Getting Started with Recursion

Outline for Today

- ***Recursive Functions***
  - A new problem-solving perspective.
- ***Recursion on Strings***
  - Featuring cute animals!

Thinking Recursively

Factorials!

- The number ***n factorial***, denoted ***n!***, is defined as

$$n \times (n - 1) \times \ldots \times 3 \times 2 \times 1$$

- Here's some examples!
  - $3! = 3 \times 2 \times 1 = 6$.
  - $4! = 4 \times 3 \times 2 \times 1 = 24$.
  - $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
  - $0! = 1$. (by definition!)
- Factorials show up in unexpected places! We'll see one later when we talk about sorting algorithms!
- Let's implement a function to compute factorials!

Computing Factorials

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!}$$

Computing Factorials

$$5! = 5 \times 4!$$

Computing Factorials

$$5! = 5 \times 4!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times \underbrace{3 \times 2 \times 1}_{3!}$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2 \times 1$$

$$2 \times 1 = 2!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = \mathbf{2}$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2$$

$$2! = 2$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = \mathbf{6}$$

$$2! = 2$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 6$$

$$2! = 2$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$5! = 5 \times 4!$

$4! = 4 \times \mathbf{6}$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$5! = 5 \times 4!$

$4! = \mathbf{24}$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 24$$

$$3! = 6$$

$$2! = 2$$

$$1! = 1$$

$$0! = 1$$

Computing Factorials

$5! = 5 \times \mathbf{24}$

$4! = 24$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$5! = \mathbf{120}$

$4! = 24$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$5! = 120$

$4! = 24$

$3! = 6$

$2! = 2$

$1! = 1$

$0! = 1$

Computing Factorials

$$5! = 5 \times 4!$$

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1 \times 0!$$

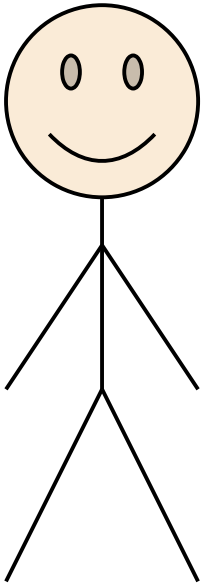
$$0! = 1$$

Another View of Factorials

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

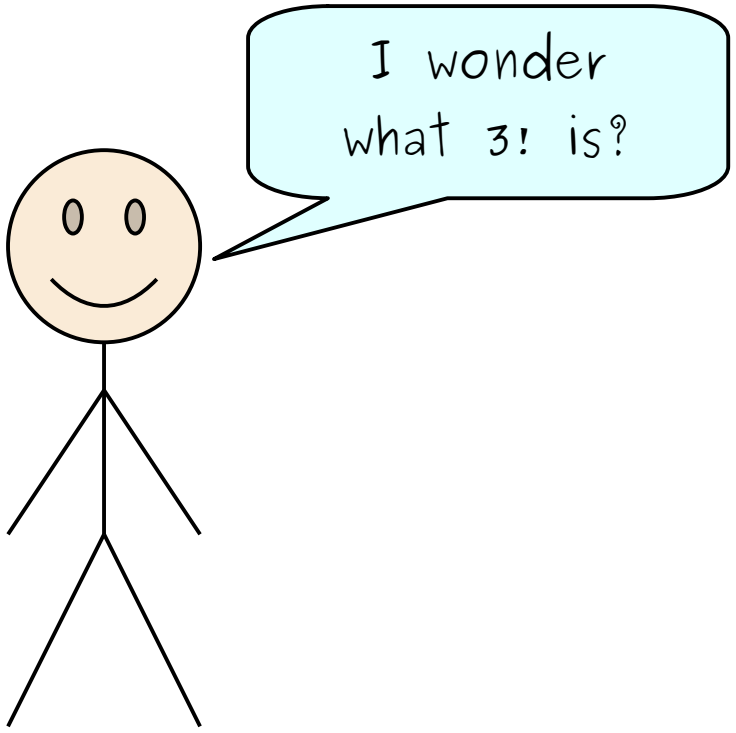
Alexes Compute Factorials

Alexes Compute Factorials



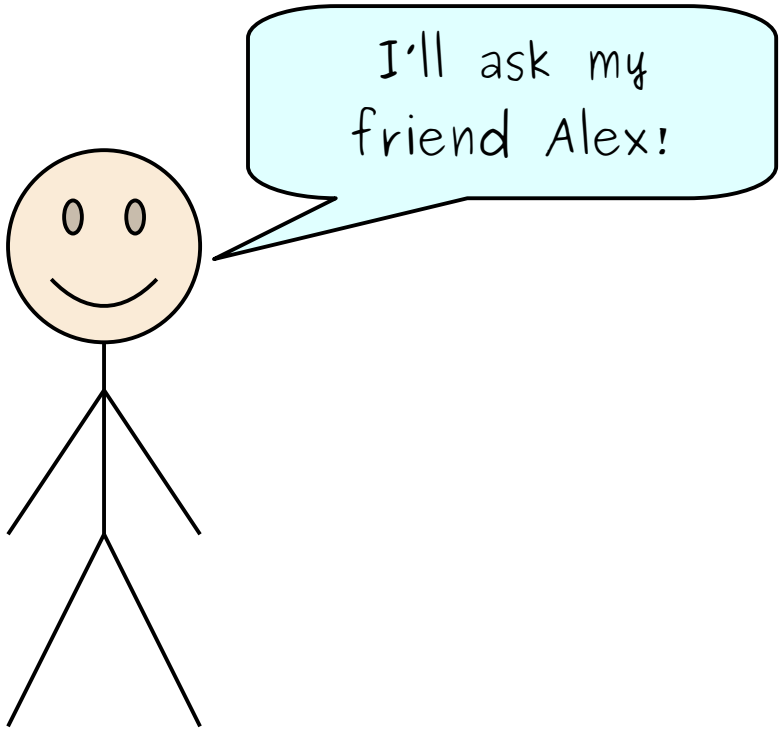
Me!

Alexes Compute Factorials

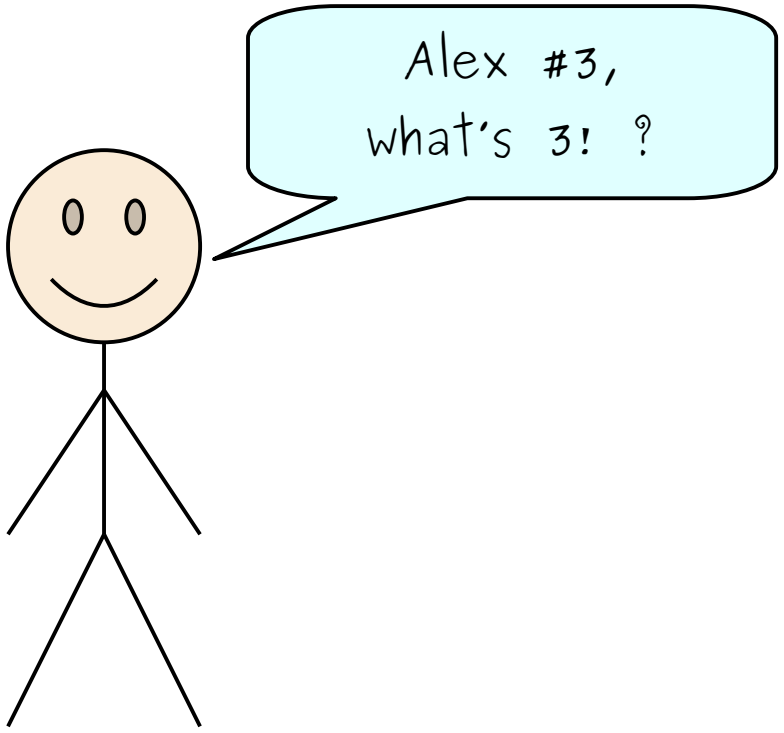


Me!

Alexes Compute Factorials

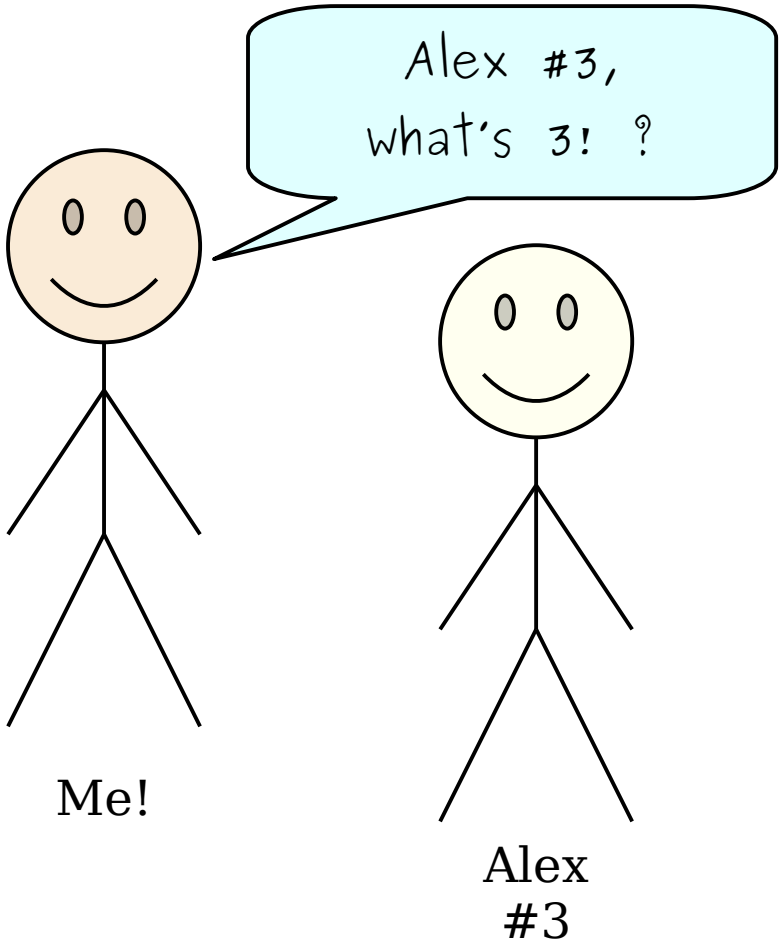


Alexes Compute Factorials

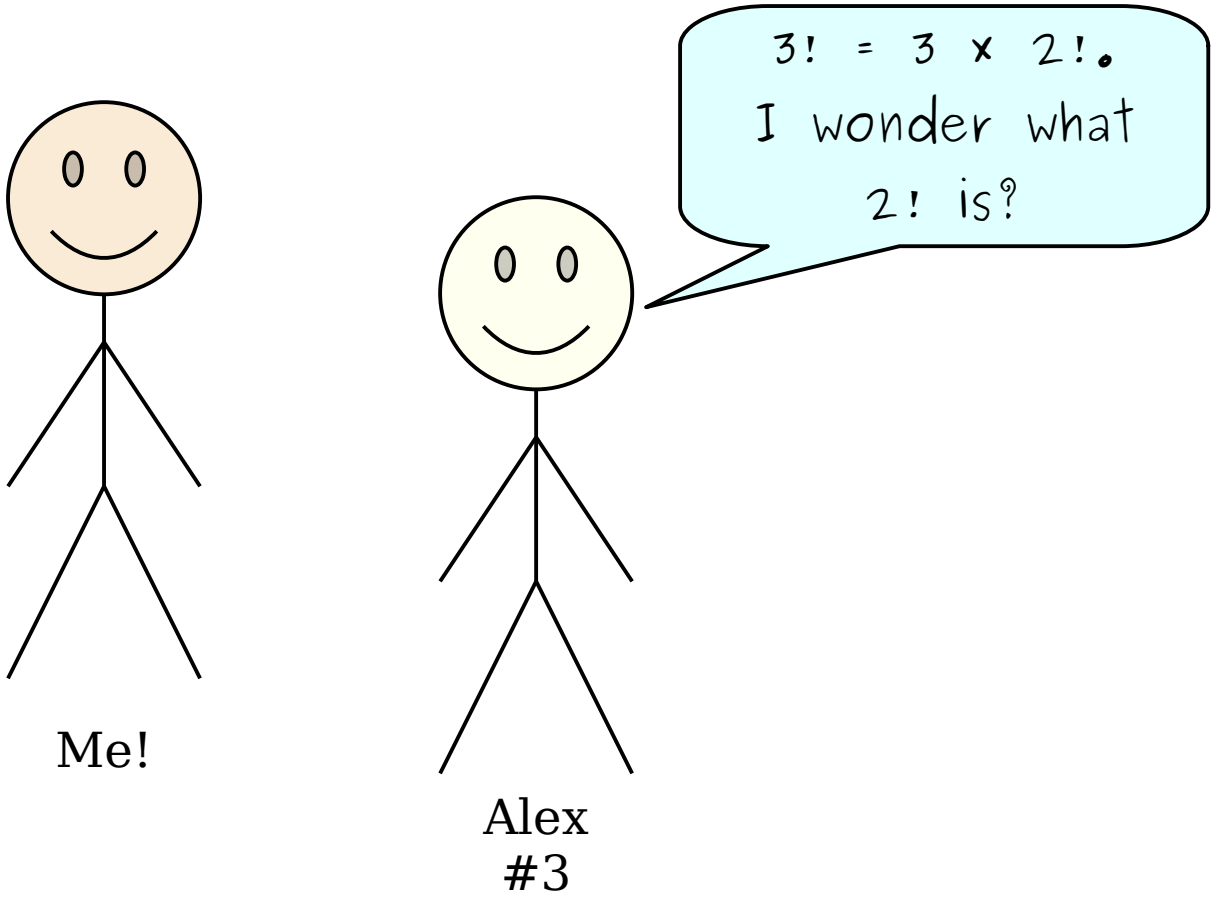


Me!

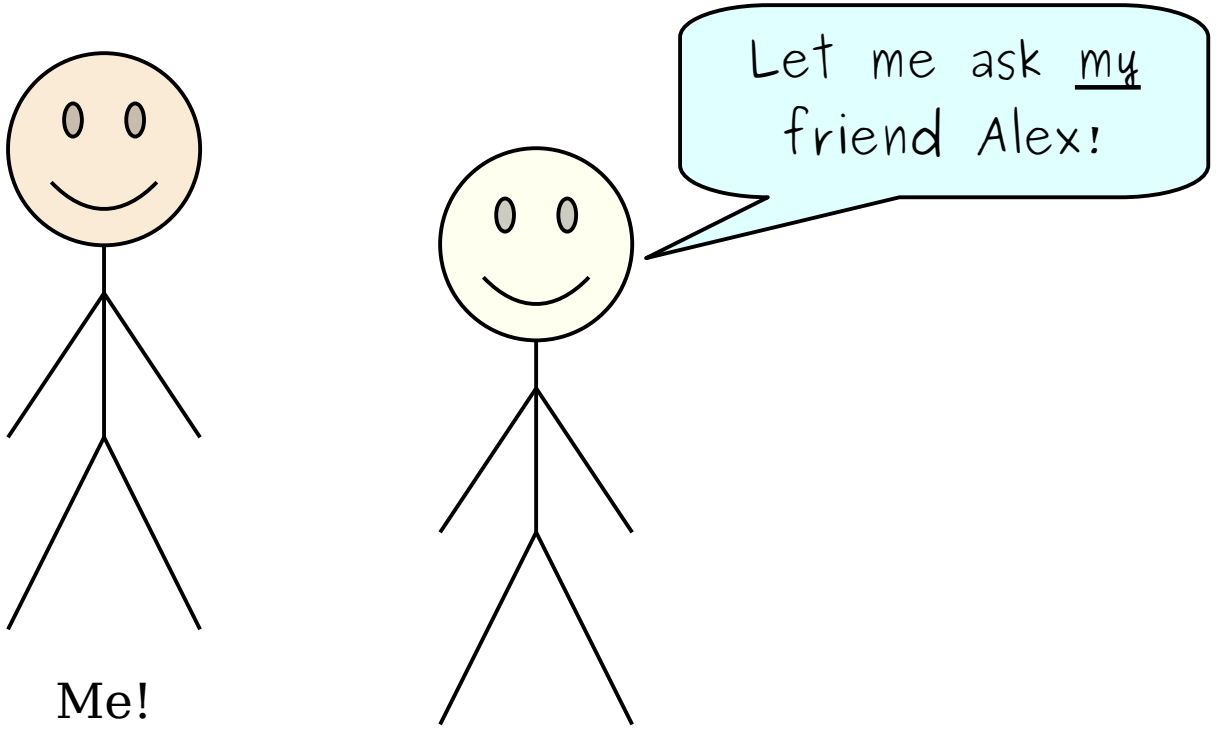
Alexes Compute Factorials



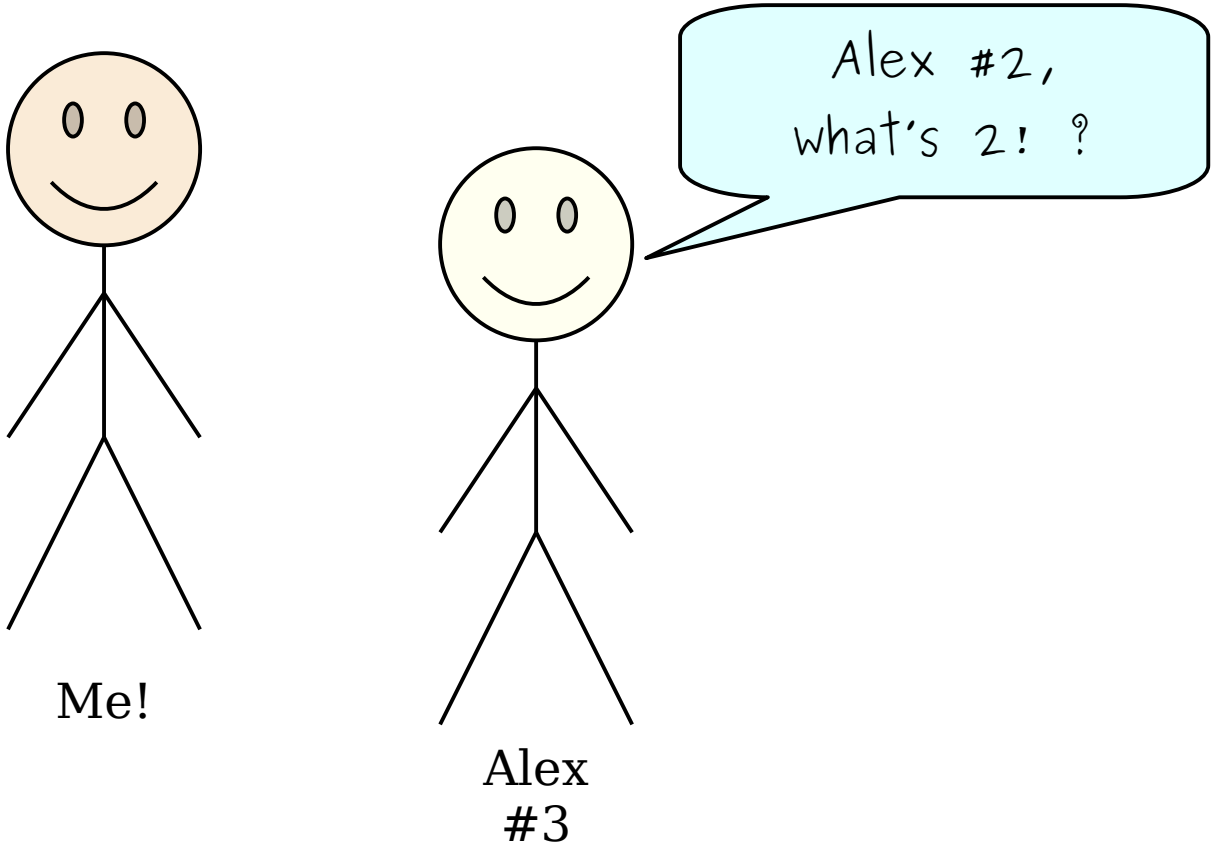
Alexes Compute Factorials



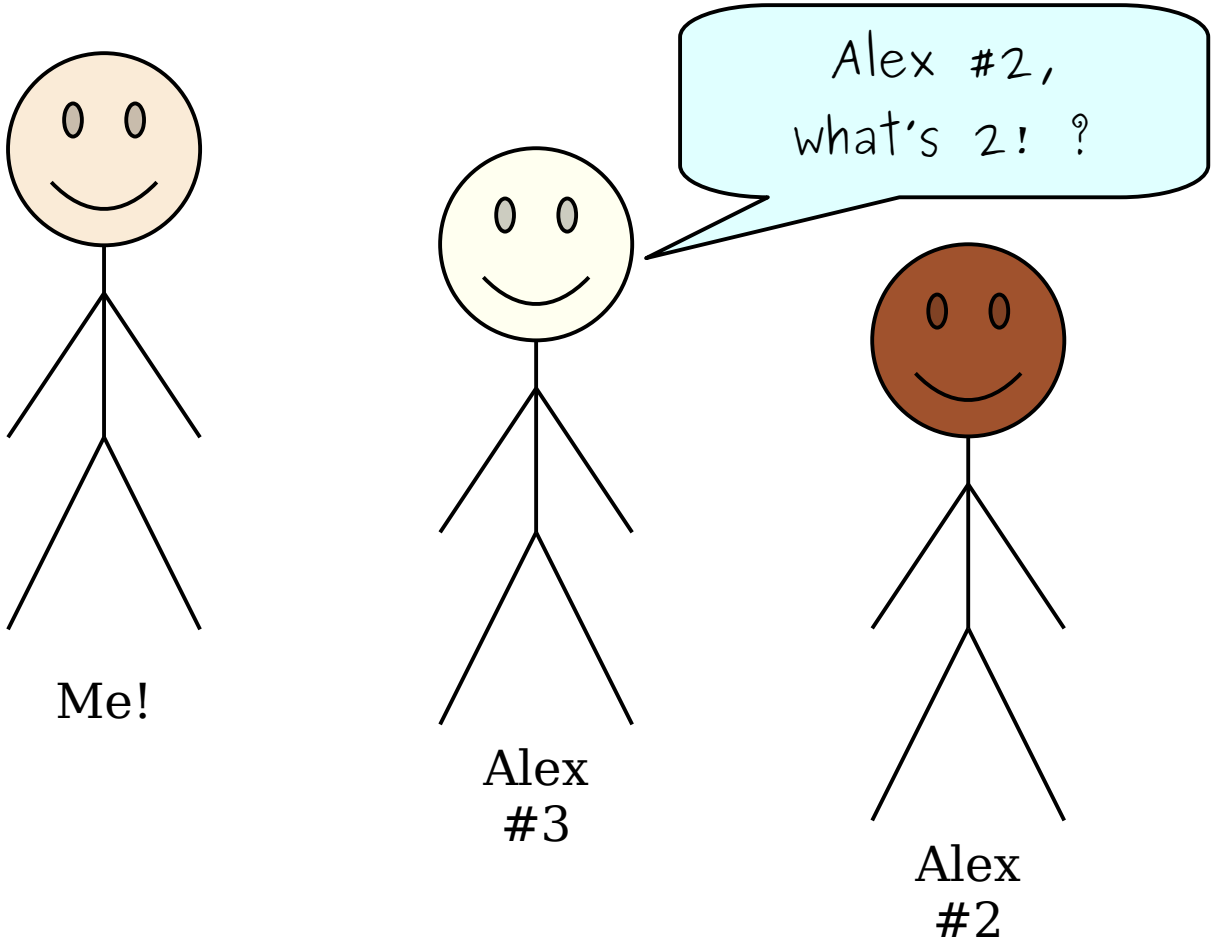
Alexes Compute Factorials



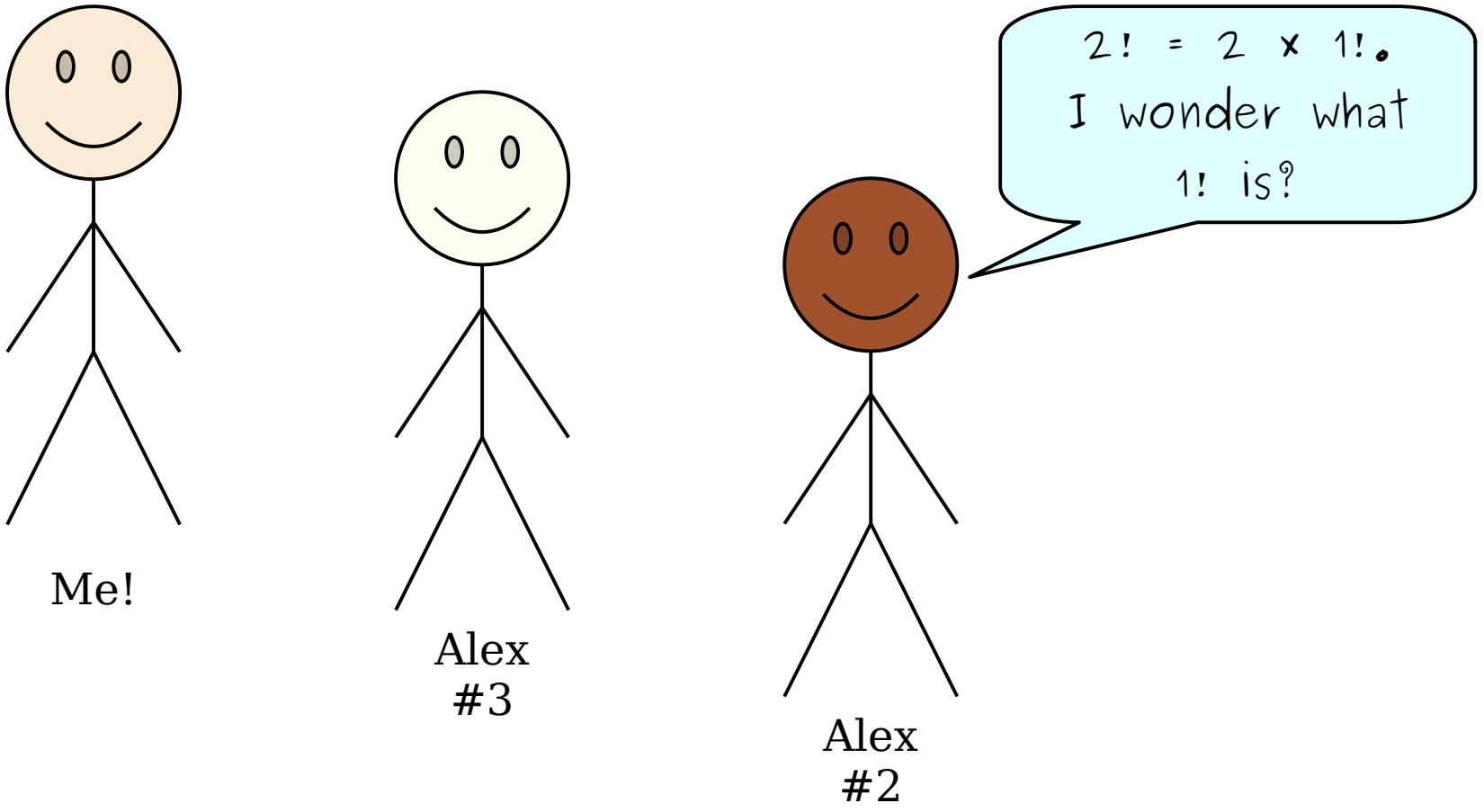
Alexes Compute Factorials



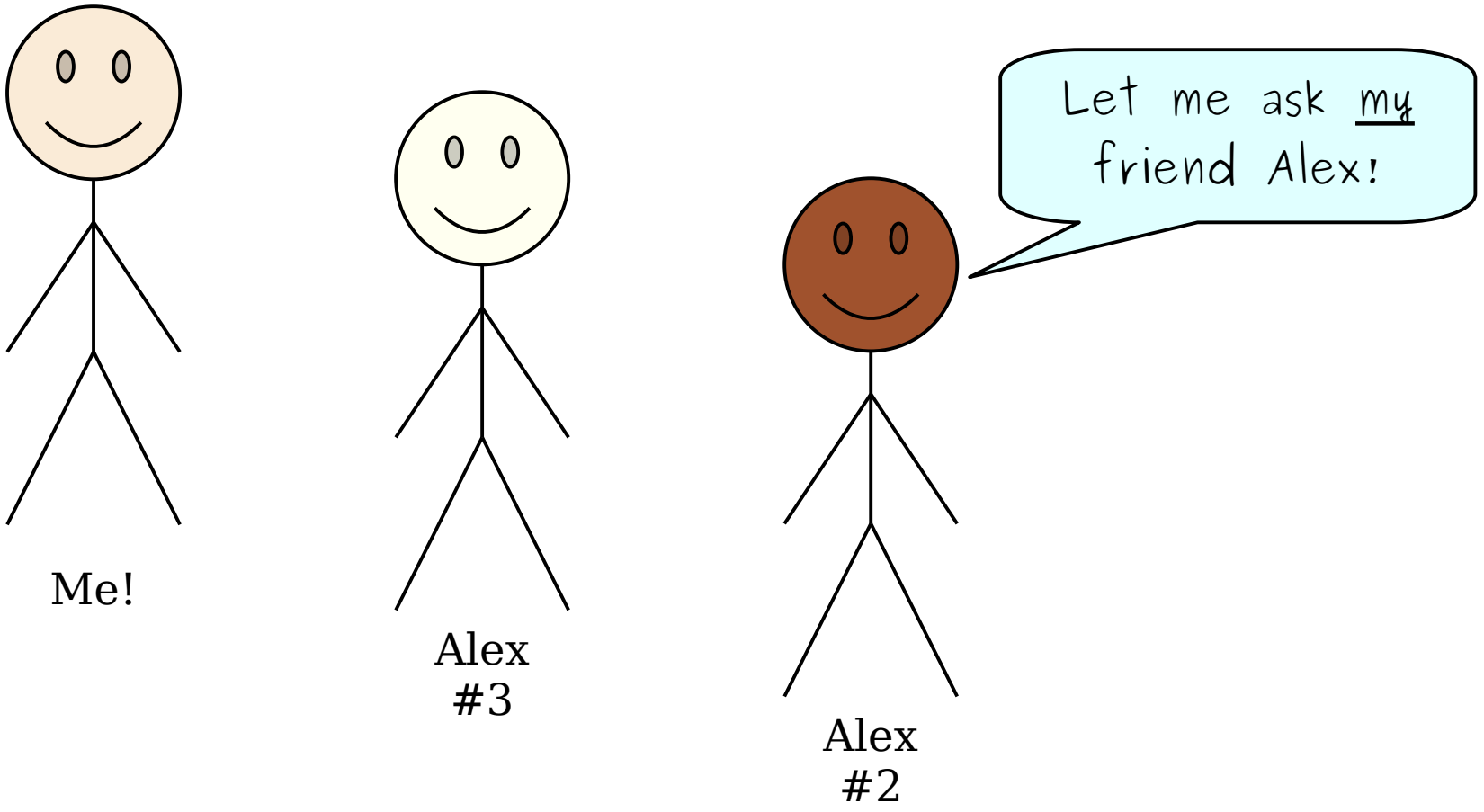
Alexes Compute Factorials



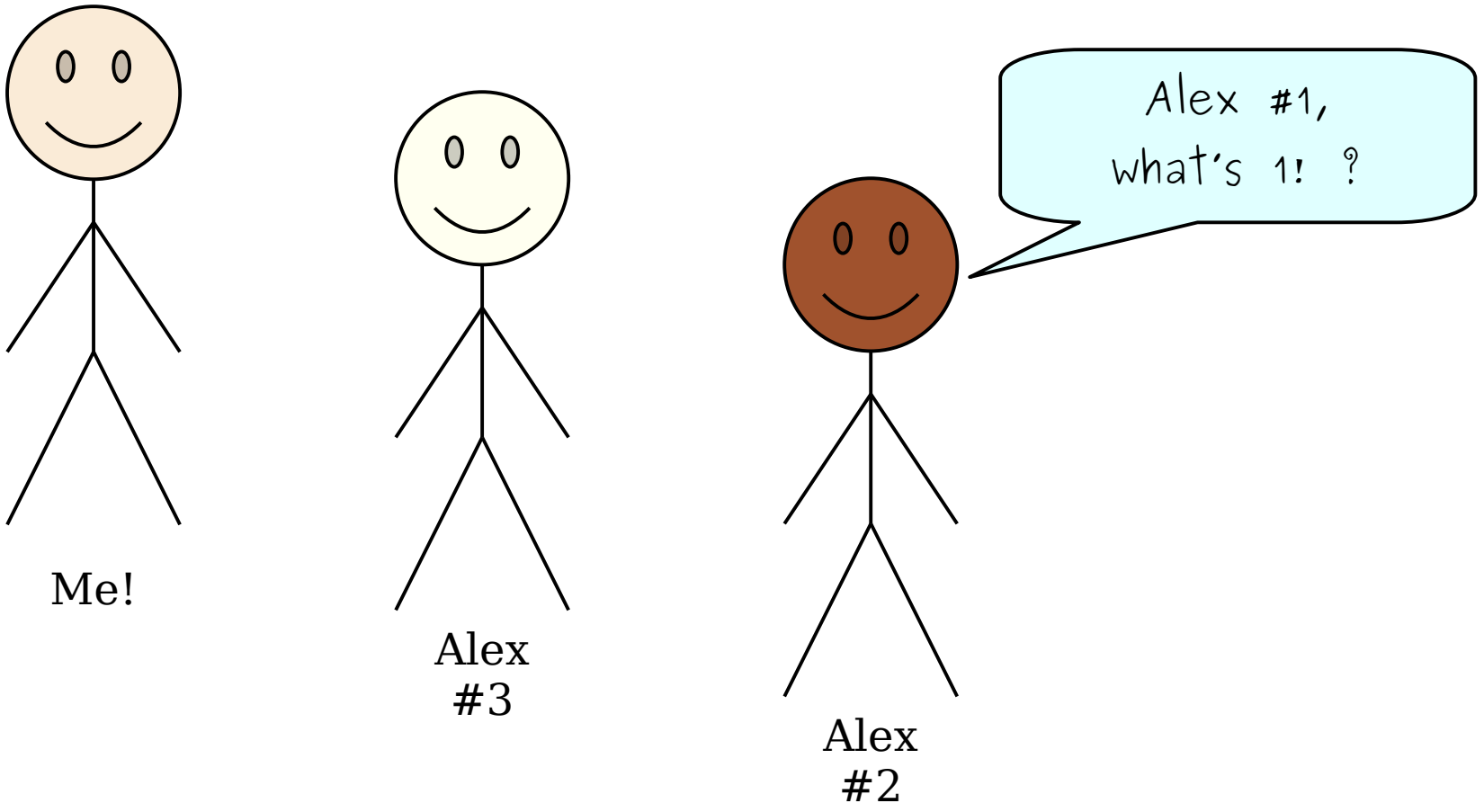
Alexes Compute Factorials



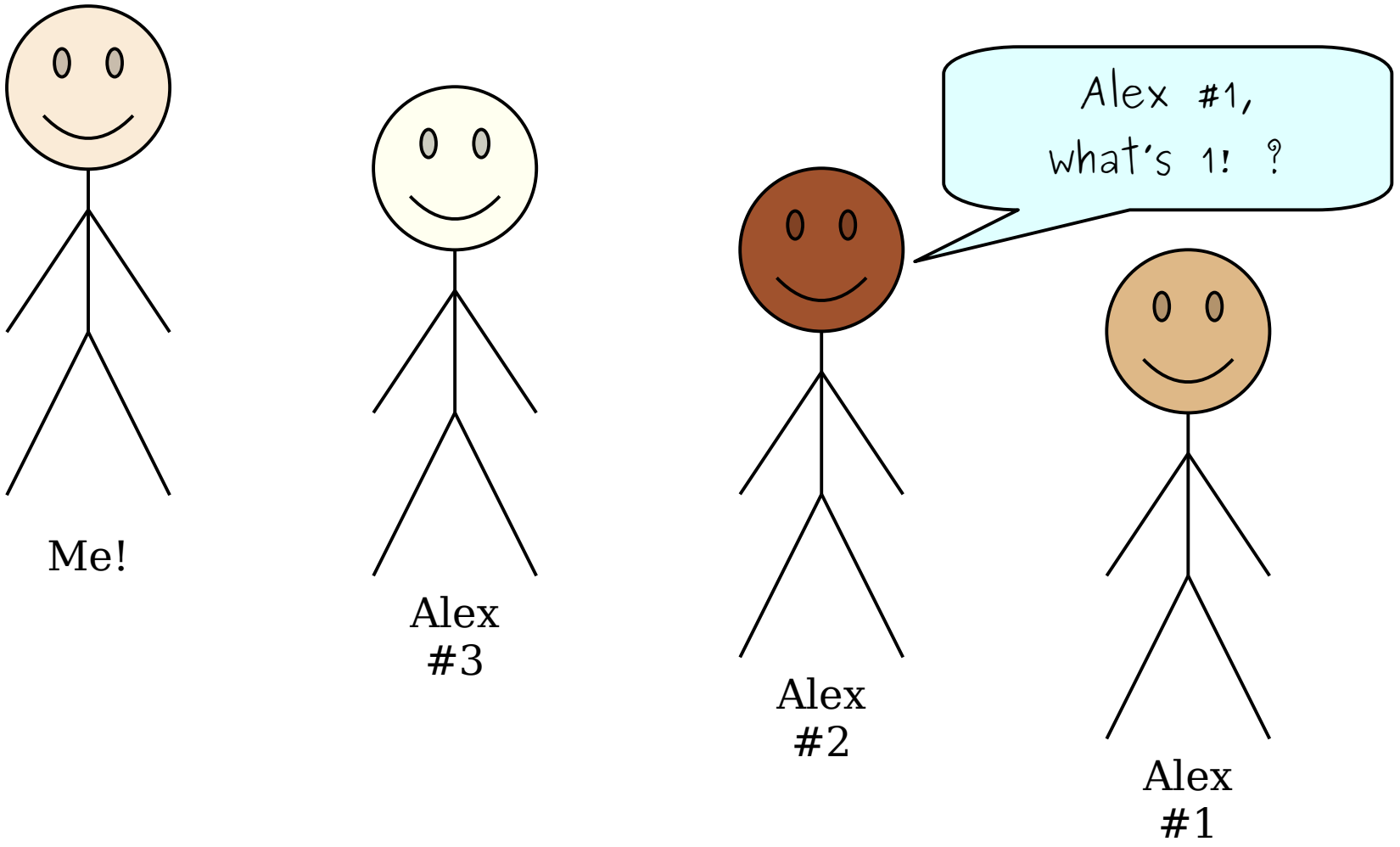
Alexes Compute Factorials



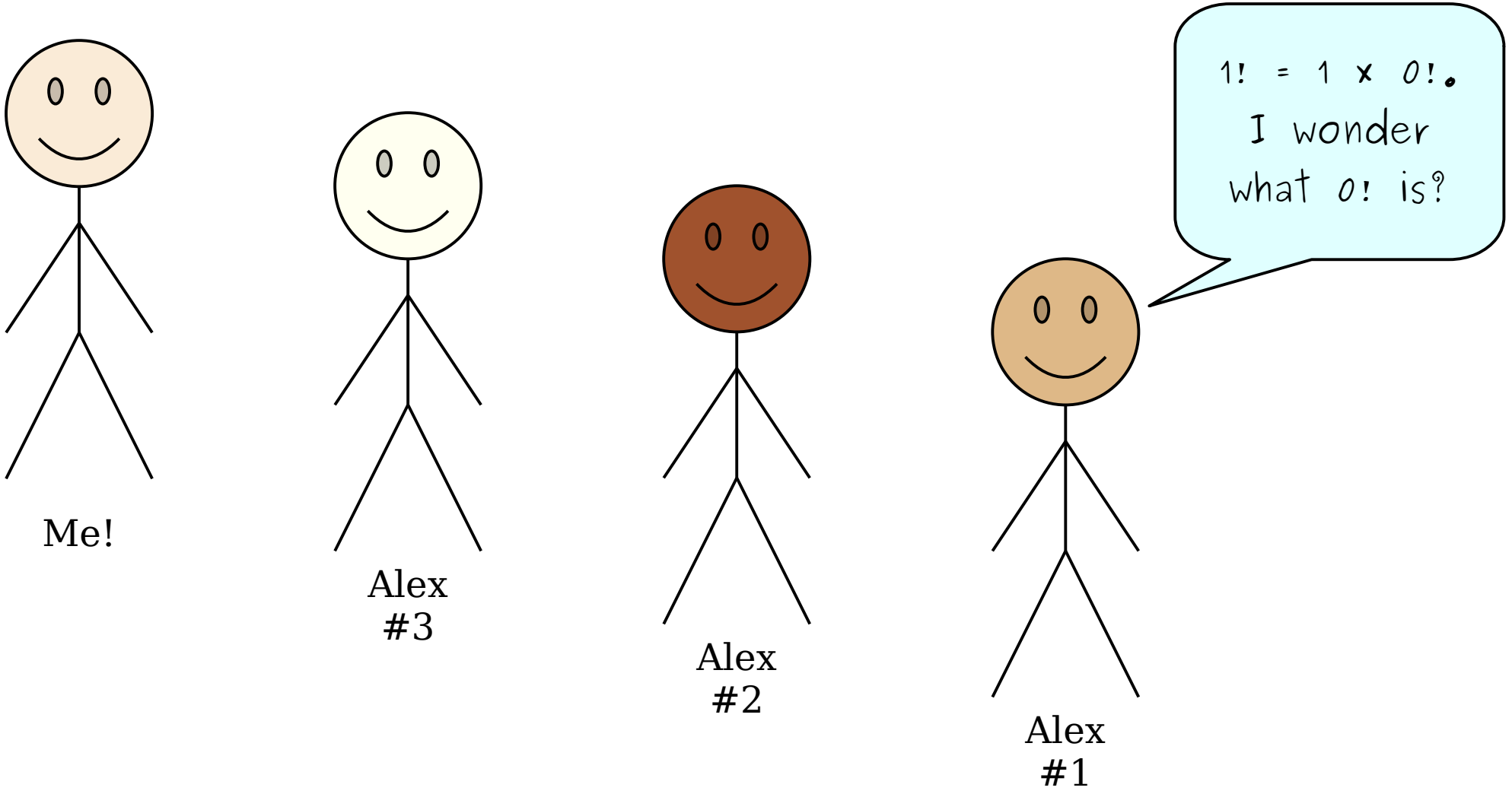
Alexes Compute Factorials



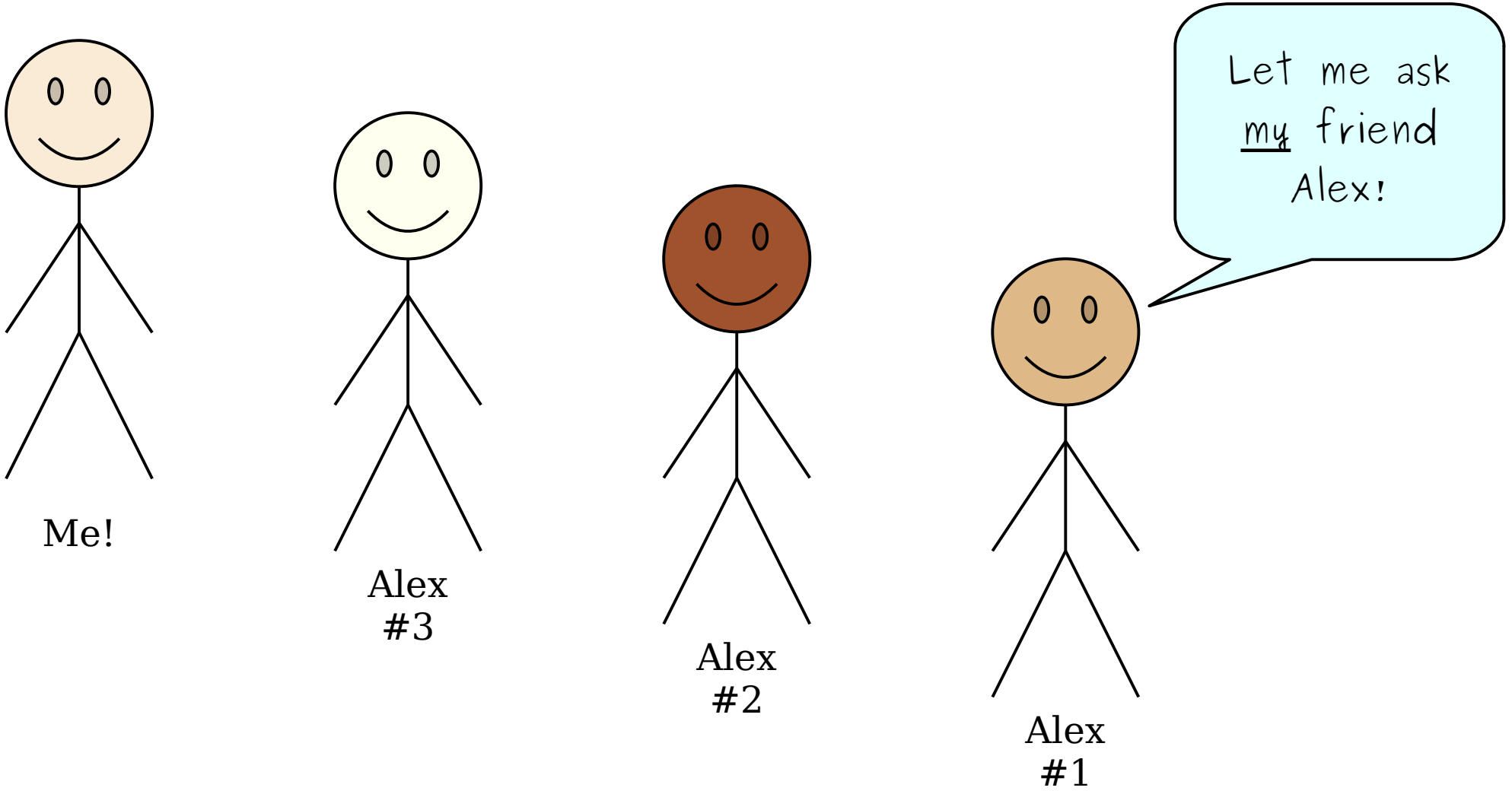
Alexes Compute Factorials



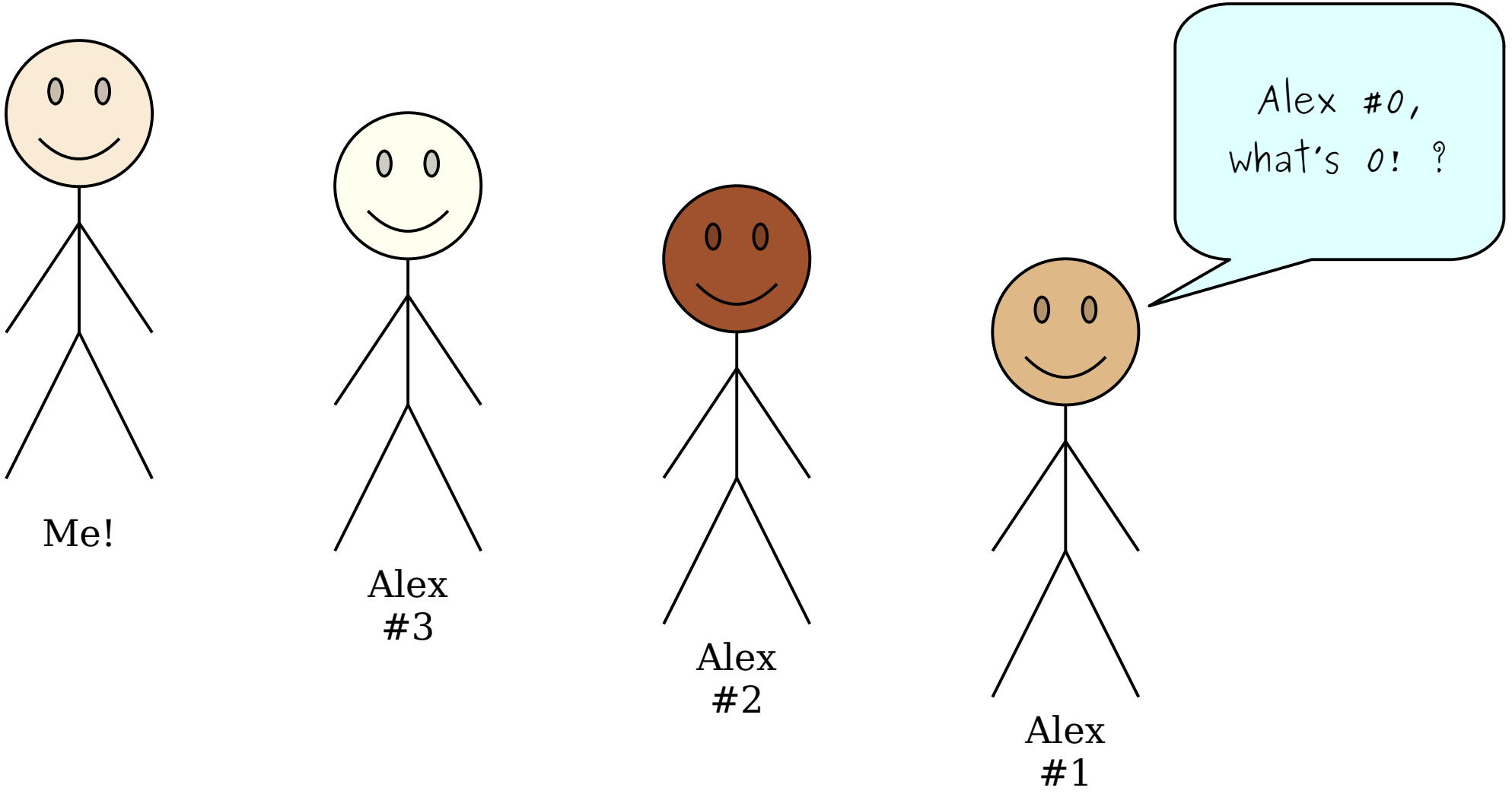
Alexes Compute Factorials



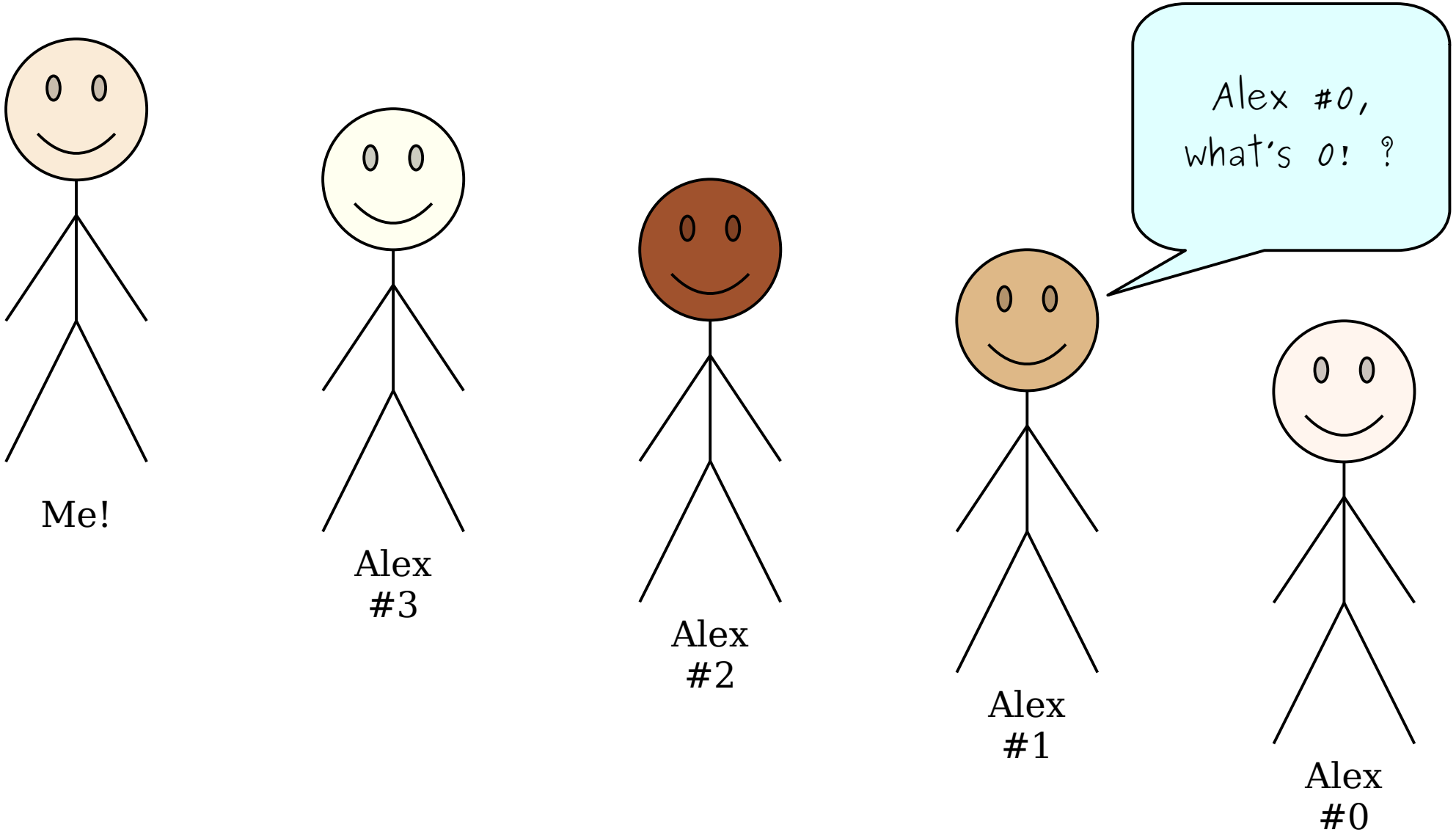
Alexes Compute Factorials



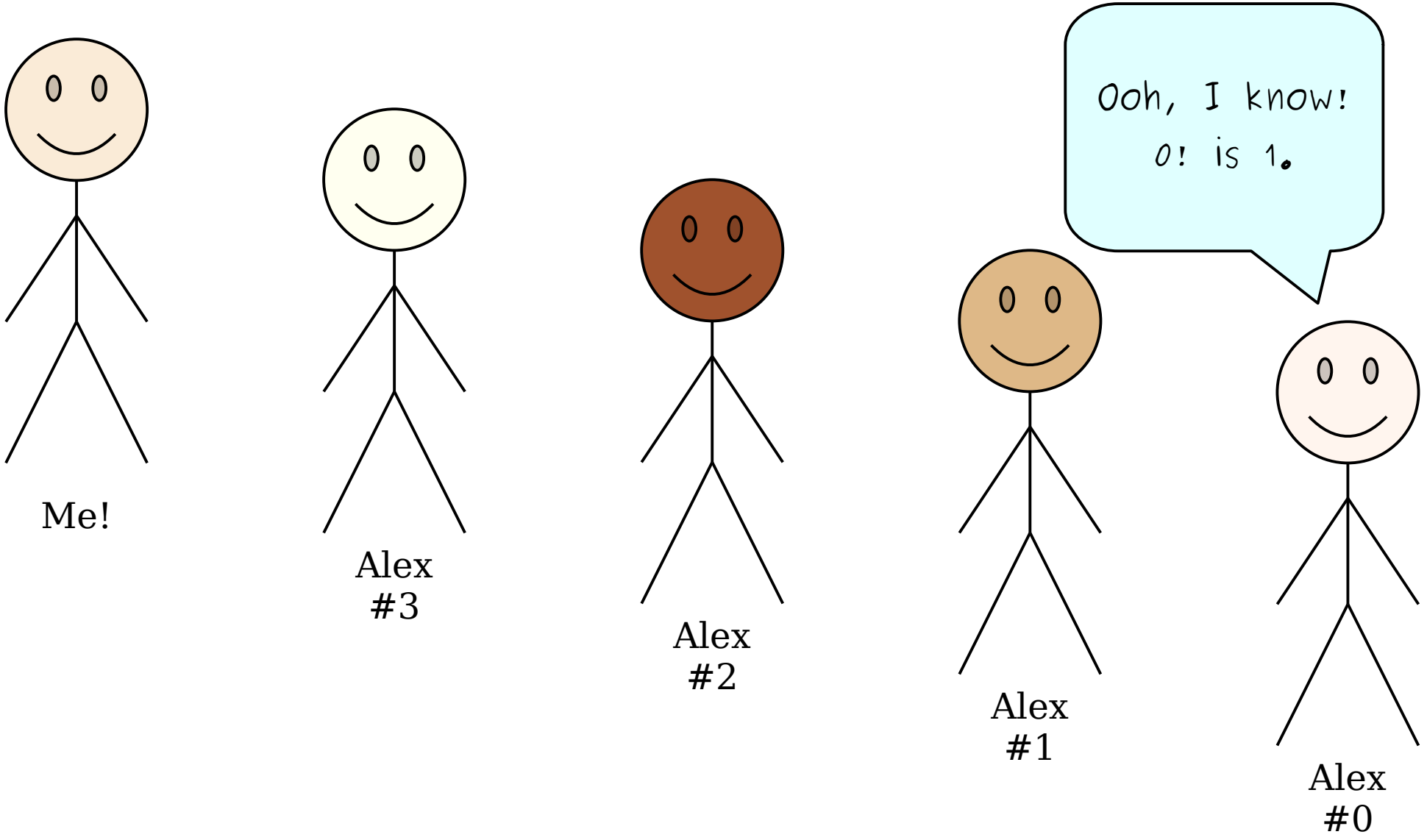
Alexes Compute Factorials



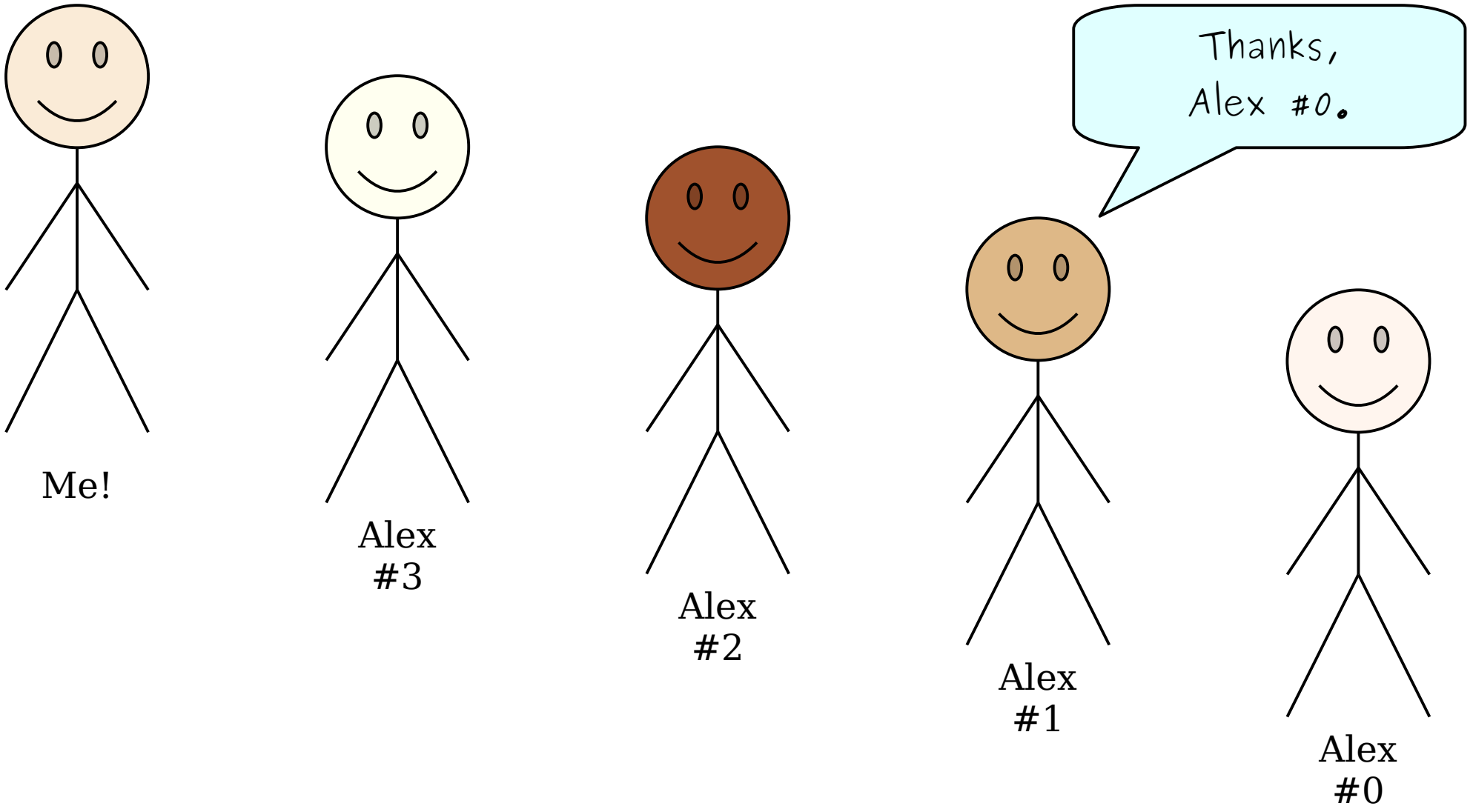
Alexes Compute Factorials



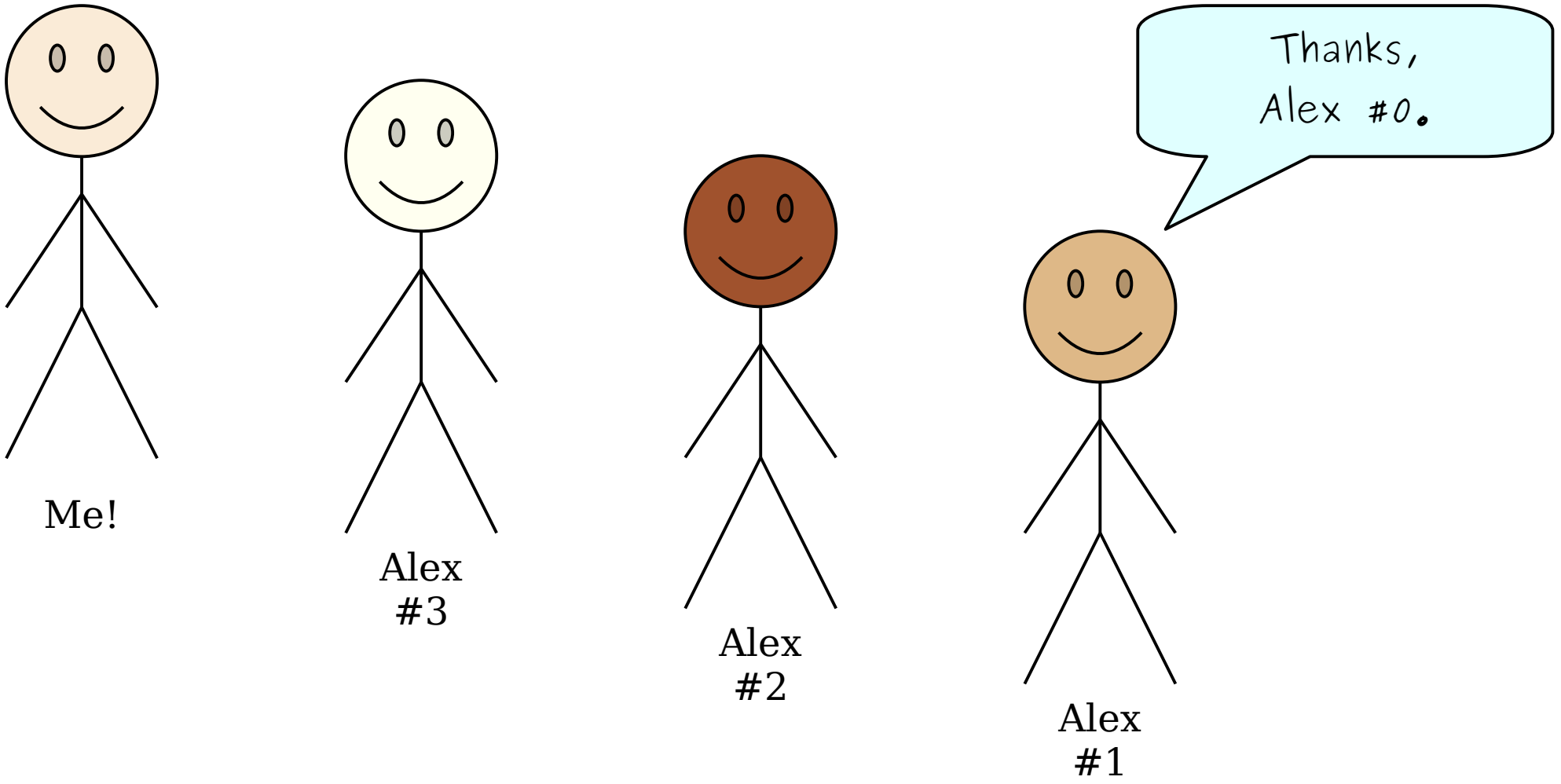
Alexes Compute Factorials



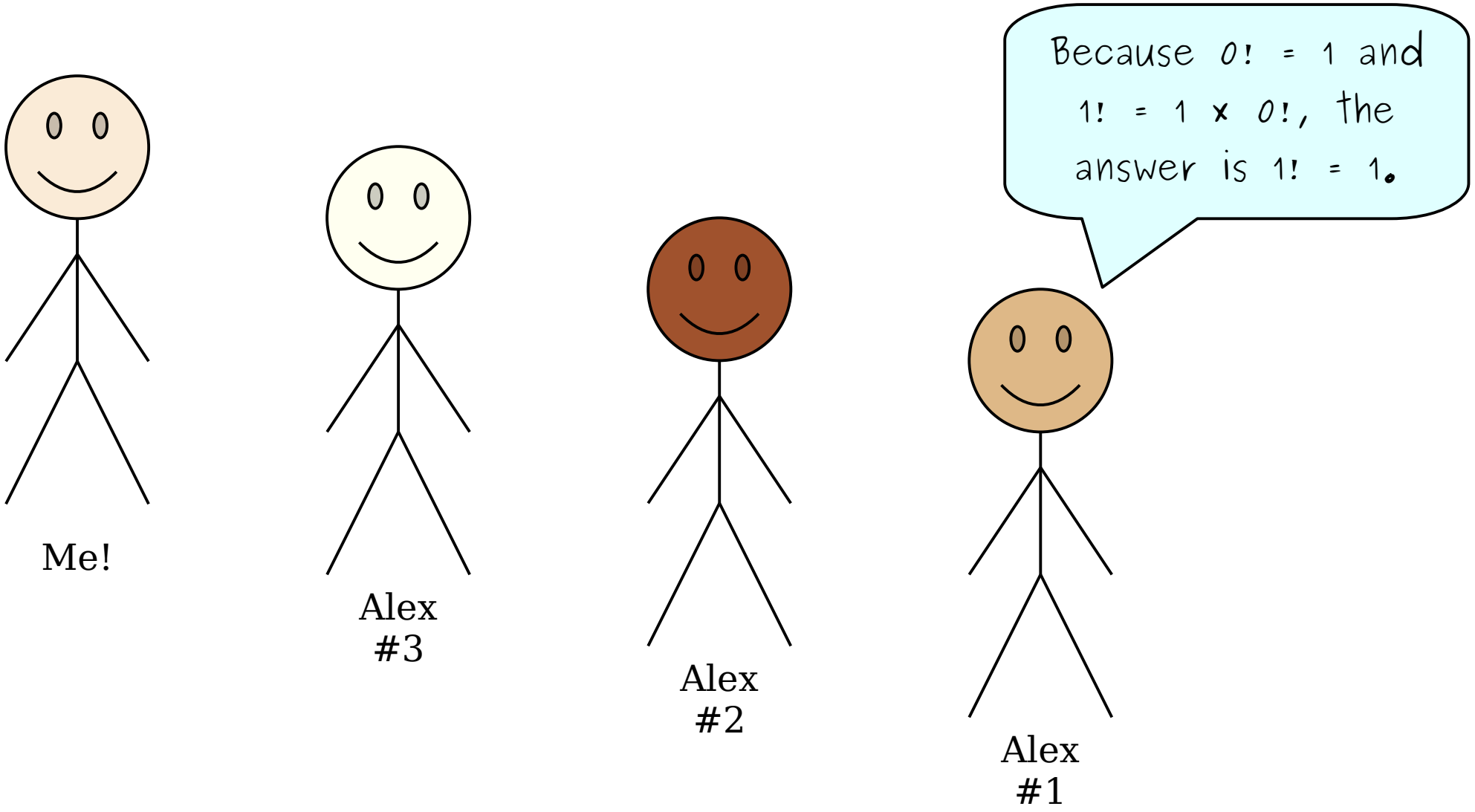
Alexes Compute Factorials



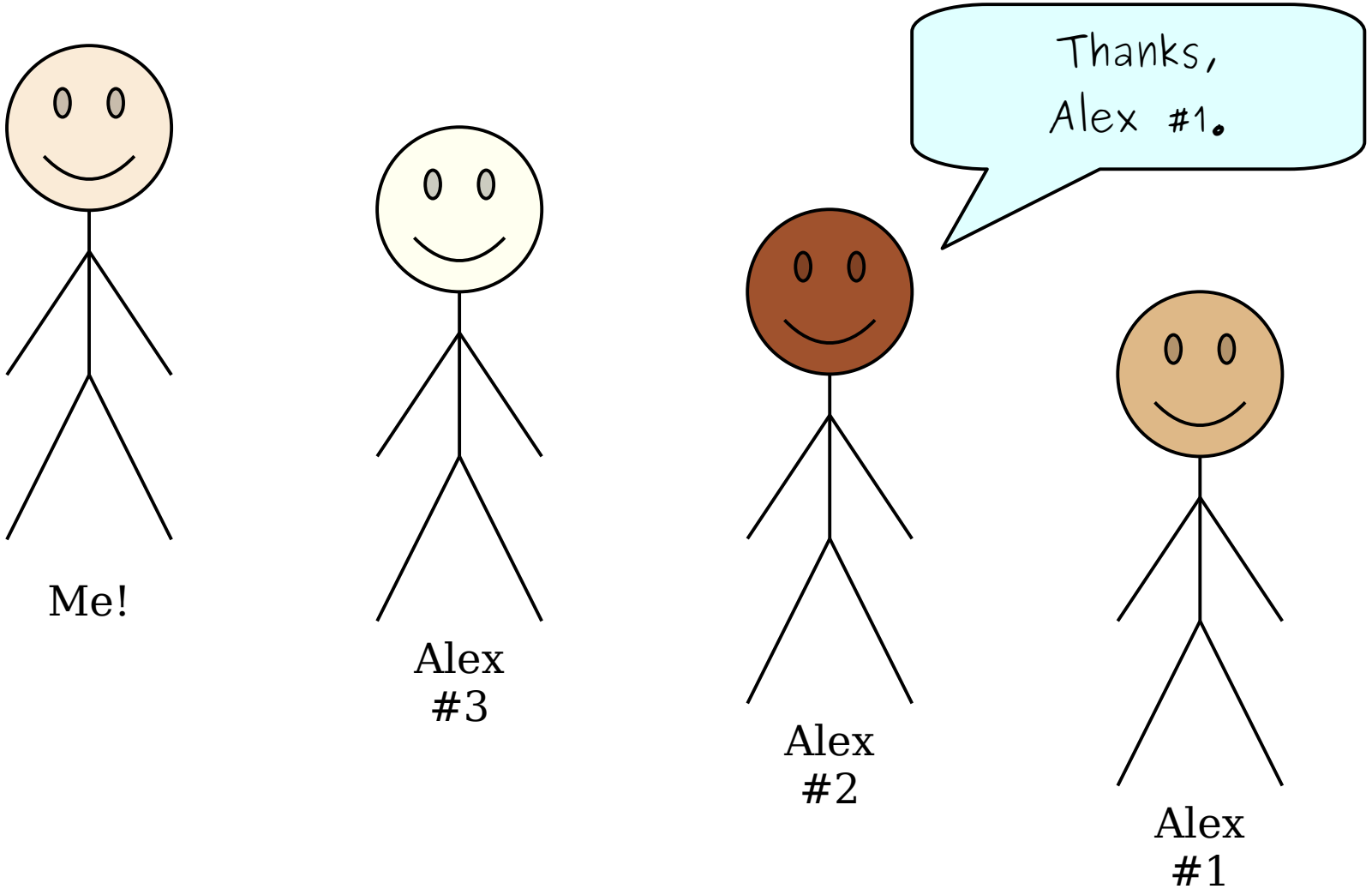
Alexes Compute Factorials



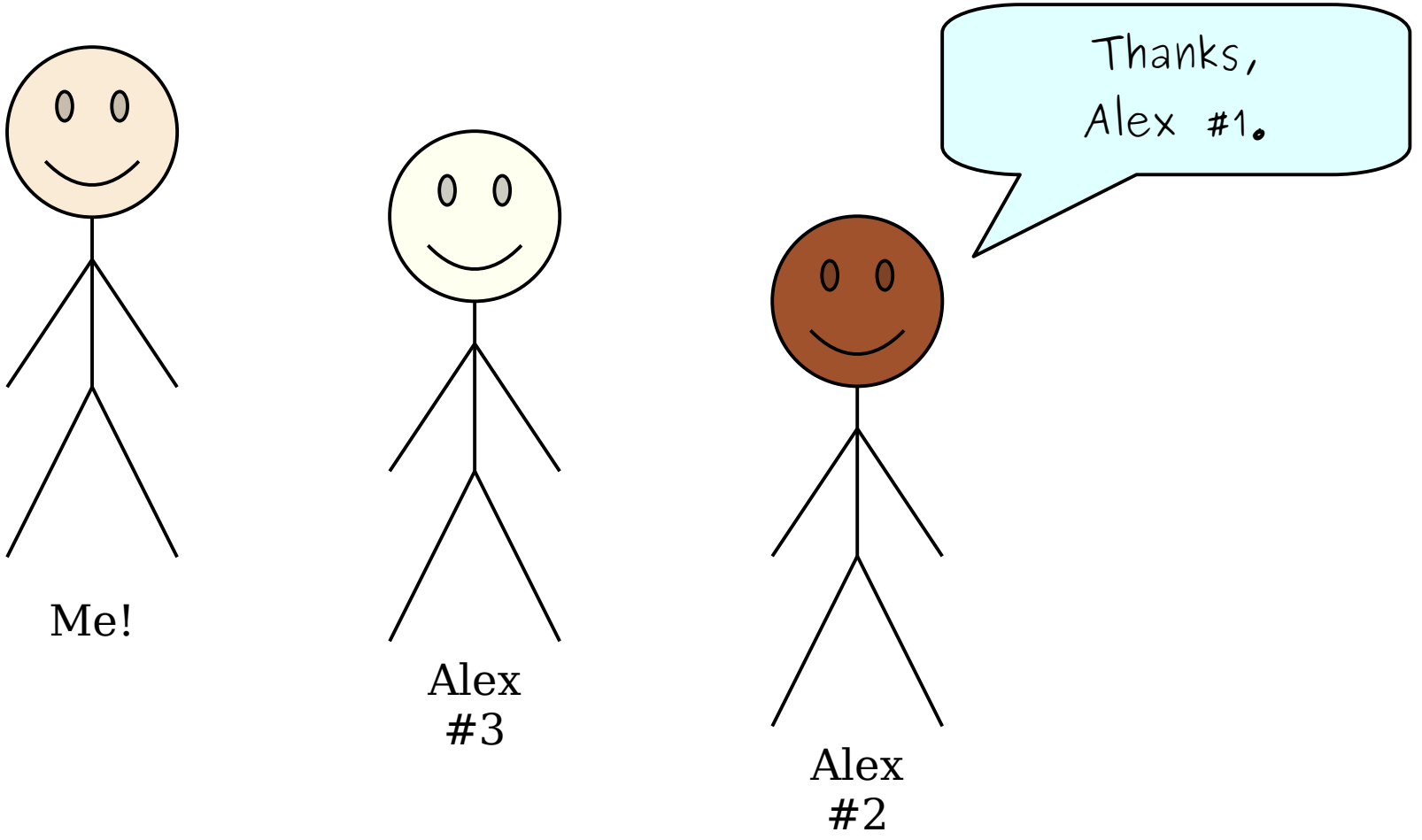
Alexes Compute Factorials



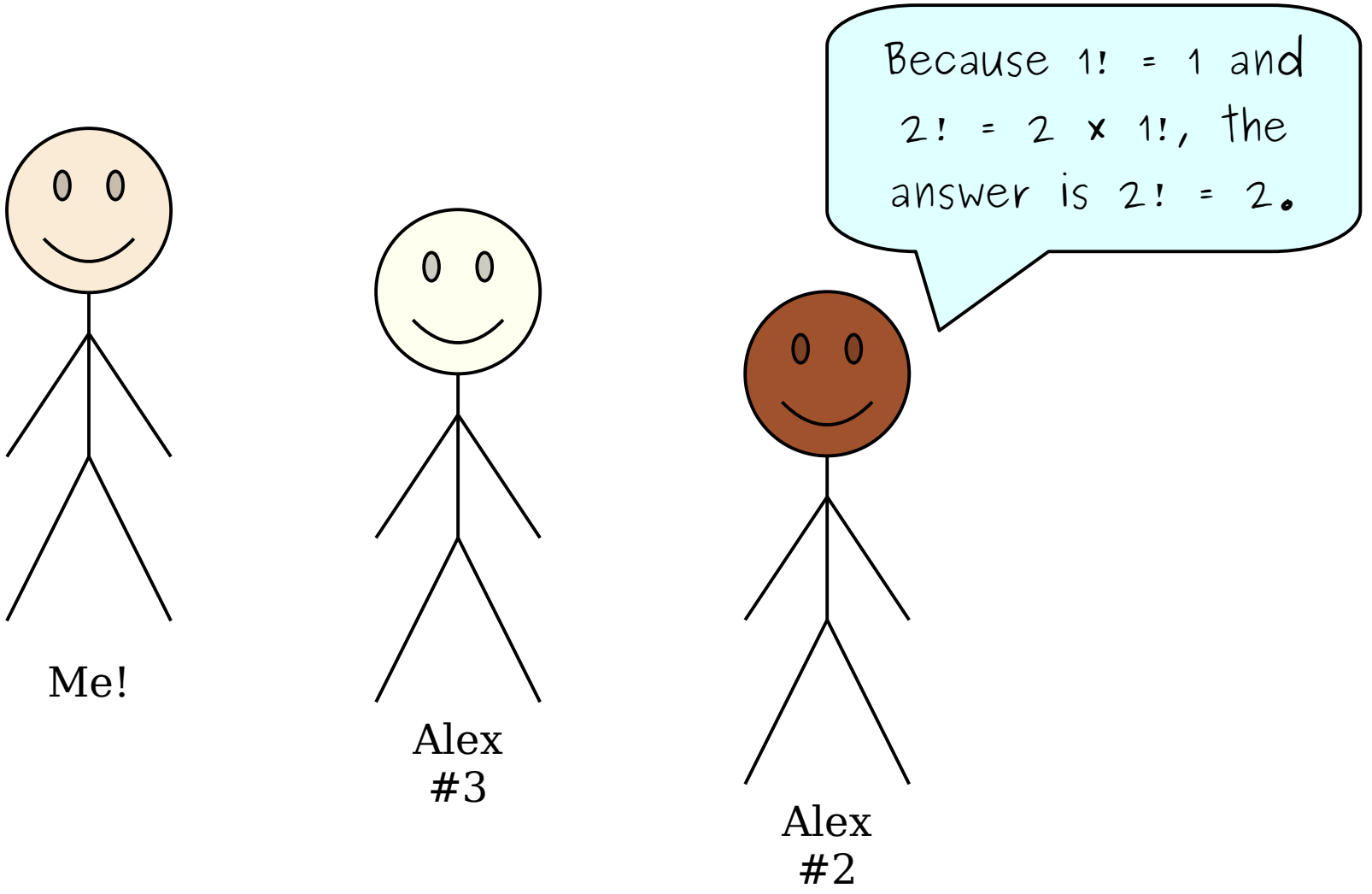
Alexes Compute Factorials



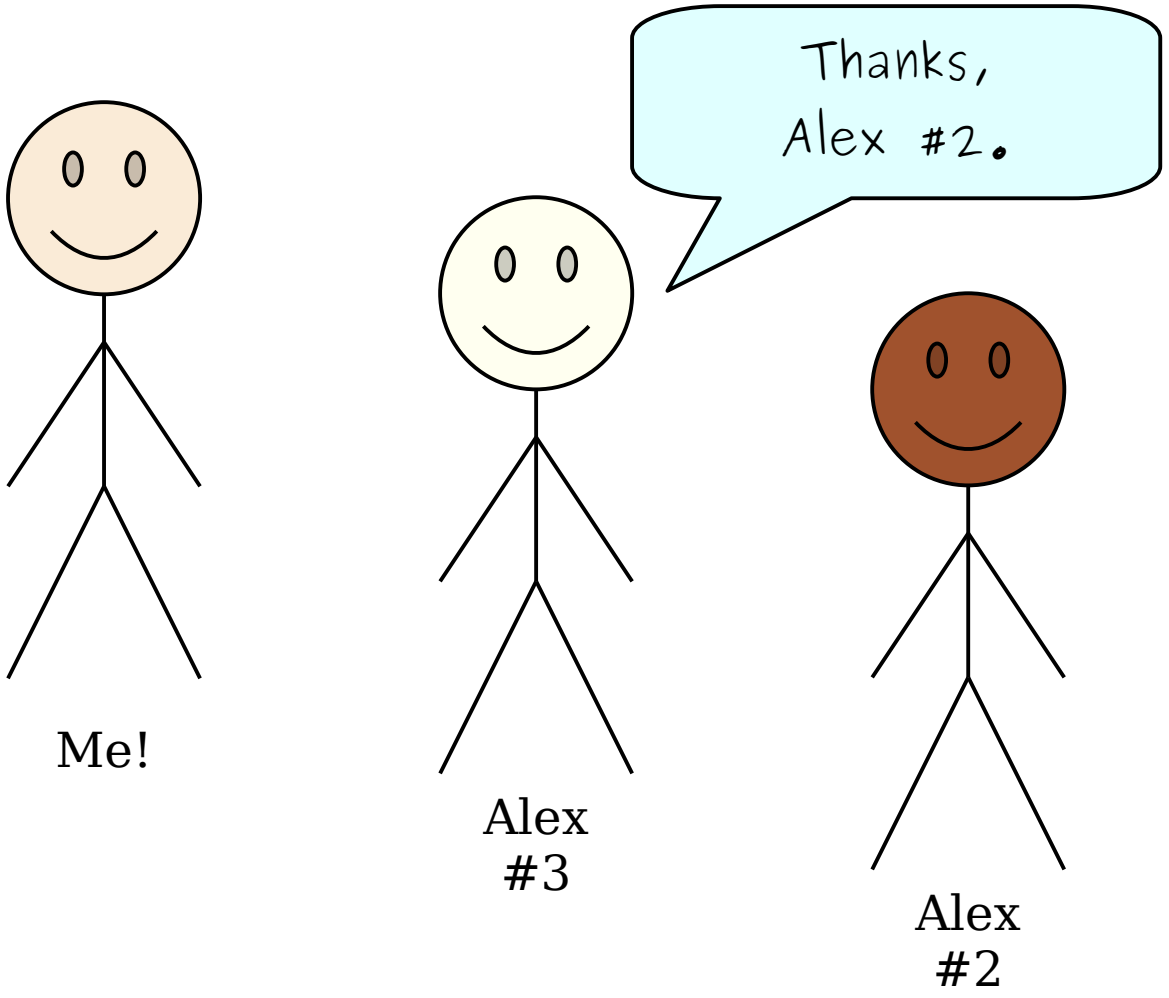
Alexes Compute Factorials



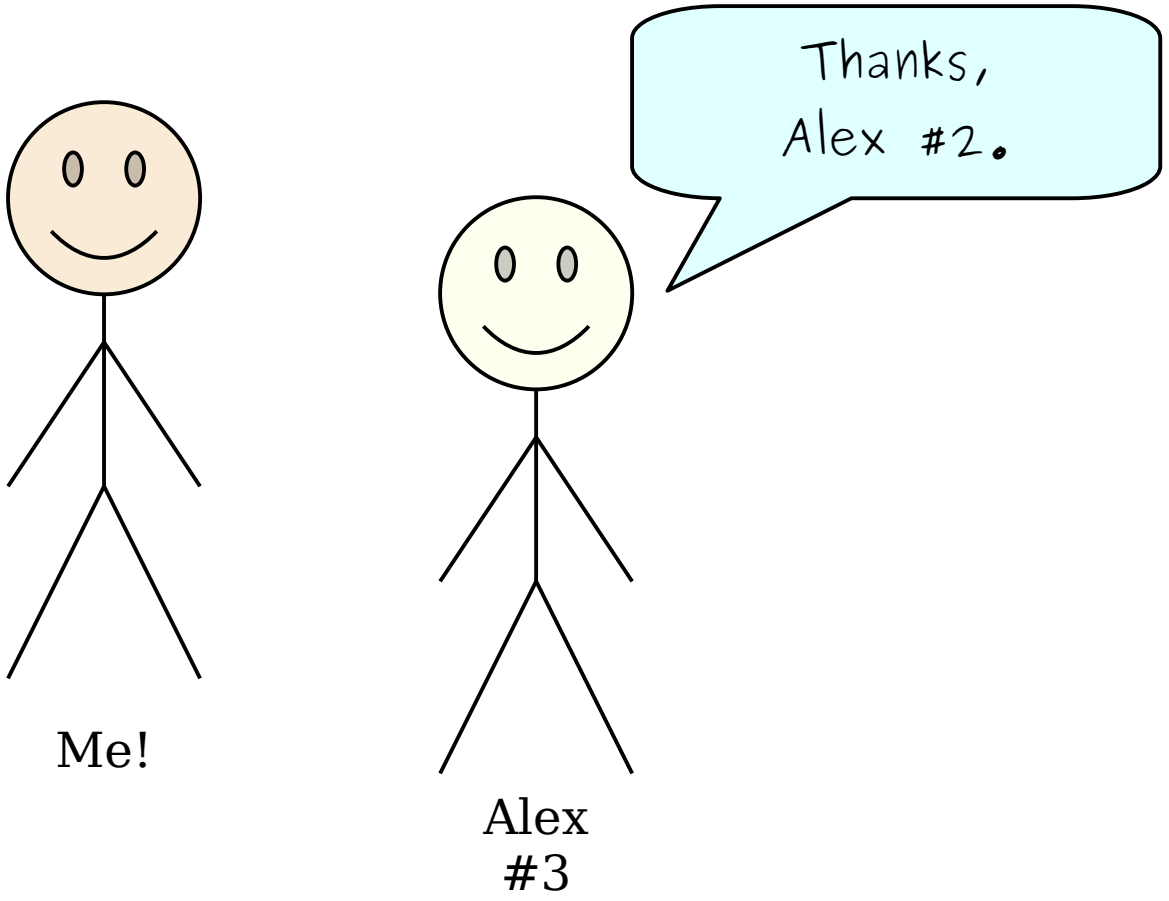
Alexes Compute Factorials



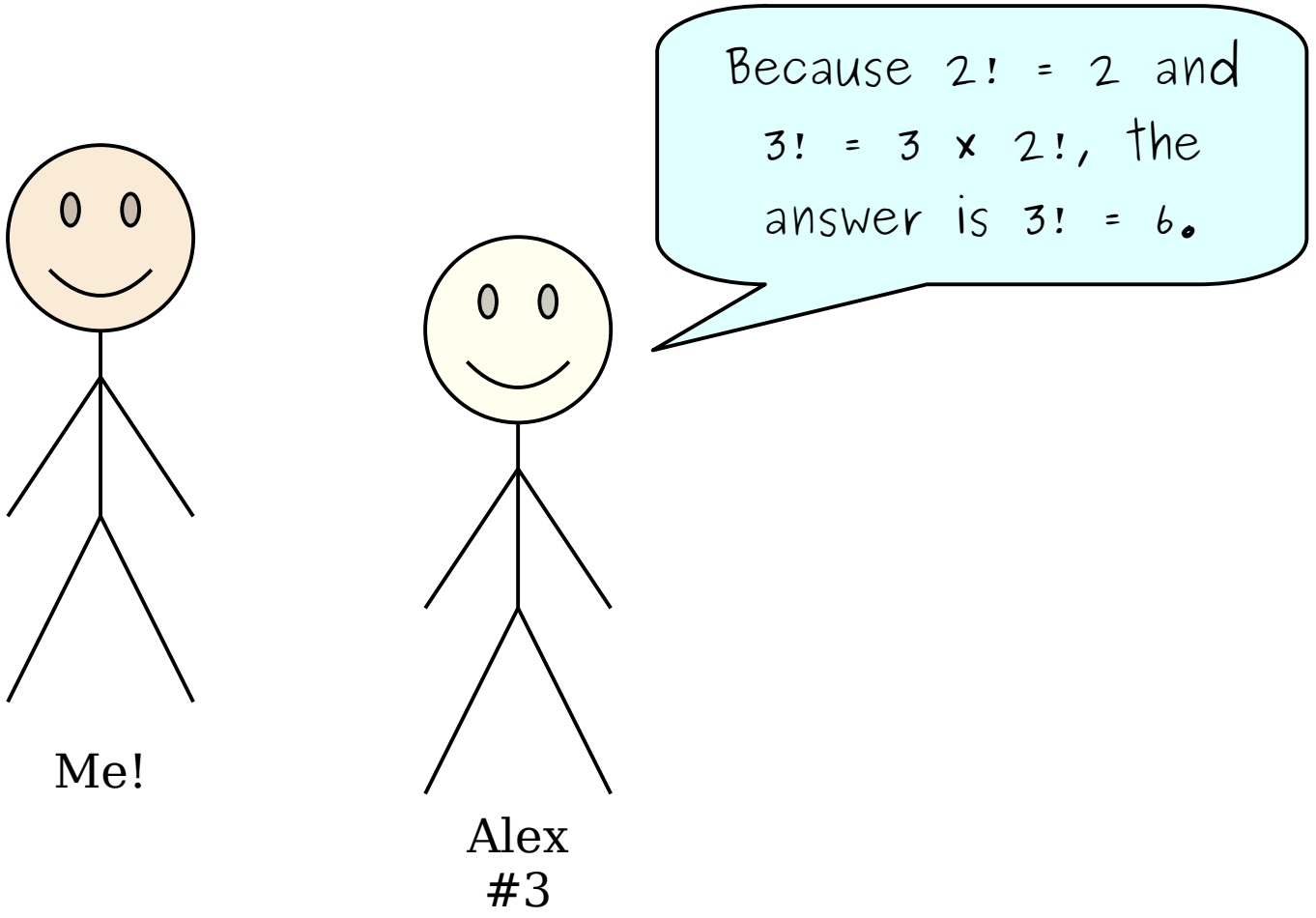
Alexes Compute Factorials



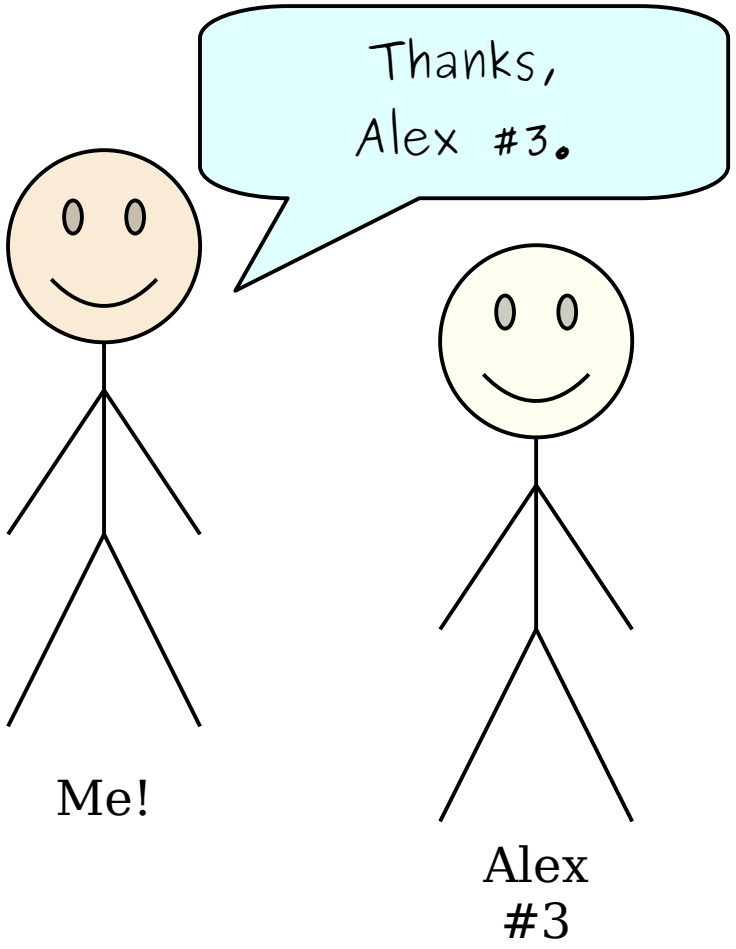
Alexes Compute Factorials



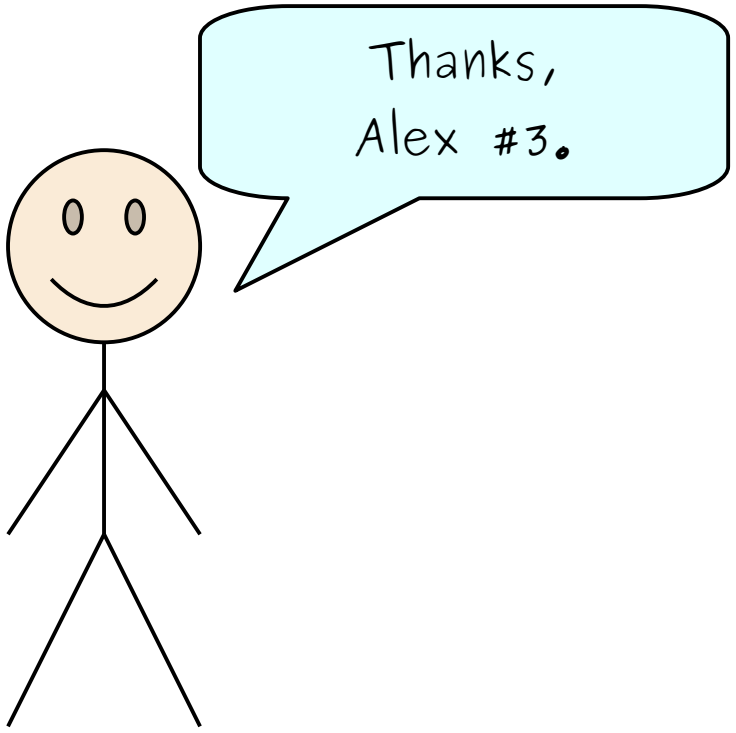
Alexes Compute Factorials



Alexes Compute Factorials

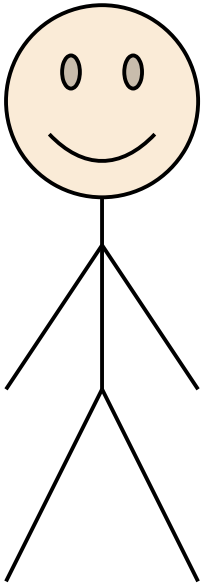


Alexes Compute Factorials



Me!

Alexes Compute Factorials



Me!

There are multiple people, each named Alex, but they're not the same person.

Each Alex is tasked with computing a different number factorial.

Each Alex gives their answer back to the previous person.

Eventually I get the answer!

Recursion in Action

```
int main() {
    int nFact = factorial(3);
    cout << "3! = " << nFact << endl;

    return 0;
}
```

Recursion in Action

```
int main() {
   int nFact = factorial(3);
   cout << "3! = " << nFact << endl;

   return 0;
}
```

Recursion in Action

```
int main() {
}
```

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

3

`int n`

Recursion in Action

```
int main() {
}
```

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

3

int n

Recursion in Action

```
int main() {
}
```

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

3

int n

Recursion in Action

```
int main() {
}
```

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

3

int n

Recursion in Action

```
int main() {
}
```

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

3

int n

Recursion in Action

```
int main() {
}
```

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

3

int n

3

Recursion in Action

```
int main() {
}
```

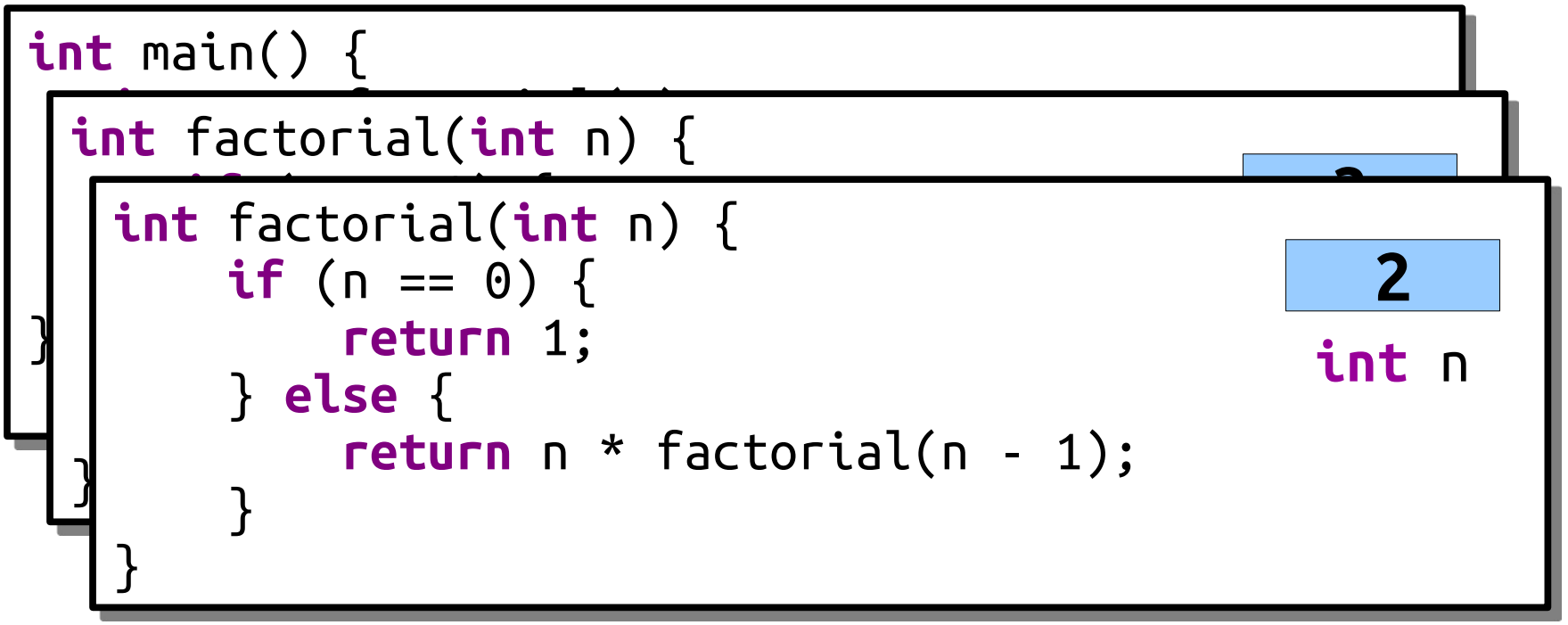
```
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

3

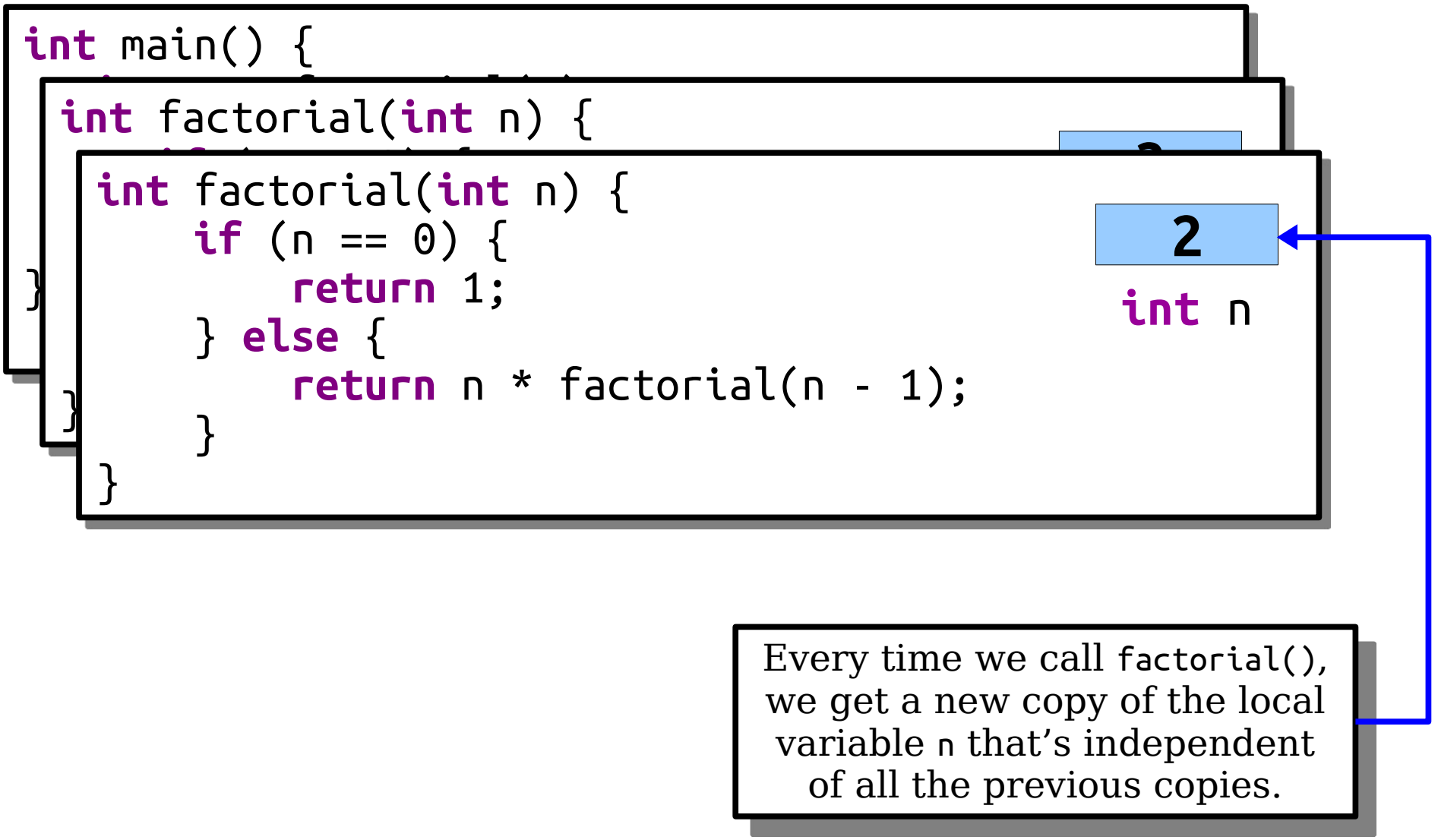
int n

3

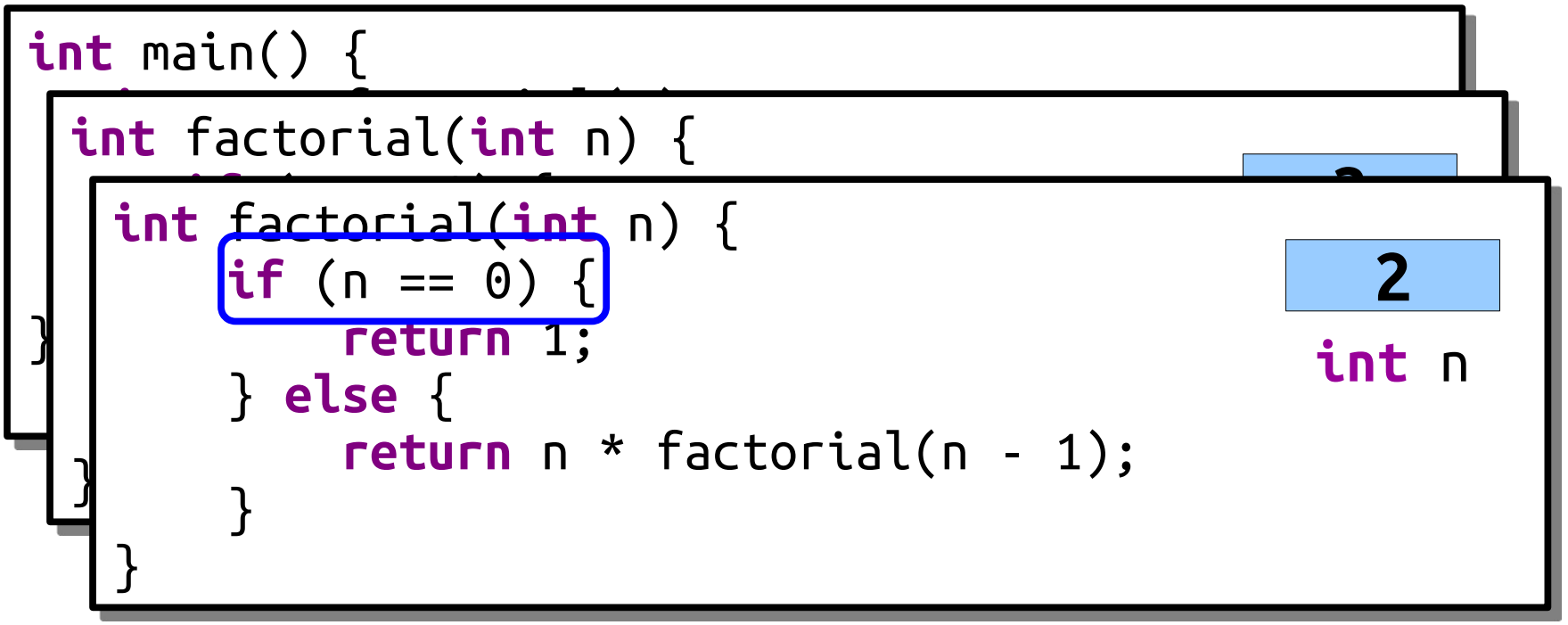
Recursion in Action



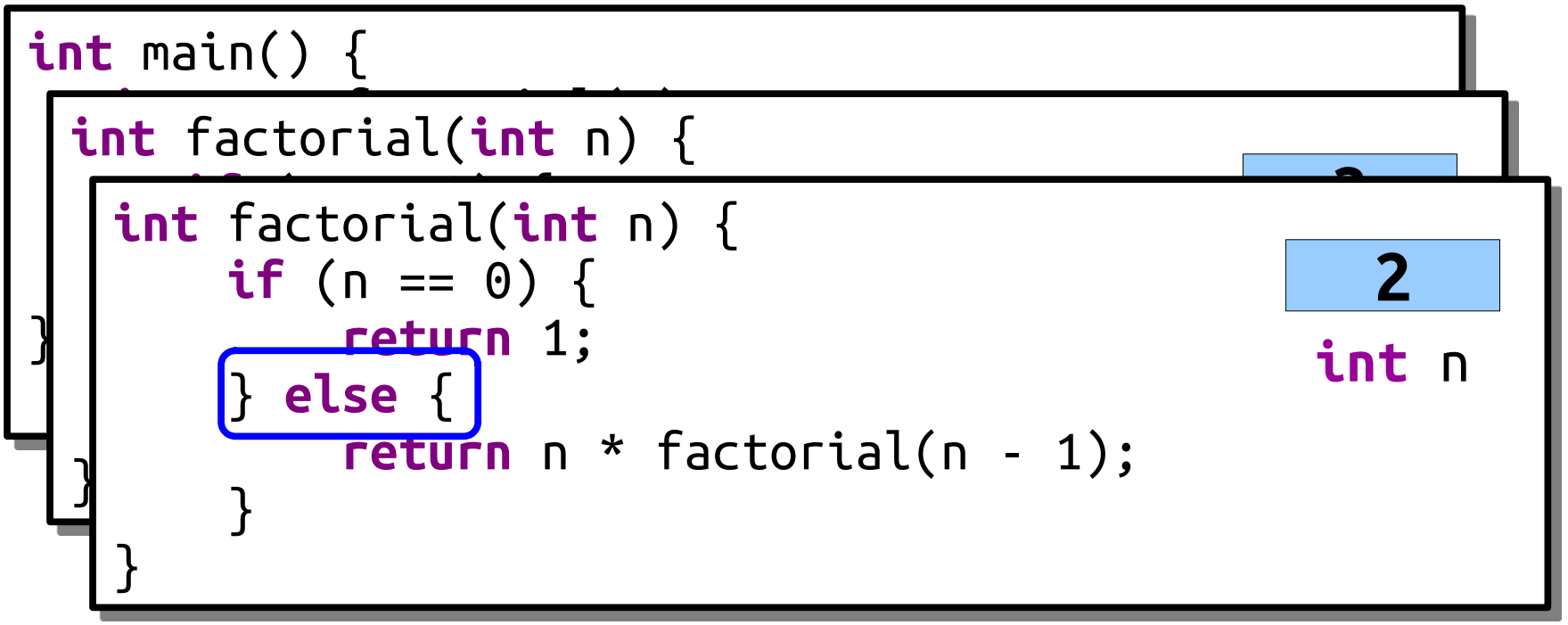
Recursion in Action



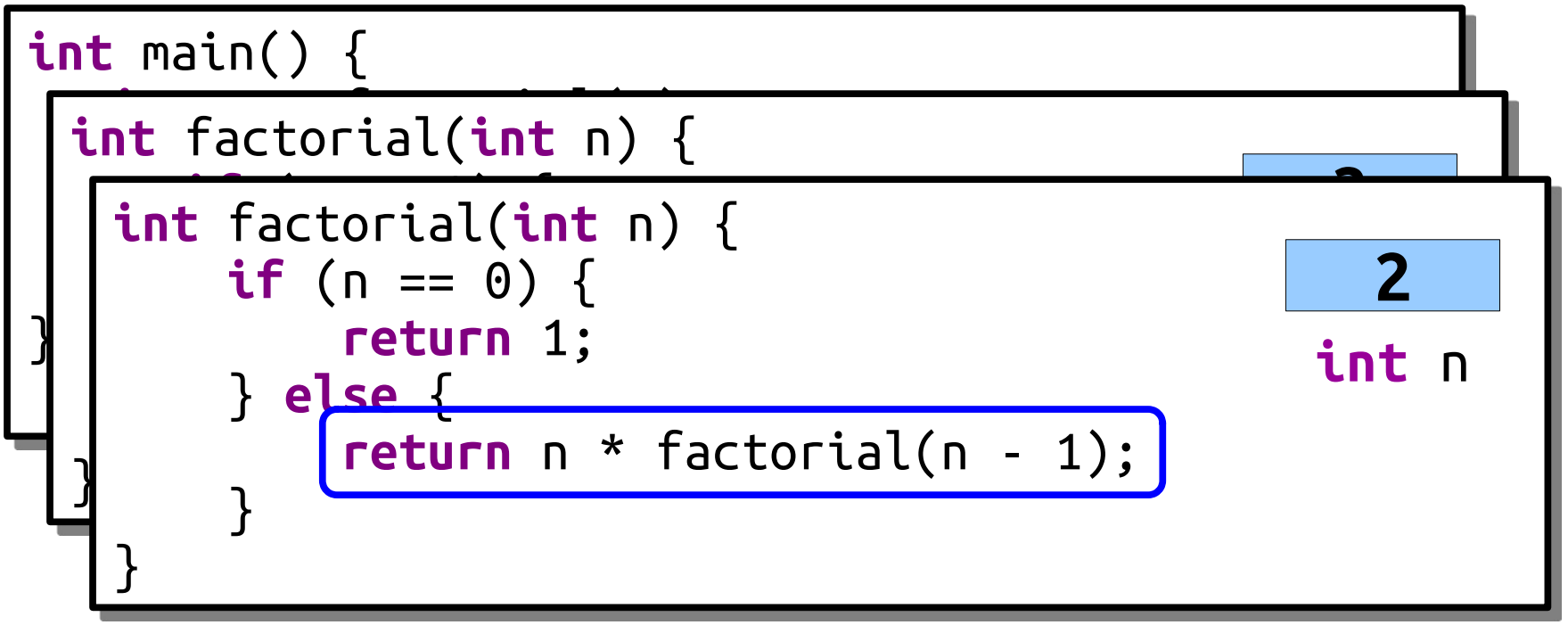
Recursion in Action



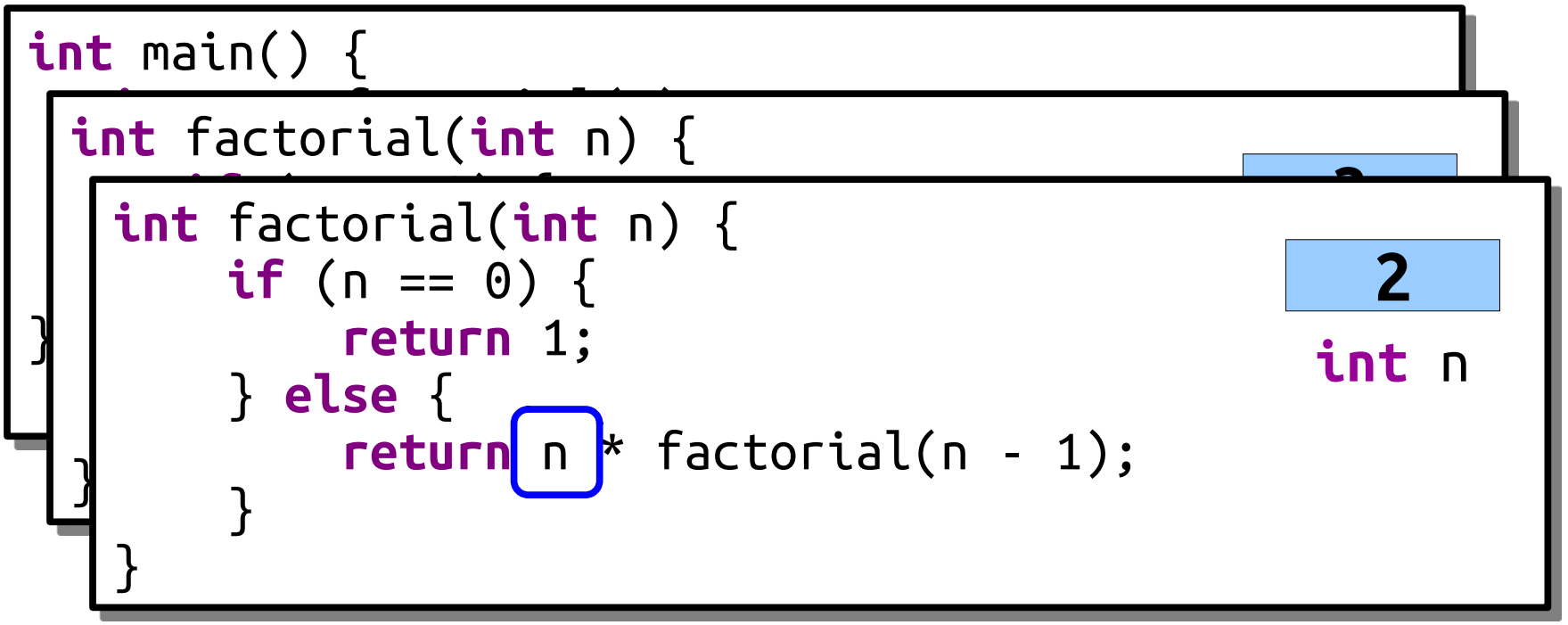
Recursion in Action



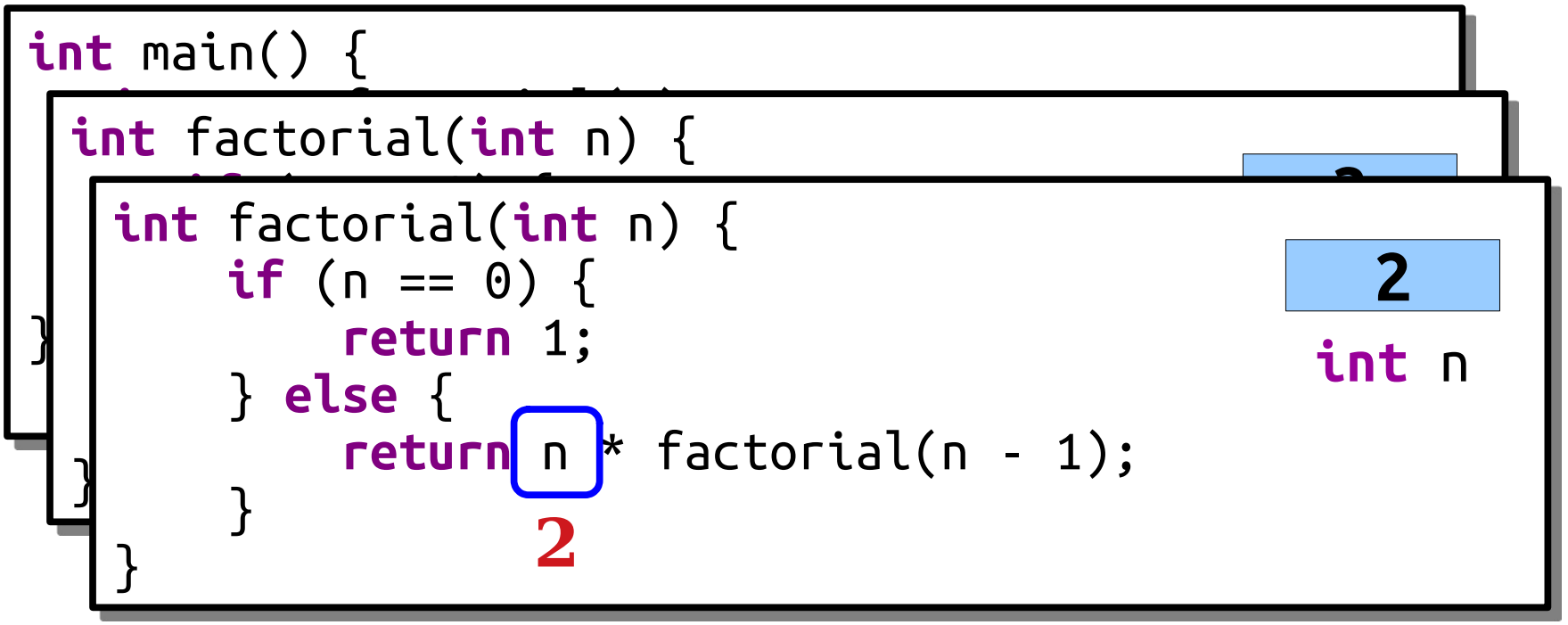
Recursion in Action



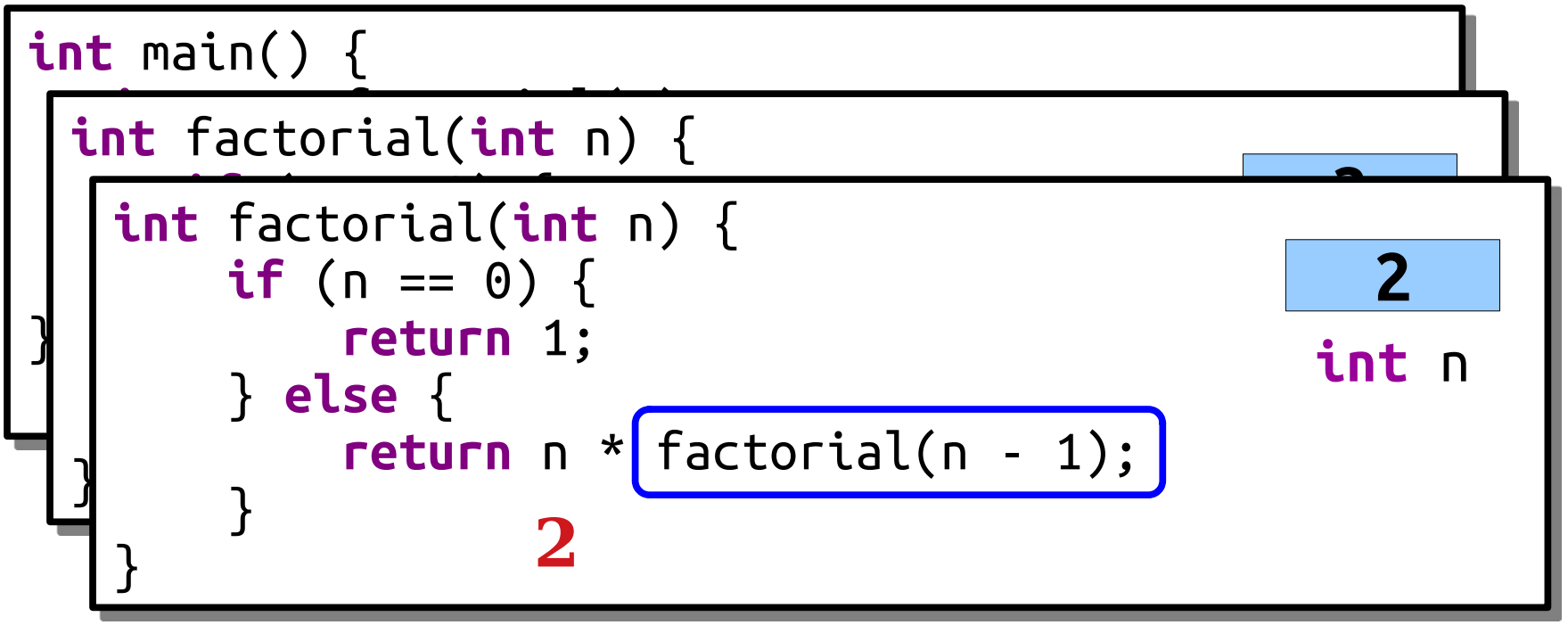
Recursion in Action



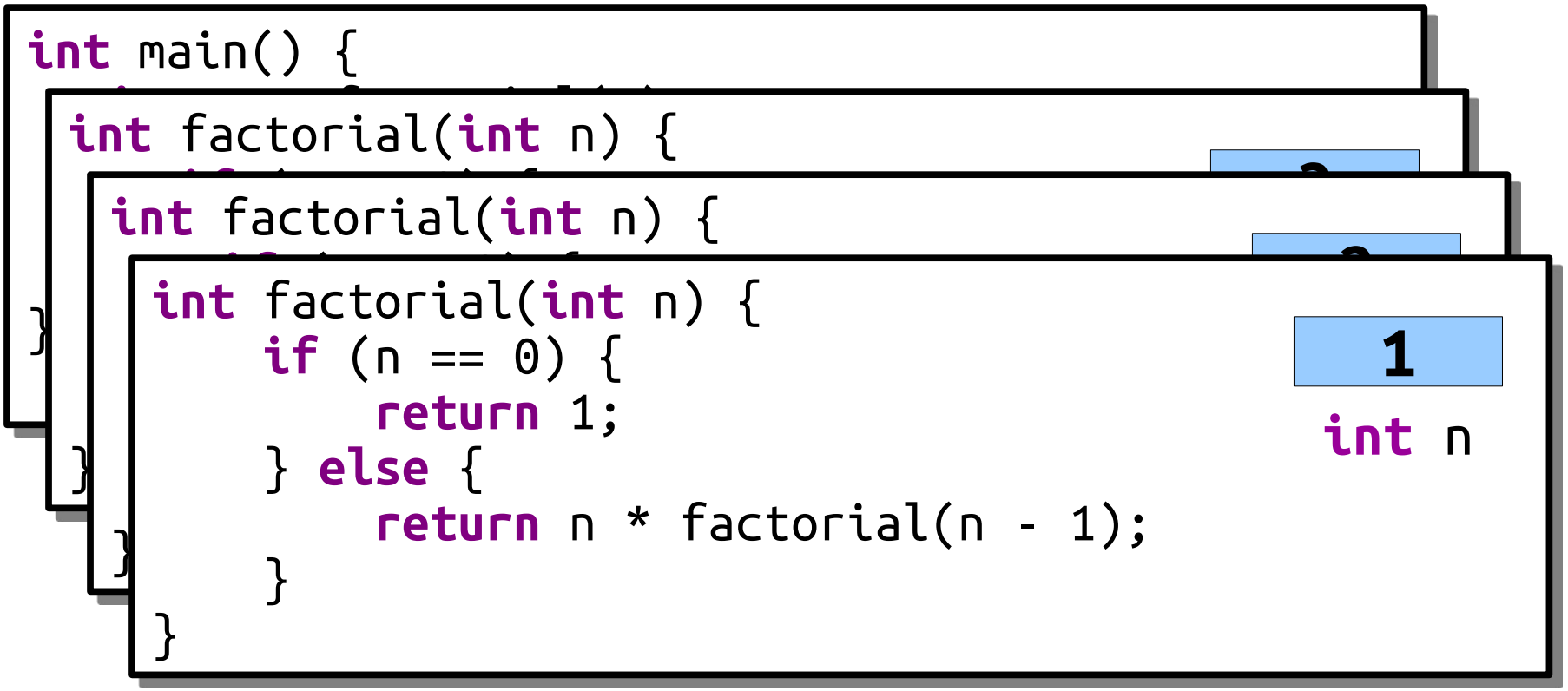
Recursion in Action



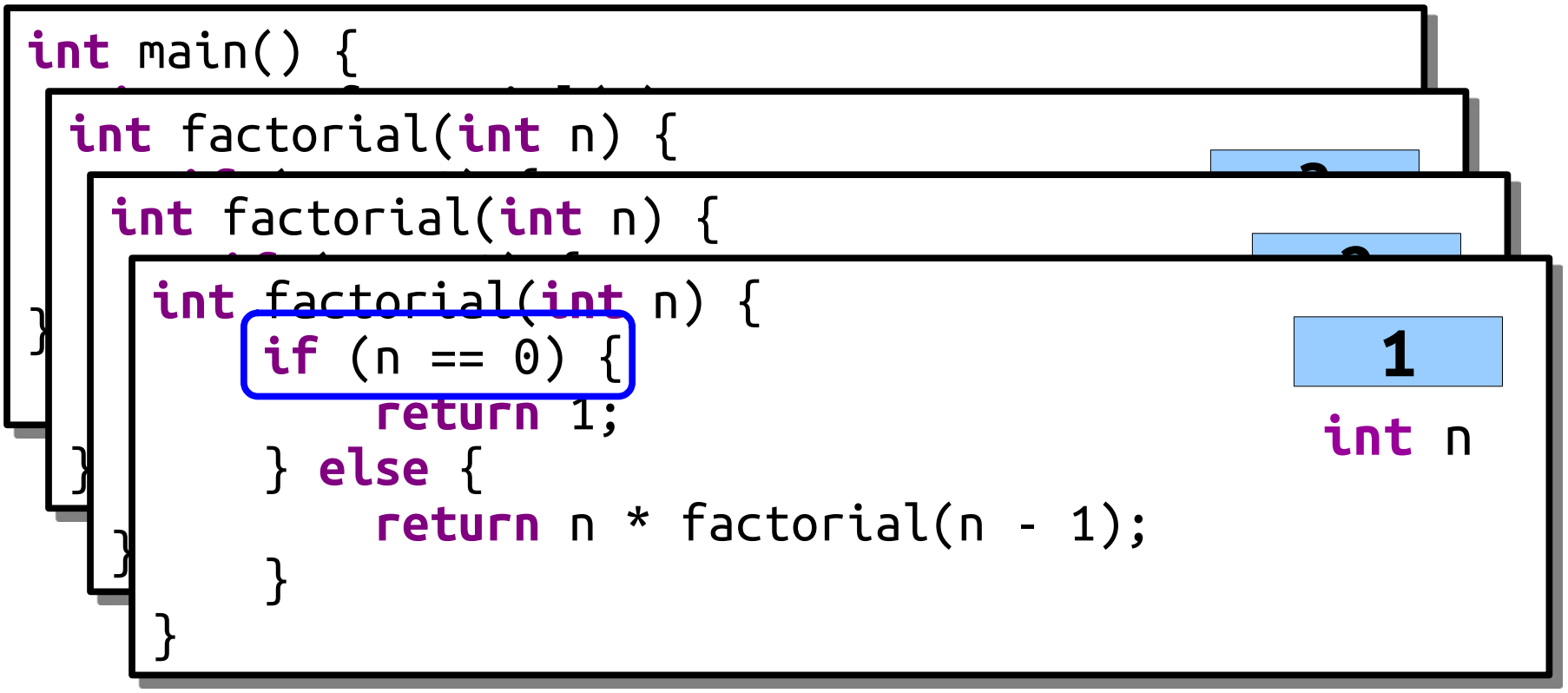
Recursion in Action



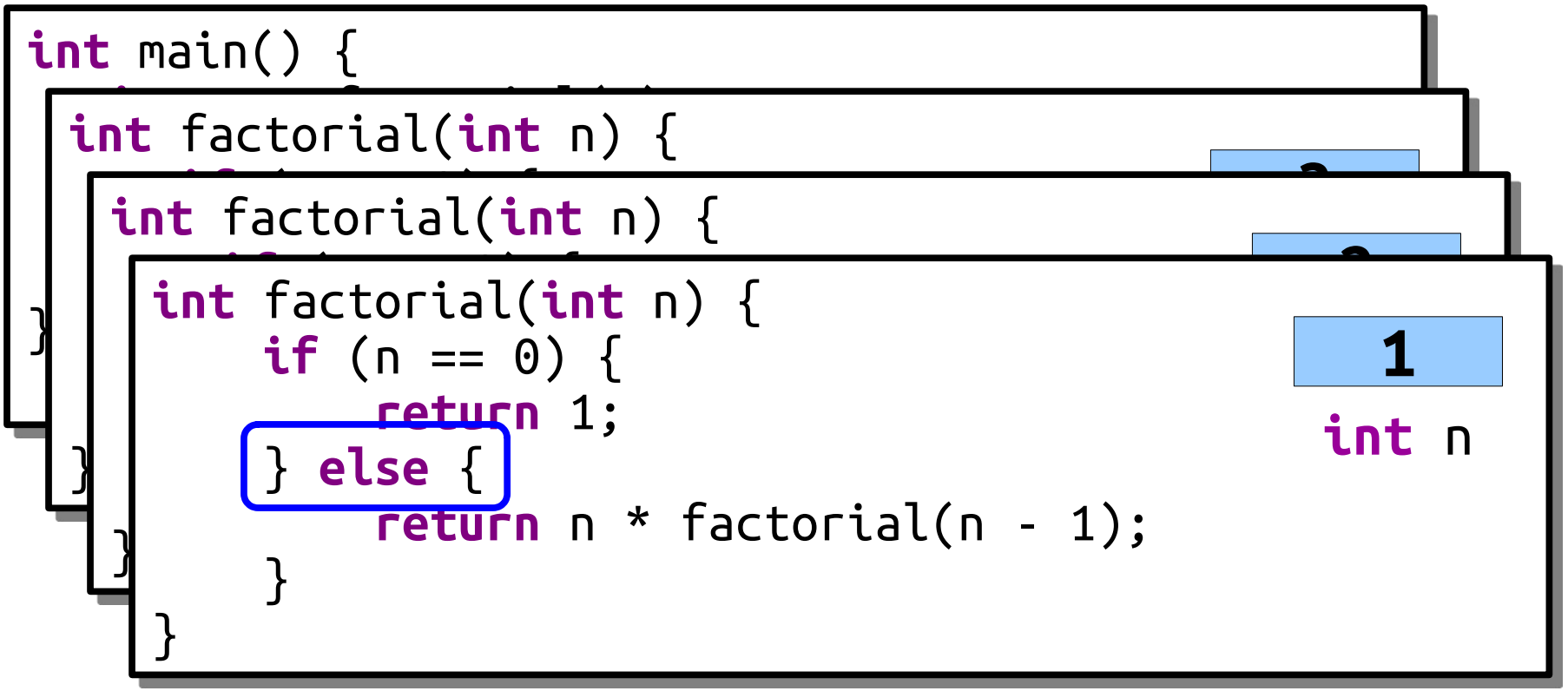
Recursion in Action



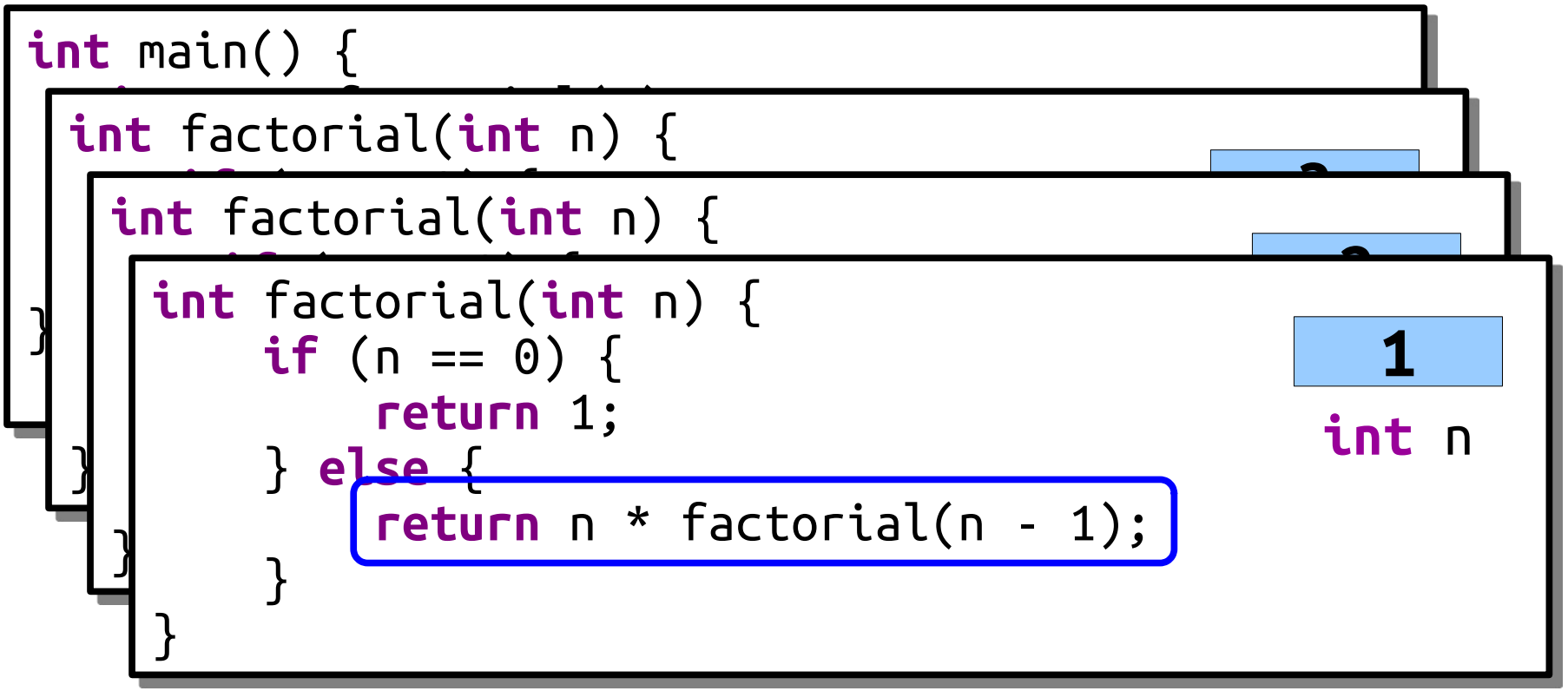
Recursion in Action



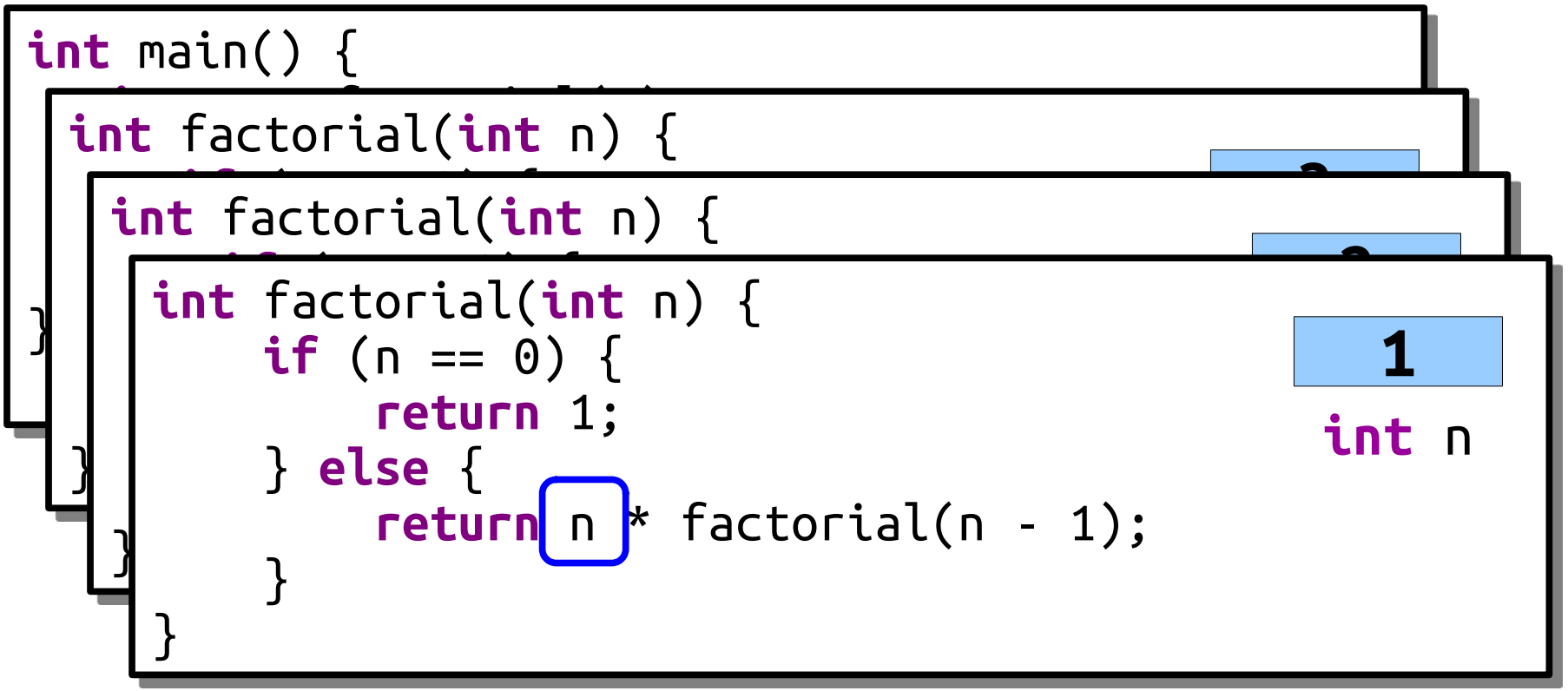
Recursion in Action



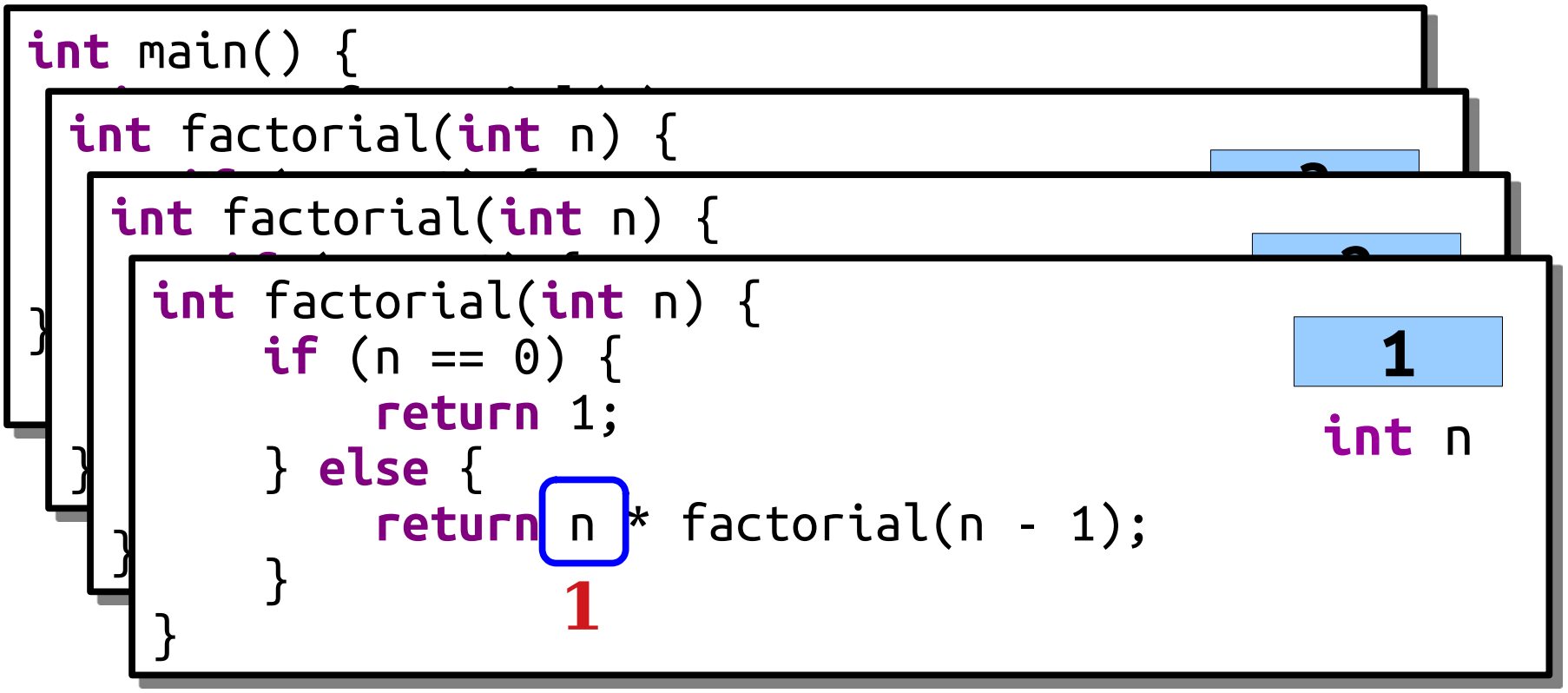
Recursion in Action



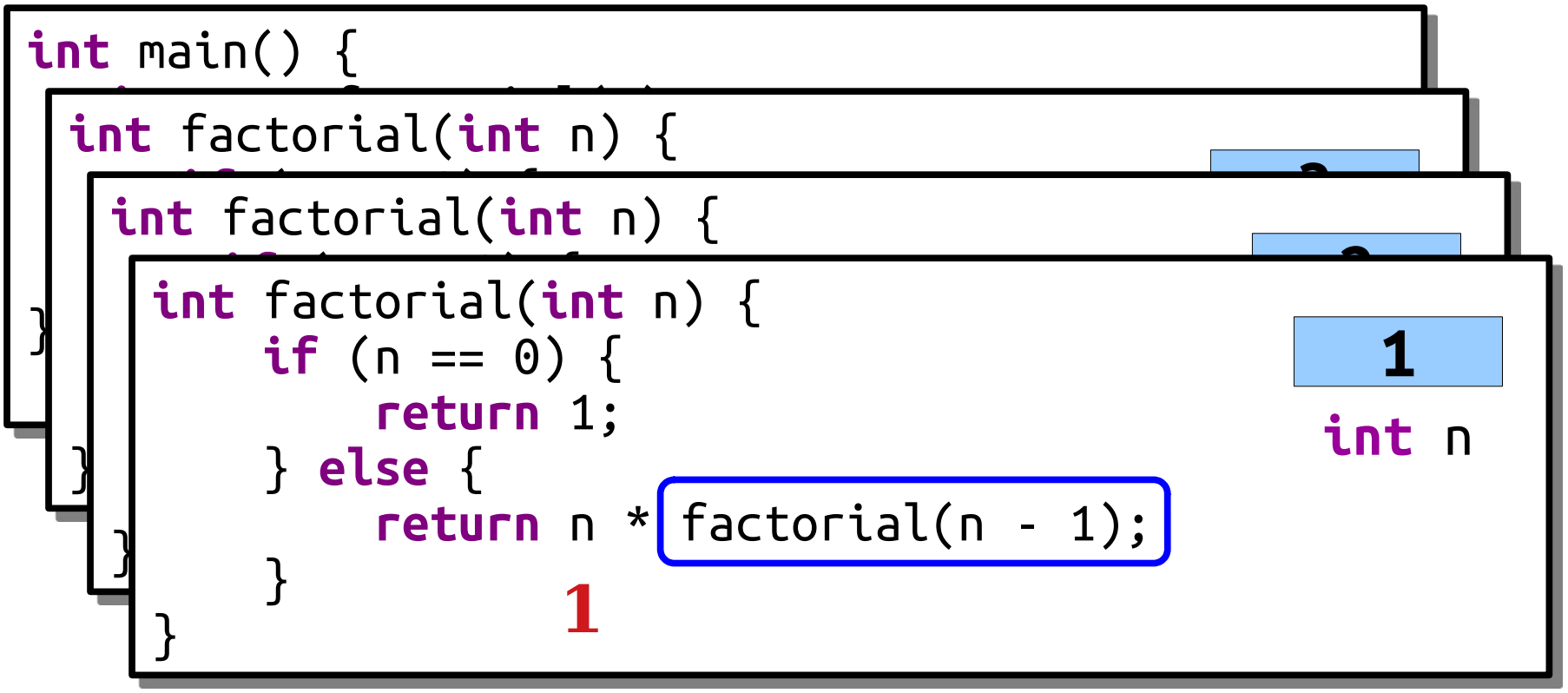
Recursion in Action



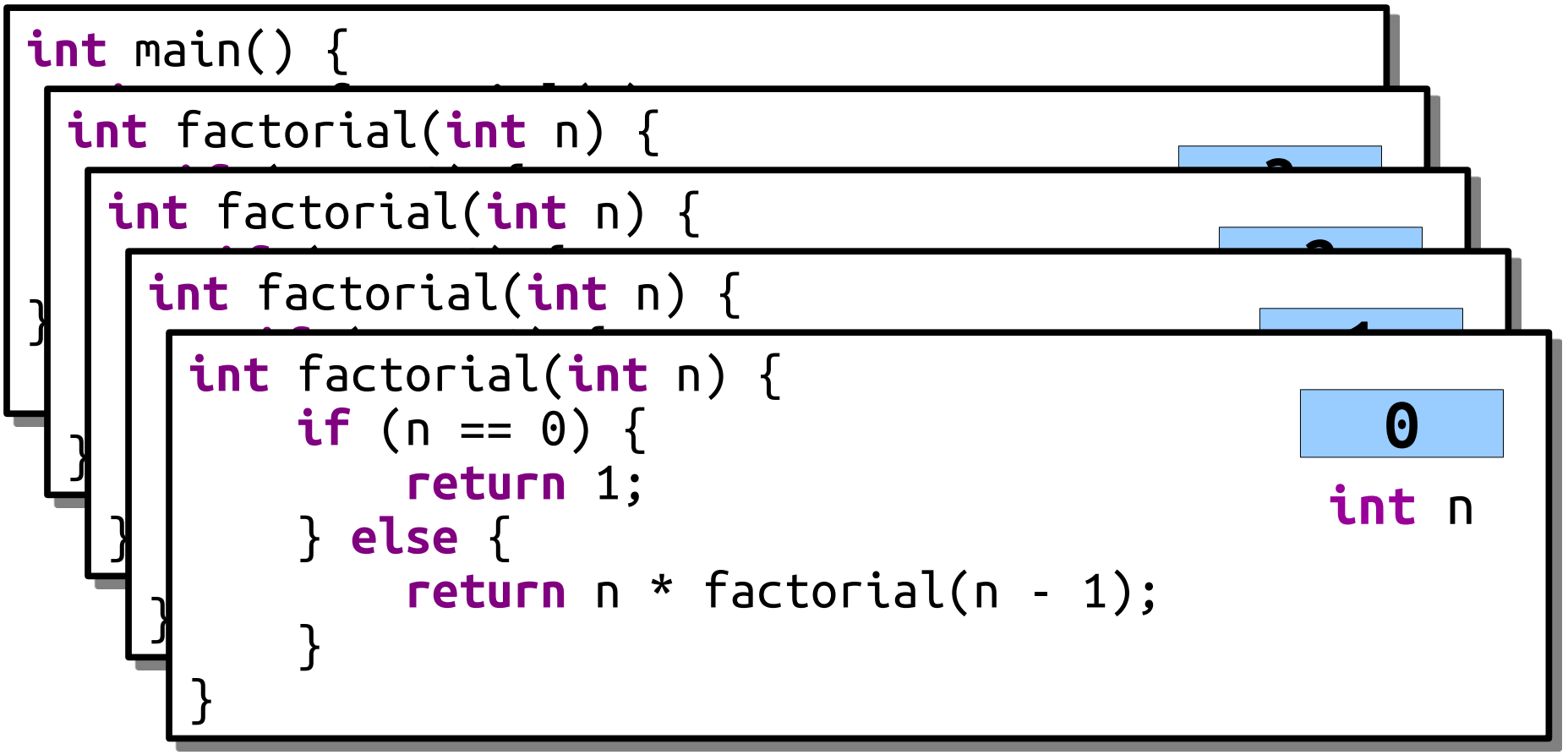
Recursion in Action



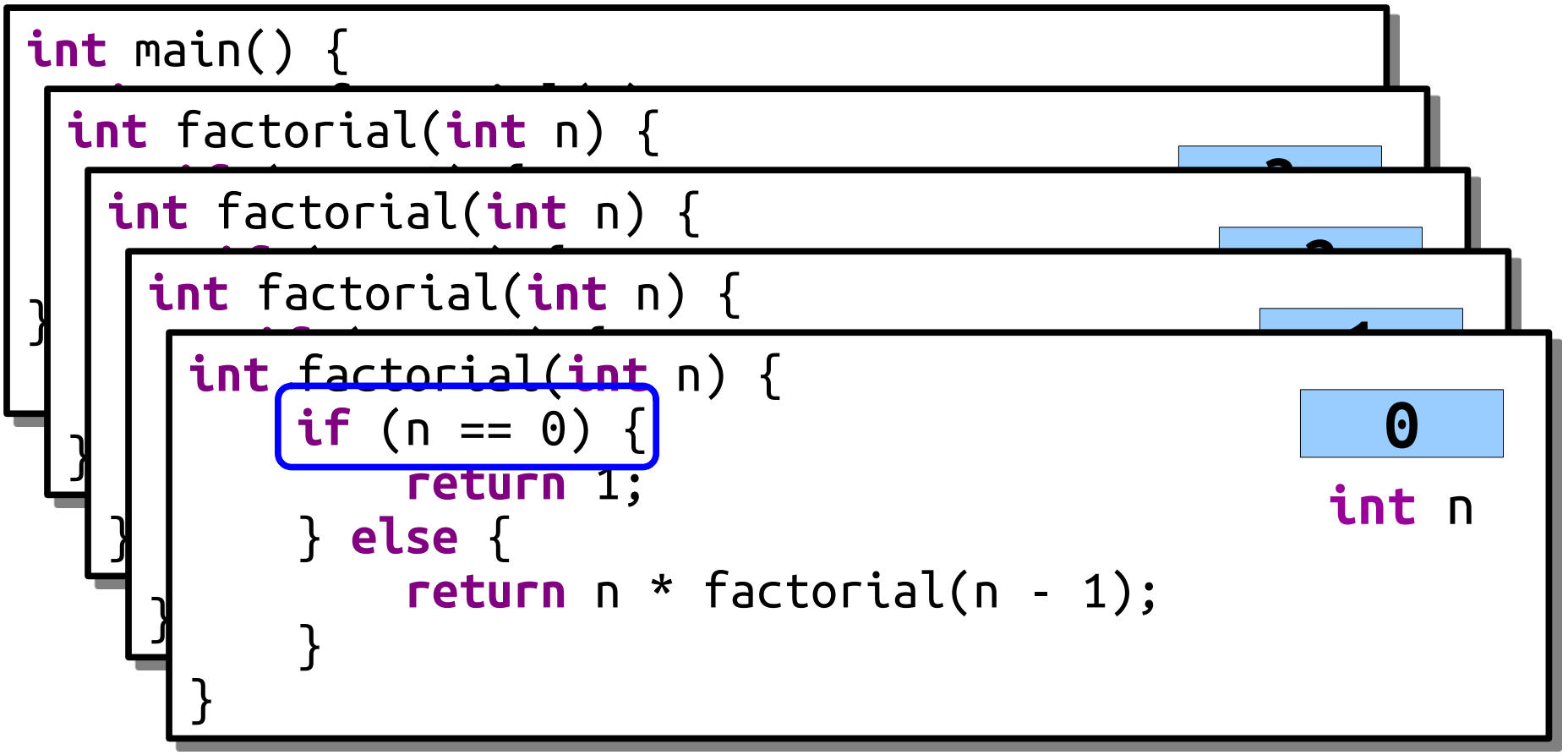
Recursion in Action



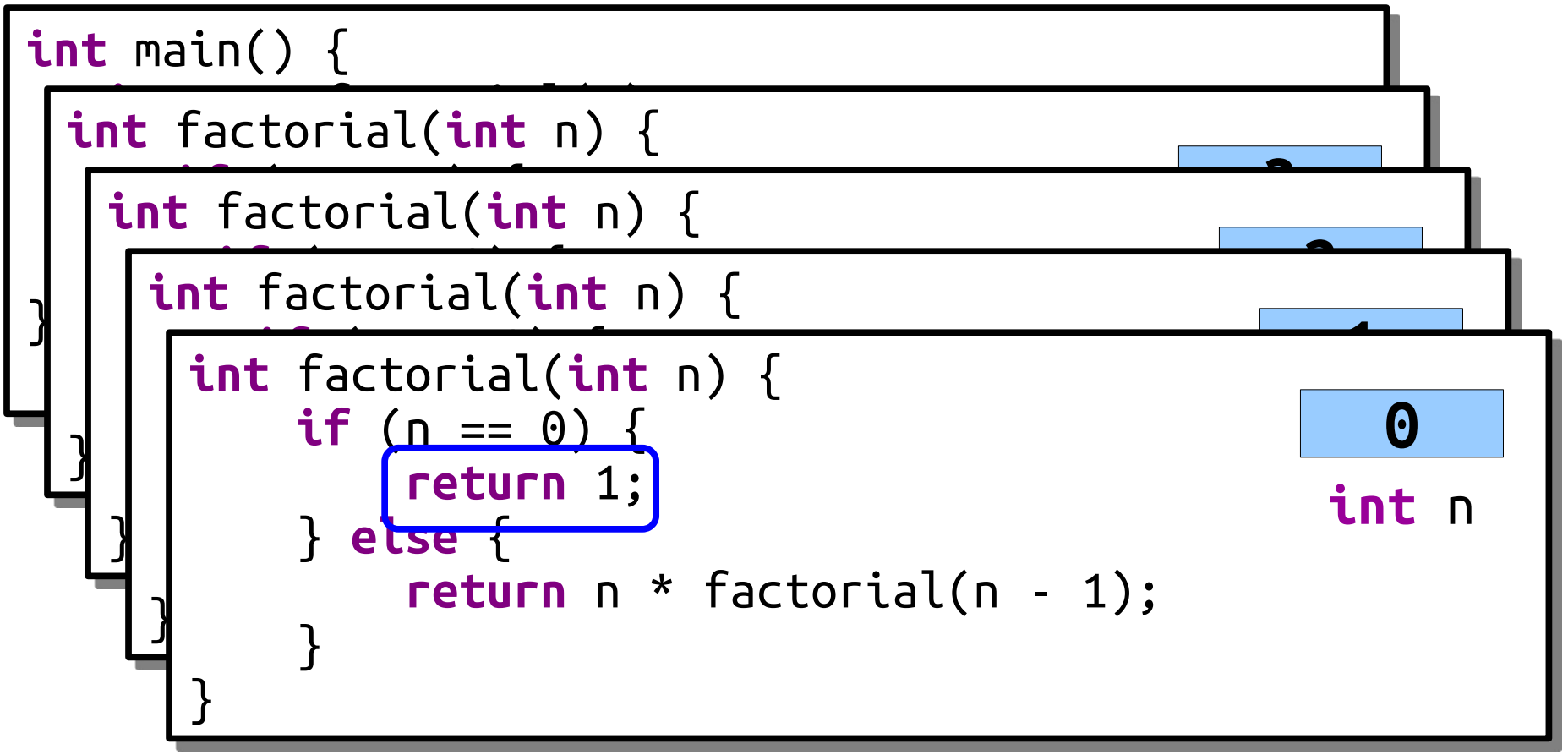
Recursion in Action



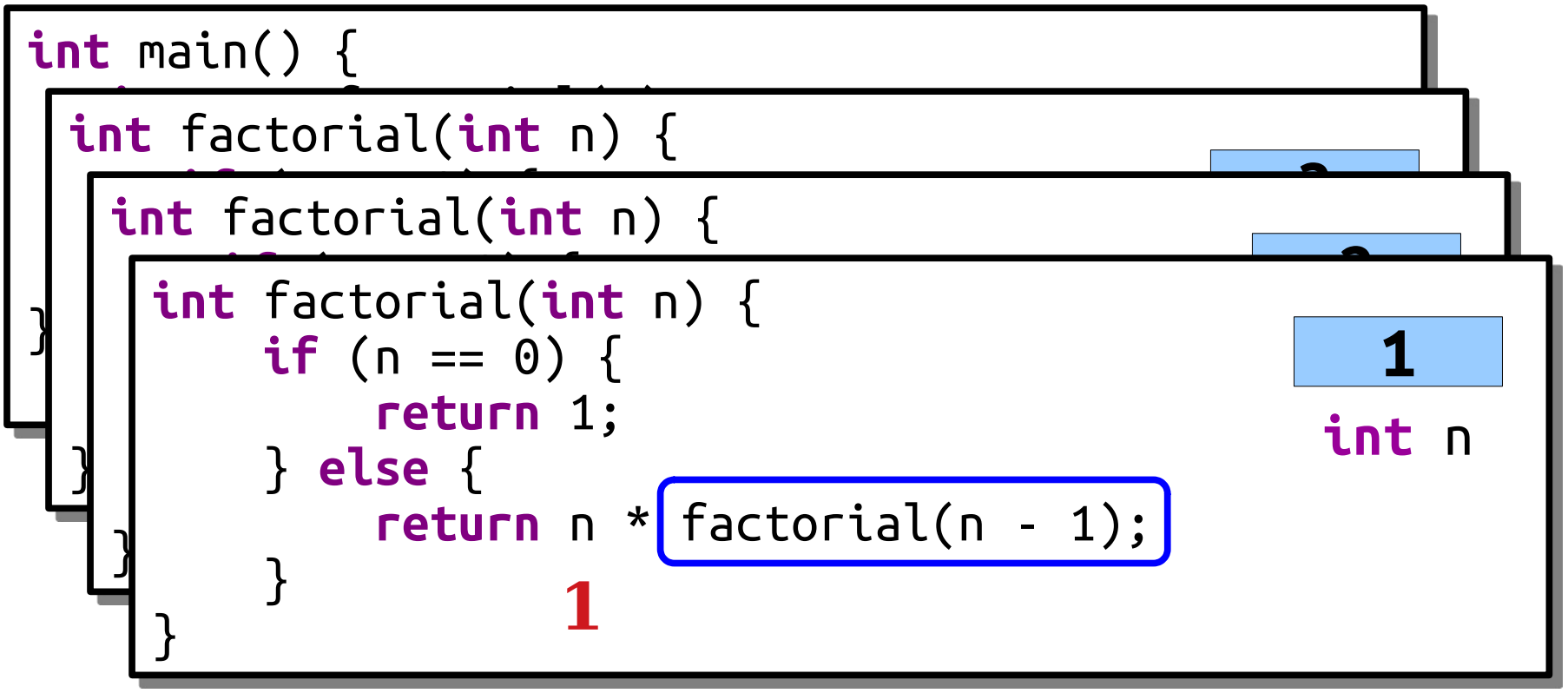
Recursion in Action



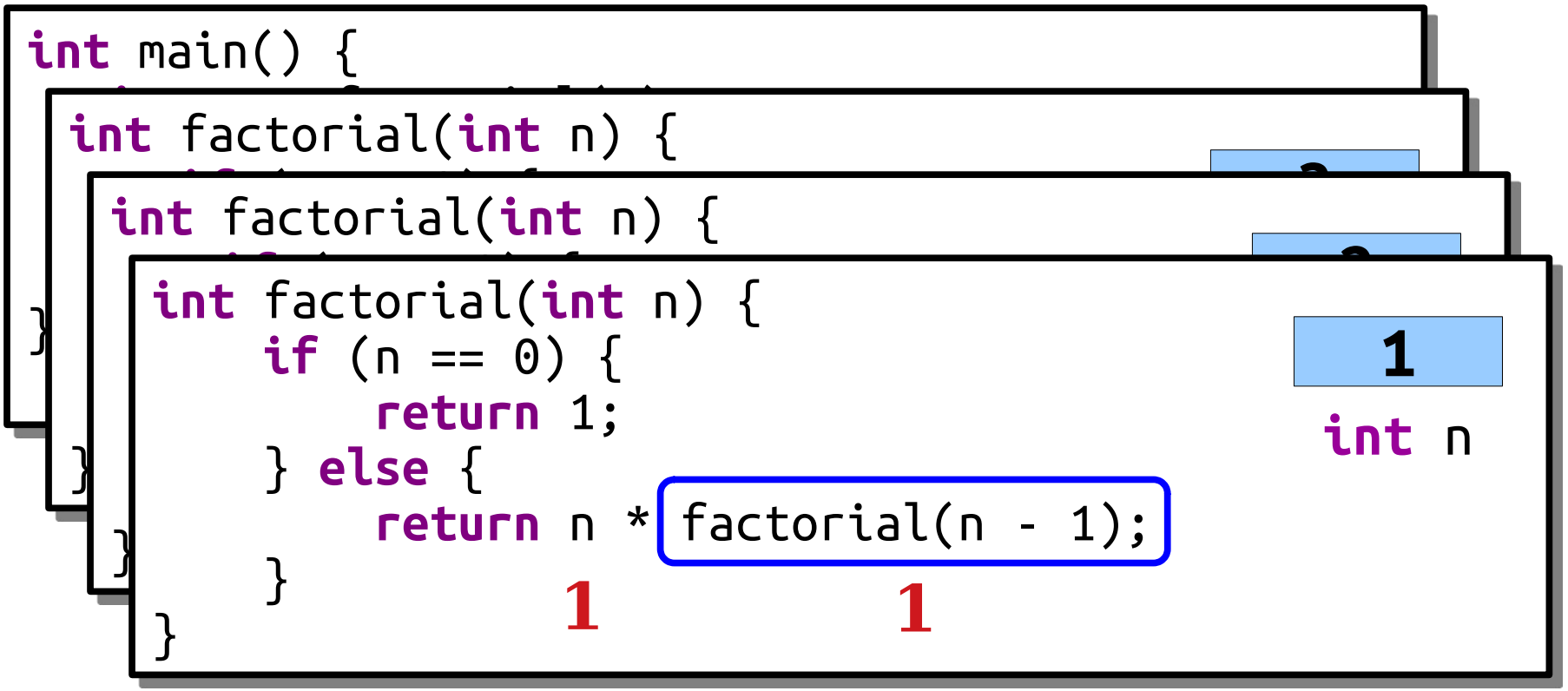
Recursion in Action



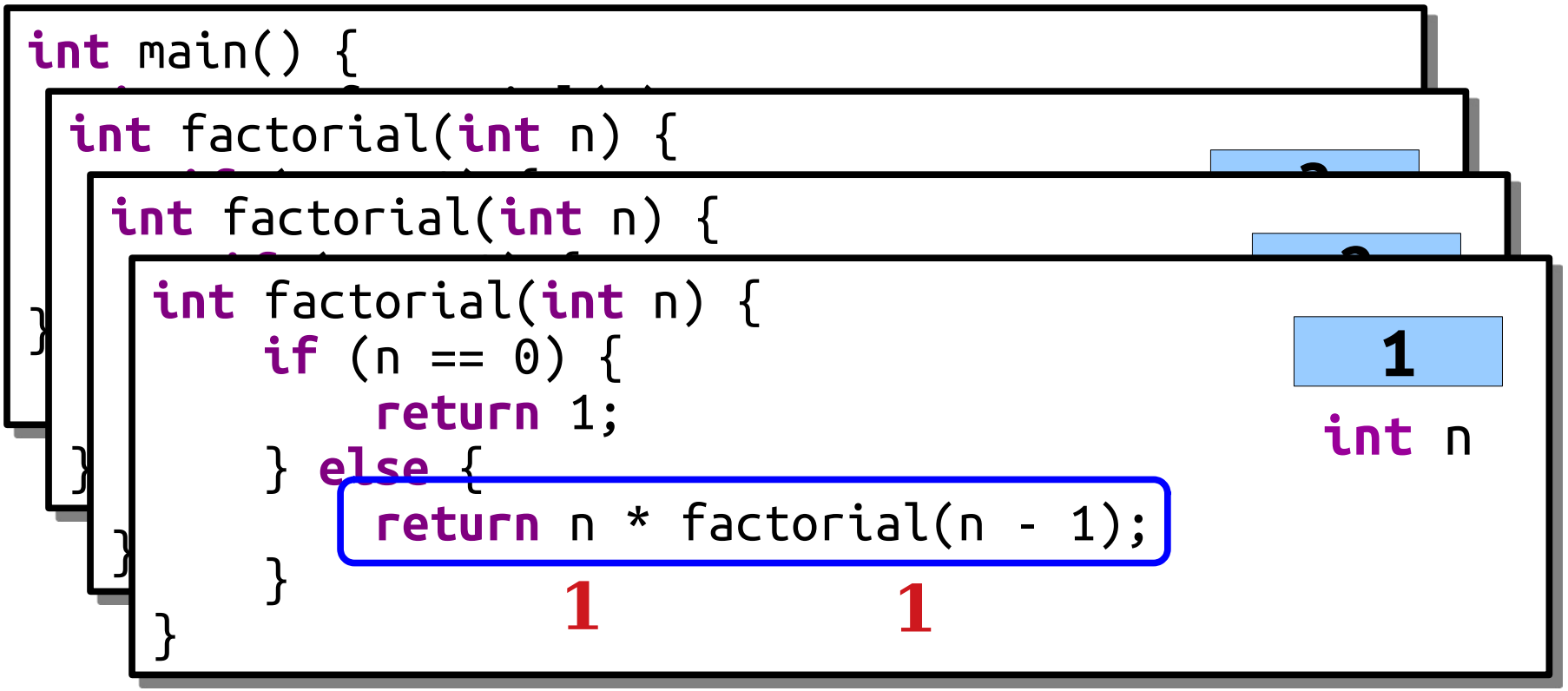
Recursion in Action



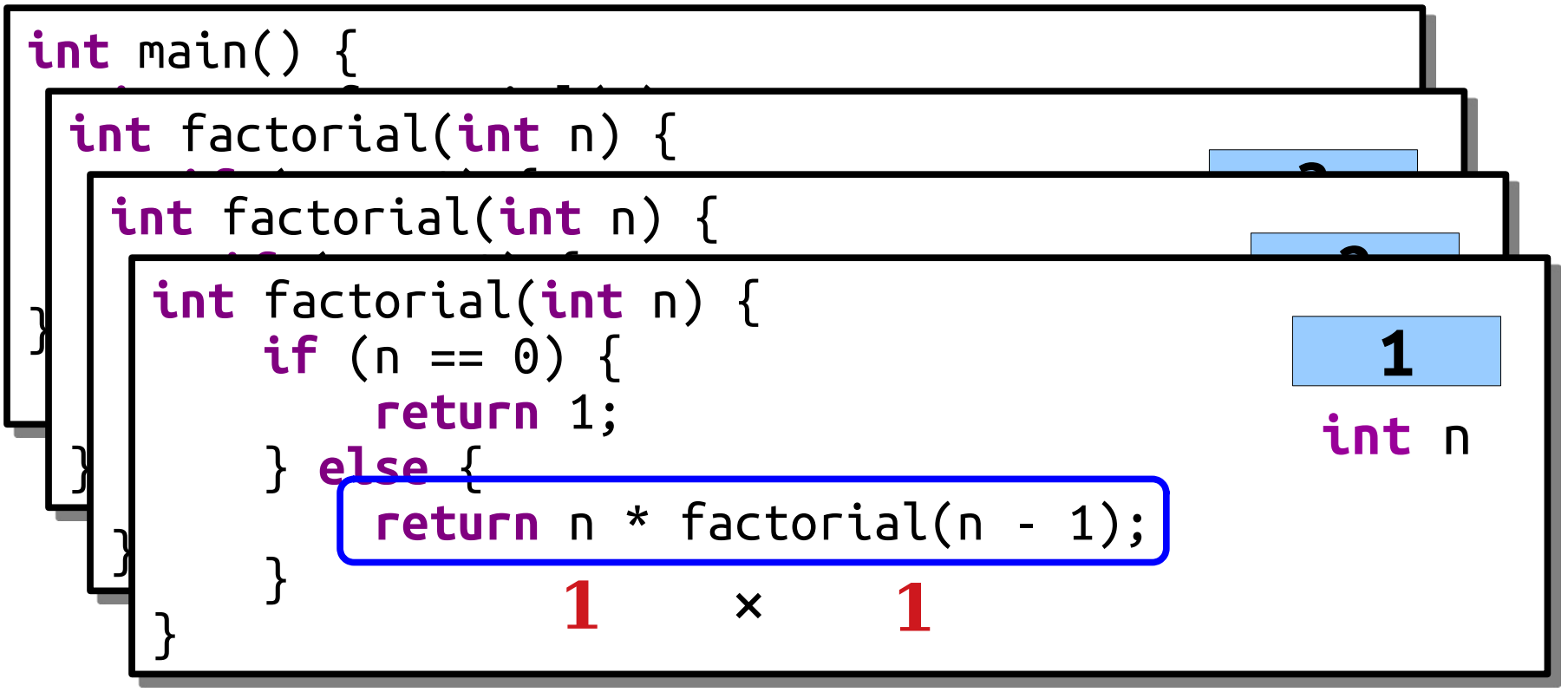
Recursion in Action



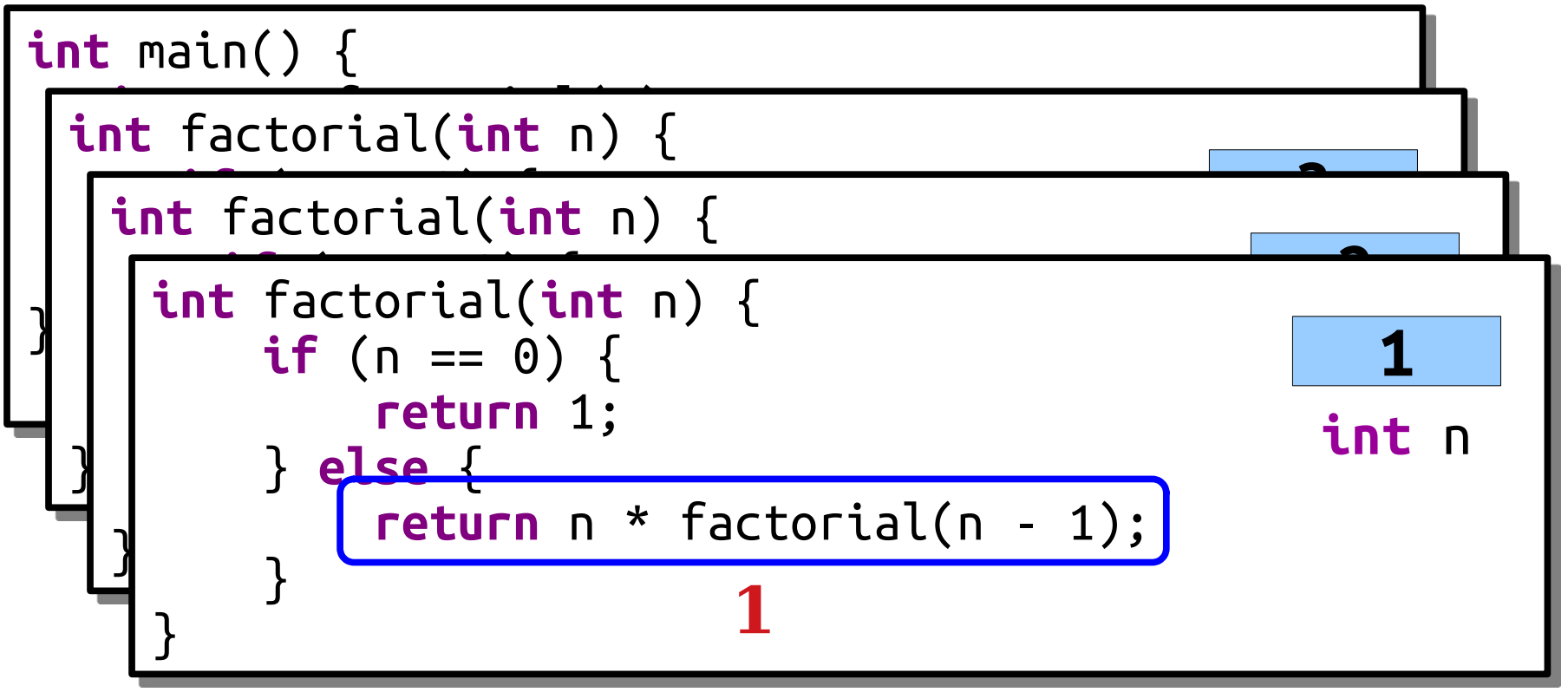
Recursion in Action



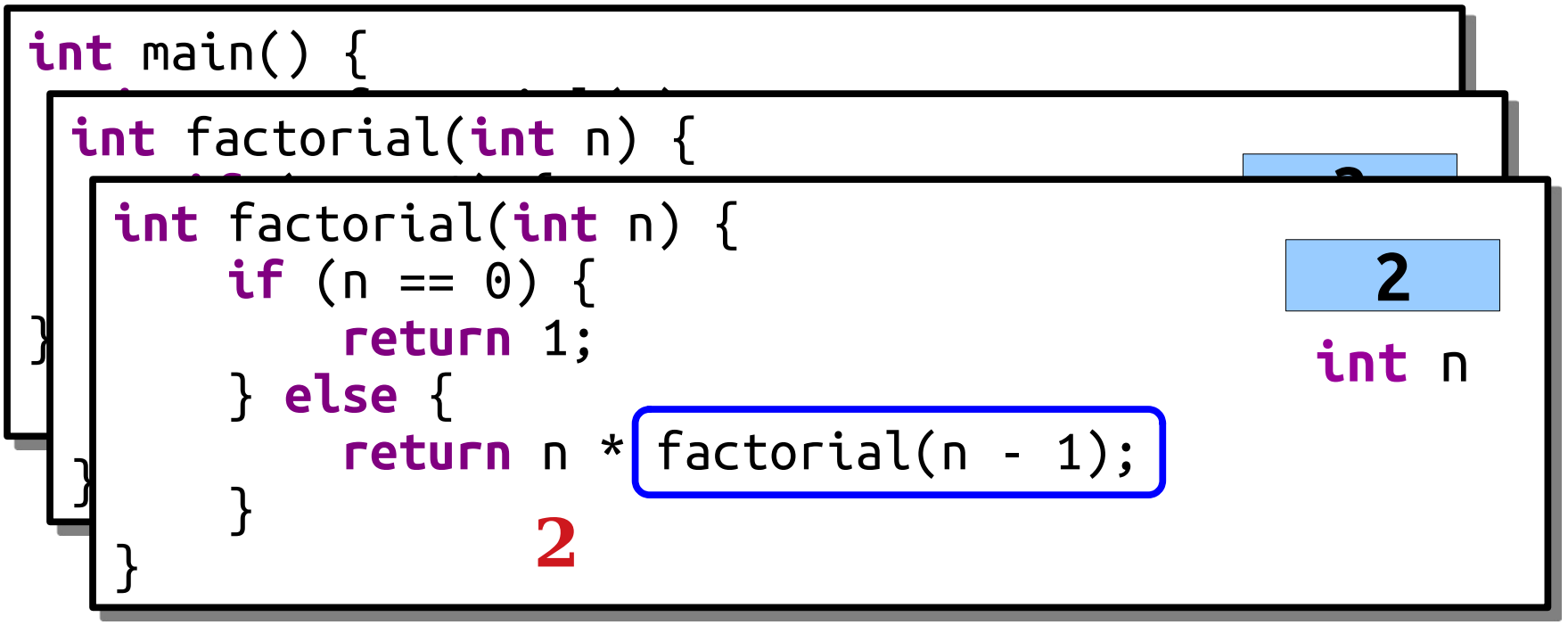
Recursion in Action



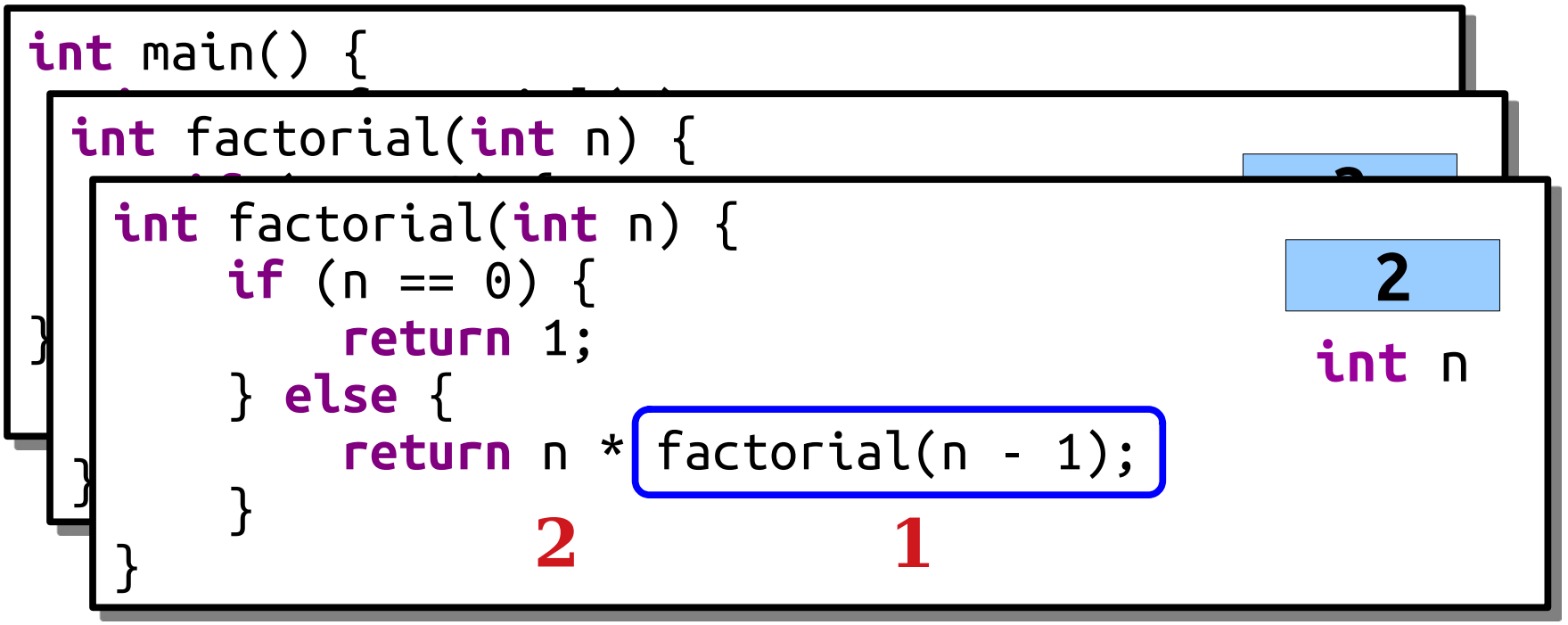
Recursion in Action



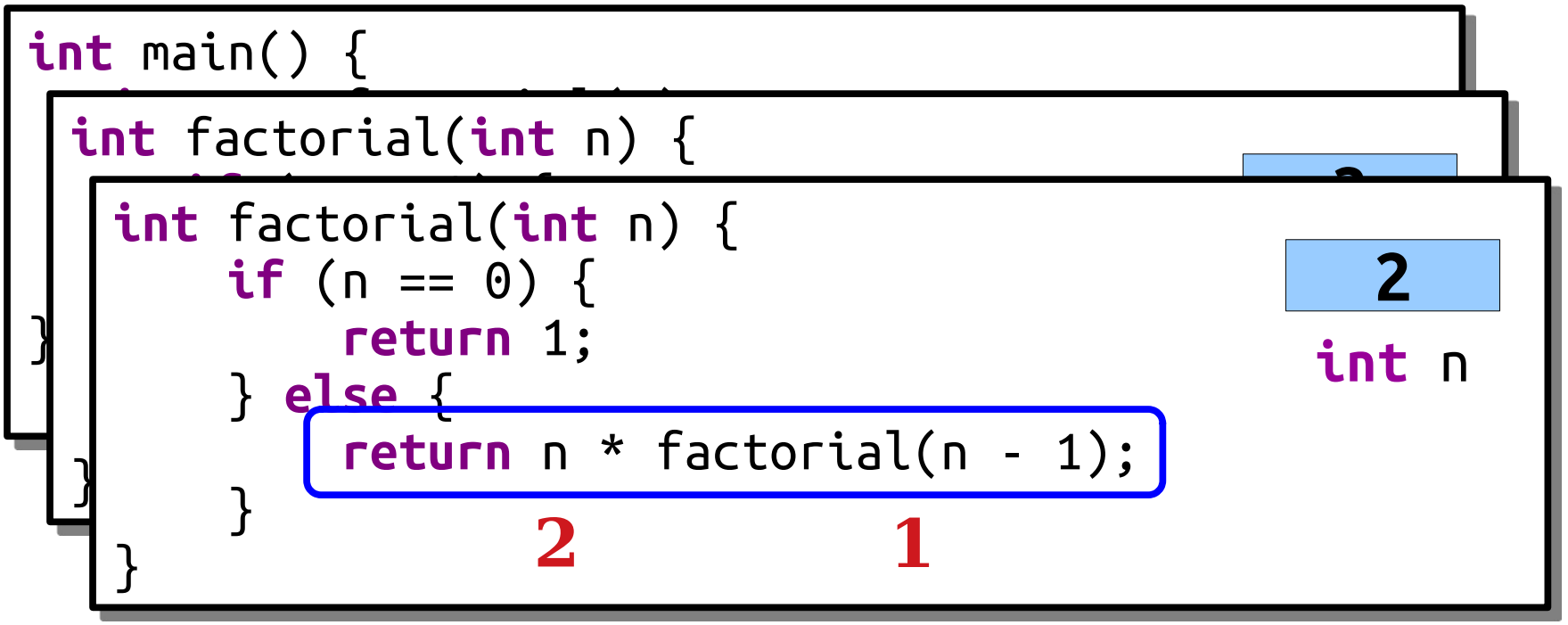
Recursion in Action



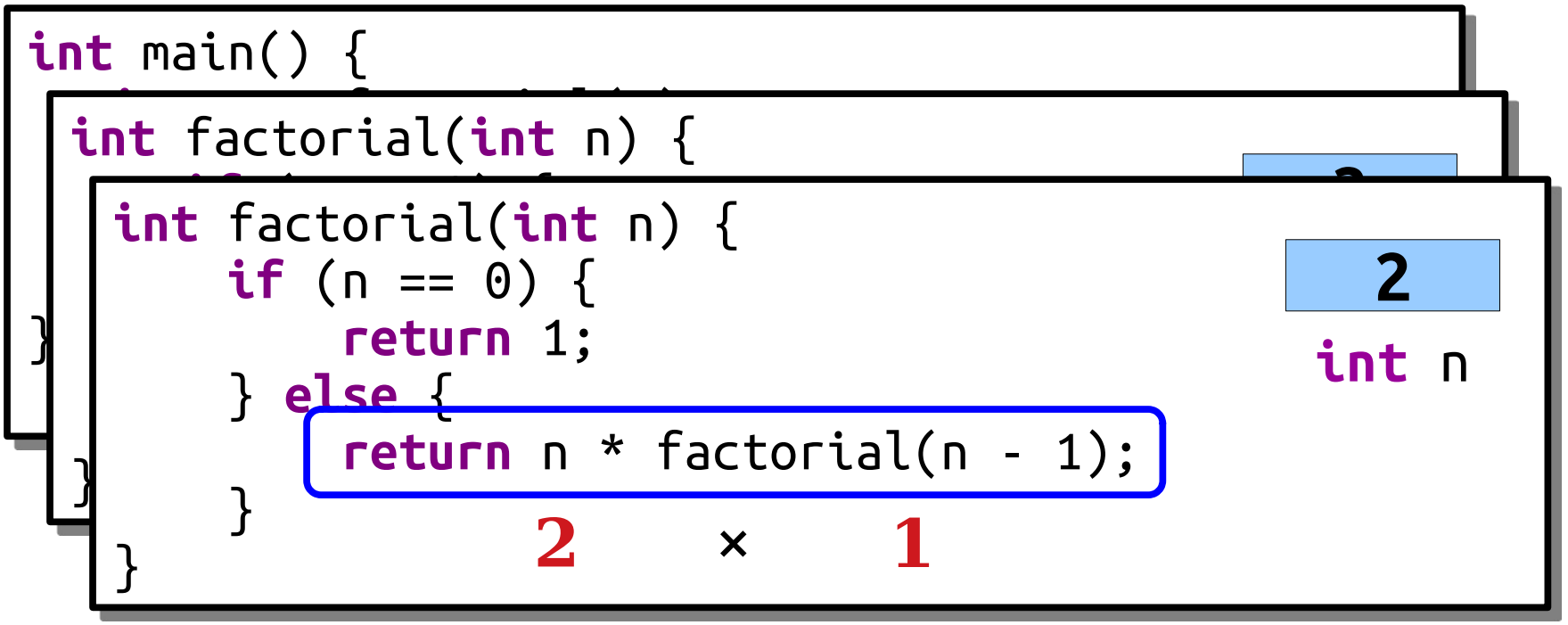
Recursion in Action



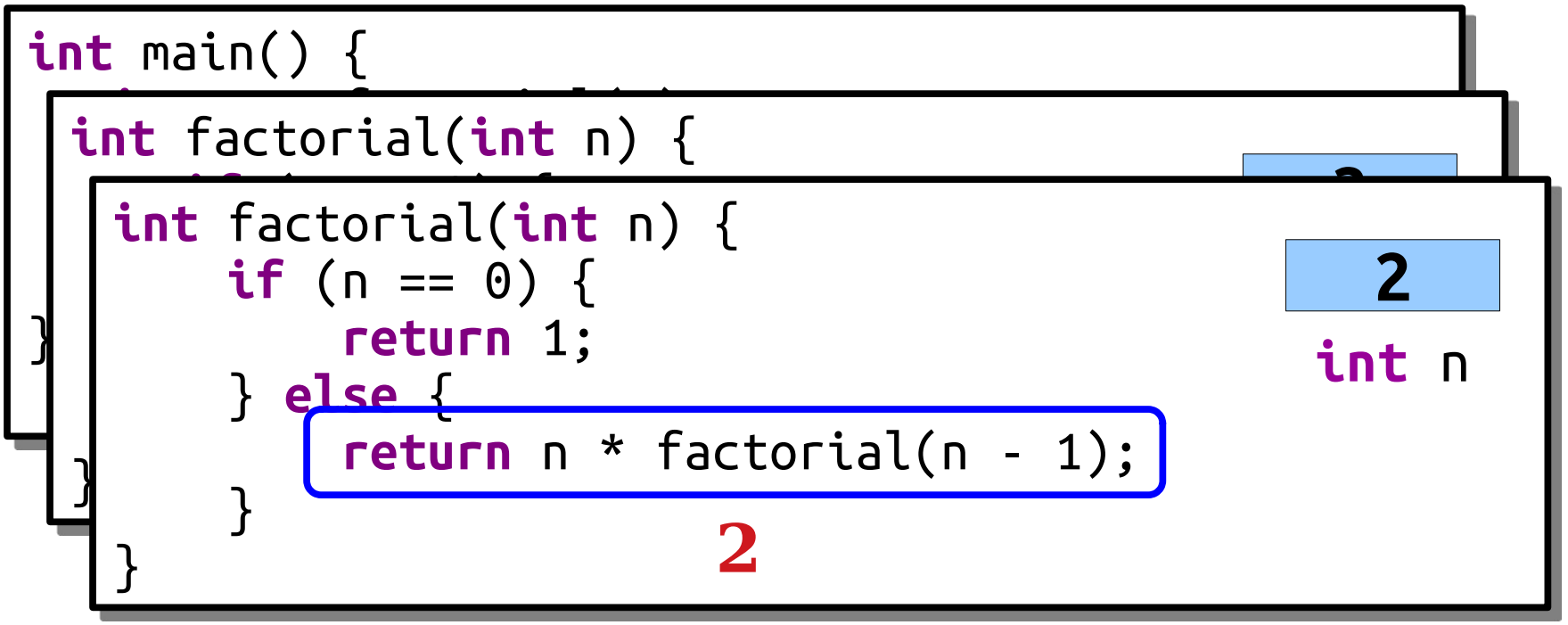
Recursion in Action



Recursion in Action



Recursion in Action



Recursion in Action

```
int main() {
}
```

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

3

int n

3

Recursion in Action

```
int main() {
}
```

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

3 2

3

int n

Recursion in Action

```
int main() {
}
```

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

3 2

3

int n

Recursion in Action

```
int main() {
}
```

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

3 × 2

3

int n

Recursion in Action

```
int main() {
}
```

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

6

3

`int n`

Recursion in Action

```
int main() {
   int nFact = factorial(3);
   cout << "3! = " << nFact << endl;

   return 0;
}
```

Recursion in Action

```
int main() {
   int nFact = factorial(3);
   cout << "3! = " << nFact << endl;

   return 0;
}
```

int nFact

Thinking Recursively

- Solving a problem with recursion requires two steps.
- First, determine how to solve the problem for simple cases.
  - This is called the ***base case***.
- Second, determine how to break down larger cases into smaller instances.
  - This is called the ***recursive step***.

Summing Up Digits

- In Lecture 1, we wrote this function to sum up the digits of a nonnegative integer:

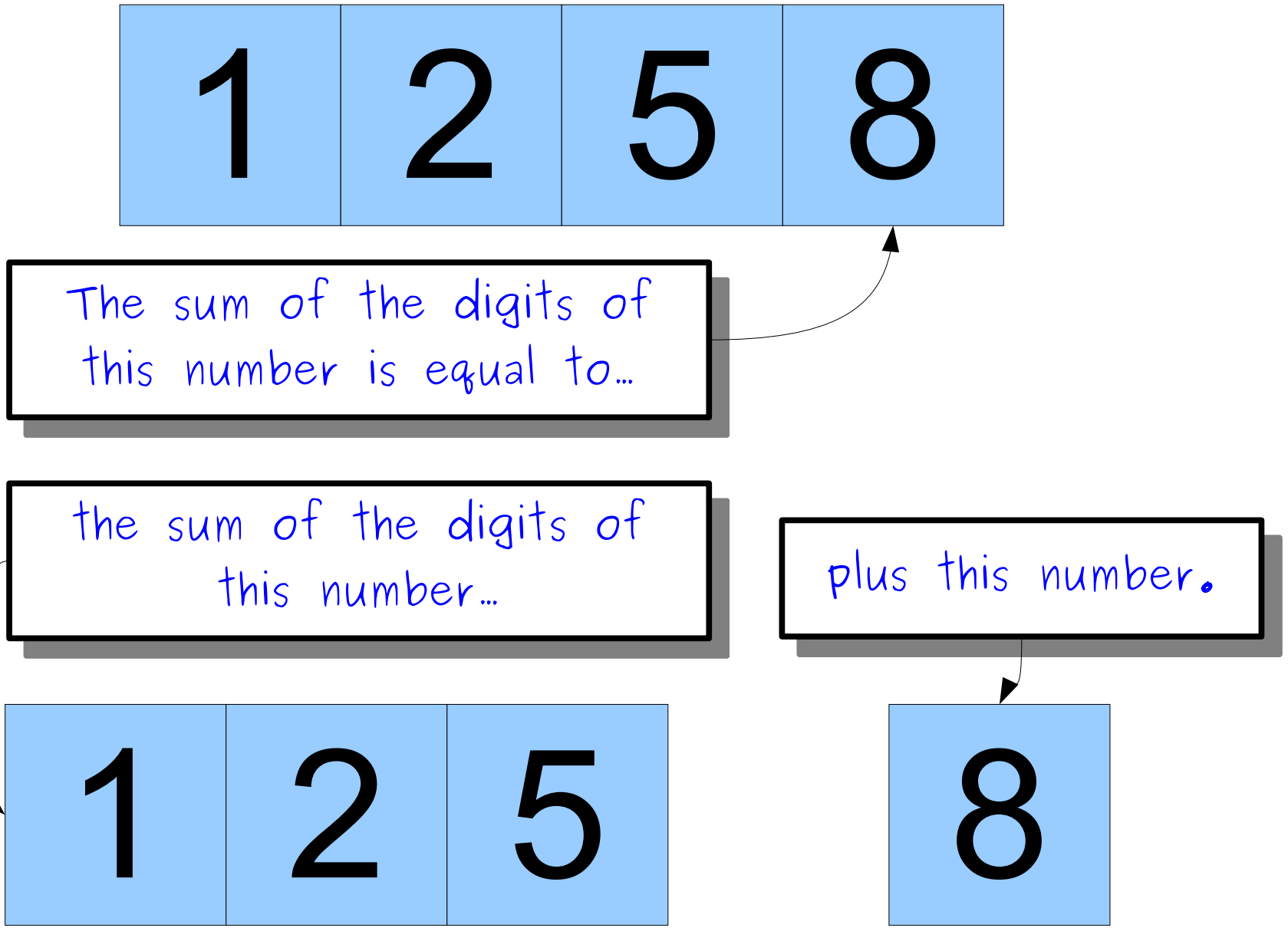
```
int sumOfDigitsOf(int n) {
    int result = 0;

    while (n > 0) {
        result += (n % 10);
        n /= 10;
    }

    return result;

}
```

- Let’s rewrite this function recursively!

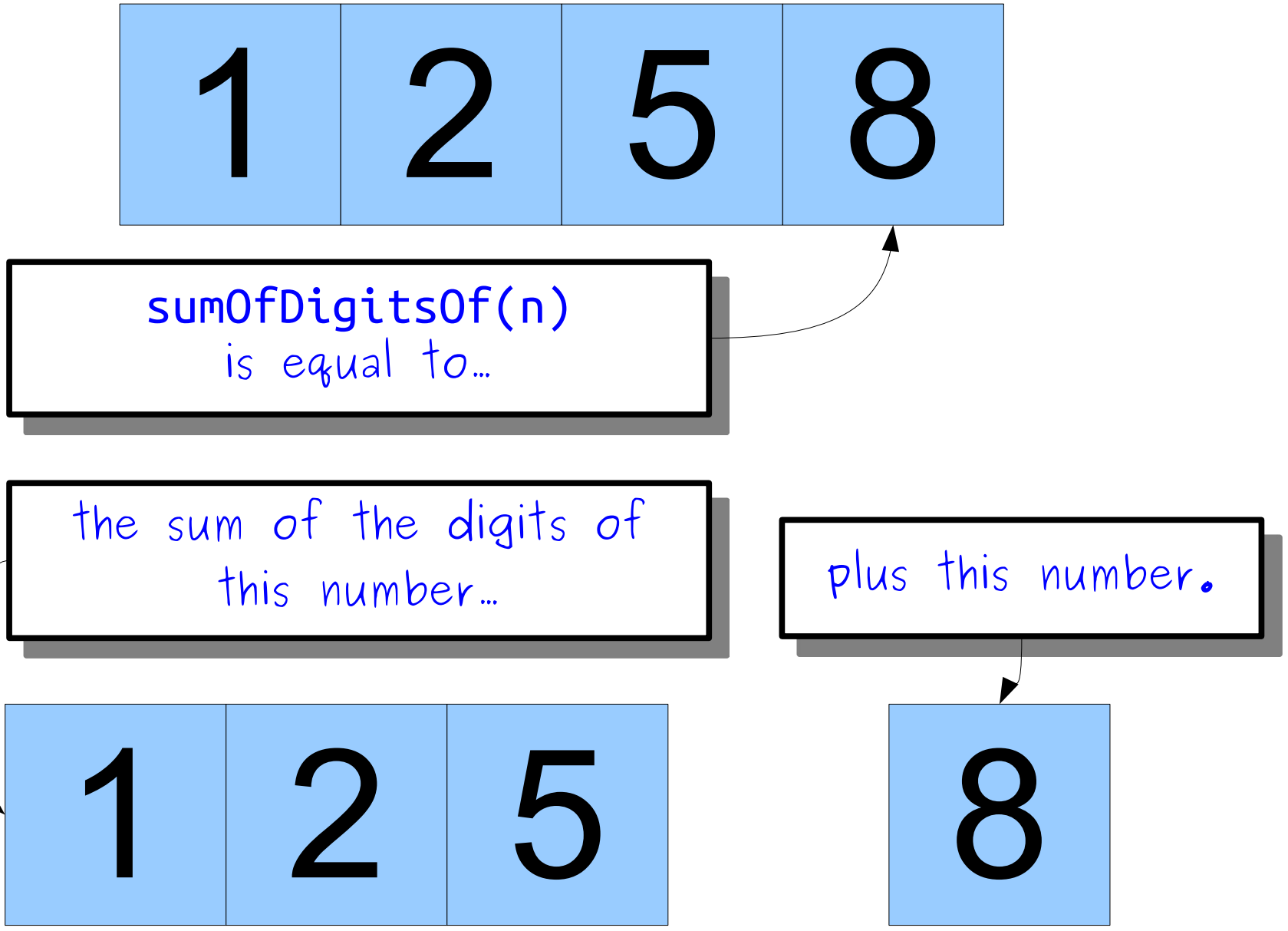
Summing Up Digits

- To write a recursive function, we need to think of a ***base case*** and a ***recursive case***.
- The ***base case*** produces answers when the input is sufficiently simple.
- The ***recursive case*** takes more complex inputs and simplifies them, taking them closer to the base case.
- What's a reasonable base case for our sum of digits function?

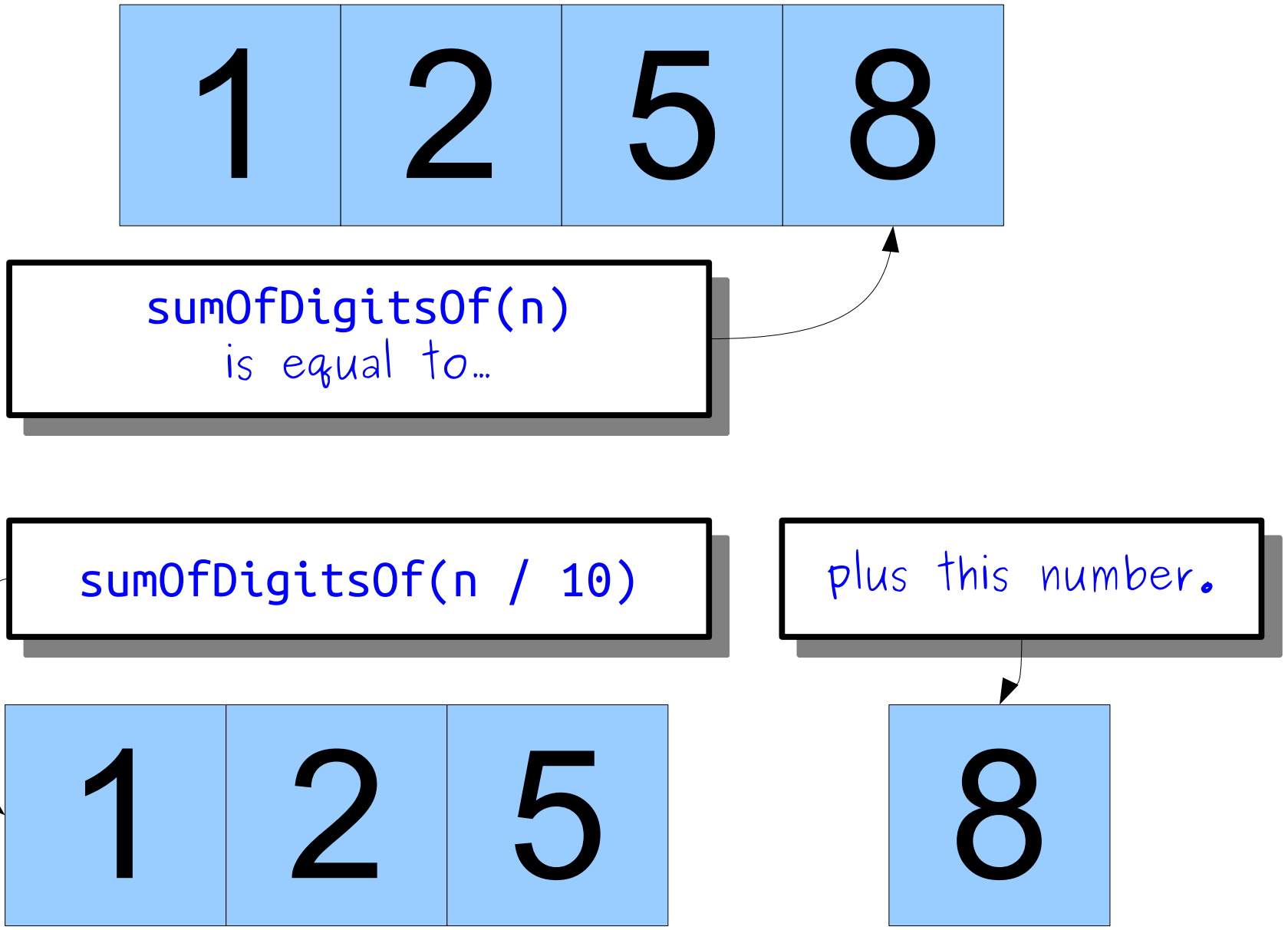
Summing Up Digits



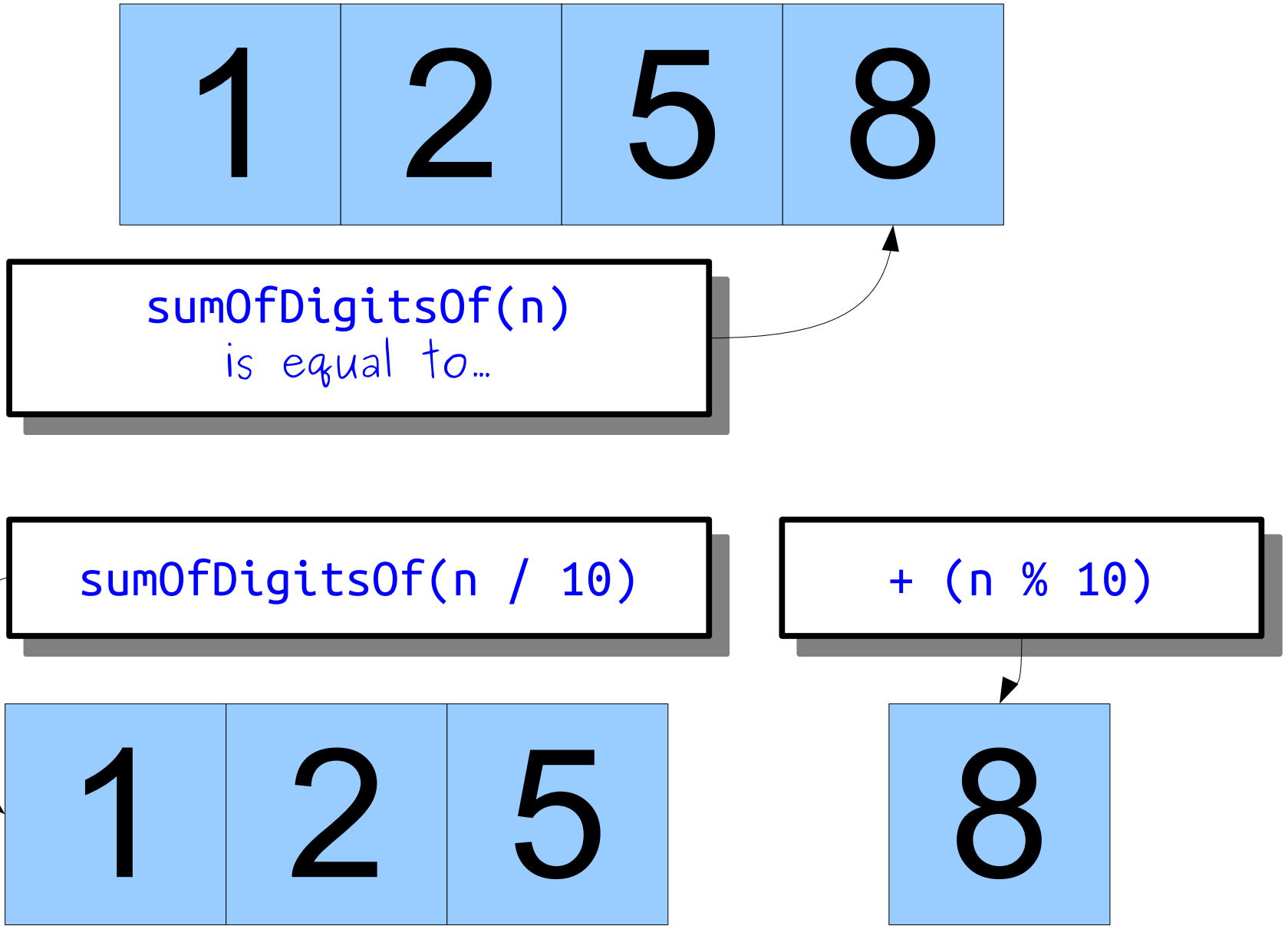
Summing Up Digits



Summing Up Digits



Summing Up Digits



Tracing the Recursion

```
int main() {
    int sum = sumOfDigitsOf(137);
    cout << "Sum is " << sum << endl;
}
```

Tracing the Recursion

```
int main() {
    int sum = sumOfDigitsOf(137);
    cout << "Sum is " << sum << endl;
}
```

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

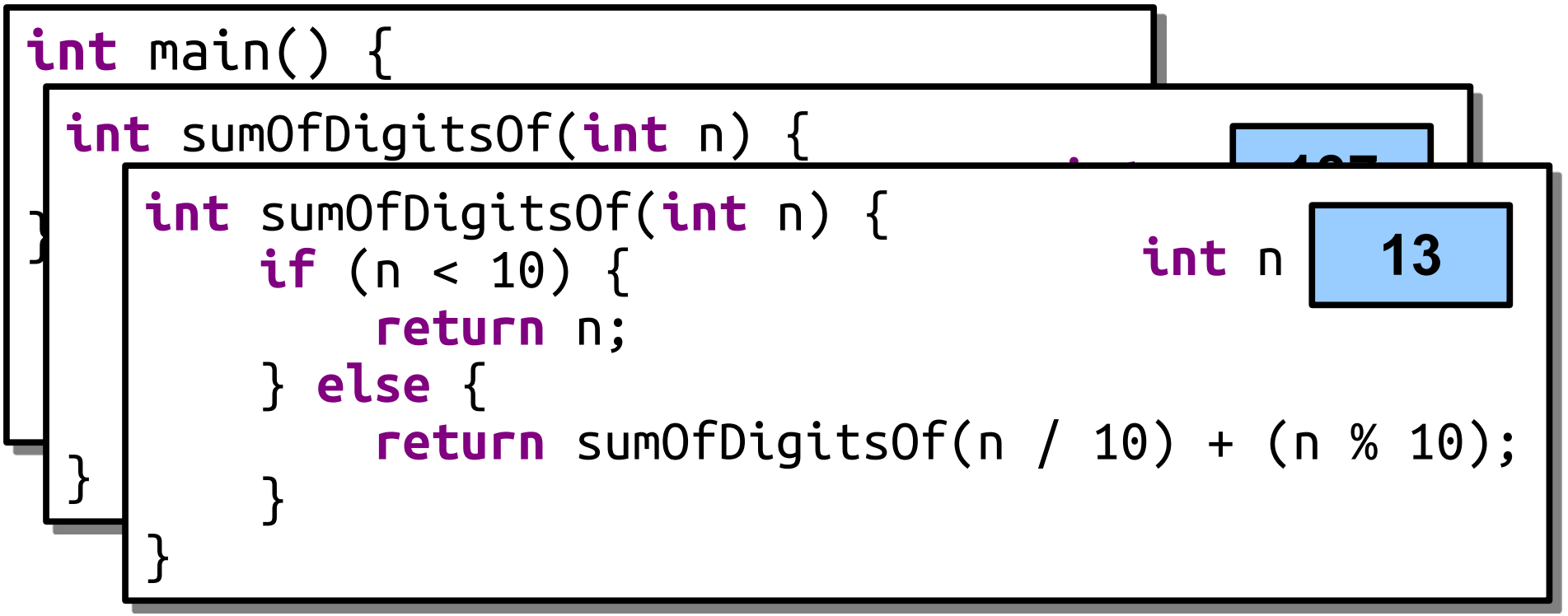
Tracing the Recursion

```
int main() {
}
```

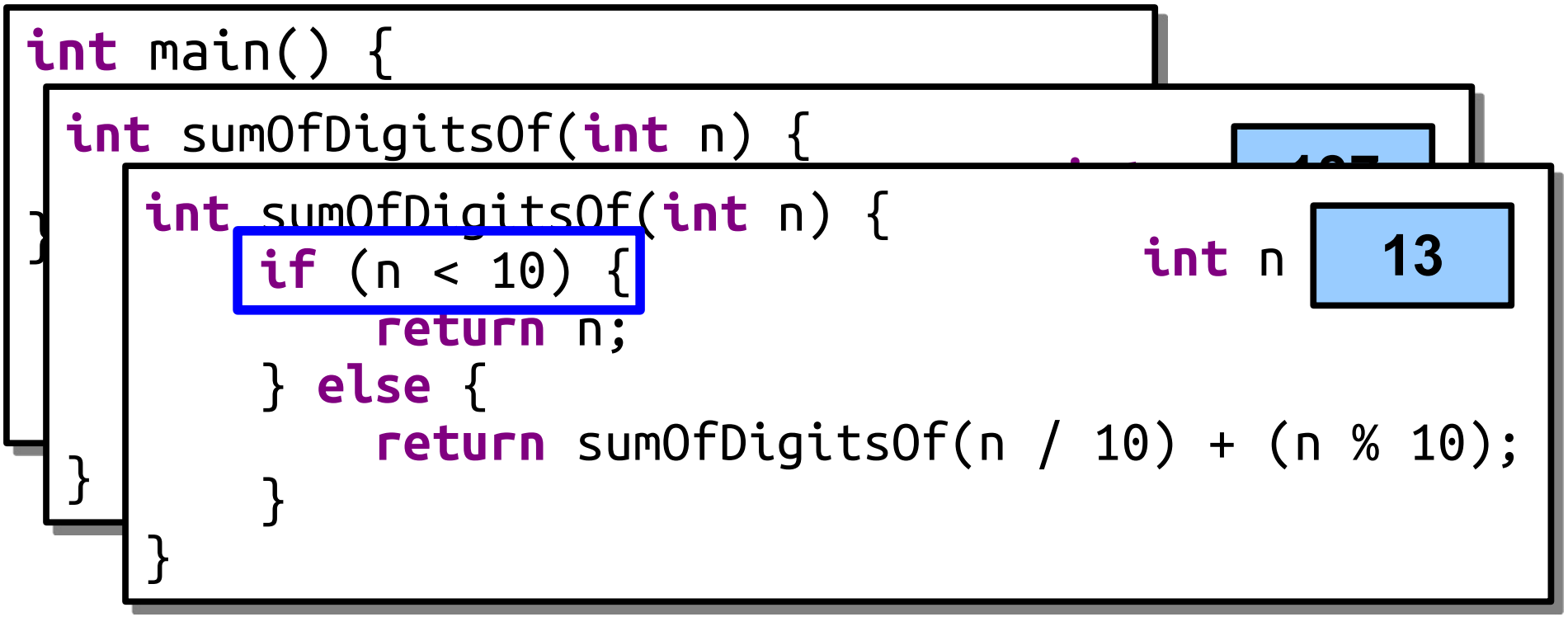
```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

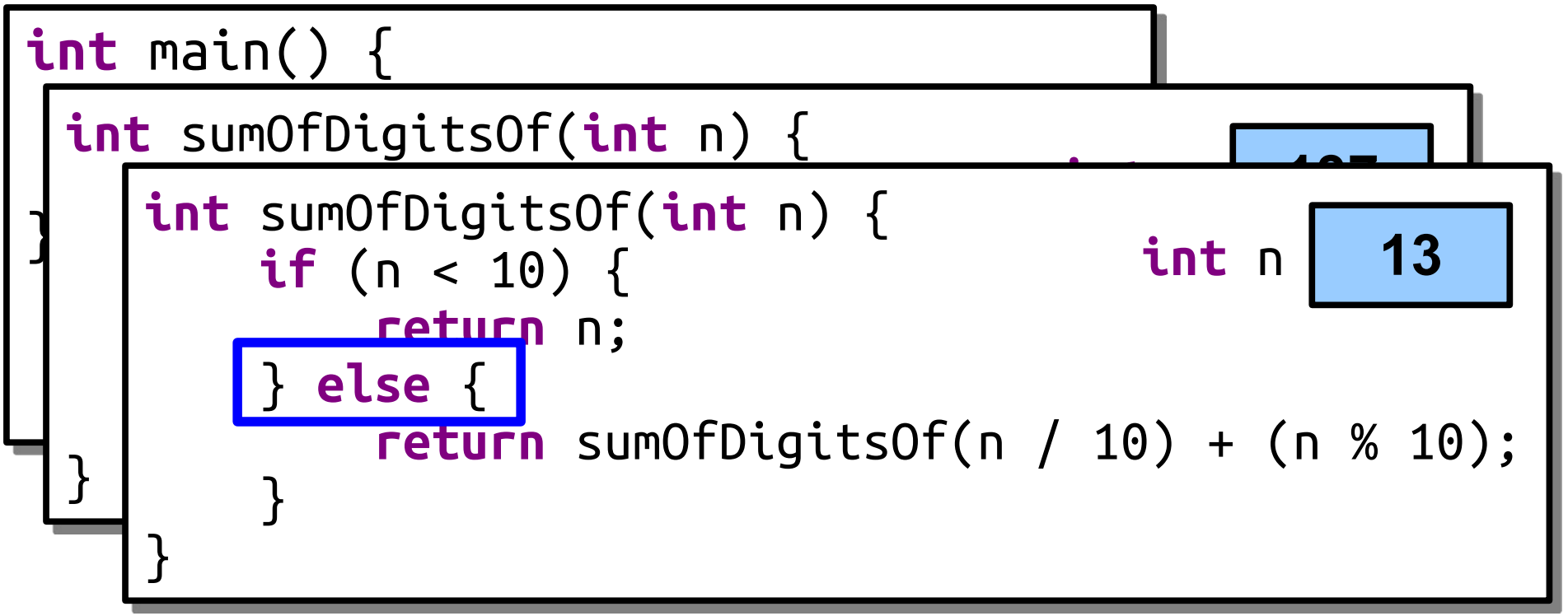
Tracing the Recursion



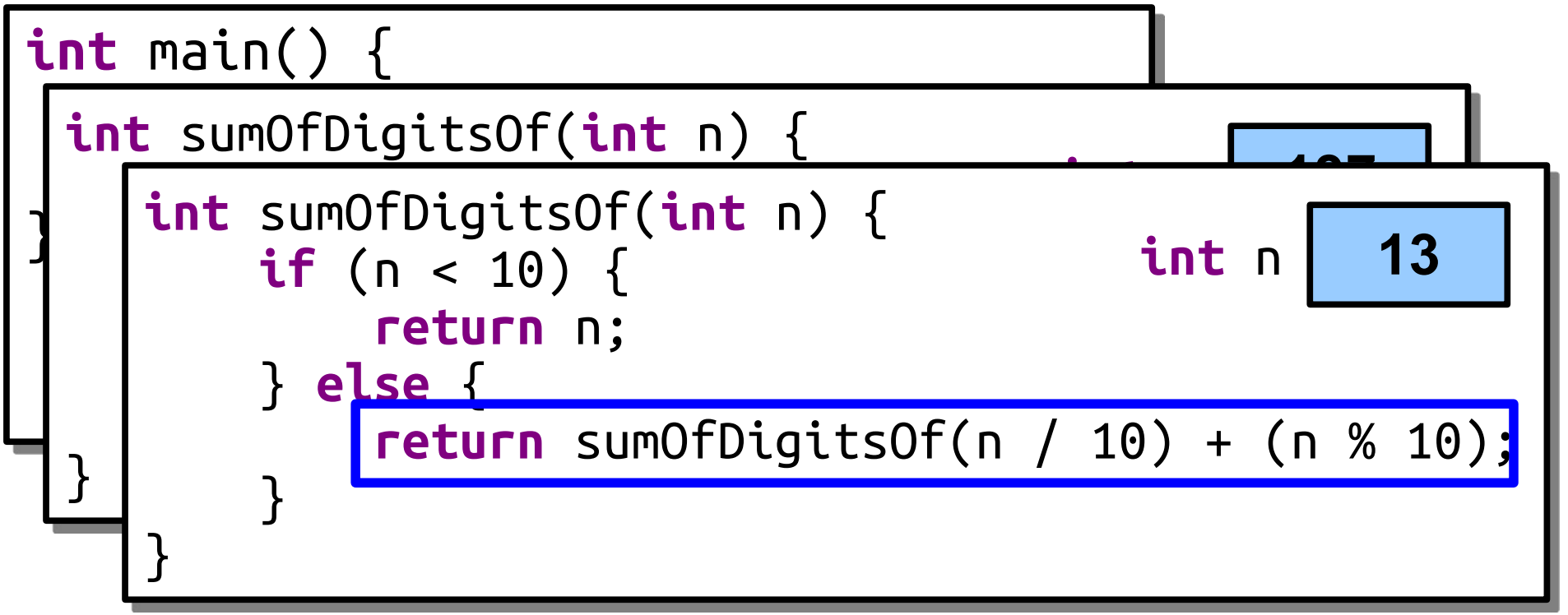
Tracing the Recursion



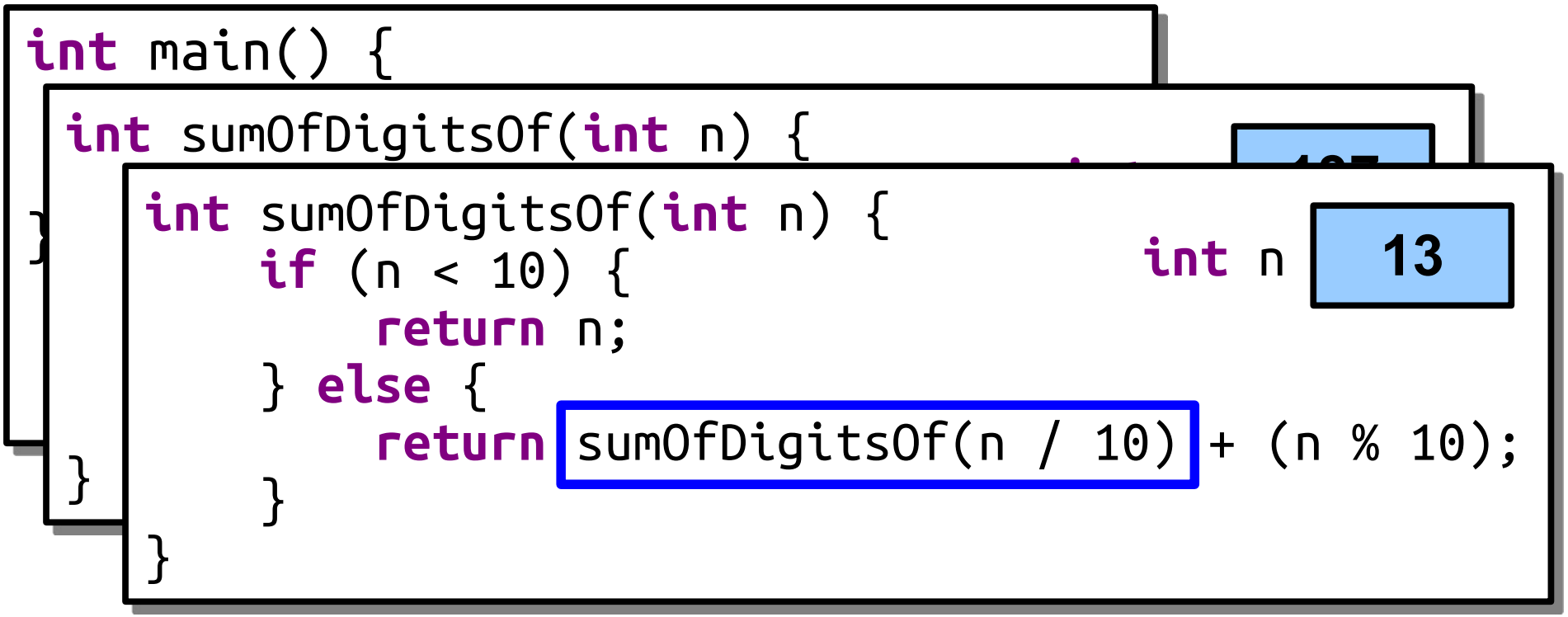
Tracing the Recursion



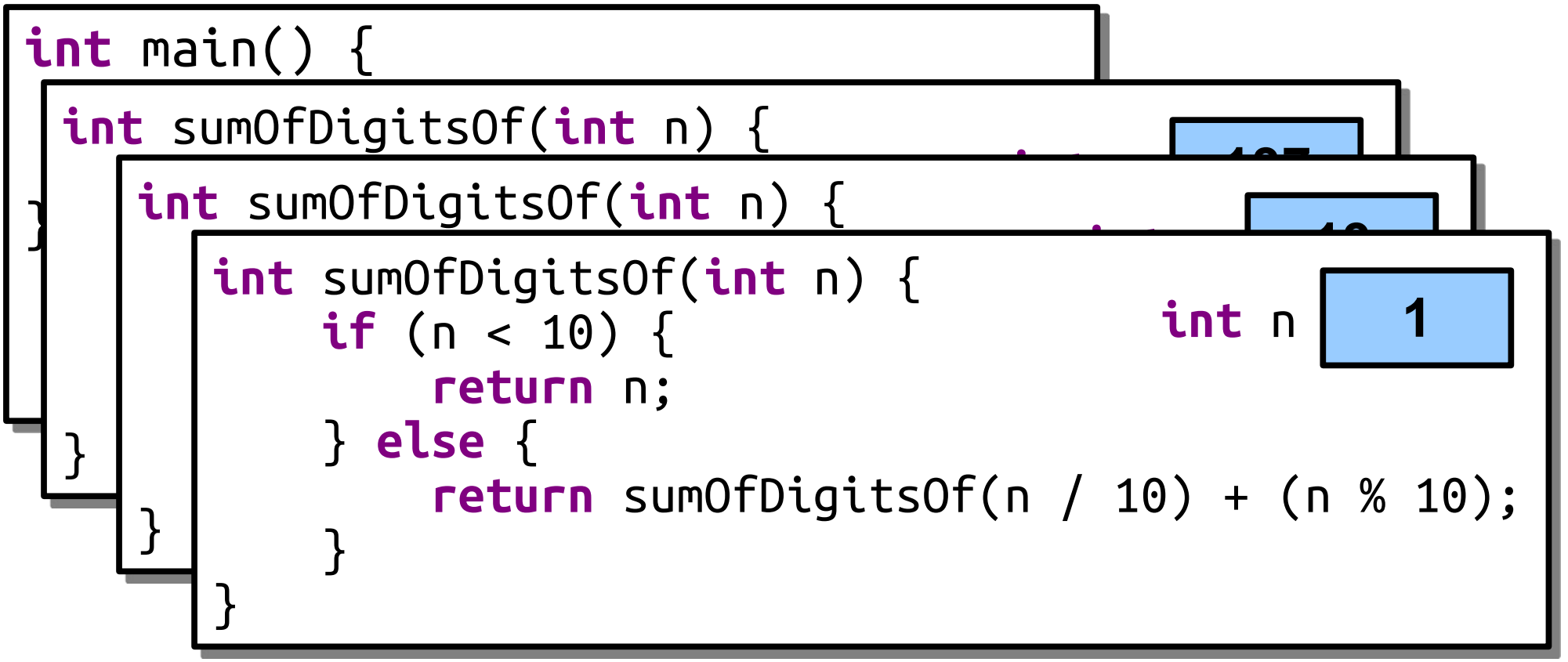
Tracing the Recursion



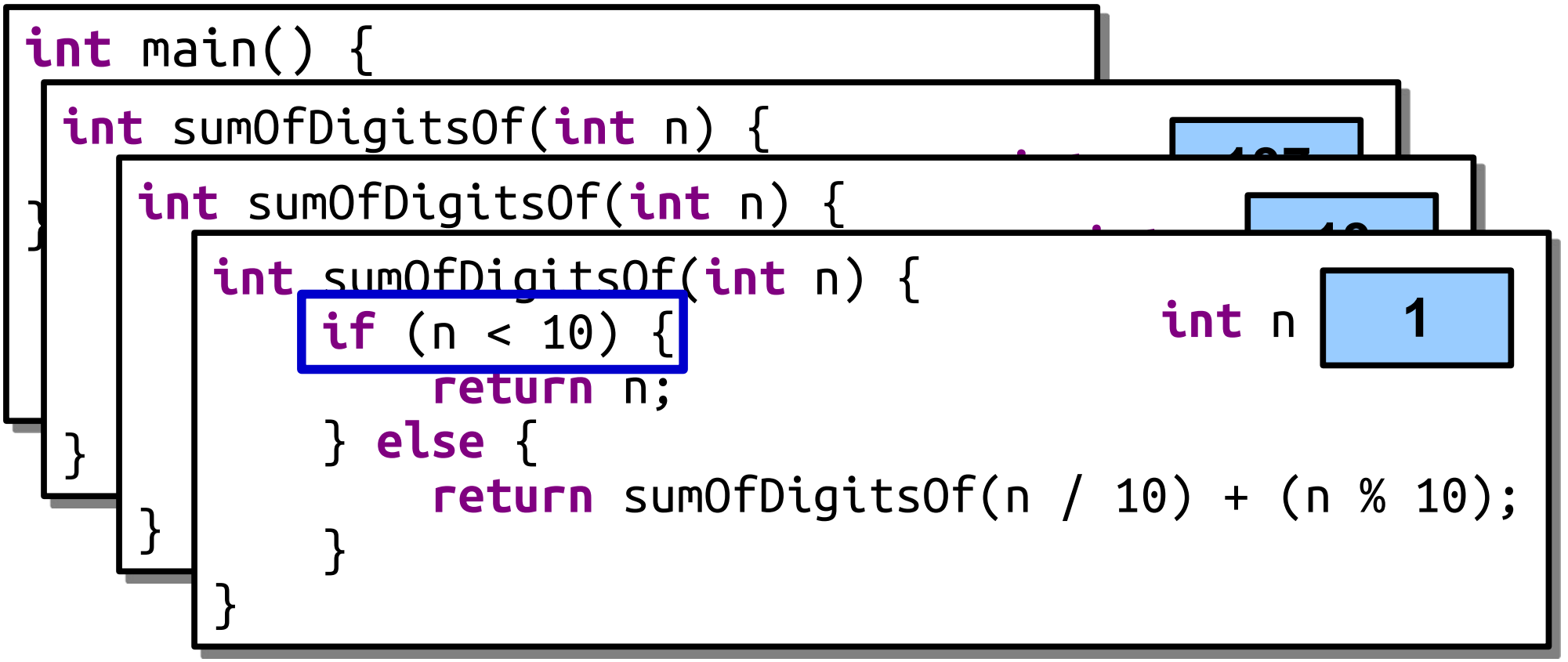
Tracing the Recursion



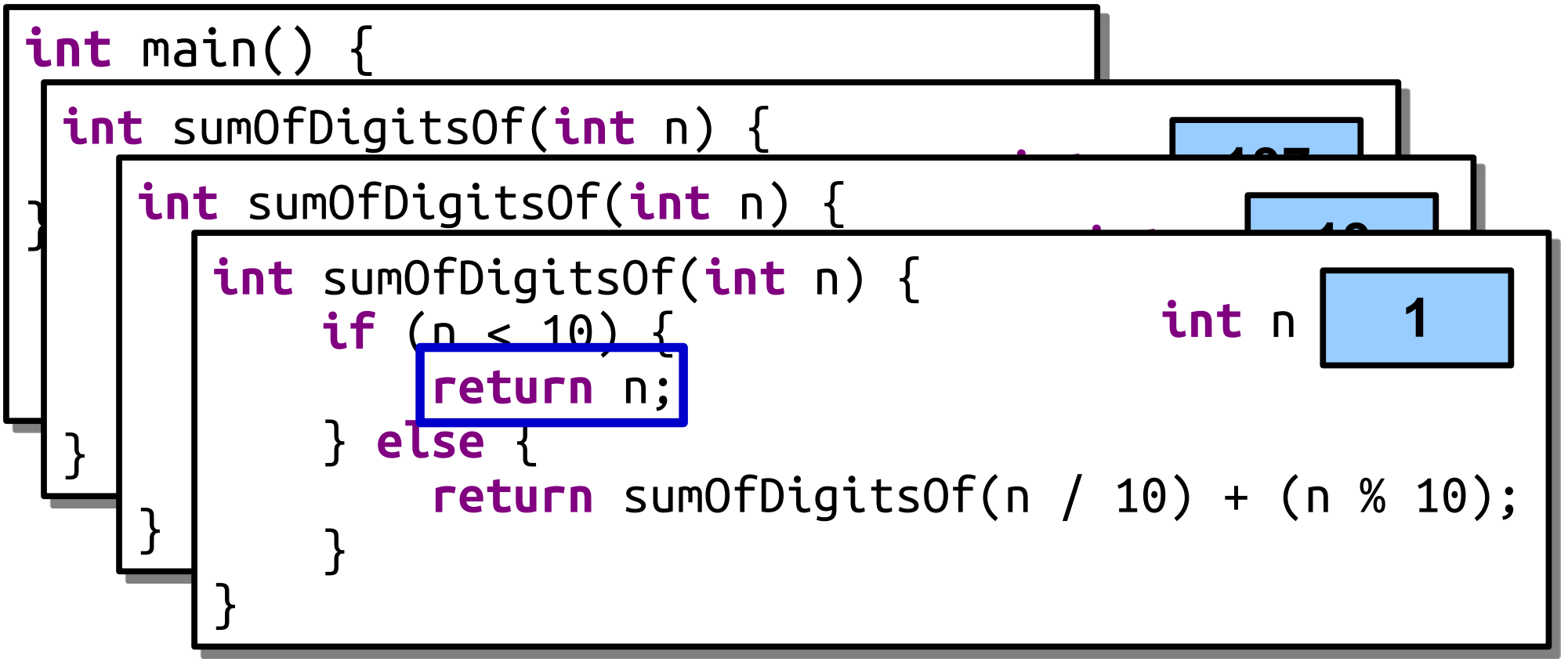
Tracing the Recursion



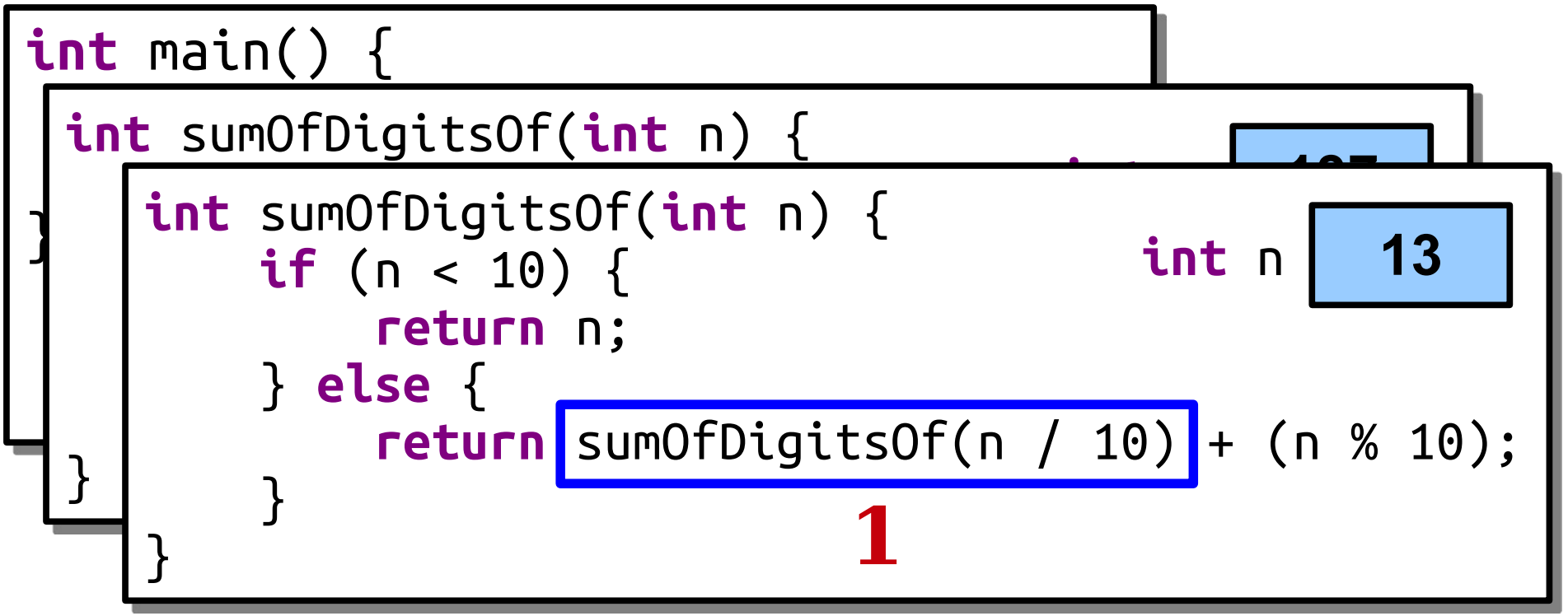
Tracing the Recursion



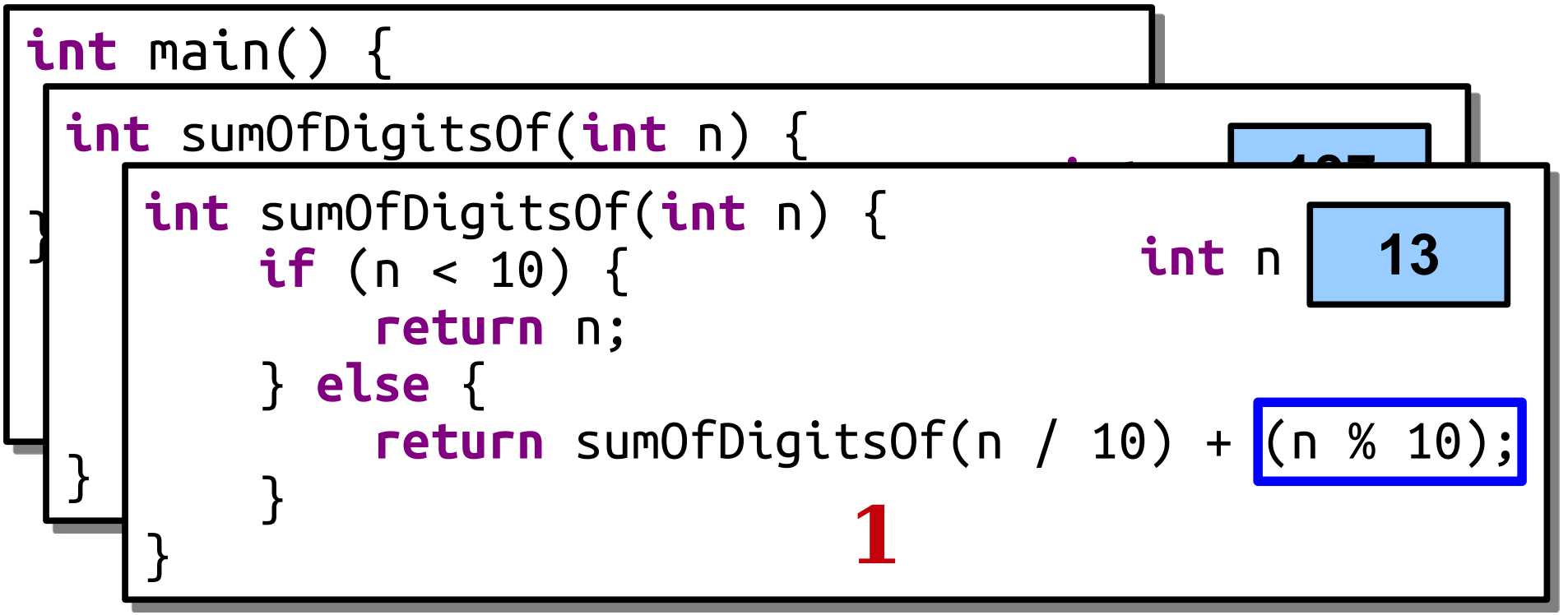
Tracing the Recursion



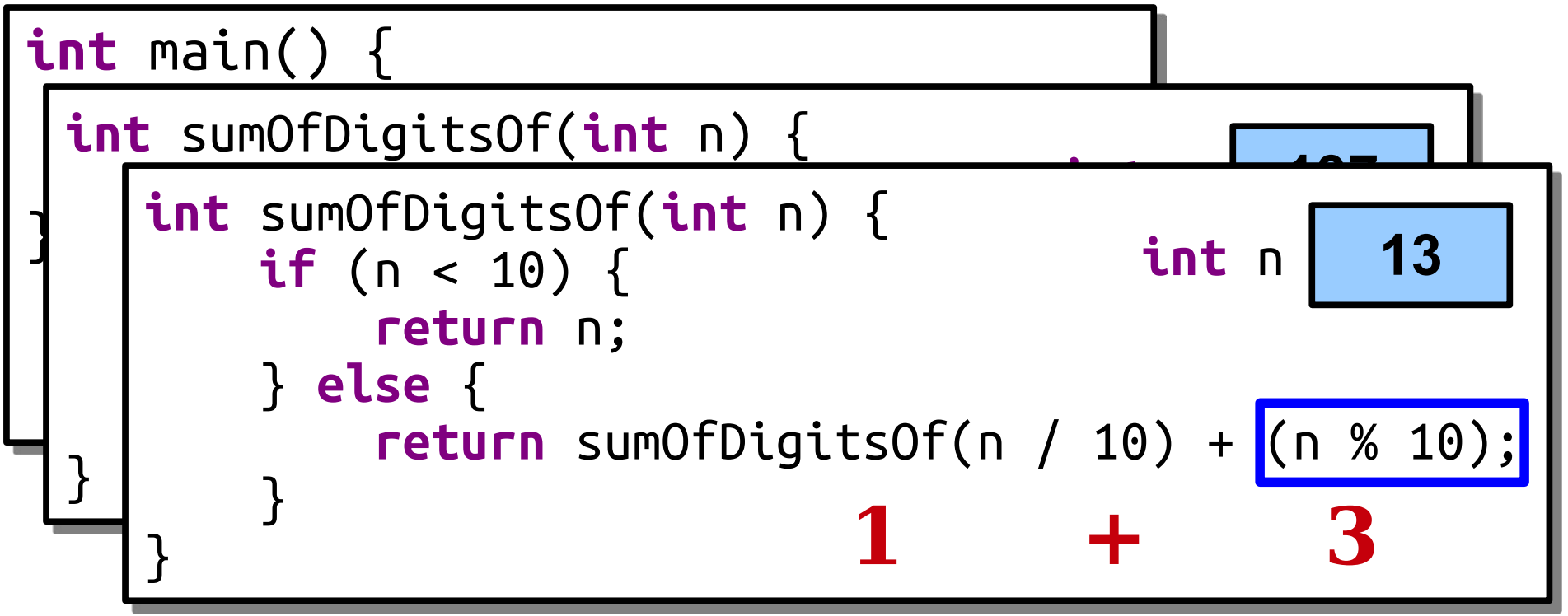
Tracing the Recursion



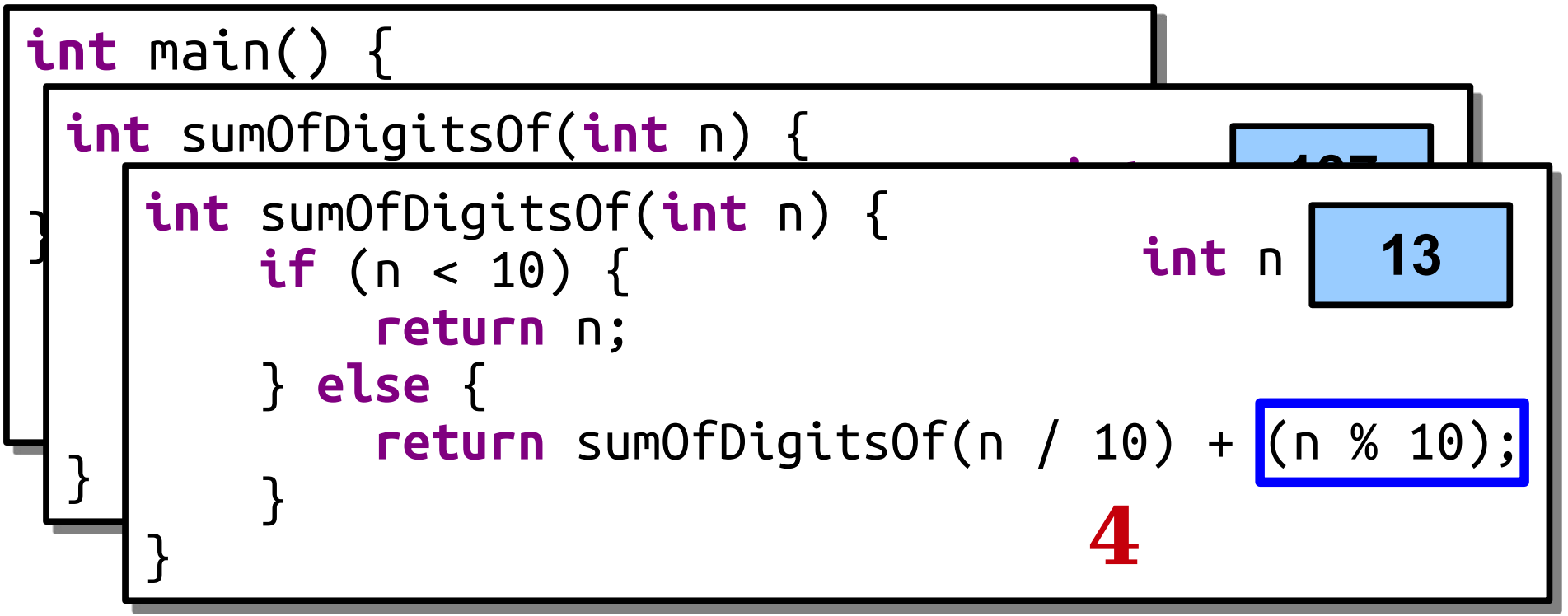
Tracing the Recursion



Tracing the Recursion



Tracing the Recursion



Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

4

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

4

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

4 + 7

Tracing the Recursion

```
int main() {
}
```

```
int sumOfDigitsOf(int n) {
    if (n < 10) {
        return n;
    } else {
        return sumOfDigitsOf(n / 10) + (n % 10);
    }
}
```

int n **137**

11

Tracing the Recursion

```
int main() {
    int sum = sumOfDigitsOf(137);
    cout << "Sum is " << sum << endl;
}
```

11

Thinking Recursively

```
if (The problem is very simple) {
    Directly solve the problem.
    Return the solution.
} else {
    Split the problem into one or more
    smaller problems with the same
    structure as the original.
    Solve each of those smaller problems.
    Combine the results to get the overall
    solution.
    Return the overall solution.
}
```

These simple cases are called *base cases*.

These are the *recursive cases*.

Time-Out for Announcements!

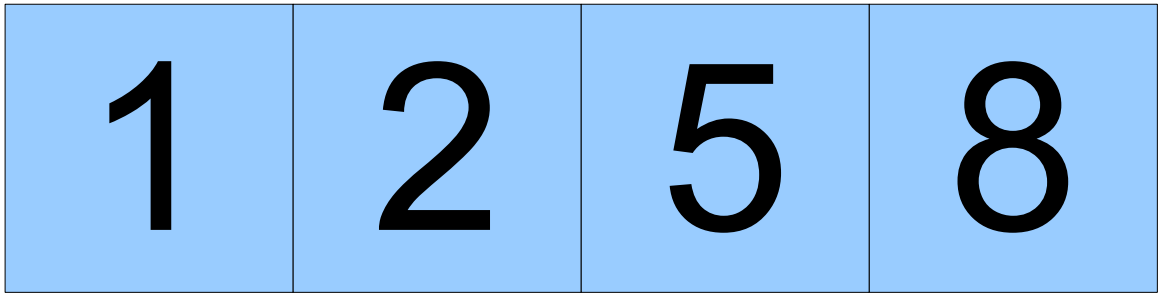
Assignment 1

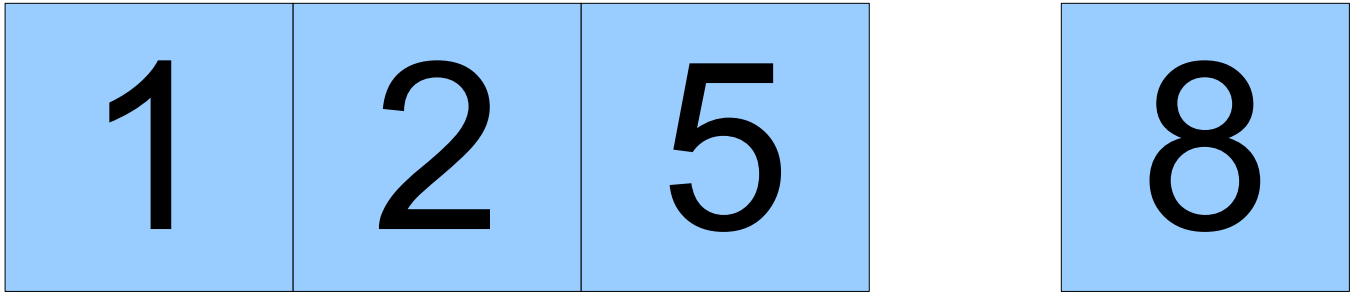
- Assignment 0 was due today.
- ***Assignment 1: Welcome to C++*** goes out today.
  - Play around with C++ and the Stanford libraries!
  - Get some practice with recursion!
  - Explore the debugger!
  - See some pretty pictures!
- We recommend making slow and steady progress on this assignment throughout the course of the week. There's a recommended timetable at the top of the assignment description.

One More Unto the Breach!

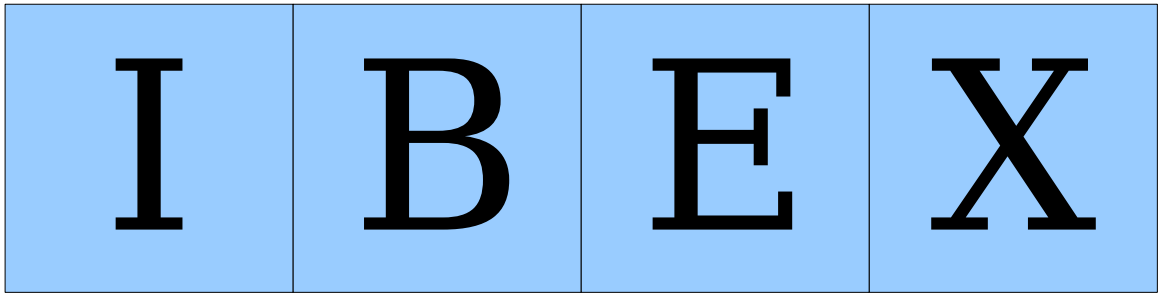
Recursion and Strings

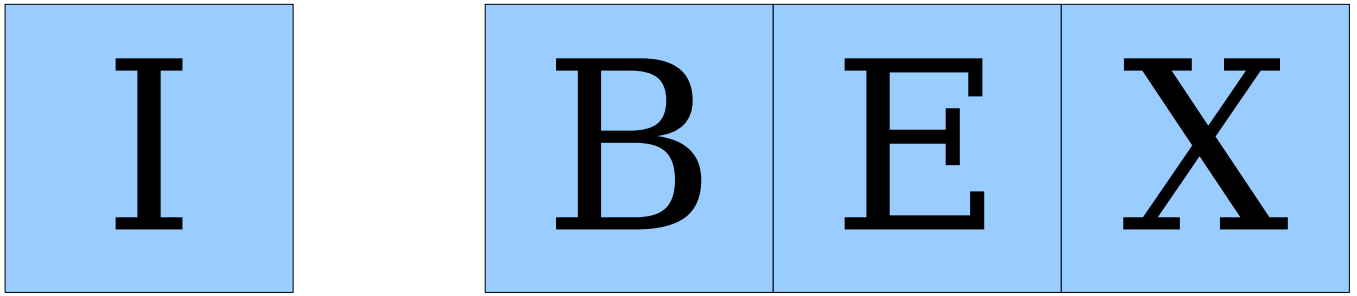
Thinking Recursively



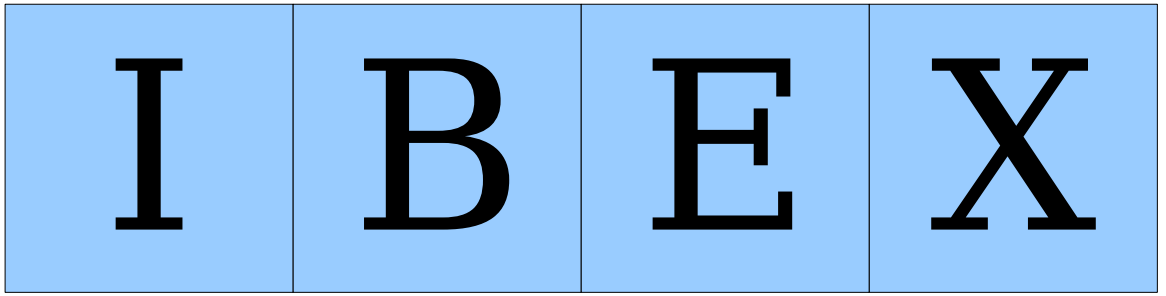


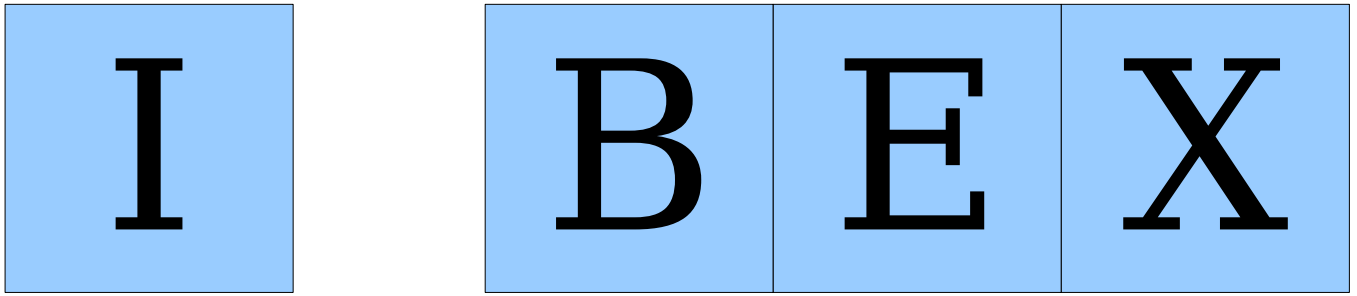
Thinking Recursively





Thinking Recursively





`str[0]`

`str.substr(1)`

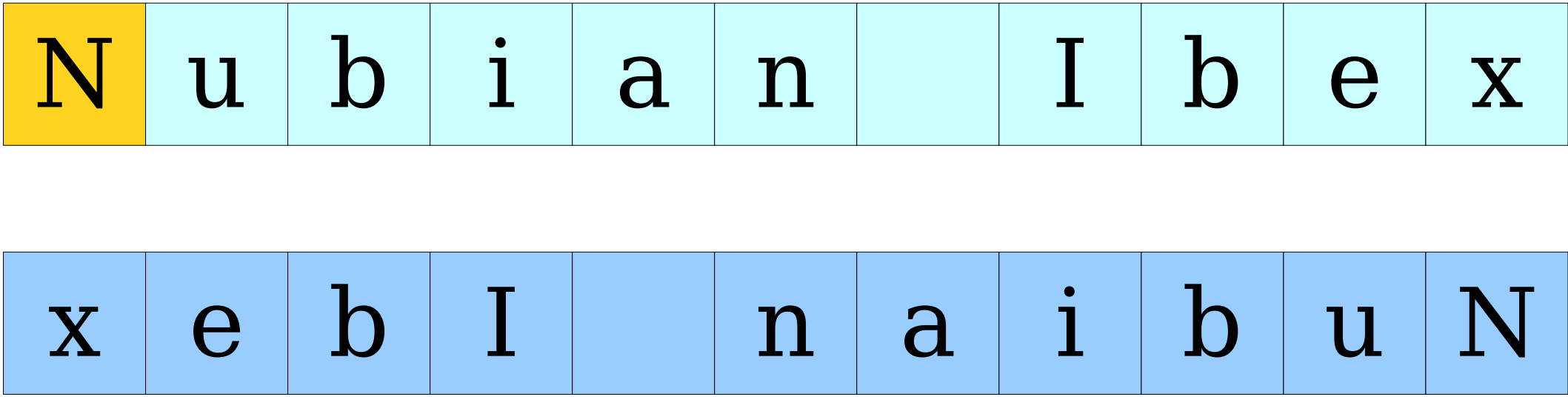
How do you reverse a string?
?gnirts a esrever uoy od woH

Reversing a String

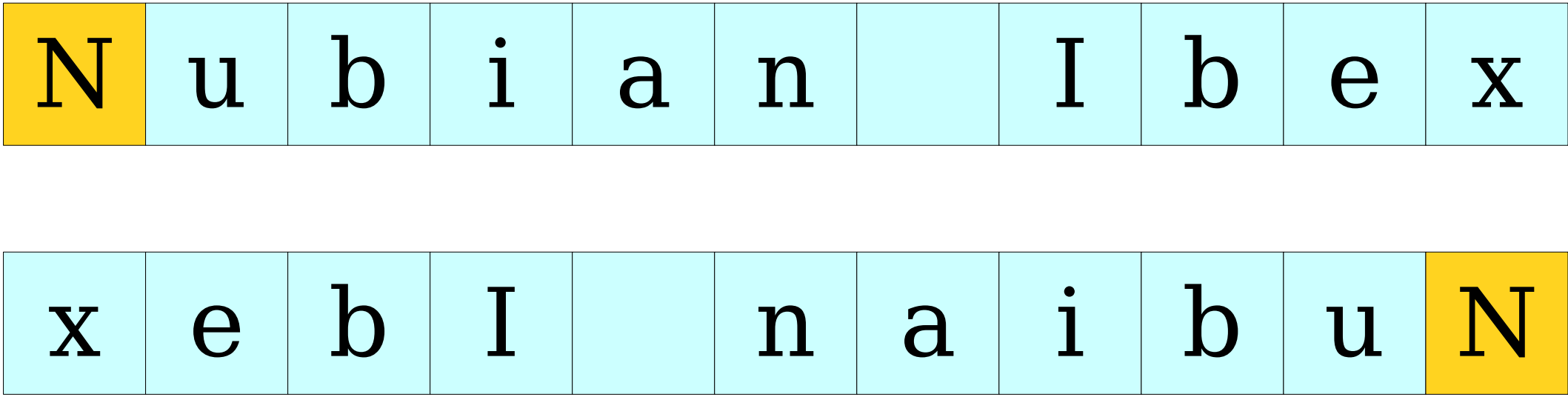
N	u	b	i	a	n		I	b	e	x

x	e	b	I		n	a	i	b	u	N

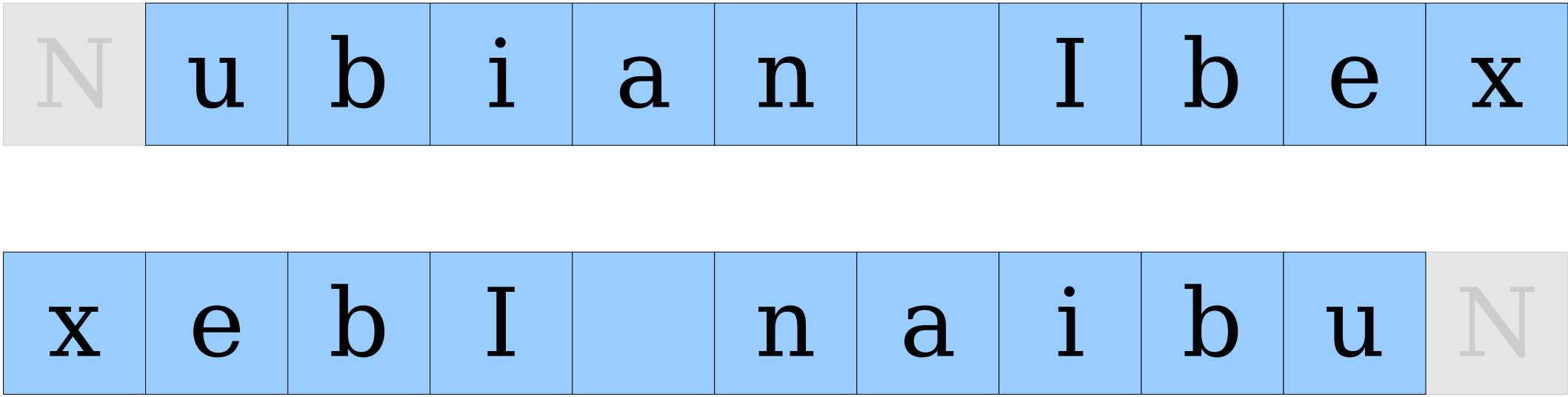
Reversing a String



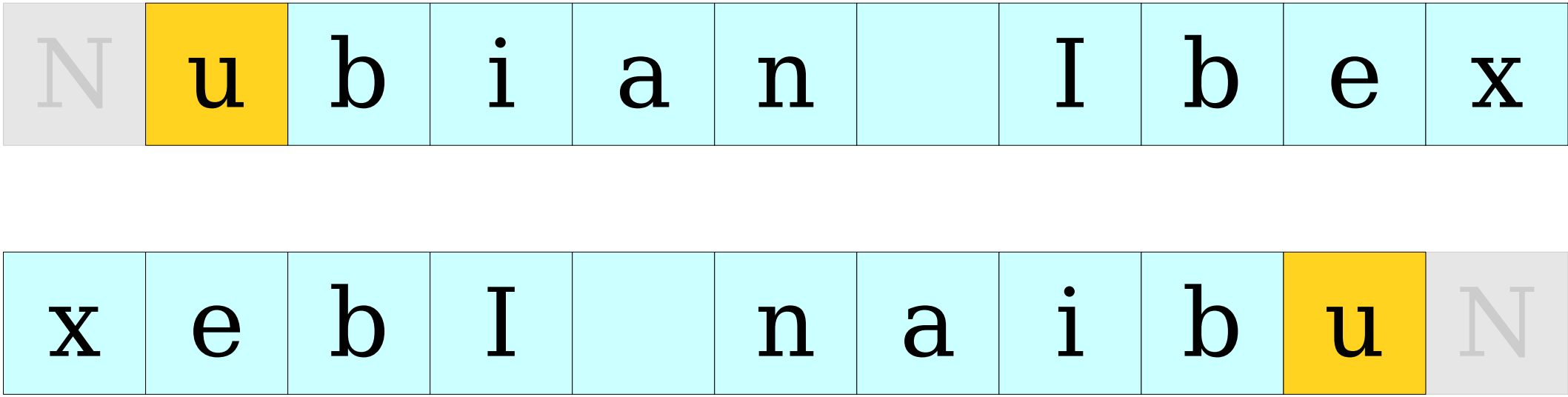
Reversing a String



Reversing a String



Reversing a String



Reversing a String Recursively

Reversing a String Recursively

reverseOf("TOP ") =

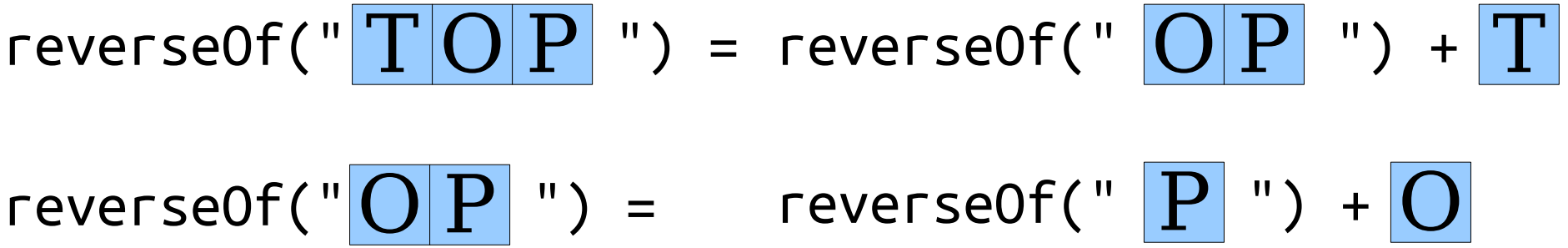
Reversing a String Recursively

Reversing a String Recursively

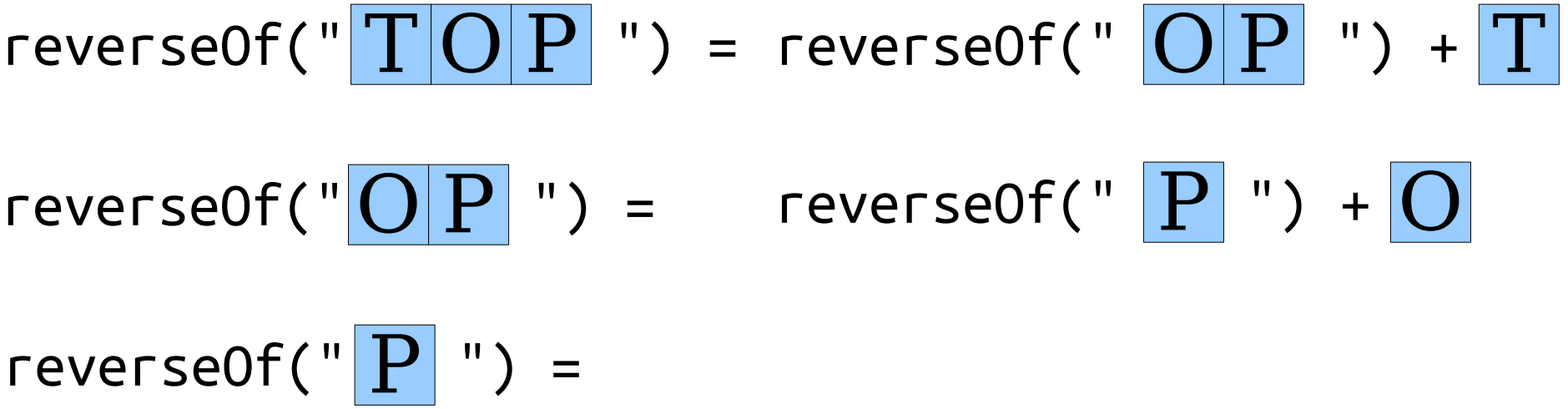
reverseOf("TOP") = reverseOf("OP") + T

reverseOf("OP") =

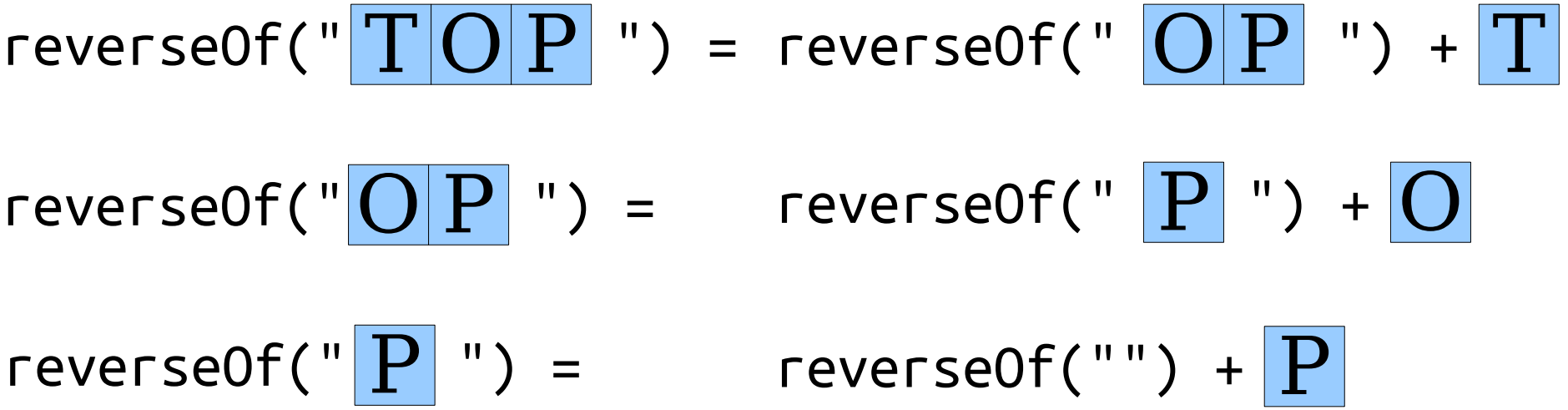
Reversing a String Recursively



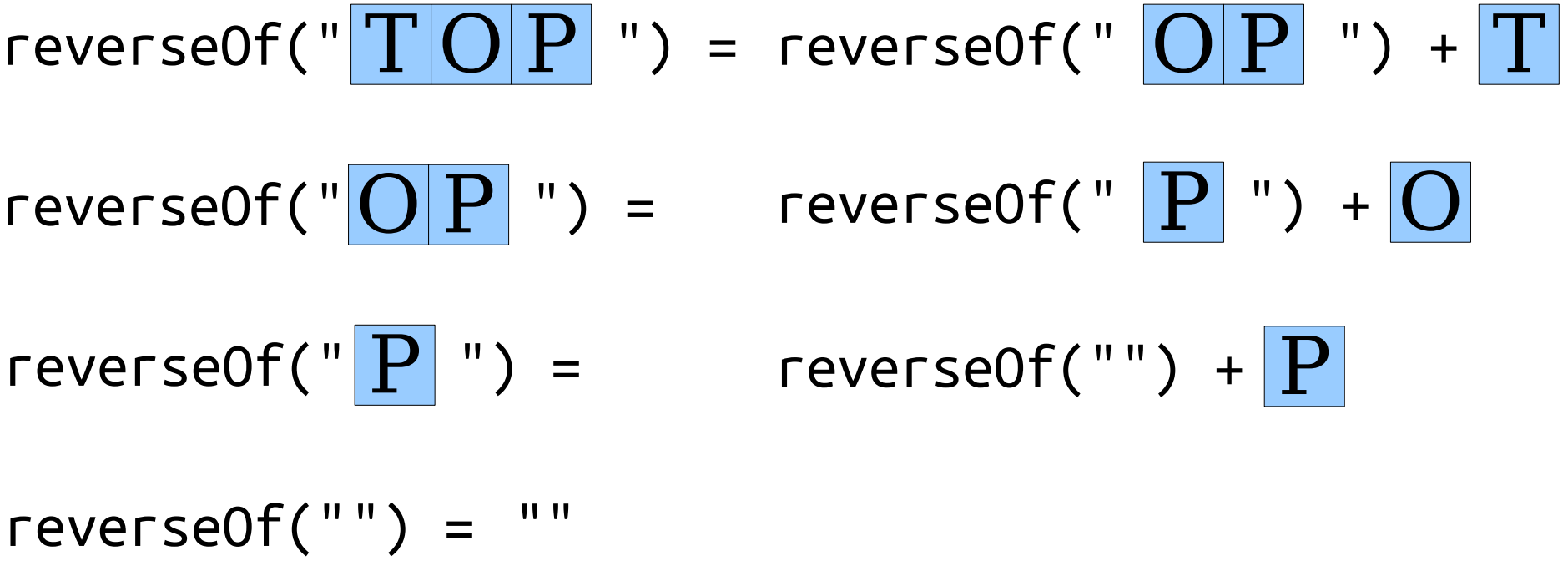
Reversing a String Recursively



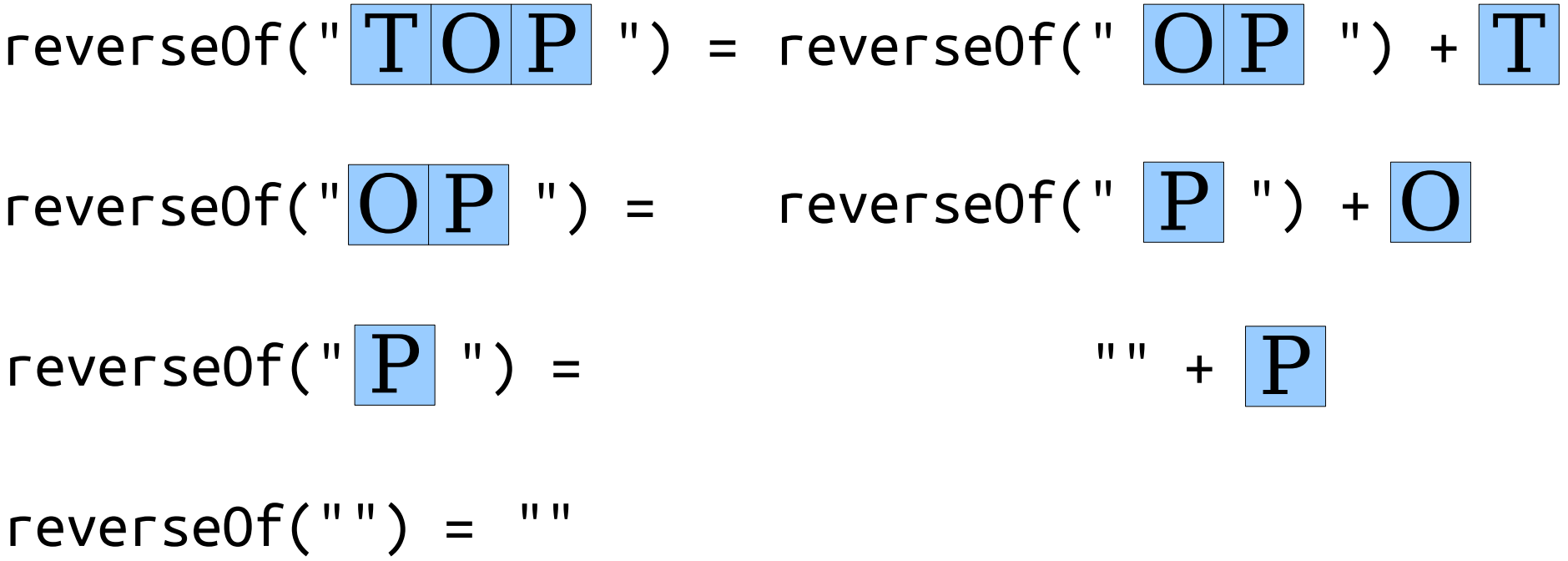
Reversing a String Recursively



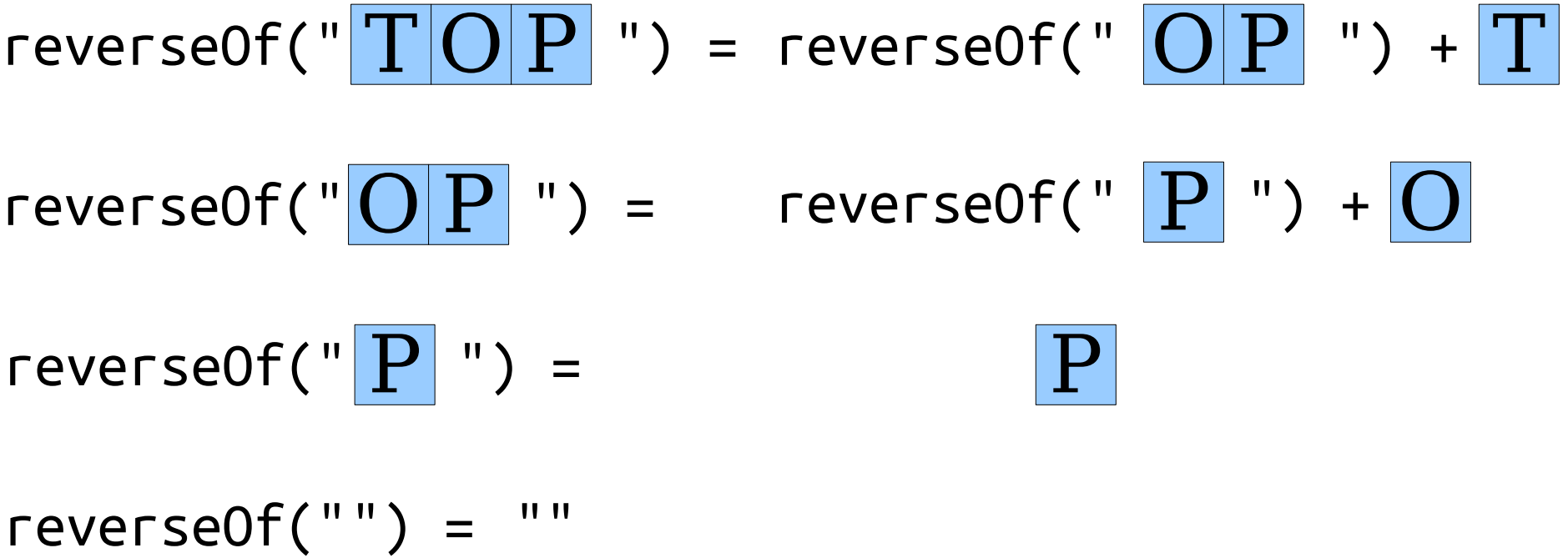
Reversing a String Recursively



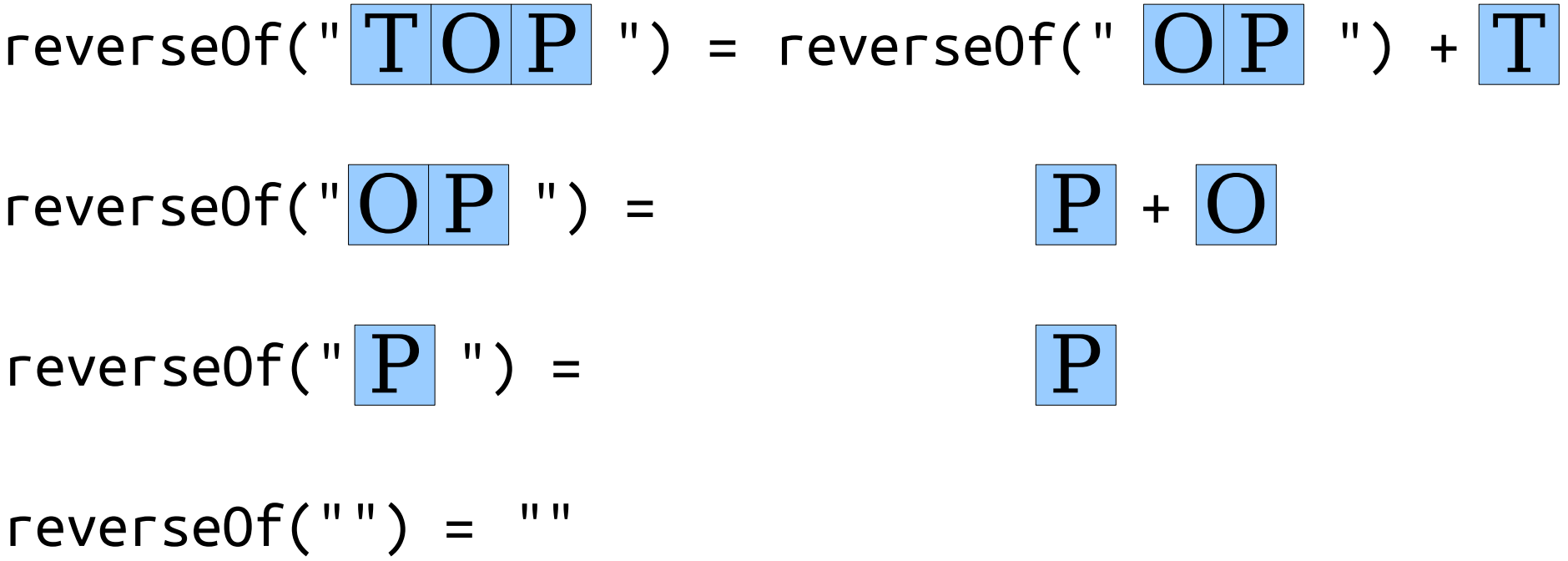
Reversing a String Recursively



Reversing a String Recursively



Reversing a String Recursively



Reversing a String Recursively

reverseOf("TOP") = reverseOf("OP") + T

reverseOf("OP") = PO

reverseOf("P") = P

reverseOf("") = ""

Reversing a String Recursively

reverseOf("TOP") = POP + T

reverseOf("OP") = PO

reverseOf("P") = P

reverseOf("") = ""

Reversing a String Recursively

reverseOf("TOP") = POT

reverseOf("OP") = PO

reverseOf("P") = P

reverseOf("") = ""

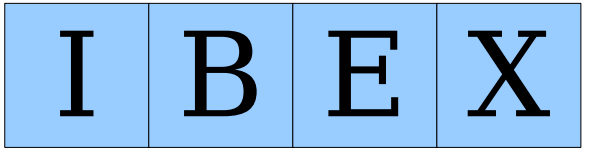
Reversing a String Recursively

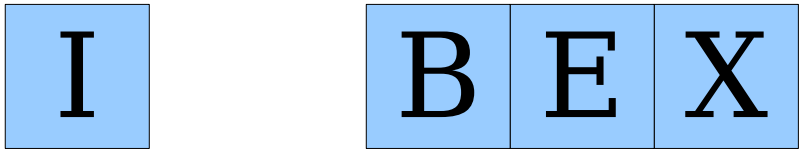
reverseOf("TOP") = reverseOf("OP") + T

reverseOf("OP") = reverseOf("P") + O

reverseOf("P") = reverseOf("") + P

reverseOf("") = ""

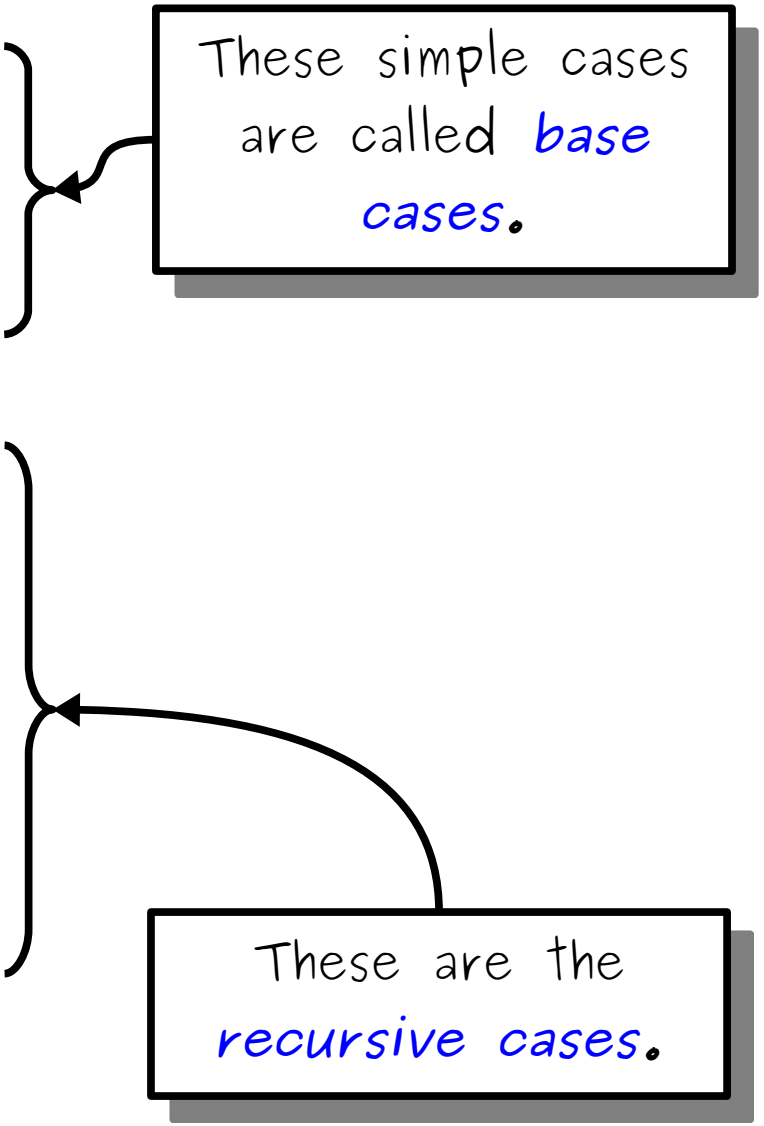




input[0] input.substr(1)

Thinking Recursively

```
if (The problem is very simple) {
   Directly solve the problem.
   Return the solution.
} else {
   Split the problem into one or more
   smaller problems with the same
   structure as the original.
   Solve each of those smaller problems.
   Combine the results to get the overall
   solution.
   Return the overall solution.
}
```



Recap from Today

- Recursion works by identifying
  - one or more ***base cases***, simple cases that can be solved directly, and
  - one or more ***recursive cases***, where a larger problem is turned into a smaller one.
- Recursion is everywhere! And you can use it on strings.

Your Action Items

- *Prepare* ***for a Discussion Section***
- ***Read Chapter 7.***
  - This chapter is all about recursion.
- ***Start Working on Assignment 1.***
  - Aim to complete the Debugger Warmups and start working on Fire.

Next Time

- ***Reference Parameters***
  - On master copies and xeroxes.
- ***`Vector`***
  - Representing sequences.
- ***Recursion on `Vectors`***
  - Of course.