UNIVERSITY *of* WISCONSIN
LA CROSSE
COMPUTER SCIENCE

Week 10: Recursion

Software Design II (CS 220): D. Mathias

1

What is recursion?

Good Answer: recursion is (often) an alternative to iteration.

Better Answer: recursion is a valuable tool for solving certain types of problems.

Best Answer: recursion is magic.

2

Seriously, what is recursion?

- A remarkably important concept and programming technique
  - A ***recursive method*** is simply one that calls itself

3

Question Considered Now

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

4

Question Considered Next

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

5

Question Considered Only Later

- ***How*** do those recursive methods work?
  - Don't worry; we will come back to this
  - Trust me, it's better this way

6

Suppose...

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

```
/**
 * Reverses a String.
 * ...
 * @return string with chars of s in reverse order (String)
 */
private static String reverseString(String s)
{
   ...
}
```

7

Suppose...

- You need to ring
- Specification is:

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

```
/**
 * Reverses a
 * ...
 * @return string w chars of s in reverse order (String)
 */
private static String reverseString(String s)
{
   ...
}
```

8

One Possible Solution

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

9

Let’s trace it

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

10

Let’s 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;	
}	

11

Let’s 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
}	

12

Let's 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
`}`	

13

Let's 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 = "ba" i = 1
`}`	

14

Let’s 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`

`}`	

15

Let’s 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 = "cba"`
	`i = 2`

`}`	

16

Let's 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"

17

Oh, Did I Mention...

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

```
/**
 * Reverses a String.
 * ...
 * @return string with chars of s in reverse order (String)
 */
private static String reverseString(String s)
{
   ...
}
```

18

A Free Lunch Sounds Good!

- The slightly nasty thing about the `FreeLunch` class is that its methods will not directly solve the problem: you have to make the problem "smaller" before you can use `FreeLunch`
- Therefore, this `reverseString` code will *not* work:

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

19

Recognizing the Smaller Problem

- A key to recursive thinking is the ability to recognize a ***smaller*** instance of the ***same*** problem "hiding inside" the problem you need to solve
- Suppose we recognize the following property of string reversal:

```
rev(<x> + a) = rev(a) + <x>
where x is a char and a is a String
```

20

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:
  1. removing the character on the left end
  2. reversing what's left
  3. adding the character that was removed onto the right end

21

The Smaller Problem

- If we had some way t
  say, then we could re
  1. removing the characte ... left end
  2. reversing what's left
  3. 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.

22

Time for our Free Lunch

- We can use the `FreeLunch` class now:

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

23

Let's trace it

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

24

Let’s trace it

How do you trace over this call? By looking at the specification, of course!

	bc"
`String s`	
	`s = "abc"` `sub = "bc"`
`String revSub =` `FreeLunch.reverseString(sub);`	

	`s = "abc"`
	`sub = "bc"`
	`revSub = "cb"`

`String result = revSub + s.charAt(0);`	

	`s = "abc"`
	`sub = "bc"`
	`revSub = "cb"`
	`result = "cba"`

25

Almost done with Lunch

- Is this code correct?:

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

26

Almost done with Lunch

- Is this code correct?:

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

This call has a precondition: s must not be the empty string (see the Java documentation)

27

Almost done with Lunch

- Is

This call has a precondition: s must not be the empty string (see the Java documentation)

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

28

Accounting for Empty s

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

29

Oh, did I mention…

- Sorry, there is no FreeLunch!

30

There's No `FreeLunch`?!?

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

31

We Don't Need a `FreeLunch`

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

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

32

A Recursive Method

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

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

33

A 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

34

The Theorem Applied

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

35

No Need For Multiple Returns

```
private static String reverseString(String s)
{
    String result = s;
    if (s.length() > 0)
    {
        String sub = s.substring(1);
        String revSub = reverseString(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.

36

Another Example

- What is the first recursive algorithm you learned? (Think about grade-school)

Addition

Consider adding 1 to an integer:

- 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
```

39

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

40

The Smaller Problem

- If we have a w
with 4 digits, fo
increment a 5-digit n by:
  – taking off the one's digit
  – incrementing it and asking: is ere is a "carry"?
  – if there is, then ***incrementing what's left***
  – putting back the updated one's digit
- Important: multiple carries don't matter

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

41

Time for Our Free Lunch

- We can use the `FreeLunch` class now:

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

42

Almost Done With Lunch

- Is this code correct?:

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

43

Done With Lunch

- Is this code correct?:

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

44

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`

45

Recursive Structure

- For problem P, can we leverage the power of recursion?

46

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
    - Factorial: (n – 1)! (n – 2)! (n – 23)!

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
    - Factorial: (n – 1)! (n – 2)! (n – 23)!
    - Fibonacci: (n – 1)! (n – 2)! (n – 12)!

49

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
    - Factorial: (n – 1)! (n – 2)! (n – 23)!
    - Fibonacci: fib(n – 1) fib(n – 2) fib(n – 12)
    - reverseString: remove first char, remove last char

50

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?

51

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?
    - This is usually the easy part

52

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?
    - This is usually the easy part
  - Can we use solutions to smaller problems to construct a solution to P?

53

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?
    - This is usually the easy part
  - Can we use solutions to smaller problems to construct a solution to P?
    - Factorial: given (n – 1)! we easily get n!

54

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?
    - This is usually the easy part
  - Can we use solutions to smaller problems to construct a solution to P?
    - Factorial: given (n – 1)! we easily get n!
    - Fibonacci: given fib(n – 1) and fib(n – 2) we easily get fib(n)

55

Recursive Structure

- For problem P, can we leverage the power of recursion?
  - Can we divide P into one or more smaller instances?
  - Can we efficiently solve the smaller instances?
    - This is usually the easy part
  - Can we use solutions to smaller problems to construct a solution to P?
    - Factorial: given (n – 1)! we easily get n!
    - Fibonacci: given fib(n – 1) and fib(n – 2) we easily get fib(n)
    - stringReverse: given rev(sub) we easily get rev(s)

56

Factorial: The canonical example

- Is this correct?

```
public int factorial(int n)
{
    int result = n * factorial(n-1);
    return result;
}
```

57

Factorial: The canonical example

- Why not?

```
public int factorial(int n)
{
    int result = n * factorial(n-1);
    return result;
}
```

58

Recursive Structure

- What should we do when an instance of P is too small to divide into smaller problems?

59

Recursive Structure

- What should we do when an instance of P is too small to divide into smaller problems?
  - Solve it! In many cases it's trivial.

60

Recursive Structure

- What should we do when an instance of P is too small to divide into smaller problems?
  - Solve it! In many cases it's trivial.
    - Factorial: 0! = 1   1! = 1

61

Recursive Structure

- What should we do when an instance of P is too small to divide into smaller problems?
  - Solve it! In many cases it's trivial.
    - Factorial: 0! = 1   1! = 1

    - Fibonacci: fib(0) = 0   fib(1) = 1

62

Recursive Structure

- What should we do when an instance of P is too small to divide into smaller problems?
  - Solve it! In many cases it's trivial.
    - Factorial: 0! = 1   1! = 1
    - Fibonacci: fib(0) = 0   fib(1) = 1
    - stringReverse: stringReverse("") = ""

63

Factorial: The canonical example

- Is this correct?

```
public int factorial(int n)
{
    if(n <= 1)
        return n;
    return n * factorial(n-1);
}
```

64

A way to reason about recursion

Bottom up – begin with the base case

65

A way to reason about recursion

- Bottom up – begin with the smallest-value case

```
public int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

- if n=0 or n=1, factorial(n) returns correct result ✓

66

A way to reason about recursion

- Bottom up – begin with the smallest-value case

```
public int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

- if n=2, factorial(n) returns 2 * factorial(1)
- We just convinced ourselves that factorial(1) is correct
- 2 * factorial(1) is the correct result for factorial(2) ✓

67

A way to reason about recursion

- Bottom up – begin with the smallest-value case

```
public int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

- if n=3, factorial(n) returns 3 * factorial(2)
- We just convinced ourselves that factorial(2) is correct
- 3 * factorial(2) is the correct result for factorial(3) ✓

68

A way to reason about recursion

- Bottom up – begin with the smallest-value case

```
public int factorial(int n)
{
    if (n <= 1)
        return 1;
    return n * factorial(n-1);
}
```

- if n=4, factorial(n) returns 4 * factorial(3)
- We just convinced ourselves that factorial(3) is correct
- 4 * factorial(3) is the correct result for factorial(4) ✓

69