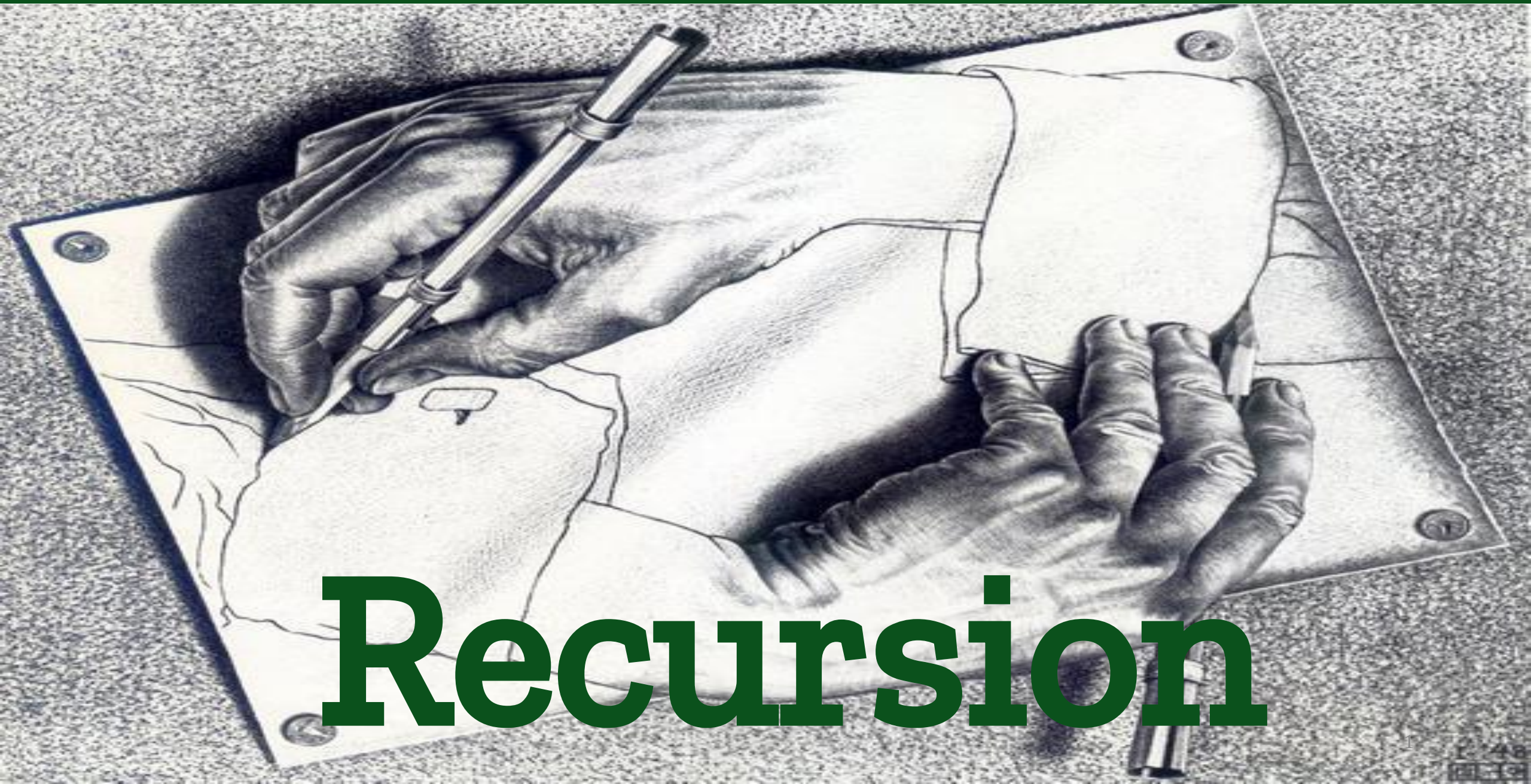


Recursion

Divide and Conquer

A divide-and-conquer algorithm:

- breaks down a problem into ~~two~~ *one* or more **sub-problems** of the same or related type,
- until these become **simple** enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.

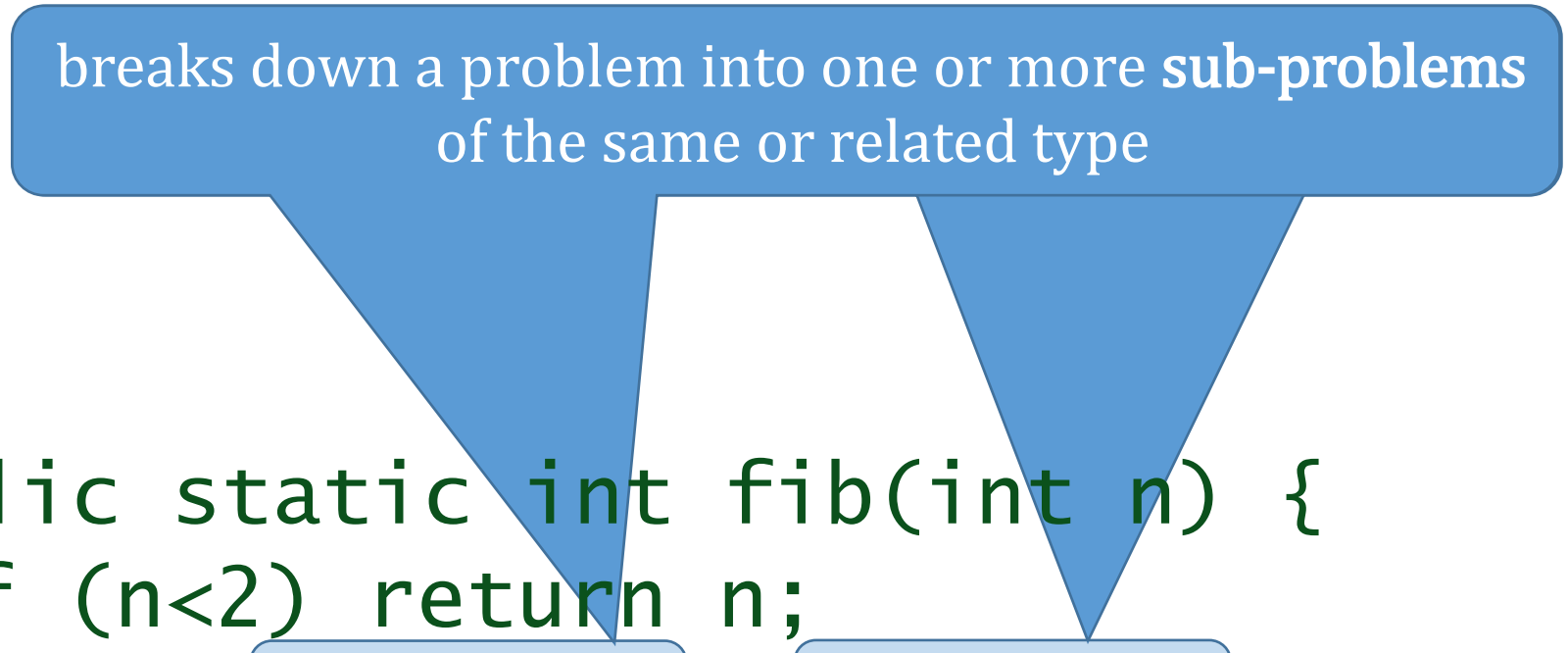
Wikipedia Divide-and-conquer algorithm

Fibonacci Numbers

- Definition: fib(0)=0, fib(1)=1, fib(n) = fib(n-1) + fib(n-2)
- Result: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Standard mathematical example for divide and conquer

```
public static int fib(int n) {
  if (n<2) return n;
  return fib(n-1) + fib(n-2);
}
```

Fibonacci Numbers




```
public static int fib(int n) {
  if (n<2) return n;
  return fib(n-1) + fib(n-2);
}
```

Fibonacci Numbers

until these become **simple** enough to be solved directly

```
public static int fib(int n) {
  if (n<2) return n;
  return fib(n-1) + fib(n-2);
}
```

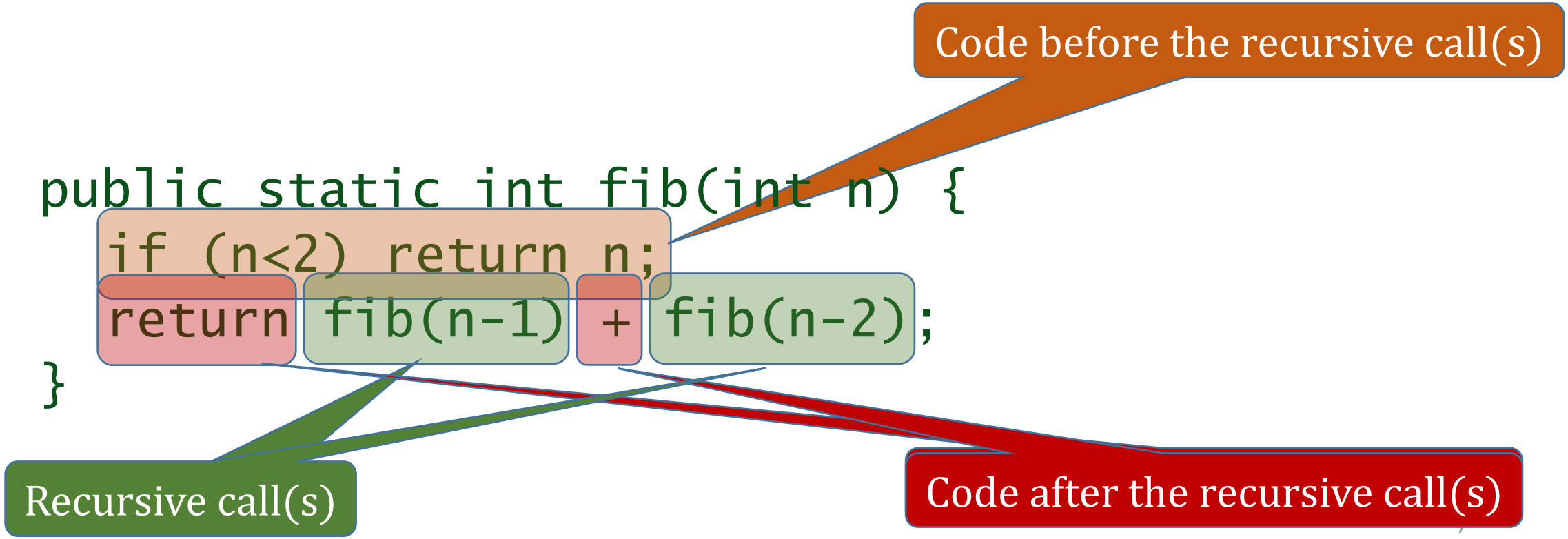
Fibonacci Numbers

The solutions to the sub-problems are then combined to give a solution to the original problem.

```
public static int fib(int n) {
  if (n<2) return n;
  return fib(n-1) + fib(n-2);
}
```

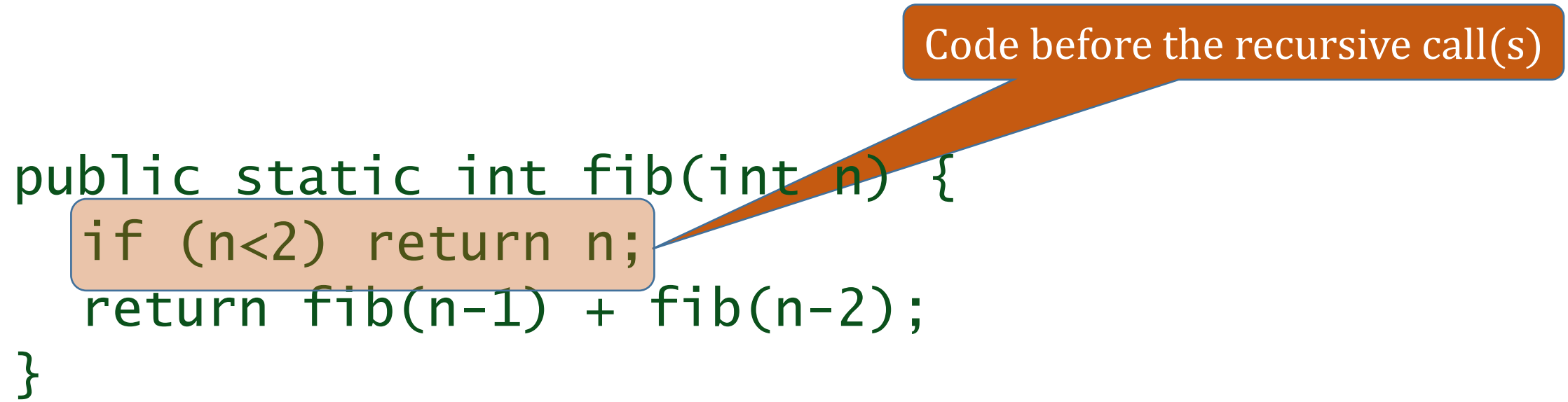
Recursion

- A *recursive method* is a method that invokes itself
- Divide recursive methods into three parts:



Avoid Infinite Recursion

- Handle the simple case ***before*** the recursive call!

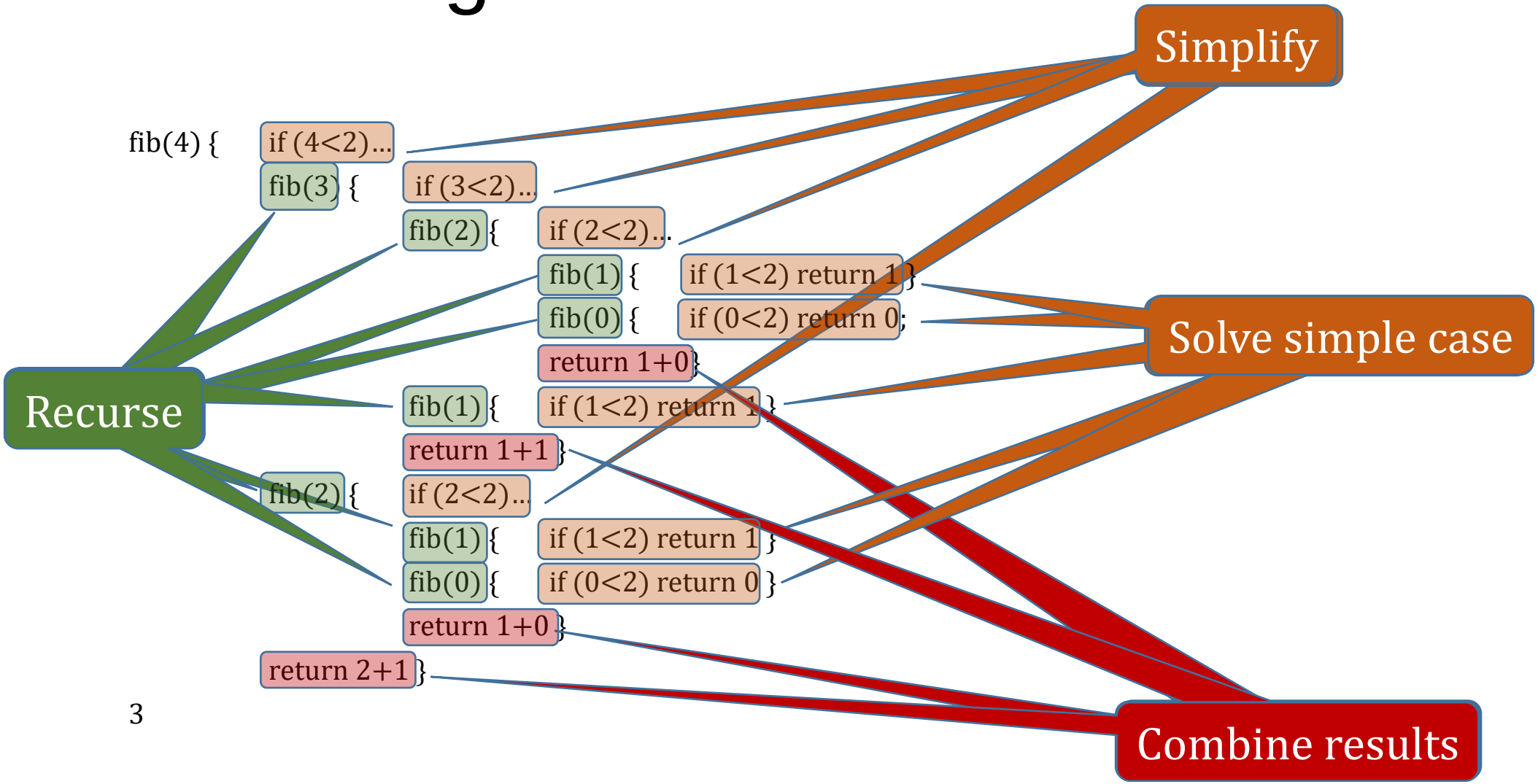


Recursive Call Stack

Lev 0	Lev 1	Lev 2	Lev 3	Lev 4
fib(4) {	if (4<2)…			
	fib(3) {	if (3<2)…		
		fib(2) {	if (2<2)…	
			fib(1) {	if (1<2) return 1 }
			fib(0) {	if (0<2) return 0 }
			return 1+0}	
		fib(1) {	if (1<2) return 1 }	
		return 1+1 }		
	fib(2) {	if (2<2)…		
		fib(1) {	if (1<2) return 1 }	
		fib(0) {	if (0<2) return 0 }	
		return 1+0		
	return 2+1 }			
3				

Call Stack			
fib(4) 1			
fib(3) 1	fib(4) 2		
fib(2) 1	fib(3) 2	fib(4) 2	
fib(1) 1	fib(2) 2	fib(3) 2	fib(4) 2
fib(0) 1	fib(2) 2	fib(3) 2	fib(4) 2
fib(2) 2	fib(3) 2	fib(4) 2	
fib(1) 1	fib(3) 2	fib(4) 2	
fib(3) 2	fib(4) 2		
fib(2) 1	fib(4) 2		
fib(1) 1	fib(2) 2	fib(4) 2	
fib(0) 1	fib(2) 2	fib(4) 2	
fib(2) 2	fib(4) 2		
fib(4) 2			

Unfolding Recursion



Efficiency vs. Elegance

- Recursive solutions are often elegant, but expensive
  - Lots of method invocation (expensive)
  - Big call stack required
  - May need to re-calculate values
    - fib(2) called twice in fib(4)
    - fib(1) called three times in fib(4)
- Compiler often optimizes recursion
  - Replace recursive calls with a loop
  - Allows elegance of recursion without performance penalty!

Think Recursion!

- Can I divide the problem into simpler problems?
- Can the simplest form of the problem be solved easily?
- If so… recursion can simplify solving the problem!