CS 5044
Object-Oriented Programming with Java

Q&A Session

Why (and why not) recursion?

- Why recursion?
  - Certain types of problems are naturally much easier to describe (thus solve) recursively
    - Sorting and searching are the most common examples, but there are many more
    - Domain-specific languages and data formats are often defined and parsed recursively
  - Recursive algorithms can be mathematically proven as correct
    - This makes testing significantly more reliable and convenient
    - The concept is nearly identical to "proof by induction"
  - Converting an *arbitrary* recursive algorithm to its iterative equivalent can be tricky
    - Converting from iterative to recursive is relatively easy (but see "why not" below)
- Why not recursion?
  - Recursion can be slower (and take more memory) than an iterative equivalent, however...
    - ...the differences are often insignificant, and speed is often not critical anyway
  - If the problem is not naturally recursive, a recursive solution is less idiomatic
    - If iteration is more straightforward, or even equivalent, that's what will be expected
- Summary:
  - Feel free to use recursion wherever it makes sense
  - Don't go out of your way to force-fit recursion into naturally iterative processes

Syntax and strategies for recursion

- No special syntax or structures are required for recursion
  - Just like calling any other method to do some work and return some value
    - The method called just happens to be the same method that is making the call
- Typical approach splits input into two groups: first part and all the rest
  - You get to decide how to split the input, such that:
    - The first part can be processed reasonably easily
    - All the rest is similar to the original input, only a bit simpler and/or smaller
      - Often exactly the same structure but with fewer elements or smaller values
  - Next, you need to actually process the first part
    - You may safely assume that you've *already solved* how to process all the rest
      - This often helps you choose how to split the input
      - This also sometimes helps in processing the first part
  - Finally, you need criteria to detect when there isn't even a first part remaining
    - Usually this is a *degenerate* case (empty String, empty collection, 0, null, etc.)
- Notice that we never actually need to solve how to process all the rest!

Factorials: recursion in practice

- Definition: factorial(n) is the product of all numbers 1 ... n *(for all n >= 1)*
  - factorial(1) = 1 = 1
  - factorial(2) = 1*2 = 2
  - factorial(3) = 1*2*3 = 6
  - factorial(4) = 1*2*3*4 = 24
  - factorial(5) = 1*2*3*4*5 = 120
  - factorial(9) = 1*2*3*4*5*6*7*8*9 = 362,880
- It's a naturally recursive algorithm, so we'll first solve this iteratively(!)
  - Why? The multiplicands look like they were generated by a loop, and we understand loops
  - We'll use while() loops just for clarity in our examples, rather than for() loops

```java
public int factorialI(int n) {  // valid for positive n only (we'll ignore any validation here)
    int product = 1;            // need an accumulator for our running total
    int i = 1;                  // start the loop index at 1
    while(i <= n) {             // repeat until we reach n
        product = product * i;  // multiply by the next number
        i = i + 1;              // increment the index
    }
    return product;             // return our accumulator
}
```

- That's great, but what about recursion?!

Factorials: recursion in practice, with actual recursion

- Let's look at the examples again, aligned differently, and with parentheses:

```
– factorial(1) =  1
– factorial(2) = (1)*2 = 1*2
– factorial(3) =        (1*2)*3 = 1*2*3
– factorial(4) =                 (1*2*3)*4 = 1*2*3*4
– factorial(5) =                            (1*2*3*4)*5
```

- A slightly different pattern emerges here...

Factorials: recursion in practice, with actual recursion

- Let's look at the examples again, aligned very differently and with parentheses:
  - `factorial(1) =  1`
  - `factorial(2) = (1)*2 = 1*2`
  - `factorial(3) =        (1*2)*3 = 1*2*3`
  - `factorial(4) =                 (1*2*3)*4 = 1*2*3*4`
  - `factorial(5) =                           (1*2*3*4)*5`
- A slightly different pattern emerges here...
  - Each factorial requires only one new multiplication, if we can use the prior calculation!

Factorials: recursion in practice, with actual recursion

- Let's look at the examples again, aligned very differently and with parentheses:
  - `factorial(1) =  1`
  - `factorial(2) = (1)*2 = 1*2`
  - `factorial(3) =        (1*2)*3 = 1*2*3`
  - `factorial(4) =                 (1*2*3)*4 = 1*2*3*4`
  - `factorial(5) =                             (1*2*3*4)*5`
- A slightly different pattern emerges here...
  - Each factorial requires only one new multiplication, if we can use the prior calculation!
    - factorial(1) = 1
    - factorial(2) = factorial(1) * 2
    - factorial(3) = factorial(2) * 3
    - factorial(4) = factorial(3) * 4
    - factorial(5) = factorial(4) * 5

This seems to be a special case...

Factorials: recursion in practice, with actual recursion

- Let's look at the examples again, aligned very differently and with parentheses:

```
– factorial(1) =  1
– factorial(2) = (1)*2 = 1*2
– factorial(3) =        (1*2)*3 = 1*2*3
– factorial(4) =                 (1*2*3)*4 = 1*2*3*4
– factorial(5) =                            (1*2*3*4)*5
```

- A slightly different pattern emerges here...
  - Each factorial requires only one *new* multiplication, if we can use the prior calculation!
    - factorial(1) = 1 — This seems to be a special case...
    - factorial(2) = factorial(1) * 2
    - factorial(3) = factorial(2) * 3
    - factorial(4) = factorial(3) * 4
    - factorial(5) = factorial(4) * 5
- Let's redefine the factorial function more generally, with a recursive definition:
  - factorial(n) = 1 *(special case, for n == 1)*
  - factorial(n) = factorial(n-1) * n *(for all n > 1)*

Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```

CALL STACK

Currently Running →

someMethod
int value: 4
int result: pending

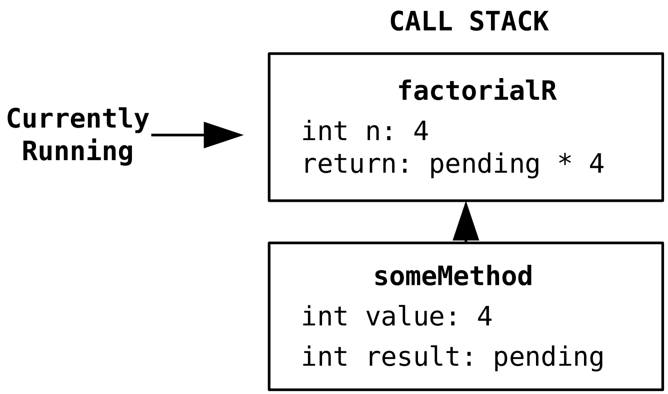
Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```



Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```

Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

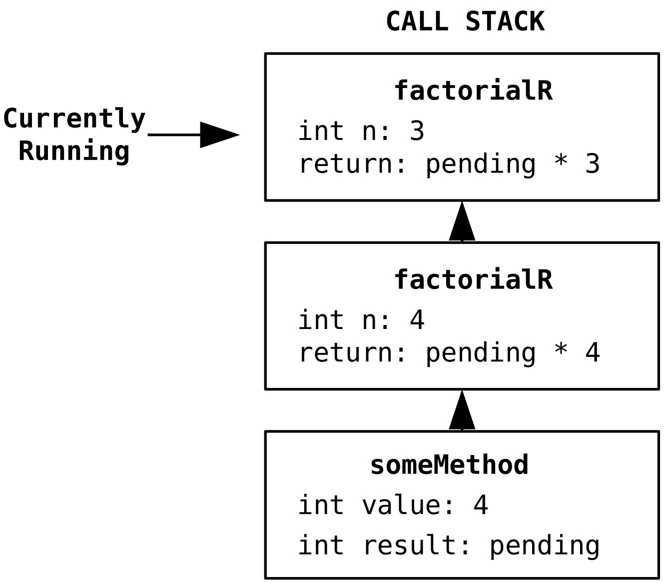
factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```



Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

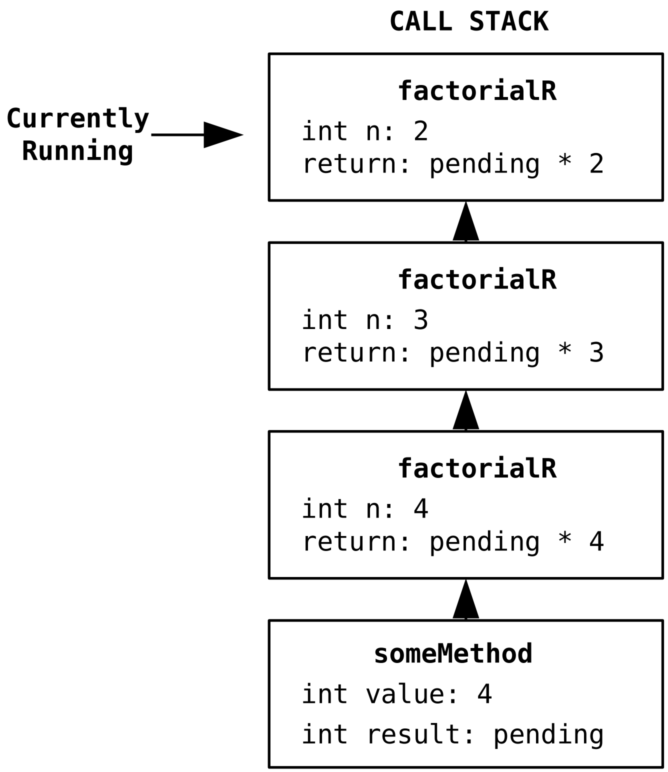
factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```



Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```java
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```java
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```

CALL STACK

Currently Running →

factorialR
int n: 1
return: pending

factorialR
int n: 2
return: pending * 2

factorialR
int n: 3
return: pending * 3

factorialR
int n: 4
return: pending * 4

someMethod
int value: 4
int result: pending

Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

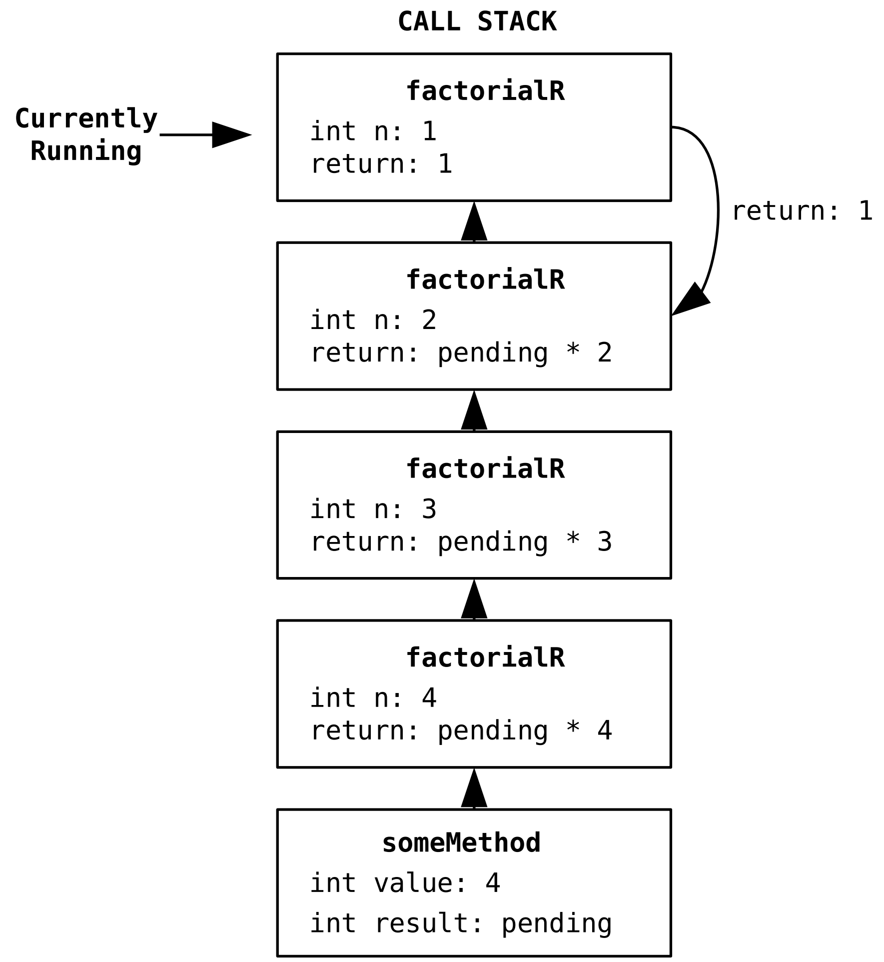
factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```java
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```java
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```



Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

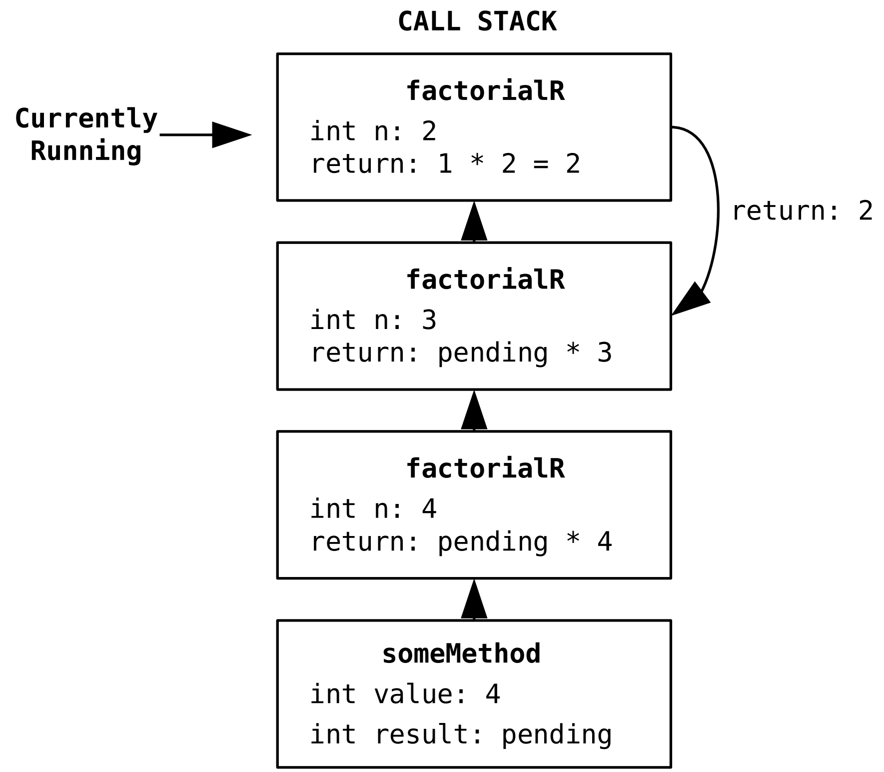
factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {        // detect terminating condition
        return 1;        // return special case
    }
    return factorialR(n - 1) * n;    // recursion!
}
```

Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```



Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

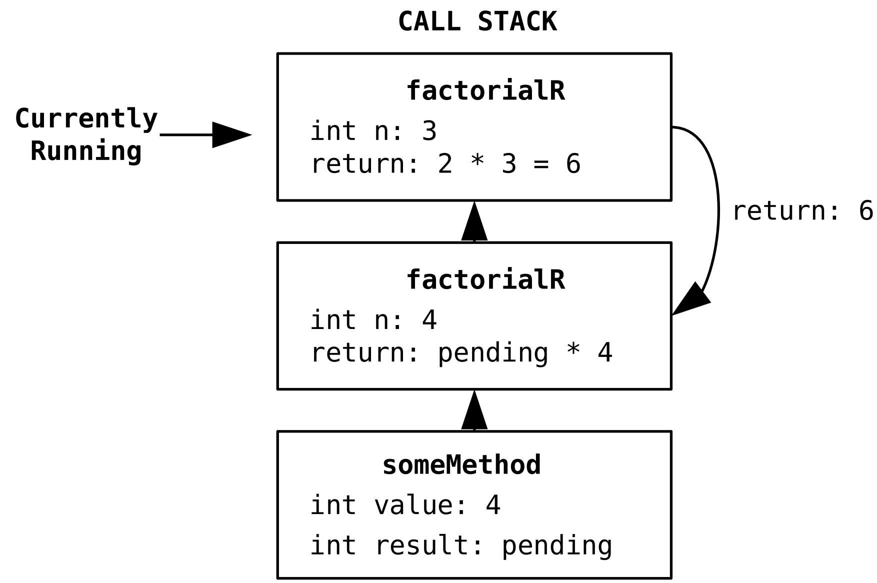
factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```



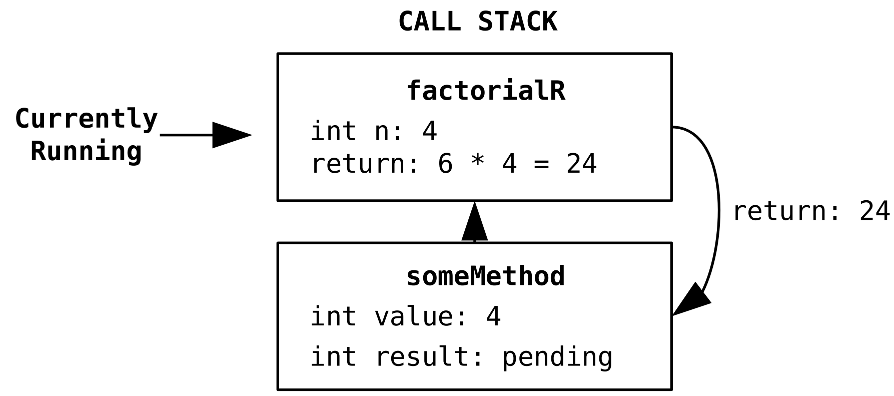
Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```
public int factorialR(int n) {
    if (n == 1) {       // detect terminating condition
        return 1;       // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```



Let's trace through a call with argument 4

```
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```

Recursion walkthrough

factorial(n) = 1 *(for n == 1)*

factorial(n) = factorial(n-1) * n *(for all n > 1)*

Notice there are no loops suggested by this definition

```java
public int factorialR(int n) {
    if (n == 1) {        // detect terminating condition
        return 1;        // return special case
    }
    return factorialR(n - 1) * n;   // recursion!
}
```

Let's trace through a call with argument 4

```java
public void someMethod() {
    int value = 4;
    int result = factorialR(value);
    System.out.println(result);
}
```

CALL STACK

Currently Running →

someMethod
int value: 4
int result: 24

Recursion vs iteration by example: find minimum

- Consider a method to find the minimum of a collection of numbers
  - The iterative approach, using a while() loop for clarity:

```
private int findMinIterative(List<Integer> list) {
    int currMin = Integer.MAX_VALUE;
    int index = 0;
    while (index < list.size()) {
        currMin = Math.min(currMin, list.get(index));
        index = index + 1;
    }
    return currMin;
}
```

  - Convert to the recursive equivalent, using a recursive helper:

```
private int findMinRecursive(List<Integer> list) {
    return findMinHelper(list, Integer.MAX_VALUE, 0);
}

private int findMinHelper(List<Integer> list, int currMin, int index) {
    if (index >= list.size()) {
        return currMin;
    }
    return findMinHelper(list, Math.min(currMin, list.get(index)), index + 1);
}
```

Recursion vs iteration by example: find minimum

- Consider a method to find the minimum of a collection of numbers
  - The iterative approach, using a while() loop for clarity:

```
private int findMinIterative(List<Integer> list) {
 1 int currMin = Integer.MAX_VALUE;
 2 int index = 0;
 3 while (index < list.size()) {
    4 currMin = Math.min(currMin, list.get(index));
    5 index = index + 1;
   }
 6 return currMin;
}
```

  - Convert to the recursive equivalent, using a recursive helper:

```
private int findMinRecursive(List<Integer> list) {
    return findMinHelper(list, Integer.MAX_VALUE, 0);
}

private int findMinHelper(List<Integer> list, int currMin, int index) {
  3 if (index >= list.size()) {
     6 return currMin;
    }
    return findMinHelper(list, Math.min(currMin, list.get(index)), index + 1);
}
```

Recursion vs iteration by example: find minimum

- Consider a method to find the minimum of a collection of numbers
  - The iterative approach, using a while() loop for clarity:

```
private int findMinIterative(List<Integer> list) {
 (1) int currMin = Integer.MAX_VALUE;
 (2) int index = 0;
 (3) while (index < list.size()) {
      (4) currMin = Math.min(currMin, list.get(index));
      (5) index = index + 1;
     }
 (6) return currMin;
}
```

  - Convert to the recursive equivalent, using a recursive helper:

```
private int findMinRecursive(List<Integer> list) {
    return findMinHelper(list, Integer.MAX_VALUE, 0);
}

private int findMinHelper(List<Integer> list, int currMin, int index) {
  (3) if (index >= list.size()) {
      (6) return currMin;
    }
    return findMinHelper(list, Math.min(currMin, list.get(index)), index + 1);
}
```

Note that the recursive approach doesn't require any local variables, nor does it mutate anything!

These can be extremely important properties in certain situations, such as in multi-threaded and/or distributed systems.

However, the price we pay is more memory usage, plus some added processing overhead, so there are always trade-offs to consider!

It's also arguably a somewhat less elegant solution than iteration; can we do any better?

Recursion vs iteration by example: find minimum

- Consider a method to find the minimum of a collection of numbers
  - The iterative approach, using a while() loop for clarity:

```
private int findMinIterative(List<Integer> list) {
    int currMin = Integer.MAX_VALUE;
    int index = 0;
    while (index < list.size()) {
        currMin = Math.min(currMin, list.get(index));
        index = index + 1;
    }
    return currMin;
}
```

  - Normally we wouldn't retain the index, and would instead just iterate with a smaller list:

```
private int findMinRecursive(List<Integer> list) {
    return findMinHelper(list, Integer.MAX_VALUE);
}

private int findMinHelper(List<Integer> list, int currMin) {
    if (list.isEmpty()) {
        return currMin;
    }
    return findMinHelper(list.subList(1, list.size()), Math.min(currMin, list.get(0)));
}
```

    - This is more fitting of the concept of separating the "first" from the "rest"

Recursion vs iteration by example: find minimum

- Consider a method to find the minimum of a collection of numbers
  - The iterative approach, using a while() loop for clarity:

```
private int findMinIterative(List<Integer> list) {
    int currMin = Integer.MAX_VALUE;
    int index = 0;
    while (index < list.size()) {
        currMin = Math.min(currMin, list.get(index));
        index = index + 1;
    }
    return currMin;
}
```

  - Further, we typically also remove the accumulator, so we no longer need the helper:

```
private int findMinRecursive(List<Integer> list) {
    if (list.isEmpty()) {
        return Integer.MAX_VALUE;
    }
    return Math.min(list.get(0), findMinRecursive(list.subList(1, list.size())));
}
```

    - The code is now slightly denser, yet somewhat *more* elegant than iteration...
      - However, unless the code is in a multi-threaded or distributed environment...
        - » ...the iterative approach would be a much less surprising way to find a minimum

Thinking recursively (with an iterative start)

- Let's try to multiply two non-negative integers, by only using addition
  - For example: 6*8 = 8 + 8 + 8 + 8 + 8 + 8
- Iterative approach (A*B):
  - Start with an accumulator set to zero, repeat A times of adding B each time

```
public int multiplyI(int a, int b) {
    int sum = 0;
    int index = a;
    while (index > 0) {
        sum = sum + b;
        index = index - 1;
    }
    return sum;
}
```

- Recursive approach (A*B):
  - Notice that 6*8 = 8 + (5*8) so we have an answer in terms of a simplified problem
    - Eventually we reach 1*8 = 8 + (0*8) then finally 0*8 = 0 (the terminating case)
  - More generally A*B = B + (A - 1)*B *(for all A > 0)*; A*B = 0 *(for A == 0)*

```
public int multiplyR(int a, int b) {
    if (a == 0) {
        return 0;
    }
    return b + multiplyR(a - 1, b);
}
```

Count on recursions (yet another example)

- Suppose we need to count the number of spaces in a string
  - Example: countSpaces("Suppose we need to count the number of spaces in a string") = 11
- Iterative approach:
  - Start with index=0 and count=0, repeat for every char, increment count if char is space

```
public int countSpacesI(String s) {
    int count = 0;
    int index = 0;
    while (index < s.length()) {
        if (s.charAt(index) == ' ') {
            count = count + 1;
        }
        index = index + 1;
    }
    return count;
}
```

- Recursive approach:
  - Total spaces = number of spaces in first char + number of spaces in all remaining chars
  - countSpaces(...) = ((firstChar is space?) 1 else 0) + countSpaces(substring after firstChar)
    - Eventually we reach an empty string, where count("") = 0 as the terminating case

```
public int countSpacesR(String s) {
    if (s.isEmpty()) {
        return 0;
    }
    return (s.charAt(0) == ' ' ? 1 : 0) + countSpacesR(s.substring(1));
}
```

Sorting and searching

- Sorting problems tend to be naturally recursive algorithms
  - No single sorting algorithm is "best" for all cases
    - Best may involve several factors: fastest, least memory, most interruptible, etc.
    - Even on a single factor, algorithms vary by best case, worst case, average, etc.
- Searching also tends to be recursive, especially for common data structures
  - There are dozens of distinct search tree algorithms, with varying properties
    - Java's `TreeMap` and `TreeSet` happen use use a "Red-Black" binary tree structure
- Order-of-the-function ("Big O Notation") performance is usually all we need to know
  - Still, depending on usage patterns, specific implementations may vary in several ways:
    - Memory usage, index size, speed of insertion, speed of retrieval, etc.
  - However, for the vast majority of applications, we simply don't care!
    - Selecting any appropriate collection class is normally the best approach
      - For example, choose `Set` or `Map` rather than `List` when `contains()` is important
    - Unless you have exceptional performance needs, just use what the language provides
      - Collections typically mention performance orders in their API documentation