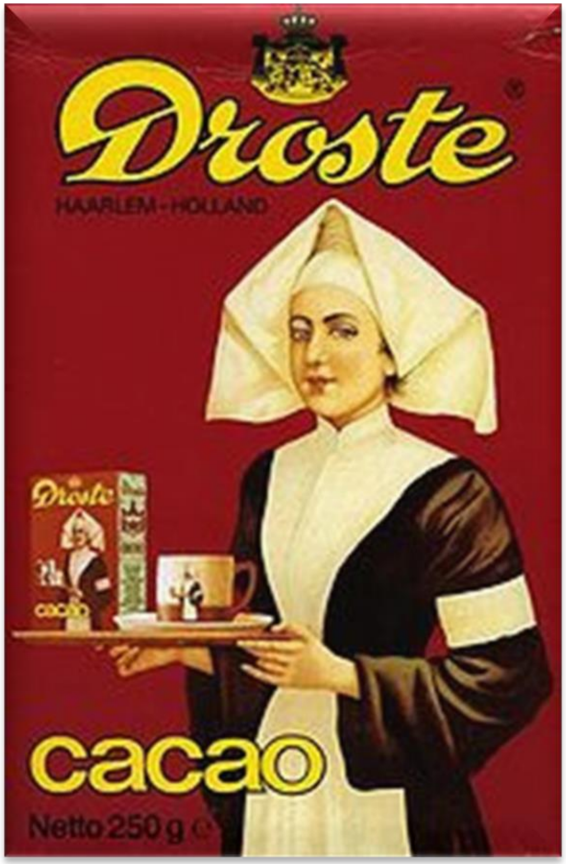
Recursion



Instructor: Andy Mirzaian

The Recursion Pattern

- **Recursion:** when a method calls itself
- Classic example: the factorial function
  n! = 1*2*3· ⋯ · (n-1)*n
- Recursive definition: $f(n) = \begin{cases} 1 & if\ n = 0 \\ n * f(n-1) & otherwise \end{cases}$

```
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

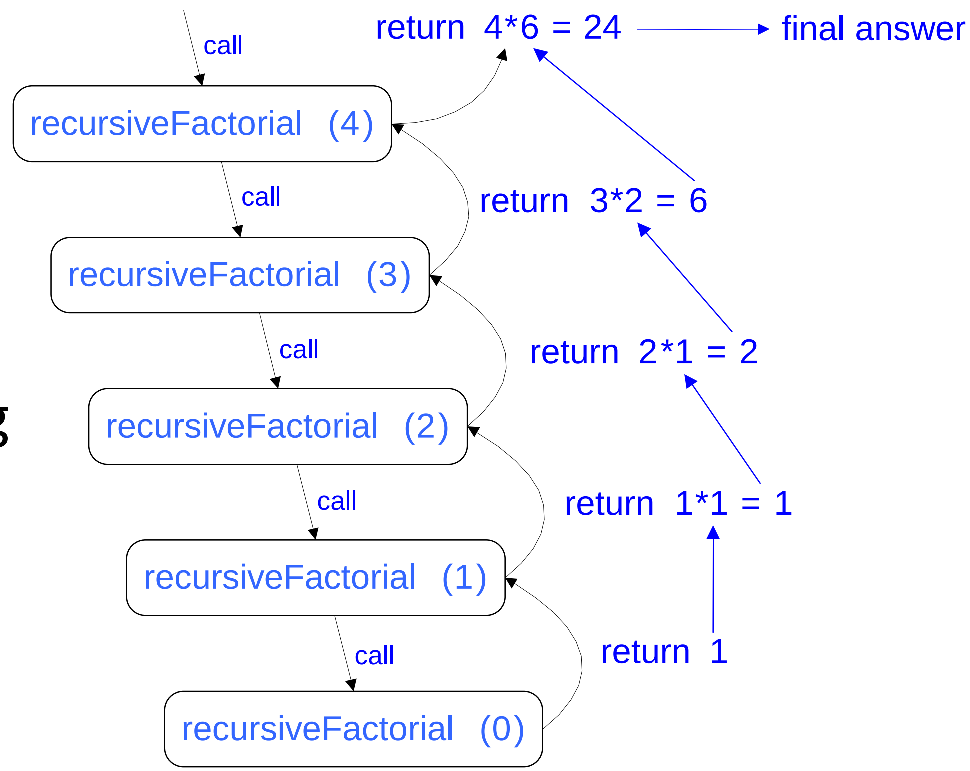
- Base case(s)
  - Values of the input variables for which we perform no recursive calls are called base cases (there should be at least one base case).
  - Every possible chain of recursive calls must eventually reach a base case.
- Recursive calls
  - Calls to the current method.
  - Each recursive 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:



Example: English Ruler

Print the ticks and numbers like an English ruler:

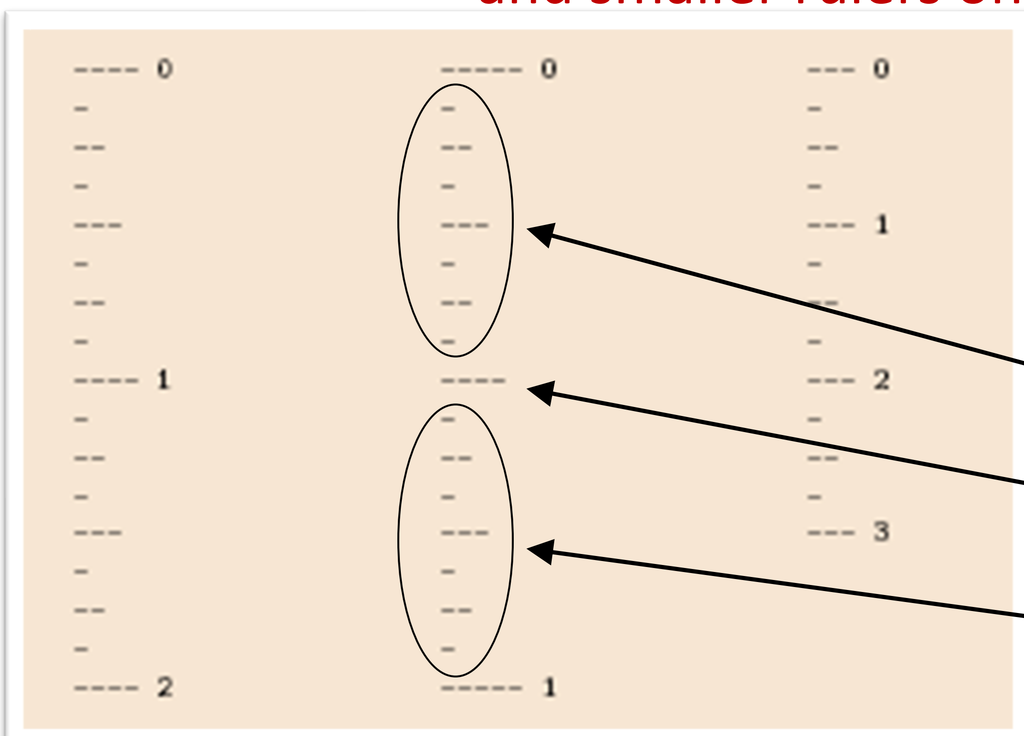
```
---- 0          ----- 0         --- 0
-               -               -
--              --              --
-               -               -
---             ---             --- 1
-               -               -
--              --              --
-               -               -
---- 1        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

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**

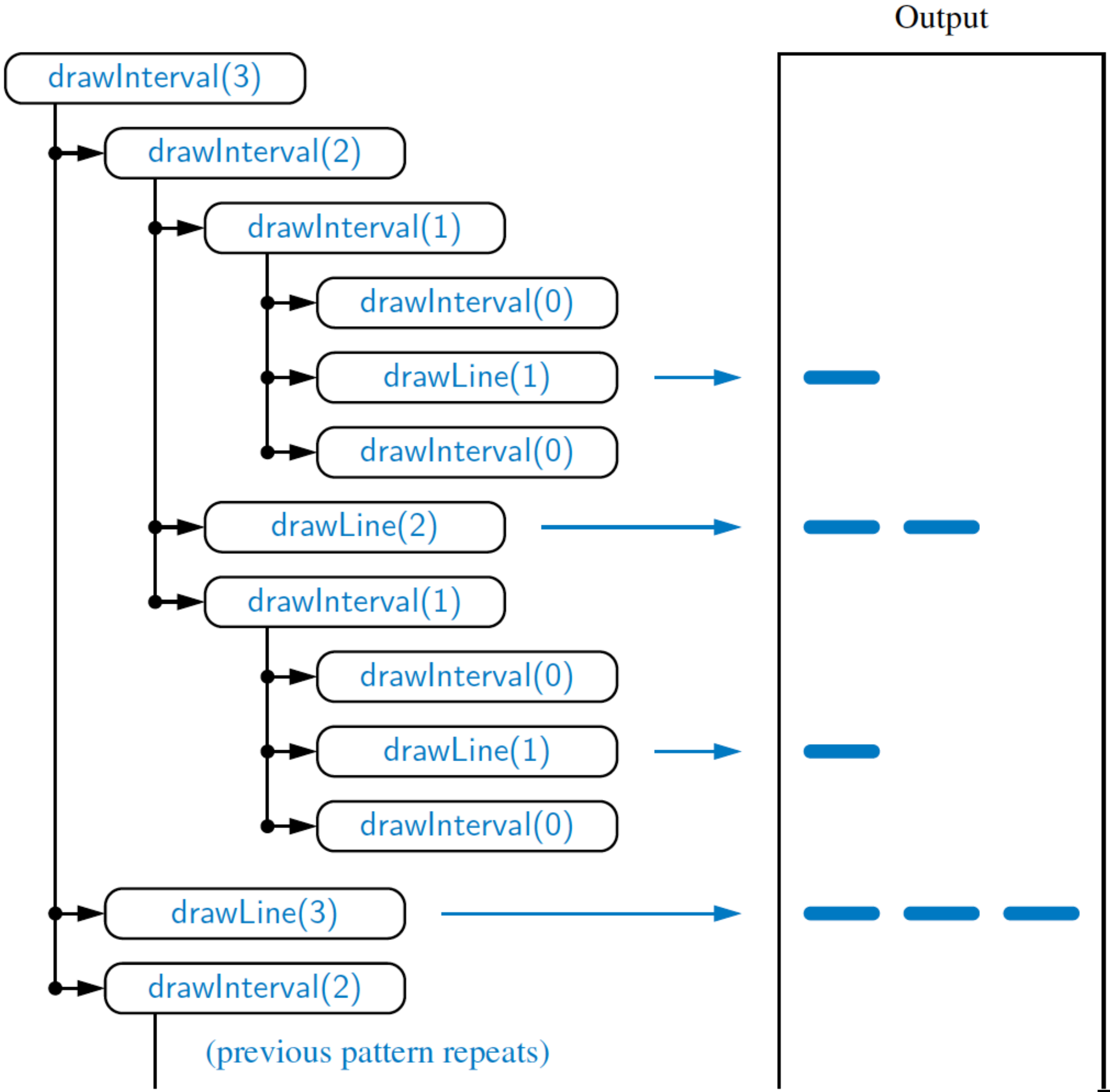
drawInterval (length - 1)

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



The Recursive Method

```
/** 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

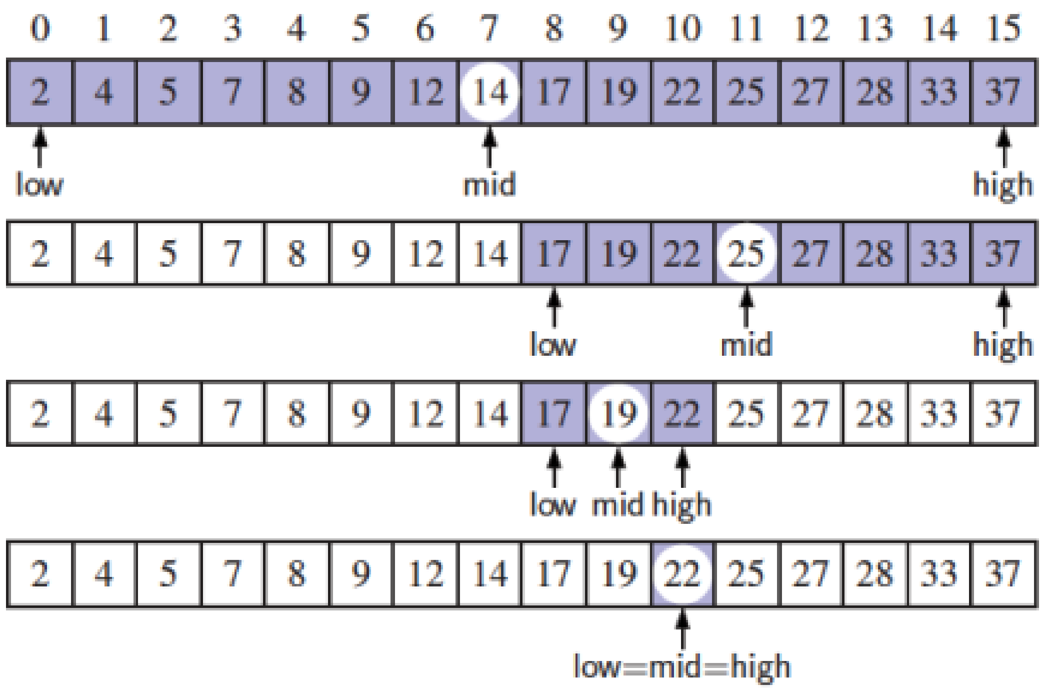
Binary Search

Search for an integer in an ordered indexed list

```java
/**
 * 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
  }
}
```

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.



Analyzing Binary Search

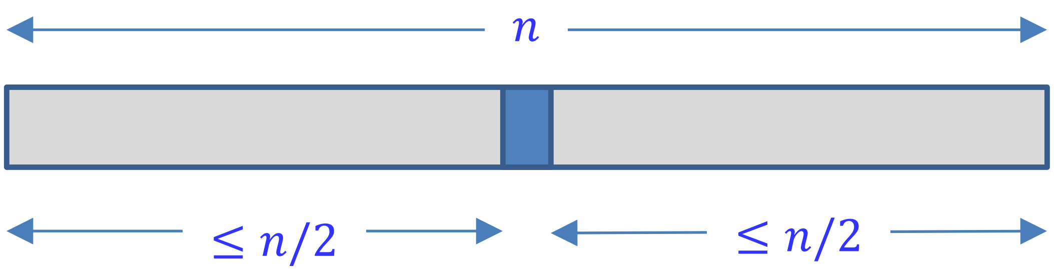
- Runs in O(log n) time:
  - The remaining portion of the list is of size **high – low + 1**.
  - After one comparison, this becomes one of the following:

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

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

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

Analyzing Binary Search by recurrence formula



- Recurrence: $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c & if\ n > 1 \\ c & if\ n \leq 1 \end{cases}$

Solve the recurrence

- Recurrence: $T(n) = \begin{cases} T\left(\frac{n}{2}\right) + c & if\ n > 1 \\ c & if\ n \leq 1 \end{cases}$
- Solution:

$$
\begin{aligned}
T(n) &= T(n/2) + \mathrm{c} \\
&= T(n/2^2) + c + c \quad = T(n/2^2) + 2c \\
&= T(n/2^3) + c + 2c \quad = T(n/2^3) + 3c \\
&\ \vdots \\
&= T\left(n/2^k\right) + kc \\
&= T(n/n) + c\log n \\
&= c + c\log n
\end{aligned}
$$

(now plug in $k = \log n\,, 2^k = n$)

Therefore, $T(n)$ is $O(\log n)$.

Linear Recursion

- Test for base cases
  - Test for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case. Each base case should be handled non-recursively.
- 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

$0 \leq n \leq |A|$

Output:

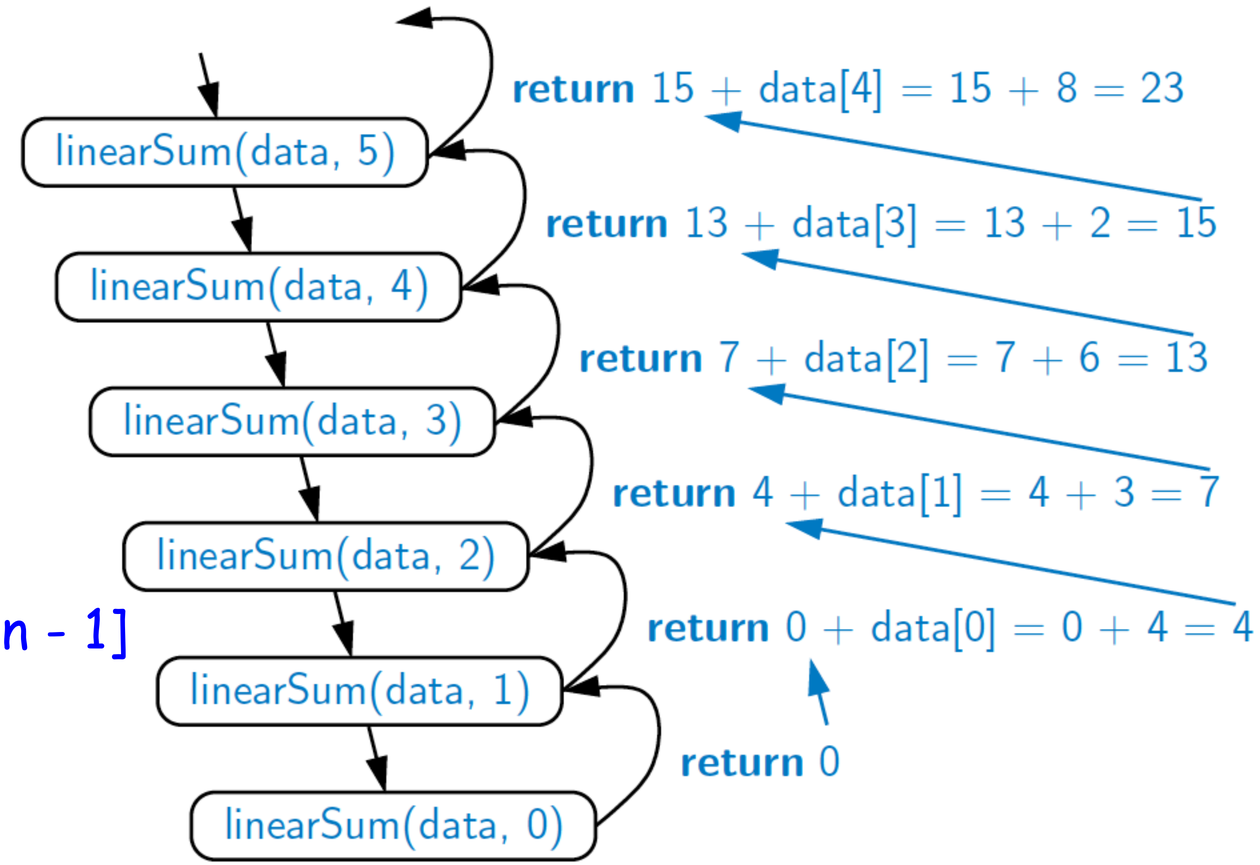
Sum of the first n integers in A

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]



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; i.e., reverse the sub-array A[i..j]

```
if  i < j  then                    // what are the base cases?
     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)

```
/** 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
  }
}
```

Analyze by recurrence

- Recurrence: $T(n) = \begin{cases} T(n-2) + c & if\ n > 1 \\ c & if\ n \leq 1 \end{cases}$

- Solution:

$$
\begin{aligned}
T(n) &= T(n-2) + \mathrm{c} \\
&= T(n-4) + c + c \\
&= T(n-6) + 3c \\
&= T(\,n-8) + 4c \\
&\vdots \\
&= T(n-2k) + kc \\
&= T(0) + cn/2 \\
&= c + cn/2
\end{aligned}
$$

(now plug in $k = n/2$)

Therefore, $T(n)$ is $O(n)$.

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 valid indices i & j

Output: sub-array A[i..j] reversed

while $i < j$ **do**

Swap A[i] and A[j]

$i \leftarrow i+1$, $j \leftarrow j-1$

end

How to Eliminate Tail Recursion

```
Algorithm reverseArray(A, i, j)
PreCond:   Array A and in-range indices i and j
PostCond:  Sub-array A[i..j] is reversed
if  i < j  then
        Swap A[i] and A[j]
        reverseArray(A, i + 1, j - 1)   // tail recursion
```

```
Algorithm reverseArray(A, i, j) // tail recursion removed
PreCond:   Array A and in-range indices i and j
PostCond:  Sub-array A[i..j] is reversed
while  i < j  do
        Swap A[i] and A[j]
        i ← i + 1;  j ← j - 1     // update parameters & iterate
```

Computing Powers

- The power function $p(x,n) = x^n$ $\quad (x \neq 0,\ int\ n \geq 0)$ can be defined recursively:

$$p(x,n) = \begin{cases} 1 & if\ n = 0 \\ x * p(x, n-1) & otherwise \end{cases}$$

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

Recursive Squaring

- A more efficient linearly recursive algorithm by using repeated squaring:
- $n = 2\lfloor n/2 \rfloor + (n \bmod 2) \implies x^n = (x^2)^{\left\lfloor \frac{n}{2} \right\rfloor} * x^{(n \bmod 2)}$

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

- **Example:**

$2^{15} = 2 * 4^7 = 2 * 4 * 16^3 = 2 * 4 * 16 * (256)^1 = 2 * 4 * 16 * 256 * (\ldots)^0$
$= 2 * 4 * 16 * 256 * 1 = 32{,}768$

Recursive Squaring Method

Algorithm Power(x, n)   // $O(\log n)$ time

Input: A number $x > 0$ and integer $n \geq 0$
Output: The value x^n

```
if n = 0 then return 1
y ← Power(x∗x , ⌊n/2⌋)
if n is odd then y ← y ∗ x
return y
```

Analysis

Algorithm Power(x, n) // O(log n) time

Input: A number $x > 0$ and integer $n \geq 0$
Output: The value x^n

```
if  n = 0  then return 1
y ← Power(x * x , ⌊n/2⌋)
if n is odd  then  y ← y * x
return y
```

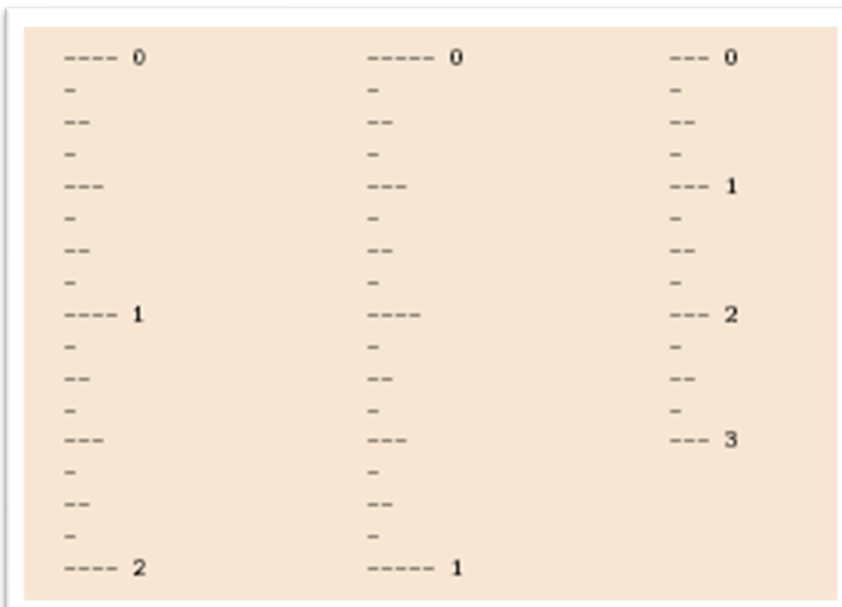
$$T(n) = T(n/2) + O(1)$$
$$\Downarrow$$
$$T(n) = O(\log n).$$

Each time we make a recursive call, we halve the 2nd argument.
Hence, we make log n recursive calls.
With each call we do O(1) work.
So, this method runs in O(log n) time.

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

Binary Recursion

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



Another Binary Recursive Method

Problem: Find element sum of an integer array A.

Algorithm BinarySum(A, i , n)

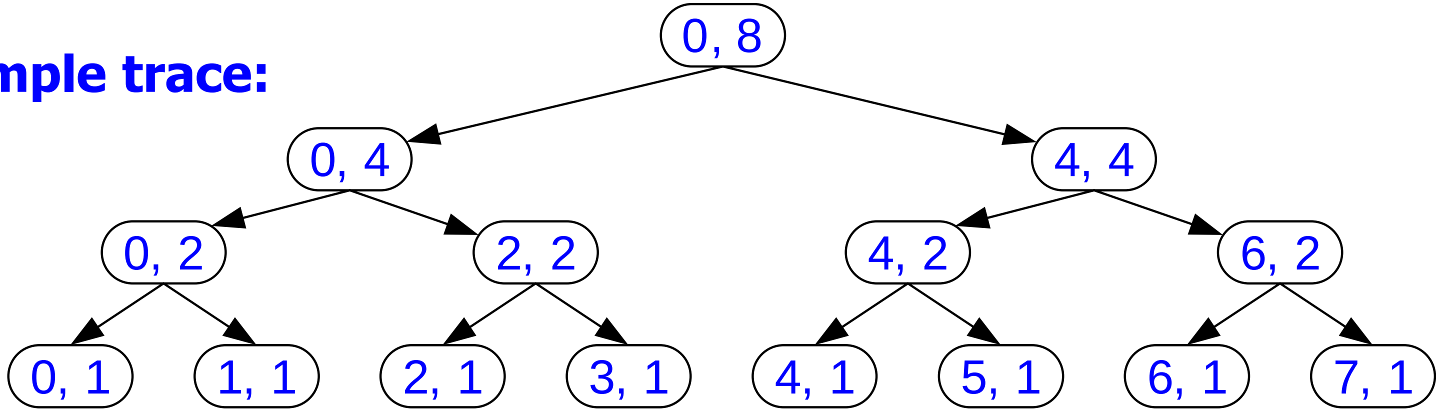
Input: An array A and integers i and n

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

if $n = 1$ **then return** A[i]

return BinarySum(A, i, $\lfloor n/2 \rfloor$) + BinarySum(A, i + $\lfloor n/2 \rfloor$, $\lceil n/2 \rceil$)

Example trace:



Fibonacci Numbers

Fibonacci numbers are defined recursively:

$$F_k = \begin{cases} k & \text{for } k = 0,1 \\ F_{k-1} + F_{k-2} & \text{for } k \geq 2 \end{cases}$$

k	0	1	2	3	4	5	6	7	8	9	10	11	12	…
F_k	0	1	1	2	3	5	8	13	21	34	55	89	144	…

$$1.4^k \leq \left(\sqrt{2}\right)^k = 2^{k/2} \leq \quad F_k \quad \leq 2^k$$

$$=$$

$$2\,F_{k-2} \leq \quad F_{k-1} + F_{k-2} \quad \leq 2\,F_{k-1}$$

Fibonacci Exponential Growth

- **Guess** an exponential solution for the recurrence $F_k = F_{k-1} + F_{k-2}$ first:
  $F_k = r^k$ (r is a constant to be determined)
- **Verify** the guess by plugging it into the recurrence:
  $r^k = r^{k-1} + r^{k-2} \implies r^2 = r + 1$
- This quadratic has two roots:
  $$\varphi = \frac{1+\sqrt{5}}{2} \cong +1.618 \quad \text{(the golden ratio)}$$
  $$\hat{\varphi} = \frac{1-\sqrt{5}}{2} \cong -0.618$$
- Any linear combination of these two solutions also satisfies the recurrence:
  $F_k = a\,\varphi^k + b\,\hat{\varphi}^k$
- Find constants a and b by the two boundary conditions: $F_0 = 0, F_1 = 1$:
  $F_k = \frac{1}{\sqrt{5}}\left(\varphi^k - \hat{\varphi}^k\right)$ (the exact solution!)
- Since $|\varphi| > 1$ and $|\hat{\varphi}| < 1$, the last term asymptotically vanishes:
  $F_k = \Theta\left(\varphi^k\right)$ (exponential growth)

Computing Fibonacci Numbers

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k)

Input: Nonnegative integer k

Output: The k^{th} Fibonacci number F_k

if $k \leq 1$

then return k

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

end

Analysis

- Let N_k be the # of elementary steps by BinaryFib(k)
- $N_k = N_{k-1} + N_{k-2} + c$ for some constant c (e.g., c = 4)
- So, $(\mathrm{c} + N_k) = (c + N_{k-1}) + (c + N_{k-2})$

  i.e., $(\mathrm{c} + N_k)$ behaves like F_k itself.
- Using this & induction, we can show

$$N_k = \Theta(F_k) = \Theta\left(\varphi^k\right) \cong \Theta\left(1.618^k\right)$$

- ***Running time is exponential in magnitude of k !!!***

A Better Fibonacci Algorithm

- Use linear recursion with stronger post-condition

```
Algorithm LinearFibonacci(k)
    Input:   A positive integer k
    Output:  Pair of Fibonacci numbers (F_k , F_k-1)

    if  k = 1  then return (1, 0)
    else
            (i , j) ← LinearFibonacci(k - 1)
            return (i + j , i)
end
```

- LinearFibonacci makes k-1 recursive calls. It's **O(k)**.
- Even **O(log k)** is possible with pure integer arithmetic (by "recursive squaring")!!!

Multiple Recursion

- Example 1: Sudoku
- Example 2: assign each **letter** to a decimal **digit**

1. *pot + pan = bib*
2. *dog + cat = pig*
3. *boy + girl = baby*

0 = l = t	*5 = i*
1 = d	*6 = r*
2 = o	*7 = g*
3 = a = c	*8 = b = n*
4 = p	*9 = y*

- Extra requirement considered next:
  map each letter to a distinct digit.

Algorithm for Multiple Recursion

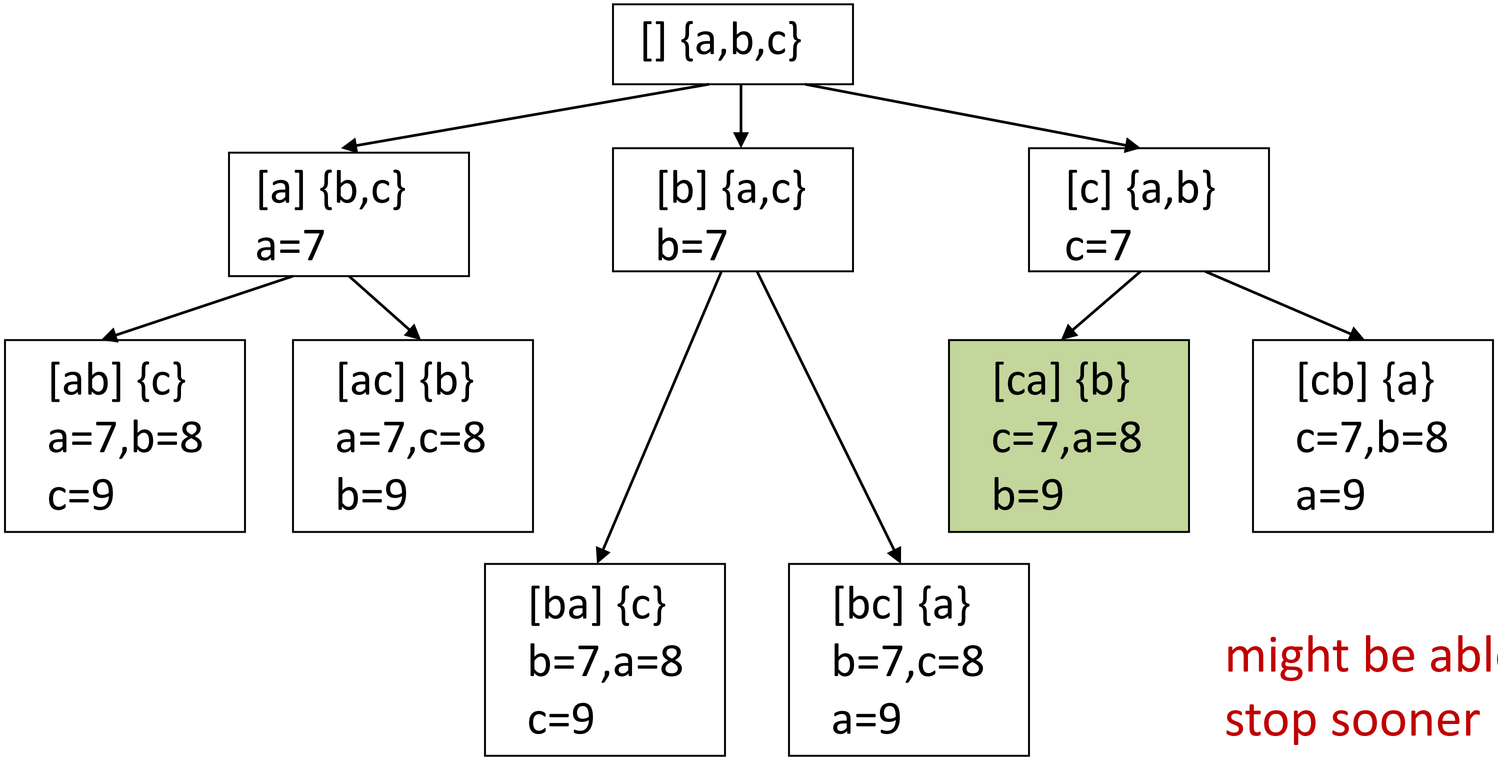
```
Algorithm PuzzleSolve(k, S, U)
 Input:    Integer k, sequence S, and set U (of unused elements)
 Output:  Enumeration of all k-length extensions to S
          using elements in U without repetitions
 for each  e  in U  do
      Remove e from U                  // e is now being used
      Add e to the end of S            // e is selected next in the permutation
      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                  //  undo selection:  e is now unused
      Remove e from the end of S       //  this is called back-tracking
```

Example

cbb + ba = abc

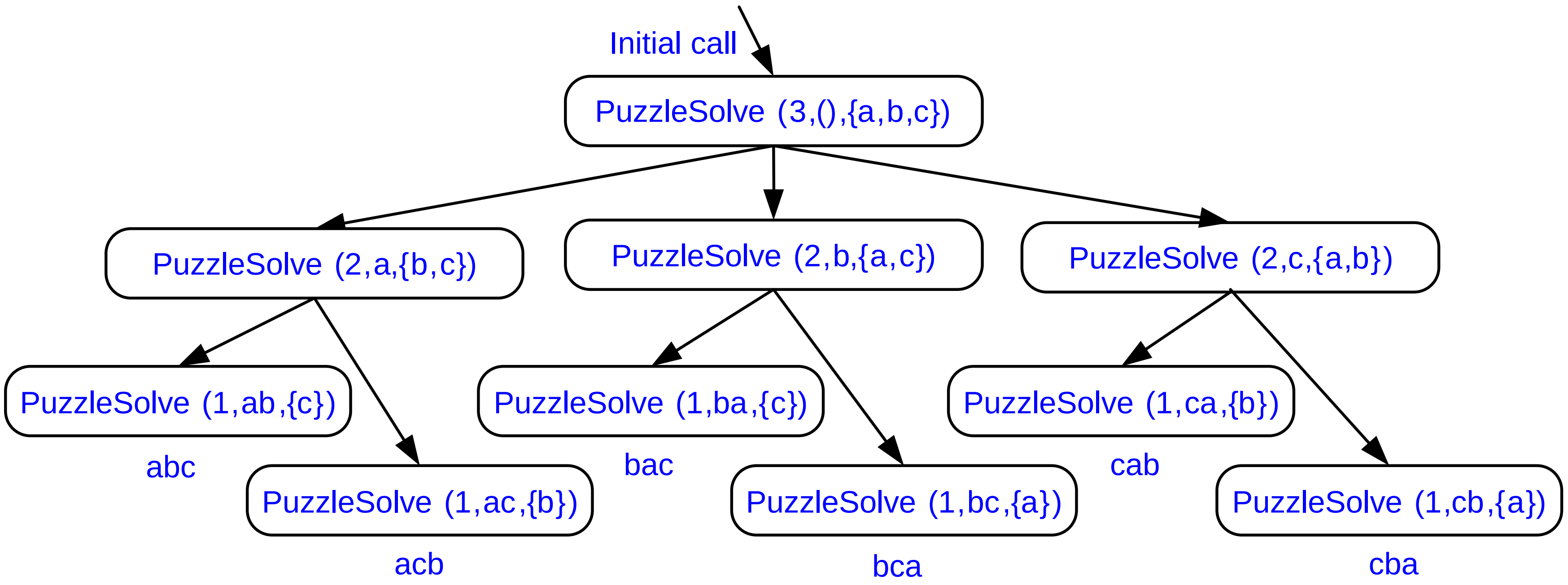
799 + 98 = 897

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



might be able to stop sooner

Visualizing PuzzleSolve



Summary

- Recursion pattern:
  - Base cases
  - Recursive cases
- Visualizing recursion
- Tail recursion
- Recursive squaring
- Linear, binary, and multiple recursion
- Examples & analysis