NPTEL MOOC

PROGRAMMING, DATA STRUCTURES AND ALGORITHMS IN PYTHON

Week 8, Lecture 1

Madhavan Mukund, Chennai Mathematical Institute
http://www.cmi.ac.in/~madhavan

Inductive definitions

- Factorial
  - $f(0) = 1$
  - $f(n) = n \times f(n-1)$
- Insertion sort
  - isort([ ]) = [ ]
  - isort([x1,x2,..,xn]) = insert(x1,isort([x2,…,xn]))

... Recursive programs

```
def factorial(n):
  if n <= 0:
    return(1)
  else:
    return(n*factorial(n-1))
```

Sub problems

- factorial(n-1) is a subproblem of factorial(n)
  - So are factorial(n-2), factorial(n-3), …, factorial(0)
- isort([x2,…,xn]) is a subproblem of isort([x1,x2,…,xn])
  - So is isort([xi,…,xj]) for any $1 \leq i \leq j \leq n$
- Solution of f(y) can be derived by combining solutions to subproblems

Evaluating subproblems

Fibonacci numbers

- fib(0) = 0
- fib(1) = 1
- fib(n) = fib(n-1) + fib(n-2)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

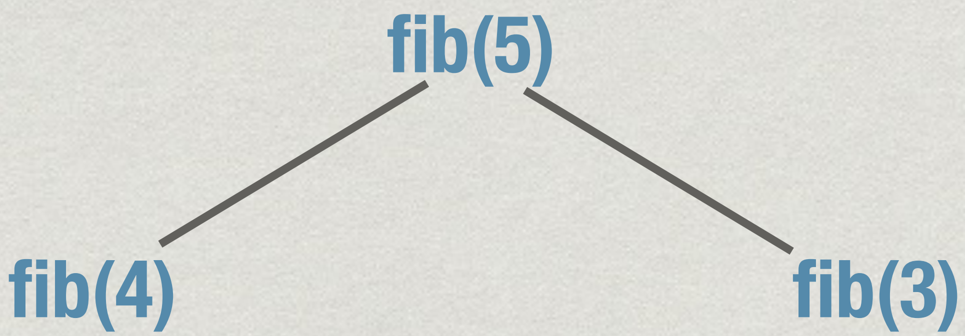
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```

fib(5)

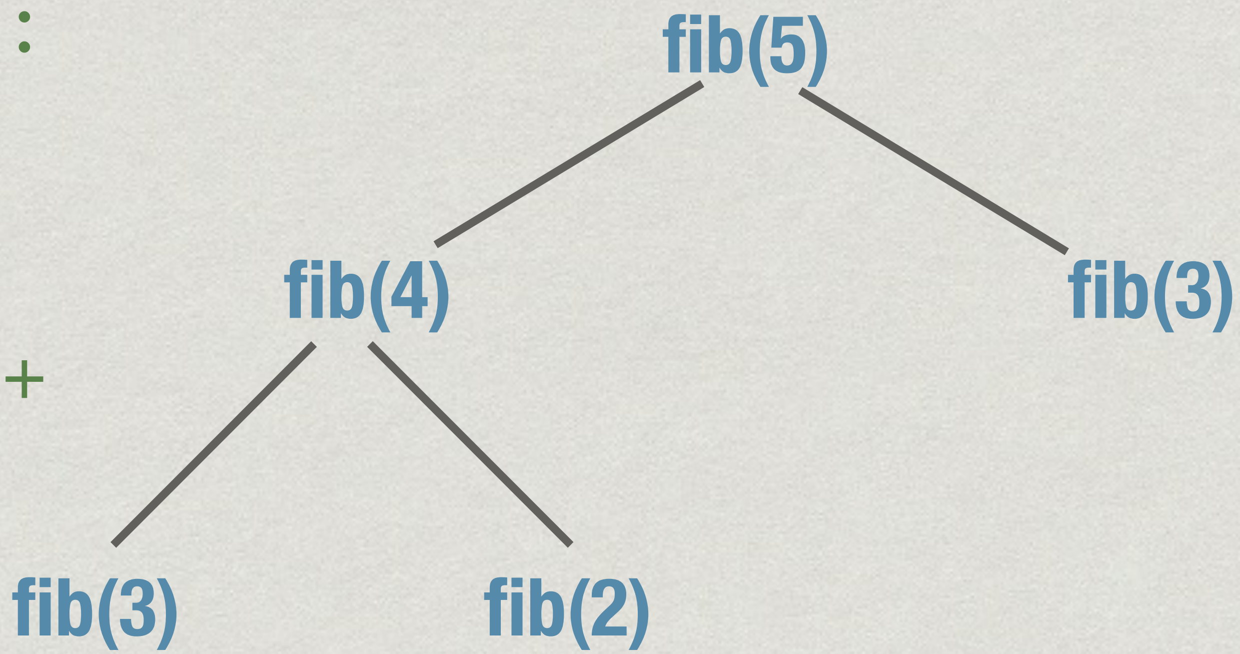
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



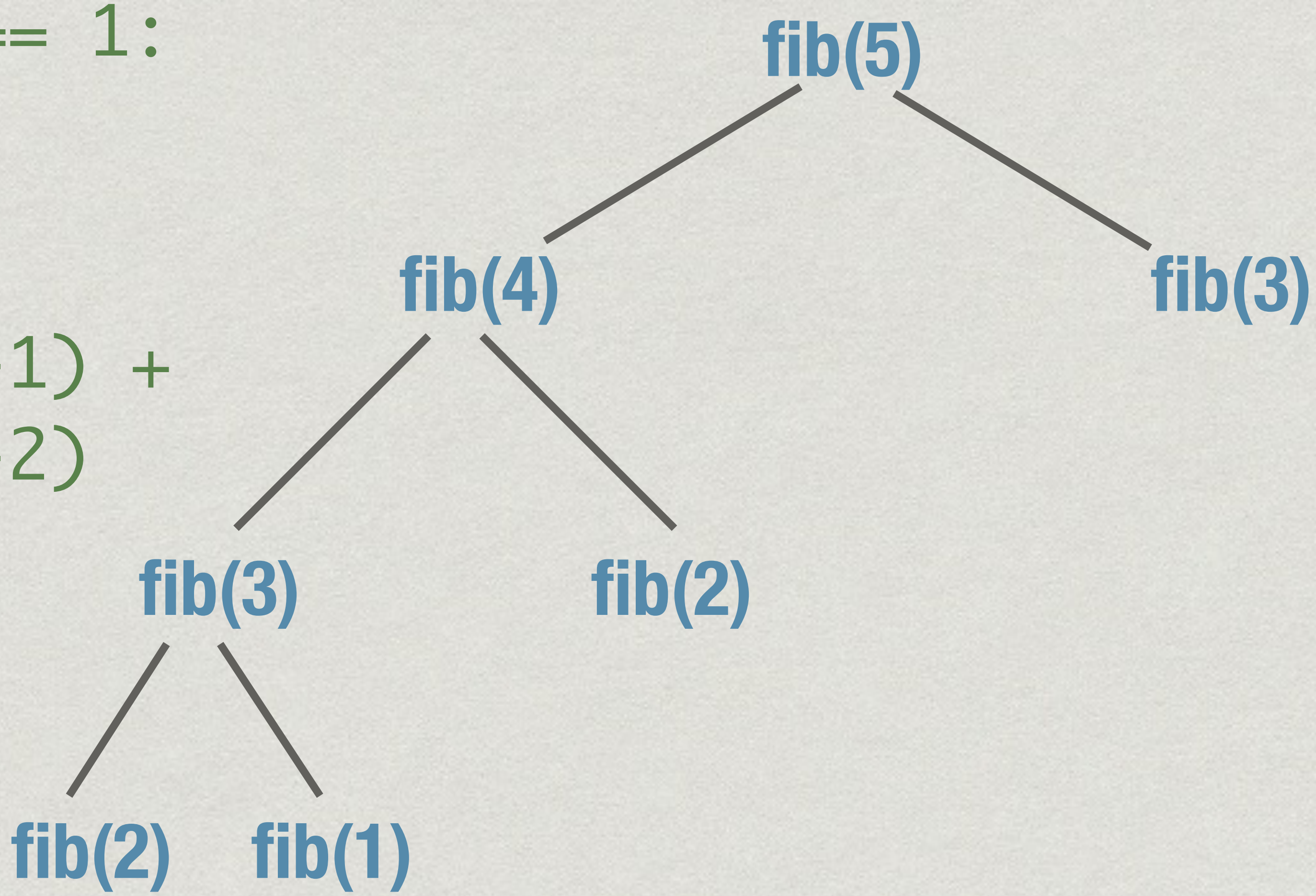
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



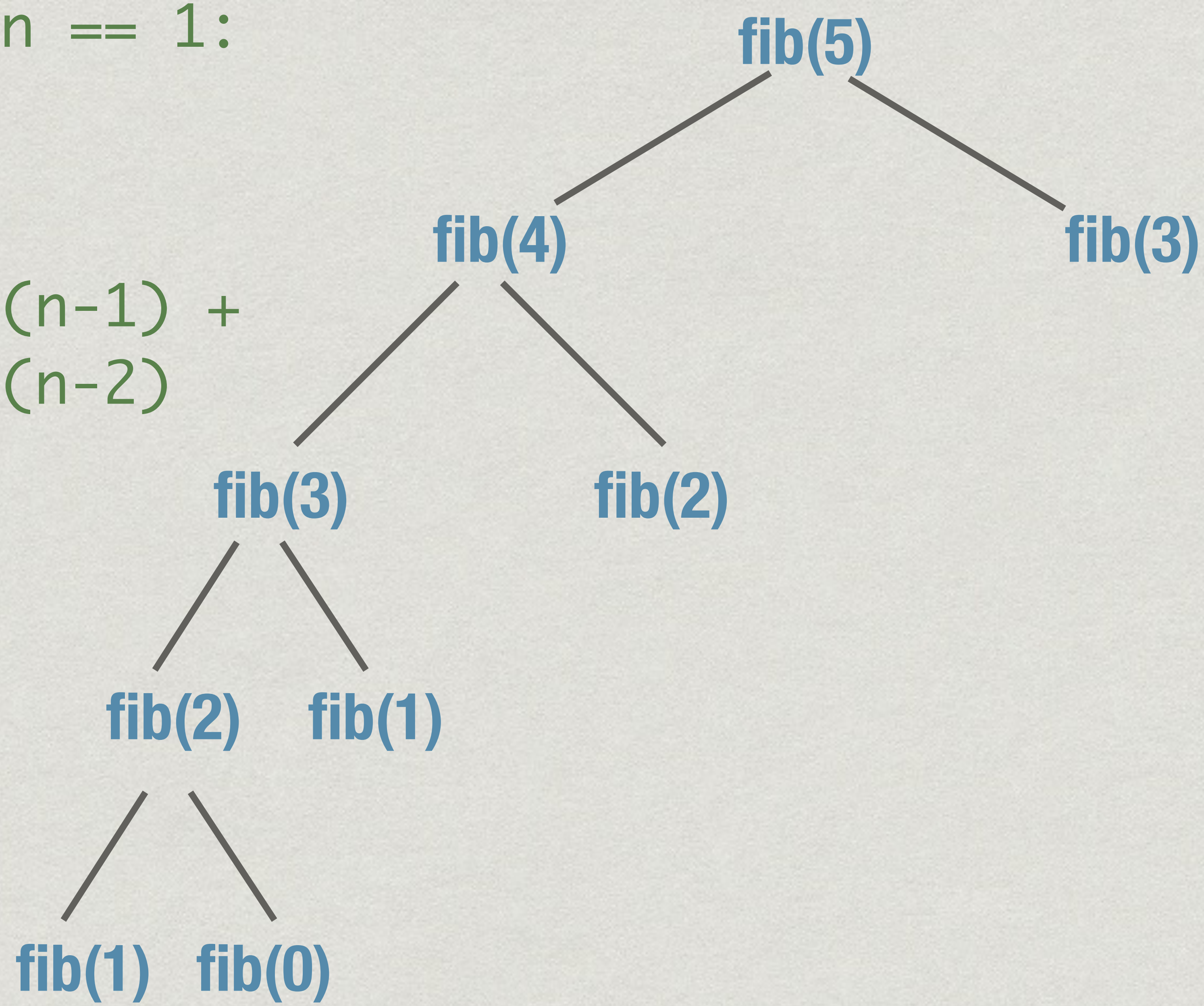
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



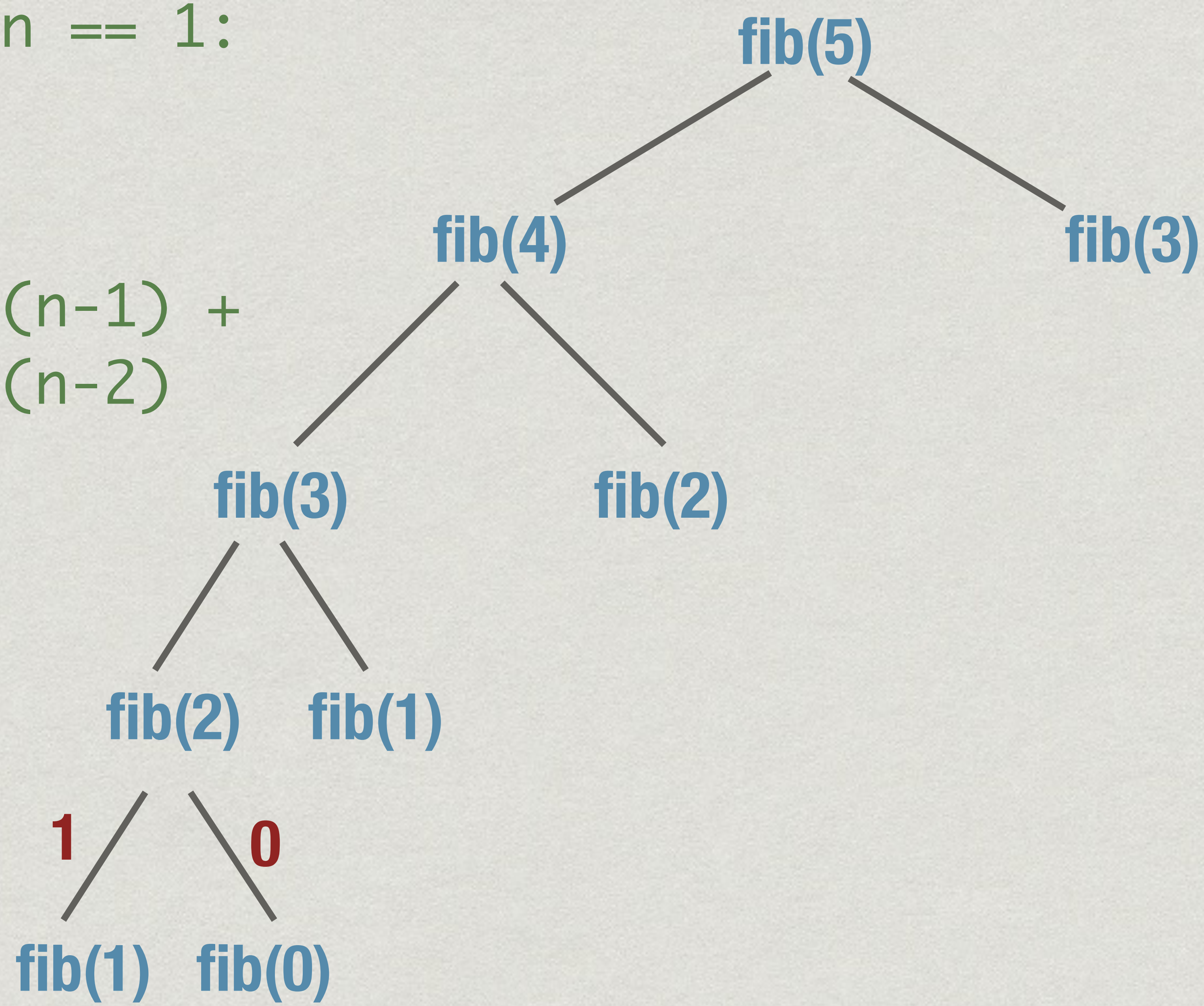
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



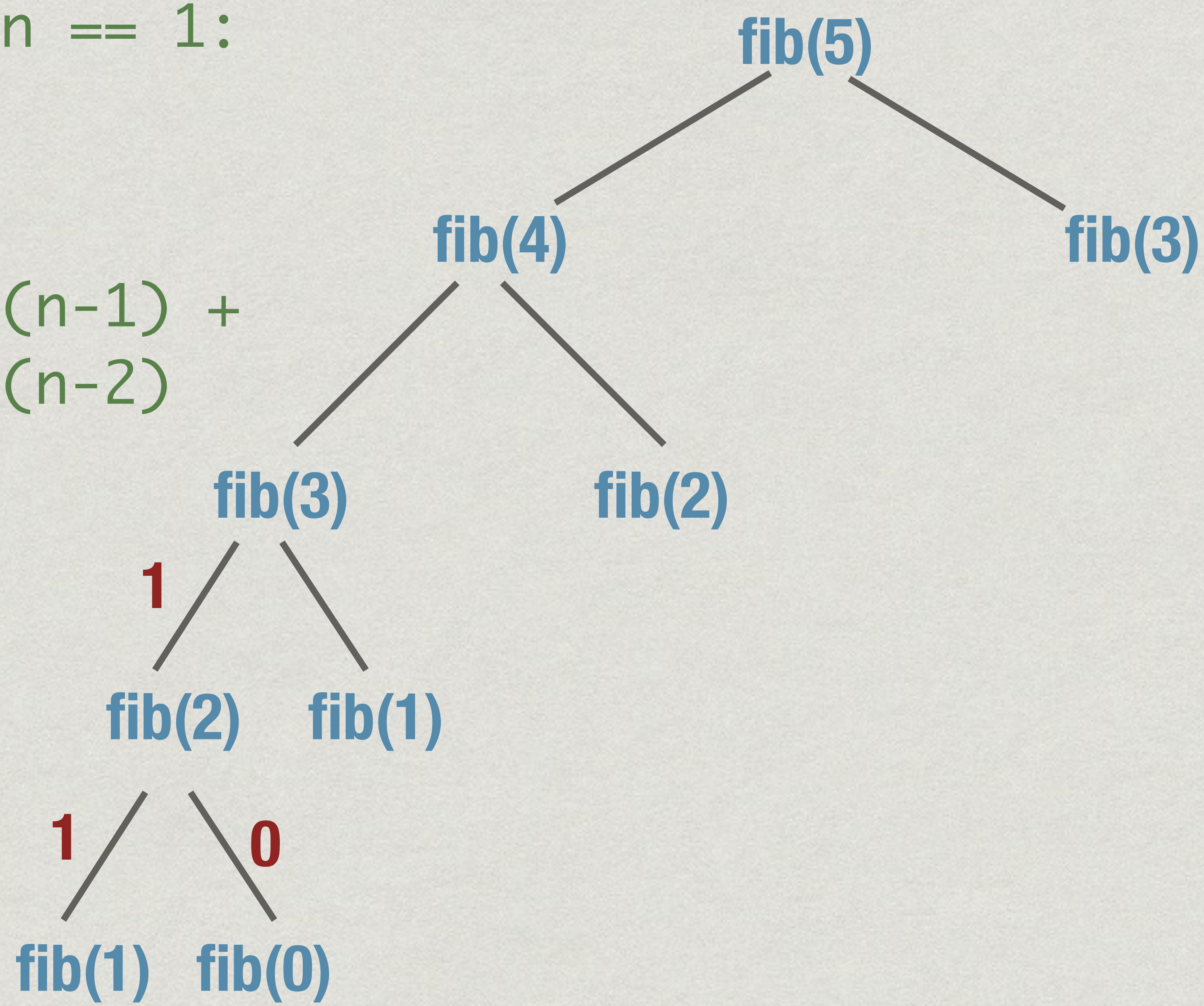
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



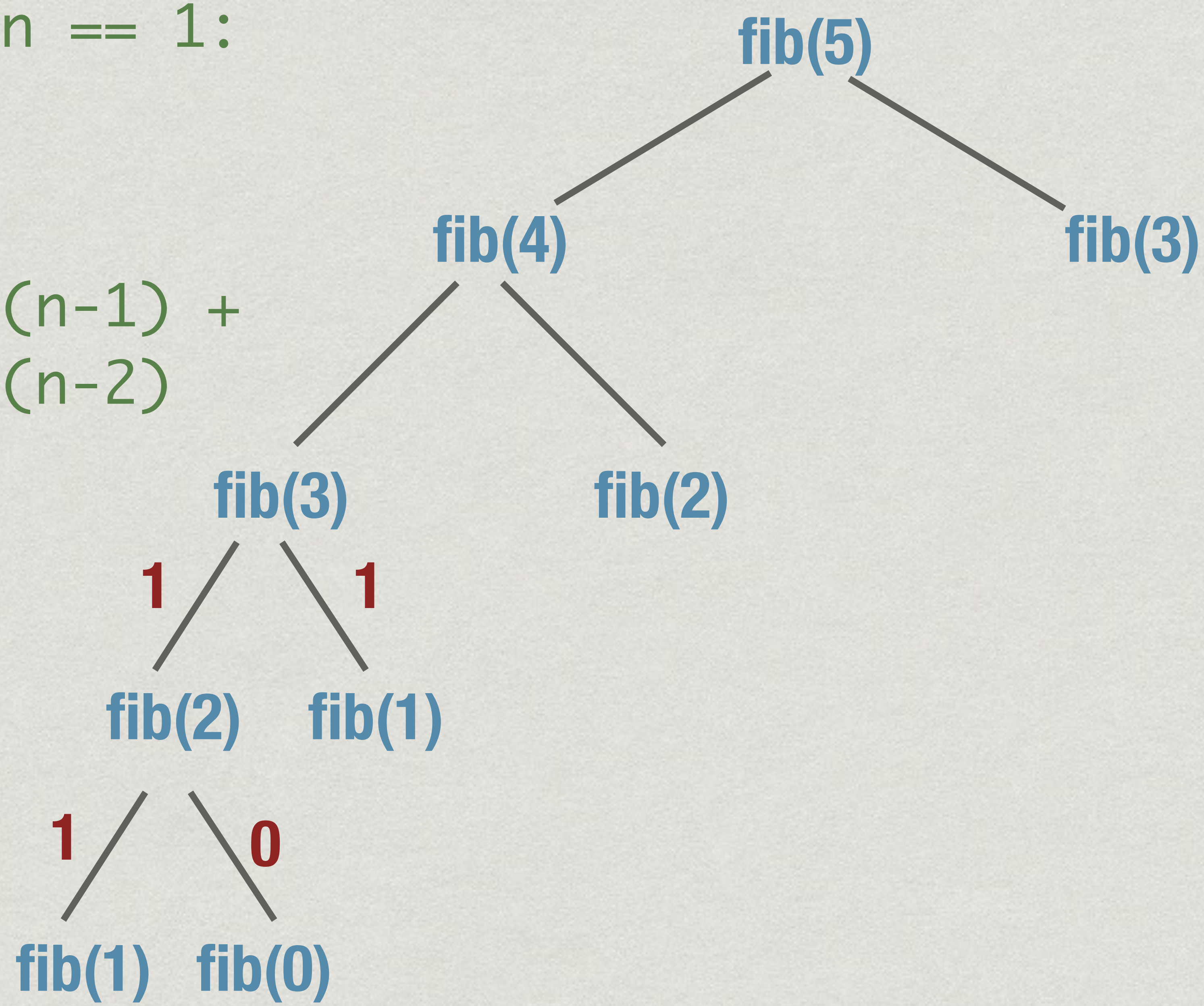
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



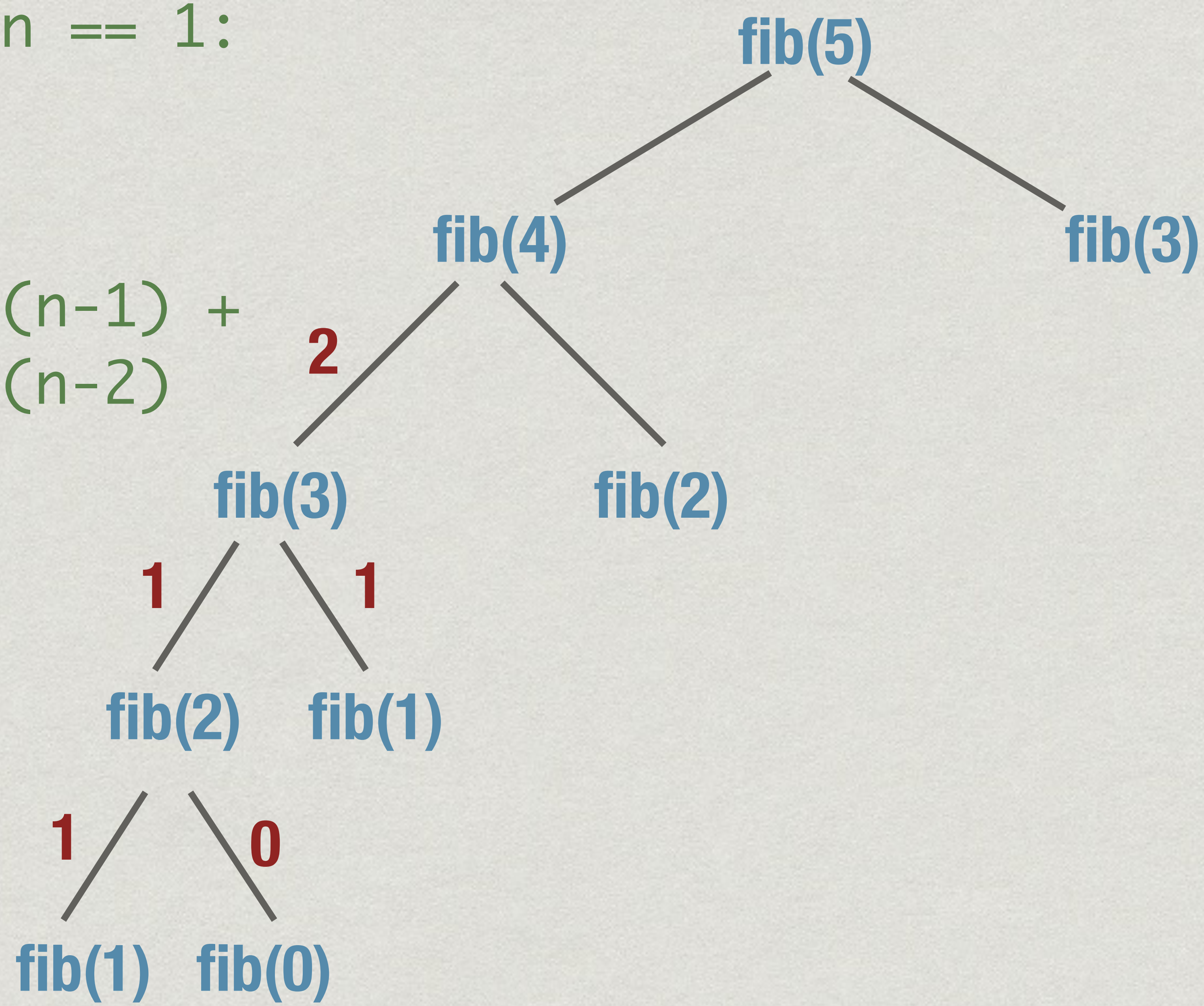
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



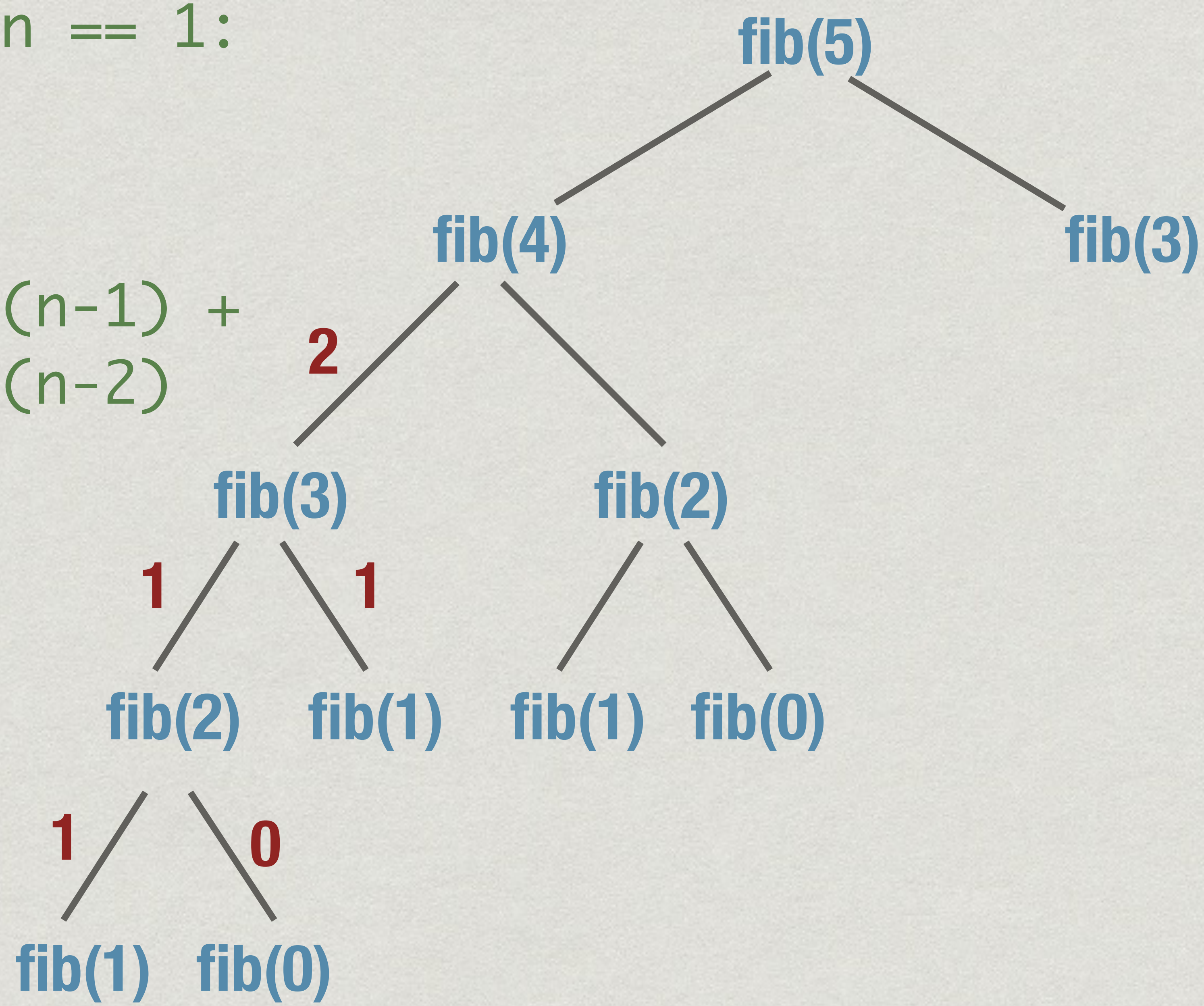
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



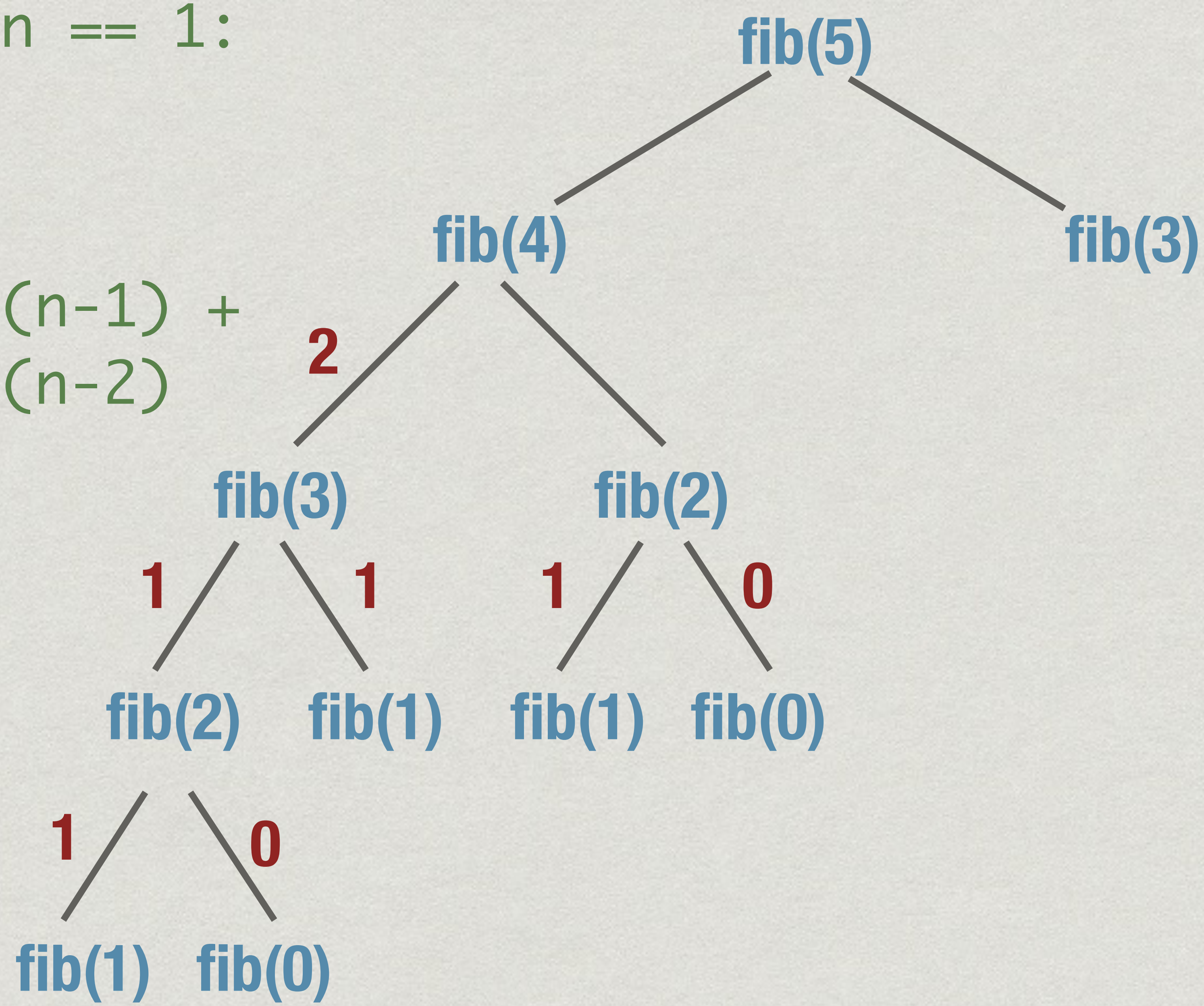
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



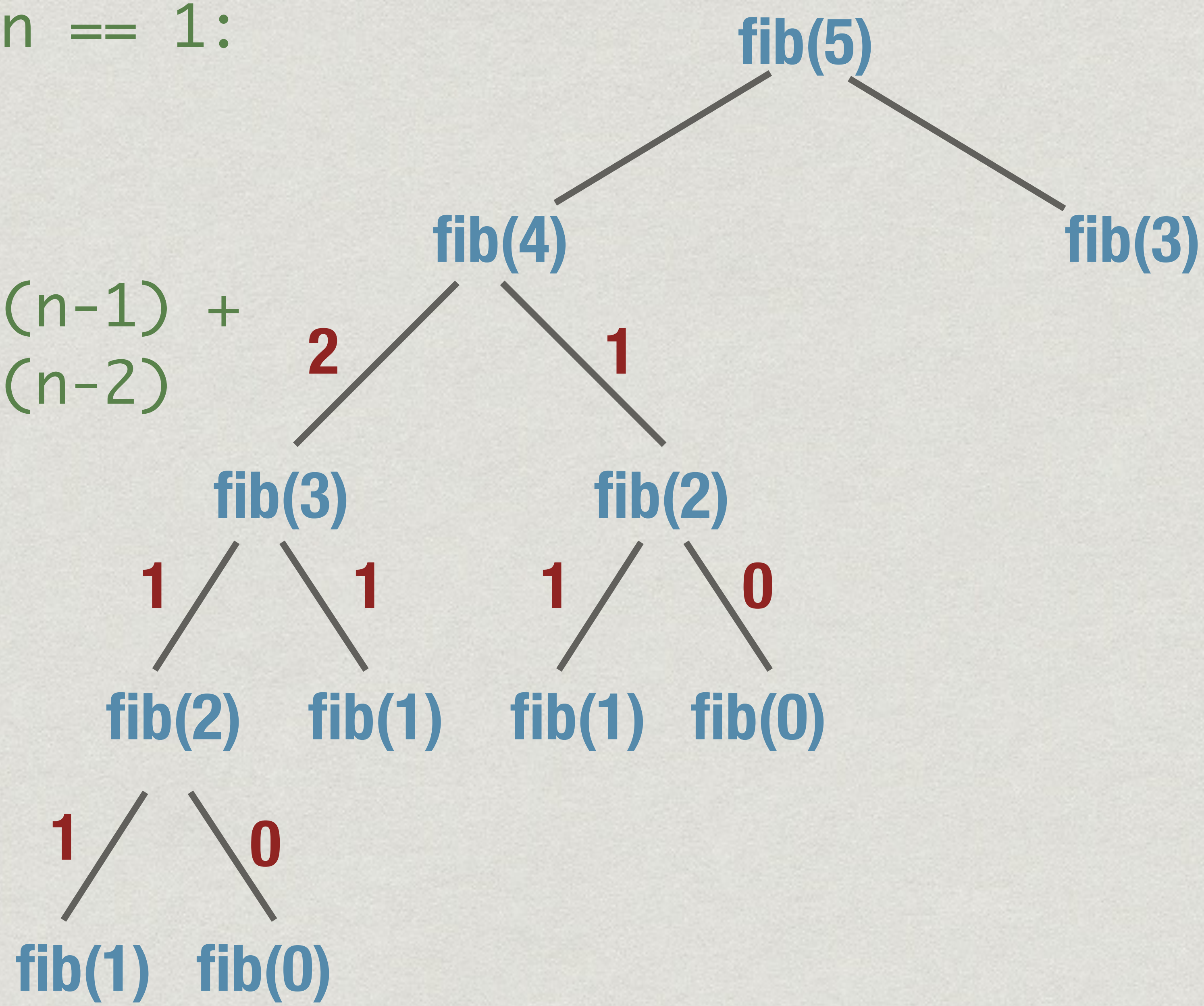
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



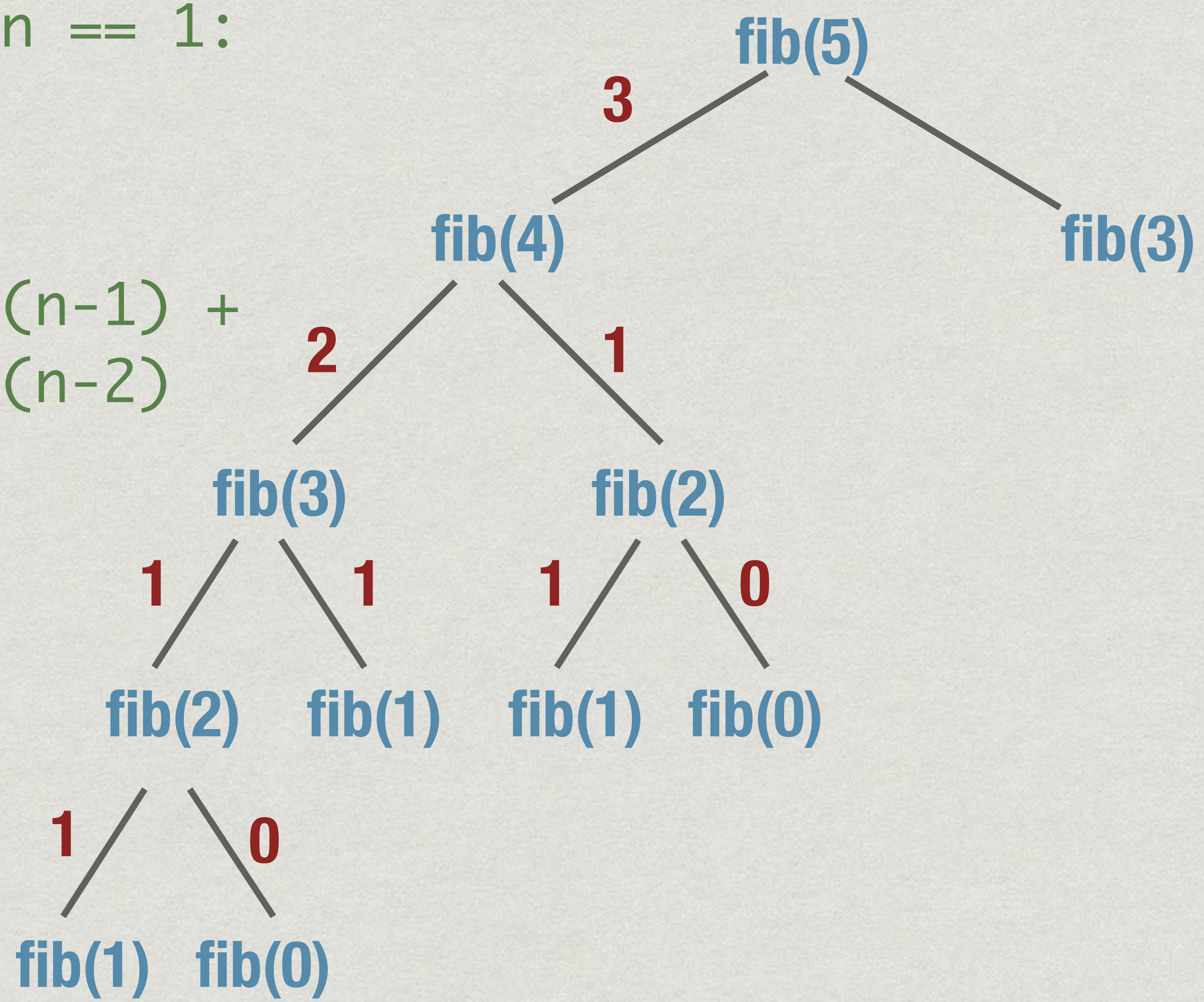
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



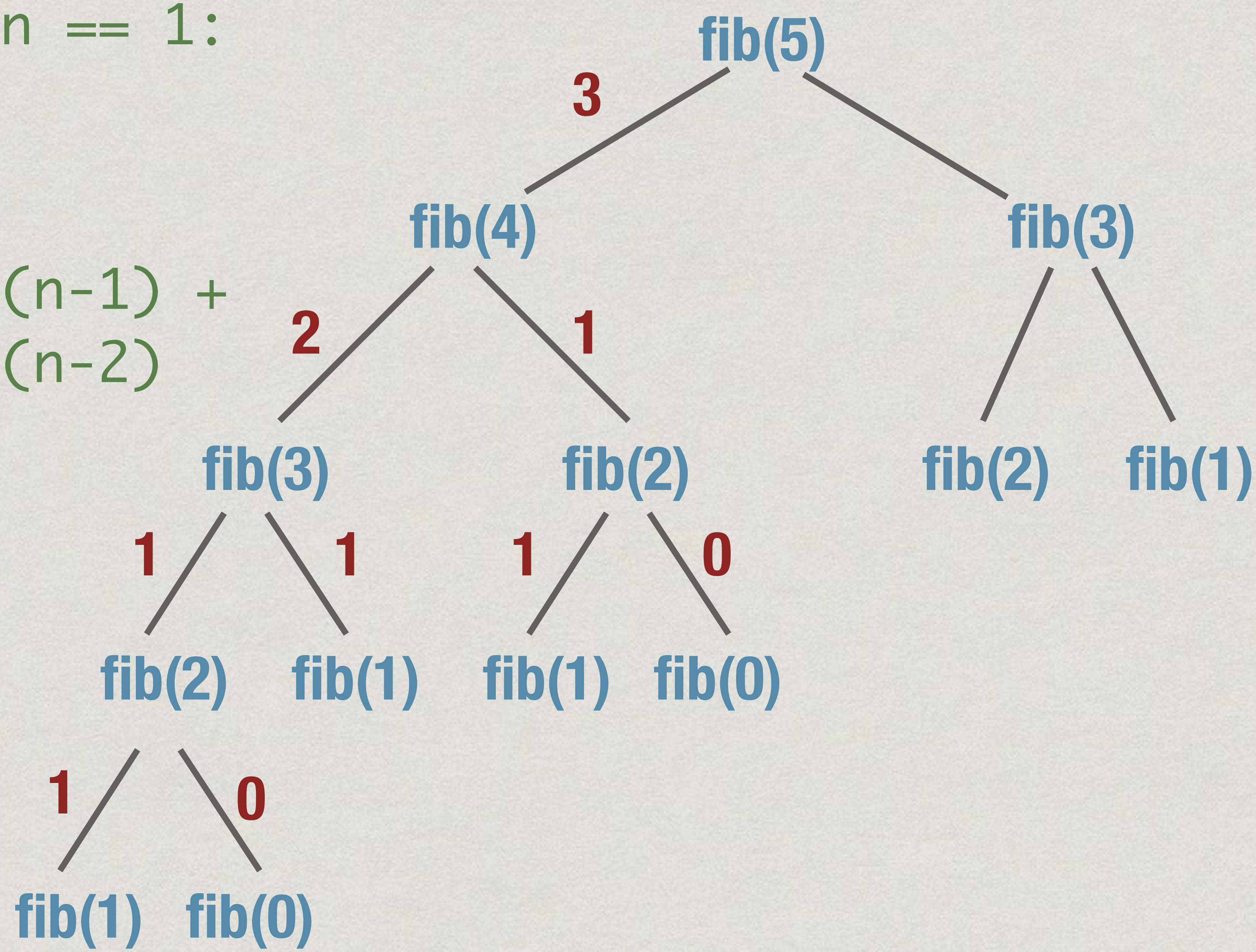
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



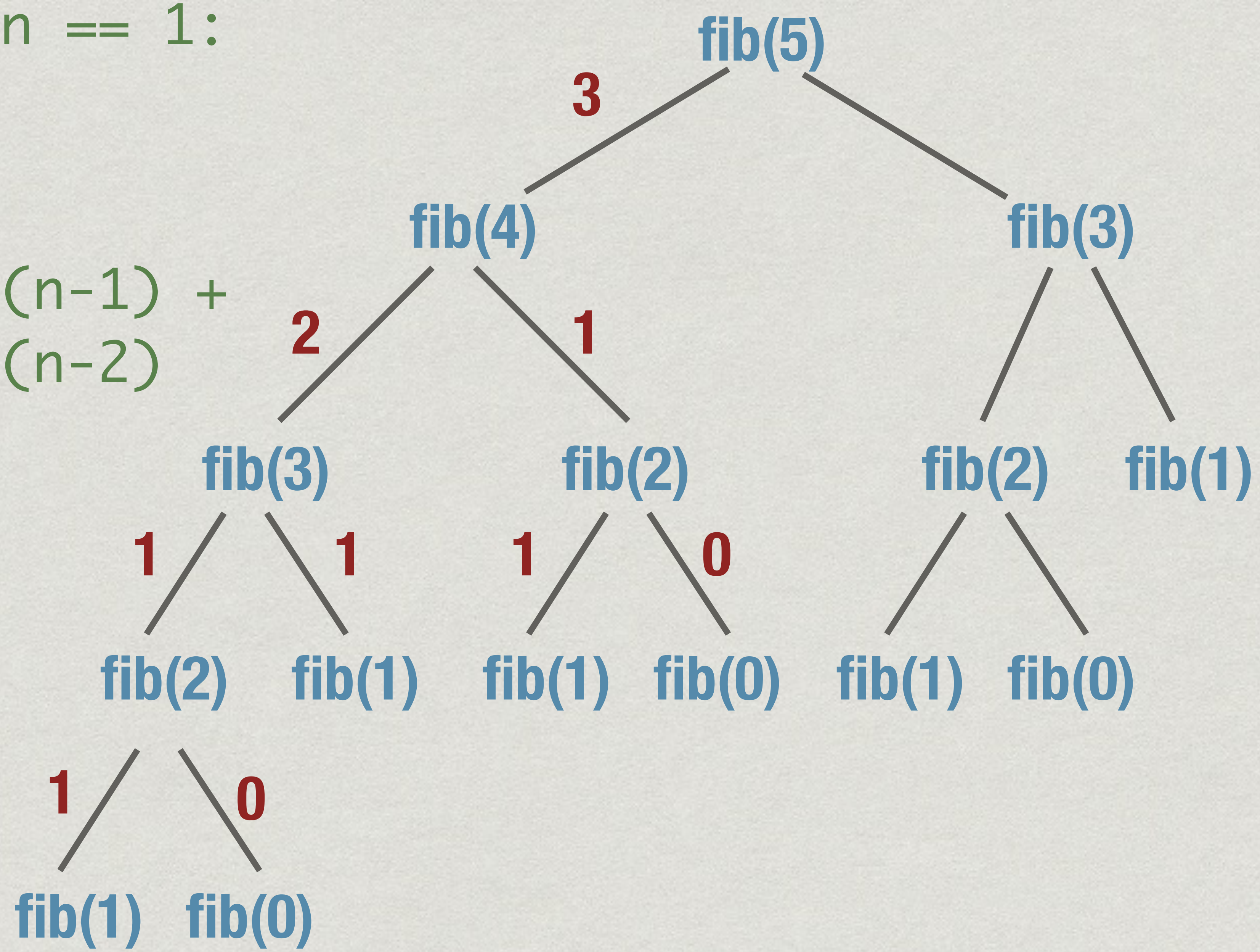
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



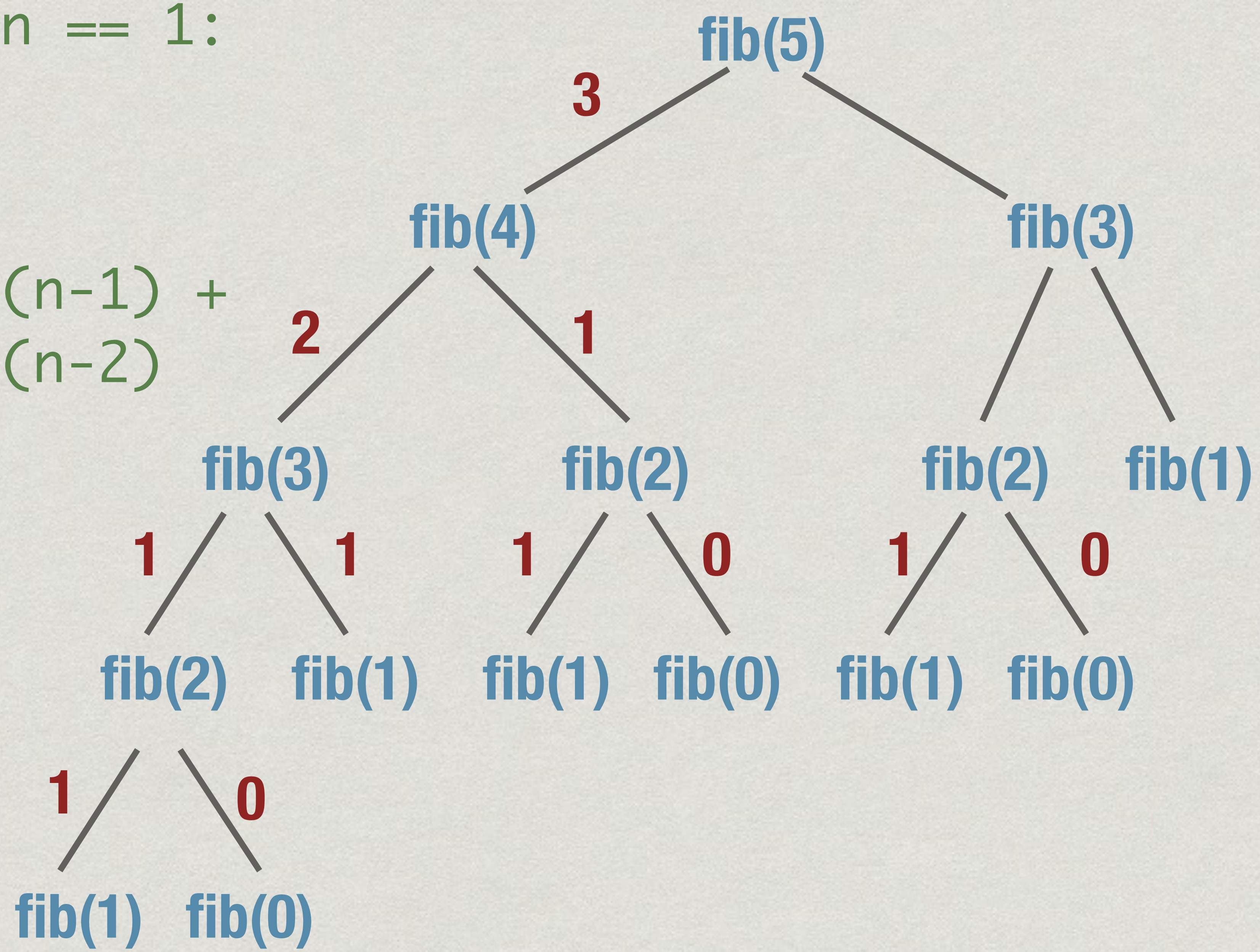
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



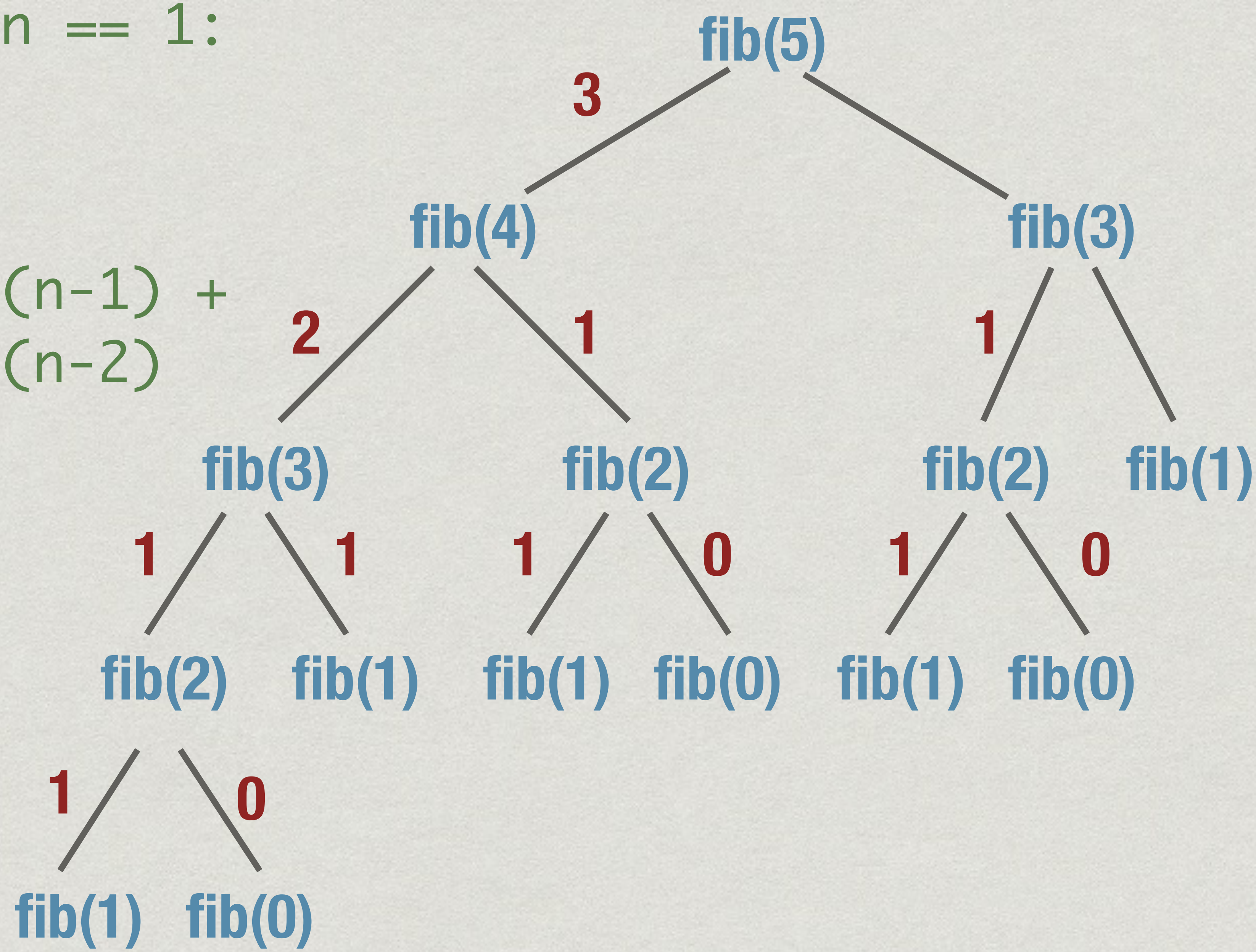
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



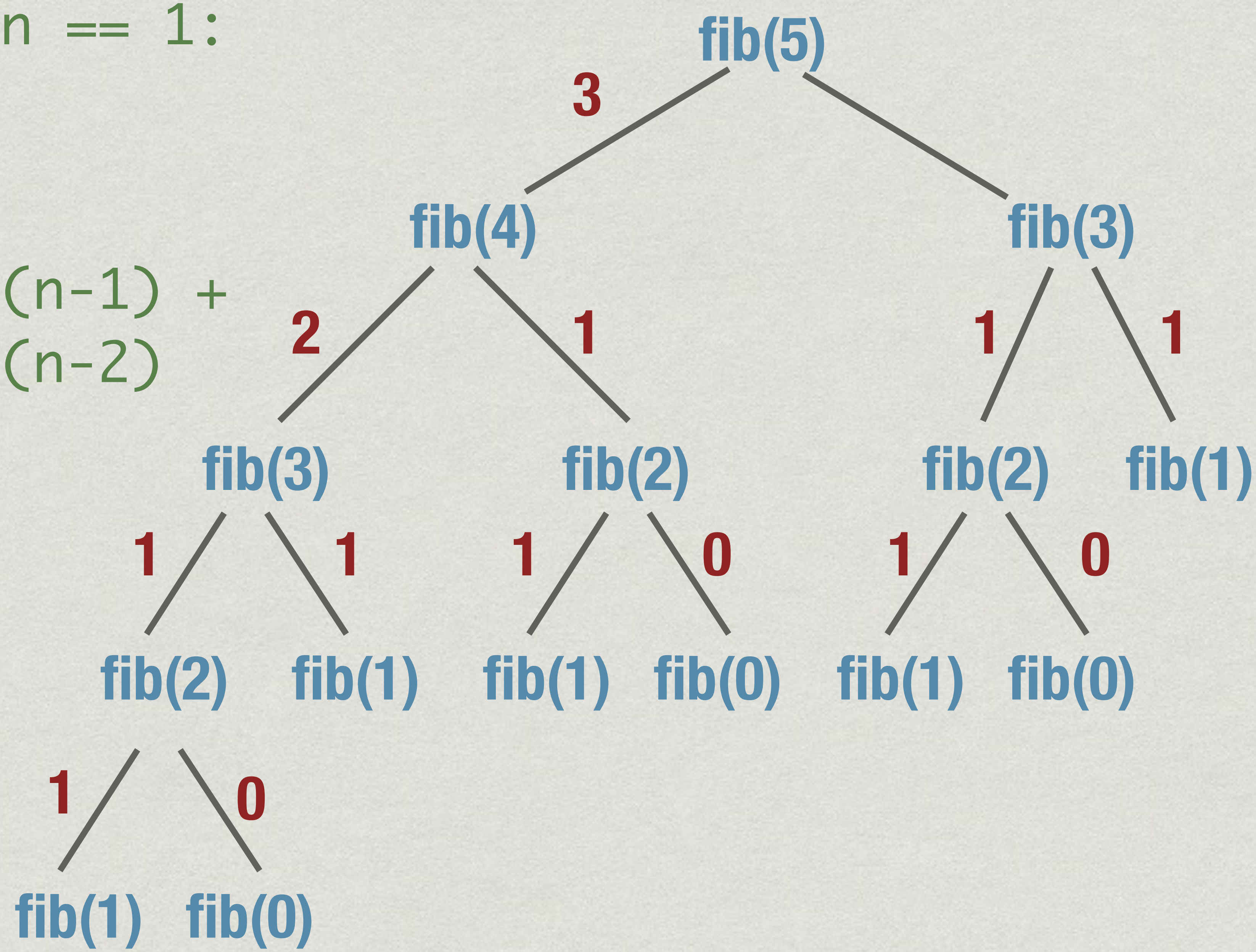
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



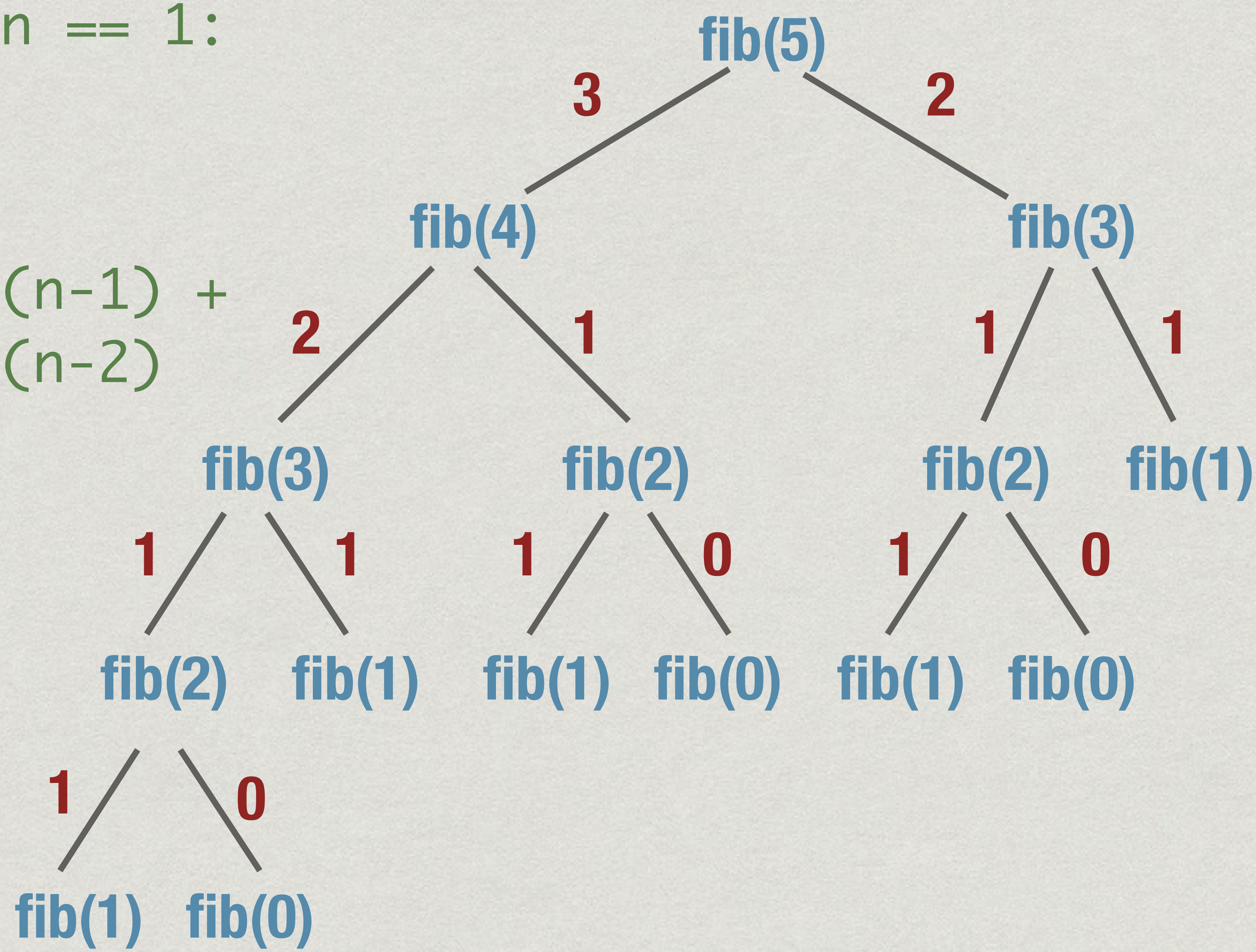
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



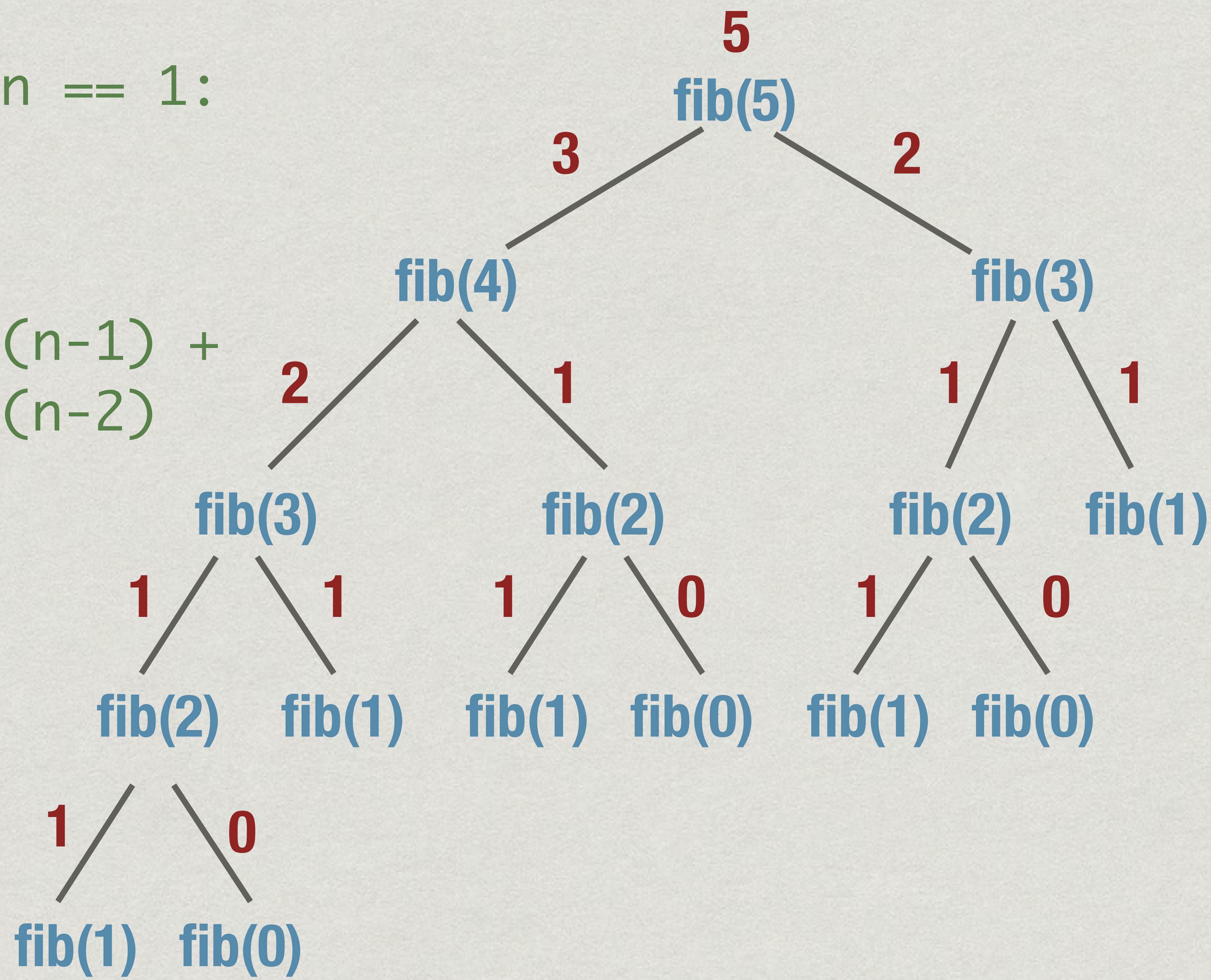
Computing fib(5)

```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Computing fib(5)

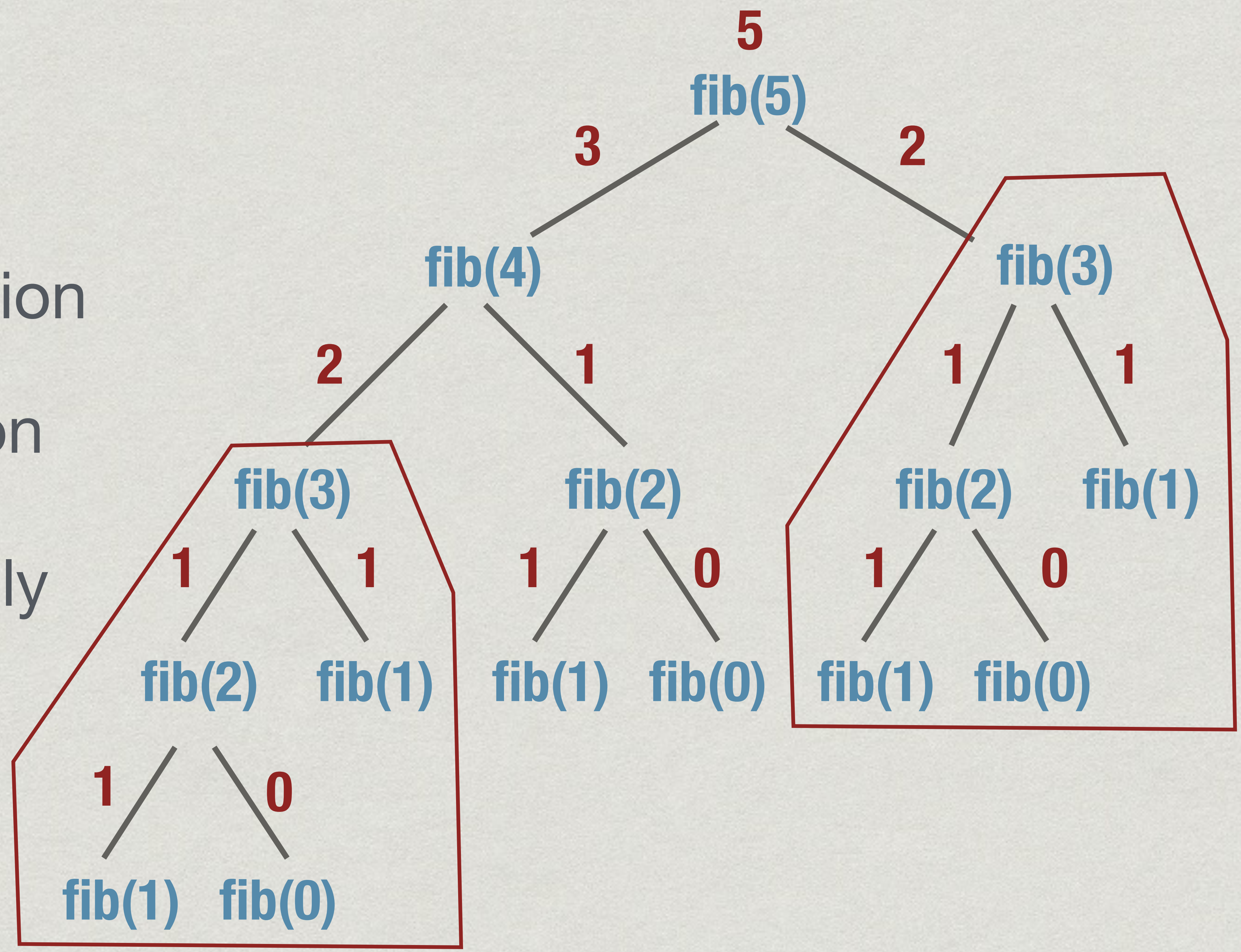
```
def fib(n):
  if n == 0 or n == 1:
    value = n
  else:
    value = fib(n-1) +
            fib(n-2)
  return(value)
```



Computing fib(5)

Overlapping subproblems

- Wasteful recomputation
- Computation tree grows exponentially



Never re-evaluate a subproblem

- Build a table of values already computed
    - Memory table
- Memoization
    - Remind yourself that this value has already been seen before

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

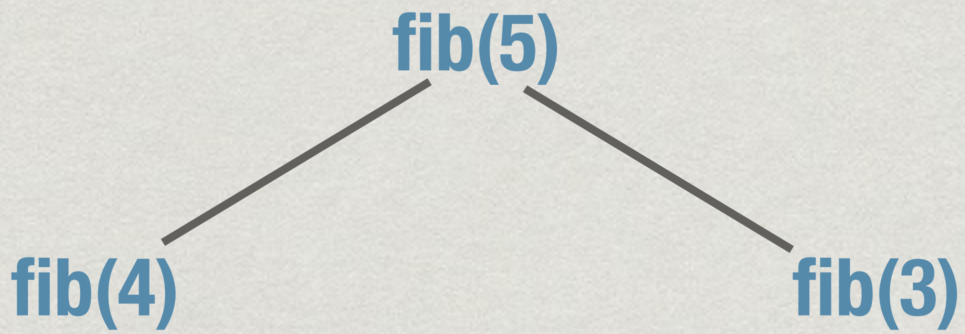
fib(5)

k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

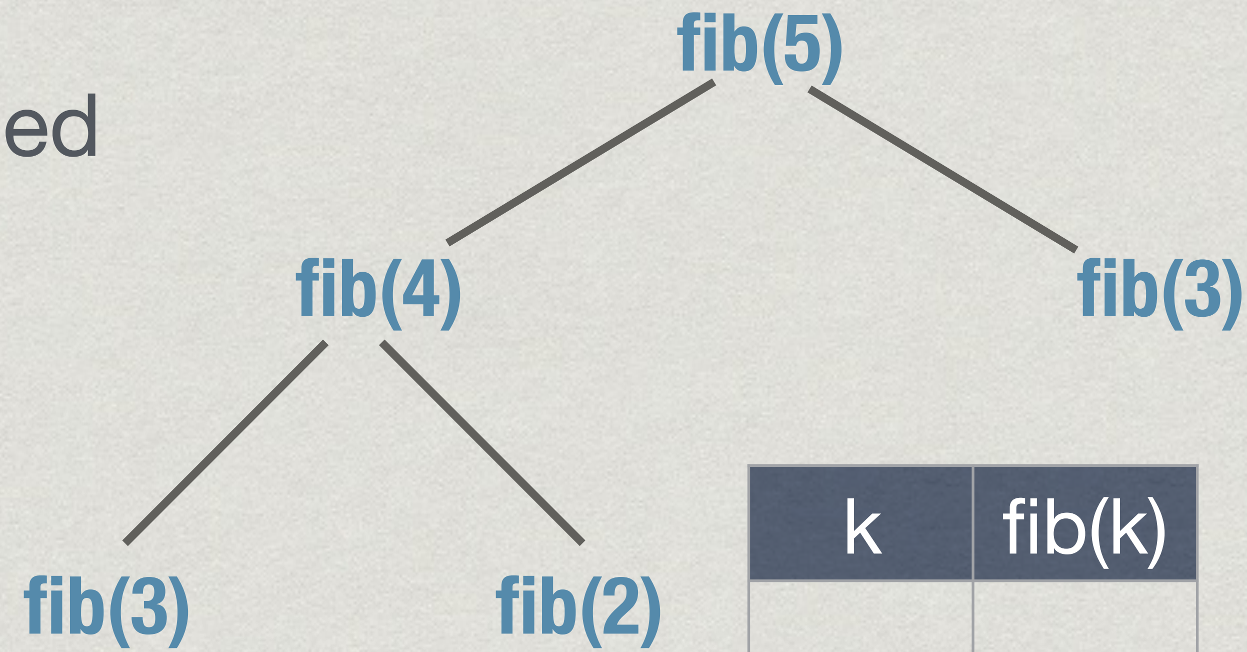


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

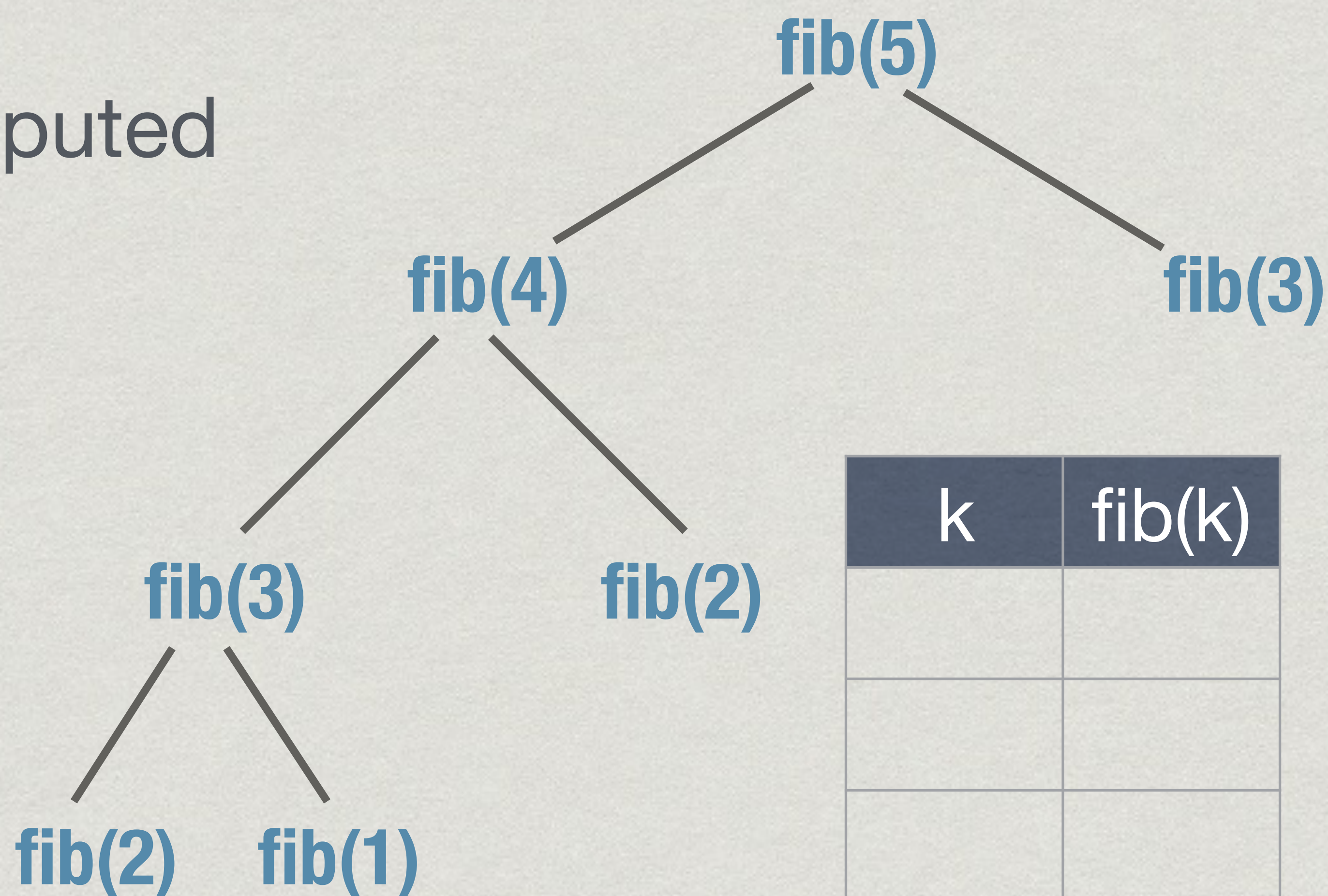


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

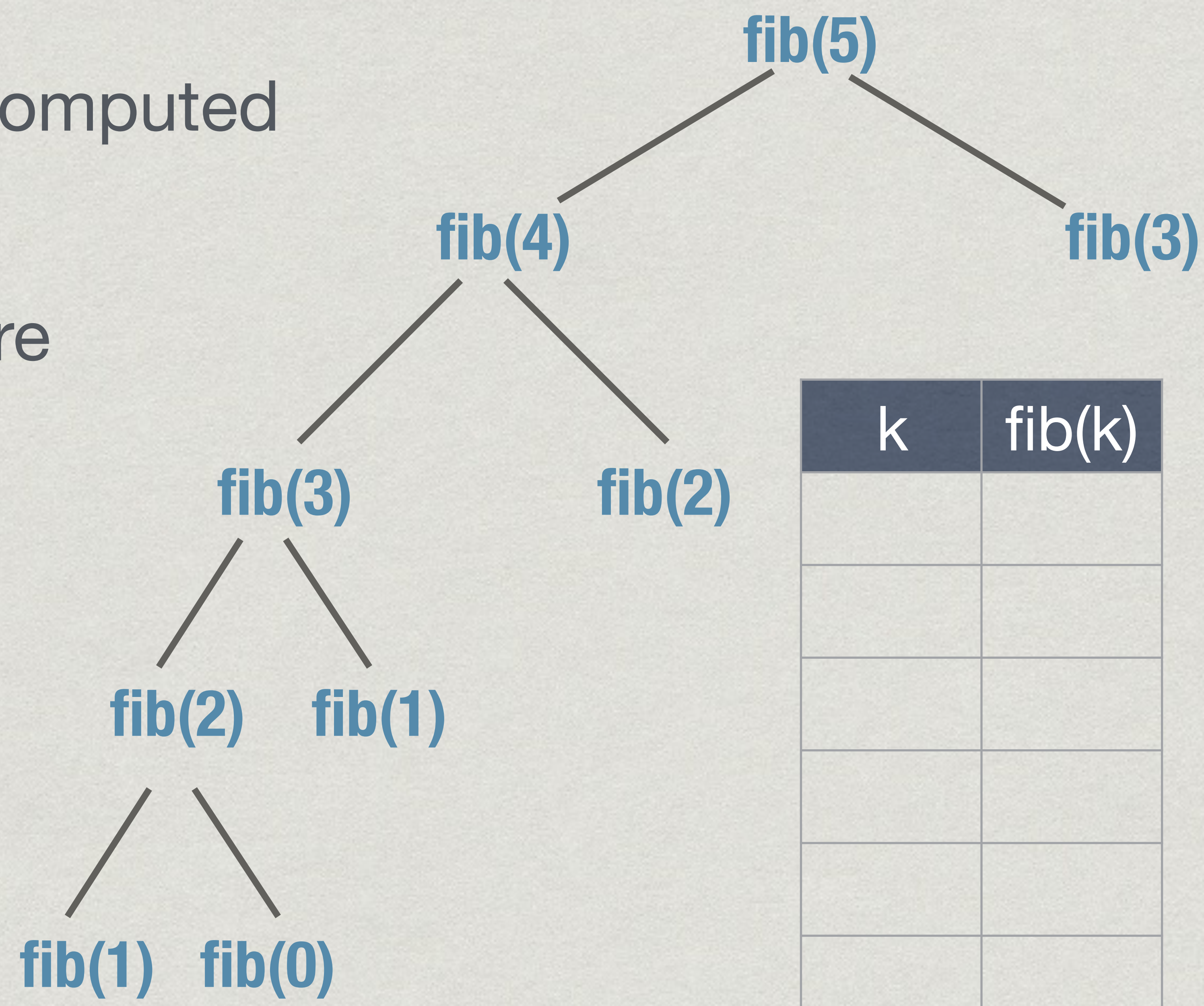


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

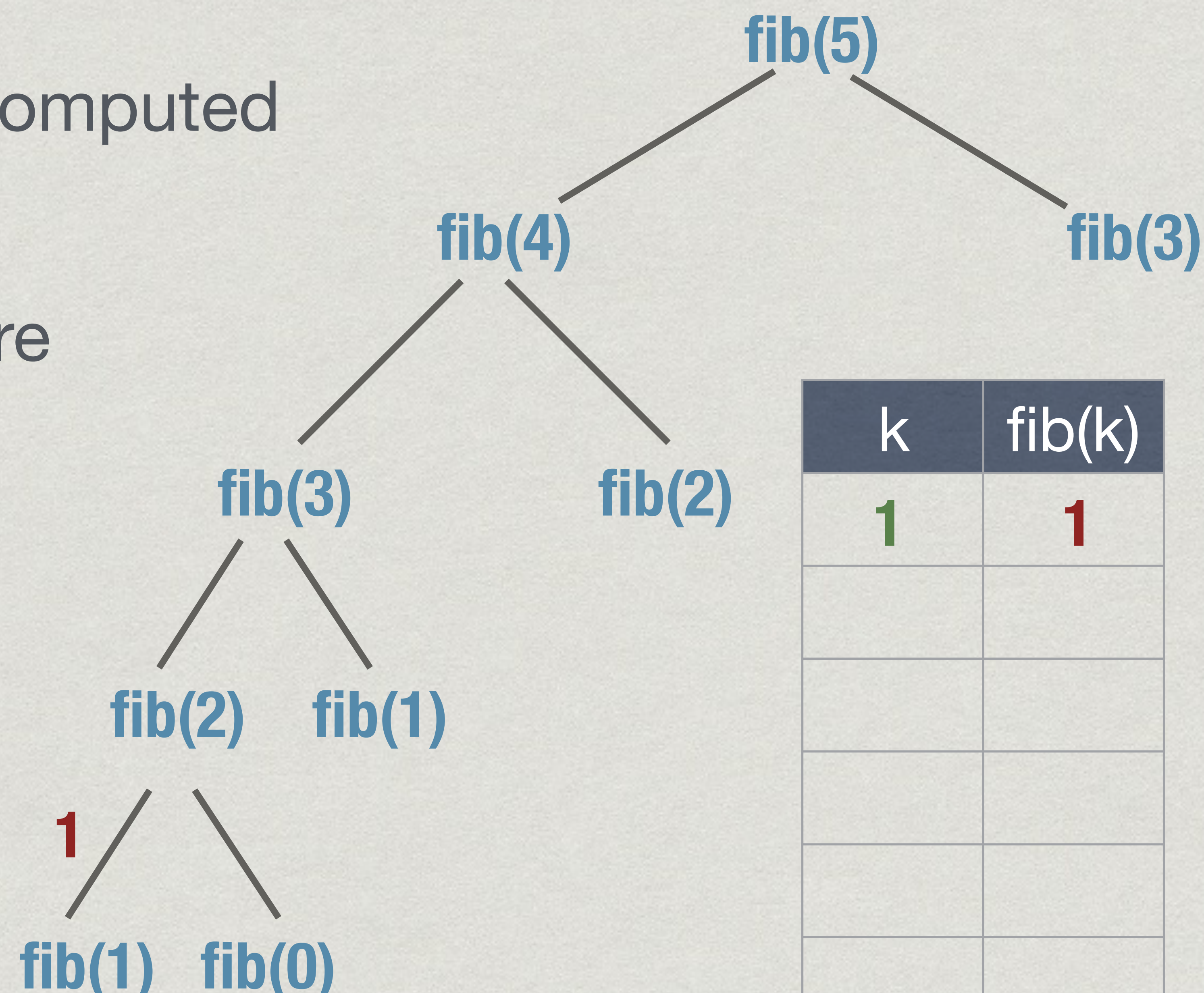


k	fib(k)

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

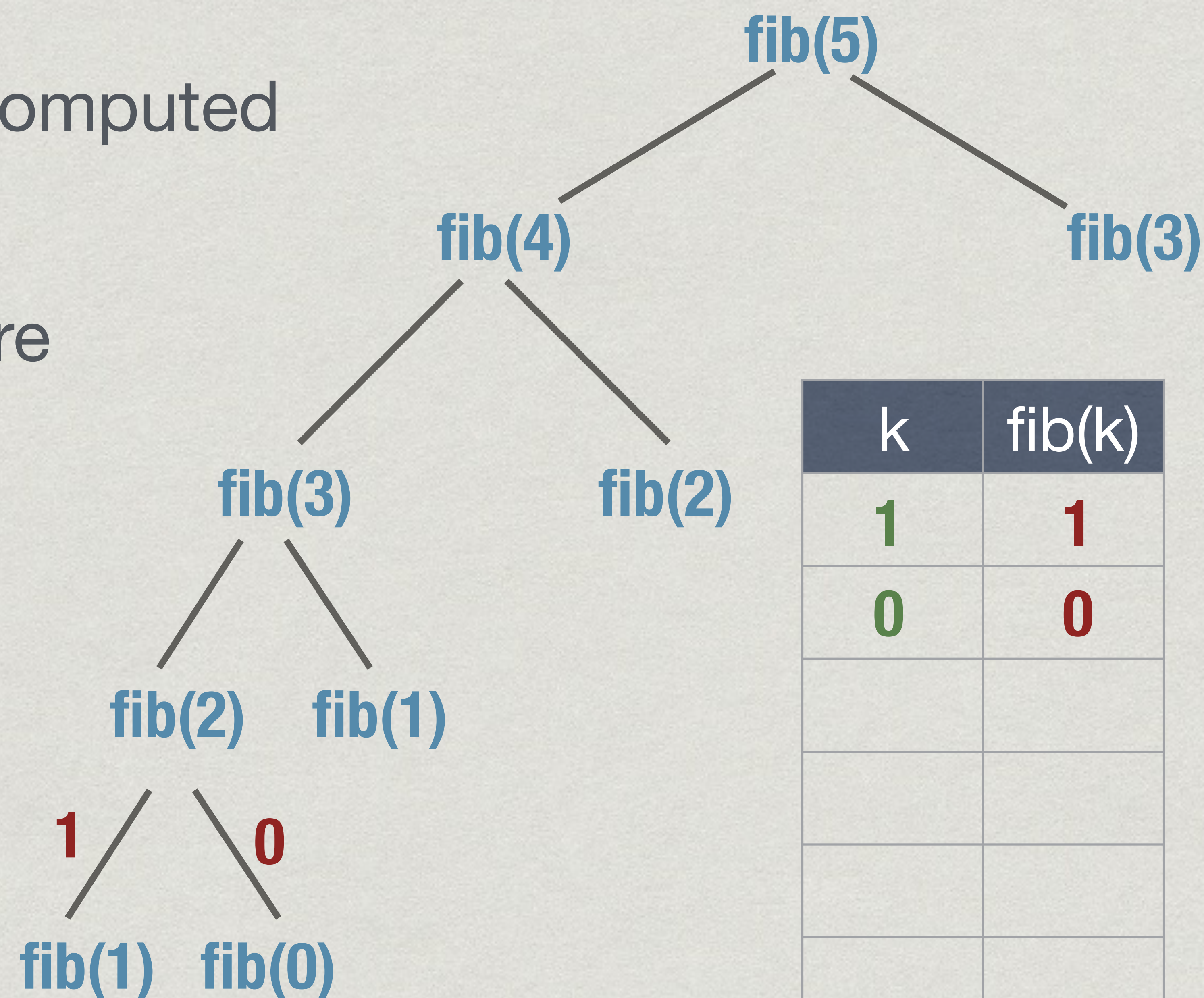


k	fib(k)
1	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

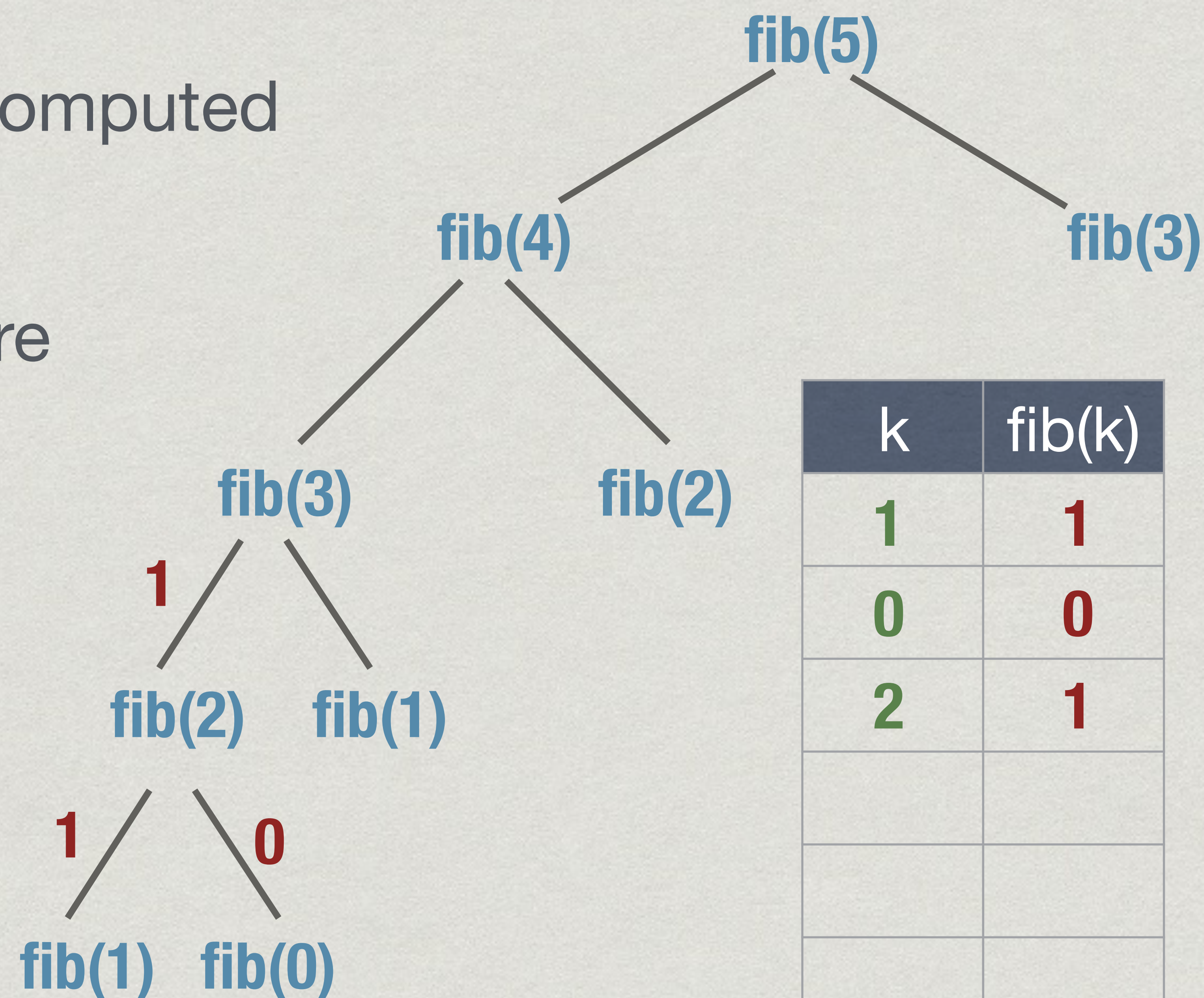


k	fib(k)
1	1
0	0

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

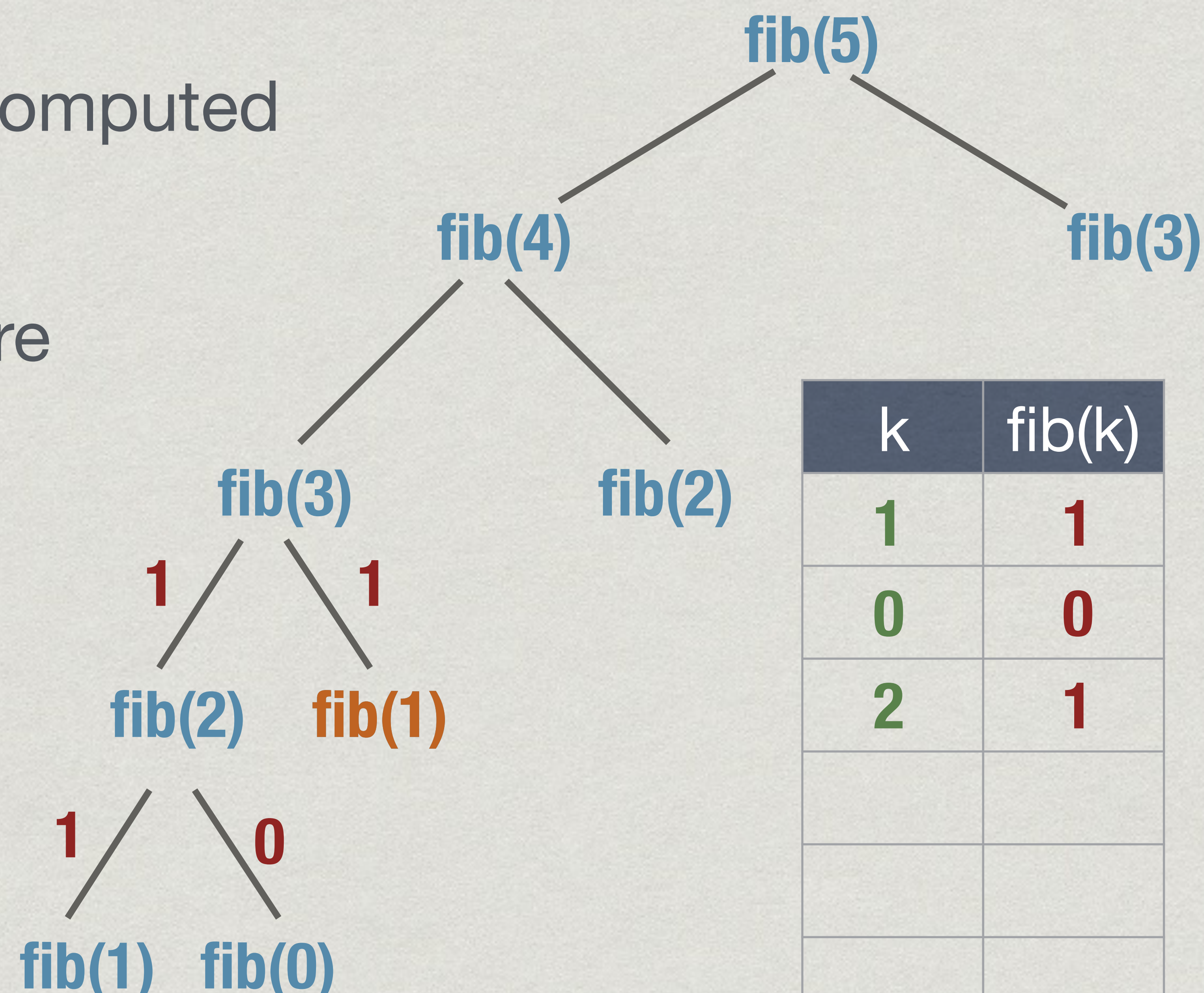


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

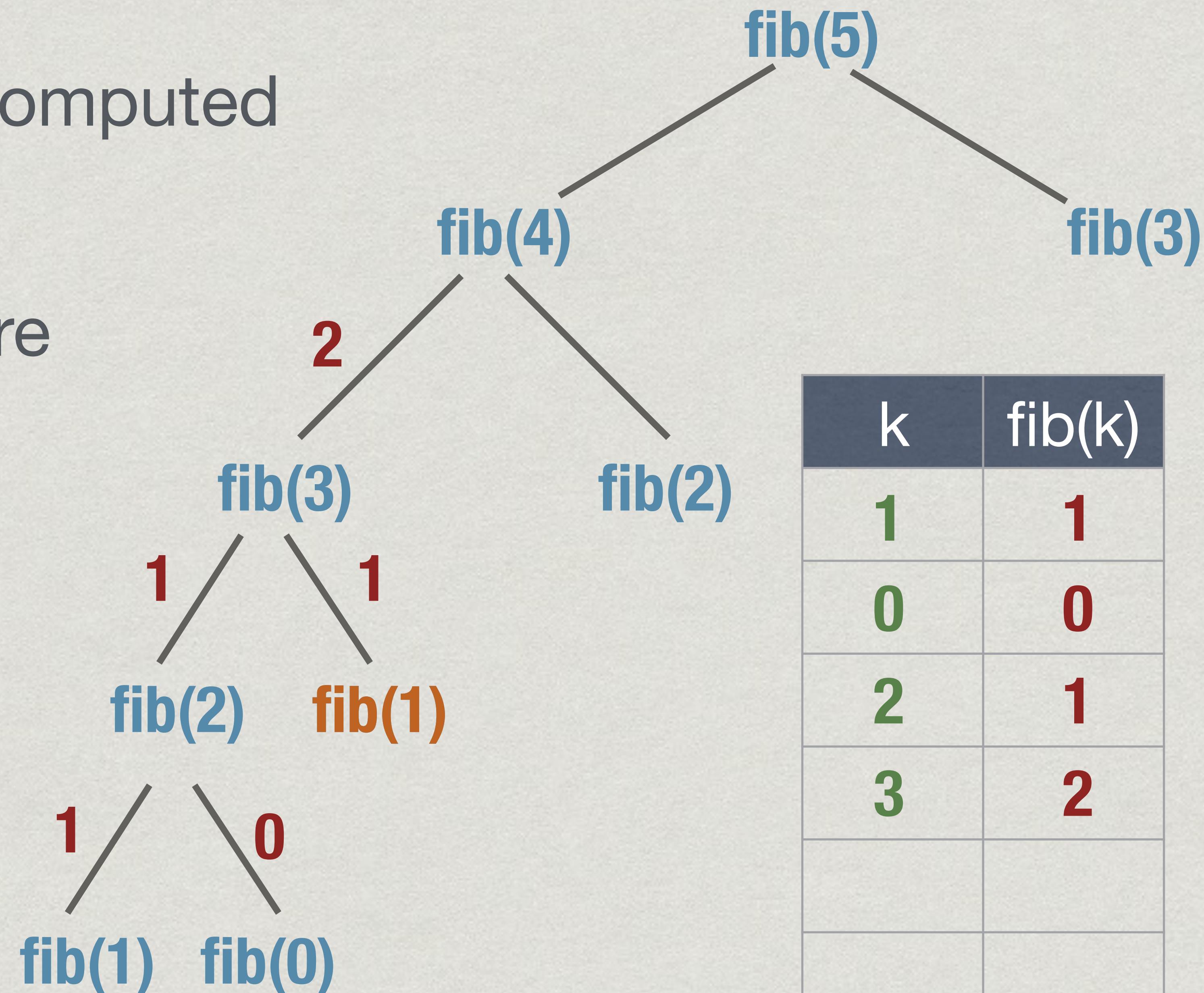


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

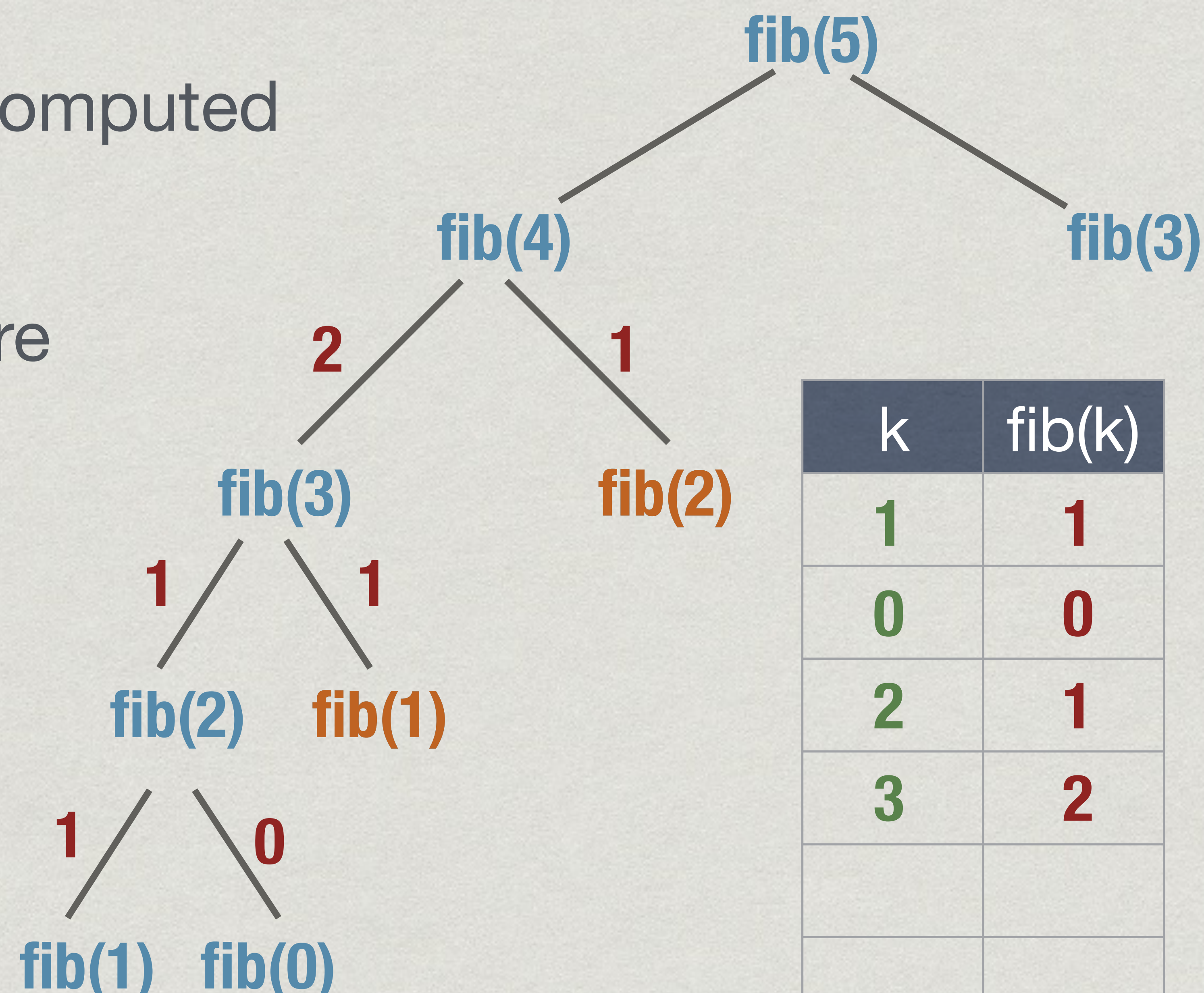


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

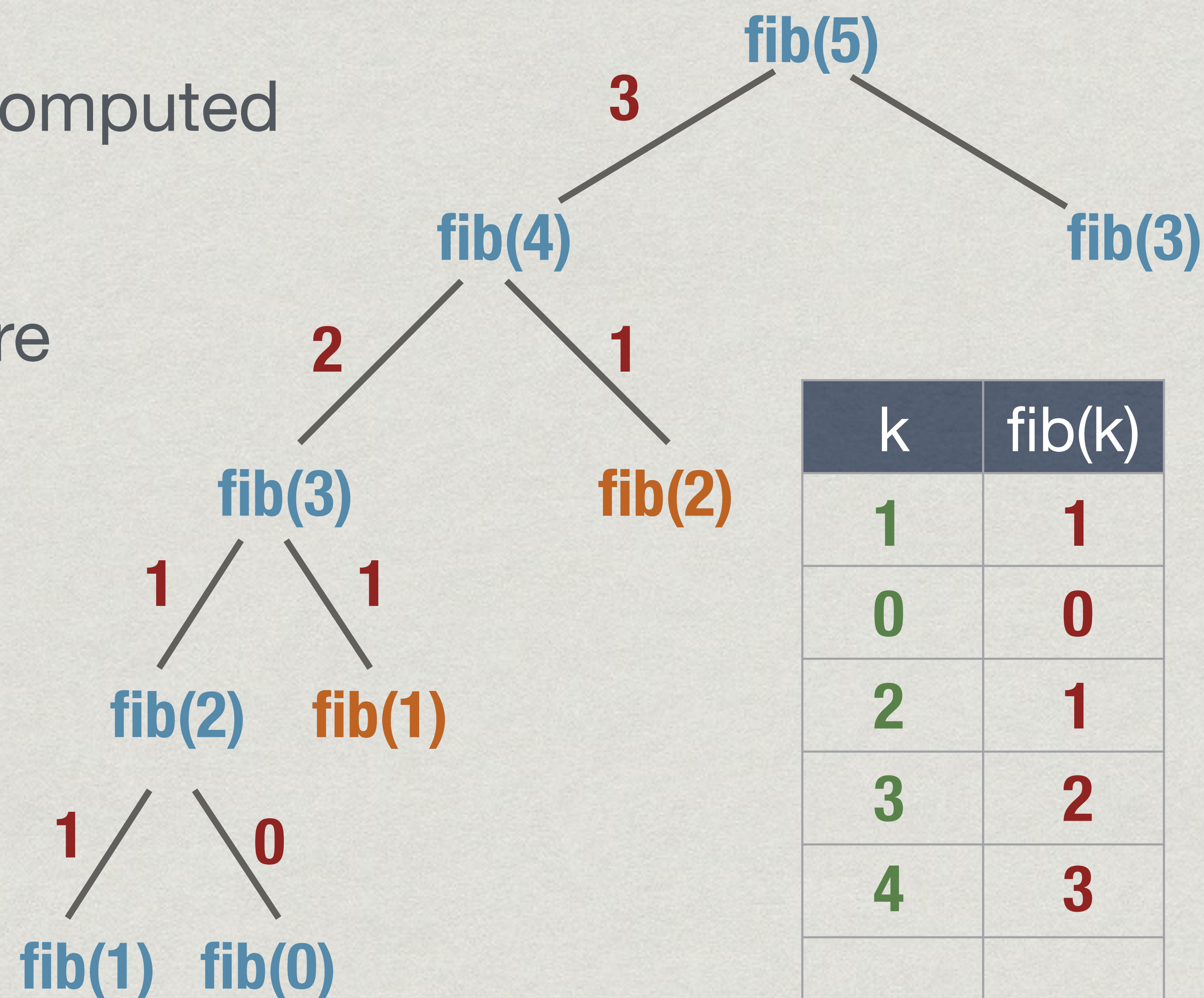


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

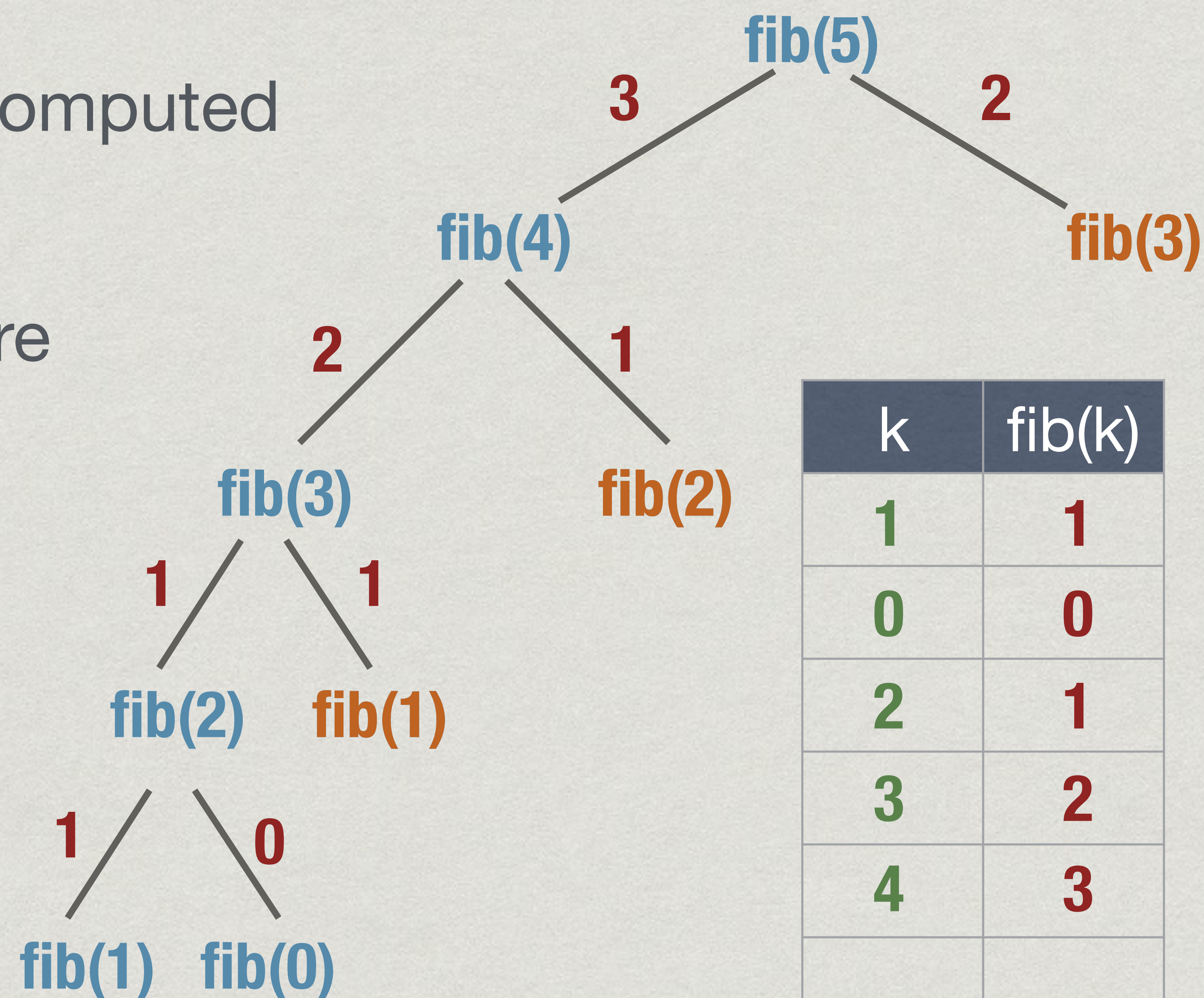


k	fib(k)
1	1
0	0
2	1
3	2
4	3

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear

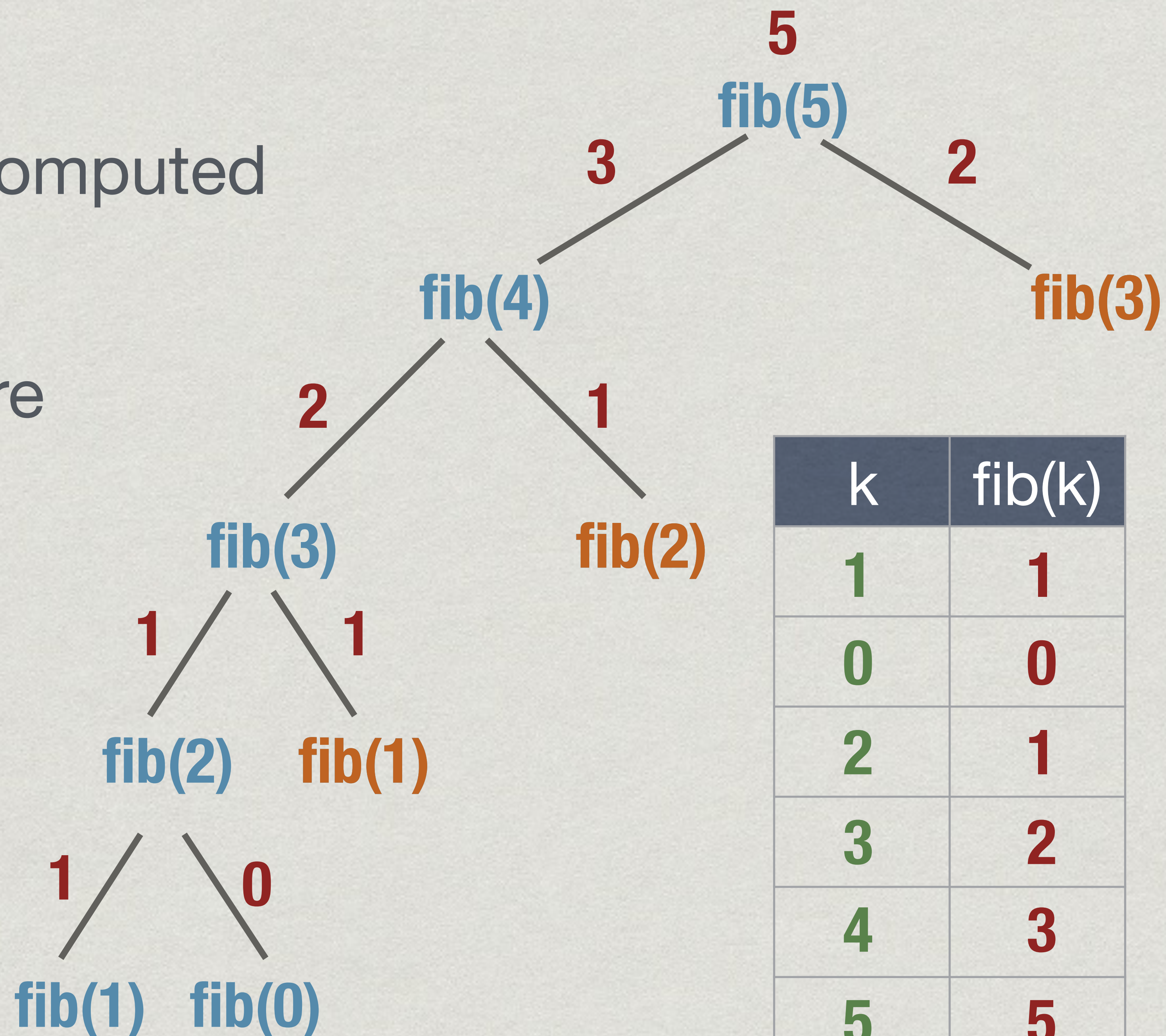


k	fib(k)
1	1
0	0
2	1
3	2
4	3

Memoized fib(5)

Memoization

- Store each newly computed value in a table
- Look up table before starting a recursive computation
- Computation tree is linear



k	fib(k)
1	1
0	0
2	1
3	2
4	3
5	5

Memoized fibonacci

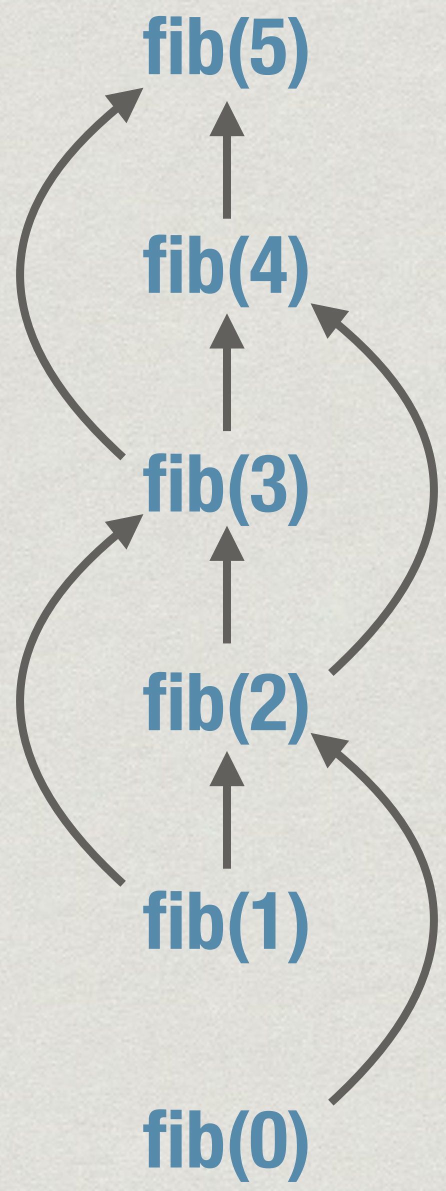
```
def fib(n):
  try:
    value = fibtable[n]   # Table is a dictionary
  except KeyError:
    if n == 0 or n == 1:
      value = n
    else:
      value = fib(n-1) + fib(n-2)
    fibtable[n] = value
  else:
    return(value)
```

In general

```
function f(x,y,z):
  try:
    value = ftable[x][y][z]
  except KeyError:
    value = expression in terms of
           subproblems
    ftable[x][y][z] = value
  else:
    return(value)
```

Dynamic programming

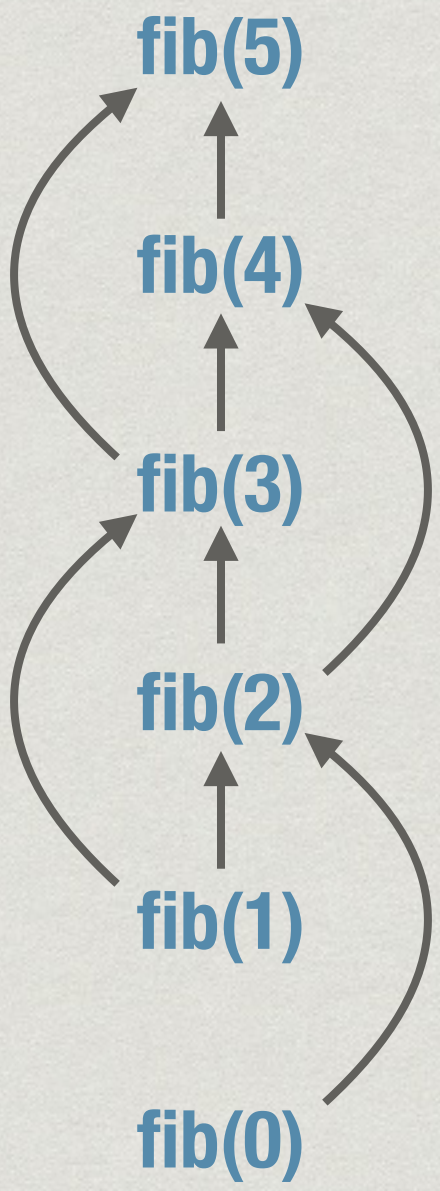
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic



Dynamic programming

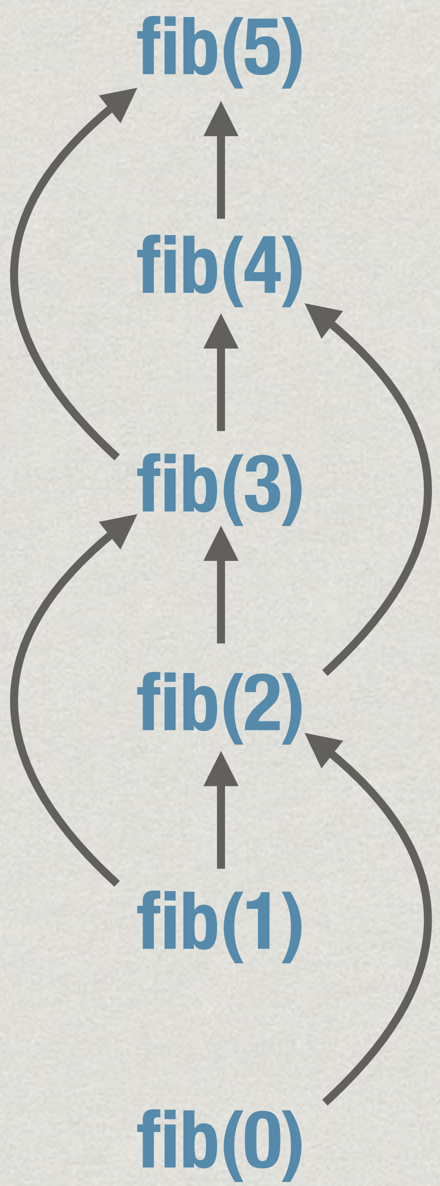
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic

k	0	1	2	3	4	5
fib(k)						



Dynamic programming

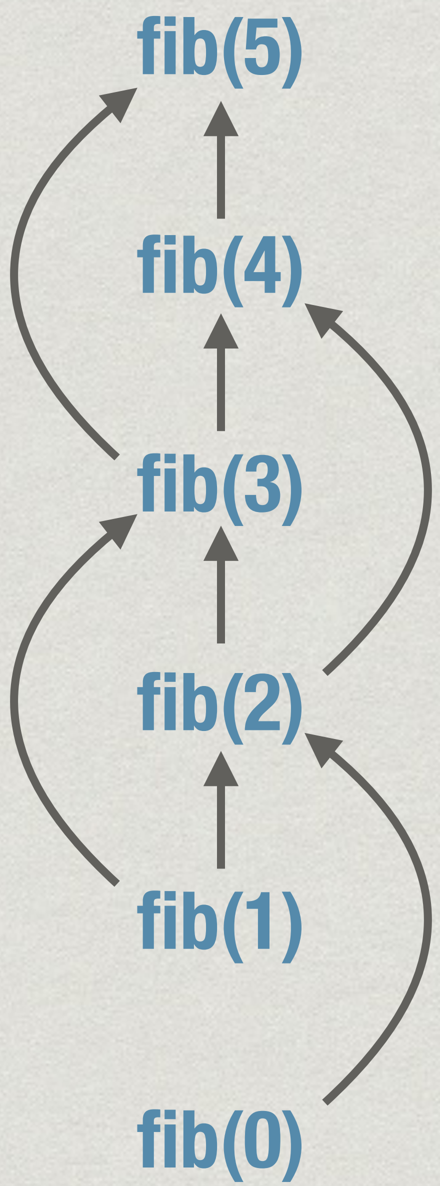
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic



k	0	1	2	3	4	5
fib(k)	0					

Dynamic programming

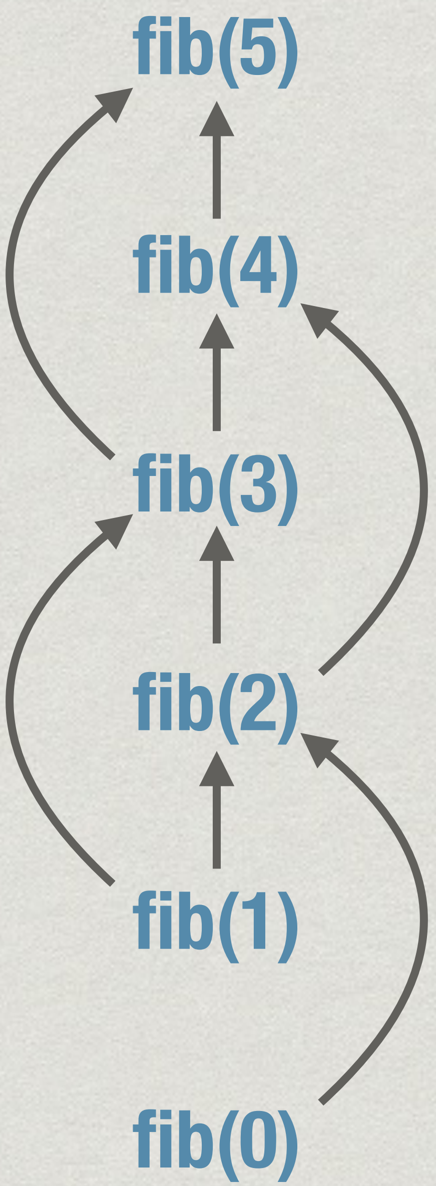
- Anticipate what the memory table looks like
    - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
    - Must be acyclic



k	0	1	2	3	4	5
fib(k)	**0**	**1**				

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic

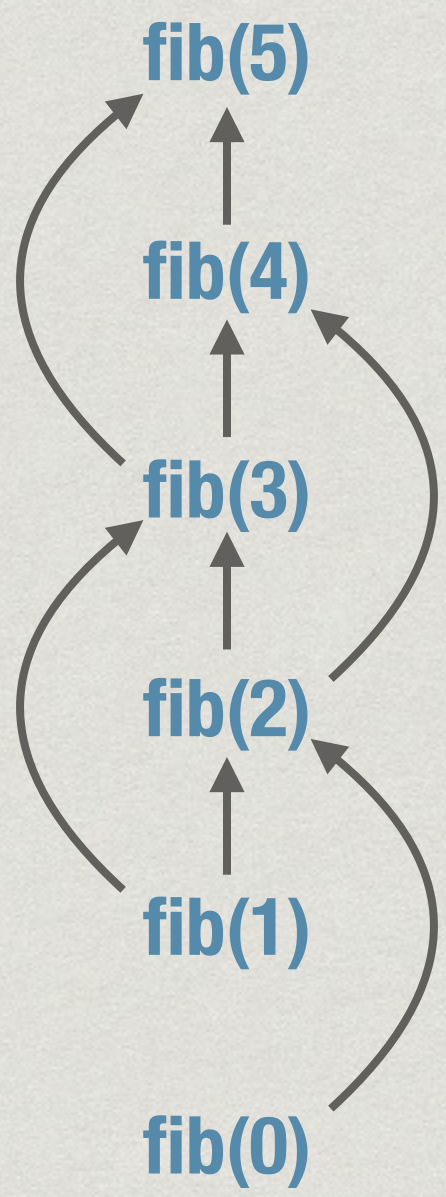


k	0	1	2	3	4	5
fib(k)	**0**	**1**	**1**			

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic

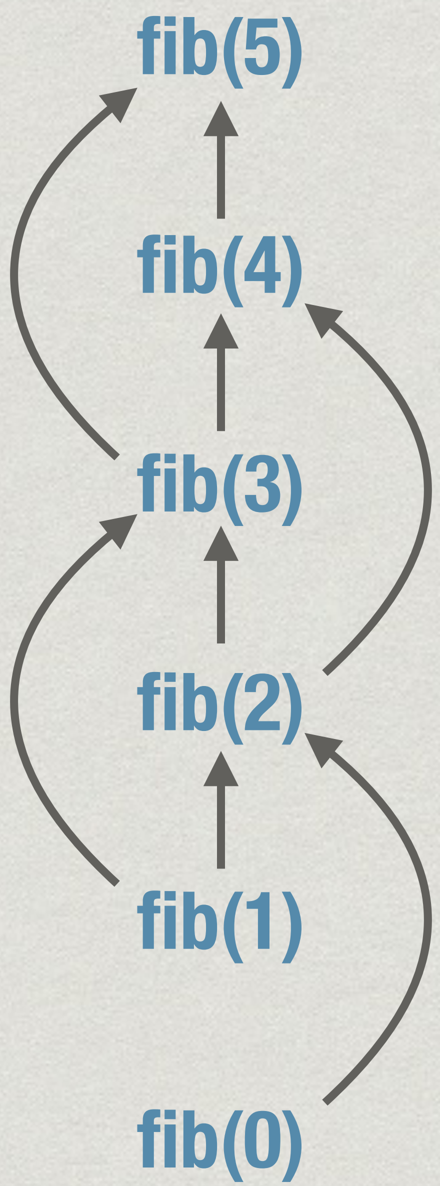
k	0	1	2	3	4	5
fib(k)	**0**	**1**	**1**	**2**		



Dynamic programming

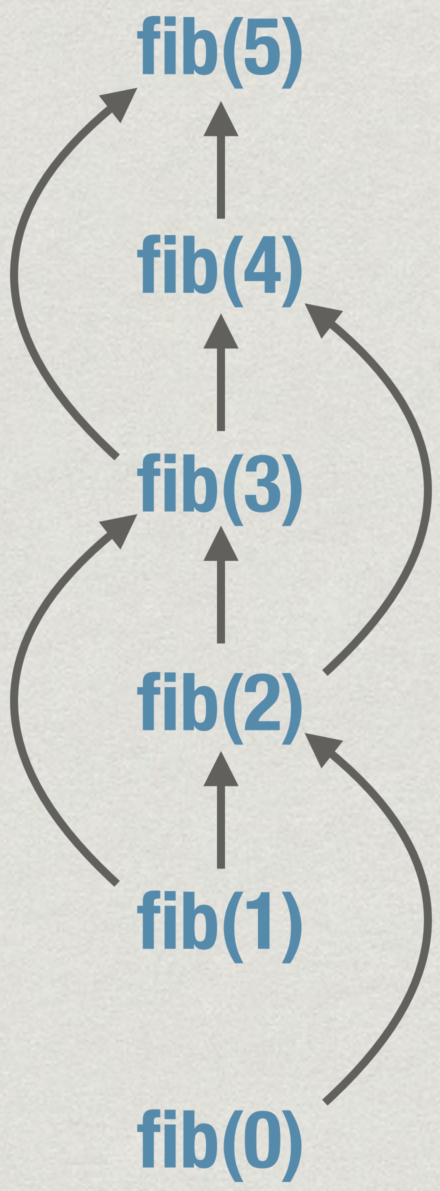
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic

k	0	1	2	3	4	5
fib(k)	**0**	**1**	**1**	**2**	**3**	



Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
- Solve subproblems in order of dependencies
  - Must be acyclic



k	0	1	2	3	4	5
fib(k)	**0**	**1**	**1**	**2**	**3**	**5**

Dynamic programming fibonacci

```
def fib(n):
  fibtable[0] = 0
  fibtable[1] = 1
  for i in range(2,n+1):
    fibtable[i] = fibtable[i-1] +
                  fibtable[i-2]

  return(fibtable[n])
```

Summary

Memoization

- Store values of subproblems in a table
- Look up the table before making a recursive call

Dynamic programming:

- Solve subproblems in order of dependency
  - Dependencies must be acyclic
- Iterative evaluation