As we’ve already seen, programming languages incorporate the idea of function calls.

– This allows us to reuse code in multiple locations within a program.
– Is there any reason that a function shouldn’t be able to reuse itself?
int factorial(int n)
{
    if(n == 0 || n == 1)
        return 1;
    else return n * factorial(n-1);
}

Coding Recursion
Coding Recursion

• Potential problem: how can the program keep track of its state?
  – There will multiple versions of “n” over the different calls of the factorial function.
  – The answer: stacks!
  – The stack is a data structure we haven’t yet seen, but may examine in brief later in the course.
Coding Recursion

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

- Each individual method call within a recursive process can be called a *frame*. 
Coding Recursion

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

• We’ll use this to denote each frame of this method’s execution.
  – Let’s try n = 5.
int factorial(int n) {
    if(n == 0 || n == 1) return 1;
    else return n * factorial(n-1);
}

Result: 1
Coding Recursion

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

Result: 6
n: 5
n: 4
n: 3
Coding Recursion

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

Result: 24

n: 4
n: 5
Coding Recursion

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

Result: 120
Coding Recursion

- Notice how recursion looks in code – we have a function that calls itself.
  - In essence, we assume that we already have a function that already does most of the work needed, aside from one small manipulation.
  - The trick is that this “already there” function is actually the function we’re writing.
Coding Recursion

• Notice how recursion looks in code – we have a function that calls itself.
  – If the base case is correct, that’s half the battle.
  – If we then can show that our step properly calculates \( f(k + 1) \) from \( f(k) \), in math speak, we then have a proper recursive solution.
    • The function calls will perform the rest.
Recursion - Fibonacci

• Let’s examine how this would work for another classic recursive problem.
  – The Fibonacci sequence:
    
    Fib(0) = 1
    Fib(1) = 1
    Fib(n) = Fib(n-2) + Fib(n-1)

  – How can we code this?
  – What parts are the base case?
  – What parts are the recursive step?
int fibonacci(int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return fibonacci(n-2) + fibonacci(n-1);
}
Recursion - Fibonacci

```c
int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        A: return fibonacci(n-2) +
        B: fibonacci(n-1);
    }
}
```

We’ll use the below graphics to aid our analysis of this

[Graphics showing the recursion process with placeholders for n, pos, and part]
Recursion - Fibonacci

if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);

res: 1

| n: 3 | pos: A | part: --- |
| n: 5 | pos: A | part: --- |
Recursion - Fibonacci

If \( n = 0 \) \( || \) \( n = 1 \)

return 1;

else

A: return fibonacci(n-2) + 

B: fibonacci(n-1);

res: 1
if(n == 0 || n == 1)
    return 1;
else
A: return fibonacci(n-2) +
B:     fibonacci(n-1);

<table>
<thead>
<tr>
<th>n:</th>
<th>2</th>
<th>pos: B</th>
<th>part: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>n:</td>
<td>3</td>
<td>pos: B</td>
<td>part: 1</td>
</tr>
<tr>
<td>n:</td>
<td>5</td>
<td>pos: A</td>
<td>part: ---</td>
</tr>
</tbody>
</table>

res: 1
Recursion - Fibonacci

```java
if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);
```

```
res: 2
```

```
n: 3 pos: B part: 1
n: 5 pos: A part: ---
```
Recursion - Fibonacci

```java
if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);
```

res: 3  
n: 5  pos: A  part: ---
Recursion - Fibonacci

if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);

res: ...

<table>
<thead>
<tr>
<th>n</th>
<th>pos</th>
<th>part</th>
<th>res</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>A</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
Didn’t we already get an answer for $n = 2$?

Yep. So I’ll save us some time.
Recursion - Fibonacci

```java
if (n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);
```

<table>
<thead>
<tr>
<th>n</th>
<th>pos</th>
<th>part</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>A</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>res</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Recursion - Fibonacci

Didn’t we already get an answer for $n = 3$?

Yep. So I’ll save us some time.

<table>
<thead>
<tr>
<th>n</th>
<th>pos</th>
<th>part</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>A</td>
<td>---</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>3</td>
</tr>
</tbody>
</table>
Recursion - Fibonacci

Didn’t we already get an answer for $n = 3$?

Yep. So I’ll save us some time.

res: 3
Recursion - Fibonacci

if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);

res: 5
n: 5 pos: B part: 3
Recursion - Fibonacci

if(n == 0 || n == 1)
    return 1;
else
    A: return fibonacci(n-2) +
    B: fibonacci(n-1);

res: 8
Recursion - Fibonacci

• Can this be done more efficiently?
  – You betcha! First off, note that we had had to recalculate some of the intermediate answers.
  – What if we could have saved those answers?
  – It’s possible, and the corresponding technique is called *dynamic programming*.
  – We’ll not worry about that for now.