CS121 Data Structures
Recursion

Monika Stepanyan
mstepanyan@aua.am

Spring 2024

Important Data and Statistics

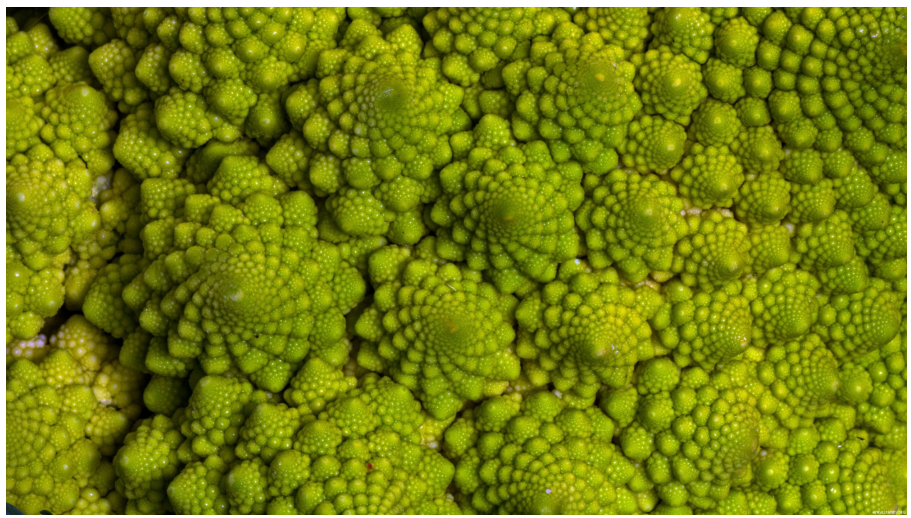
- 28 classes remaining
- 46 days till the Midterm exam I
- 37 days till the Spring Break

Recursion

Recursion is a process to achieve repetition within a computer program

It is a technique by which a method makes one or more calls to itself during execution

By recursion a data structure relies upon smaller instances of the very same type of structure in its representation



Recursion By Example

- the factorial function, $n!$
- binary search
- an English ruler

The Factorial Function

For any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1 & \text{if } n \geq 1 \end{cases}$$

The **recursive definition**

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n \geq 1 \end{cases}$$

recursive definition = one or more base cases (fixed values) + one or more recursive cases (in terms of itself)

The Factorial Function: A Recursive Implementation

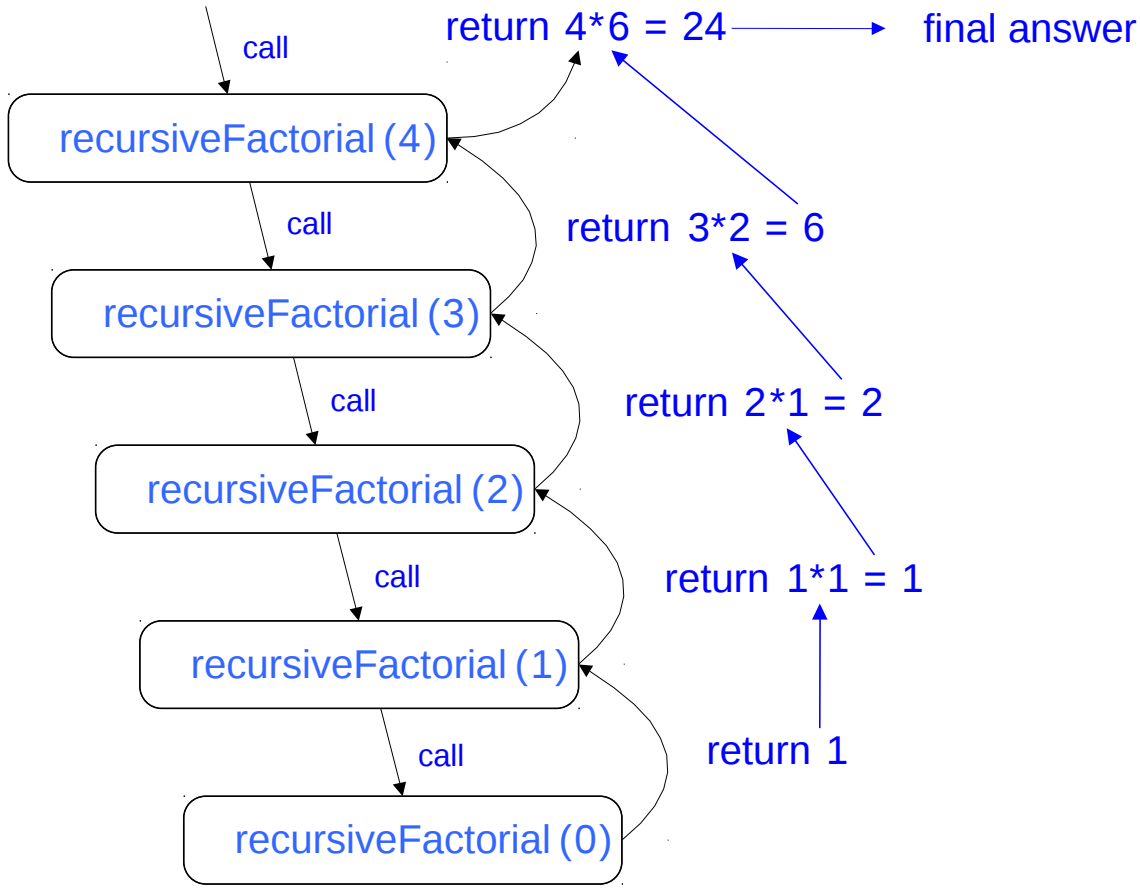
```
1  public static int factorial(int n) throws IllegalArgumentException {
2    if (n < 0)
3      throw new IllegalArgumentException( ); //must be nonnegative
4    else if (n == 0)
5      return 1;                                    // base case
6    else
7      return n * factorial(n-1);          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

Recursion Trace

A box for each recursive call

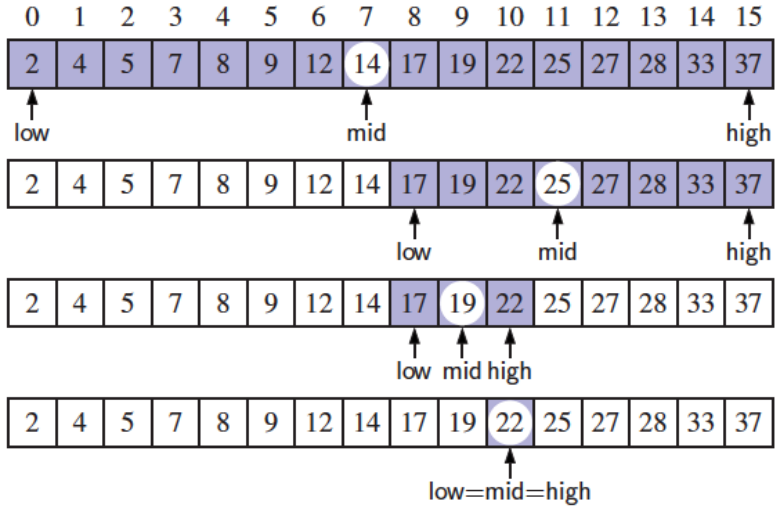
An arrow from each caller to callee

An arrow from each callee to caller showing return value



Binary Search

Binary search is used to efficiently locate a target value within a sorted sequence of n elements stored in an array



We consider three cases:

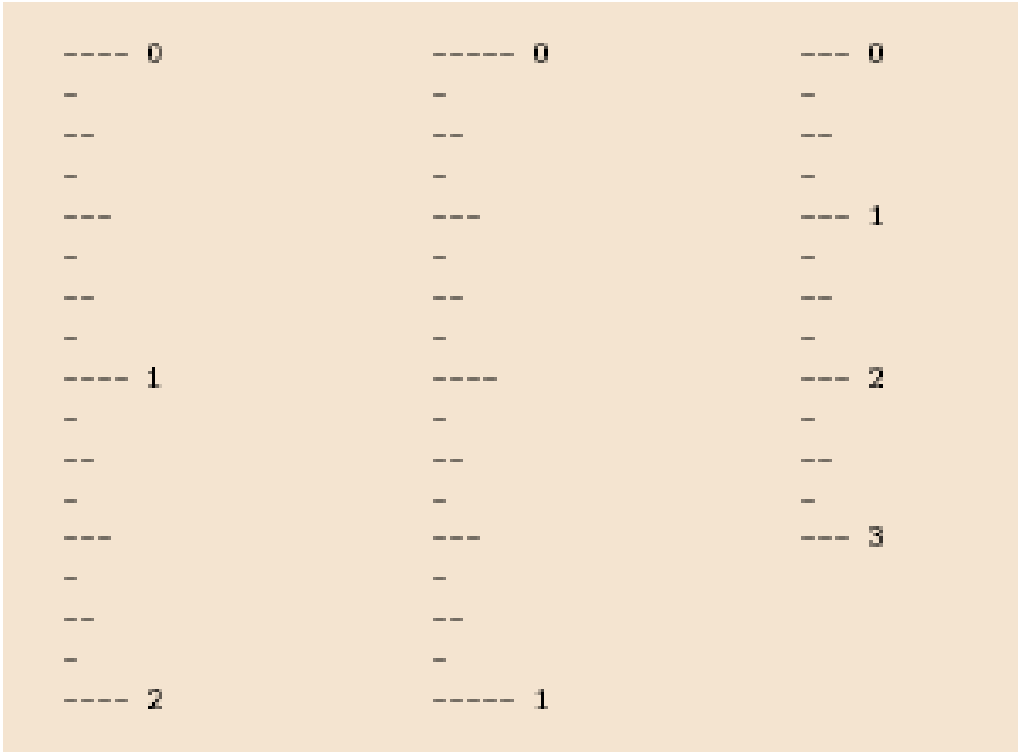
- If the target equals data[mid], then we have found the target.
- If target $<$ data[mid], then we recur on the first half of the sequence.
- If target $>$ data[mid], then we recur on the second half of the sequence.

Binary Search: A Recursive Implementation

```
1  /**
2   * Returns true if the target value is found in the indicated portion of the array.
3   * This search only considers the 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 }
```

Drawing an English Ruler

Major ticks designate whole inches while minor ticks are placed at intervals $\frac{1}{2}$, $\frac{1}{4}$, etc. between two major ticks

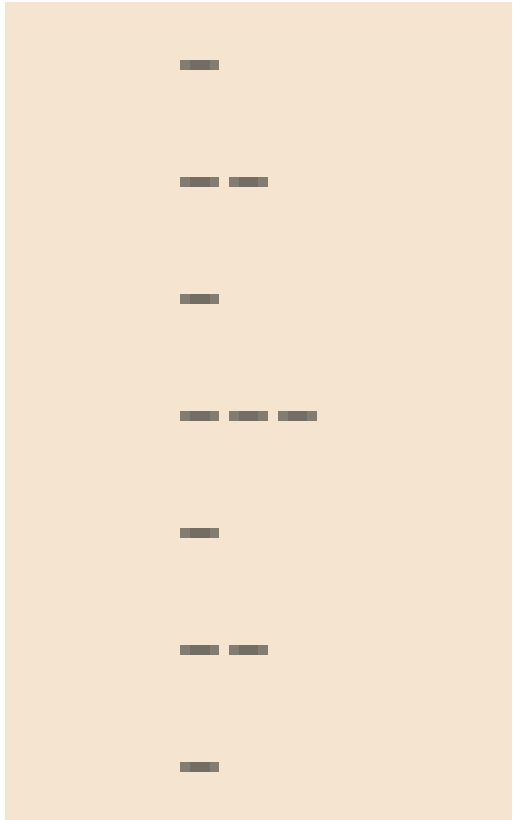


The English ruler pattern is a simple example of a **fractal**, i.e. a shape that has a self-recursive structure at various levels of magnification

Drawing an English Ruler

An interval with a central tick length $L \geq 1$ is composed of:

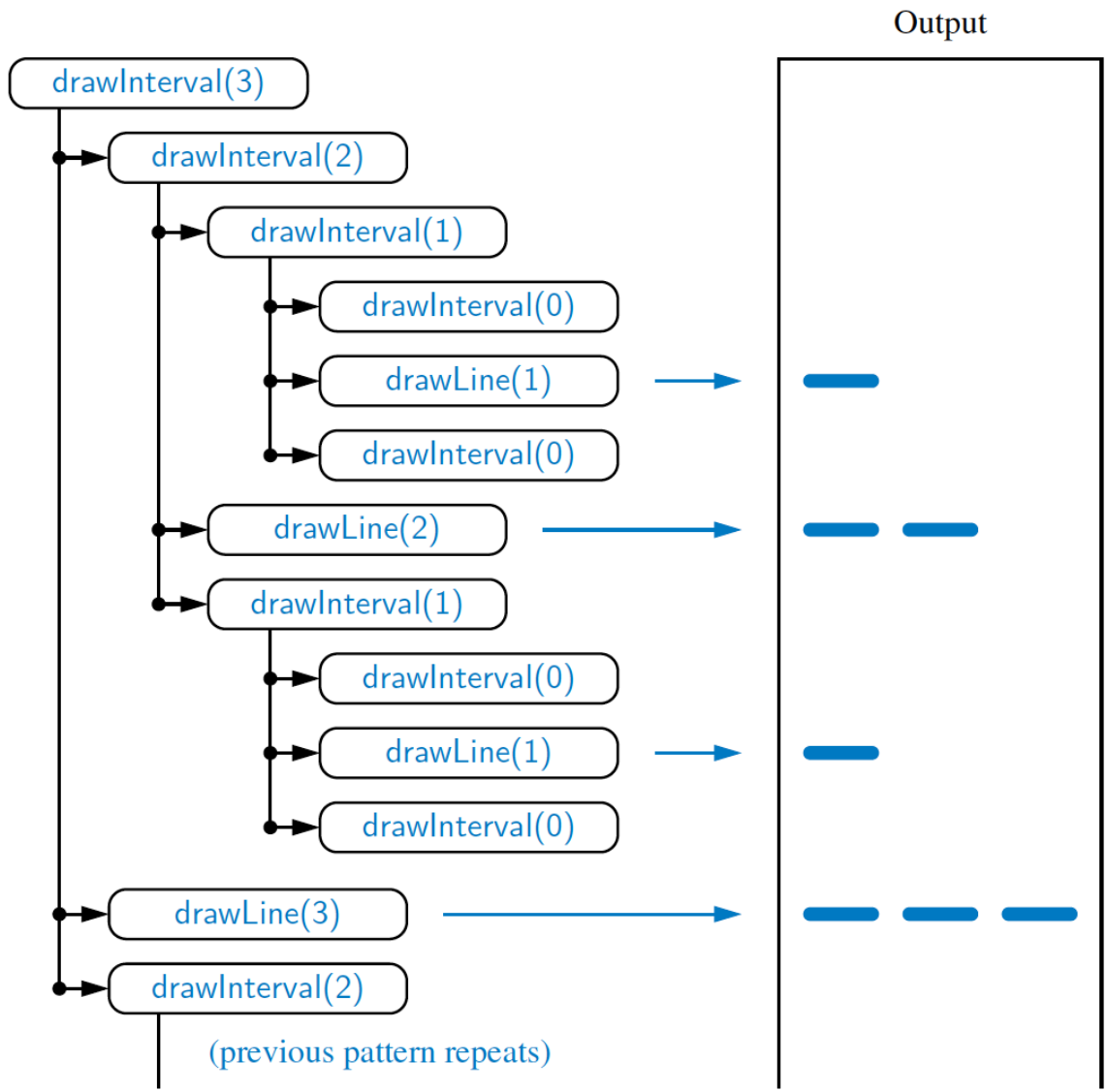
- an interval with a central tick length $L - 1$
- a single tick of length L
- an interval with a central tick length $L - 1$



Recursion Trace for English Ruler

An interval with a central tick length $L \geq 1$ is composed of:

- an interval with a central tick length $L - 1$
- a single tick of length L
- an interval with a central tick length $L - 1$



English Ruler: A Recursive Implementation

```
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 (no 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 /** Draws a line with the given tick length (but no label). */
24 private static void drawLine(int tickLength) {
25     drawLine(tickLength, -1);
26 }
```

The Factorial Function: Efficiency

For each invocation of the method, only account for the number of operations that are performed within the body of that activation. Then take the sum over all activations

A total of $n + 1$ activations (n, $n - 1$, ..., 1, 0)

A constant number of operations in each activation, i.e. $O(1)$

Thus, the overall number of operations is $O(n)$

Best-case running time? Space complexity?

Binary Search: Efficiency

A constant number of operations in each activation, i.e. $O(1)$

Proposition: The binary search algorithm runs in $O(\log n)$ time for a sorted array with n elements.

Justification: With each recursive call the number of candidate elements still to be searched is: $\text{high} - \text{low} + 1$

reduced by at least one-half, i.e.

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

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

After j^{th} call, the number of candidates at most $n/2^j$

$n/2^r < 1$. Thus we have $r = \lfloor \log n \rfloor + 1$, i.e. binary search runs in $O(\log n)$ time.

Drawing an English Ruler: Efficiency

Proposition: For $c \geq 0$, a call to drawInerval(c) results in precisely $2^c - 1$ lines of output.

Justification: A formal proof by *induction*

base case: drawInterval(0) generates no output, and
$2^0 - 1 = 1 - 1 = 0$

induction step: drawInterval(c) prints lines one more (center line) than twice the number generated by drawInterval($c - 1$);
$1 + 2 \times (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$

Drawing an English Ruler: Efficiency

Proposition: For $c \geq 0$, a call to drawInerval(c) results in precisely $2^c - 1$ lines of output.

Justification: A formal proof by *induction*

base case: drawInterval(0) generates no output, and
$2^0 - 1 = 1 - 1 = 0$

induction step: drawInterval(c) prints lines one more (center line) than twice the number generated by drawInterval($c - 1$);
$1 + 2 \times (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$

Hence, the overall number of operations is $\Omega(2^n)$

By further analysis, it is also $O(2^n)$; thus it is $\Theta(2^n)$

Types of Recursion

linear recursion a recursive call starts at most one other

binary recursion a recursive call may start two others

multiple recursion a recursive call may start three or more others

Linear Recursion

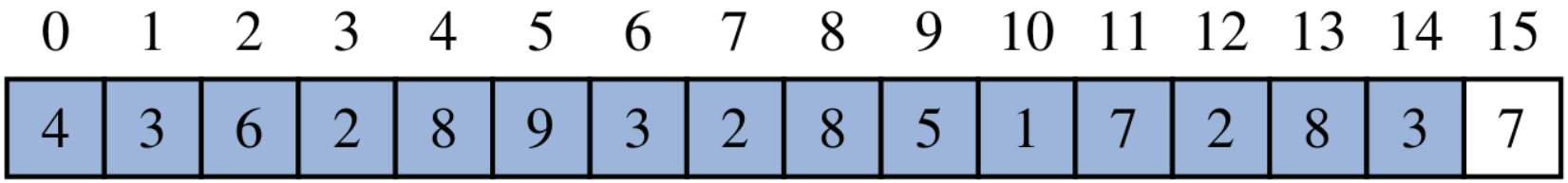
Examples we have seen?

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

Examples of linear recursion with non-linear running time?

Sum of Array Elements

We want to compute the sum of an array of n integers using recursion



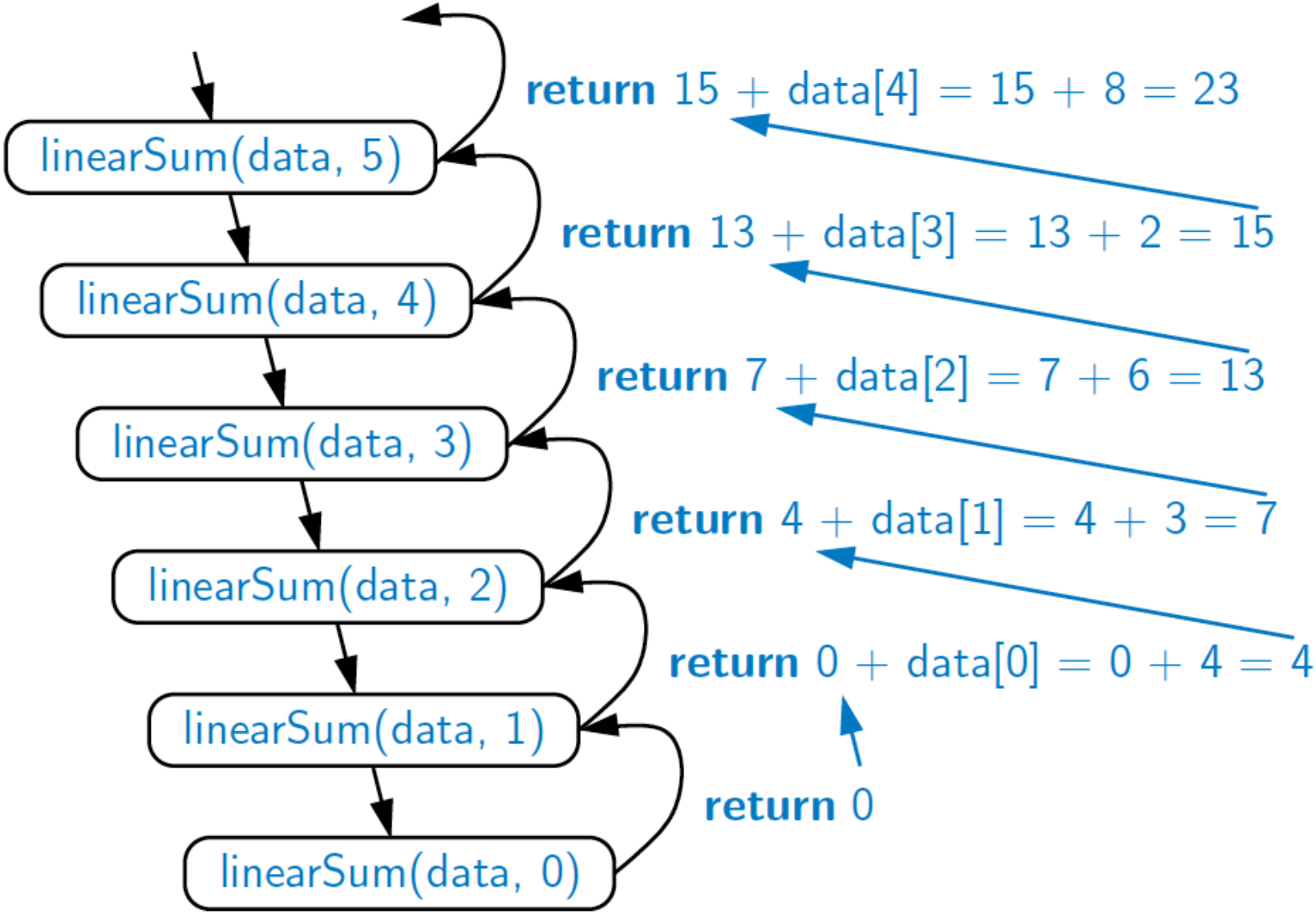
If $n = 0$ the sum is 0; for $n > 0$, add the last number to the sum of the first $n - 1$

Sum of Array Elements: A Recursive Implementation

```
1  /** Returns the sum of the first n integers of the given array. */
2  public static int linearSum(int[ ] data, int n) {
3    if (n == 0)
4      return 0;
5    else
6      return linearSum(data, n-1) + data[n-1];
7  }
```

Sum of Array Elements: Recursion Trace

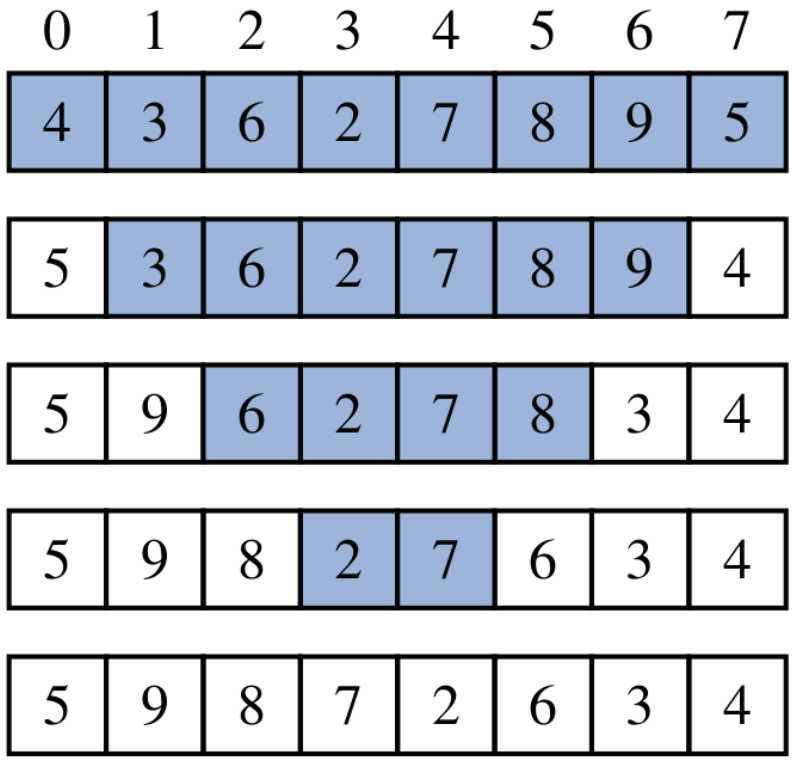
$data = [4, 3, 6, 2, 8]$



time $O(n)$; memory $O(n)$

Reversing a Sequence

We want to reverse the n elements of an array



swap the first and last elements and so on

Reversing a Sequence: A Recursive Implementation

```
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  }
```

time $O(n)$; memory $O(n)$

Computing Powers

We want to raise a number x to an arbitrary nonnegative integer n, i.e. $power(x, n) = x^n$

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

since $x^n = x \cdot x^{n-1}$ for $n > 0$

Computing Powers: A Recursive Implementation

```
1  /** Computes the value of x raised to the nth power,
2      for nonnegative integer n. */
3  public static double power(double x, int n) {
4    if (n == 0)
5      return 1;
6    else
7      return x * power(x, n-1);
8  }
```

time $O(n)$; memory $O(n)$

Computing Powers: A Faster Approach

We want to raise a number x to an arbitrary nonnegative integer n, i.e. $power(x, n) = x^n$

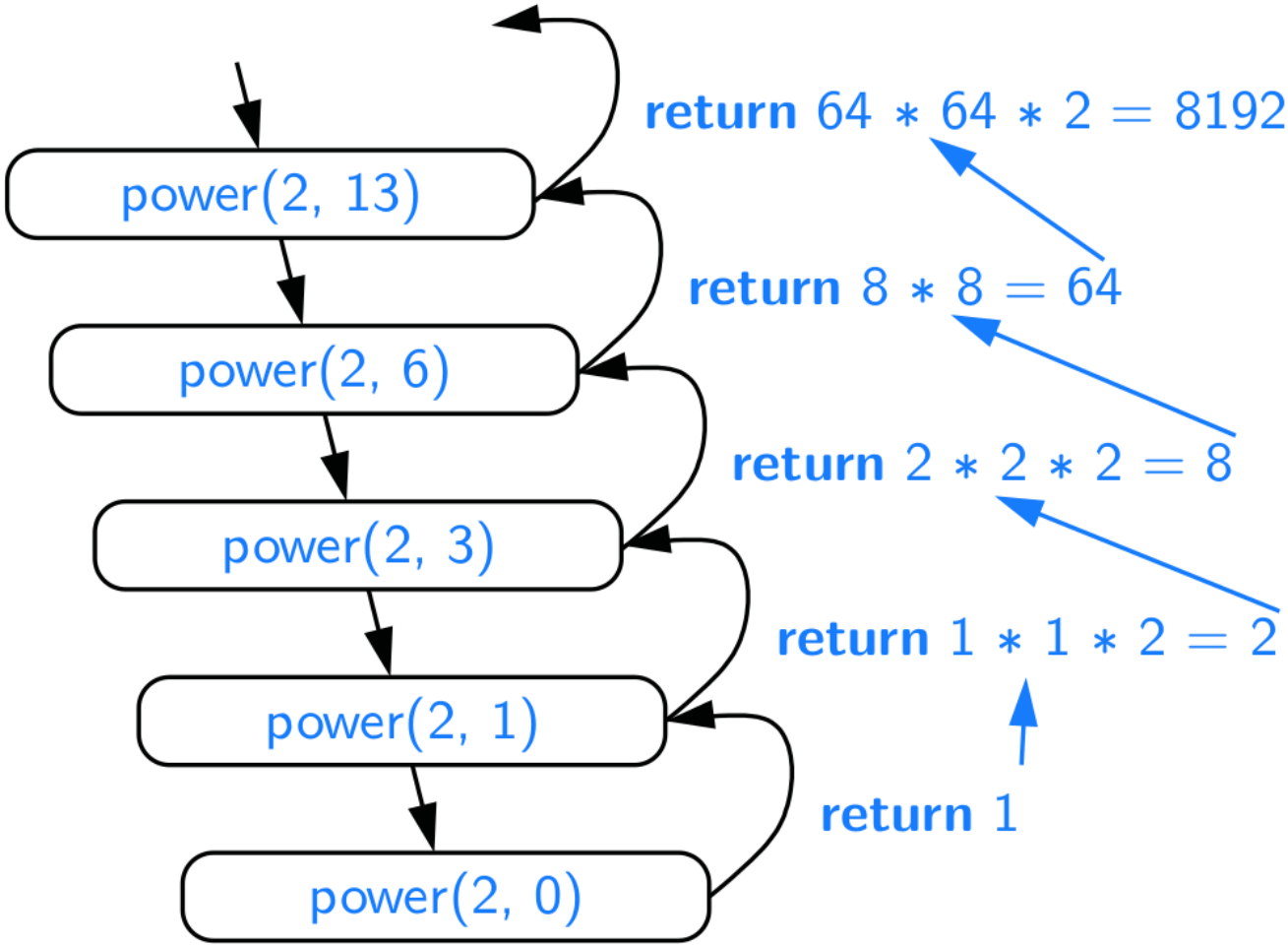
$$power(x, n) = \begin{cases} 1 & \text{if } n = 0 \\ (power(x, \lfloor \frac{n}{2} \rfloor))^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ (power(x, \lfloor \frac{n}{2} \rfloor))^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

since $x^n = (x^k)^2$ for even n, and $x^n = (x^k)^2 \cdot x$ for odd n, where $k = \lfloor \frac{n}{2} \rfloor$

Computing Powers: A Fast Recursive Implementation

```
1   /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2   public static double power(double x, int n) {
3     if (n == 0)
4       return 1;
5     else {
6       double partial = power(x, n/2);          // rely on truncated division of n
7       double result = partial * partial;
8       if (n % 2 == 1)                          // if n odd, include extra factor of x
9         result *= x;
10      return result;
11    }
12  }
```

Computing Powers: Recursion Trace



time $O(\log n)$; memory $O(\log n)$

Binary Recursion

Examples we have seen?

Sum of Array Elements

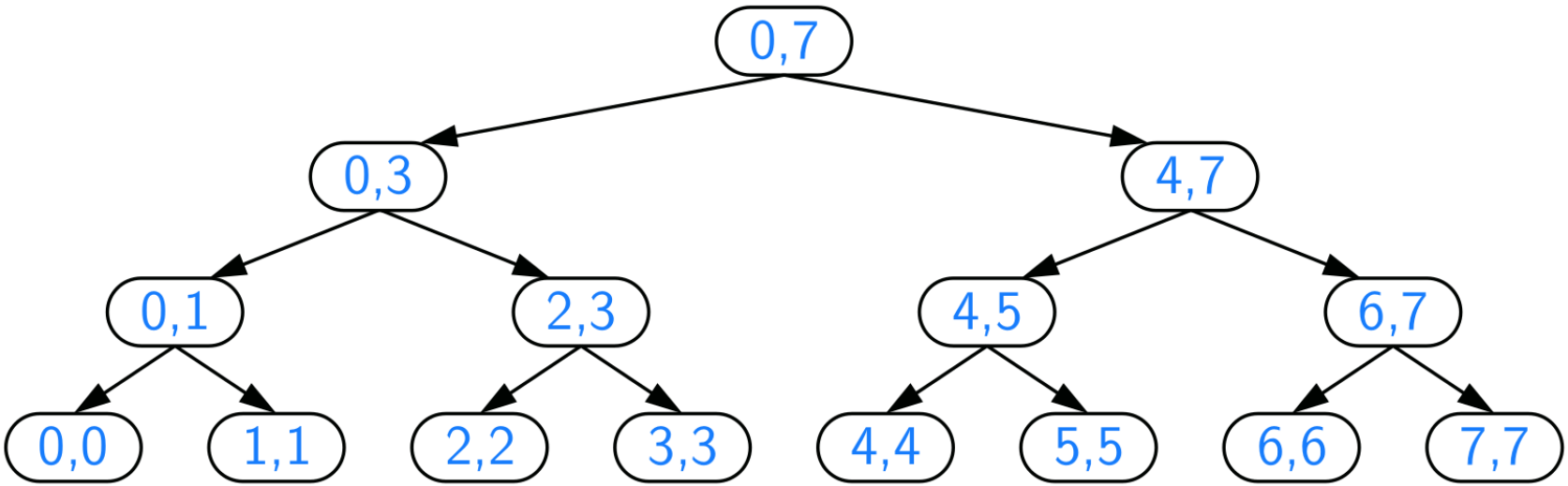
We want to compute the sum of an array of n integers using *binary* recursion

Recursively compute the sum of the first half, and the sum of the second half, and add those sums together

Sum of Array Elements: A Binary Recursive Implementation

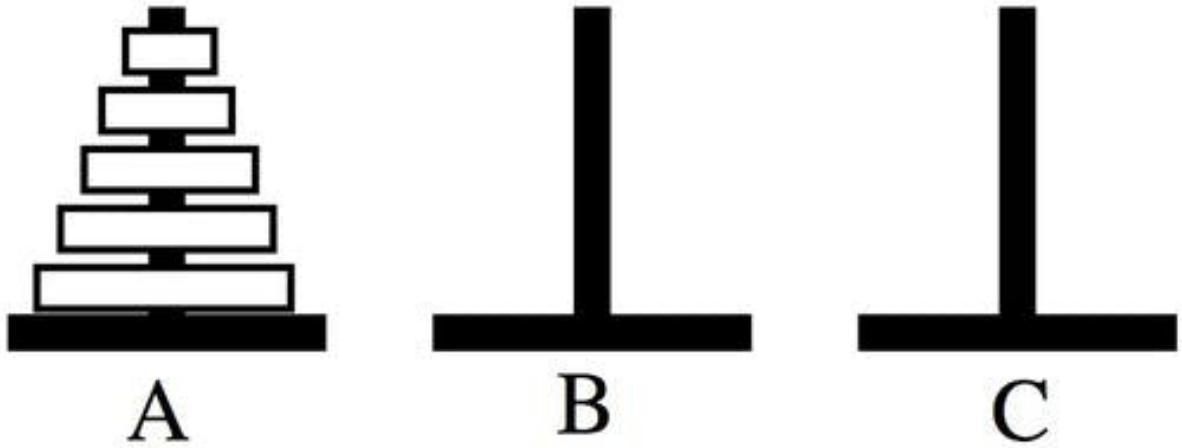
```
1  /** Returns the sum of subarray data[low] through data[high] inclusive. */
2  public static int binarySum(int[ ] data, int low, int high) {
3    if (low > high)                    // zero elements in subarray
4      return 0;
5    else if (low == high)              // one element in subarray
6      return data[low];
7    else {
8      int mid = (low + high) / 2;
9      return binarySum(data, low, mid) + binarySum(data, mid+1, high);
10   }
11 }
```

Sum of Array Elements: Binary Recursion Trace



time $O(n)$; memory $O(\log n)$ (excluding the array)

Towers of Hanoi



Move all the disks from peg A to peg C, moving one disk at a time, so that we never place a larger disk on top of a smaller one

Towers of Hanoi: A Binary Recursive Implementation

```
1  /** Java recursive function to solve tower of hanoi puzzle */
2  public static void towerOfHanoi(int n, char from, char to, char aux) {
3    if (n == 1) {
4      System.out.println("Move disk 1 from rod " +  from + " to rod " + to);
5      return;
6    }
7    towerOfHanoi(n-1, from, aux, to);
8    System.out.println("Move disk " + n + " from rod " +  from
9       + " to rod " + to);
10   towerOfHanoi(n-1, aux, to, from);
11 }
```

time $O(2^n)$; memory $O(n)$

Parameterizing a Recursion

Sometimes we need to reparameterize the signature of the method to define recursive subproblems

binarySearch(data, target, low, high) vs. binarySearch(data, target)

Other examples: reverseArray, linearSum, binarySum

A standard technique for a cleaner public interface: make the recursive version private, and introduce a cleaner public method

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearch(int[ ] data, int target) {
  return binarySearch(data, target, 0, data.length - 1);
}
```

Misuse of Recursion

- inefficient recursion, i.e. large execution times as a result of bad recursive design
- infinite recursion, i.e. making recursive calls without reaching a base case
- large recursive depths reaching the memory limit

Fibonacci Numbers: Inefficient Recursion

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-2} + F_{n-1} \quad \text{for } n > 1$$

Fibonacci Numbers: Inefficient Recursion

$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-2} + F_{n-1} \quad \text{for } n > 1$$

```
1  /** Returns the nth Fibonacci number (inefficiently). */
2  public static long fibonacciBad(int n) {
3    if (n <= 1)
4      return n;
5    else
6      return fibonacciBad(n-2) + fibonacciBad(n-1);
7  }
```

Fibonacci Numbers: Inefficient Recursion

The number of calls more than doubles for each two consecutive indices

$$
\begin{aligned}
c_0 &= 1 \\
c_1 &= 1 \\
c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\
c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\
c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\
c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\
c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\
c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\
c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67
\end{aligned}
$$

The number of calls $c_n > 2^{n/2}$ is exponential in n

Fibonacci Numbers: Efficient Recursion

Let's rely on linear recursion instead!

```
1  /** Returns array containing the pair of Fibonacci numbers,
2   F(n) and F(n-1). */
3  public static long[ ] fibonacciGood(int n) {
4    if (n <= 1) {
5          long[ ] answer = {n, 0};
6          return answer;
7    }
8    else {
9          long[ ] temp = fibonacciGood(n - 1);
10                                          // returns {F(n-1), F(n-2)}
11         long[ ] answer = {temp[0] + temp[1], temp[0]};
12                                          // we want {F(n), F(n-1)}
13         return answer;
14   }
15 }
```

time $O(n)$; memory $O(n)$

Tail Recursion

Tail recursion occurs when a linearly recursive method makes its recursive call as its last step

Tail recursion examples: binarySearch, reverseArray

Such methods can be easily converted to non-recursive methods (which saves on some resources)

```
1  /** Returns true if the target value is found in the data array. */
2  public static boolean binarySearchIterative(int[ ] data, int target) {
3    int low = 0;
4    int high = data.length - 1;
5    while (low <= high) {
6      int mid = (low + high) / 2;
7      if (target == data[mid])              // found a match
8        return true;
9      else if (target < data[mid])
10       high = mid - 1;                     // only consider values left of mid
11     else
12       low = mid + 1;                      // only consider values right of mid
13   }
14   return false;                           // loop ended without success
15 }
```

Summary

Reading

Chapter 5 Recursion

Questions?