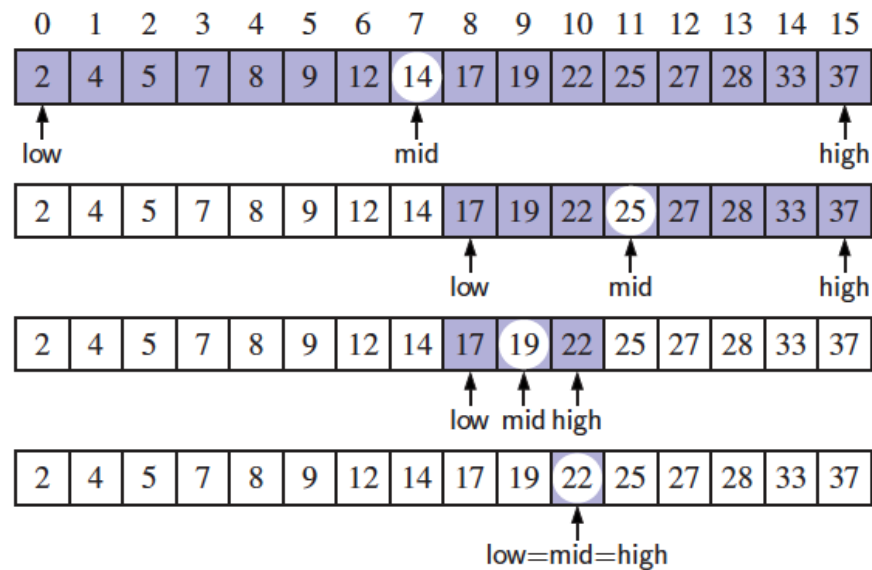
Binary Search

- Search for an integer, target, in an ordered list.

```
1 def binary_search(data, target, low, high):
2     """ Return True if target is found in indicated portion of a Python list.
3
4     The search only considers the portion from data[low] to data[high] inclusive.
5     """
6     if low > high:
7         return False                                # interval is empty; no match
8     else:
9         mid = (low + high) // 2
10        if target == data[mid]:                      # found a match
11            return True
12        elif target < data[mid]:
13            # recur on the portion left of the middle
14            return binary_search(data, target, low, mid - 1)
15        else:
16            # recur on the portion right of the middle
17            return binary_search(data, target, mid + 1, high)
```

Visualizing Binary Search

- We consider three cases:
 - If the target equals $\text{data}[\text{mid}]$, then we have found the target.
 - If $\text{target} < \text{data}[\text{mid}]$, then we recur on the first half of the sequence.
 - If $\text{target} > \text{data}[\text{mid}]$, then we recur on the second half of the sequence.



Analyzing Binary Search

- Runs in $O(\log n)$ time.
 - The remaining portion of the list is of size $\text{high} - \text{low} + 1$.
 - After one comparison, this becomes one of the following:

$$(\text{mid} - 1) - \text{low} + 1 = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor - \text{low} \leq \frac{\text{high} - \text{low} + 1}{2}$$

$$\text{high} - (\text{mid} + 1) + 1 = \text{high} - \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor \leq \frac{\text{high} - \text{low} + 1}{2}.$$

- Thus, each recursive call divides the search region in half; hence, there can be at most $\log n$ levels.

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.

□ Recur once

- 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

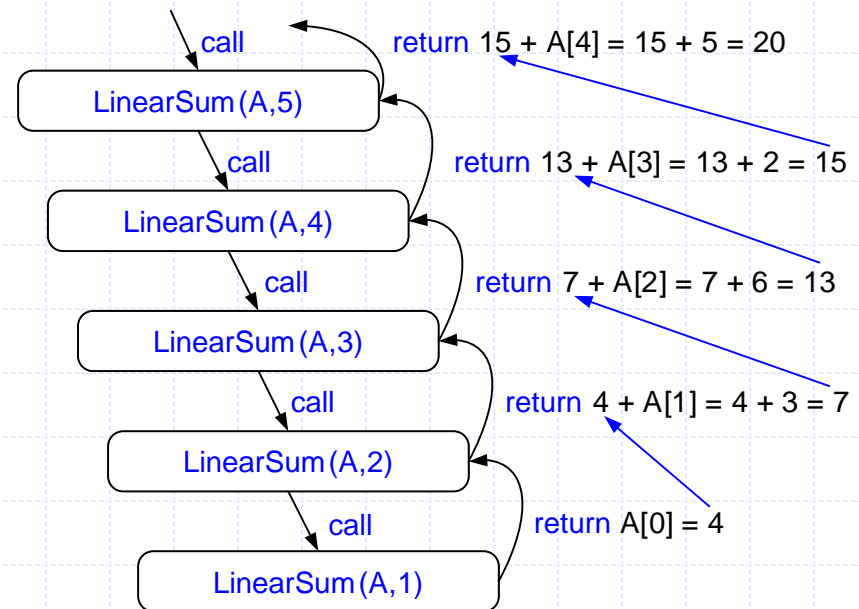
if $n = 1$ **then**

return $A[0]$

else

return LinearSum($A, n - 1$) + $A[n - 1]$

Example recursion trace:



Reversing an Array

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 parameters that are passed to the method.
- For example, we defined the array reversal method as `ReverseArray(A, i, j)`, not `ReverseArray(A)`.
- Python version:

```
1 def reverse(S, start, stop):
2     """Reverse elements in implicit slice S[start:stop]."""
3     if start < stop - 1:                                # if at least 2 elements:
4         S[start], S[stop-1] = S[stop-1], S[start]      # swap first and last
5         reverse(S, start+1, stop-1)                    # recur on rest
```

Computing Powers

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

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

- This leads to a 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 $n \geq 0$

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 $y \cdot y$

Analysis

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 $A[j]$

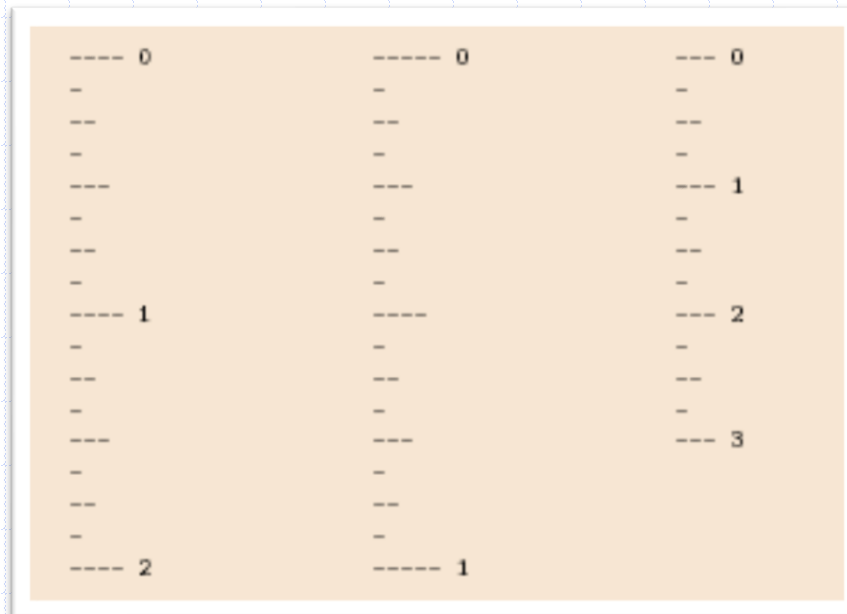
$i = i + 1$

$j = j - 1$

return

Binary Recursion

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



Another Binary Recursive 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

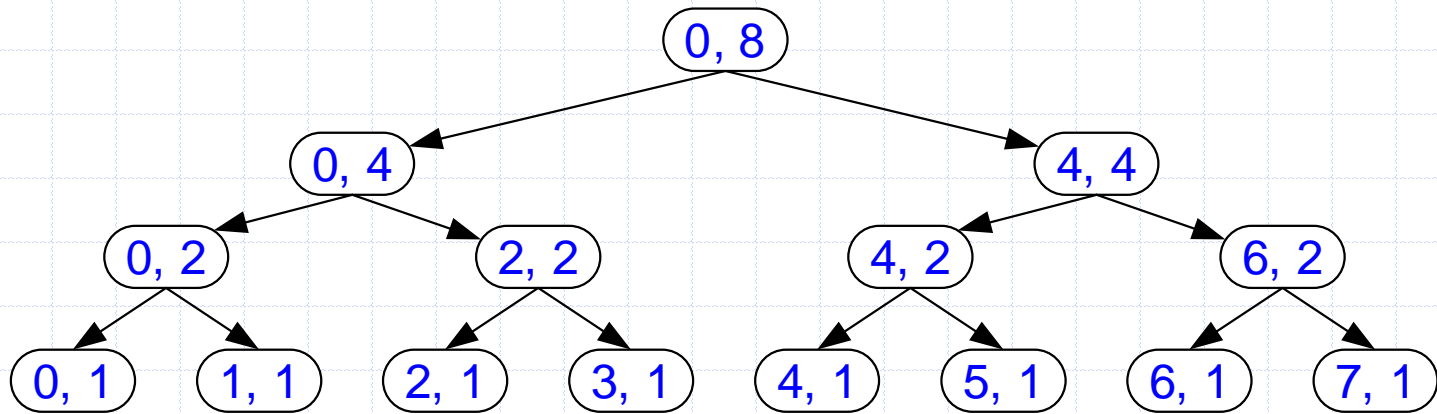
Output: The sum of the n integers in A starting at index i

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:

$$F_0 = 0$$

$$F_1 = 1$$

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

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The k th Fibonacci number F_k

if $k = 1$ **then**

return k

else

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

$(i, j) = \text{LinearFibonacci}(k - 1)$

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**

Remove e from U {e is now being used}

Add e to the end of S

if k = 1 **then**

Test whether S is a configuration that solves the puzzle

if S solves the puzzle **then**

return "Solution found: " S

else

PuzzleSolve(k - 1, S,U)

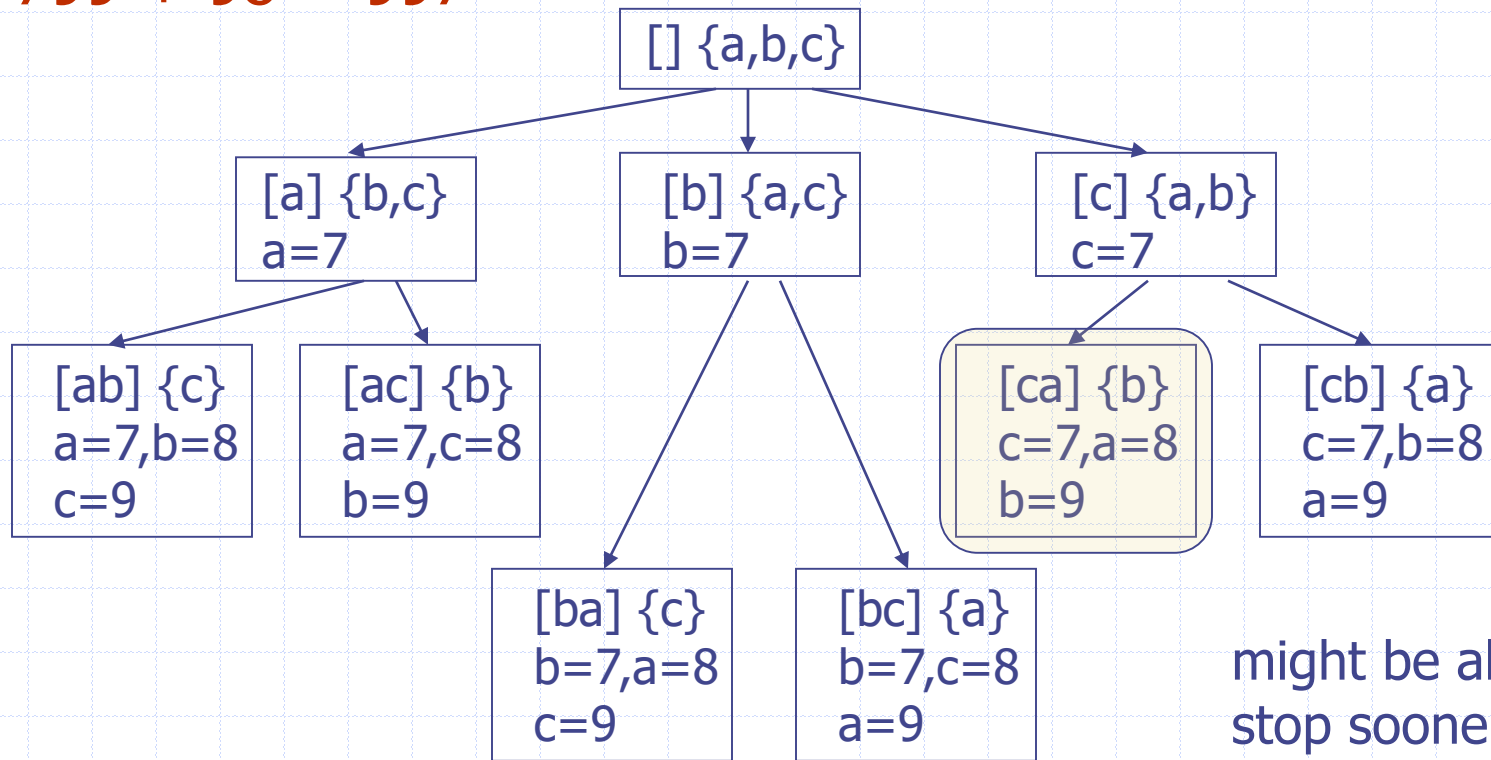
Add e back to U {e is now unused}

Remove e from the end of S

Example

$$\begin{aligned}cbb + ba &= abc \\ 799 + 98 &= 997\end{aligned}$$

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



might be able to
stop sooner

Visualizing PuzzleSolve

