Recursion
Recursion

• We might describe an implementation of an algorithm as being particularly “elegant” if it solves a problem in a way that is both interesting and easy to visualize.

• The technique of recursion is a very common way to implement such an “elegant” solution.

• The definition of a recursive function is one that, as part of its execution, invokes itself.
Recursion

• The factorial function ($n!$) is defined over all positive integers.

• $n!$ equals all of the positive integers less than or equal to $n$, multiplied together.

• Thinking in terms of programming, we’ll define the mathematical function $n!$ as `fact(n)`. 
Recursion

\[
\begin{align*}
\text{fact}(1) &= 1 \\
\text{fact}(2) &= 2 \times 1 \\
\text{fact}(3) &= 3 \times 2 \times 1 \\
\text{fact}(4) &= 4 \times 3 \times 2 \times 1 \\
\text{fact}(5) &= 5 \times 4 \times 3 \times 2 \times 1 \\
\cdots
\end{align*}
\]
Recursion

\[ \text{fact}(1) = 1 \]
\[ \text{fact}(2) = 2 \times \text{fact}(1) \]
\[ \text{fact}(3) = 3 \times 2 \times 1 \]
\[ \text{fact}(4) = 4 \times 3 \times 2 \times 1 \]
\[ \text{fact}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]
\[ \ldots \]
Recursion

\[ \text{fact}(1) = 1 \]
\[ \text{fact}(2) = 2 \times \text{fact}(1) \]
\[ \text{fact}(3) = 3 \times \text{fact}(2) \]
\[ \text{fact}(4) = 4 \times 3 \times 2 \times 1 \]
\[ \text{fact}(5) = 5 \times 4 \times 3 \times 2 \times 1 \]
\[ \ldots \]
Recursion

\[
\begin{align*}
\text{fact}(1) & = 1 \\
\text{fact}(2) & = 2 \times \text{fact}(1) \\
\text{fact}(3) & = 3 \times \text{fact}(2) \\
\text{fact}(4) & = 4 \times \text{fact}(3) \\
\text{fact}(5) & = 5 \times 4 \times 3 \times 2 \times 1 \\
& \quad \ldots
\end{align*}
\]
Recursion

\[ \text{fact}(1) = 1 \]
\[ \text{fact}(2) = 2 \times \text{fact}(1) \]
\[ \text{fact}(3) = 3 \times \text{fact}(2) \]
\[ \text{fact}(4) = 4 \times \text{fact}(3) \]
\[ \text{fact}(5) = 5 \times \text{fact}(4) \]
\[ \ldots \]
Recursion

\[ \text{fact}(n) = n \times \text{fact}(n-1) \]
Recursion

• This forms the basis for a **recursive definition** of the factorial function.

• Every recursive function has two cases that could apply, given any input.
  • The *base case*, which when triggered will terminate the recursive process.
  • The *recursive case*, which is where the recursion will actually occur.
Recursion

\[ \text{fact}(1) = 1 \]
\[ \text{fact}(2) = 2 \times \text{fact}(1) \]
\[ \text{fact}(3) = 3 \times \text{fact}(2) \]
\[ \text{fact}(4) = 4 \times \text{fact}(3) \]
\[ \text{fact}(5) = 5 \times \text{fact}(4) \]
\[ \ldots \]
Recursion

```c
int fact(int n)
{
    // base case
    // recursive case
}
```
Recursion

```c
int fact(int n)
{
    if (n == 1)
    {
        return 1;
    }

    // recursive case
}
```
Recursion

\[ \text{fact}(1) = 1 \]
\[ \text{fact}(2) = 2 \times \text{fact}(1) \]
\[ \text{fact}(3) = 3 \times \text{fact}(2) \]
\[ \text{fact}(4) = 4 \times \text{fact}(3) \]
\[ \text{fact}(5) = 5 \times \text{fact}(4) \]
\ldots
int fact(int n)
{
    if (n == 1)
    {
        return 1;
    }

    // recursive case
}
Recursion

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

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

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

• In general, but not always, recursive functions replace loops in non-recursive functions.
Recursion

• In general, but not always, recursive functions replace loops in non-recursive functions.

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

```c
int fact2(int n)
{
    int product = 1;
    while(n > 0)
    {
        product *= n;
        n--;
    }
    return product;
}
```
Recursion

• In general, but not always, recursive functions replace loops in non-recursive functions.

• It’s also possible to have more than one base or recursive case, if the program might recurse or terminate in different ways, depending on the input being passed in.
Recursion

• **Multiple base cases:** The Fibonacci number sequence is defined as follows:
  • The first element is 0.
  • The second element is 1.
  • The $n^{th}$ element is the sum of the $(n-1)^{th}$ and $(n-2)^{th}$ elements.

• **Multiple recursive cases:** The Collatz conjecture.
Recursion

The Collatz conjecture is applies to positive integers and speculates that it is always possible to get “back to 1” if you follow these steps:

- If $n$ is 1, stop.
- Otherwise, if $n$ is even, repeat this process on $n/2$.
- Otherwise, if $n$ is odd, repeat this process on $3n + 1$.

Write a recursive function $\text{collatz}(n)$ that calculates how many steps it takes to get to 1 if you start from $n$ and recurse as indicated above.
<table>
<thead>
<tr>
<th>n</th>
<th>collatz(n)</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2 → 1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3 → 10 → 5 → 16 → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4 → 2 → 1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5 → 16 → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>15</td>
<td>17</td>
<td>15 → 46 → 23 → 70 → ... → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>27</td>
<td>111</td>
<td>27 → 82 → 41 → 124 → ... → 8 → 4 → 2 → 1</td>
</tr>
<tr>
<td>50</td>
<td>24</td>
<td>50 → 25 → 76 → 38 → ... → 8 → 4 → 2 → 1</td>
</tr>
</tbody>
</table>
Recursion

```c
int collatz(int n)
{
    // base case
    if (n == 1)
        return 0;
    // even numbers
    else if ((n % 2) == 0)
        return 1 + collatz(n/2);
    // odd numbers
    else
        return 1 + collatz(3*n + 1);
}
```