
Recursion

- A recursive method is a method that calls itself (directly or indirectly.)
- A recursive definition is a definition of something in terms of itself
- Some recursive definitions don't make sense, (e.g. from Webster's: growl: to utter a growl), but others do
- For example:
 - A *list of numbers* is either:
 - * A single number, or
 - * A number followed by a list of numbers.
 - For example:
 - * 5 is a list of numbers
 - * 7, 5 is a list of numbers (because 5 is a list)
 - * 6, 7, 5 is a list of numbers (because 7, 5 is a list)
 - * 8, 6, 7, 5 is a list of numbers (because 6, 7, 5 is a list)

Recursive functions

- Factorial: the factorial of a natural number n , written $n!$ is the multiplication of the first n positive integers, i.e.

$$n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 2) \cdot (n - 1) \cdot n \quad (1)$$

But note that

$$1 \cdot 2 \cdot 3 \cdot \dots \cdot (n - 2) \cdot (n - 1) = (n - 1)! \quad (2)$$

So by (1) and (2) we get

$$n! = (n - 1)! \cdot n \quad (3)$$

But we have to assume a “base case”, by defining

$$0! = 1 \quad (4)$$

Recursive functions (contd.)

Hence, (3) and (4) together gives us an alternative, and recursive definition of (1):

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

This can be implemented as a static recursive method:

```
static int factorial(int n)
{
    if (n == 0) {
        return 1;
    }
    return factorial(n-1)*n;
}
```

Execution of recursive methods

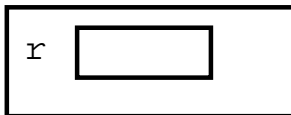
Consider the following client for this factorial function:

```
int r;  
r = factorial(4);
```

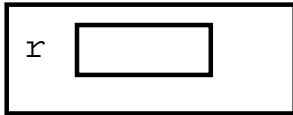
Its execution proceeds as follows:

This is executed in some frame:

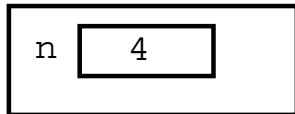
Some frame



When we call `factorial(4)`; a new frame for the method is created:
Some frame



factorial frame



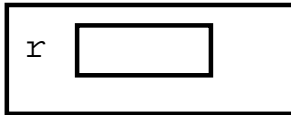
We execute the body of `factorial`; `n` is not 0 so we execute

```
return factorial(n-1)*n;
```

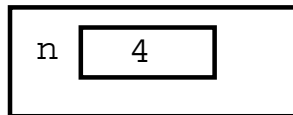
which in this frame is the same as

```
return factorial(4-1)*4;
```

Some frame



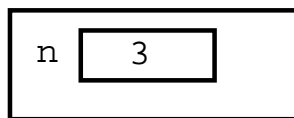
factorial frame



pending computation:

`return factorial(3)*4;`

factorial frame



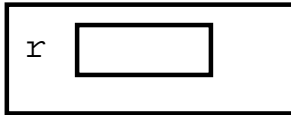
Again, we execute the body of factorial;
again, n is not 0 so we execute

`return factorial(n-1)*n;`

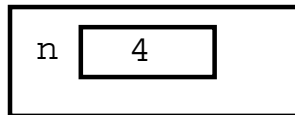
which in this frame is the same as

`return factorial(3-1)*3;`

Some frame



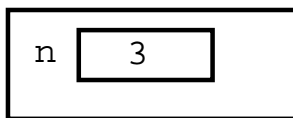
factorial frame



pending computation:

`return factorial(3)*4;`

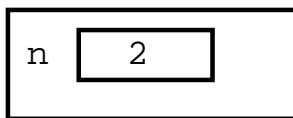
factorial frame



pending computation:

`return factorial(2)*3;`

factorial frame



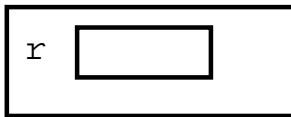
Again, we execute the body of factorial;
again, n is not 0 so we execute

`return factorial(n-1)*n;`

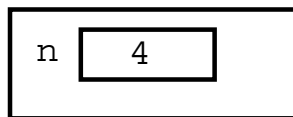
which in this frame is the same as

`return factorial(2-1)*2;`

Some frame



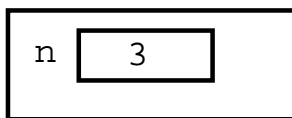
factorial frame



pending computation:

`return factorial(3)*4;`

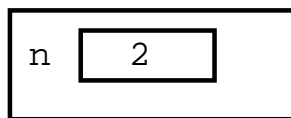
factorial frame



pending computation:

`return factorial(2)*3;`

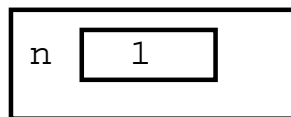
factorial frame



pending computation:

`return factorial(1)*2;`

factorial frame



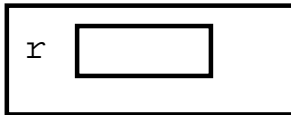
Again, we execute the body of factorial;
again, n is not 0 so we execute

`return factorial(n-1)*n;`

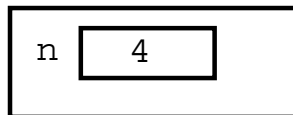
which in this frame is the same as

`return factorial(1-1)*1;`

Some frame



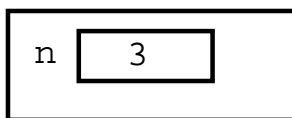
factorial frame



pending computation:

`return factorial(3)*4;`

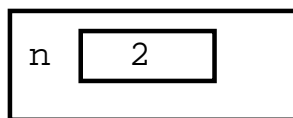
factorial frame



pending computation:

`return factorial(2)*3;`

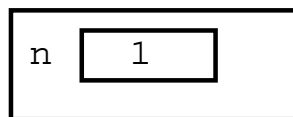
factorial frame



pending computation:

`return factorial(1)*2;`

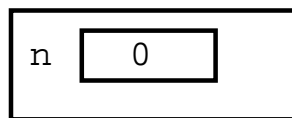
factorial frame



pending computation:

`return factorial(0)*1;`

factorial frame

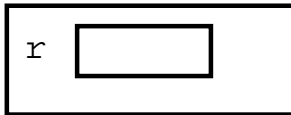


Now, we have reached the base case, and n is 0, so we execute:

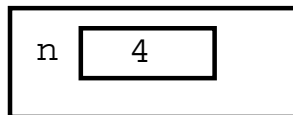
`return 1;`

We get rid of the frame, and pass the returned value to the caller

Some frame



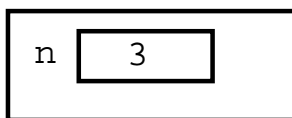
factorial frame



pending computation:

```
return factorial(3)*4;
```

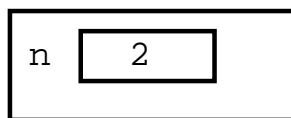
factorial frame



pending computation:

```
return factorial(2)*3;
```

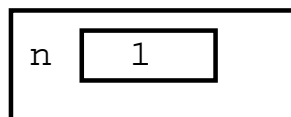
factorial frame



pending computation:

```
return factorial(1)*2;
```

factorial frame



The pending computation here was:

```
return factorial(0)*1;
```

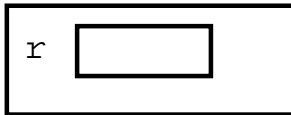
and the method called `factorial(0)`

returned 1, so this pending computation is now:

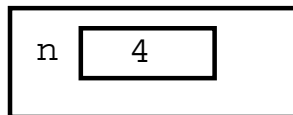
```
return 1*1;
```

We get rid of the frame, and pass the returned value to the caller

Some frame



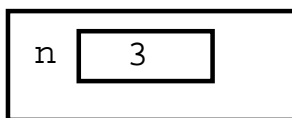
factorial frame



pending computation:

`return factorial(3)*4;`

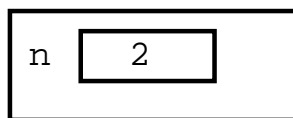
factorial frame



pending computation:

`return factorial(2)*3;`

factorial frame



The pending computation here was:

`return factorial(1)*2;`

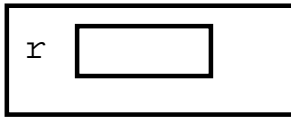
and the method called `factorial(1)`

returned 1, so this pending computation is now:

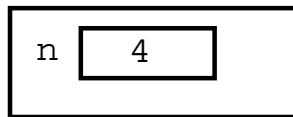
`return 1*2;`

We get rid of the frame, and pass the returned value to the caller

Some frame



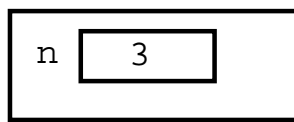
factorial frame



pending computation:

```
return factorial(3)*4;
```

factorial frame



The pending computation here was:

```
return factorial(2)*3;
```

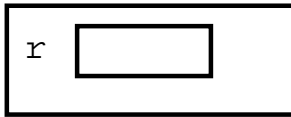
and the method called `factorial(2)`

returned 2, so this pending computation is now:

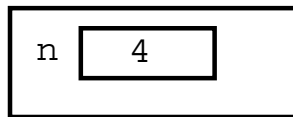
```
return 2*3;
```

We get rid of the frame, and pass the returned value to the caller

Some frame



factorial frame



The pending computation here was:

```
return factorial(3)*4;
```

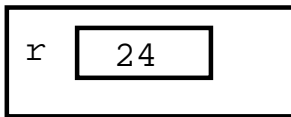
and the method called `factorial(3)`

returned 6, so this pending computation is now:

```
return 6*4;
```

We get rid of the frame, and pass the returned value to the caller

Some frame



The pending computation here was:

```
r = factorial(4);
```

which returned 24, so this pending computation is now:

```
r = 24;
```

Recursion on other types

- Problem: given a string s , return the reverse of the string
- Analysis:
 - Notation:
 - * $\text{rev}(s)$ is the reverse of s
 - * s_i is the i -th character of s
 - * $\text{len}(s)$ is the length of s
 - * $\text{rest}(s)$ is the string s without its first character s_0
(i.e. $\text{rest}(s) = s_1s_2\dots s_n$ where $n = \text{len}(s) - 1$)
 - Formal definition of reverse:

$$\text{rev}(s) = \begin{cases} "" & \text{if } s = "" \\ \text{rev}(\text{rest}(s)) + s_0 & \text{otherwise} \end{cases}$$

Reverse (contd.)

- For example:

$$\begin{aligned}\text{rev}(\text{"abcd"}) &= \text{rev}(\text{"bcd"}) + 'a' \\ &= (\text{rev}(\text{"cd"}) + 'b') + 'a' \\ &= ((\text{rev}(\text{"d"}) + 'c') + 'b') + 'a' \\ &= (((\text{rev}(\text{""}) + 'd') + 'c') + 'b') + 'a' \\ &= (((\text{""} + 'd') + 'c') + 'b') + 'a' \\ &= ((\text{"d"} + 'c') + 'b') + 'a' \\ &= (\text{"dc"} + 'b') + 'a' \\ &= \text{"dcb"} + 'a' \\ &= \text{"dcba"}\end{aligned}$$

Reverse (contd.)

```
public class MoreStringOperations {
    static String reverse(String s)
    {
        if (s.equals("")) {
            return "";
        }
        return reverse(rest(s))+s.charAt(0);
    }
    static String rest(String s)
    {
        String result = "";
        int i = 1;
        while (i < s.length()) {
            result = result + s.charAt(i);
            i++;
        }
        return result;
    }
}
```

Double recursion

- Problem: Compute the n -th Fibonacci number
- Analysis: The Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ... is defined by:

$$fib(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Implementation:

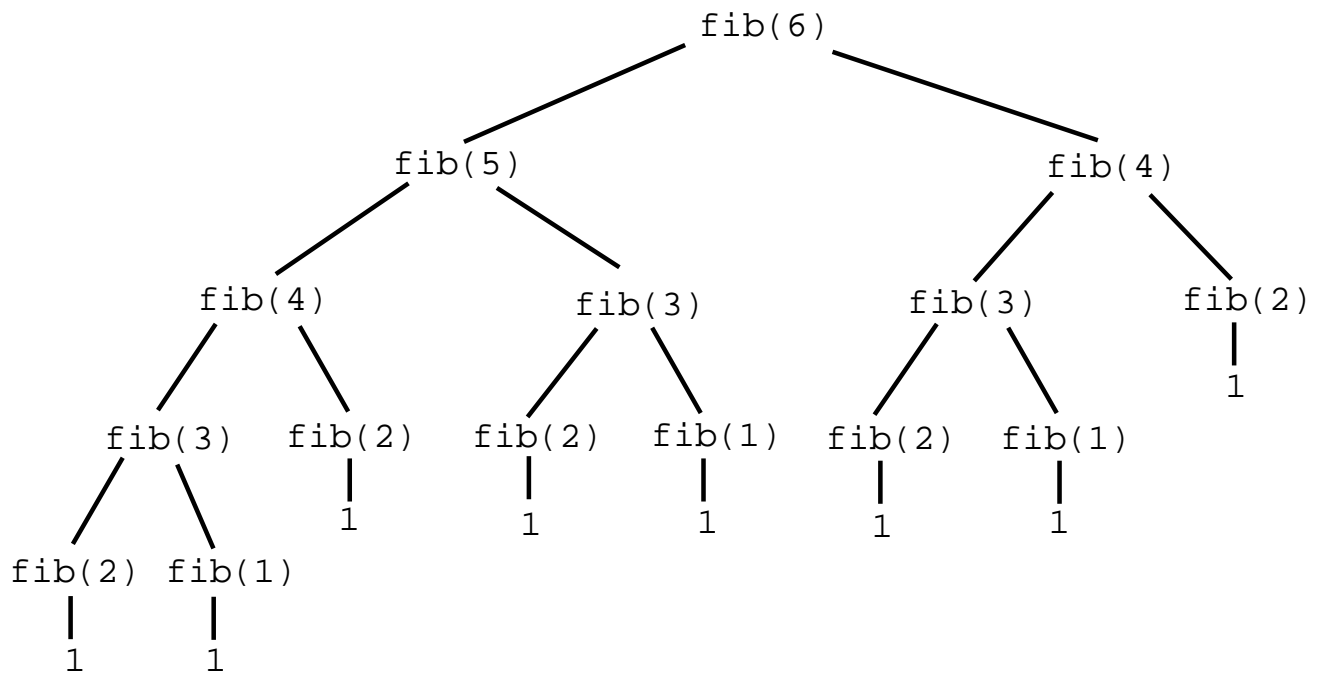
```
static int fib(int n)
{
    if (n <= 2) {
        return 1;
    }
    return fib(n-1)+fib(n-2);
}
```

Iteration vs recursion

- Iterative solution to the Fibonacci problem:

```
static int fib(int n)
{
    int a, b, c, i;
    a = 1;
    b = 1;
    c = 1;
    i = 3;
    while (i <= n) {
        c = a + b;
        a = b;
        b = c;
        i++;
    }
    return c;
}
```

Execution trees



Recursion and termination

```
static void f(int n)
{
    System.out.println(n);
    f(n);
}
```

Recursion and termination

$f(5)$

$f(5)$

$f(5)$

$f(5)$

.

.

.

Recursion and termination

```
static int f(int n)
{
    System.out.println(n);
    return f(n) + 1;
}
```

Recursion and termination

```
f(5)
return f(5) + 1
return (f(5) + 1) + 1
return ((f(5) + 1) + 1) + 1
.
.
.
```

Recursion and termination

```
static int f(int n)
{
    System.out.println(n);
    return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return f(-1)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    System.out.println(n);
    return f(n);
}
```

Recursion and termination

```
f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
return f(5)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    System.out.println(n);
    return f(n-1);
}
```

Recursion and termination

```
f(5)
return f(4)
return f(3)
return f(2)
return f(1)
return f(0)
return 1
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    System.out.println(n);
    return f(n - 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    System.out.println(n);
    return f(n / 2) + 1;
}
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    System.out.println(n);
    return f(n + 2);
}
```

Recursion and termination

```
f(5)
returns f(7)
returns f(9)
returns f(11)
returns f(13)
.
.
.
```

Recursion and termination

```
static int f(int n)
{
    if (n == 0) return 1;
    else if (n % 2 == 0) return n / 2;
    else return f( f( 3*n+1 ) );
}
```

Recursion

```
void p(...)  
{  
    ...  
    q(...);  
    ...  
}  
void q(...);  
{  
    ...  
    r(...);  
    ...  
}  
void r(...)  
{  
    ...  
}
```

Recursion

```
void p(...)  
{  
    ...  
    p(...);  
    ...  
}
```

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \end{cases}$$

```
double pow(double b, int n)
{
    if (n == 0) return 1;
    return b * pow(b, n-1);
}
```

Recursion

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{if } n > 0 \text{ and } n \text{ is odd} \\ (b^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ is even} \end{cases}$$

```
double fastpow(double b, int n)
{
    if (n == 0) return 1;
    if (n % 2 == 1) return b * fastpow(b, n-1);
    double p = fastpow(b, n/2);
    return p * p;
}
```

Mutual Recursion

```
void p(...)  
{  
    ...  
    q(...);  
    ...  
}  
void q(...)  
{  
    ...  
    p(...);  
    ...  
}
```

The end