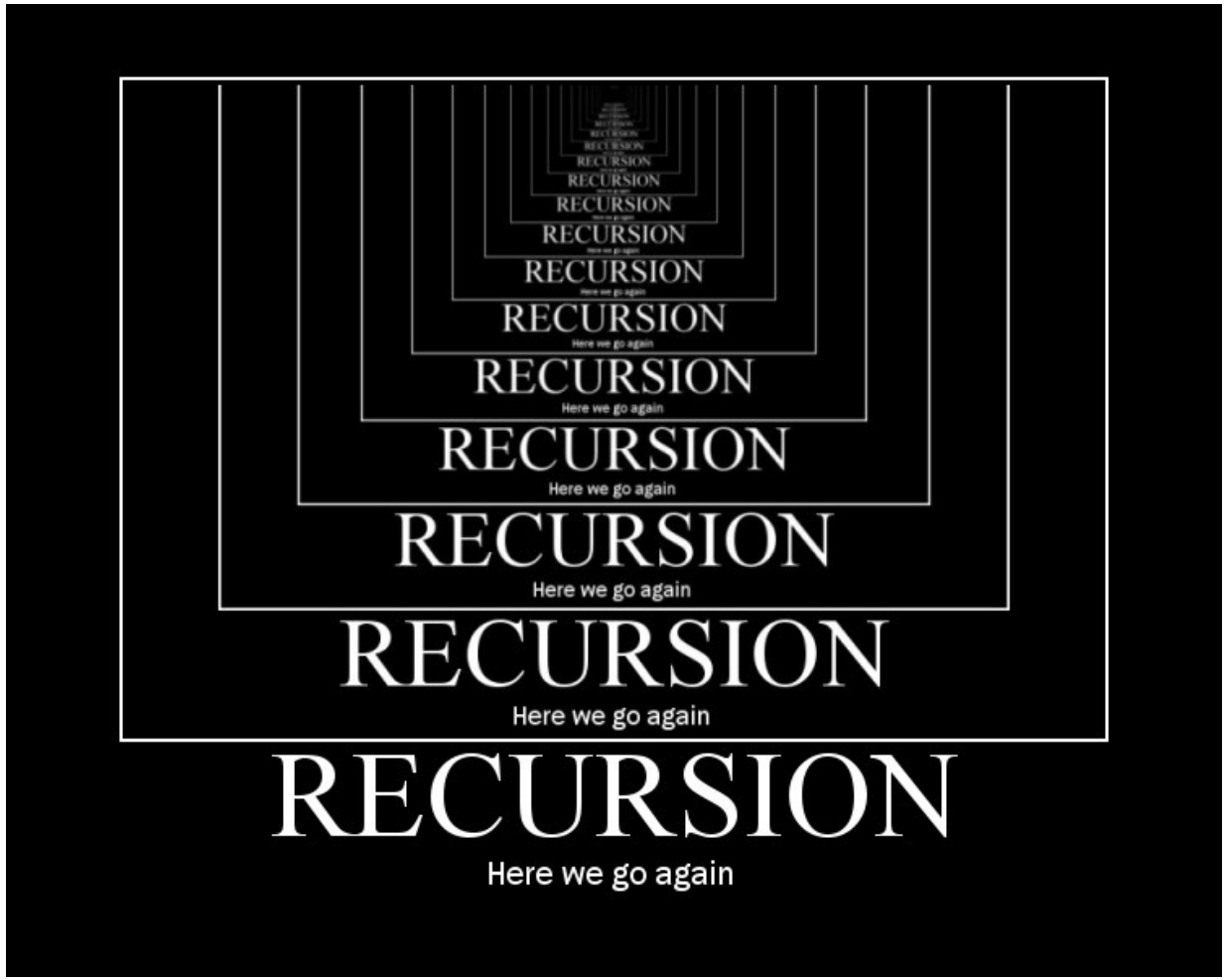
# Recursion

# What is recursion?

- Recursion is the process of defining a problem (or the solution to a problem) in terms of (a simpler version of) itself.



# Real World Analogy

- You are waiting at the front of a line and would like to know how many other people are in the line, but you can't get out of line and you can only see the person behind you. How do you find out how many people are in the line?

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

- Ask the person behind us how many people are in the line.

# Real World Analogy

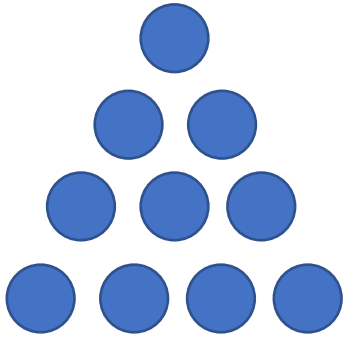
- Ask the person behind us how many people are in the line.

# Recursion in CS

- Recursive solutions involve a function that calls itself
- This **requires** a base case
  - The simplest solvable instance of the problem.
- Recursion isn't always a good or efficient solution
  - If the **depth of the recursion** (number of calls) is too large, the program crashes!
  - Overhead associated with function calls
- Can produce elegant/cleaner solutions to certain problems

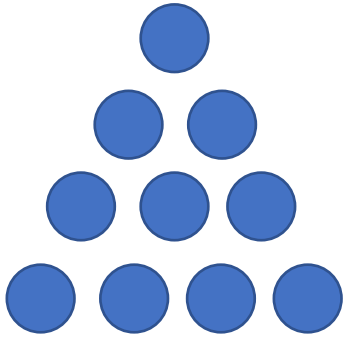
# Triangle Numbers

- The $n^{th}$ triangle number $T_n$ is: 1 + 2 + 3 … + n
- $T_0$ is 0 (empty sum)
- $T_4 = 1 + 2 + 3 + 4 = 10$

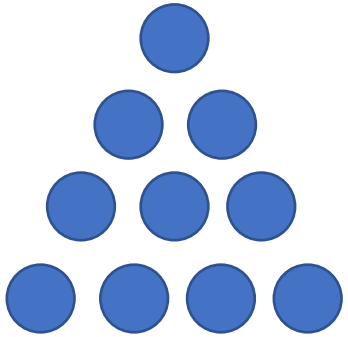


# Triangle Numbers

- The $n^{th}$ triangle number $T_n$ is: 1 + 2 + 3 … + n
- $T_0$ is 0 (empty sum)
- $T_4 = \underbrace{1 + 2 + 3}_{T_3} + 4 = 10$
- $T_4 = T_3 + 4$



# Triangle Numbers



- The $n^{th}$ triangle number $T_n$ is: 1 + 2 + 3 … + n
- $T_0$ is 0 (empty sum)
- $T_4 = \underbrace{1 + 2 + 3}_{T_3} + 4 = 10$
- $T_4 = T_3 + 4$

$$Tn = \begin{cases} 0 & \text{if } n = 0 \\ T_{n-1} + n & \text{otherwise} \end{cases}$$

# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

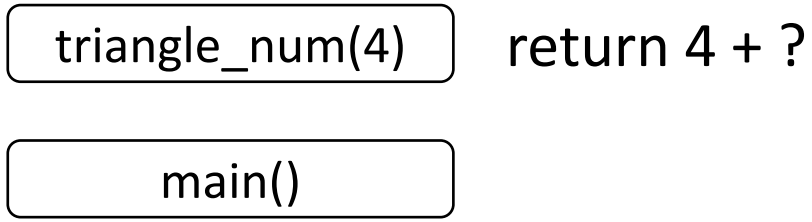
The Call Stack

# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

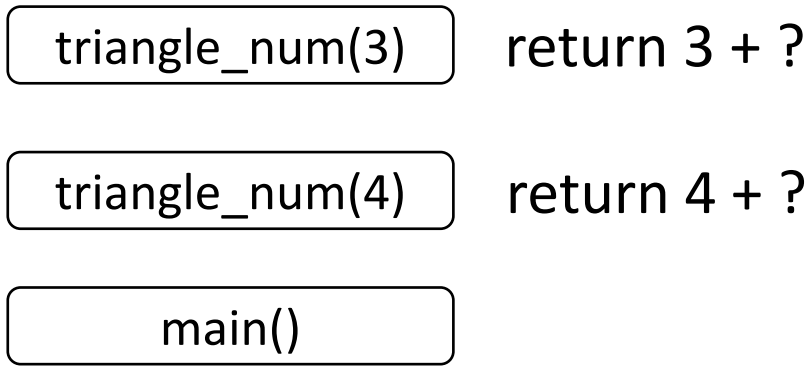


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

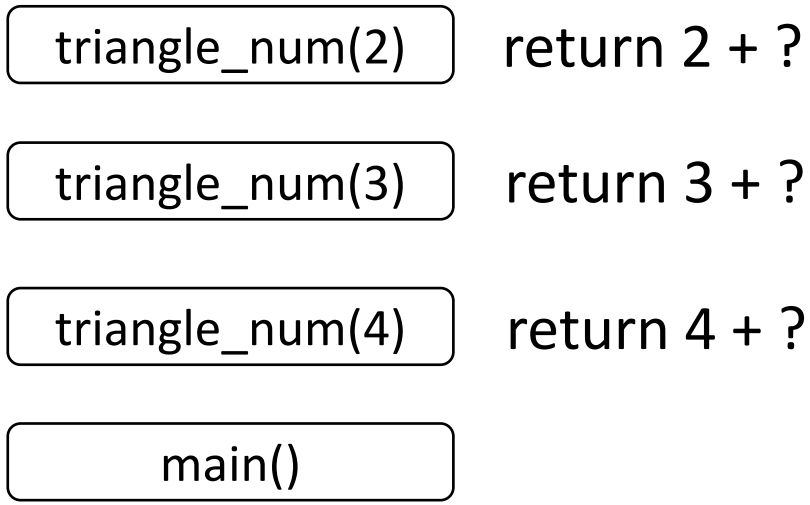


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

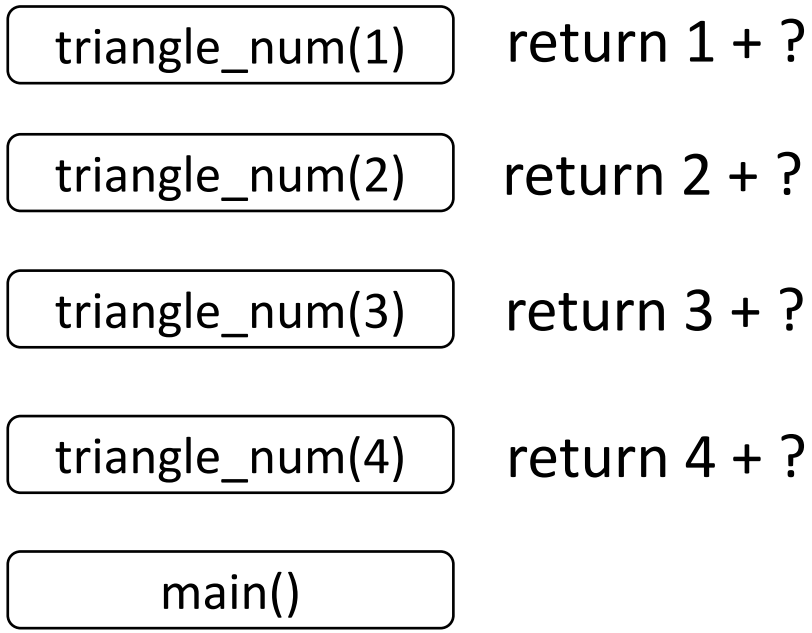


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

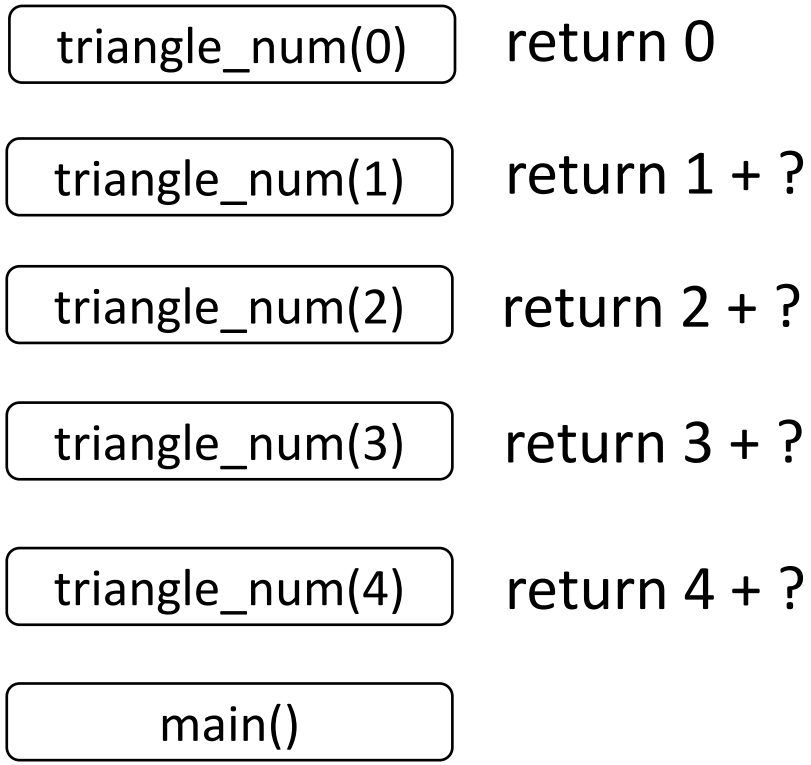


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

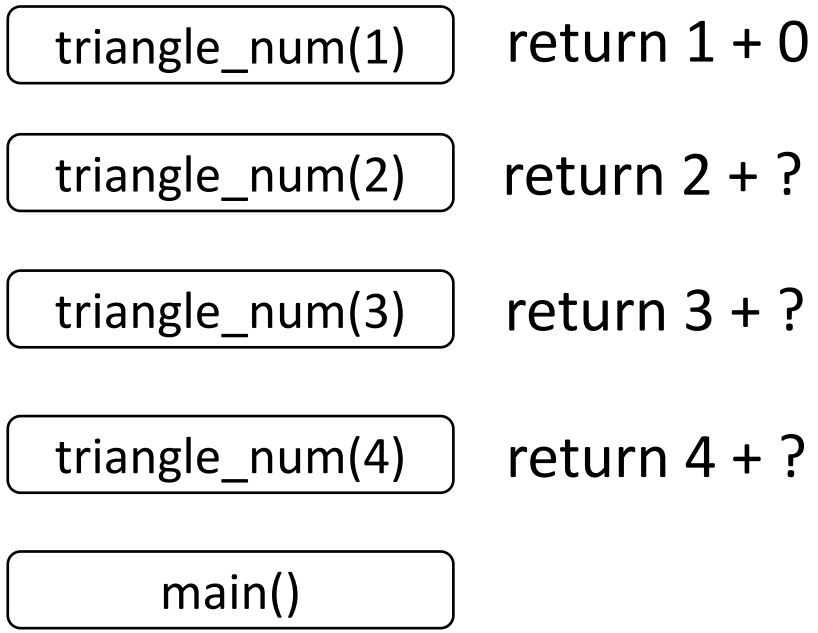


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

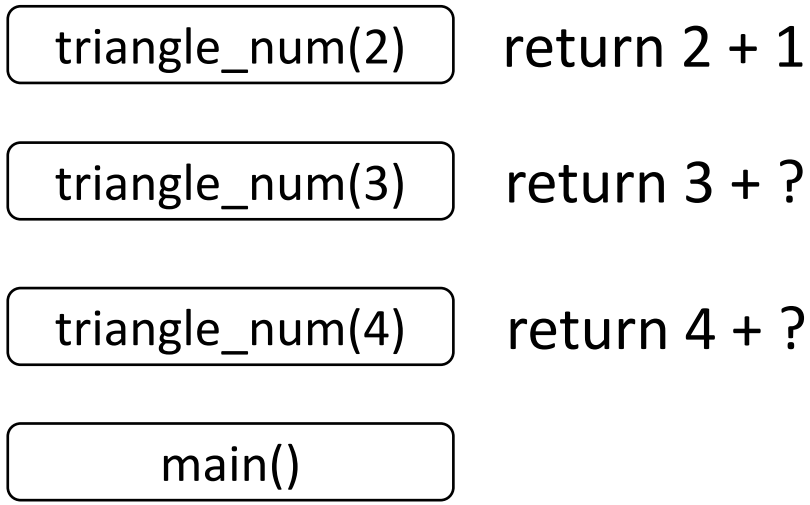


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

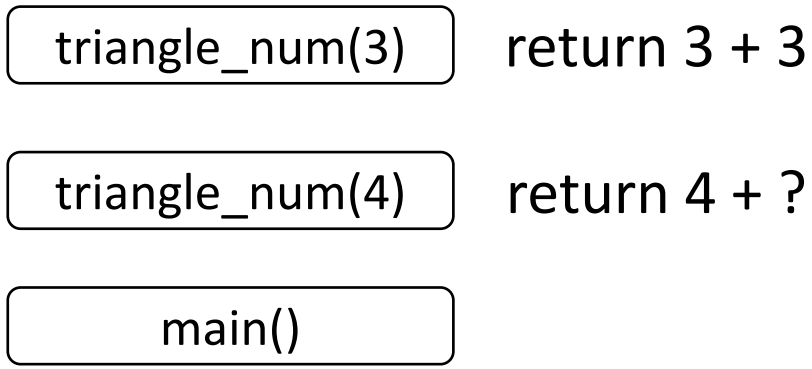


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

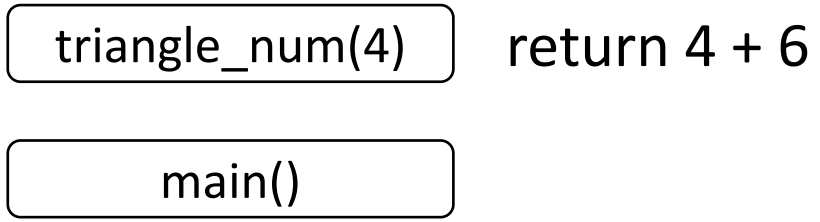


# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack



# Triangle Numbers

```
unsigned triangle_num(unsigned n) {
  if (n == 0) {
    // base case
    return 0;
  }
  else {
    return n + triangle_num(n - 1);
  }
}
```

n = 4

The Call Stack

main() 10

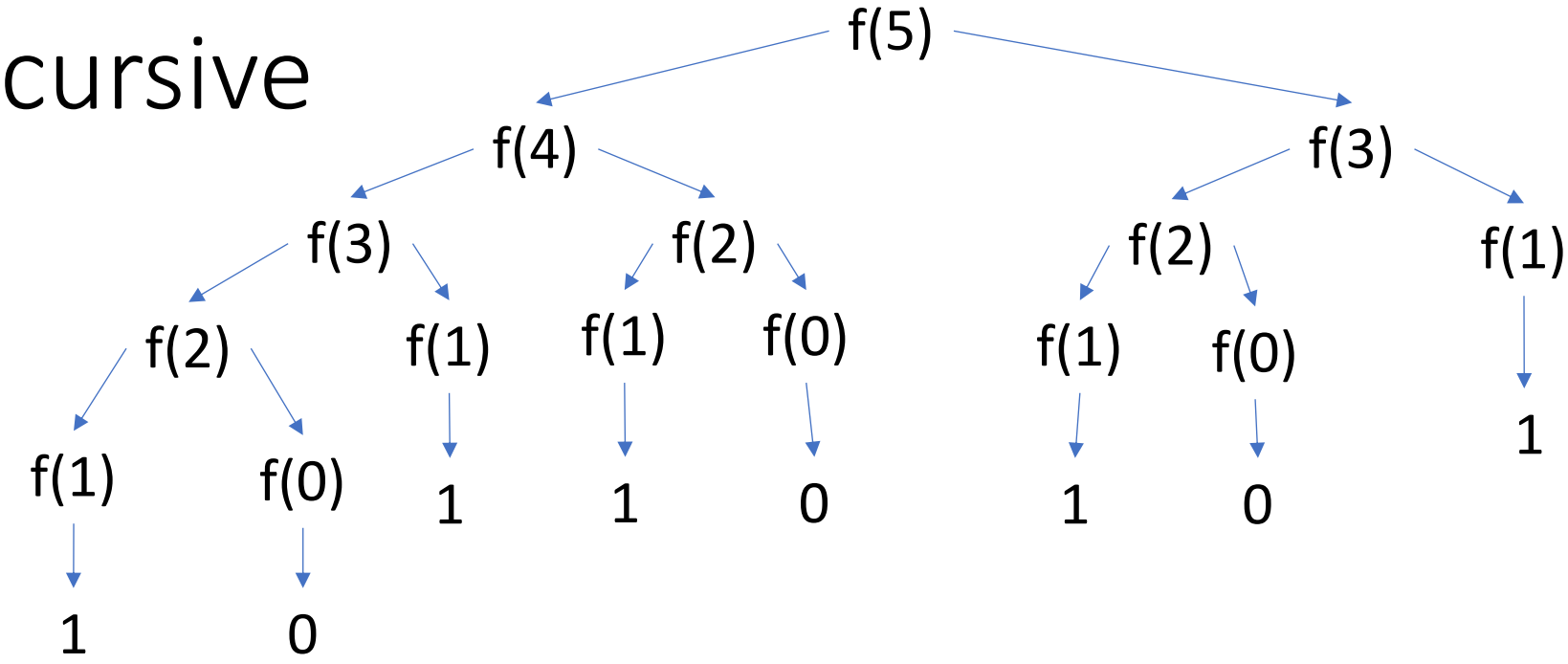
# Wait you said it isn’t always good…

# Fibonacci Recursive

```
long fibbo(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```

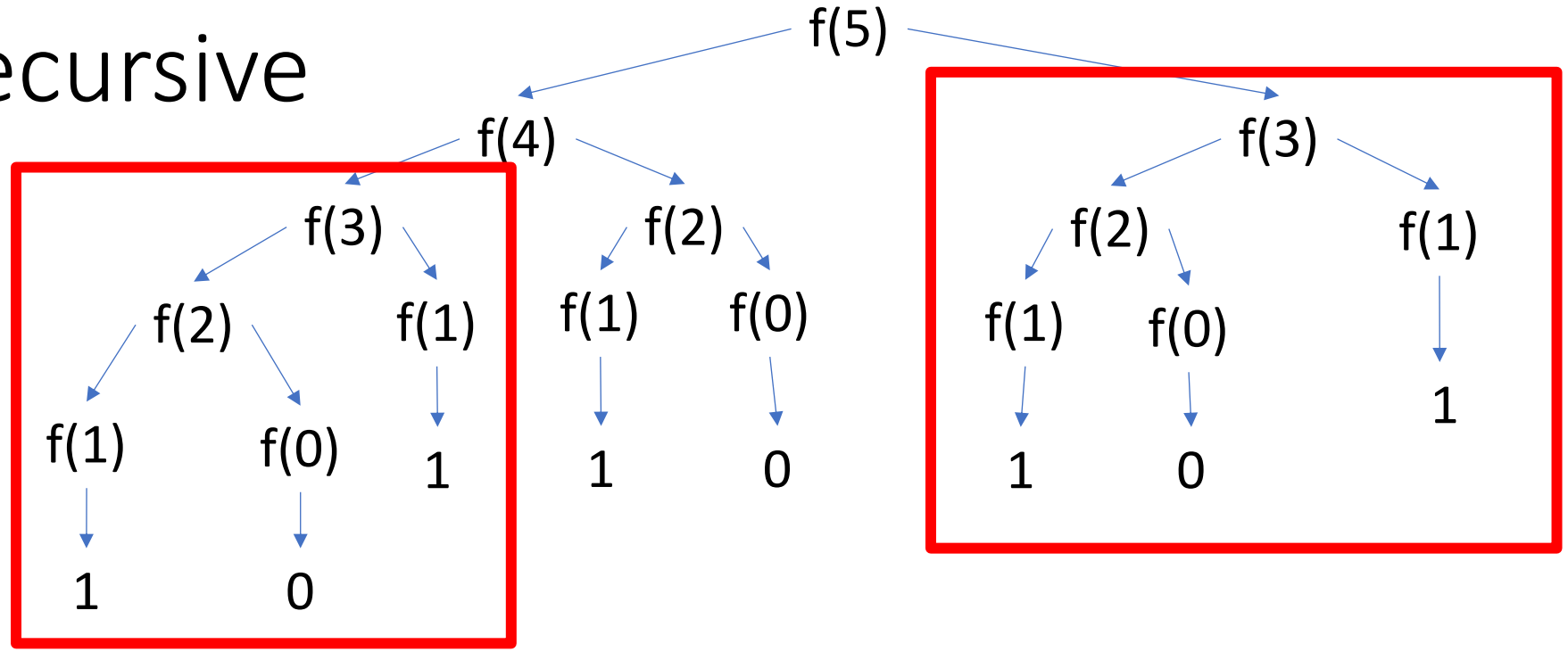
# Fibonacci Recursive

```
long fibbo(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```



# Fibonacci Recursive

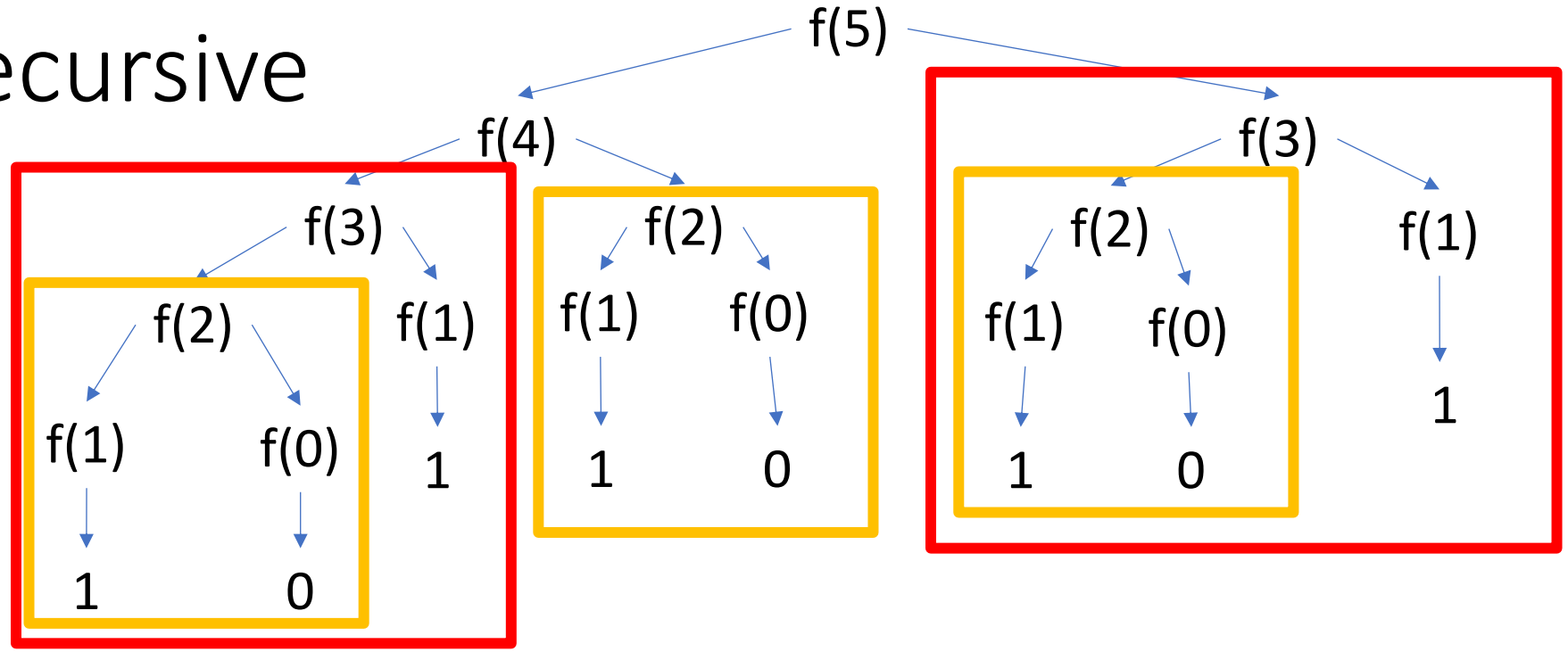
```
long fibbo(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```




Lots of duplicated work!

# Fibonacci Recursive

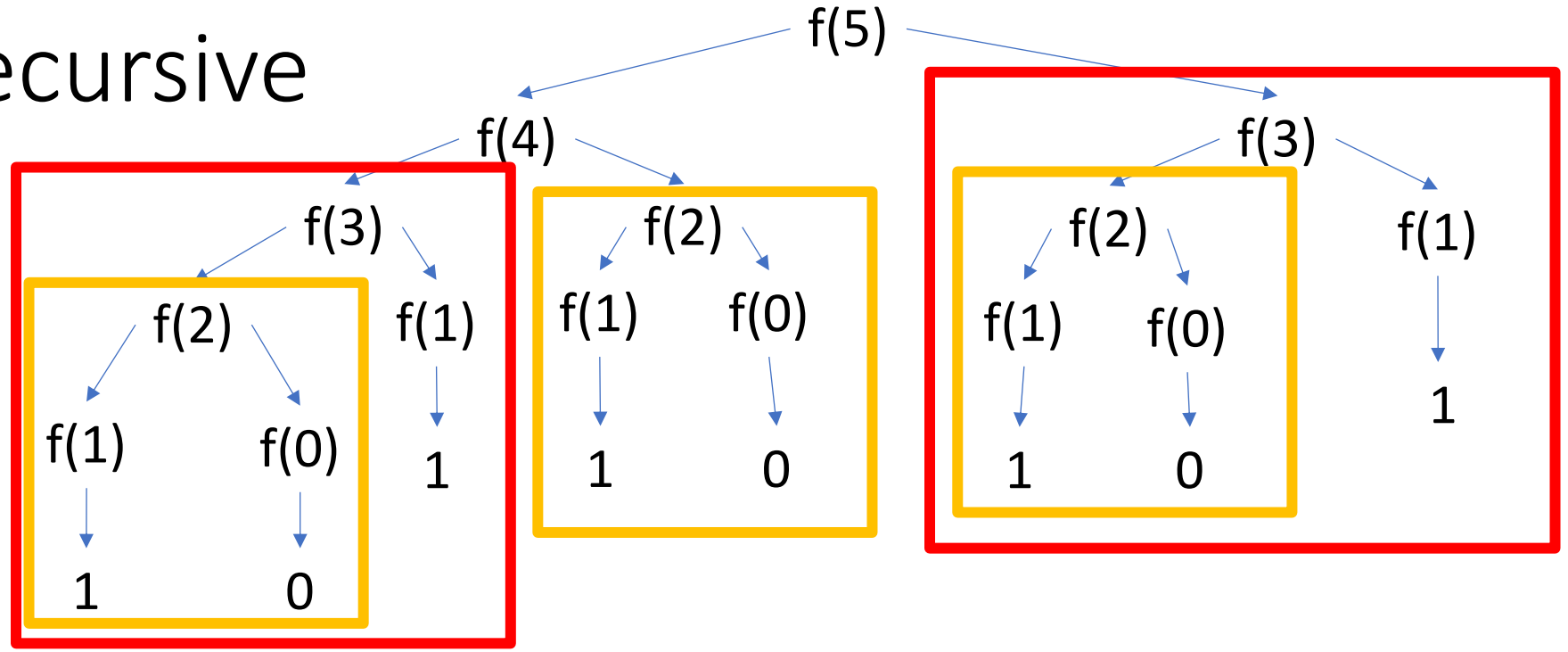
```
long fibbo(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```




Lots of duplicated work! LOTS!

# Fibonacci Recursive

```
long fibbo(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```




Lots of duplicated work! LOTS!

As the tree of calls gets bigger, this gets even worse!