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

```
fact(1) = 1
fact(2) = 2 * 1
fact(3) = 3 * 2 * 1
fact(4) = 4 * 3 * 2 * 1
fact(5) = 5 * 4 * 3 * 2 * 1
...
```

Recursion

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * 2 * 1
fact(4) = 4 * 3 * 2 * 1
fact(5) = 5 * 4 * 3 * 2 * 1
...
```

Recursion

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * fact(2)
fact(4) = 4 * 3 * 2 * 1
fact(5) = 5 * 4 * 3 * 2 * 1
...
```

Recursion

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * fact(2)
fact(4) = 4 * fact(3)
fact(5) = 5 * 4 * 3 * 2 * 1
...
```

Recursion

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * fact(2)
fact(4) = 4 * fact(3)
fact(5) = 5 * fact(4)
...
```

Recursion

```
fact(n) = n * 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

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * fact(2)
fact(4) = 4 * fact(3)
fact(5) = 5 * fact(4)
...
```

Recursion

```
int fact(int n)
{
	// base case

	// recursive case
}
```

Recursion

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

    // recursive case
}
```

Recursion

```
fact(1) = 1
fact(2) = 2 * fact(1)
fact(3) = 3 * fact(2)
fact(4) = 4 * fact(3)
fact(5) = 5 * fact(4)
...
```

Recursion

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

    // recursive case
}
```

Recursion

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

Recursion

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

Recursion

```
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.

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

```
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 3*n* + 1.

- Write a recursive function `collatz(n)` that calculates how many steps it takes to get to 1 if you start from n and recurse as indicated above.

Recursion

n	`collatz(n)`	Steps
1	0	1
2	1	2 → 1
3	7	3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
4	2	4 → 2 → 1
5	5	5 → 16 → 8 → 4 → 2 → 1
6	8	6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
7	16	7 → 22 → 11 → 34 → 17 → 52 → 26 → 13 → 40 → 20 → 10 → 5 → 16 → 8 → 4 → 2 → 1
8	3	8 → 4 → 2 → 1
15	17	15 → 46 → 23 → 70 → … → 8 → 4 → 2 → 1
27	111	27 → 82 → 41 → 124 → … → 8 → 4 → 2 → 1
50	24	50 → 25 → 76 → 38 → … → 8 → 4 → 2 → 1

Recursion

```
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);
}
```