Programming Abstractions

CS106B

Cynthia Lee
Today’s Topics

Recursion!
- Functions calling functions

Next time:
- More recursion! It’s Recursion Week!
- Like Shark Week, but more nerdy
Recursion!

The exclamation point isn’t there only because this is so exciting; it also relates to our first recursion example....
Factorial!

\[ n! = n(n - 1)(n - 2)(n - 3)(n) \]
Factorial!

\[ n! = n(n - 1)(n - 2)(n - 3)(n - 4) \ldots (2)(1) \]

An alternate mathematical formulation:

\[
 n! = \begin{cases} 
 1 & \text{if } n = 1 \\
 n(n - 1)! & \text{otherwise}
\end{cases}
\]

Translated to code

```cpp
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * someFunctionThatKnowsFactorialOfNMinus1();
    }
}
```
Factorial!

Mathematical definition

\[ n! = \begin{cases} 
1 & \text{if } n = 1 \\
(n(n-1))! & \text{otherwise}
\end{cases} \]

Translated to code

```c
int factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```
The recursive function pattern

Always two parts:

**Base case:**
- This problem is so tiny, it’s hardly a problem anymore! Just give answer.

**Recursive case:**
- This problem is still a bit large, let’s bite off just one piece, and let recursion handle what remains.

Translated to code

```cpp
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1);  // delegate smaller problem
    }
}
```
The recursive function pattern

*Recursive case:*  
- This problem is still a bit large, let’s **bite off just one piece**, and **let recursion handle what remains.**

```c
int factorial(int n) {
    if (n == 1) { // Easy! Return trivial answer
        return 1;
    } else { // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1); // delegate smaller problem
    }
}
```
The recursive function pattern

*Recursive case:*  
- This problem is still a bit large, **let’s bite off just one piece**, and **let recursion** handle what remains.

```c
int factorial(int n) {
    if (n == 1) { // Easy! Return trivial answer
        return 1;
    } else { // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1); // delegate smaller problem
    }
}
```

This is “one bite” of the problem—just doing one of the many many product calculations.
The recursive function pattern

*Recursive case:*  
- This problem is still a bit large, let’s bite off just one piece, and let recursion handle what remains.

```c
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    }
    else {  // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1);  // delegate smaller problem
    }
}
```

This is delegating the rest of the work—all the other product calculations—to the recursive call.
The recursive function pattern

*Recursive case:*  
- This problem is still a bit large, let’s bite off just one piece, and let recursion handle what remains.

```c
int factorial(int n) {
    if (n == 1) {  // Easy! Return trivial answer
        return 1;
    } else {  // Not easy enough yet! Break into "smaller" problem
        return n * factorial(n - 1);  // delegate smaller problem
    }
}
```

This is “one bite” of the problem—just doing one of the many many product calculations.

This is delegating the rest of the work—all the other product calculations—to the recursive call.
Recap: the recursive function pattern

- Recursion is a way of taking a big problem and repeatedly breaking it into smaller and smaller pieces until it is so small that it can be so easily solved that it almost doesn't even need solving.

- There are two parts of a recursive algorithm:
  - base case: where we identify that the problem is so small that we trivially solve it and return that result
  - recursive case: where we see that the problem is still a bit too big for our taste, so we chop it into smaller bits and call ourselves (the function we are in now) on the smaller bits to find out the answer to the problem we face
Digging deeper in the recursion

Looking at how recursion works “under the hood”
Factorial!

Math definition

\[ n! = \begin{cases} 
1 & \text{if } n = 1 \\
n(n - 1) & \text{otherwise} 
\end{cases} \]

Recursive code

```cpp
int factorial(int n) {
    cout << n << endl; // added code
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}
```

What is the third thing printed when we call `factorial(4)`?
A. 1  
B. 2  
C. 3  
D. 4  
E. Other/none/more  

Stanford University
How does this look in memory?

* Take CS107 to learn much more!!
How does this look in memory?

Recursive code:

```c++
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
Fun fact:
The “stack” part of memory is a stack

Function **call** = **push** a stack frame
Function **return** = **pop** a stack frame

* Take CS107 to learn much more!!
The “stack” part of memory is a stack

Recursive code

```c
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction()
```
The “stack” part of memory is a stack

Recursive code

```c
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction() {
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
The “stack” part of memory is a stack

```cpp
Recursive code

```
The “stack” part of memory is a stack

Recursive code

```cpp
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction() {
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
What is the fourth value ever returned when we call factorial(4)?
A. 4
B. 6
C. 10
D. 24
E. Other/none/more than one

Recursive code

```cpp
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
The “stack” part of memory is a stack

Recursive code

```cpp
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}

void myfunction()
{
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
The “stack” part of memory is a stack

Recursive code

```c
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}
```

void myfunction(){
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
The “stack” part of memory is a stack

### Recursive code

```cpp
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}
```

```cpp
void myfunction() {
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```

<table>
<thead>
<tr>
<th>main()</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>myfunction() x:</td>
<td>4</td>
<td>xfac:</td>
<td>4</td>
</tr>
<tr>
<td>factorial() n:</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>factorial() n:</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Heap

Return 6
The “stack” part of memory is a stack

Recursive code

```c++
int factorial(int n) {
    cout << n << endl;
    if (n == 1) return 1;
    else return n * factorial(n - 1);
}
```

```c++
void myfunction()
{
    int x = 4;
    int xfac = 0;
    xfac = factorial(x);
}
```
Factorial!

Iterative version

```c
int factorial(int n) {
    int f = 1;
    while (n > 1) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```

Recursive version

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

NOTE: sometimes iterative can be much faster because it doesn’t have to push and pop stack frames. Method calls have overhead in terms of space and time (to set up and tear down).