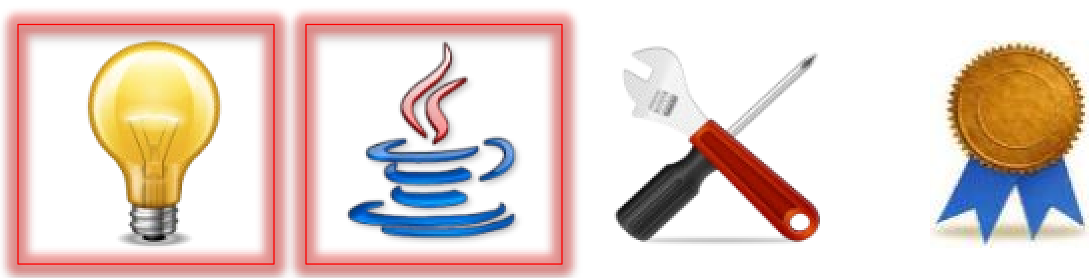
Recursion: Thinking About It



Recursion

- A remarkably important concept and programming technique in computer science is ***recursion***
  - A ***recursive method*** is simply one that calls itself
- There are two quite different views of recursion!
  - We ask for your patience as we introduce them one at a time...

Question Considered Now

- ***How should you think about*** recursion so you can use it to develop elegant recursive methods to solve certain problems?

Question Considered Next

- ***Why*** do those recursive methods work?

Question Considered Only Later

- ***How*** do those recursive methods work?
  - Don't worry; we will come back to this
  - If you *start* by insisting on knowing the answer to this question, you may never be fully capable of developing elegant recursive solutions to problems!

Suppose...

- You need to reverse a `String`
- Contract specification looks like this:

```
/**
 * Reverses a String.
 * ...
 * @ensures
 * reversedString = rev(s)
*/
private static String reversedString(String s) {...}
```

Suppose...

- You need to reverse a `String`
- Contract specification looks like:

```
/**
 * Reverses a String.
 * ...
 * @ensures
 * reversedString = rev(s)
 */
private static String reversedString(String s) {...}
```

Try to implement it (i.e., write the method body).

One Possible Solution

```
private static String reversedString(String s) {
   String rs = "";
   for (int i = 0; i < s.length(); i++) {
      rs = s.charAt(i) + rs;
   }
   return rs;
}
```

Trace It

	`s = "abc"` `rs = ""`
`for (int i = 0; i < s.length(); i++) {`	
`   rs = s.charAt(i) + rs;`	
`}`	

Trace It: Iteration 1

	s = "abc" *rs = ""*
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" *rs = ""* *i = 0*
`rs = s.charAt(i) + rs;`	
`}`	

Trace It: Iteration 1

	s = "abc" *rs = ""*
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" *rs = ""* *i = 0*
`    rs = s.charAt(i) + rs;`	
	s = "abc" *rs = "a"* *i = 0*
`}`	

Trace It: Iteration 2

	s = "abc" *rs = ""*
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" *rs = "a"* *i = 1*
`rs = s.charAt(i) + rs;`	
	s = "abc" *rs = "a"* *i = 0*
`}`	

Trace It: Iteration 2

Code	State
	s = "abc" *rs = ""*
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" *rs = "a"* *i = 1*
`   rs = s.charAt(i) + rs;`	
	s = "abc" *rs = "ba"* *i = 1*
`}`	

Trace It: Iteration 3

	s = "abc" *rs = ""*
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" *rs = "ba"* *i = 2*
`rs = s.charAt(i) + rs;`	
	s = "abc" *rs = "ba"* *i = 1*
`}`	

Trace It: Iteration 3

Code	State
	s = "abc" rs = ""
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" rs = "ba" i = 2
`rs = s.charAt(i) + rs;`	
	s = "abc" rs = "cba" i = 2
`}`	

Trace It: Ready to Return

	s = "abc" rs = ""
`for (int i = 0; i < s.length(); i++) {`	
	s = "abc" rs = "ba" i = 2
`rs = s.charAt(i) + rs;`	
	s = "abc" rs = "cba" i = 2
`}`	
	s = "abc" rs = "cba"

Oh, Did I Mention...

- There is already a static method in the class `FreeLunch`, with exactly the same contract:

```
/**
 * Reverses a String.
 * ...
 * @ensures
 * reversedString = rev(s)
*/
private static String reversedString(String s) {...}
```

A Free Lunch Sounds Good!

- The slightly nasty thing about the `FreeLunch` class is that its methods will not directly solve your problem: you have to make your problem “smaller” first
- This `reversedString` code will *not* work:

```
private static String reversedString(String s) {
    return FreeLunch.reversedString(s);
}
```

Recognizing the Smaller Problem

- A key to recursive thinking is the ability to recognize some ***smaller*** instance of the ***same*** problem “hiding inside” the problem you need to solve
- Here, suppose we recognize the following property of string reversal:

```
rev(<x> * a) = rev(a) * <x>
```

The Smaller Problem

- If we had some way to reverse a string of length 4, say, then we could reverse a string of length 5 by:
  – removing the character on the left end
  – reversing what's left
  – adding the character that was removed onto the right end

The Smaller Problem

- If we had so… length 4, say… string of length 5
  - removing the character on the left end
  - ***reversing what's left***
  - adding the character that was removed onto the right end

> This is a ***smaller*** instance of exactly the ***same*** problem as we need to solve.

Time for Our Free Lunch

- We can use the `FreeLunch` class now:

```
private static String reversedString(String s) {
   String sub = s.substring(1);
   String revSub =
     FreeLunch.reversedString(sub);
   String result = revSub + s.charAt(0);
   return result;
}
```

Trace It

	`s = "abc"`
`String sub = s.substring(1);`	
	`s = "abc"` `sub = "bc"`
`String revSub =` `  FreeLunch.reversedString(sub);`	
	`s = "abc"` `sub = "bc"` `revSub = "cb"`
`String result = revSub + s.charAt(0);`	
	`s = "abc"` `sub = "bc"` `revSub = "cb"` `result = "cba"`

Trace It

How do you trace over this call? By looking at the contract, as usual!

Code	State
	s = "abc"
	s = "abc" sub = "bc"
`String revSub = FreeLunch.reversedString(sub);`	
	s = "abc" sub = "bc" revSub = "cb"
`String result = revSub + s.charAt(0);`	
	s = "abc" sub = "bc" revSub = "cb" result = "cba"

Almost Done With Lunch

- Is this code correct?

```
private static String reversedString(String s) {
  String sub = s.substring(1);
  String revSub =
    FreeLunch.reversedString(sub);
  String result = revSub + s.charAt(0);
  return result;
}
```

Almost Done With Lunch

- Is this code correct?

```
private static String reversedString(String s) {
   String sub = s.substring(1);
   String revSub =
      FreeLunch.r        edString(sub);
   String resul        Sub + s.charAt(0);
   return
}
```

This call has a precondition: `s` must not be the empty string (which can be gleaned from the `String` API with a careful reading).

Almost Done With Lunch

This call has a precondition: `s` must not be the empty string (which can be gleaned from the `String` API with a careful reading).

```
                                        s) {
    String revSub =
      FreeLunch.reversedStr    (sub);
    String result = revSub + s.charAt(0);
    return result;
}
```

Accounting for Empty `s`

```
private static String reversedString(String s) {
  if (s.length() == 0) {
    return s;
  } else {
    String sub = s.substring(1);
    String revSub =
      FreeLunch.reversedString(sub);
    String result = revSub + s.charAt(0);
    return result;
  }
}
```

Accounting for Empty `s`

```
private static String reversedString(String s) {
   if (s.length() == 0) {
      return s;
   } else {
      String sub            bstring(1);
      Str
         F
      Str
      retu
   }
}
```

This test could also be done as:

`s.equals("")`

but not as:

`s == ""`

Accounting for Empty `s`

```java
private static String reversedString(String s) {
    if (s.length() == 0) {
        return s;
    } else {
        String sub = s.     tring(1);
        String revS
            FreeLunch
        String resu
        return resu
    }
}
```

Returning an empty string could also be written as:
`return "";`

Oh, Did I Mention...

- Sorry, there is no `FreeLunch`!

There Is No FreeLunch?!?

```
private static String reversedString(String s) {
  if (s.length() == 0) {
    return s;
  } else {
    String sub = s.substring(1);
    String revSub =
      FreeLunch.reversedString(sub);
    String result = revSub + s.charAt(0);
    return result;
  }
}
```

There Is No `FreeLunch`?!?

```
private static String reversedString(String s) {
  if (s.length() == 0) {
    return s;
  } else {
    String sub = s.substring(1);
    String revSub =
      reversedString(sub);
    String result = revSub + s.charAt(0);
    return result;
  }
}
```

We Don't Need a FreeLunch

We just wrote the code for `reversedString`, so we can call ***our own*** version rather than the one from `FreeLunch`.

```
private s
   if (s
      ret
   } else {
      String s     s.substring(1);
      String  evSub =
         reversedString(sub);
      String result = revSub + s.charAt(0);
      return result;
   }
}
```

A Recursive Method

Note that the body of `reversedString` now calls itself, so we just wrote a ***recursive method***.

```
priva
   if

   } else {
      String         s.substring(1);
      String  evSub =
         reversedString(sub);
      String result = revSub + s.charAt(0);
      return result;
   }
}
```

Crucial Theorem for Recursion

- If your code for a method is correct when it calls the (hypothetical) `FreeLunch` version of the method — remember, it must be on a ***smaller*** instance of the problem — then your code is *still* correct when you replace every call to the `FreeLunch` version with a ***recursive*** call to your own version

Theorem Applied

- If the code that makes a call to `FreeLunch.reversedString` is correct, then so is the code that makes a ***recursive*** call to `reversedString`
- Remember: this is so only because the call to `FreeLunch.reversedString` is for a ***smaller*** problem, i.e., a string with smaller length

No Need For Multiple Returns

```
private static String reversedString(String s) {
    String result = s;
    if (s.length() > 0) {
        String sub = s.substring(1);
        String revSub = reversedString(sub);
        result = revSub + s.charAt(0);
    }
    return result;
}
```

Alternative solution with a ***single*** return. In this case, multiple returns are not necessary and they do not provide a better solution.

Another Example: Suppose...

- You need to increment a `NaturalNumber`

```
/**
 * Increments a NaturalNumber.
 * ...
 * @updates n
 * @ensures
 * n = #n + 1
*/
private static void increment (NaturalNumber n) {...}
```

Another Exa

- You need to increm

```
/**
 * Increments a Natura
 * ...
 * @updates n
 * @ensures
 * n = #n + 1
 */
private static void increment (NaturalNumber n) {...}
```

Try to implement it (i.e., write the method body) using *only* the kernel methods:

`multiplyBy10`
`divideBy10`
`isZero`

Not So Easy

- Unlike string reversal, there is no straightforward iterative solution to this problem
- So, let’s try a recursive solution...
- Can you recognize the smaller problem?

Recognizing the Smaller Problem

- Think about how you would increment (add 1 to) a number using the grade-school arithmetic algorithm
- Examples:

```
  41072        41079        41999
+     1      +     1      +     1
  -----        -----        -----
  41073        41080        42000
```

Recognizing the Smaller Problem

- Think about how you would increment (add 1 to) a number using the grade-school arithmetic algorithm
- Examples:

41072	41079	41999
+ 1	+ 1	+ 1
41073	41080	42000

The Smaller Problem

- If we had some way to increment a number with 4 digits, say, then we could increment a 5-digit number by:
  - taking off the one's digit
  - incrementing it and asking: is there is a "carry"?
  - if there is, then incrementing what's left
  - putting back the updated one's digit
- Important: multiple carries don't matter

The Smaller Problem

This is a ***smaller*** instance of exactly the ***same*** problem as we need to solve.

- If we had sor with 4 digits, 5-digit number by:
  - taking off the one's digit
  - incrementing it and asking: i there is a "carry"?
  - if there is, then ***incrementing what's left***
  - putting back the updated one's digit
- Important: multiple carries don't matter

Time for Our Free Lunch

- We can use the `FreeLunch` class now:

```
private static void increment (NaturalNumber n) {
   int onesDigit = n.divideBy10();
   onesDigit++;
   if (onesDigit == 10) {
      onesDigit = 0;
      FreeLunch.increment(n);
   }
   n.multiplyBy10(onesDigit);
}
```

Almost Done With Lunch

- Is this code correct?

```
private static void increment (NaturalNumber n) {
   int onesDigit = n.divideBy10();
   onesDigit++;
   if (onesDigit == 10) {
      onesDigit = 0;
      FreeLunch.increment(n);
   }
   n.multiplyBy10(onesDigit);
}
```

Done With Lunch

- Is this code correct?

```
private static void increment (NaturalNumber n) {
   int onesDigit = n.divideBy10();
   onesDigit++;
   if (onesDigit == 10) {
      onesDigit = 0;
      increment(n);
   }
   n.multiplyBy10(onesDigit);
}
```

Theorem Applied

- If the code that makes a call to `FreeLunch.increment` is correct, then so is the code that makes a ***recursive*** call to `increment`
- Remember: this is so only because the call to `FreeLunch.increment` is for a ***smaller*** problem, i.e., a number less than the incoming value of `n`

Another Example

```
/**
 * Raises an int to a power.
 * ...
 * @requires
 * p >= 0  and  [n ^ (p) is within int range]
 * @ensures
 * power = n ^ (p)
 */
private static int power(int n, int p) {...}
```

A Hidden Smaller Problem

- Can you recognize a smaller problem of the same kind hiding inside the computation of n^p?
- Here is a mathematical property that might help you see one:

$$n^p = n * n^{p-1} \qquad (for\ p > 0)$$

A Hidden Smaller Problem

- Can you recognize a smaller problem of the same kind hiding inside the computation of n^p?
- Here is a mathematical property that might help you see one:

$$n^p = n * n^{p-1} \quad \text{(for } p > 0\text{)}$$

A Hidden Smaller Problem

- Can you rec
  the same ki
  computation of
- Here is a m ematical property that might help yo see one:

$$n^p = n * n^{p-1} \qquad (for\ p > 0)$$

Can you write the code for `power` as specified earlier, based on this property? (You also need to account for $p = 0$.)

Another Hidden Smaller Problem

- Here is a *different* mathematical property that might help you see a *different* smaller problem of the same kind:

$$n^p = (n^{p/2})^2 \qquad \text{(for even } p > 1\text{)}$$

Another Hidden Smaller Problem

- Here is a *different* mathematical property that might help you see a *different* smaller problem of the same kind:

$$n^p = (n^{p/2})^2 \qquad \text{(for even } p > 1\text{)}$$

Another Hi[...]

- Here is a *di*[...] that might h[...] problem of th[...] kind:

$$n^p = (n^{p/2})^2 \qquad \text{(for even } p > 1\text{)}$$

Can you write the code for `power` as specified earlier, based on this property? (You also need to account for all the other values of `p`.)

Fast Powering

- If you can write the code by using the second property as a guide, your implementation will be much faster than by using the first property
  - And much faster than the obvious iterative code!
- This really matters when you adapt the algorithm to work with `NaturalNumber` rather than **`int`**

Remaining Steps

- Use `FreeLunch` when you need to solve a smaller problem of the same kind (making sure it really *is* smaller in some sense!)
- Show that your code is correct *assuming* `FreeLunch.power` has the same ***contract*** as the `power` code you're writing
- Replace any calls to `FreeLunch.power` with recursive calls to your own version of `power`
- Sit back and let the theorem about recursion show that your now-recursive code is correct