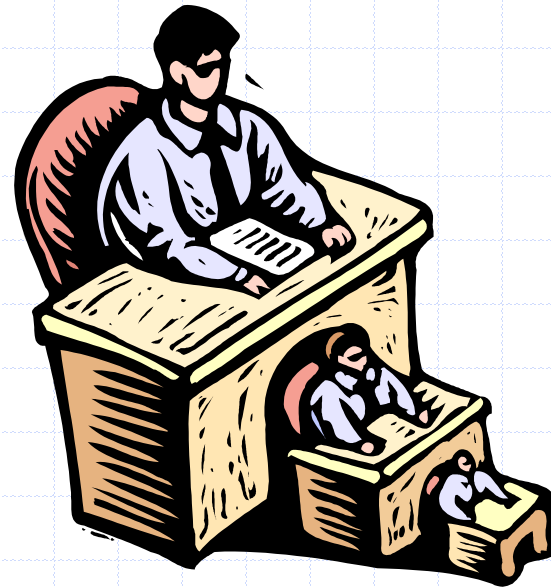


Presentation for use with the textbook **Data Structures and Algorithms in Java, 6th edition**, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

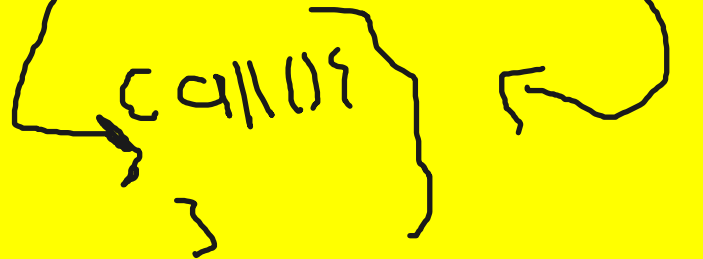
Recursion



Analyzing Recursive Algorithms

- 1) ☐ Number of primitive operations
- 2) ☐ Big-Oh notation
- ☐ #primitive operations of a particular activation of a method
عدد
- ☐ Overall #operations = Summation over all activations

داخل المنثود



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**.

بازگشت

عائنه تخلص

The Recursion Pattern

Recursion no prefred

لازم لما تعمل تحلل الاشئ صح لانها مكلفة

- ❑ **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 Java method:

```
1 public static int factorial(int n) throws IllegalArgumentException {
2     if (n < 0)
3         throw new IllegalArgumentException(); // argument must be nonnegative
4     else if (n == 0) n == 1 // يمكن أوفر ع حالي لفة عن طريق إضافة
5         return 1; // base case
6     else
7         return n * factorial(n-1); // recursive case
8 }
```

caller
↙ ↘
callee

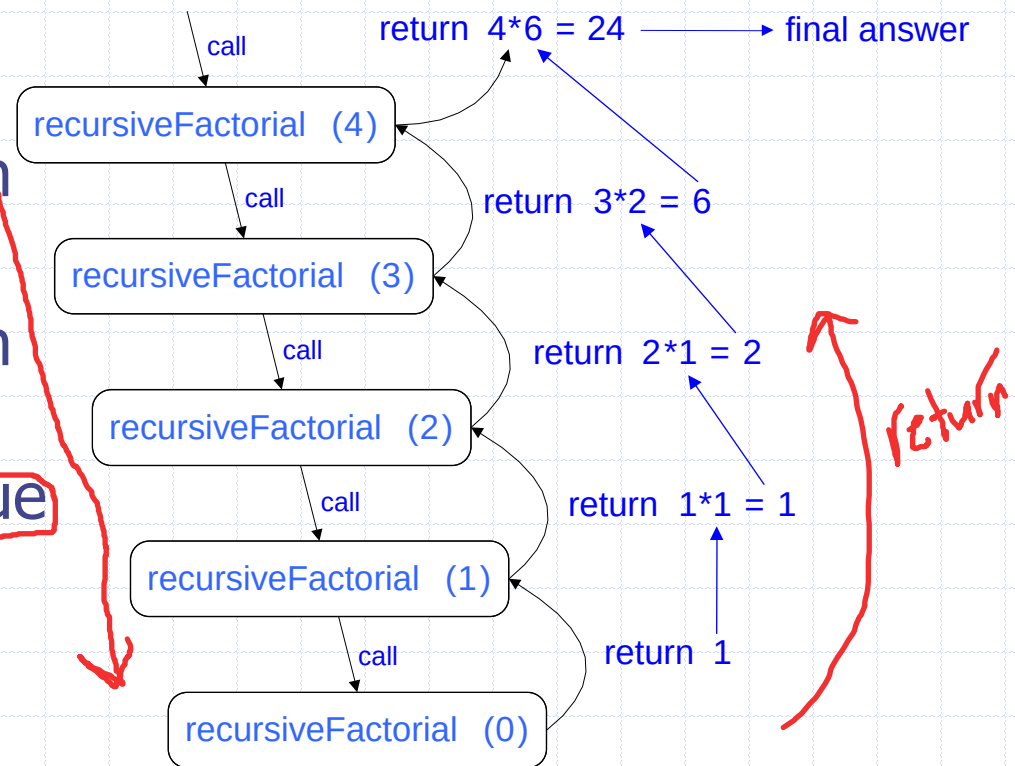
Visualizing Recursion

□ 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

recursion

□ Example



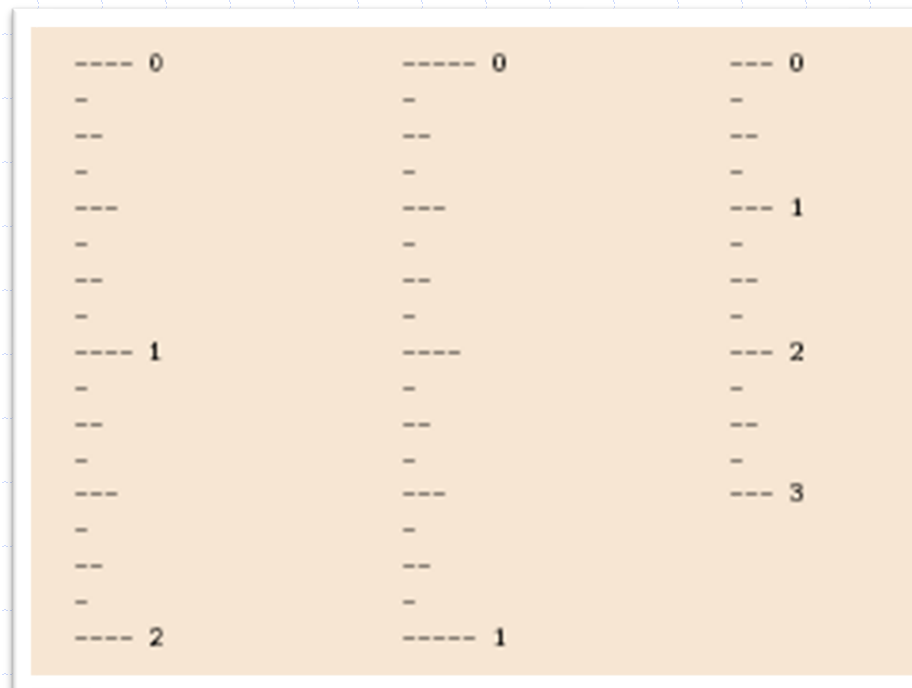
Analyzing Factorial

- $n+1$ activations
- Each individual activation executes a constant #operations
- Overall #operations is $O(n)$ time.

$$\left. \begin{array}{l} 4! = 4 \times 3! \\ 3! = 3 \times 2! \\ 2! = 2 \times 1! \\ 1! = 1 \times 0! \\ 0! = 1 \end{array} \right\} n+1 = 5$$

Example: English Ruler

- Print the ticks and numbers like an English ruler:

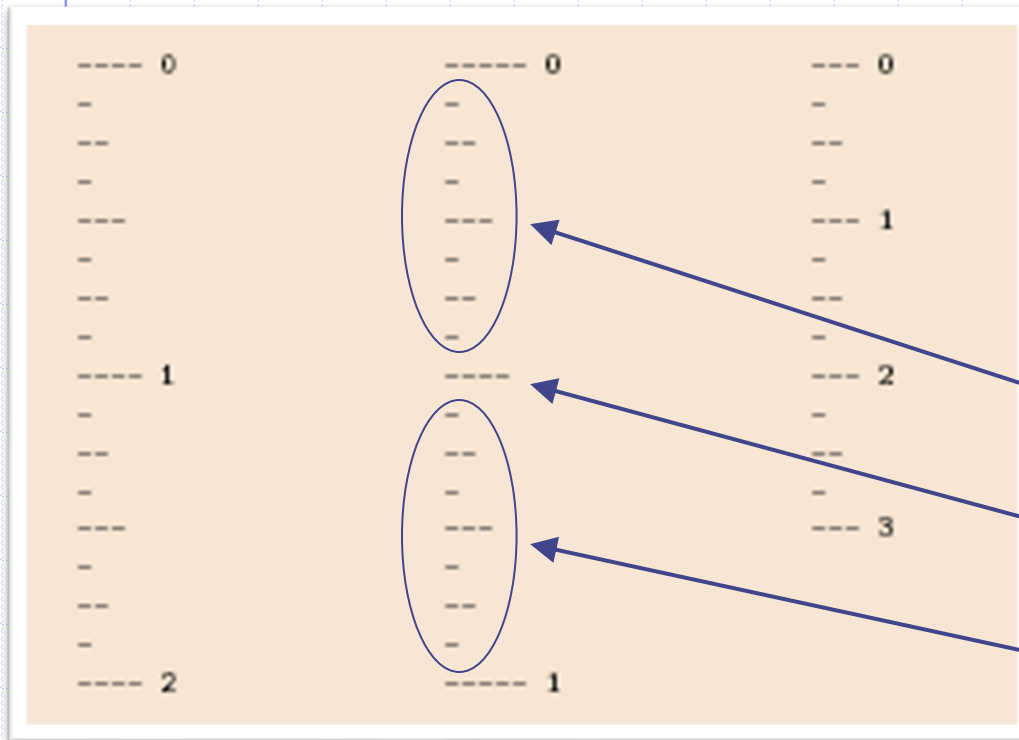


Using Recursion

`drawInterval(length)`

Input: length of a 'tick'

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



`drawInterval(length)`

if(length > 0) then

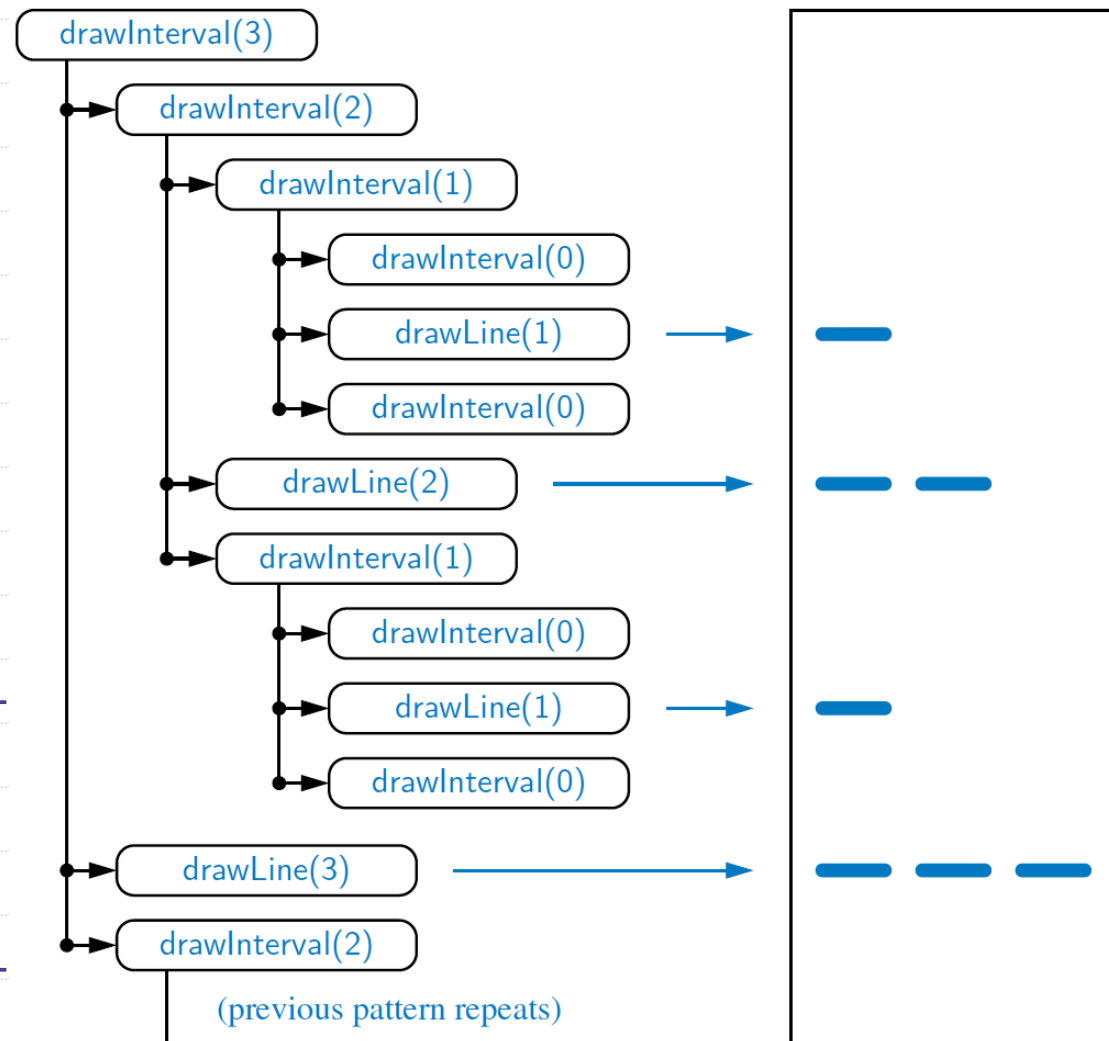
`drawInterval ( length - 1 )`

draw line of the given length

`drawInterval ( 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 $L-1$
 - A 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  /** Draws an English ruler for the given number of inches and major tick length. */
2  public static void drawRuler(int nInches, int majorLength) {
3      drawLine(majorLength, 0);           // draw inch 0 line and label
4      for (int j = 1; j <= nInches; j++) {
5          drawInterval(majorLength - 1); // draw interior ticks for inch
6          drawLine(majorLength, j);      // draw inch j line and label
7      }
8  }
9  private static void drawInterval(int centralLength) {
10     if (centralLength >= 1) {           // otherwise, do nothing
11         drawInterval(centralLength - 1); // recursively draw top interval
12         drawLine(centralLength);        // draw center tick line (without label)
13         drawInterval(centralLength - 1); // recursively draw bottom interval
14     }
15 }
16 private static void drawLine(int tickLength, int tickLabel) {
17     for (int j = 0; j < tickLength; j++) {
18         System.out.print("-");
19         if (tickLabel >= 0)
20             System.out.print(" " + tickLabel);
21         System.out.print("\n");
22     }
23 }
24 /** Draws a line with the given tick length (but no label). */
25 private static void drawLine(int tickLength) {
26     drawLine(tickLength, -1);
27 }

```

Note the two recursive calls

-- --
 ديم -- --

0 ---
 1, 5 ---

1 ---
 1, 5 ---
 2 ---

عشان ما يطبع الرقم

Analyzing English Ruler

- we consider the fundamental question of how many total lines of output are generated by an initial call to drawInterval(c), where c denotes the center length.
- This is a reasonable benchmark for the overall efficiency of the algorithm as each line of output is based upon a call to the drawLine utility, and each recursive call to drawInterval with nonzero parameter makes exactly one direct call to drawLine.
- We know that a call to drawInterval(c) for $c > 0$ spawns two calls to drawInterval($c-1$) and a single call to drawLine. We will rely on this intuition to prove the following claim.
- For $c \geq 0$, a call to drawInterval(c) results in precisely $2^c - 1$ lines of output.

Justification: We provide a formal proof of this claim by

induction

$I_n(3)$
↓
 $I(2)$
 $I(1)$
 $I(0)$

Proof

Base case : if $c=0$, $2^0-1 >>> 0$

hypos , we assume that is correct for drawIntreveal (c)

induction we have (c+1) complexcity is $2^{(c+1)} - 1$

c+1 step = $2 * c \text{ step} + 1$

$2*(2^c-1)+1 = 2^{(c+1)} - 2 + 1 >>> 2^{(c+1)} - 1$

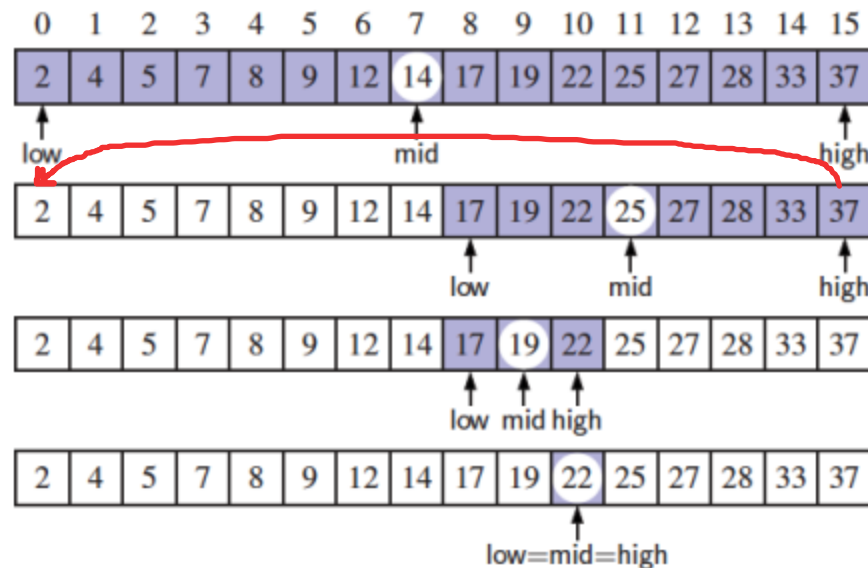
Binary Search

Search for an integer in an ordered list

```
1  /**
2   * Returns true if the target value is found in the indicated portion of the data array.
3   * This search only considers the array portion from data[low] to data[high] inclusive.
4   */
5  public static boolean binarySearch(int[ ] data, int target, int low, int high) {
6      if (low > high)
7          return false;                                // interval empty; no match
8      else {
9          int mid = (low + high) / 2;
10         if (target == data[mid])
11             return true;                                // found a match
12         else if (target < data[mid])
13             return binarySearch(data, target, low, mid - 1); // recur left of the middle
14         else
15             return binarySearch(data, target, mid + 1, high); // recur right of the middle
16     }
17 }
```

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 $high - low + 1$
 - After one comparison, this becomes one of the following:

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

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

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

Disk Usage

```
1  /**
2   * Calculates the total disk usage (in bytes) of the portion of the file system rooted
3   * at the given path, while printing a summary akin to the standard 'du' Unix tool.
4   */
5   public static long diskUsage(File root) {
6       long total = root.length();           // start with direct disk usage
7       if (root.isDirectory()) {            // and if this is a directory,
8           for (String childname : root.list()) { // then for each child
9               File child = new File(root, childname); // compose full path to child
10              total += diskUsage(child);        // add child's usage to total
11          }
12      }
13      System.out.println(total + "\t" + root); // descriptive output
14      return total;                          // return the grand total
15  }
```

Code Fragment 5.5: A recursive method for reporting disk usage of a file system.

Further Examples of Recursion

- Linear recursion (factorial, binary search, summing elements of an array, reversing a sequence, ..)
- Binary recursion (Drawing English ruler, binary sum of an array, ...)
- Multiple recursion (Calculating disk space of a file system, summation puzzles, ...)

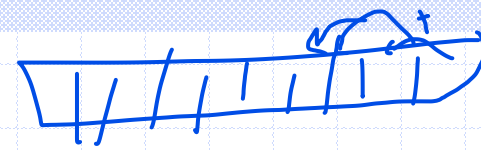
Linear Recursion

□ 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** ^{finally} 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 ^{if else or}
- Define each possible recursive call so that it makes progress towards a base case.



Example of Linear Recursion

Algorithm **linearSum**(A, n):

Input:

Array, A, of integers
Integer n such that
 $0 \leq n \leq |A|$

Output:

Sum of the first n
integers in A

if $n = 0$ then

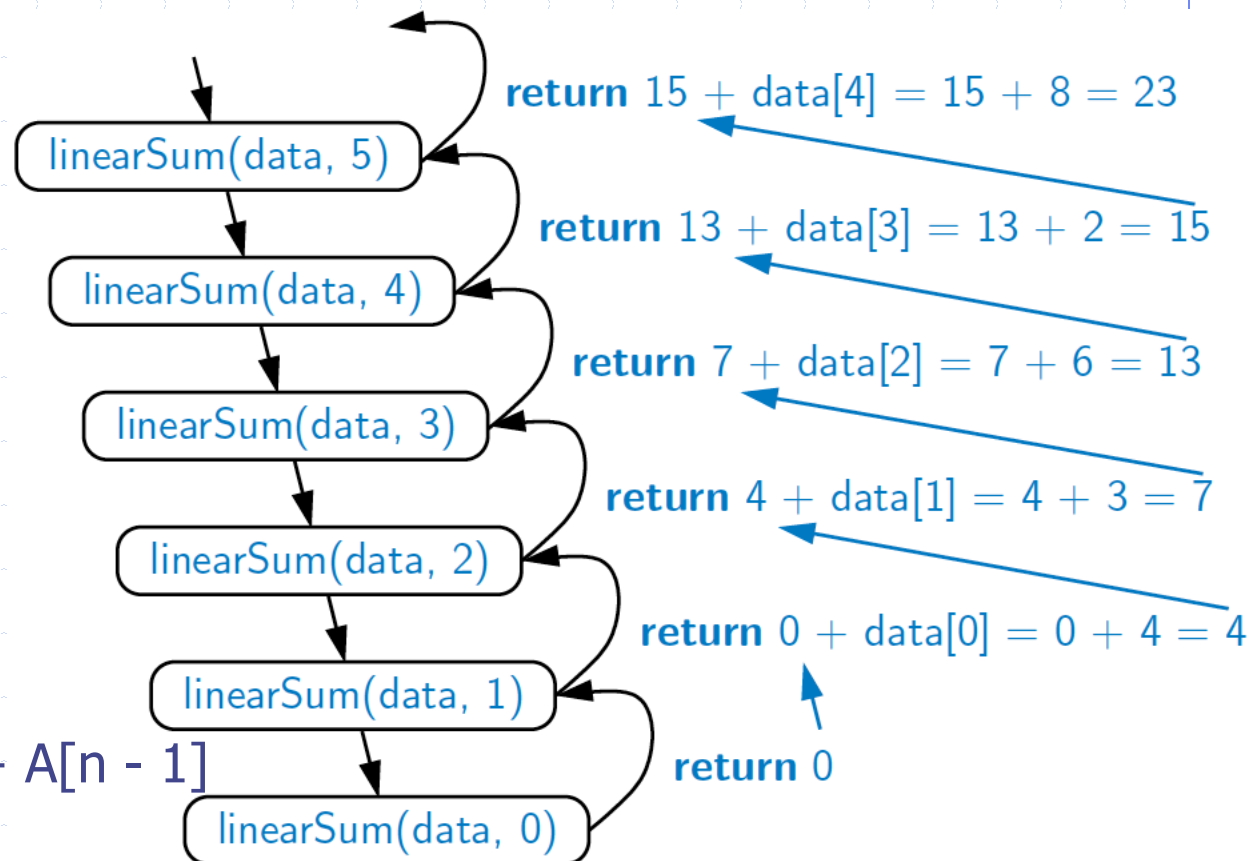
return 0

else

return

linearSum(A, n - 1) + A[n - 1]

Recursion trace of **linearSum**(data, 5)
called on array data = [4, 3, 6, 2, 8]



Example of Linear Recursion

Algorithm **linearSum**(A, n):

Input:

Array, A, of integers
Integer n such that
 $0 \leq n \leq |A|$

Output:

Sum of the first n
integers in A

if $n = 0$ then

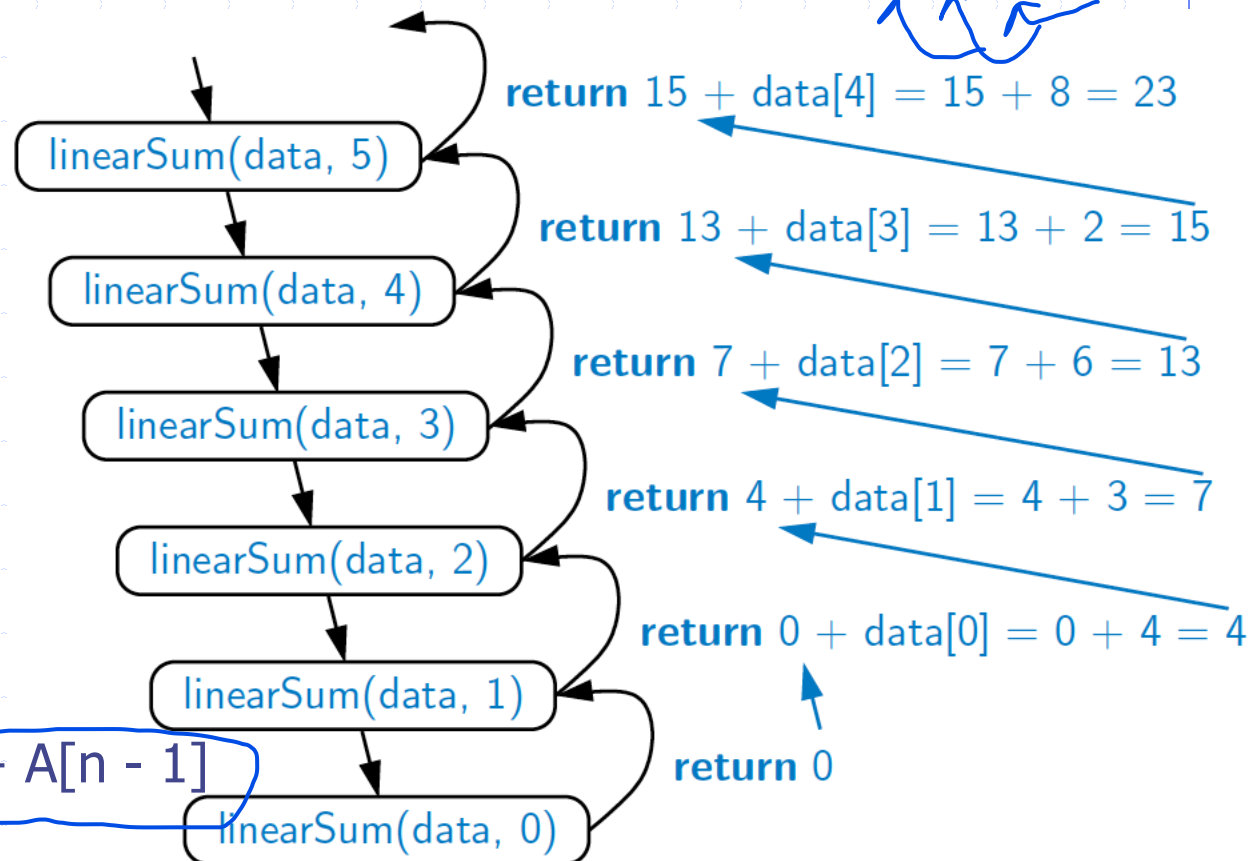
return 0

else

return

linearSum(A, n - 1) + A[n - 1]

Recursion trace of **linearSum**(data, 5)
called on array data = [4, 3, 6, 2, 8]



Example of Linear Recursion (cont.)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	3	6	2	8	9	3	2	8	5	1	7	2	8	3	7

Figure 5.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n - 1$.

Can we optimise the recursion to be a tail-recursion?

$\text{sum } A(A, n, s) \{$
if $n = 0 \rightarrow \text{return } s$
else $\rightarrow \text{sum } A(A, n-1, A[n-1] + s)$

✓

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

if $i < j$ then

 Swap $A[i]$ and $A[j]$

`reverseArray(A, i + 1, j - 1)`

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)**

```
1  /** Reverses the contents of subarray data[low] through data[high] inclusive. */
2  public static void reverseArray(int[ ] data, int low, int high) {
3      if (low < high) {                                // if at least two elements in subarray
4          int temp = data[low];                        // swap data[low] and data[high]
5          data[low] = data[high];
6          data[high] = temp;
7          reverseArray(data, low + 1, high - 1);      // recur on the rest
8      }
9  }
```

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 an power function that runs in $O(n)$ time (for we make n recursive calls)
- We can do better than this, however (see the next slide)

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 $y \cdot y$

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

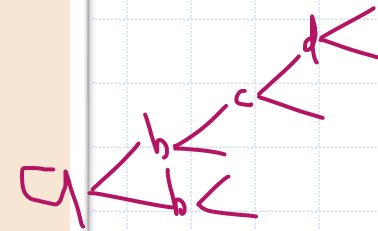
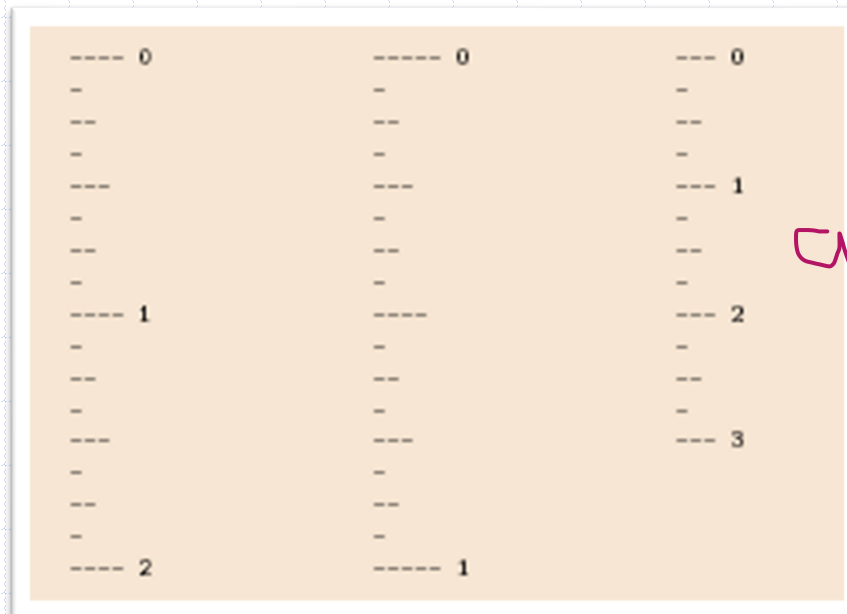
 j = j - 1

return

such as
recursive

Binary Recursion

- Binary recursion occurs whenever there are two recursive calls for each non-base case.
- Example from before: the **drawInterval** 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

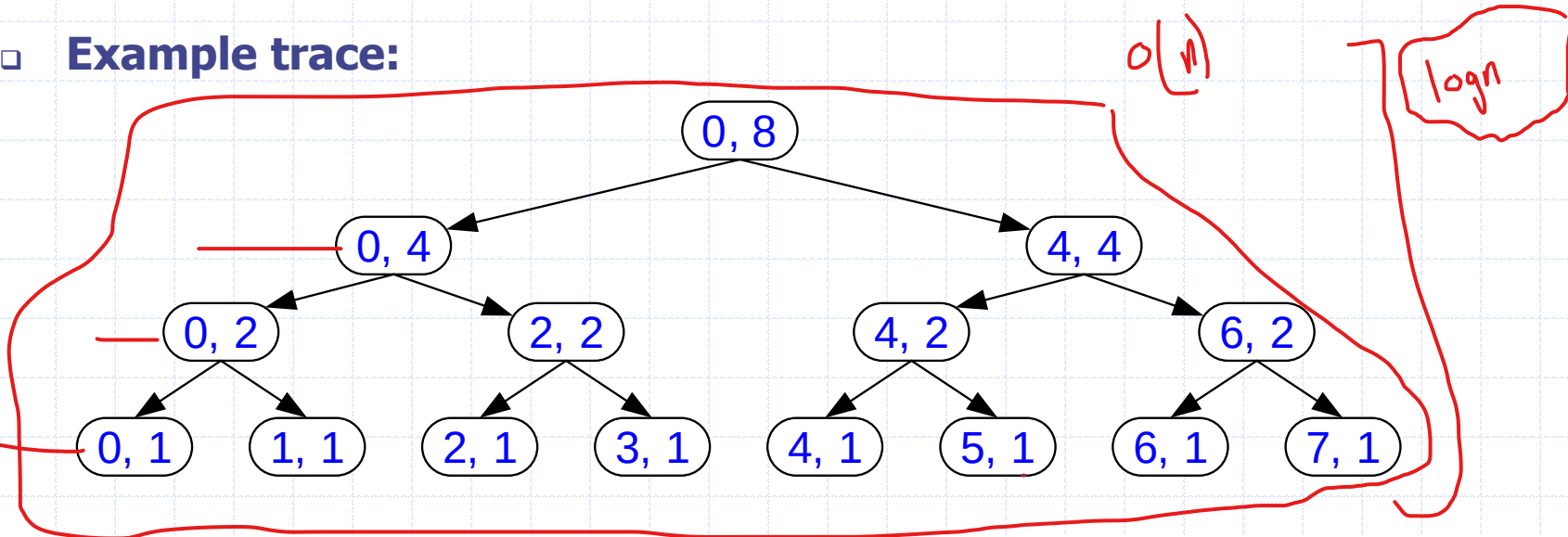
if $n = 1$ **then**

return A[i]

return BinarySum(A, i, $n/2$) + BinarySum(A, $i + n/2$, $n/2$)

عدد الـ n - 1
Level = $\log n$

- 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** // $k = 0$

return k

else

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

Analysis

□ Let n_k be the number of recursive calls by **BinaryFib**(k)

■ $n_0 = 1$

■ $n_1 = 1$

■ $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$

■ $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$

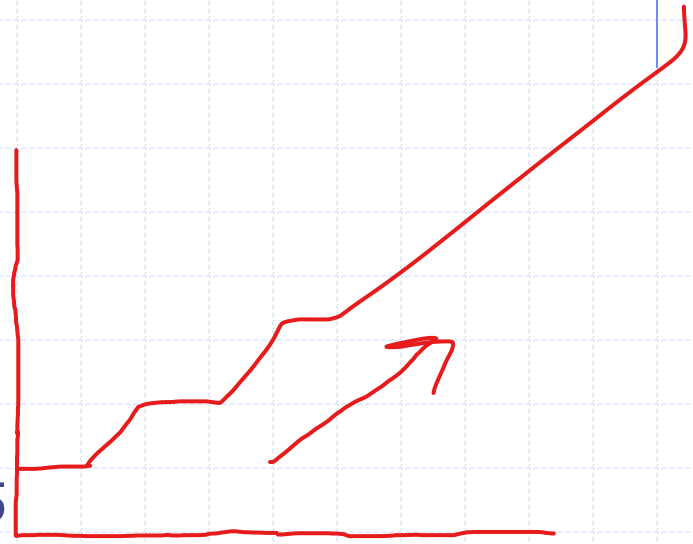
■ $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$

■ $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$

■ $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$

■ $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$

■ $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$



□ 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 $k \leq 1$ **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**

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)`

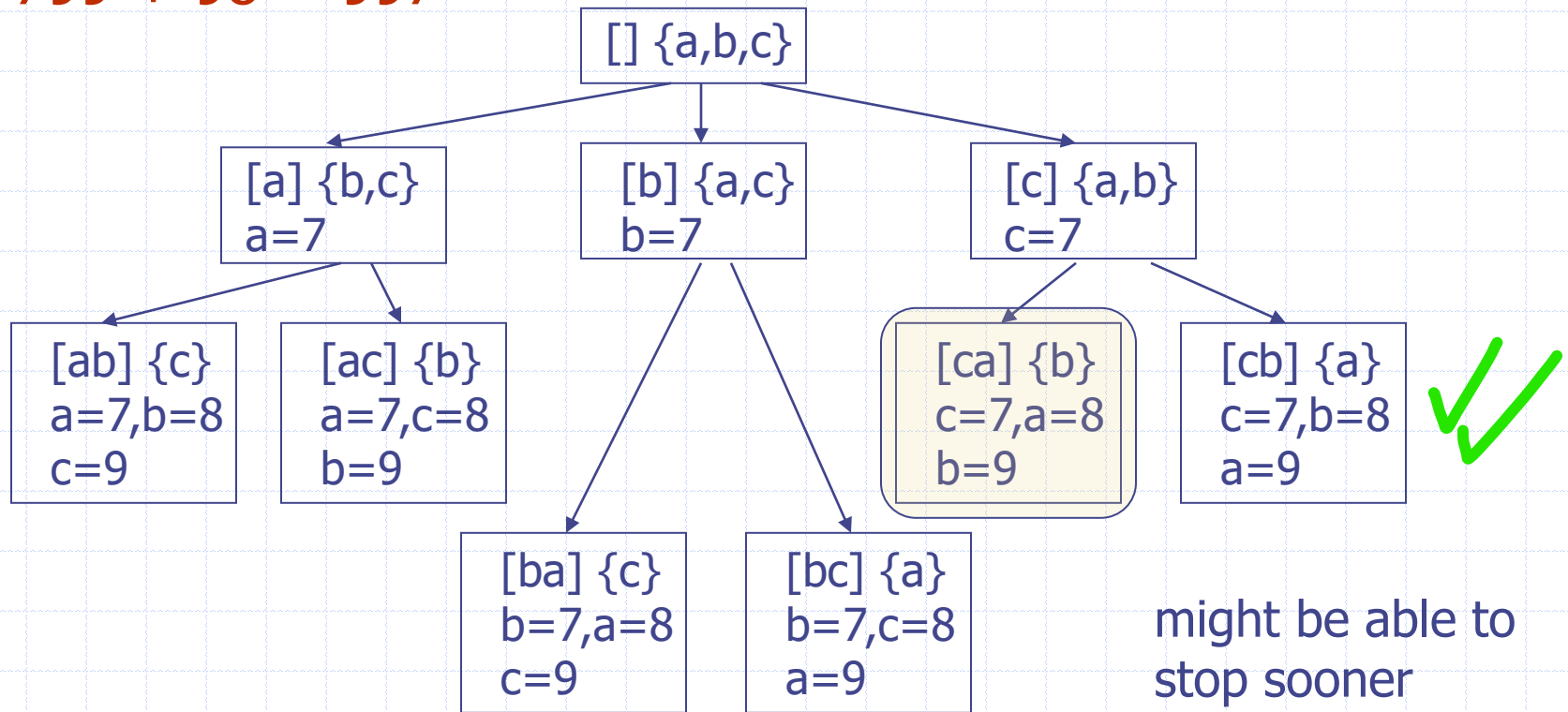
Add e back to U { e is now unused}

Remove e from the end of S

Example

$cbb + ba = abc$
 $799 + 98 = 997$

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



Visualizing PuzzleSolve

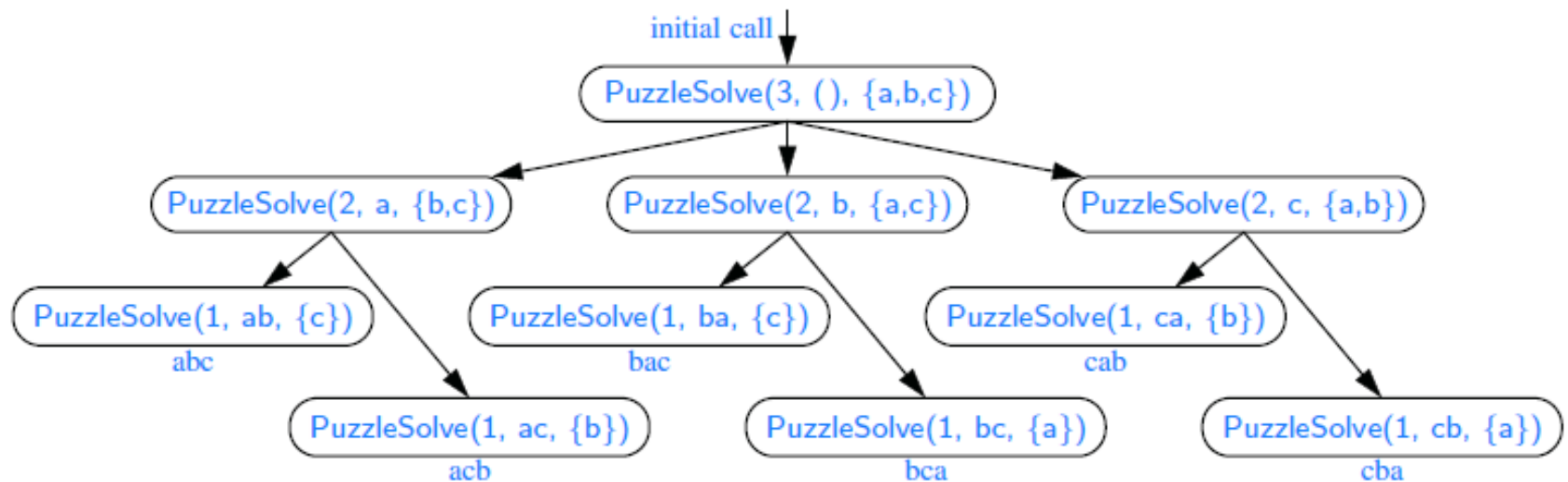


Figure 5.14: Recursion trace for an execution of `PuzzleSolve(3, S, U)`, where S is empty and $U = \{a, b, c\}$. This execution generates and tests all permutations of a, b , and c . We show the permutations generated directly below their respective boxes.