"It isn't what you do, but how you do it." John Wooden

Recursion Part 2

Fibonacci sequence's
exponential growth
and a tail-recursive solution

But when is recursion not the answer?

- The Fibonacci sequence can be defined as follows:

  $F_0 = 0$

  $F_1 = 1$

  $F_n = F_{n-1} + F_{n-2}$ for $n >= 2$

- The sequence looks like:

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

  F_0 F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8

Fibonacci sequence in nature

The Nautilus Shell Spiral

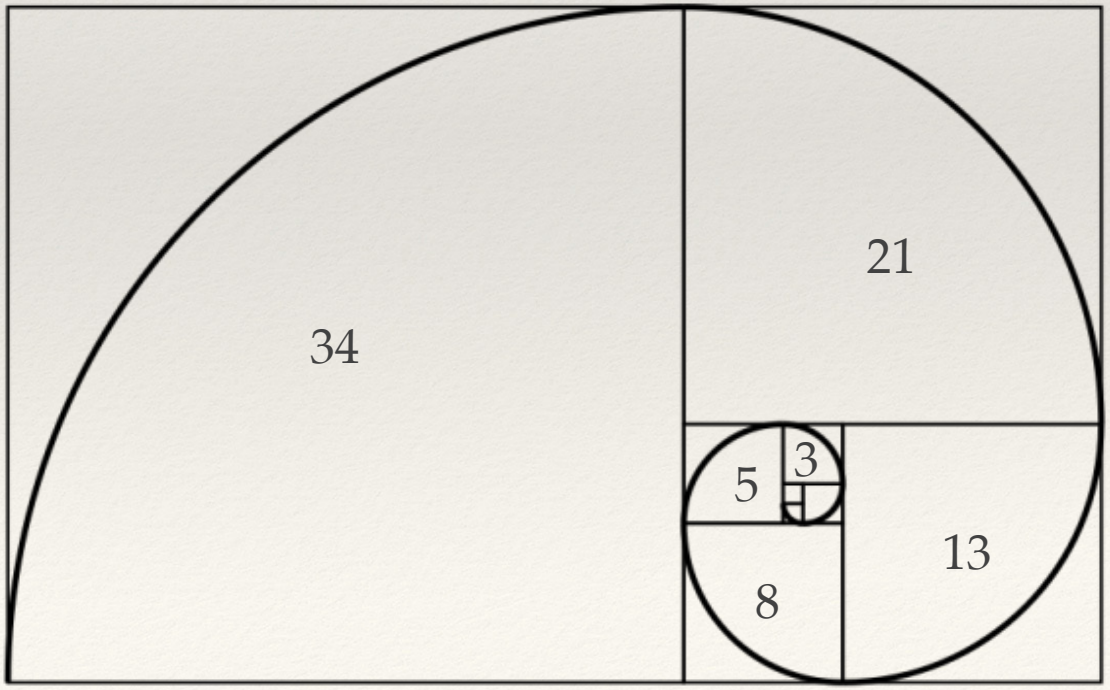
Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, 34, …

F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8 F_9

side-length

of a bounding square of the spiral curve is the side-lengths of the previous two squares added together



number = side-length of square
side-length of inner most squares = 1

Recursive Fibonacci

Fibonacci sequence definition

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n >= 2$

Intuitive Implementation

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

base case

recursive step

Recursive Fibonacci

Fibonacci sequence definition

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n >= 2$

Intuitive Implementation

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

Recursive Fibonacci

Fibonacci sequence definition

$F_0 = 0$

$F_1 = 1$

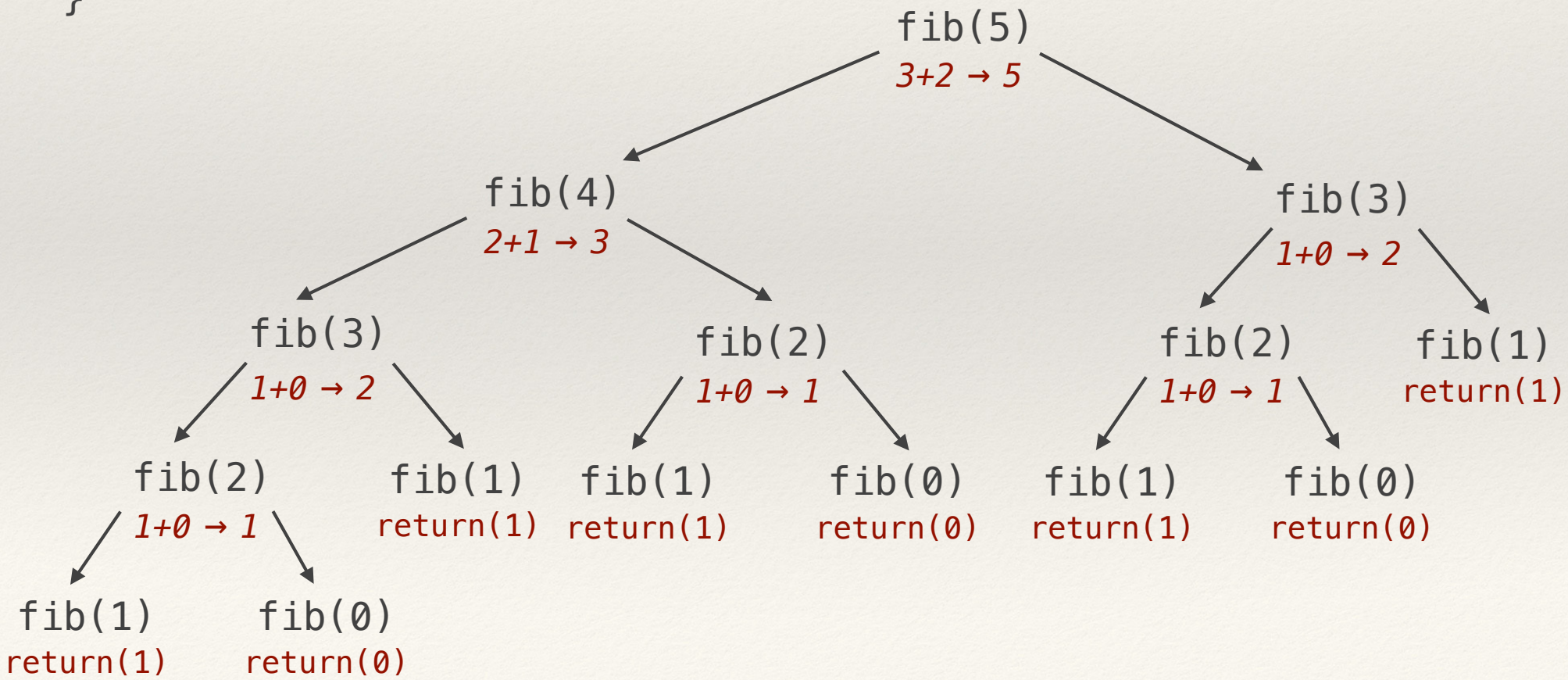
$F_n = F_{n-1} + F_{n-2}$ for $n >= 2$

Intuitive Implementation

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

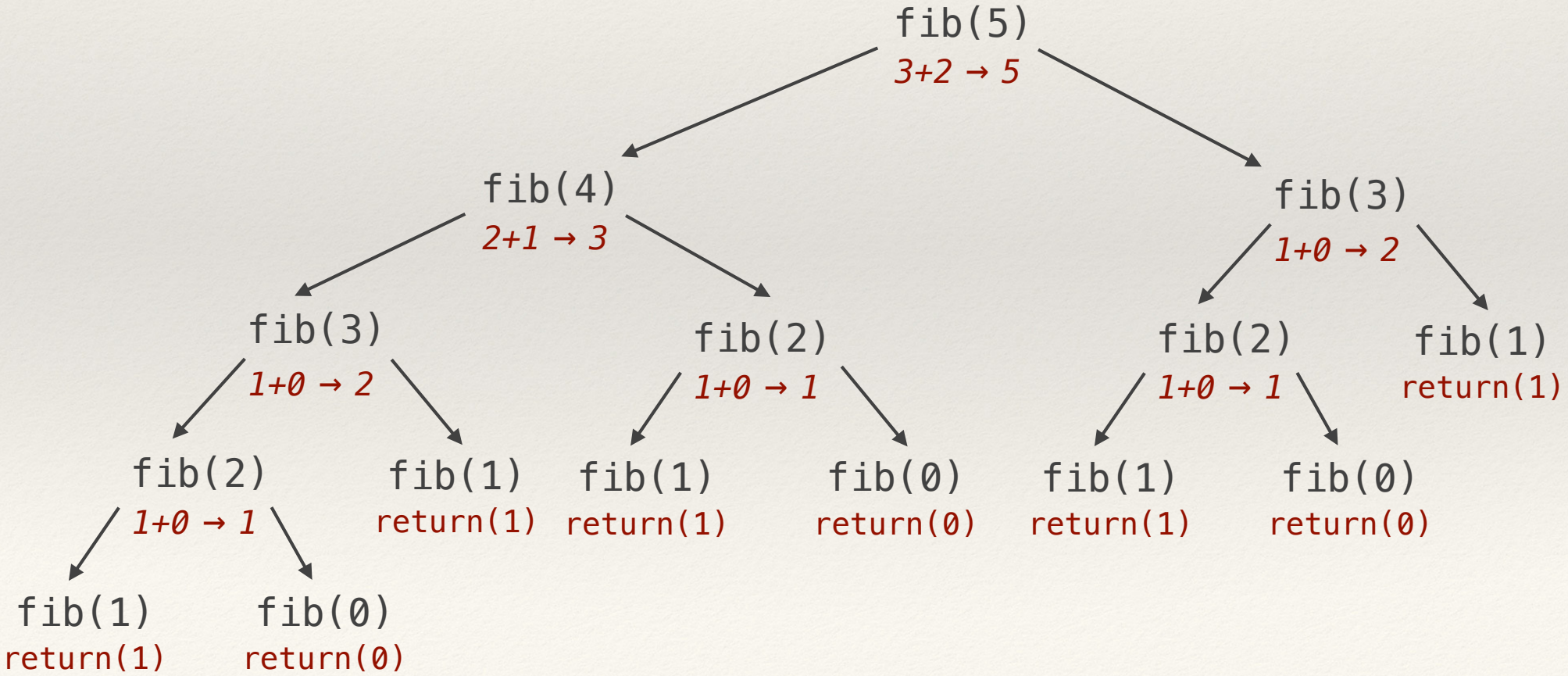
```
int main ( int argc, char *argv[] )
{
   int i, n;
   n = atoi ( argv[1] );
   for ( i=0; i<=n; i++ ) {
      printf ( "%d ", fib(i) );
   }
   printf ( "\n" );
   return (0);
}
```

```
$ ./fib 8
0 1 1 2 3 5 8 13 21
```



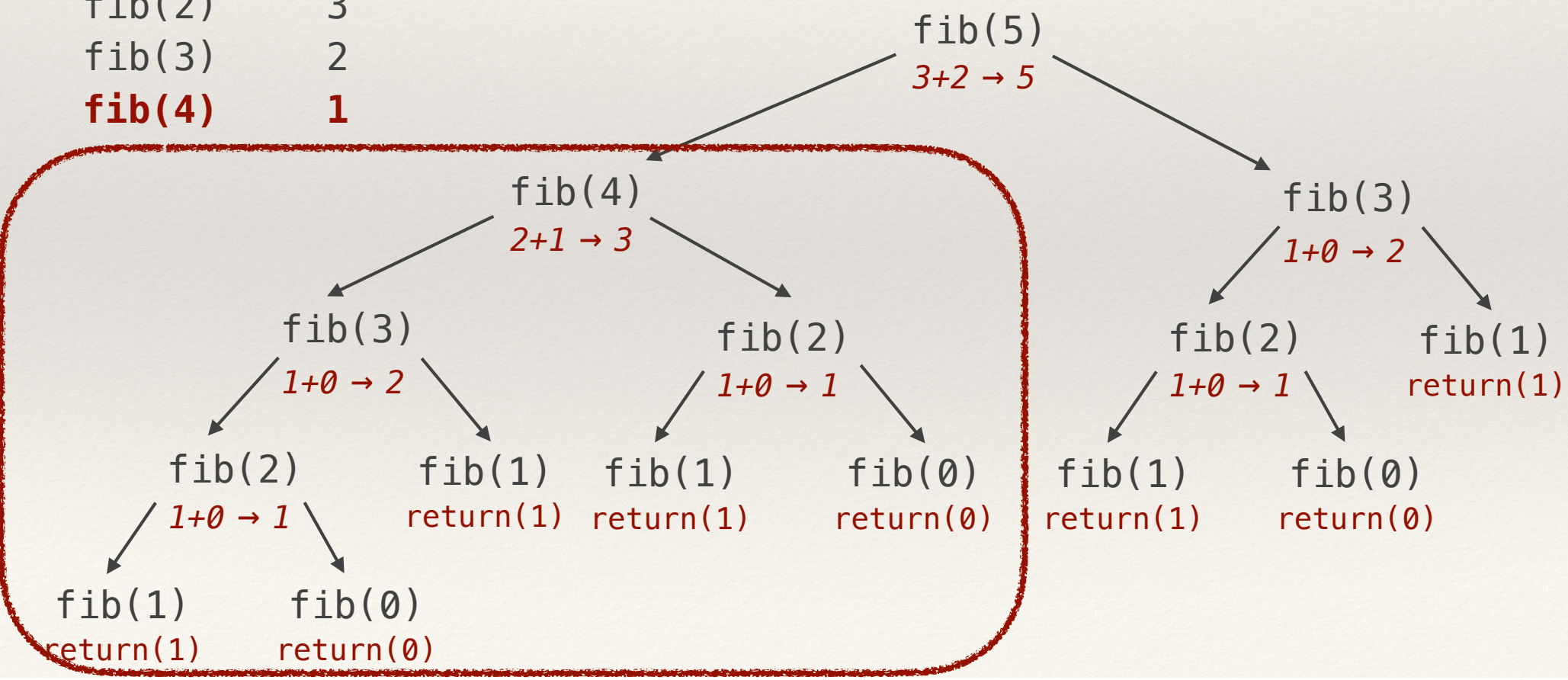
But…

- This implementation **is very inefficient**
  - when building the answer
    it **recomputes** a given number **over and over again**



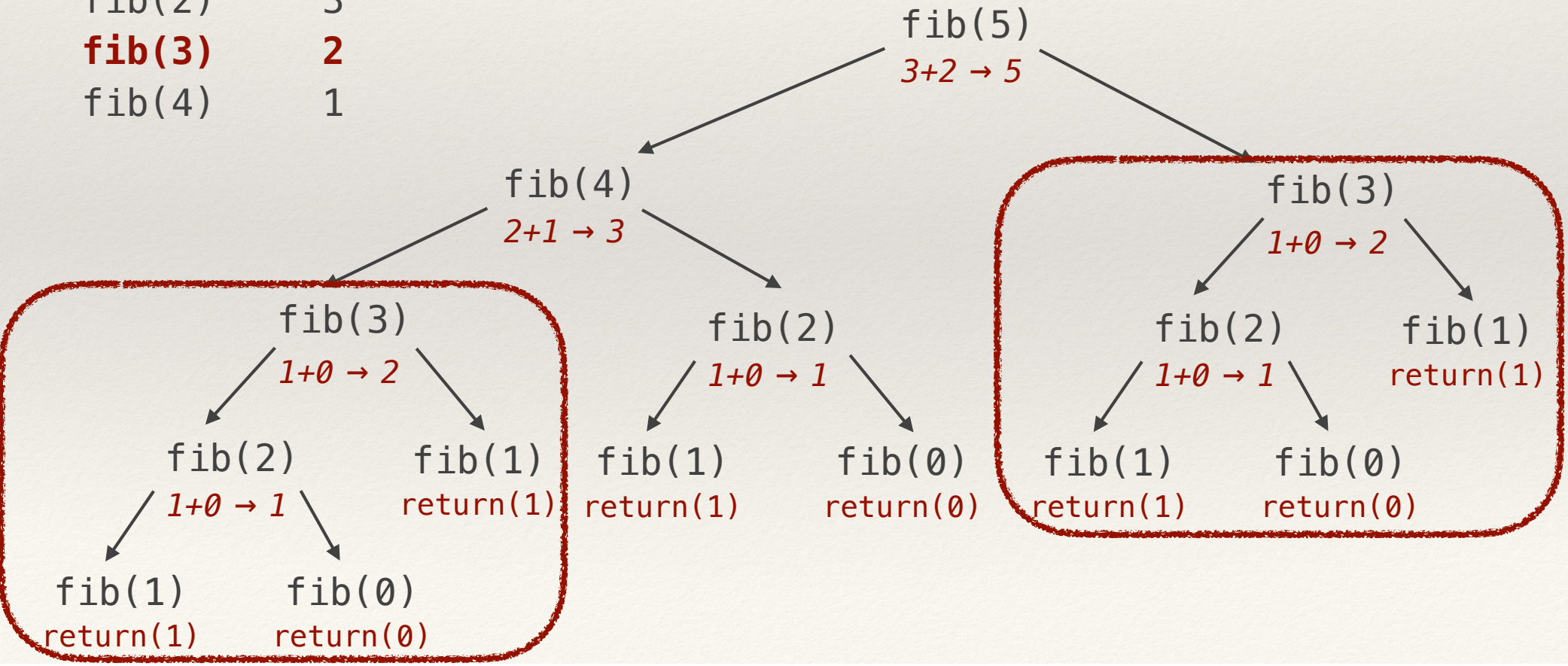
Computing `fib(5)`

	Times computed (again)
`fib(0)`	3
`fib(1)`	4
`fib(2)`	3
`fib(3)`	2
`fib(4)`	**1**



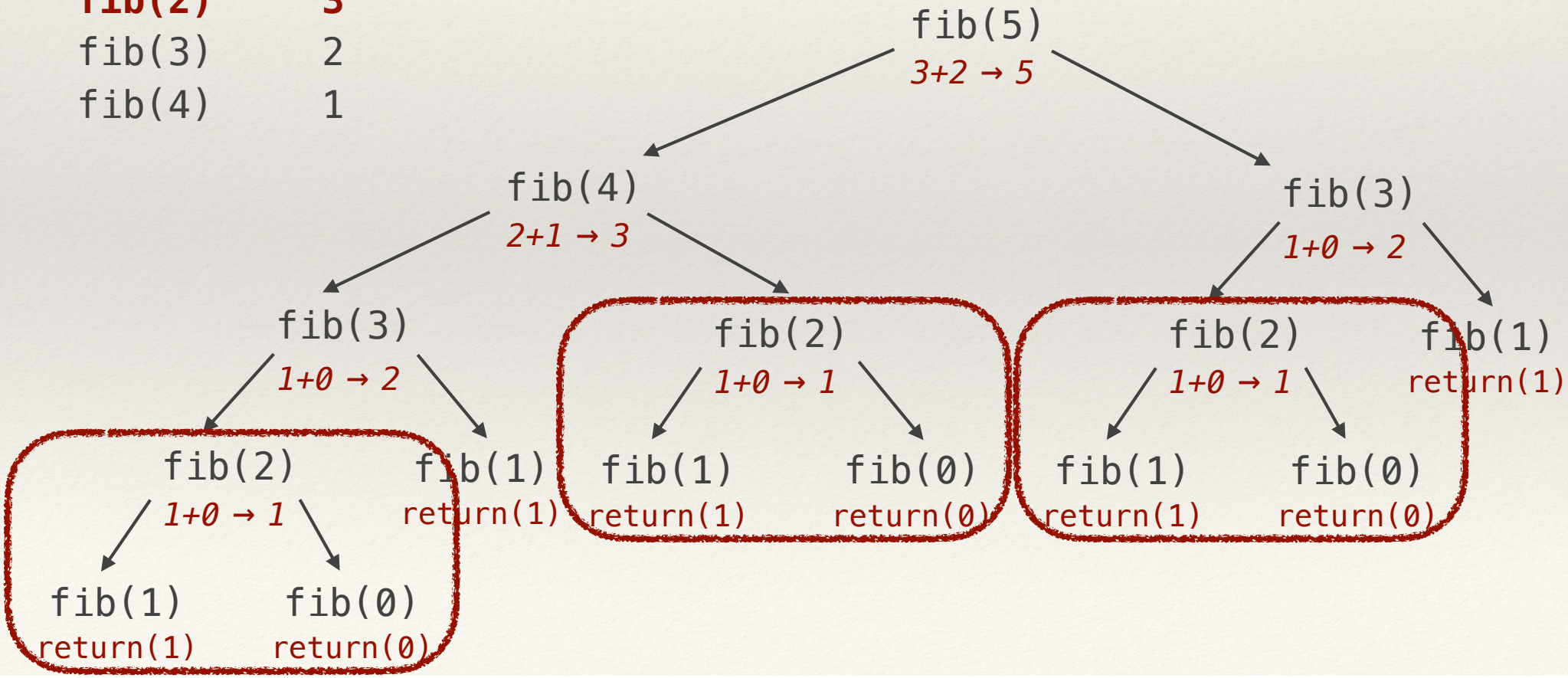
Computing `fib(5)`

	Times computed (again)
`fib(0)`	3
`fib(1)`	4
`fib(2)`	3
`fib(3)`	**2**
`fib(4)`	1



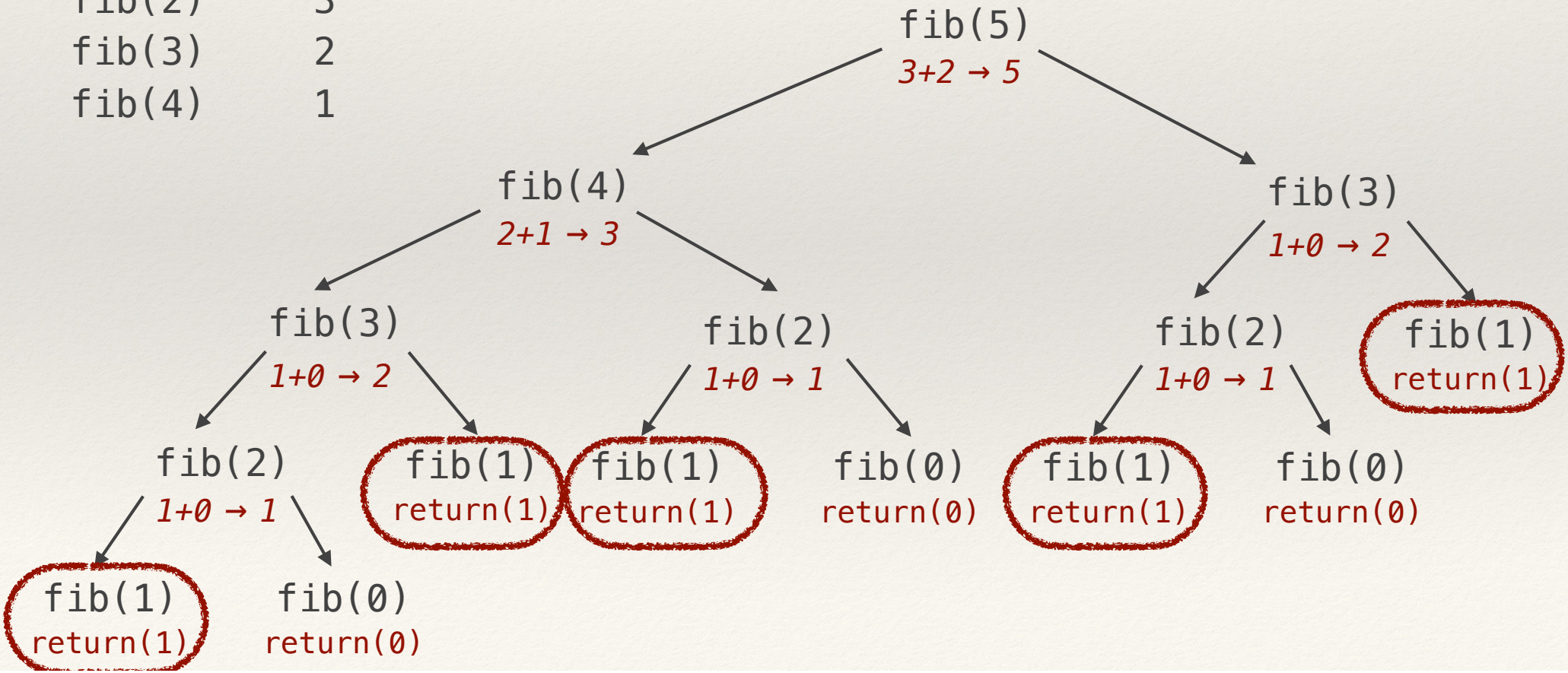
Computing `fib(5)`

	Times computed (again)
`fib(0)`	3
`fib(1)`	4
`fib(2)`	**3**
`fib(3)`	2
`fib(4)`	1



Computing `fib(5)`

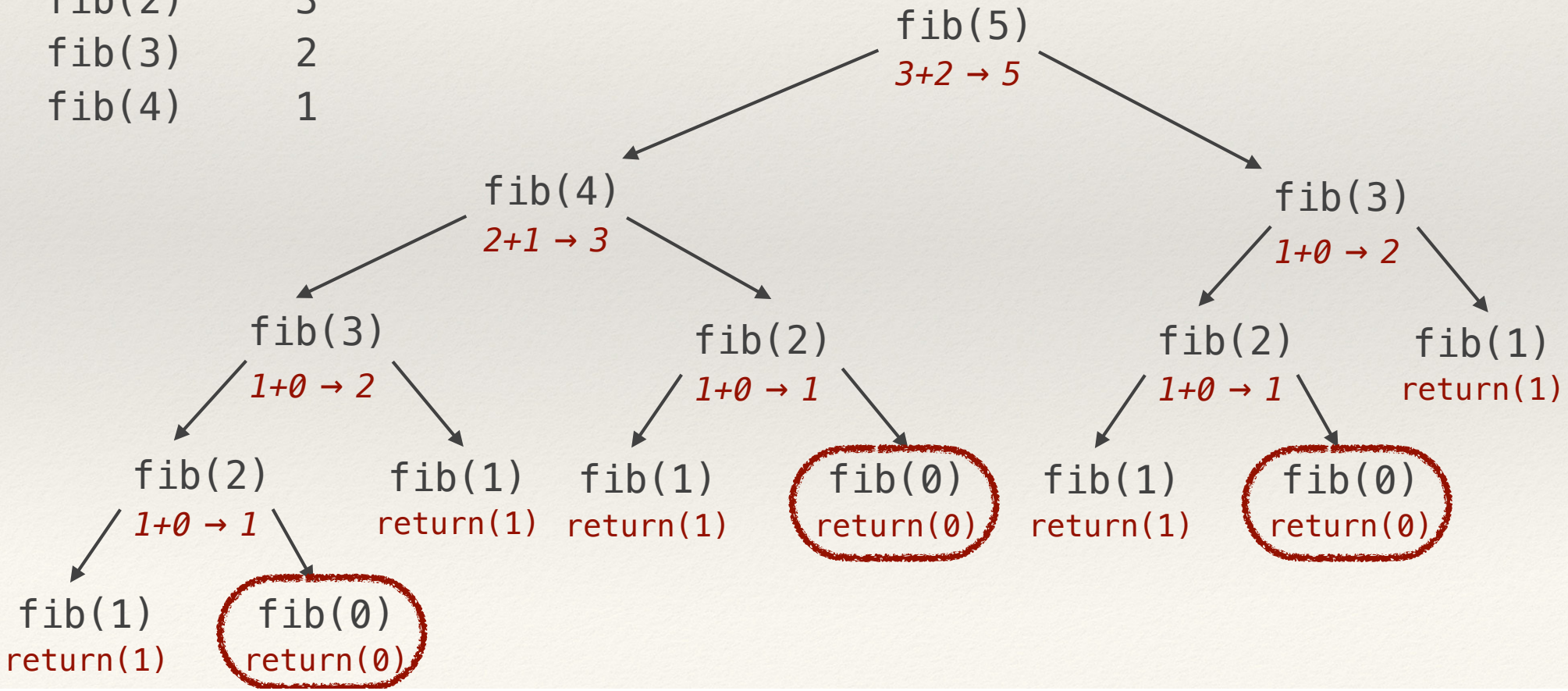
	Times computed (again)
`fib(0)`	3
`fib(1)`	**4**
`fib(2)`	3
`fib(3)`	2
`fib(4)`	1



Computing `fib(5)`

Times computed
(*again and again*)

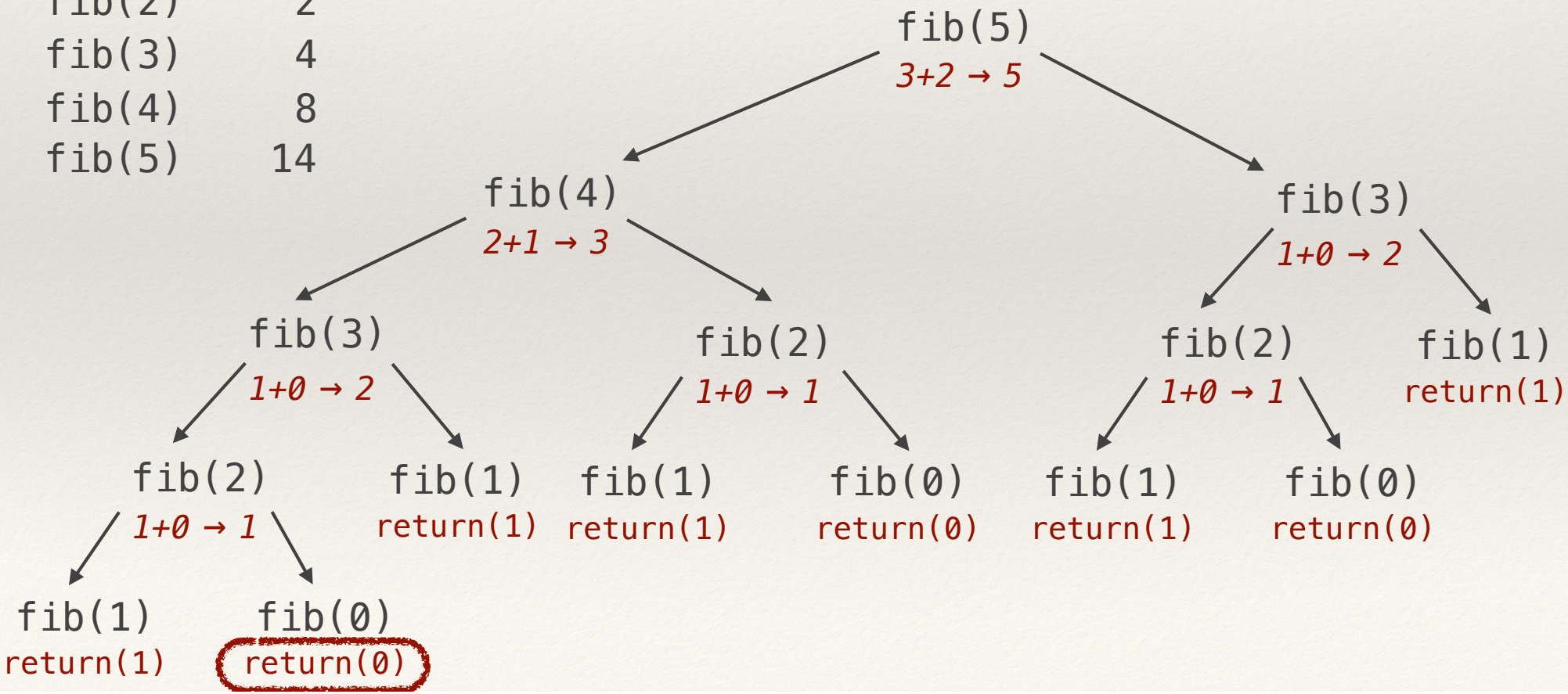
`fib(0)`	**3**
`fib(1)`	4
`fib(2)`	3
`fib(3)`	2
`fib(4)`	1



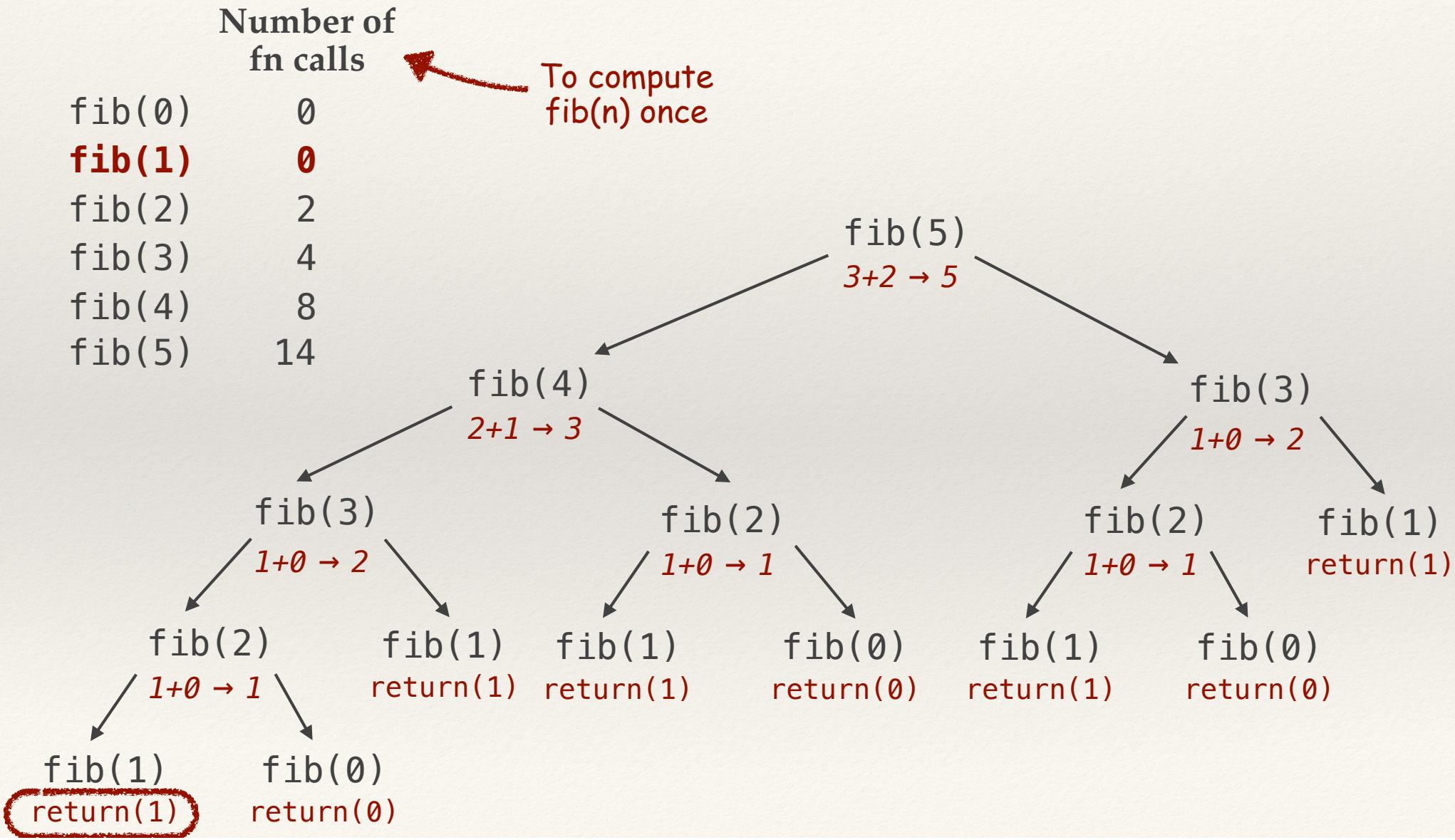
Computing `fib(5)`

	Number of fn calls
fib(0)	**0**
fib(1)	0
fib(2)	2
fib(3)	4
fib(4)	8
fib(5)	14

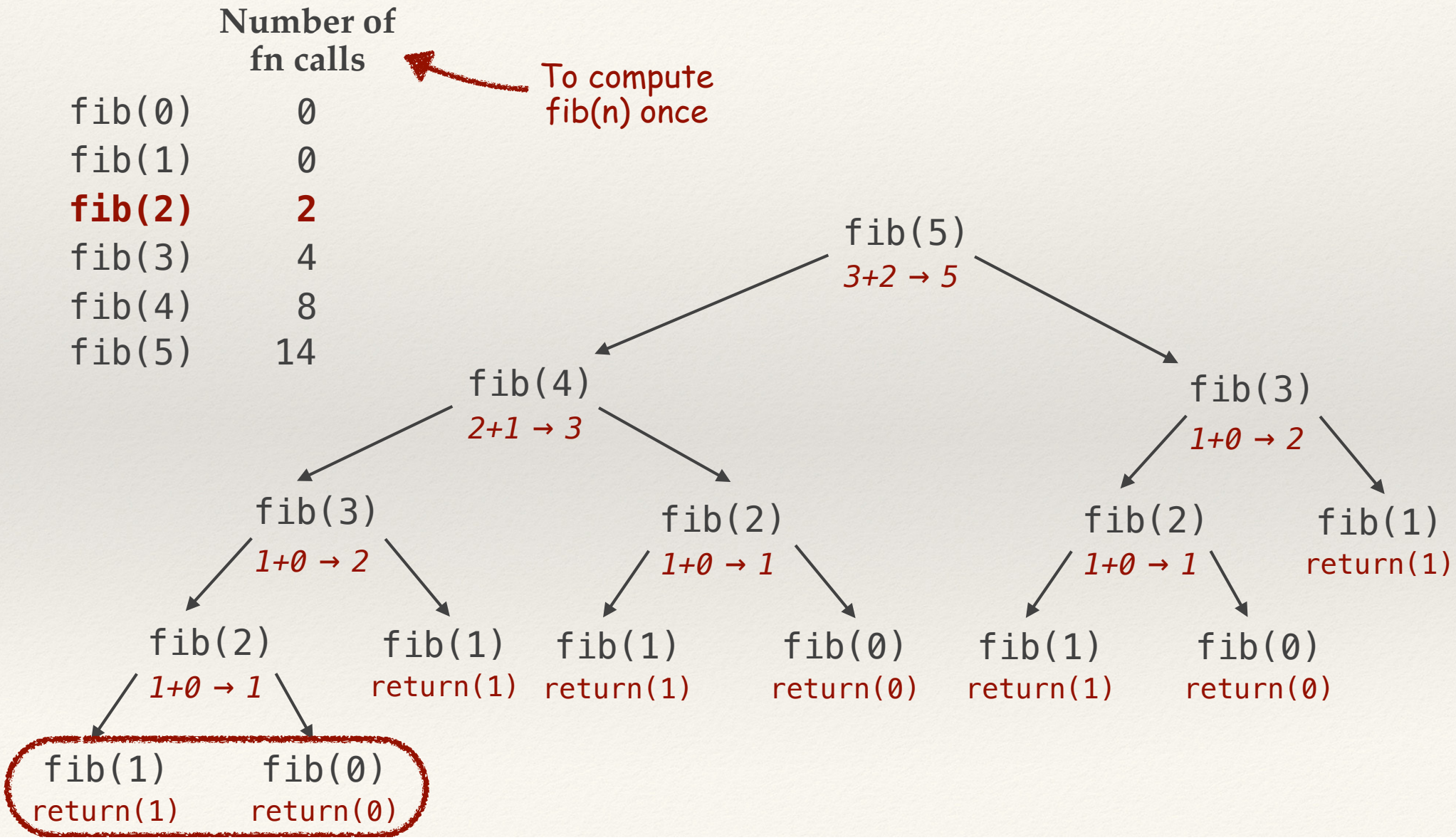
To compute fib(n) once



Computing `fib(5)`



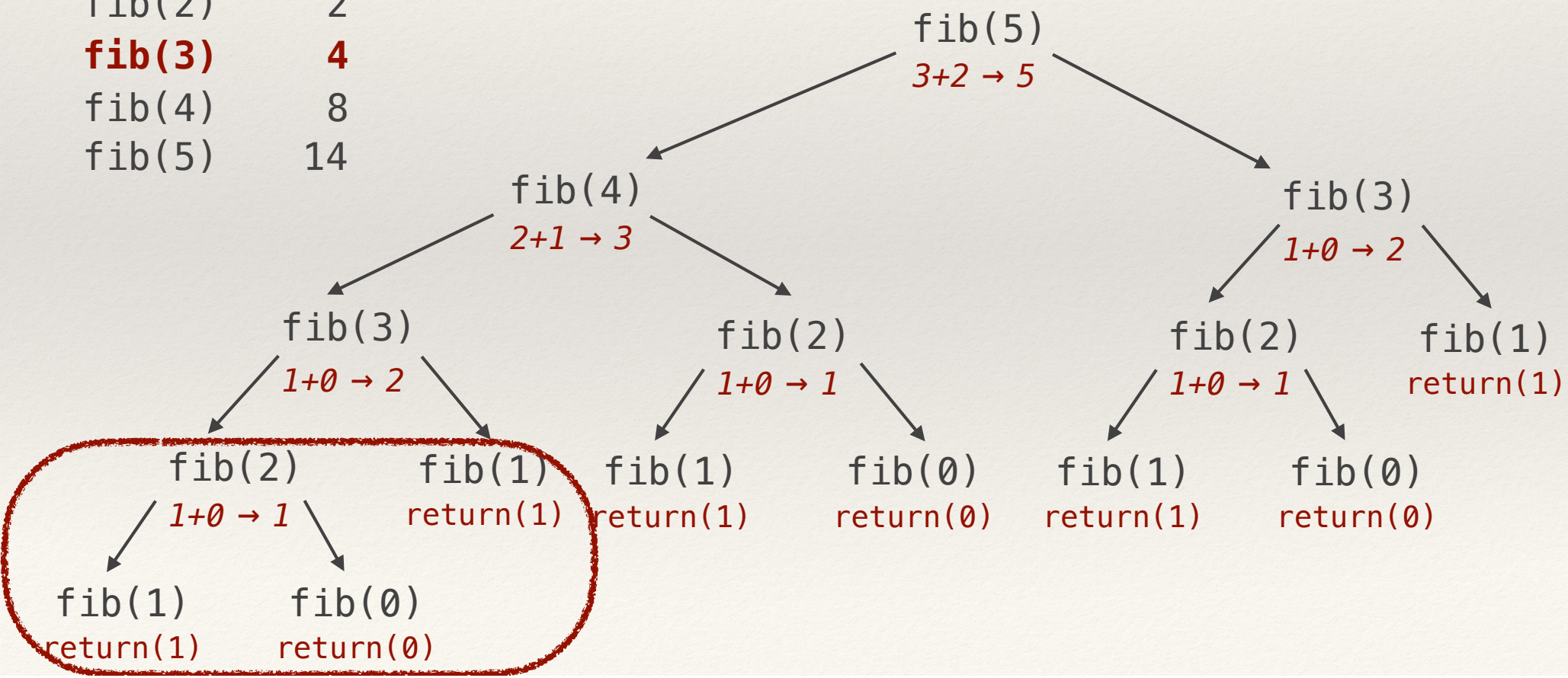
Computing `fib(5)`



Computing `fib(5)`

	Number of fn calls
fib(0)	0
fib(1)	0
fib(2)	2
fib(3)	**4**
fib(4)	8
fib(5)	14

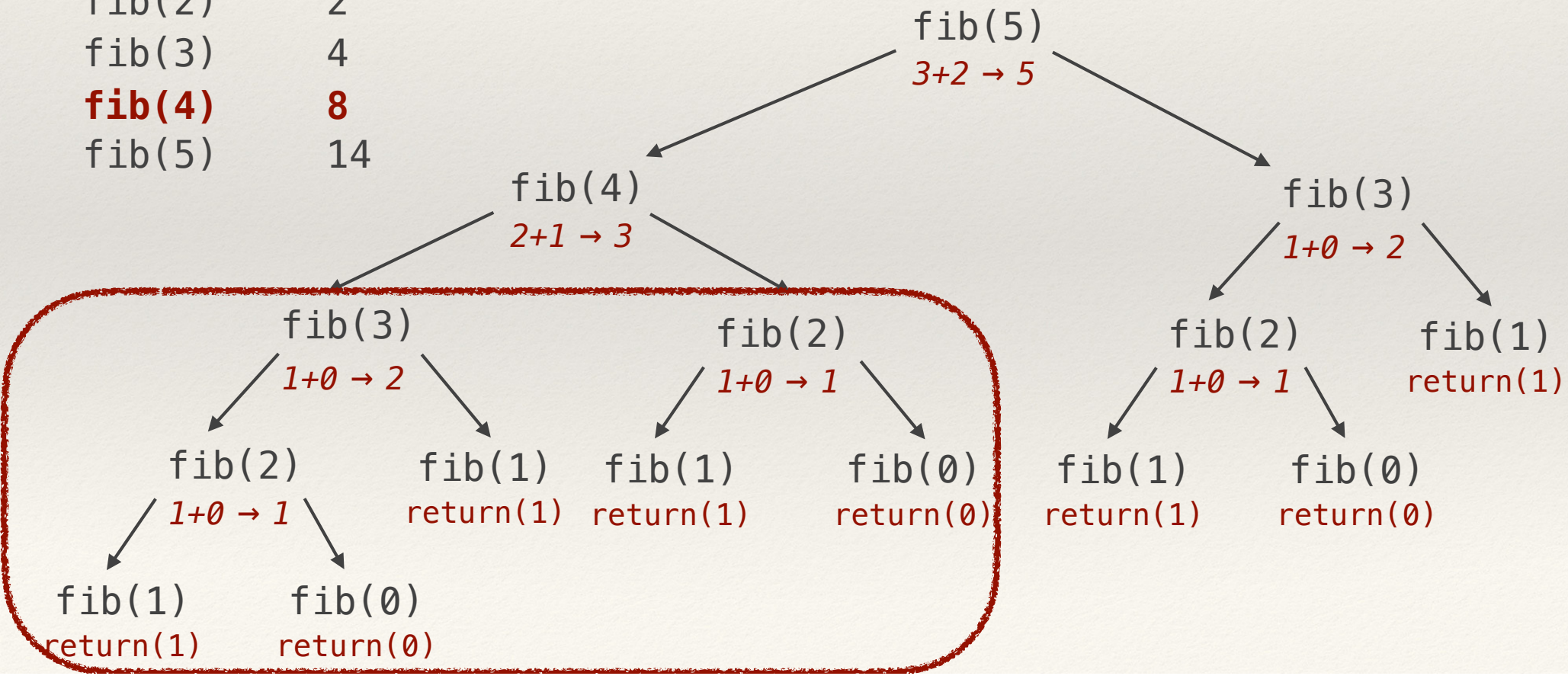
To compute fib(n) once



Computing `fib(5)`

	Number of fn calls
fib(0)	0
fib(1)	0
fib(2)	2
fib(3)	4
fib(4)	**8**
fib(5)	14

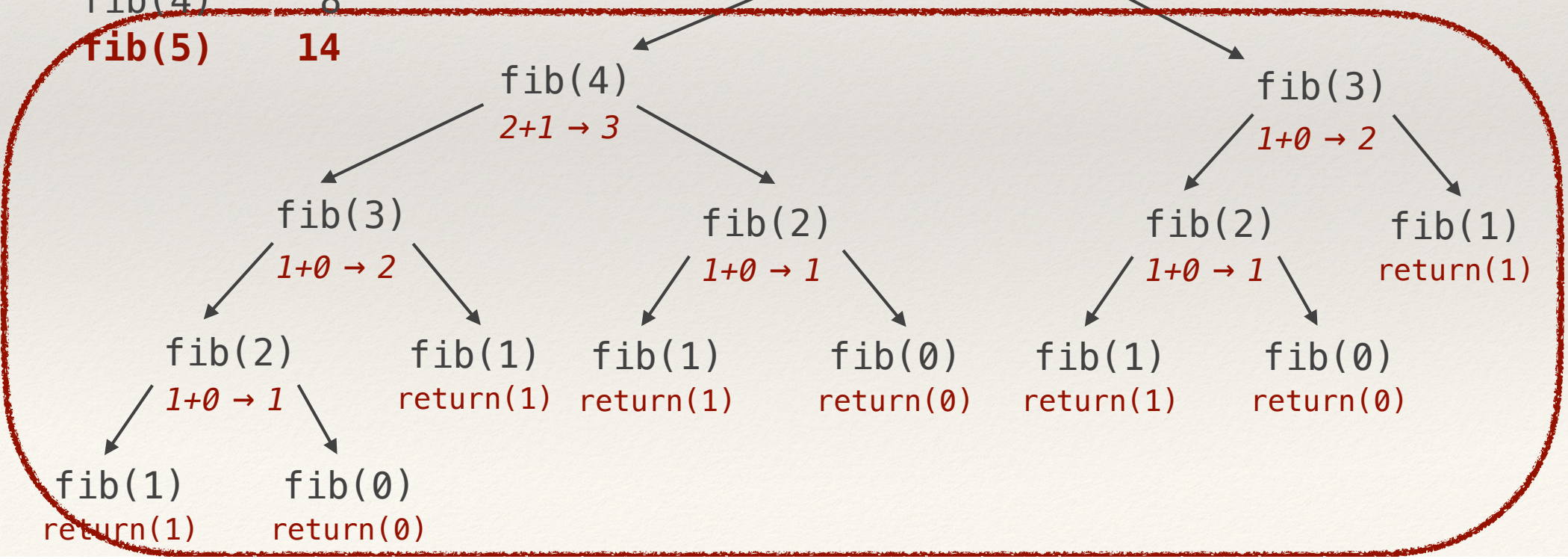
To compute fib(n) once



Computing `fib(5)`

	Number of fn calls
fib(0)	0
fib(1)	0
fib(2)	2
fib(3)	4
fib(4)	8
fib(5)	**14**

To compute fib(n) once



Recursive Fibonacci is Exponential

	Number of fn calls
fib(0)	0
fib(1)	0
fib(2)	2
fib(3)	4
fib(4)	8
fib(5)	14
fib(6)	24
fib(7)	40
fib(8)	66
fib(9)	108
fib(10)	176
fib(11)	286
fib(12)	464
fib(13)	752
fib(14)	1218

	Number of fn calls
fib(15)	1972
fib(16)	3192
fib(17)	5166
fib(18)	8360
fib(19)	13528
fib(20)	21890
fib(21)	35420
fib(22)	57312
fib(23)	92734
fib(24)	150048
fib(25)	242784
fib(26)	392834
fib(27)	635620
fib(28)	1028456
fib(29)	1664078

Exponential Growth!!!!

	Time
fib(30)	0.006s
fib(40)	0.629s
fib(50)	1m 16s
fib(60)	2h 35m 28s
fib(70)	13d 6h 41m 47s

Solution 1: The Iterative Approach

- The problem is that we have stated the problem incorrectly. We are not looking to find the *n*th term but are instead looking at a way to construct a sequence.
- This is better done as an iterative (loop-based) function.

Iterative implementation of Fibonacci

```
int fib_itr ( int n ) {
   int i;
   int oneBack, twoBack, current;

   if ( n <= 0 ) {
      return (0);
   } else if ( n == 1 ) {
      return (1);
   } else {
      twoBack = 0;
      oneBack = 1;
      for ( i=2; i<=n; i++ ) {
         current = twoBack + oneBack;
         twoBack = oneBack;
         oneBack = current;
      }
      return (current);
   }
}
```

- each row a previous fib(n)
- only called once

Computing fib(14)

i	current	oneBack	twoBack
0	0		
1	1	0	
2	1	1	0
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3
7	13	8	5
8	21	13	5
9	34	21	13
10	45	34	21
11	79	45	34
12	124	79	45
13	203	124	79
14	327	203	124

Very fast

Solution 2: Tail Recursion

So does that mean recursion is bad after all?

- *at least for Fibonacci*

Many people think so!

- *commonly taught that way*

No!! We just set up the recursion poorly

All iteration can be implemented through recursion, and perform just as quickly, by means of *tail recursion*

Tail Recursive Implementation of Fibonacci

```
int fib_tail ( int n ) {
    return (n < 2) ? n : fib_tr(1, 0, 2, n);
}

int fib_tr ( int oneback, int twoback, int n, int N ) {
    int current =  oneback + twoback;
    return (n == N) ? current : fib_tr(current, oneback, n + 1, N);
}
```

- each row a prev recursive call to fib(n)
- only called once

Computing fib(6)

fib_tr(n)	current	oneback	twoback
0	0		
1	1	0	
2	1	1	0
3	2	1	1
4	3	2	1
5	5	3	2
6	8	5	3

Just as fast

Tail Recursive Implementation of Fibonacci

```
int fib_tail ( int n ) {
    return (n < 2) ? n : fib_tr(1, 0, 2, n);
}

int fib_tr ( int oneback, int twoback, int n, int N ) {
    int current =  oneback + twoback;
    return (n == N) ? current : fib_tr(current, oneback, n + 1, N);
}
```

The trick is to

- compute the current "iteration" answer
- then recurse, supplying the growing answer as a parameter to the next step
- finally return the answer when you are finished

Tail Recursive Implementation of Fibonacci

```
int fib_tail ( int n ) {
    return (n < 2) ? n : fib_tr(1, 0, 2, n);
}

int fib_tr ( int oneback, int twoback, int n, int N ) {
    int current =  oneback + twoback;
    return (n == N) ? current : fib_tr(current, oneback, n + 1, N);
}
```

Why is it called tail recursion?

- Notice that if the recursion function is called,
  … it is the final calculation
- The only thing done when the function completes,
  … is to return the value back out
  - No extra computation is done!
- This is called a “tail call”
  - if the recursive step is in the “tail position”
    it is called *tail recursion*

Tail Recursive Implementation of Fibonacci

```
int fib_tail ( int n ) {
    return (n < 2) ? n : fib_tr(1, 0, 2, n);
}

int fib_tr ( int oneback, int twoback, int n, int N ) {
    int current =  oneback + twoback;
    return (n == N) ? current : fib_tr(current, oneback, n + 1, N);
}
```

Optimizing for tail recursion

- There is still an advantage to the iterative solution…
  - calling a function takes set up time
  - The iterative solution does not call a function
  - The tail recursive solution calls *n functions*
- However, many compilers can recognize a "tail call"
  - Those compilers will use the same function code and stack frame instead of creating a new one when performing the recursive call
  - thus taking no time to set up, i.e it is just as quick as iterative code

Tail Recursive Implementation of Fibonacci

```
int fib_tail ( int n ) {
    return (n < 2) ? n : fib_tr(1, 0, 2, n);
}

int fib_tr ( int oneback, int twoback, int n, int N ) {
    int current =  oneback + twoback;
    return (n == N) ? current : fib_tr(current, oneback, n + 1, N);
}
```

Optimizing for tail recursion

- gcc can recognize a tail call and perform tail recursion, but…
  - You have to tell it to do so
- This is done using optimization flags
  - `-foptimize-sibling-calls` will enable tail recursion
  - and which is part of the -O2 optimization level setting
- So you should call gcc using
  - `gcc -Wall -foptimize-sibling-calls -o fib fibonacci.c`

or

  - `gcc -Wall -O2 -o fib fibonacci.c`