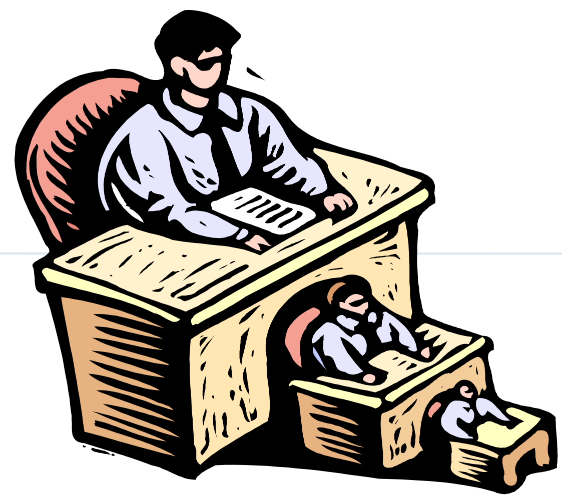
RECURSION (CH 5)

A pattern for solving algorithm design problems



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

THE RECURSION PATTERN EXAMPLE

- **Recursion**: when a method calls itself
- Classic example – the factorial function:
  n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

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

- As a Java method:

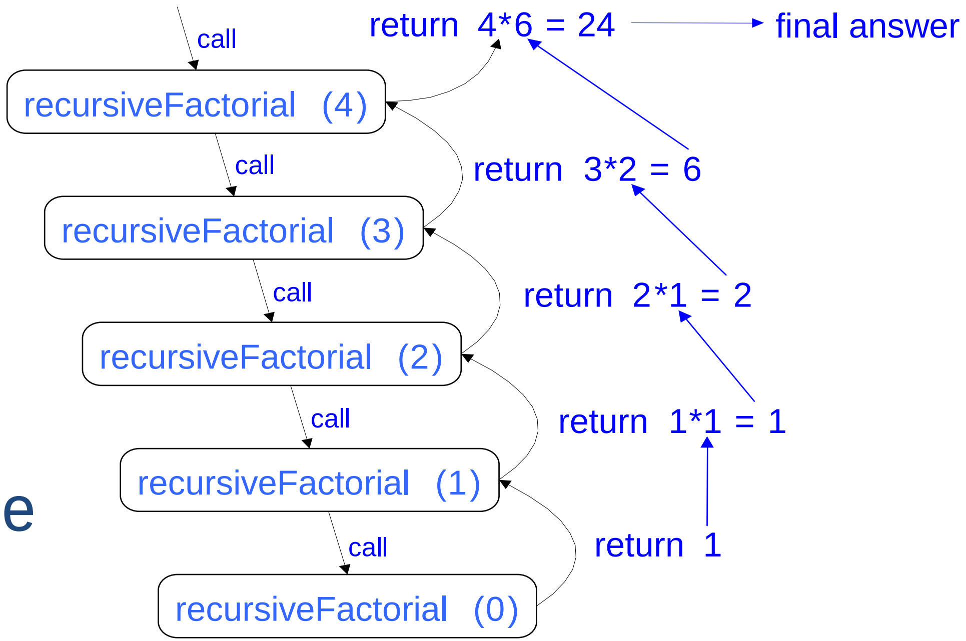
```
public static int factorial(int n) throws IllegalArgumentException {
  if (n < 0)
    throw new IllegalArgumentException();      // argument must be nonnegative
  else if (n == 0)
    return 1;                                   // base case
  else
    return n * factorial(n−1);                  // recursive case
}
```

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

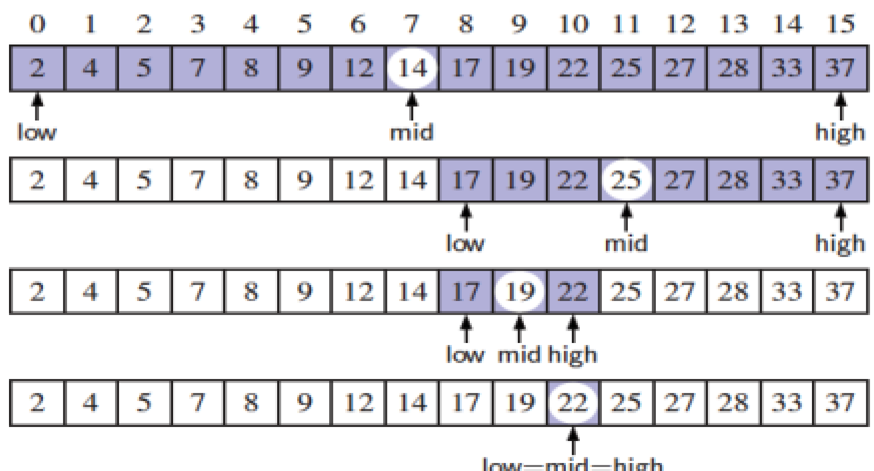
- Recursion trace
  - A box for each recursive call
  - An arrow from each caller to callee
  - An arrow from each callee to caller showing return value
- Example



VISUALIZING BINARY SEARCH

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

INPUT: Values stored in sorted order within an array (sequence is ***sorted*** and ***indexable***,)
OUTPUT: Location of the target value.



BINARY SEARCH

Search for an integer in an ordered list

```
/**
 * Returns true if the target value is found in the indicated portion of the data array.
 * This search only considers the array portion from data[low] to data[high] inclusive.
 */
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                                        // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                                       // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);   // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high);  // recur right of the middle
  }
}
```

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:

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

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

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

TYPES OF RECURSION

- ***Linear recursion :*** If a recursive call starts at most one other.
- ***Binary recursion***: If a recursive call may start two others.
- ***Multiple recursion:*** If a recursive call may start three or more others.

Terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time.

LINEAR RECURSION

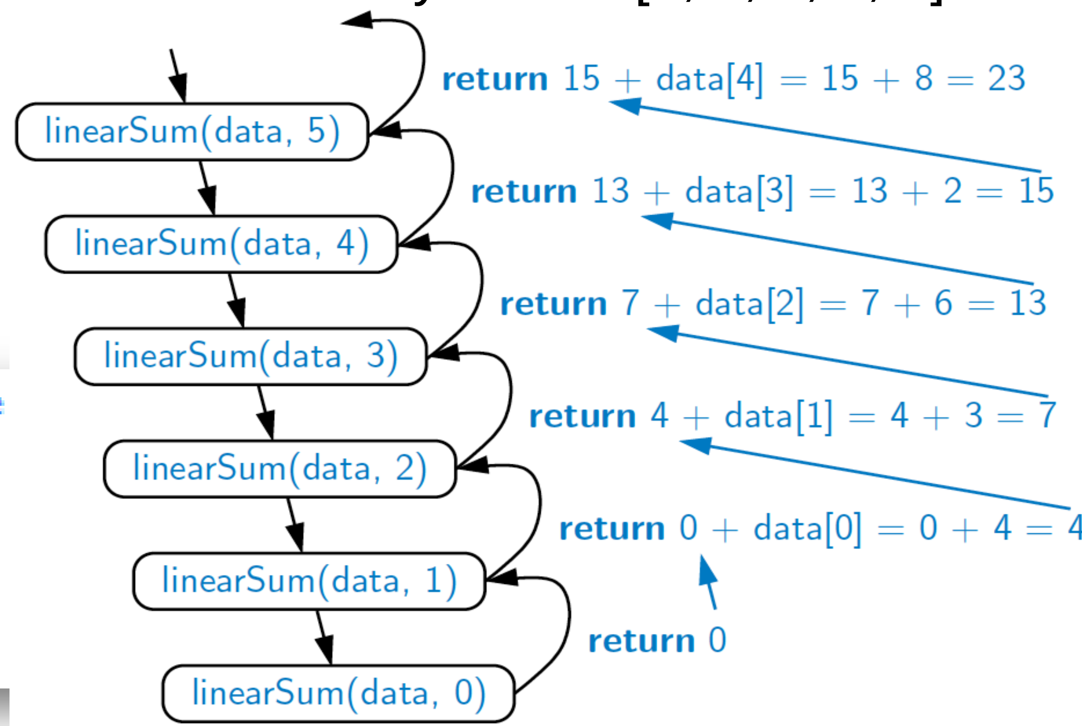
- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step 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:
  Array, A, of integers
  Integer n such that
    $0 \le n \le |A|$
Output:
  Sum of the first n integers in A.

```
/** Returns the sum of the first n integers of the
public static int linearSum(int[ ] data, int n) {
  if (n == 0)
    return 0;
  else
    return linearSum(data, n−1) + data[n−1];
}
```

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

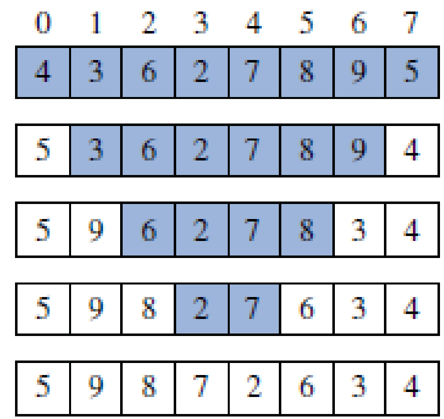


REVERSING AN ARRAY

Problem: Reverse the *n* elements of an array, so that the first element becomes the last, the second element becomes second to the last, and so on.

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)
return
```



Terminates after a total of (1 + $floor\left(\frac{n}{2}\right)$) recursive calls. Because each call involves a constant amount of work, the entire process **runs in *O*(*n*) time**.

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)

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

RECURSIVE ALGORITHMS FOR COMPUTING POWERS

Problem: Raise a number *x* to an arbitrary nonnegative integer *n*.

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

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

```java
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else
    return x * power(x, n−1);
}
```

- 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

OBSERVATION: We consider the expression $\left(x^k\right)^2$, where $k = floor(\frac{n}{2})$.

When n is odd, $floor\left(\frac{n}{2}\right) = \frac{n-1}{2}$, $\left(x^k\right)^2 = x^{n-1}$ and therefore $x^n = \left(x^k\right)^2 * x$.

When n is even, $floor(\frac{n}{2}) = \frac{n}{2}$ and therefore $\left(x^k\right)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$.

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

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \left\lfloor \frac{n}{2} \right\rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- 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

$O(\log n)$ recursive calls.

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 = Power(x, (n - 1)/ 2)
    return x · y · y
  else
    y = Power(x, n/ 2)
    return y · y

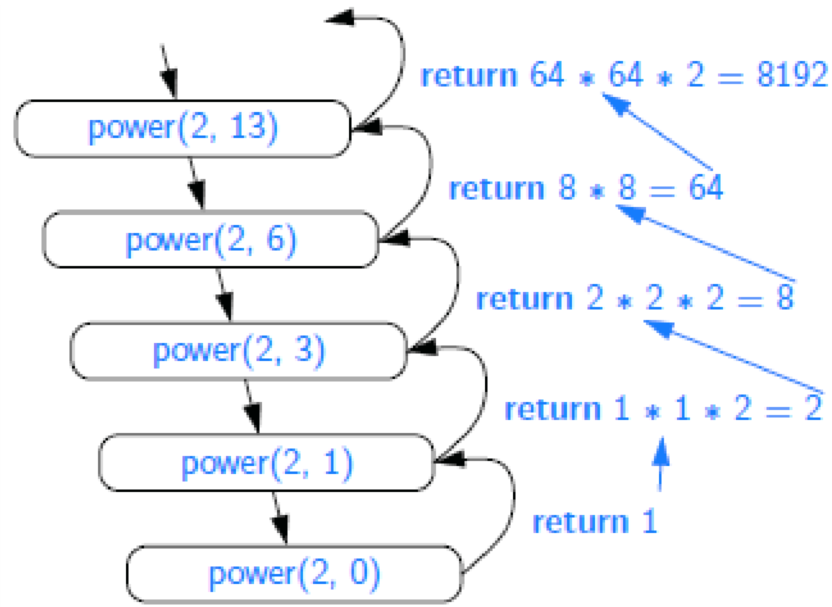
Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

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

RECURSIVE SQUARING METHOD

```
public static double power(double x, int n) {
  if (n == 0)
    return 1;
  else {
    double partial = power(x, n/2);
    double result = partial * partial;
    if (n % 2 == 1)
      result *= x;
    return result;
  }
}
```

Example: power(2, 13)



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.

```
---- 0          ----- 0          --- 0
-               -                -
--              --               --
-               -                -
---             ---              --- 1
-               -                -
--              --               --
-               -                -
---- 1          ----             --- 2
-               -                -
--              --               --
-               -                -
---             ---              --- 3
-               -
--              --
-               -
---- 2          ----- 1
```

EXAMPLE: DRAWING ENGLISH RULER

- Print the ticks and numbers like an English ruler:

2-inch ruler with major tick length 4;

```
---- 0
-
--
-
---
-
--
-
---- 1
-
--
-
---
-
--
-
---- 2
```

1-inch ruler with major tick length 5;

```
----- 0
-
--
-
---
-
--
-
----
-
--
-
---
-
--
-
----- 1
```

3-inch ruler with major tick length 3.

```
--- 0
-
--
-
--- 1
-
--
-
--- 2
-
--
-
--- 3
```

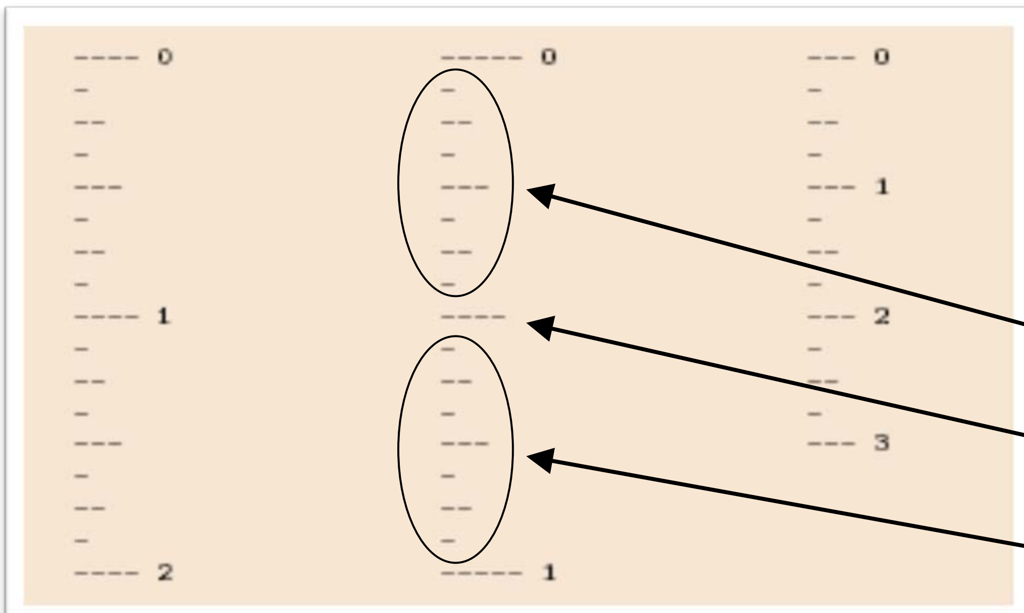
Slide by Matt Stallmann included with permission.

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

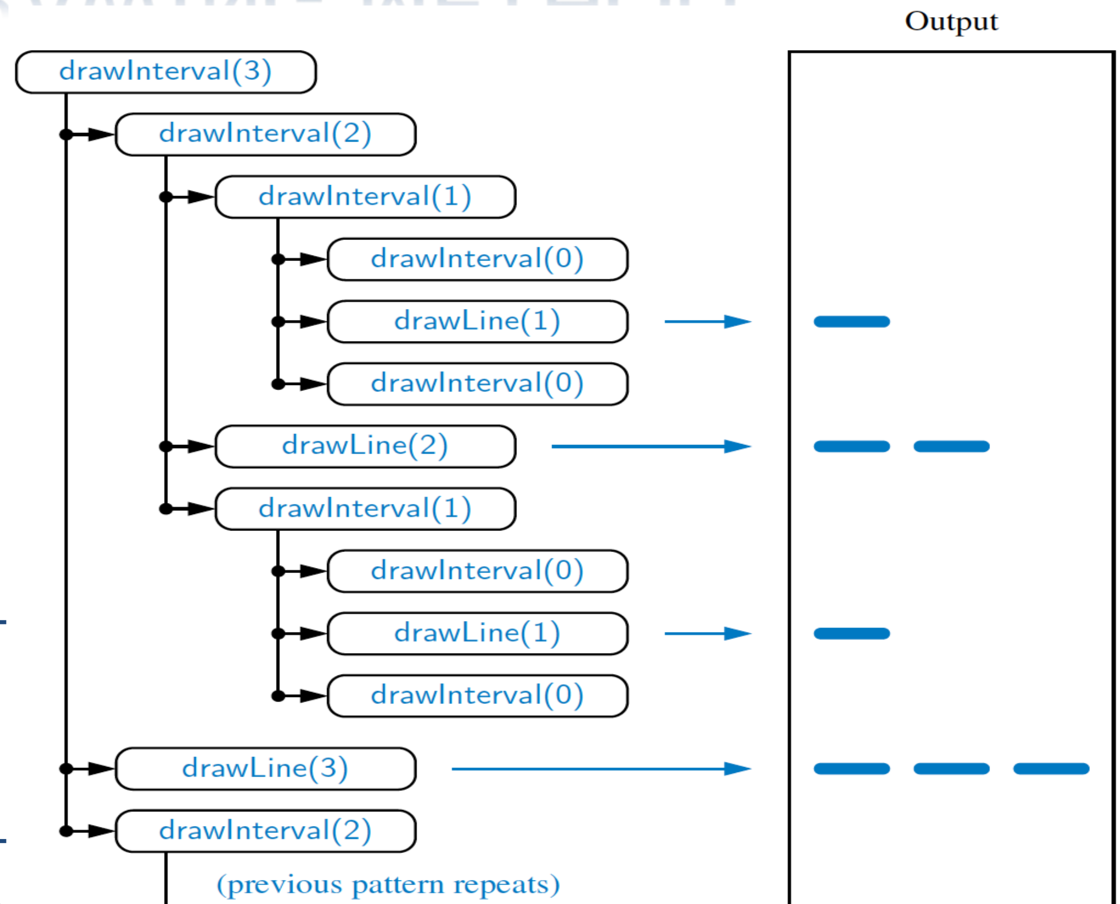
drawInterval (length – 1)

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
  - 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

```
/** Draws an English ruler for the given number of inches and major tick length. */
public static void drawRuler(int nInches, int majorLength) {
  drawLine(majorLength, 0);                    // draw inch 0 line and label
  for (int j = 1; j <= nInches; j++) {
    drawInterval(majorLength − 1);             // draw interior ticks for inch
    drawLine(majorLength, j);                  // draw inch j line and label
  }
}
private static void drawInterval(int centralLength) {
  if (centralLength >= 1) {                    // otherwise, do nothing
    drawInterval(centralLength − 1);           // recursively draw top interval
    drawLine(centralLength);                   // draw center tick line (without label)
    drawInterval(centralLength − 1);           // recursively draw bottom interval
  }
}
private static void drawLine(int tickLength, int tickLabel) {
  for (int j = 0; j < tickLength; j++)
    System.out.print("-");
  if (tickLabel >= 0)
    System.out.print(" " + tickLabel);
  System.out.print("\n");
}
/** Draws a line with the given tick length (but no label). */
private static void drawLine(int tickLength) {
  drawLine(tickLength, −1);
}
```

Note the two recursive calls

ANOTHER BINARY RECURSIVE METHOD

- **Problem:** Add all the numbers in an integer array A
- **Solution strategy:** Recursively compute the sum of the first half, and the sum of the second half, and add those sums together.

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:

BinarySum(data, 0, 8)

Space: binarySum uses $O(\log n)$ amount of additional space, which is a big improvement over the $O(n)$ space used by the linearSum method.

Time : However, the running time of is $O(n)$.

MULTIPLE RECURSION

Multiple recursion : a process in which a method may make more than two recursive calls.

- Motivating example: summation puzzles
  - *pot* + *pan* = *bib*
  - *dog* + *cat* = *pig*
  - *boy* + *girl* = *baby*

To solve such a puzzle, we need to assign a unique digit (that is, 0,1, . . . ,9) to each letter in the equation, in order to make the equation true.

- Multiple recursion:
  - makes potentially many recursive calls
  - not just one or two

ALGORITHM FOR MULTIPLE RECURSION

If the number of possible configurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one.

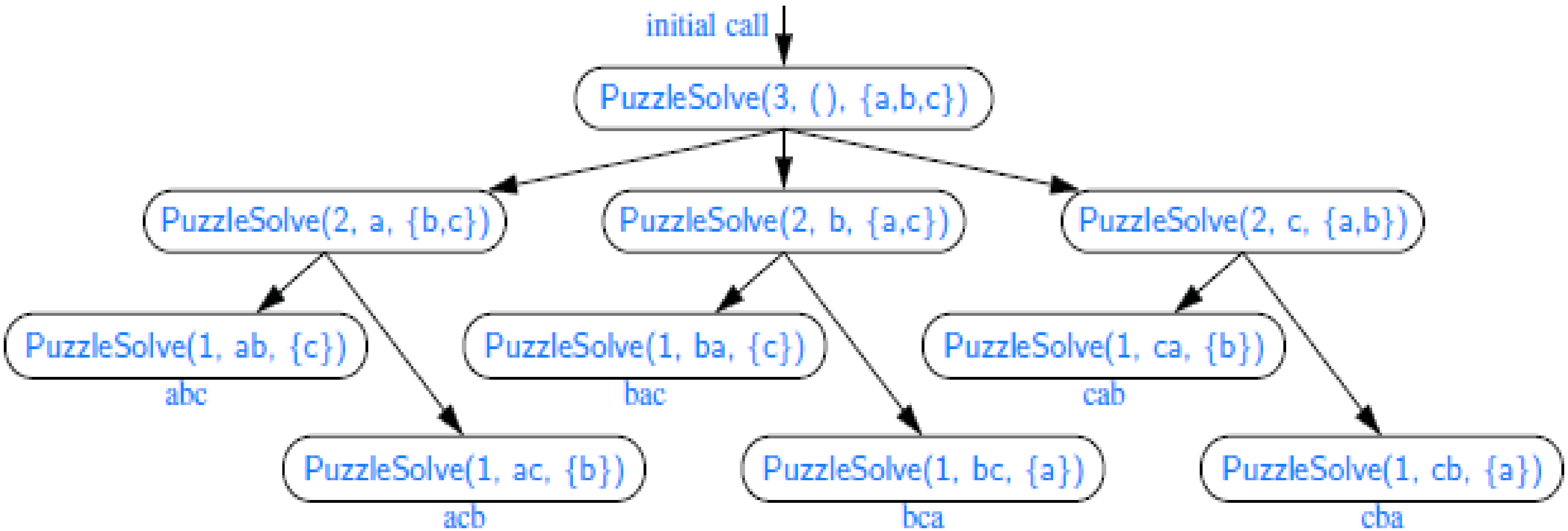
Algorithm PuzzleSolve(k, S, U):

Input: An integer k, sequence S, and set U

Output: An enumeration of all k-length extensions to S using elements in U without repetitions

```
for each e in U do
    Add e to the end of S
    Remove e from U                              {e is now being used}
    if k == 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            add S to output                      {a solution}
    else
        PuzzleSolve(k − 1, S, U)                 {a recursive call}
    Remove e from the end of S
    Add e back to U                              {e is now considered as unused}
```

Recursion trace for an execution of PuzzleSolve(3, *S*, *U*), where *S* is empty and $U = \{a, b, c\}$.

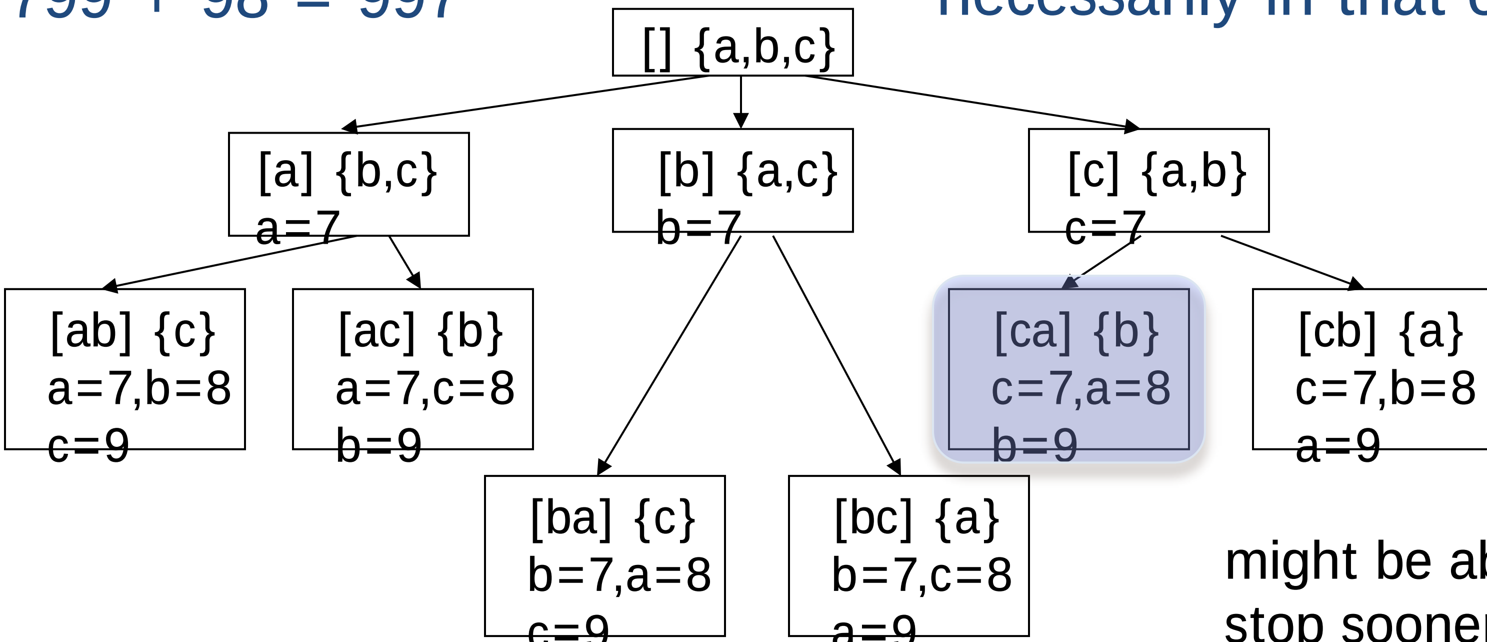


EXAMPLE

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

PITFALLS OF RECURSION

- Recursion can easily be misused in various ways.

ELEMENT UNIQUENESS PROBLEM, REVISITED

RROBLEM: Given an array with *n* elements, are all the elements of that collection are distinct from each other?

```
public static boolean unique1(int[ ] data) {
  int n = data.length;
  for (int j=0; j < n−1; j++)
    for (int k=j+1; k < n; k++)
      if (data[j] == data[k])
        return false;
  return true;
}
```

$O(n^2)$

brute force method

```
public static boolean unique2(int[ ] data) {
  int n = data.length;
  int[ ] temp = Arrays.copyOf(data, n);
  Arrays.sort(temp);
  for (int j=0; j < n−1; j++)
    if (temp[j] == temp[j+1])
      return false;
  return true;
}
```

$O(n\log n)$

sorting based

```
/** Returns true if there are no duplicate values from data[low] through data[high].*/
public static boolean unique3(int[ ] data, int low, int high) {
  if (low >= high) return true;                          // at most one item
  else if (!unique3(data, low, high−1)) return false;    // duplicate in first n−1
  else if (!unique3(data, low+1, high)) return false;    // duplicate in last n−1
  else return (data[low] != data[high]);                 // do first and last differ?
}
```

$O(2^n)$

Recursive unique3 for testing element uniqueness

ANALYSIS OF RECURSIVE UNIQUE3

- unique3 is a terribly inefficient use of recursion!!

Let n denote the number of entries under consideration:

$$n = 1 + \text{high} - \text{low}$$

Base case (n = 1): running time of unique3 is $O(1)$ since there are no recursive calls and the nonrecursive part of each call uses $O(1)$ time.

General case (n>1): a single call to unique3 for a problem of size n may result in two recursive calls on problems of size $n-1$, and so on. Thus, in the worst case, the total number of method calls is given by the geometric summation:

$$1 + 2 + 4 + \cdots + 2^{n-1} = 2^n - 1 = \boxed{O(2^n)}$$

COMPUTING FIBONACCI NUMBERS

Fibonacci numbers are defined recursively:

$$F_0 = 0$$

$$F_1 = 1$$

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

```java
/** Returns the nth Fibonacci number (inefficiently). */
public static long fibonacciBad(int n) {
  if (n <= 1)
    return n;
  else
    return fibonacciBad(n−2) + fibonacciBad(n−1);
}
```

Recursive algorithm (inefficiently):

Algorithm BinaryFib(k):
 Input: Nonnegative integer k
 Output: The kth 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) = LinearFibonacci(k – 1)
  return (i +j, i)

Each invocation
1) makes only one recursive call and
2) decreases the argument *n* by 1.

```
/** Returns array containing the pair of Fibonacci numbers, F(n) and F(n−1). */
public static long[ ] fibonacciGood(int n) {
  if (n <= 1) {
    long[ ] answer = {n, 0};
    return answer;
  } else {
    long[ ] temp = fibonacciGood(n − 1);              // returns {F_{n−1}, F_{n−2}}
    long[ ] answer = {temp[0] + temp[1], temp[0]};    // we want {F_n, F_{n−1}}
    return answer;
  }
}
```

runs in $O(n)$ time.