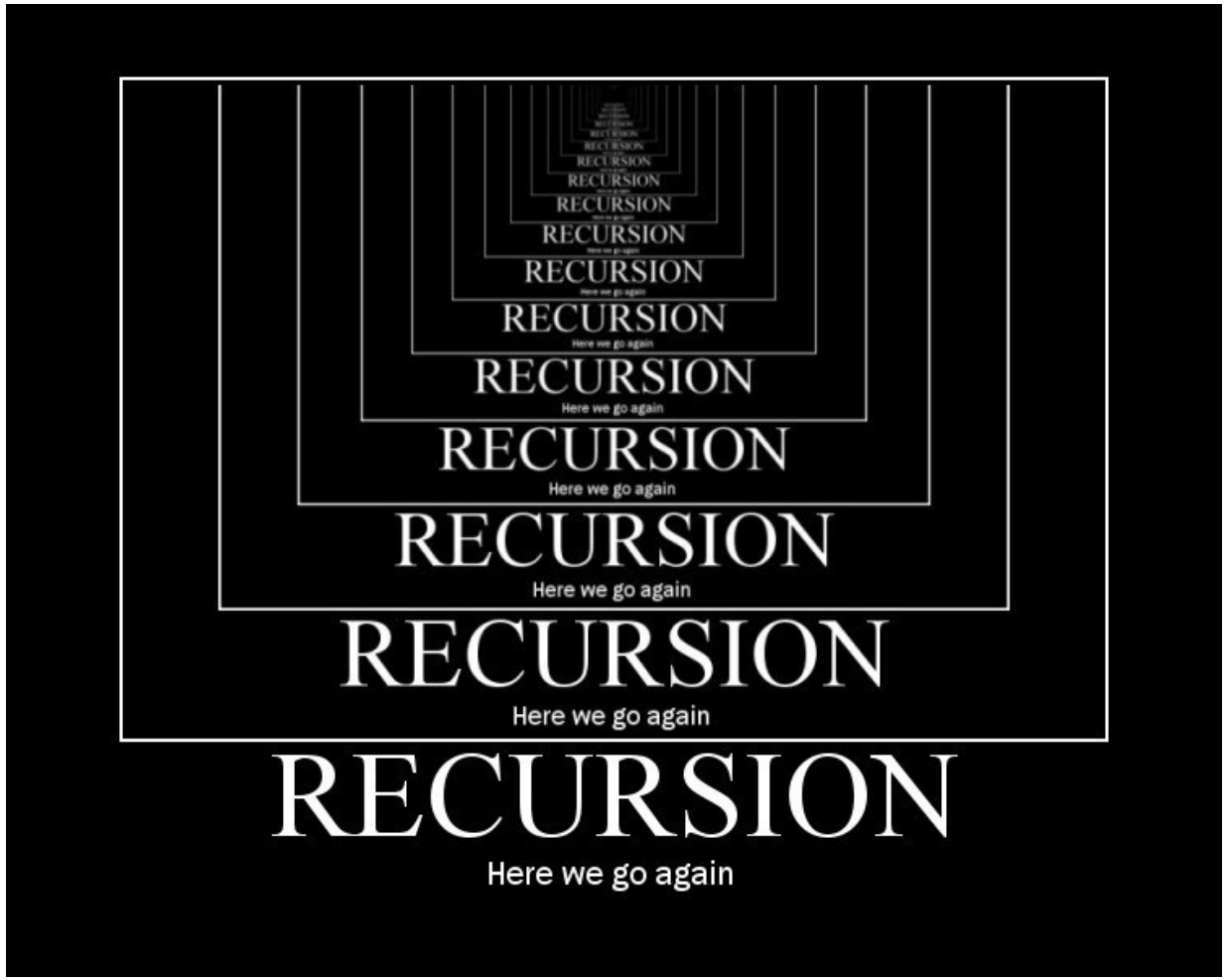
Recursion

What is recursion?

- Recursion is the process of defining a problem (or the solution to a problem) interms 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 theline?

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.

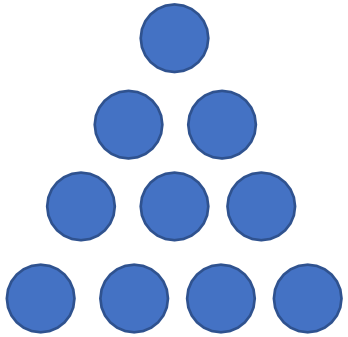
Recursion in CS

- 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

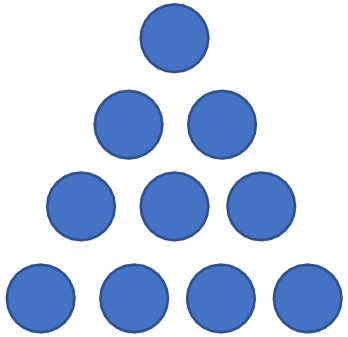
- The n^{th} triangle number T_n is: $1 + 2 + 3 \ldots + n$

- $T_4 = 1 + 2 + 3 + 4 = 10$



Triangle Numbers

- The n^{th} triangle number T_n is: $1 + 2 + 3 \ldots + n$



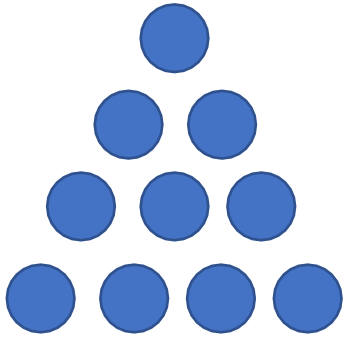
- $T_4 = 1 + 2 + 3 + 4 = 10$

T_3

- $T_4 = T_3 + 4$

Triangle Numbers

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



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

Triangle Numbers

```
unsigned triangle_iterative(unsigned n){
    unsigned triangle = 0;

    for (unsigned i = 1; i <= n; ++i) {
        triangle += i;
    }

    return triangle;
}
```

```
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

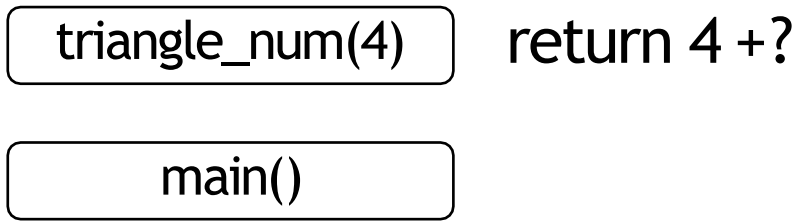
The Call Stack

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

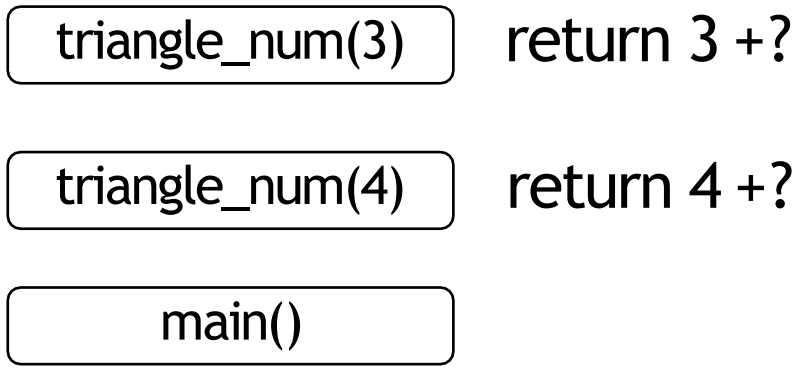


Triangle Numbers

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

n = 4

The Call Stack

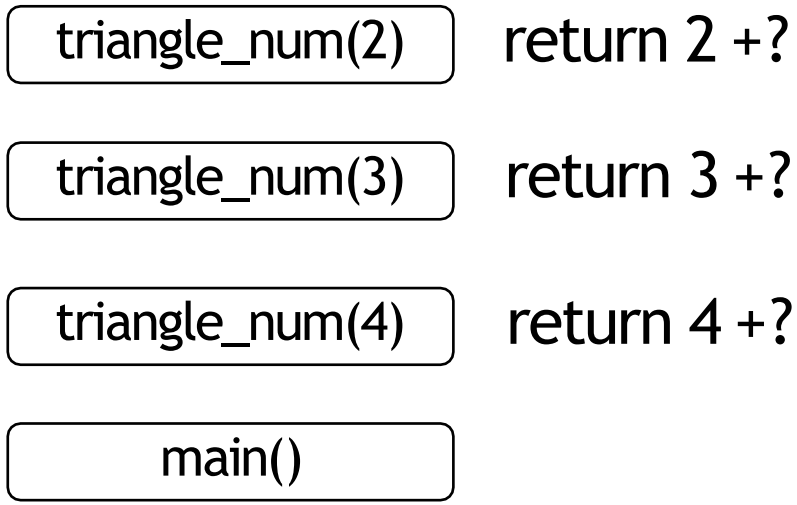


Triangle Numbers

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

n = 4

The Call Stack

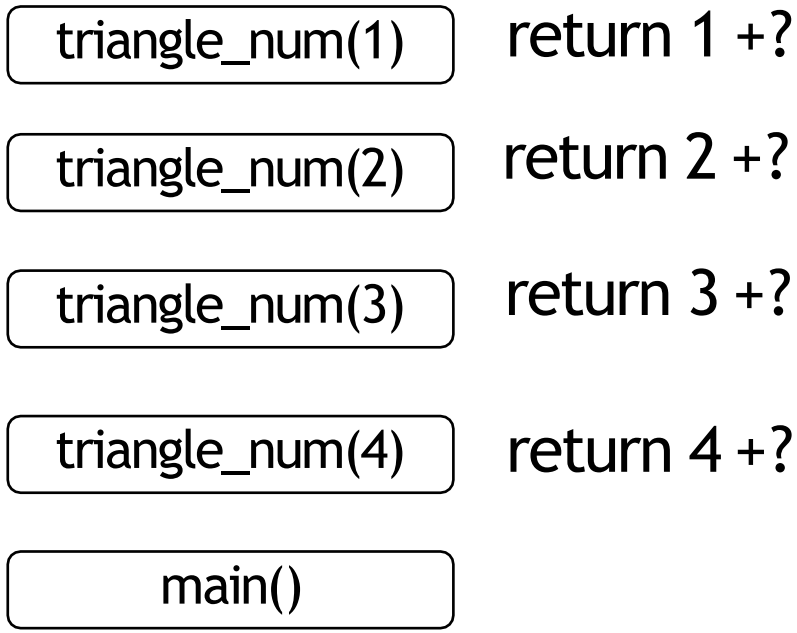


Triangle Numbers

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

n = 4

The Call Stack

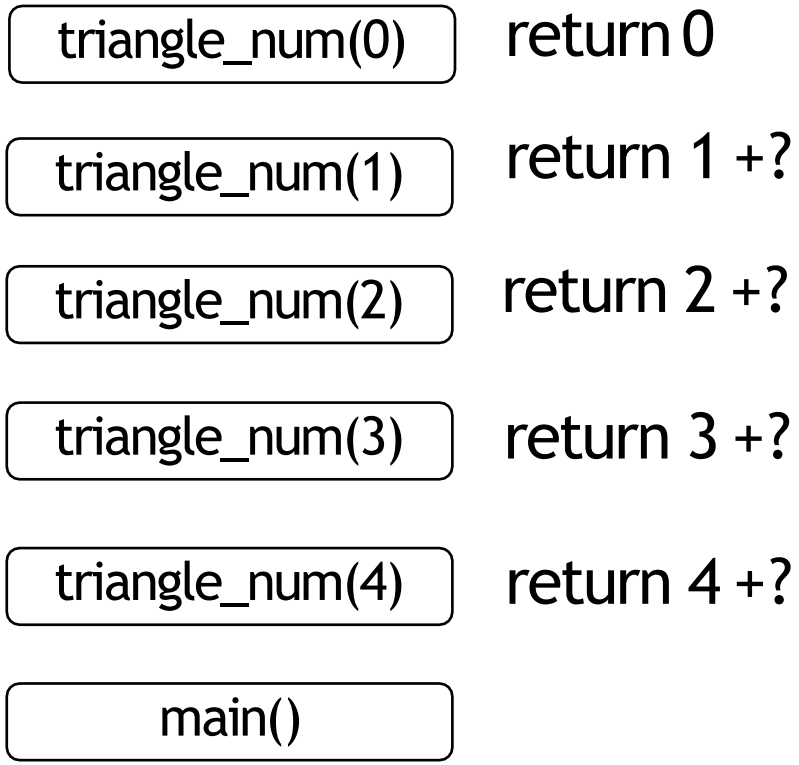


Triangle Numbers

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

n = 4

The Call Stack

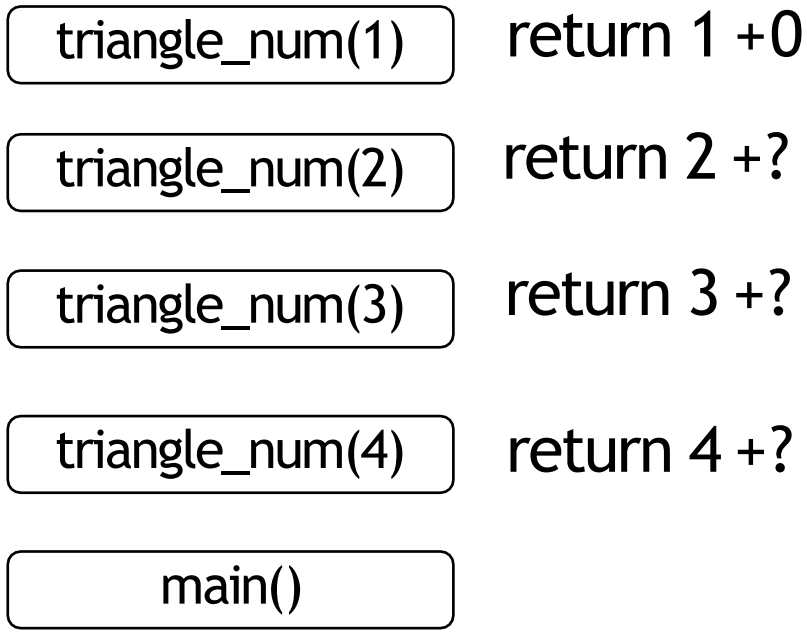


Triangle Numbers

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

n = 4

The Call Stack

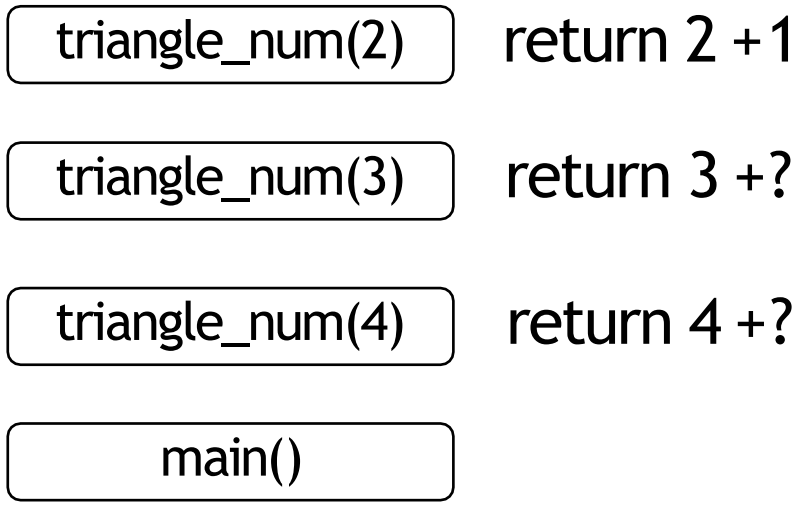


Triangle Numbers

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

n = 4

The Call Stack

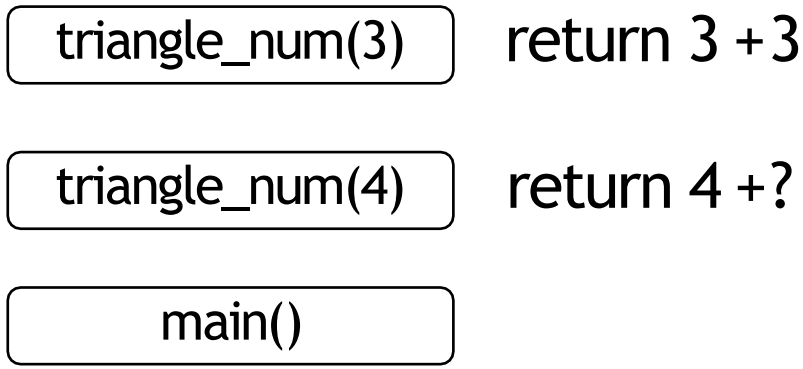


Triangle Numbers

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

n = 4

The Call Stack

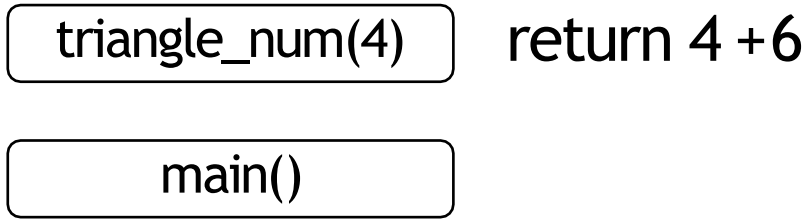


Triangle Numbers

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

n = 4

The Call Stack

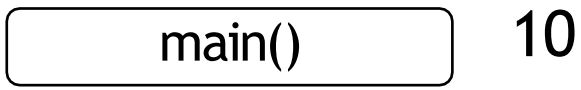


Triangle Numbers

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

n = 4

The Call Stack



10

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (          ) {
        // base case
     return (      );
    }
    else {
     return                                                    ;
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (size == 1) {
        // base case
     return [redacted];
    }
    else {
     return [redacted];
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (size == 1) {
        // base case
     return array[0];
    }
    else {
     return [redacted];
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (size == 1) {
        // base case
     return array[0];
    }
    else {
     return array[size - 1] + array_sum(array, size - 1);
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (size == 0) {
        // base case
     return a[redacted];
    }
    else {
     return[redacted];
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

```
int array_sum(int *array, size_t size) {
    if (size == 0) {
        // base case
     return 0;
    }
    else {
     return [redacted];
    }
}
```

Array Sum

Iterative Approach

```
int array_sum(int *array, size_t size) {
    int sum = 0;

    for(int i = 0; i < size; ++i){
        sum += array[i];
    }

    return sum;
}
```

Recursive Approach

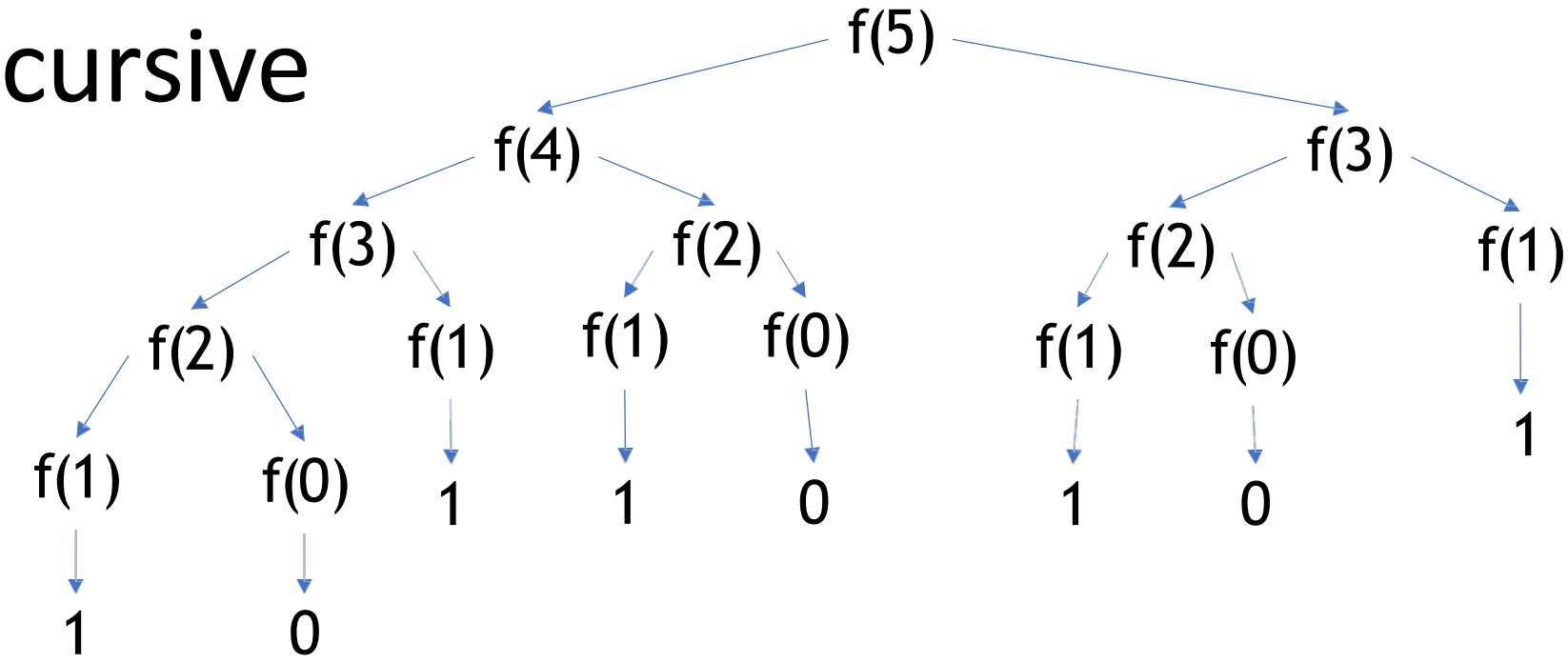
```
int array_sum(int *array, size_t size) {
    if (size == 0) {
        // base case
     return 0;
    }
    else {
     return array[size - 1] + array_sum(array, size - 1);
    }
}
```

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);
}
```

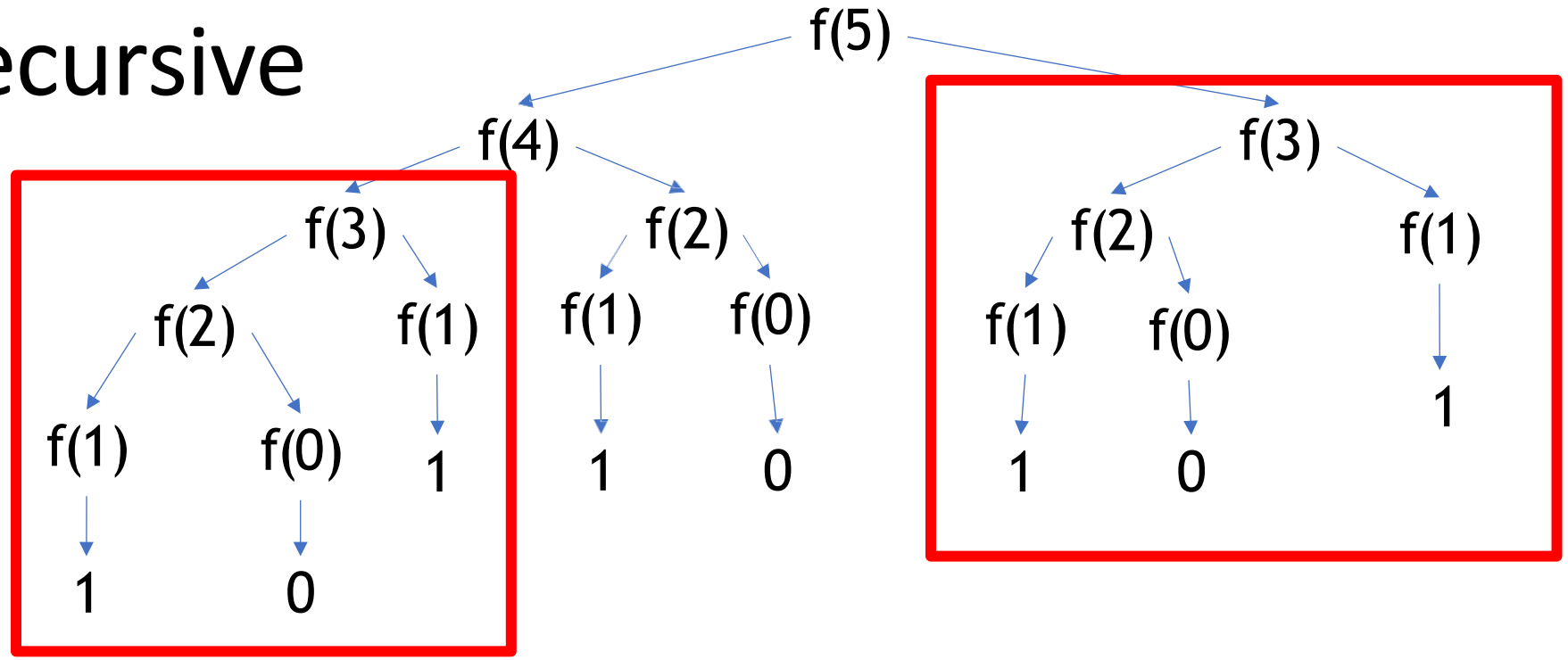
Fibonacci Recursive

```
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

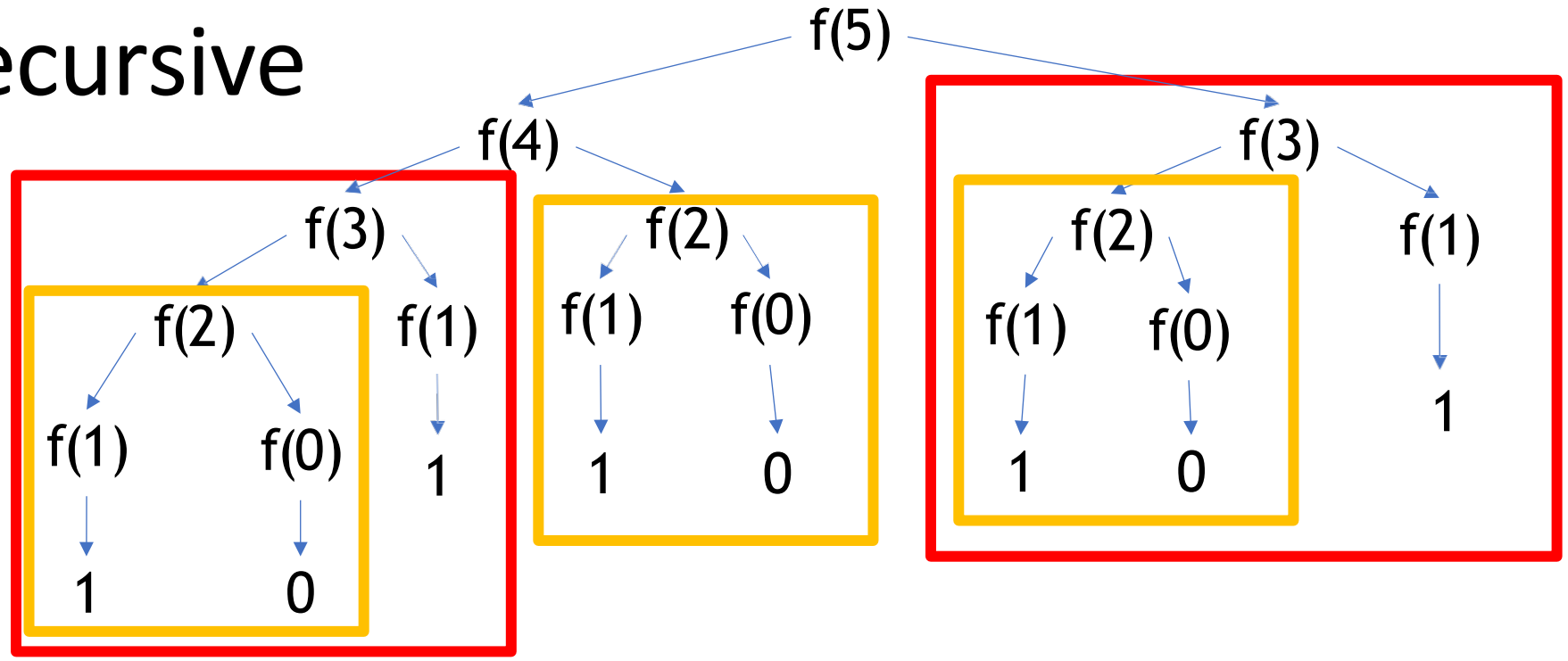
```
long fibbo(int n){
  if (n == 0)
  return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```



Lots of duplicated work!

Fibonacci Recursive

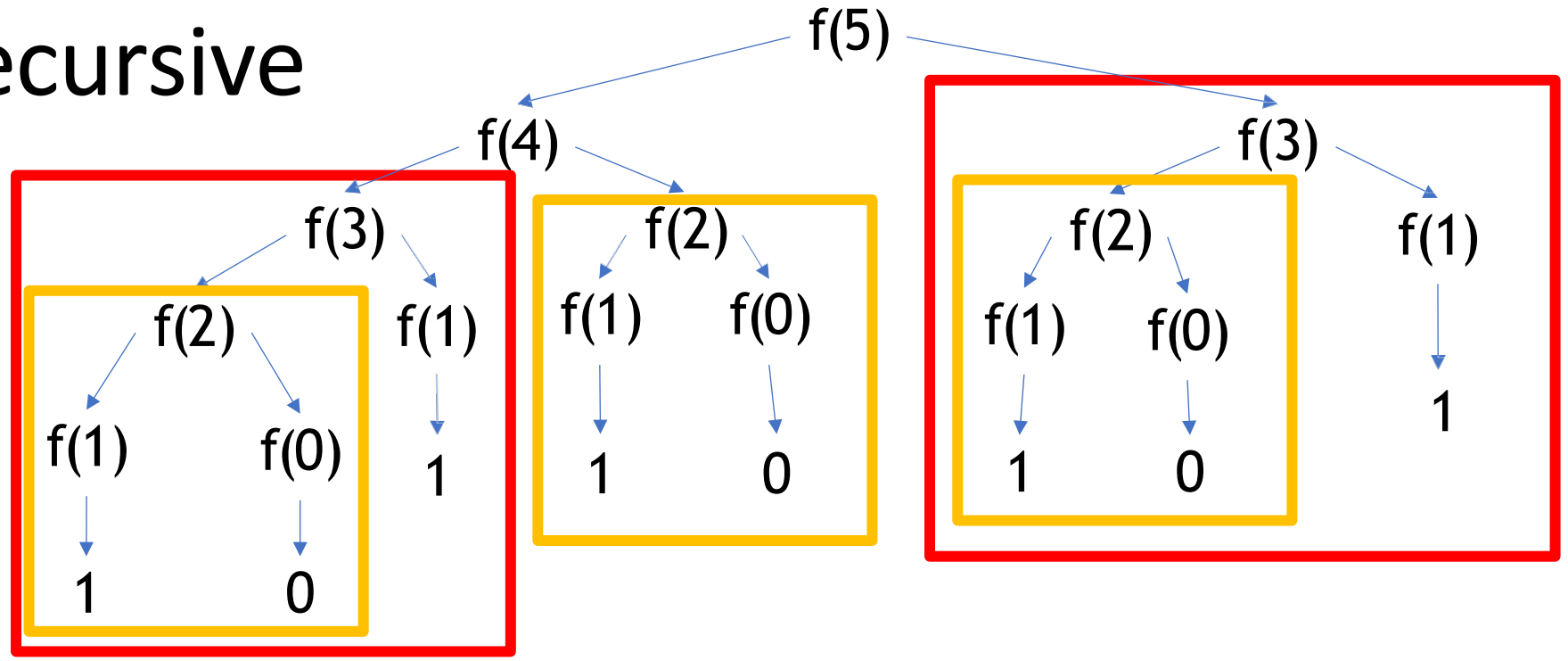
```
long fibbo(int n){
  if (n == 0)
  return 0;
  else if (n == 1)
    return 1;
  else
    return fibbo(n - 1) +
           fibbo(n - 2);
}
```



Lots of duplicatedwork!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!

Fibonacci

Iterative Approach

```
unsigned long long fibbo_iterative(int n) {
    unsigned long long current = 1;
    unsigned long long previous = 0;

    for (int i = 0; i < n; ++i) {
        unsigned long long temp = current;
        current += previous;
        previous = temp;
    }
    return previous;
}
```

Recursive Approach

```
unsigned long long fibbo_recursive(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        //Add the previous two values in the sequence together
        return fibbo_recursive(n - 1) + fibbo_recursive(n - 2);
}
```

Fibonacci

Iterative Approach

- Iterative Fibonacci is slightly more complex than the recursive version
- performs consistently.

Recursive Approach

- Not ideal
- Larger numbers of n cause the process to take an unreasonable amount of time.

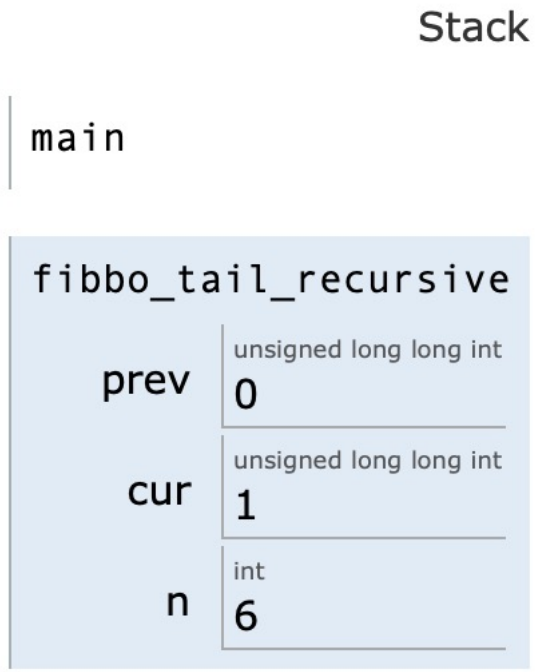
Tail-Recursive Fibonacci Recursive

```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```

- Very similar to the iterative version.
- What makes this "tail" recursive
  - recursive call is the last statement in the function
  - all the data necessary to perform the calculation for next recursive call is only passed as parameters
- This version does not suffer the problem of performing repeated and redundant calculations

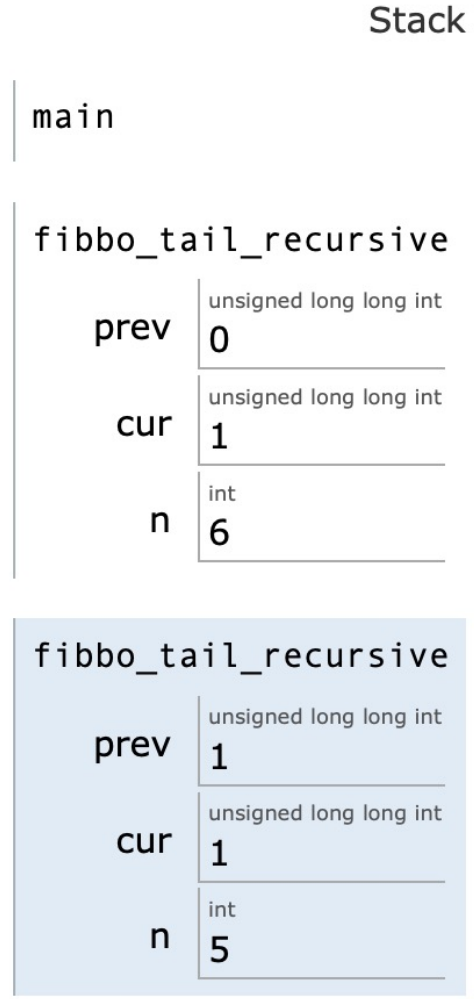
Tail-Recursive Fibonacci Recursive

```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```



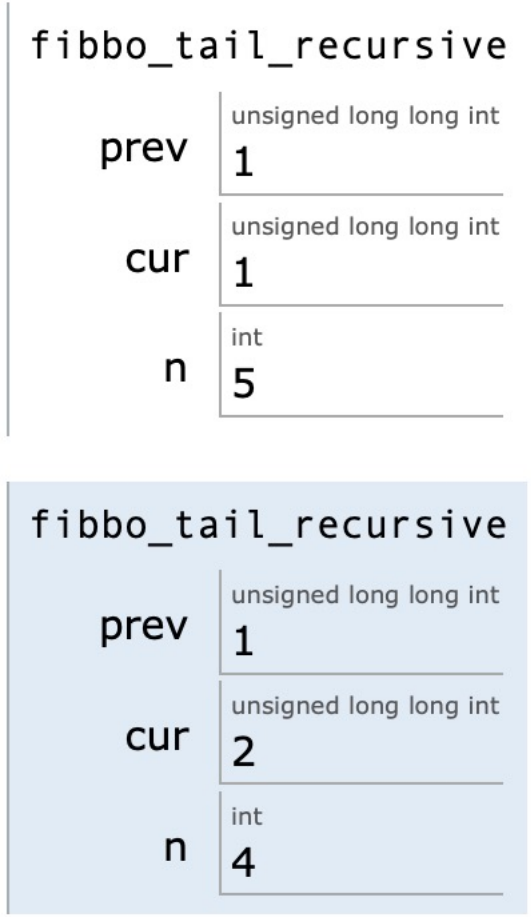
Tail-Recursive Fibonacci Recursive

```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```



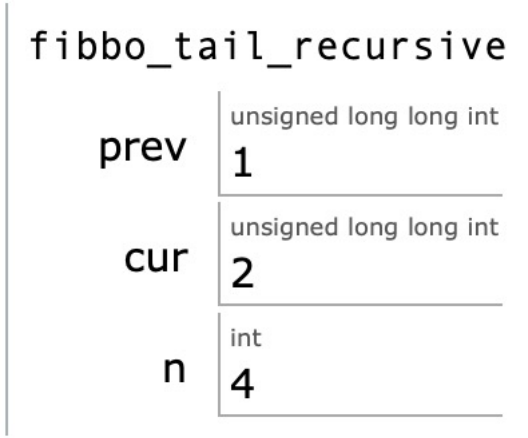
Tail-Recursive Fibonacci Recursive

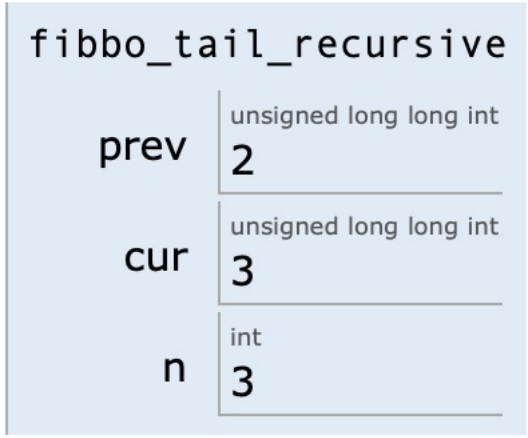
```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```



Tail-Recursive Fibonacci Recursive

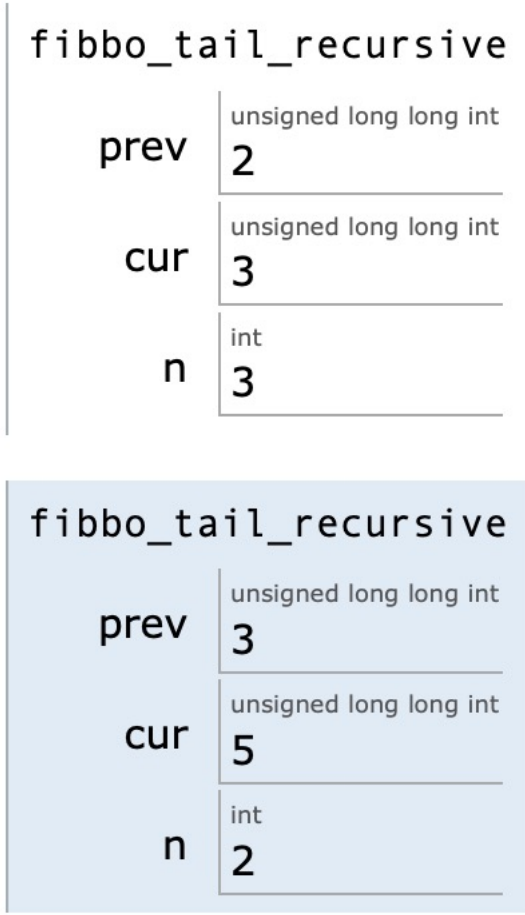
```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```





Tail-Recursive Fibonacci Recursive

```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```



Tail-Recursive Fibonacci Recursive

```
unsigned long long fibbo_tail_recursive(unsigned long
  long prev, unsigned long long cur, int n) {
    if (n == 0)
        return prev;
    else if (n == 1)
        return cur;
    else {
        return fibbo_tail_recursive(cur, cur + prev, n-1);
    }
}
```