NPTEL MOOC,JAN-FEB 2015
Week 7, Module 2

DESIGN AND ANALYSIS OF ALGORITHMS

Memoization

MADHAVAN MUKUND, CHENNAI MATHEMATICAL INSTITUTE
http://www.cmi.ac.in/~madhavan

Inductive definitions

- Factorial
  - f(0) = 1
  - f(n) = n × f(n-1)
- Insertion sort
  - isort([ ]) = [ ]
  - isort([x_1,x_2,..,x_n]) = insert(x_1,isort([x_2,…,x_n]))

... Recursive programs

```
int 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($[x_2,\ldots,x_n]$) is a subproblem of isort($[x_1,x_2,\ldots,x_n]$)
  - So is isort($[x_i,\ldots,x_j]$) 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)

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

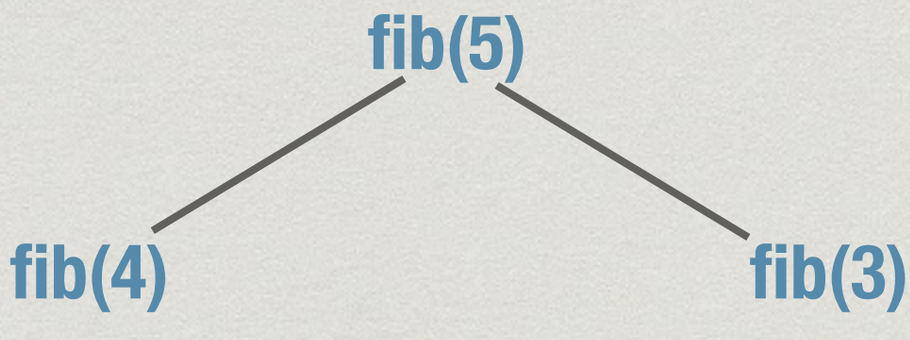
Computing fib(5)

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

fib(5)

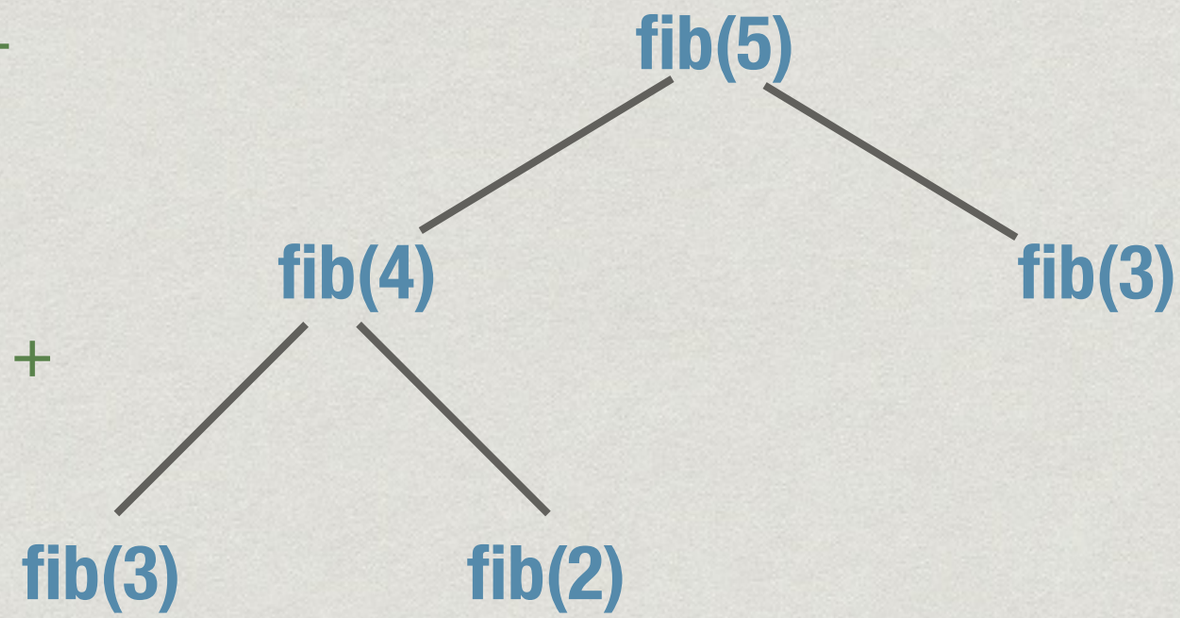
Computing fib(5)

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



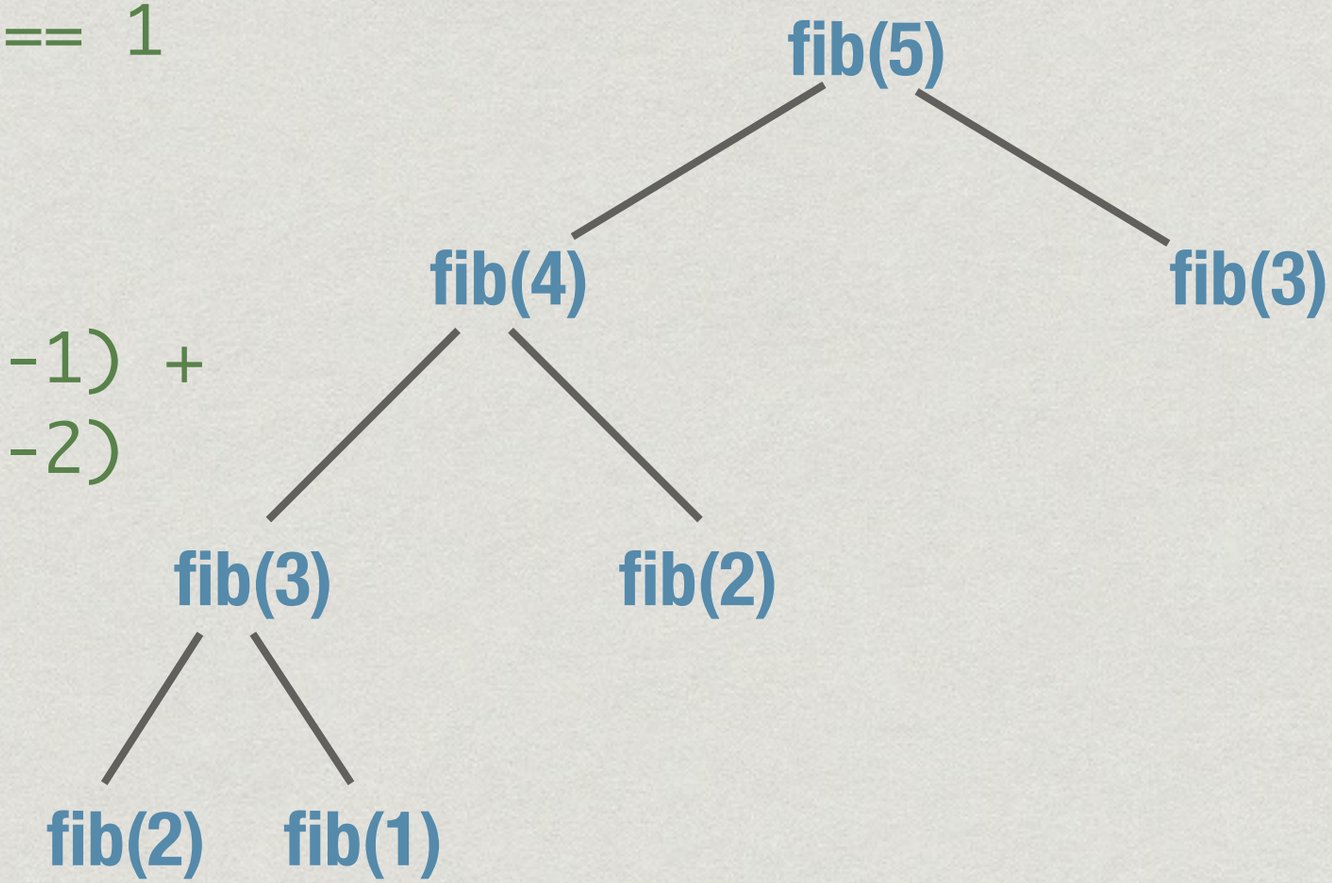
Computing fib(5)

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



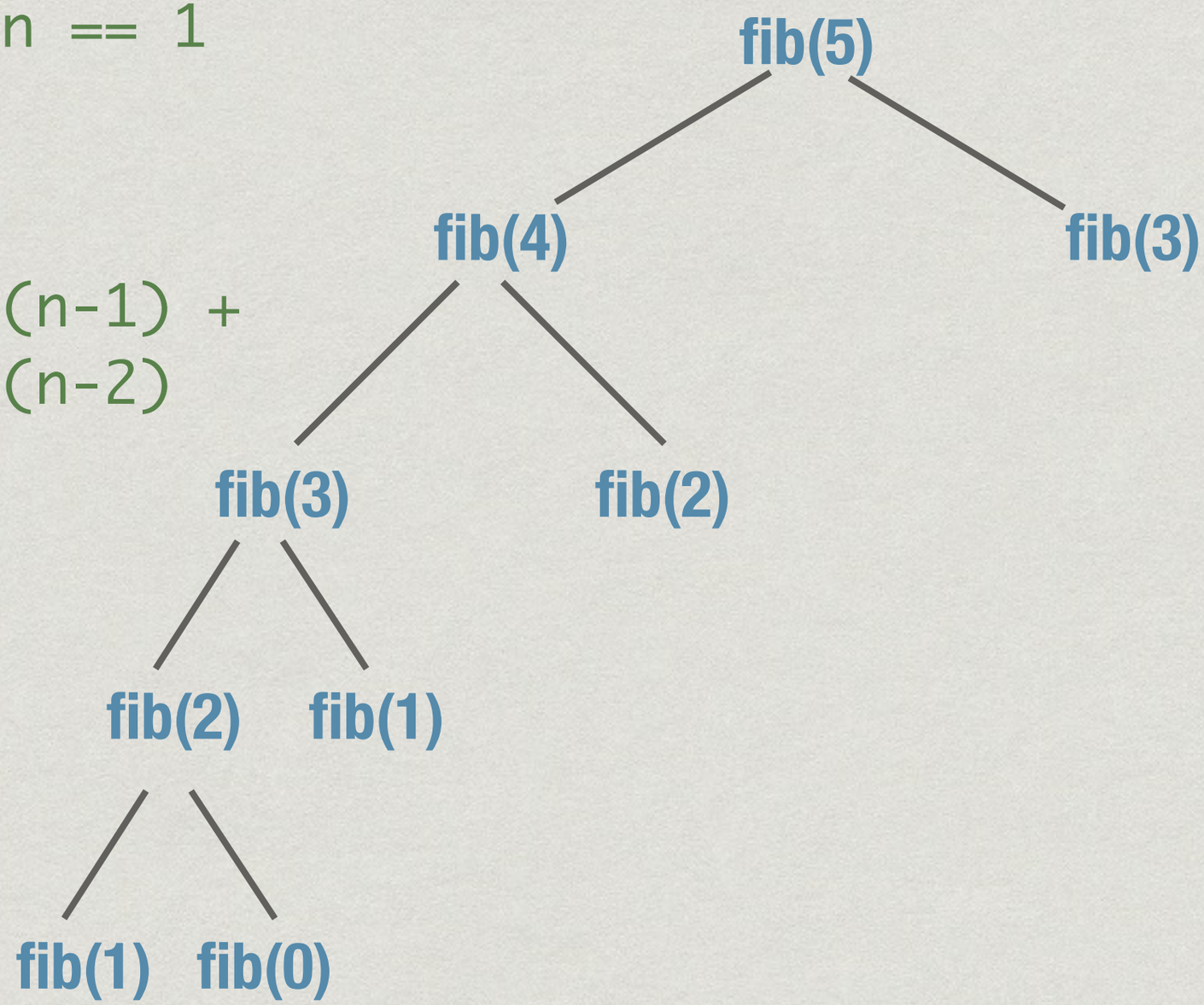
Computing fib(5)

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



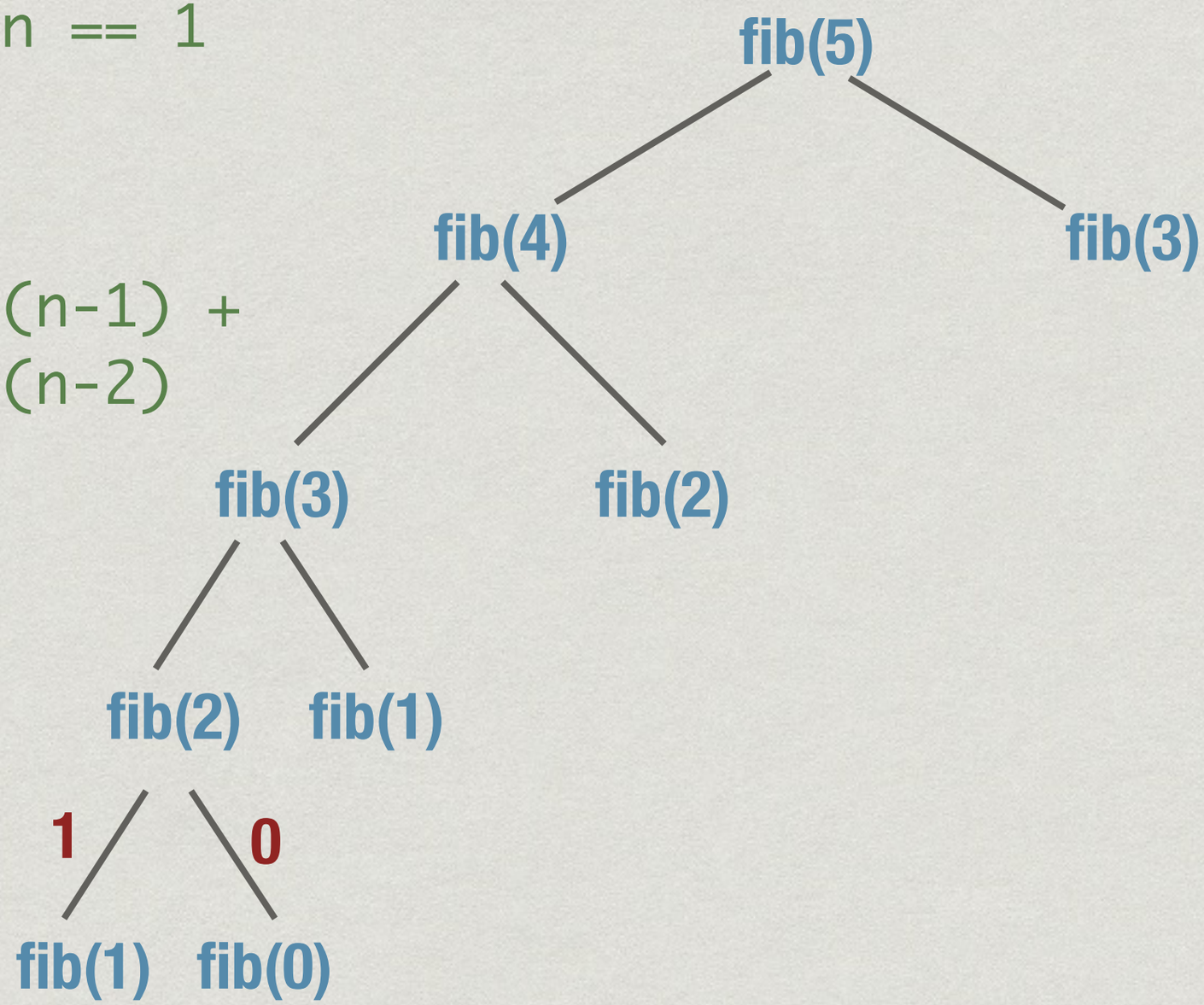
Computing fib(5)

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



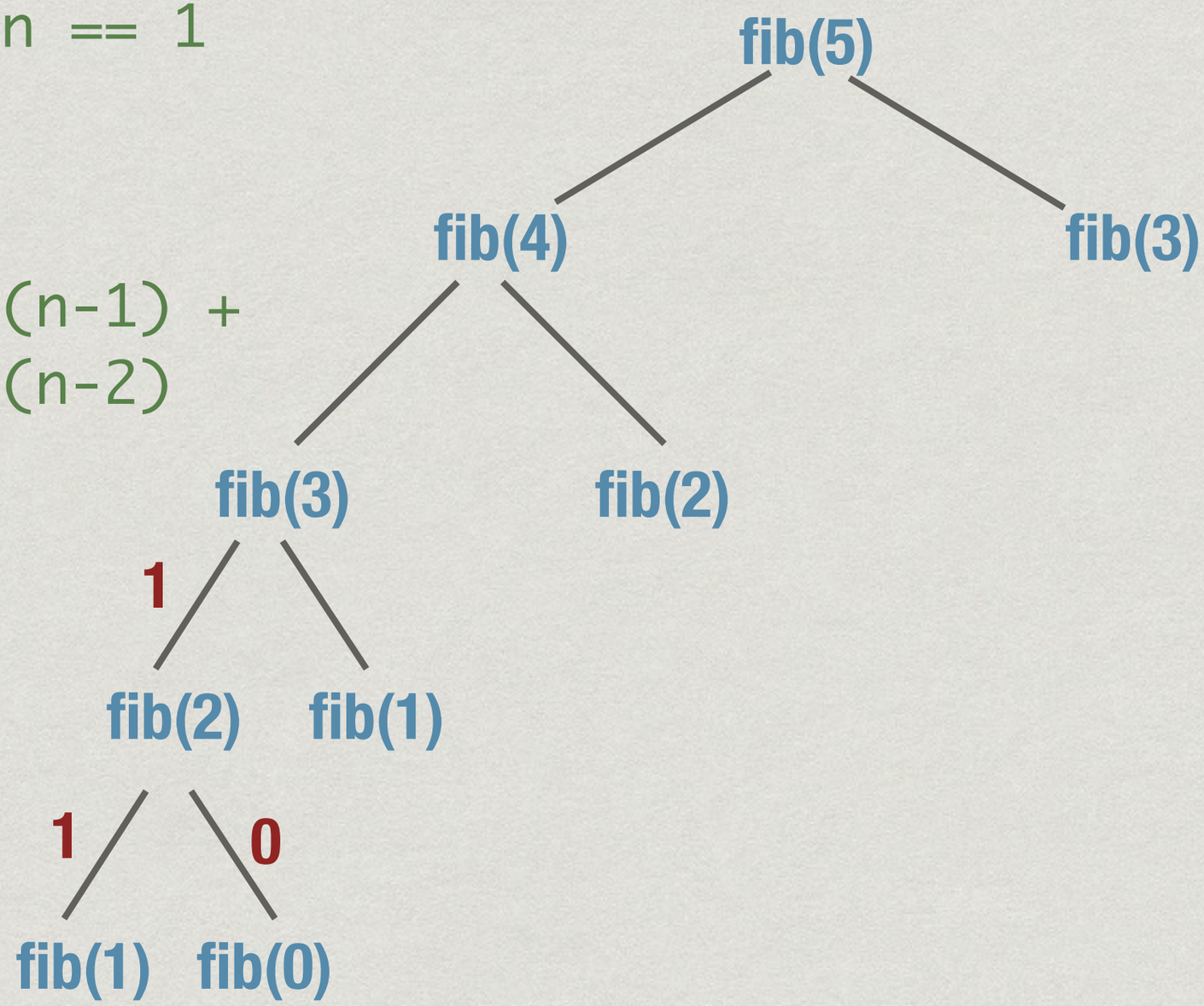
Computing fib(5)

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



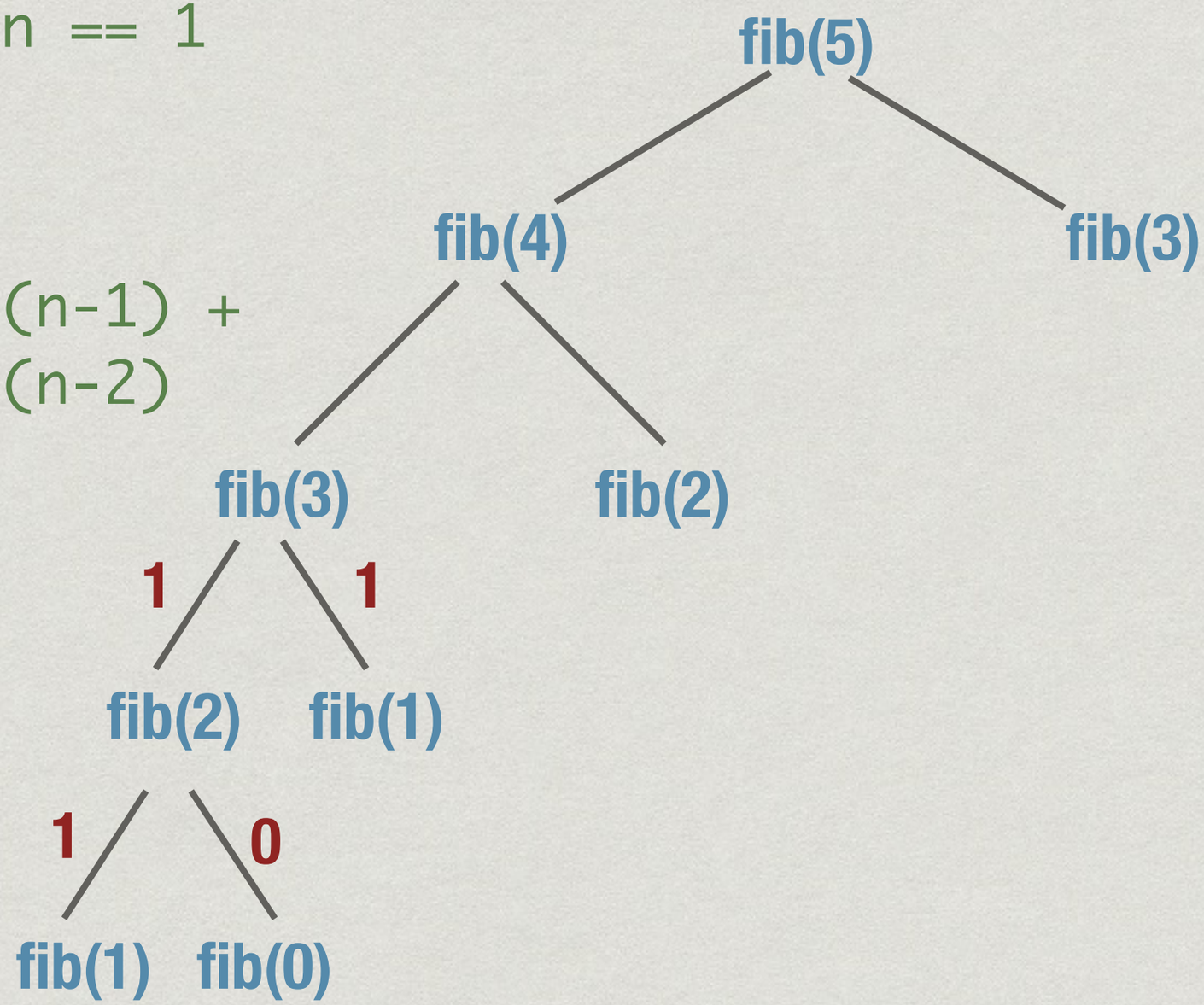
Computing fib(5)

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



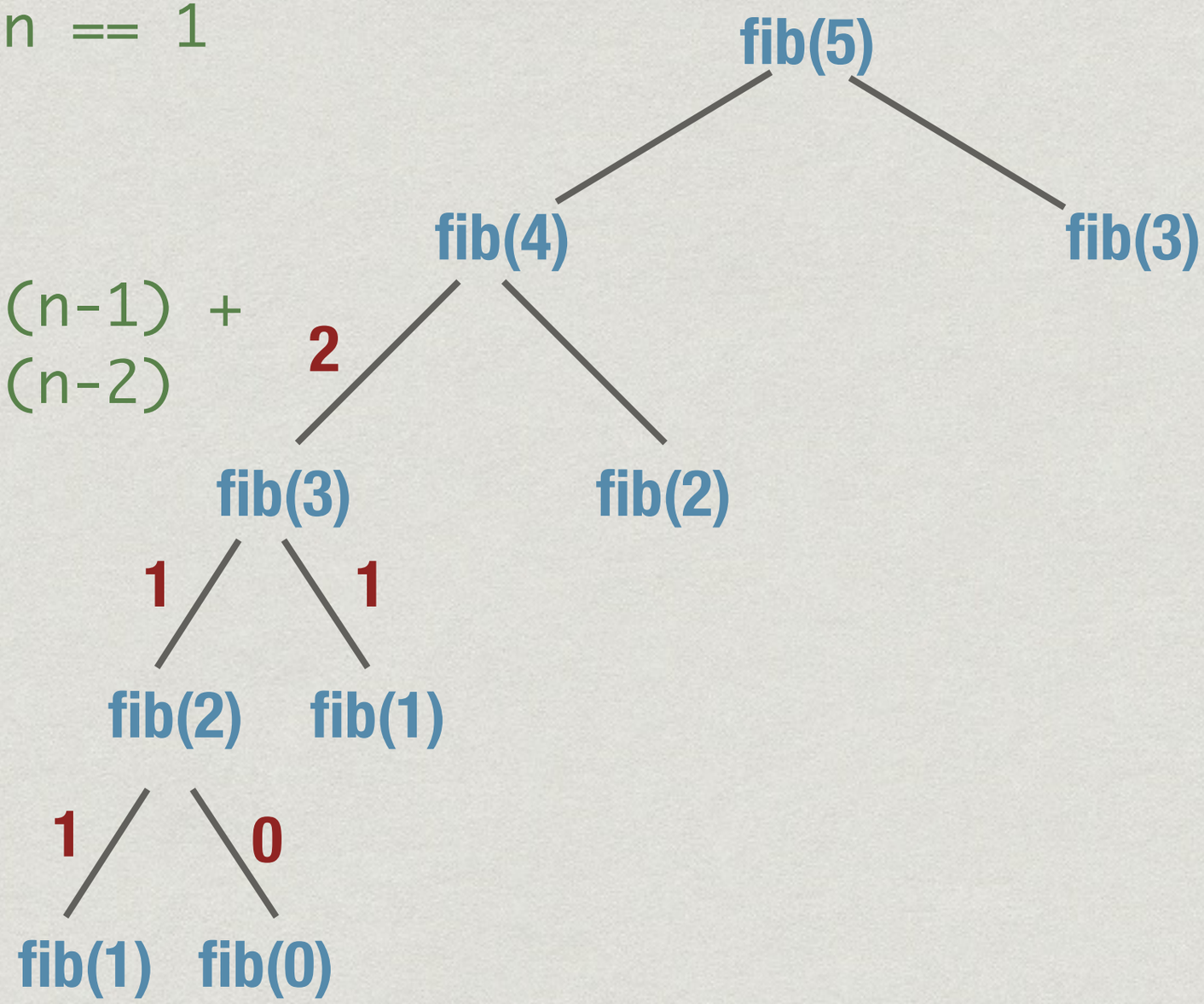
Computing fib(5)

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



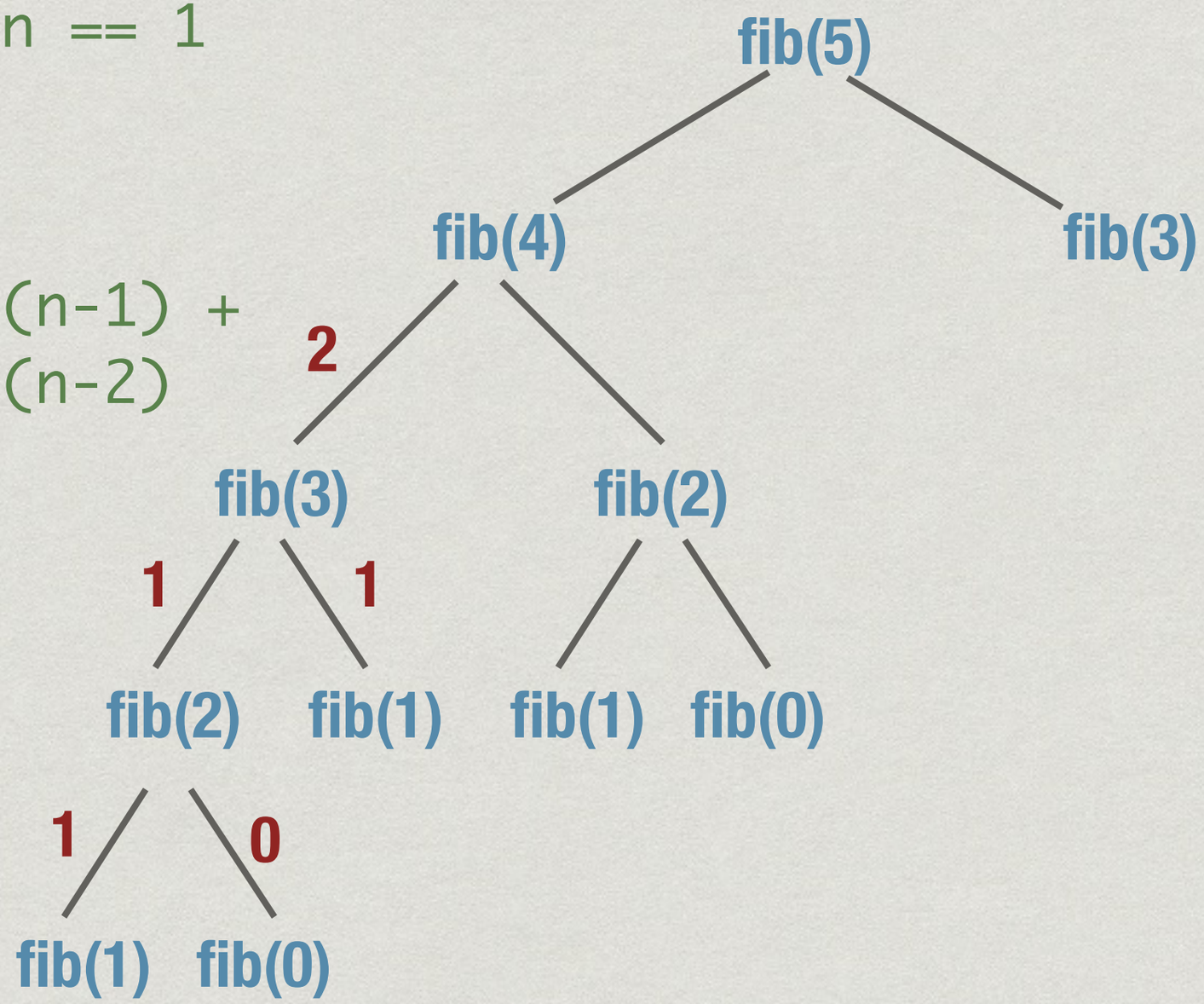
Computing fib(5)

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



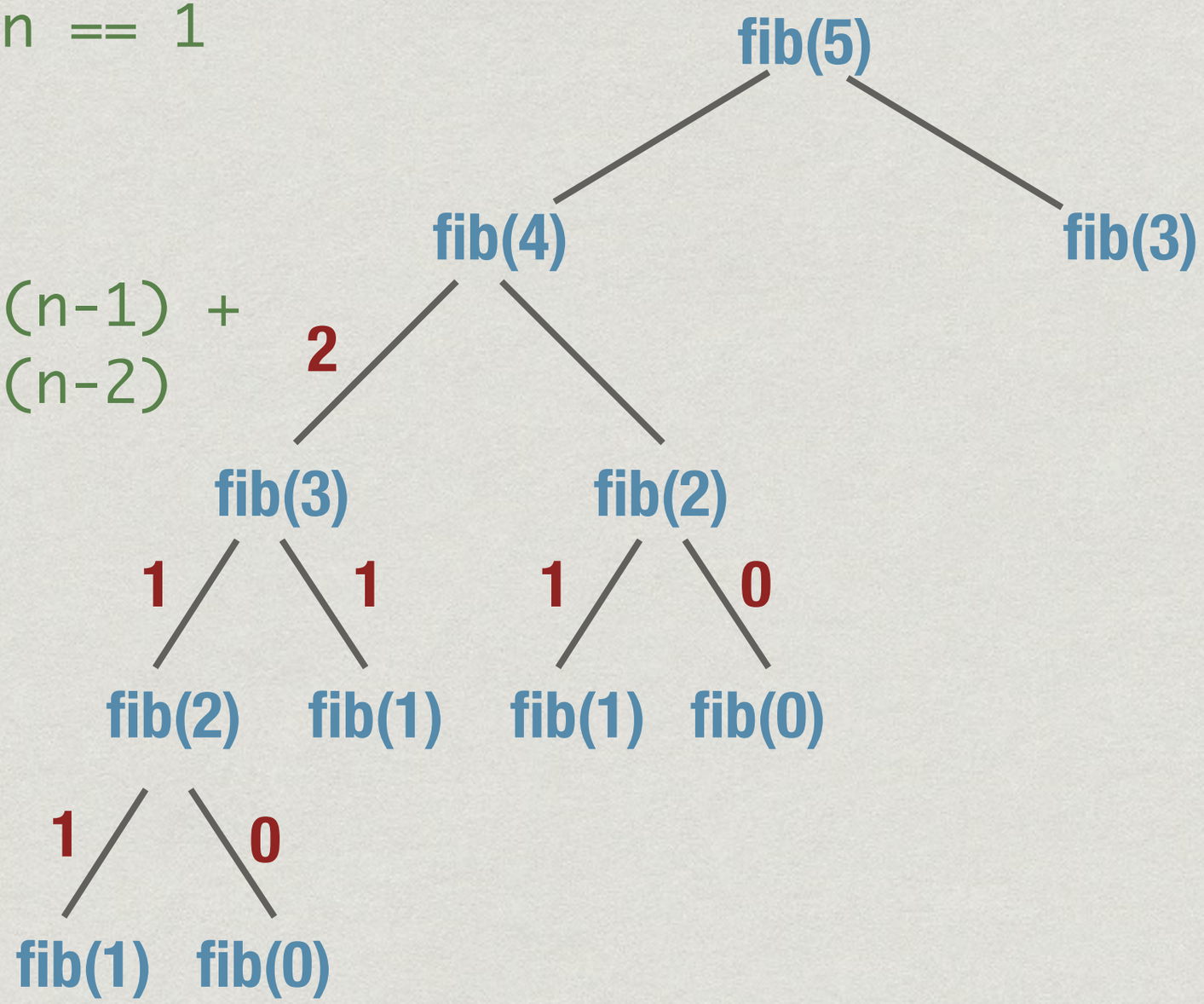
Computing fib(5)

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



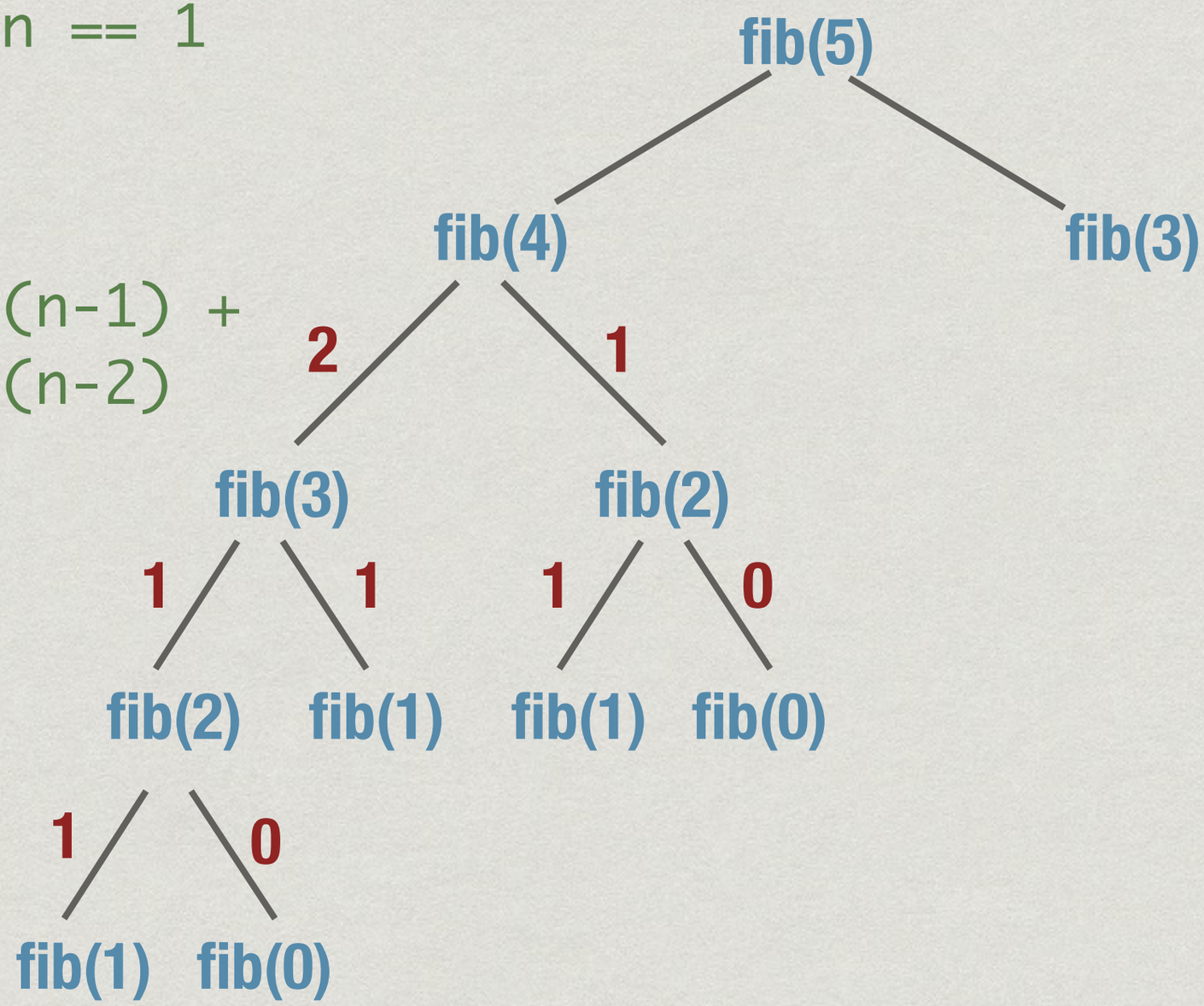
Computing fib(5)

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



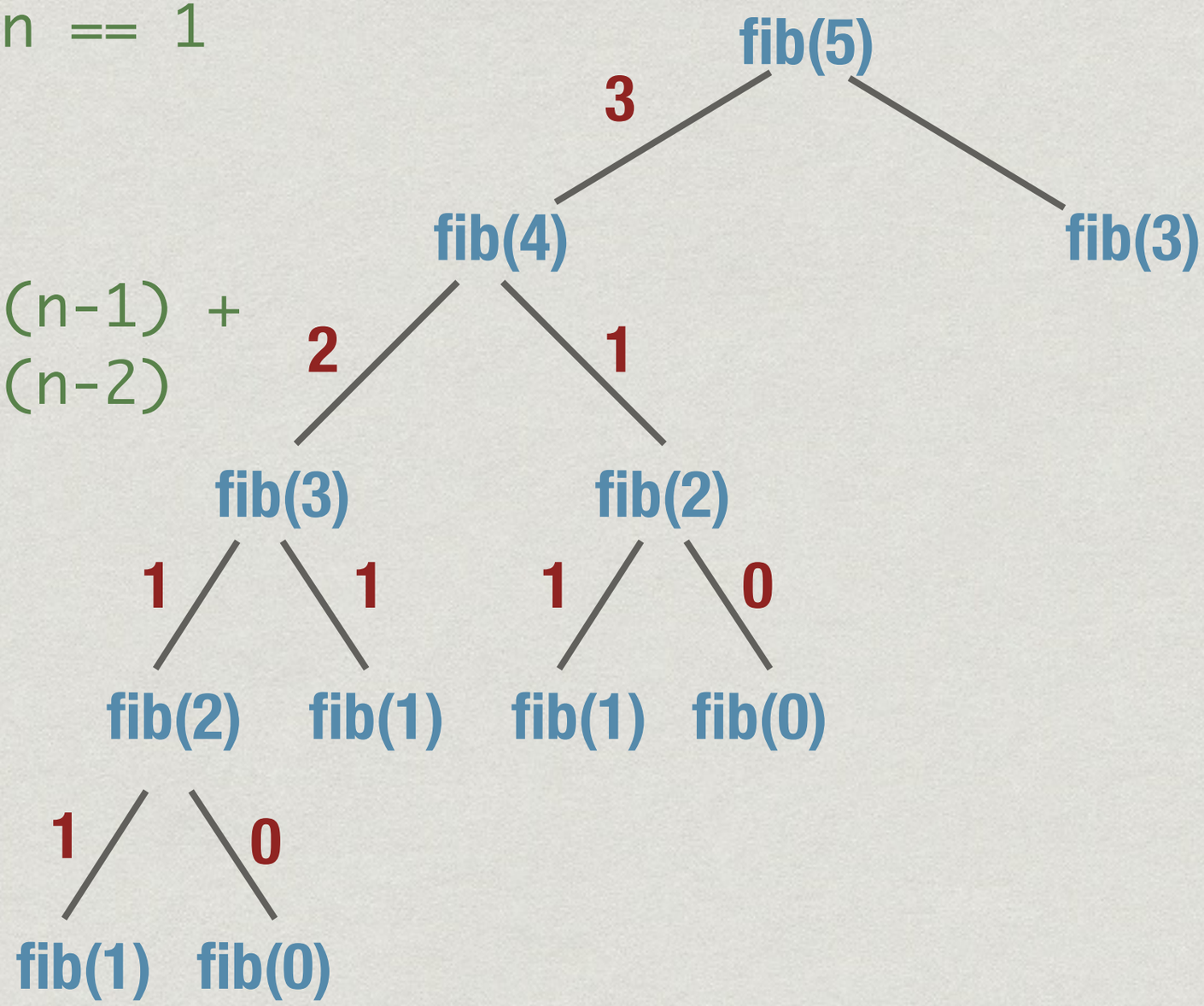
Computing fib(5)

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



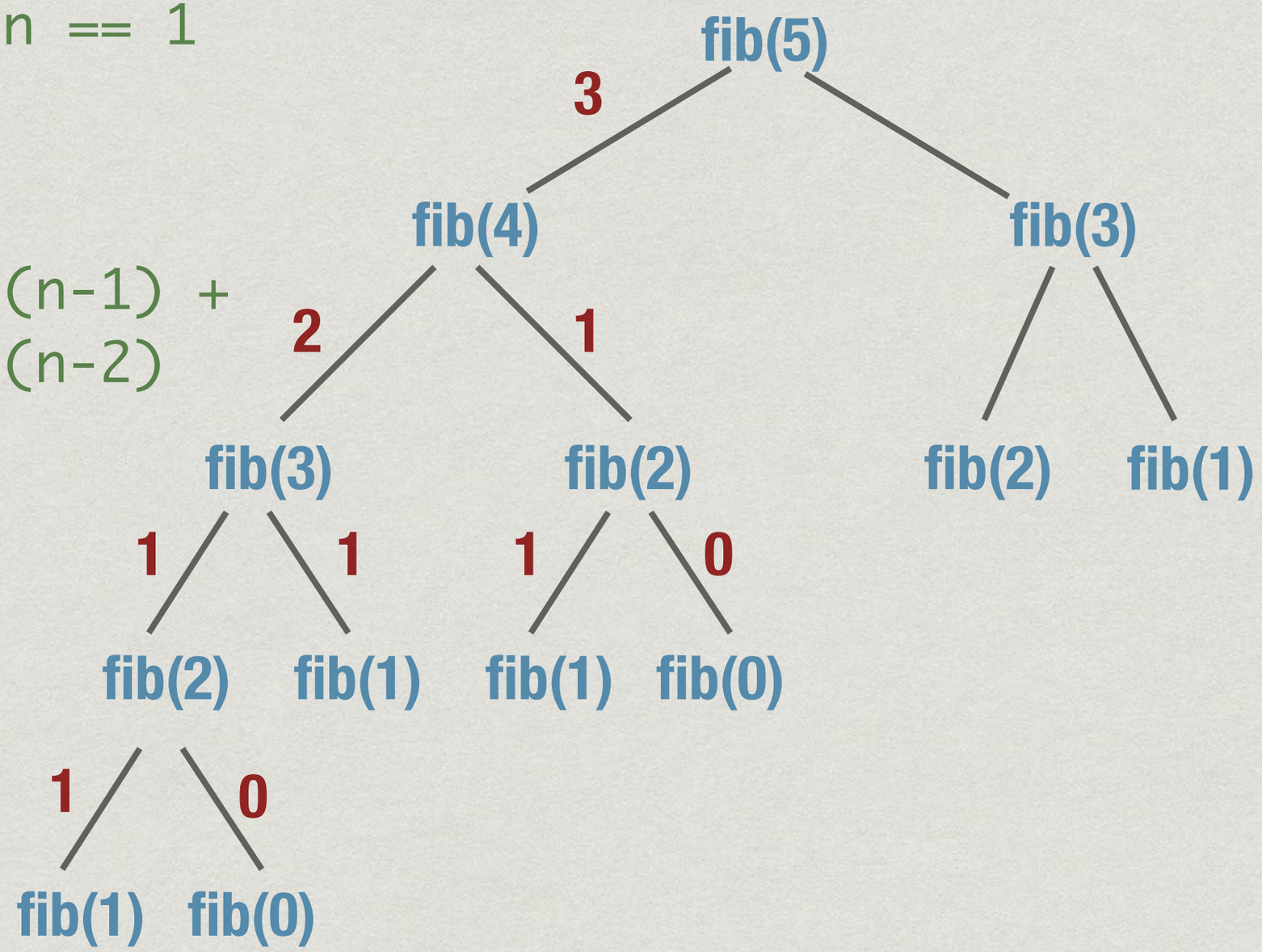
Computing fib(5)

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



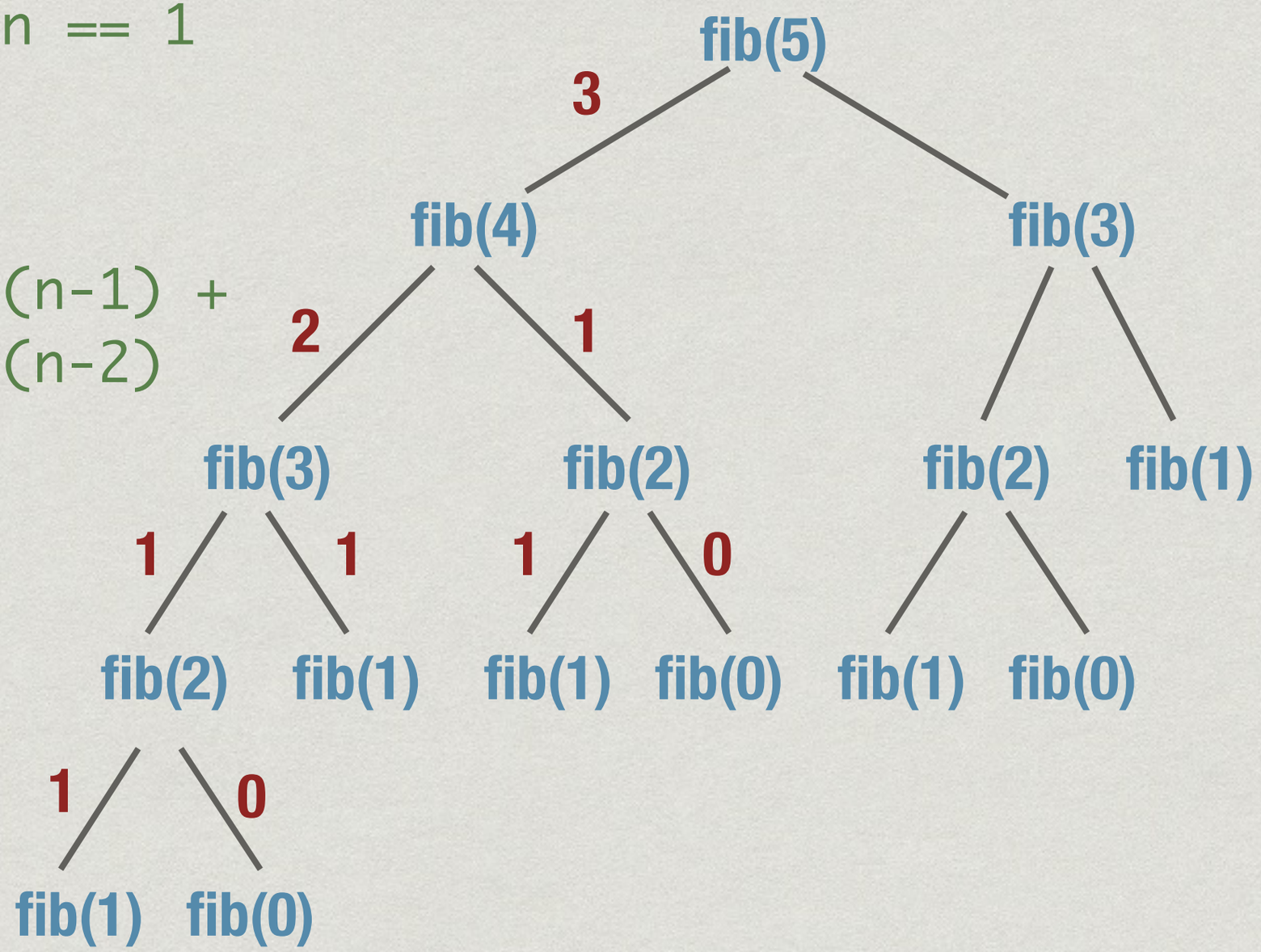
Computing fib(5)

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



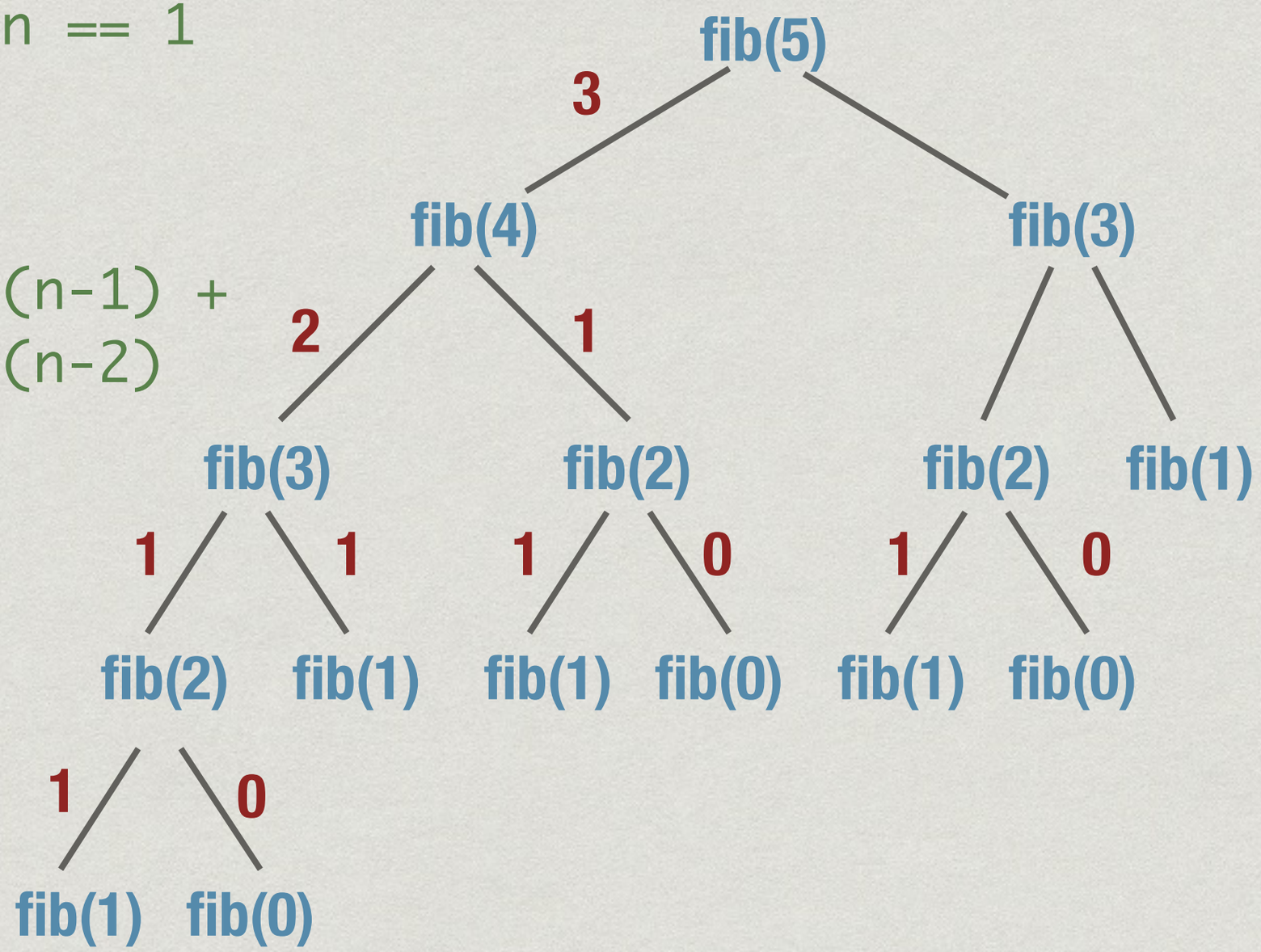
Computing fib(5)

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



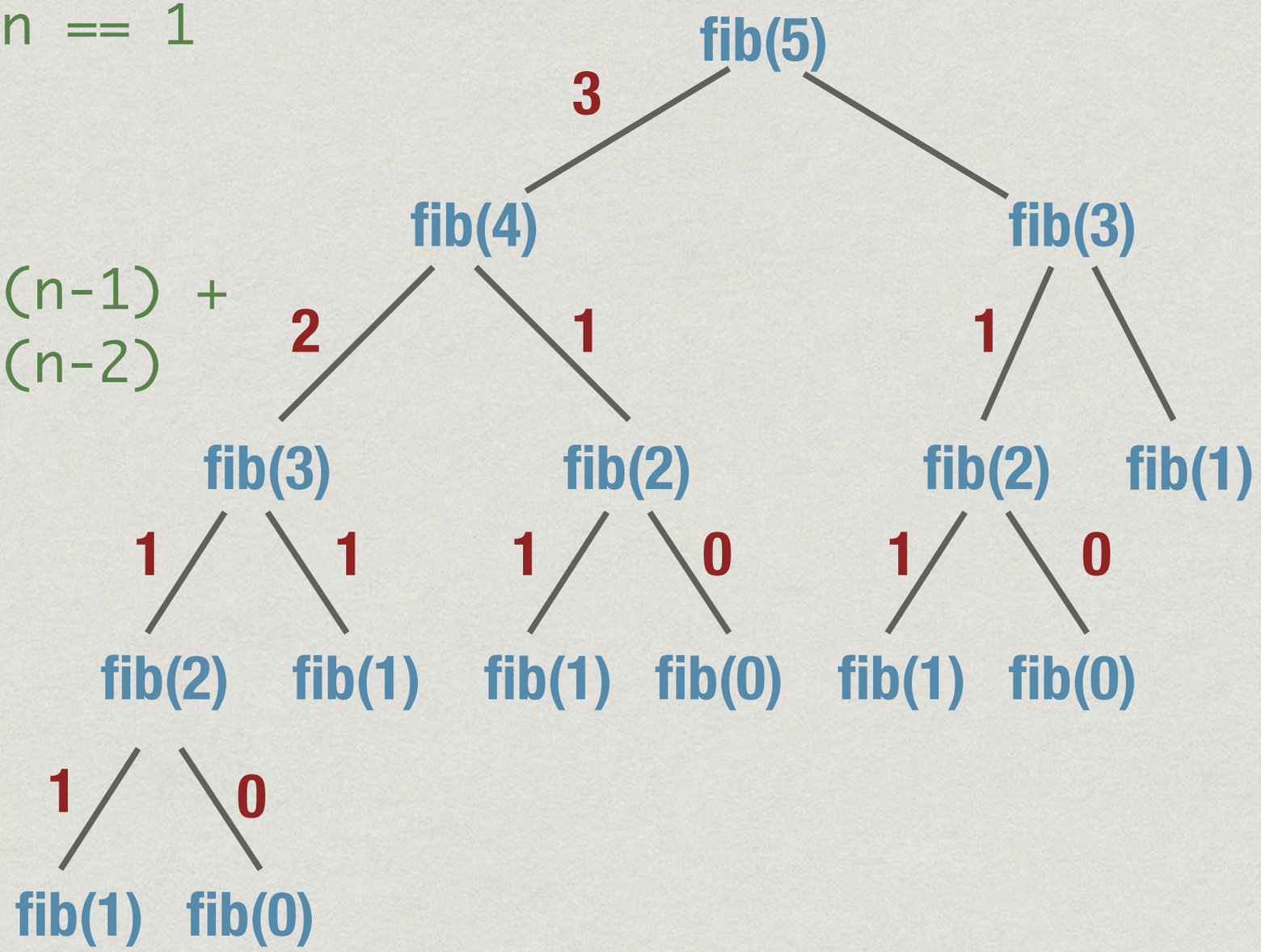
Computing fib(5)

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



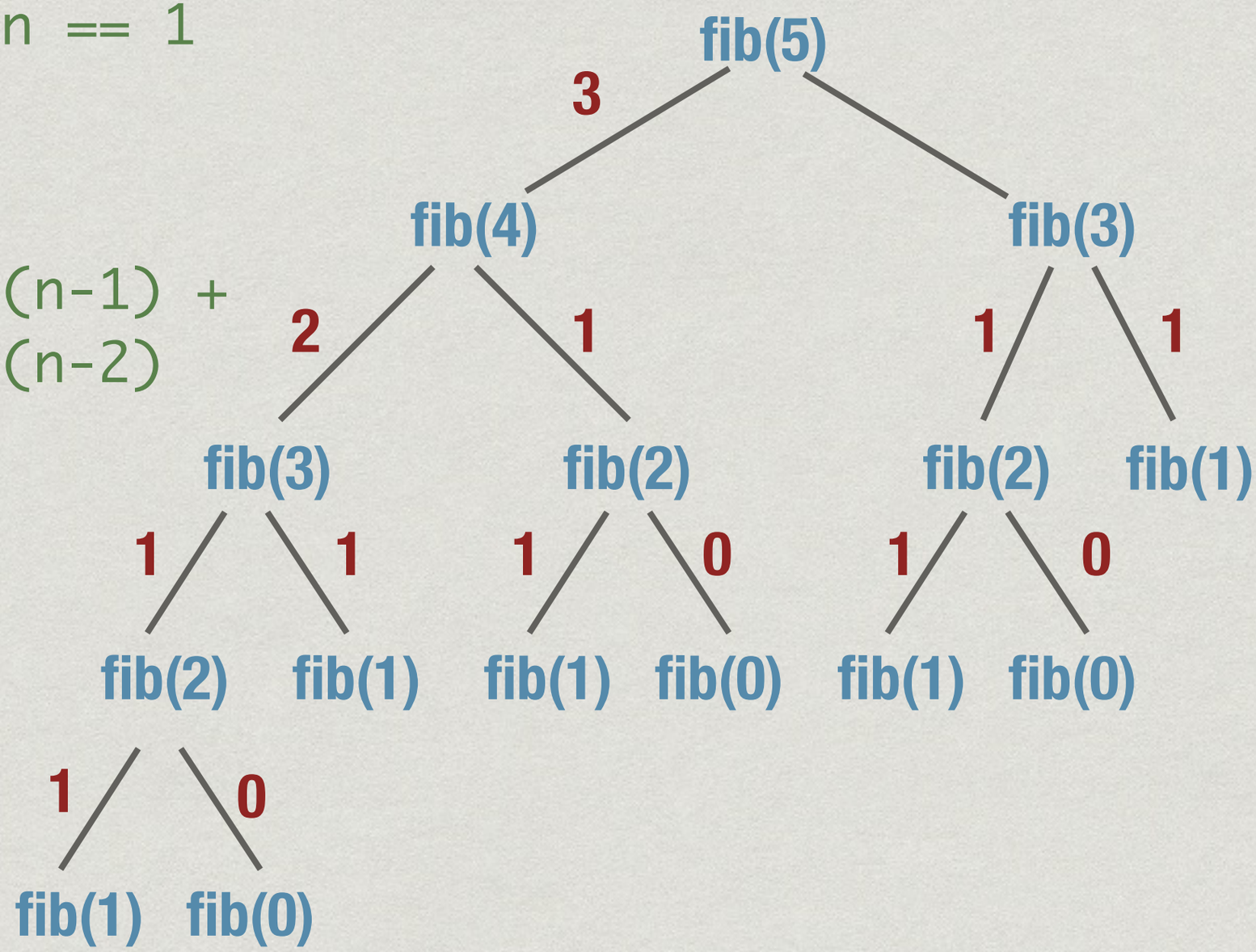
Computing fib(5)

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



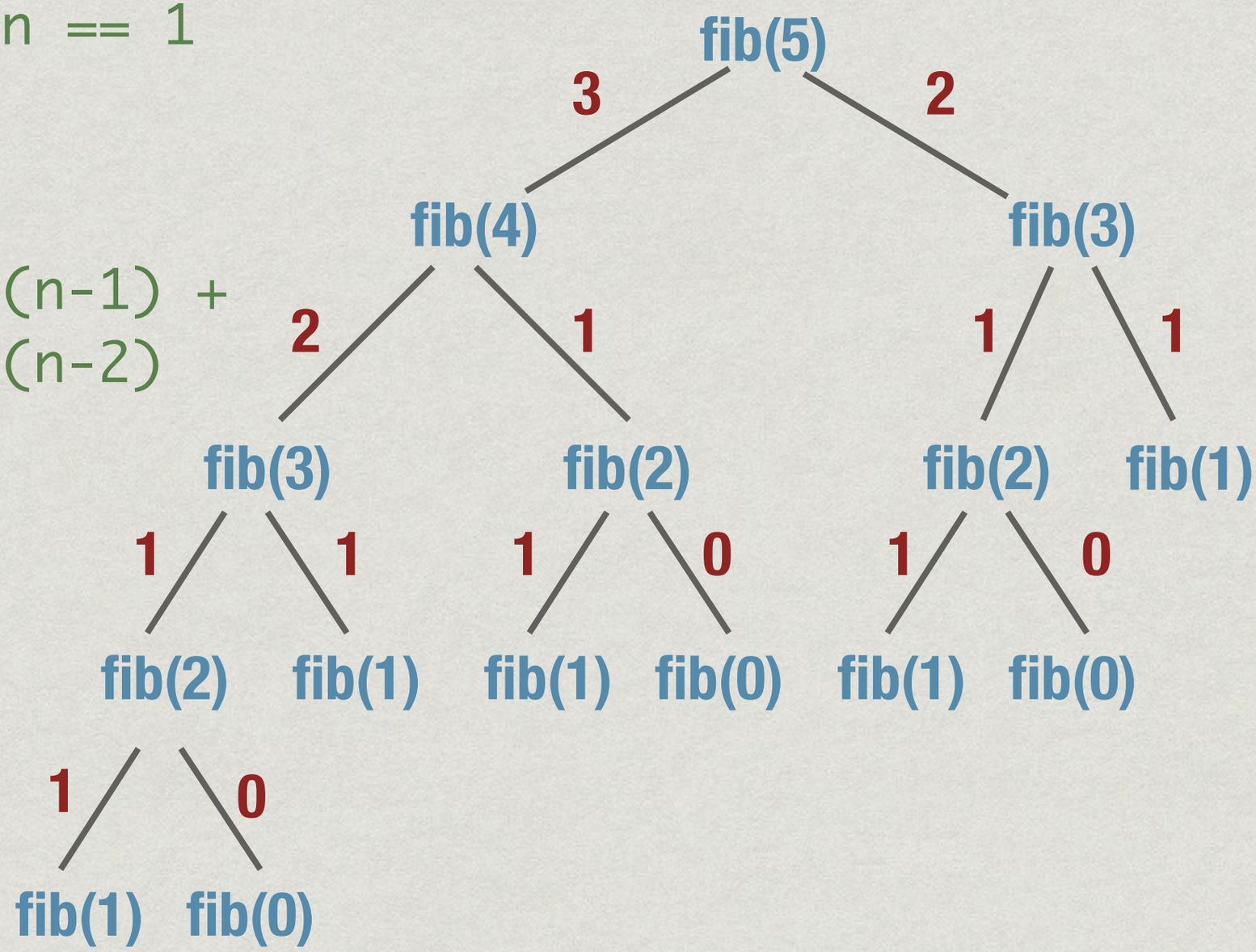
Computing fib(5)

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



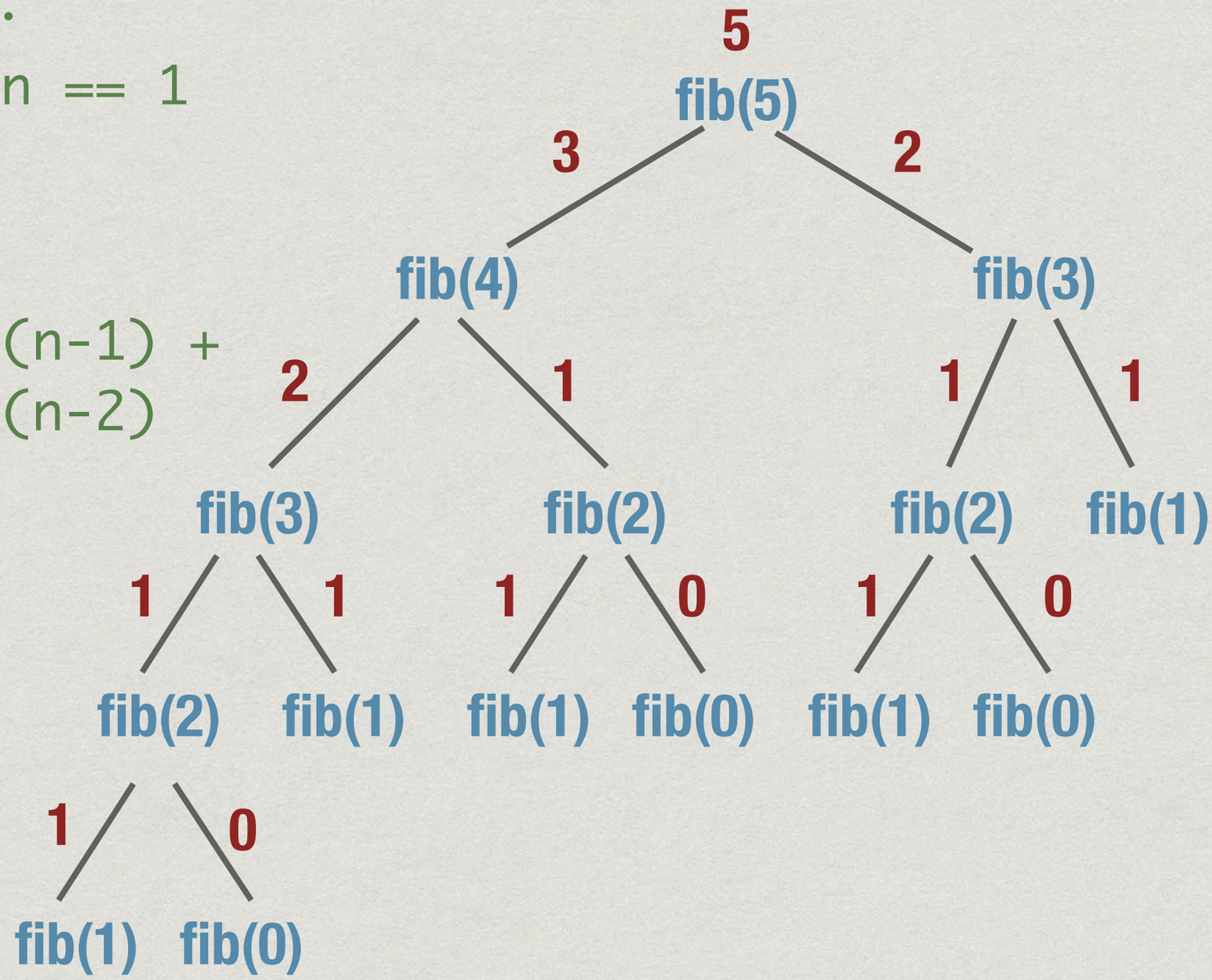
Computing fib(5)

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



Computing fib(5)

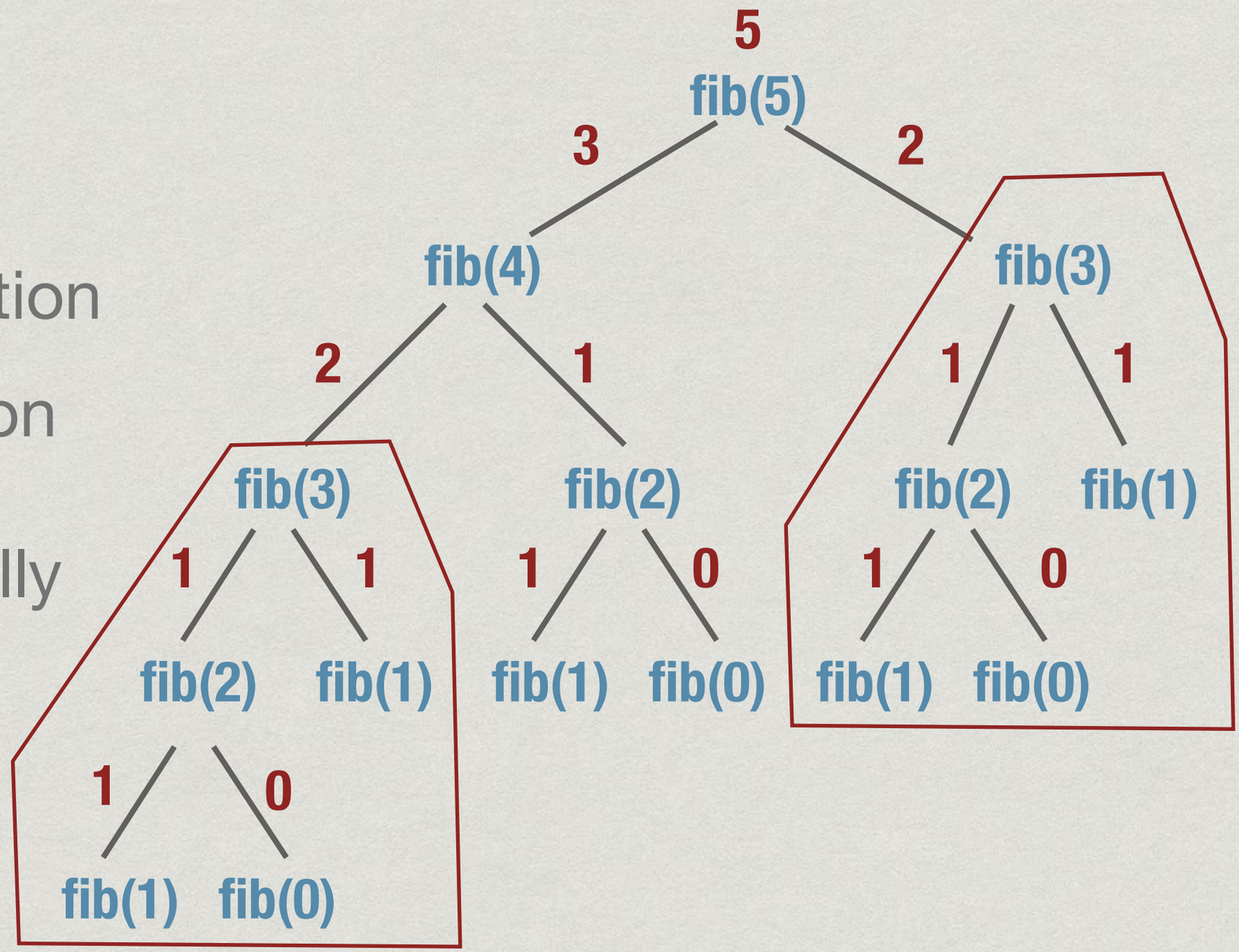
```
function fib(n):
   if n == 0 or n == 1
      value = n
   else
      value = fib(n-1) +
              fib(n-2)
   return(value)
```



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

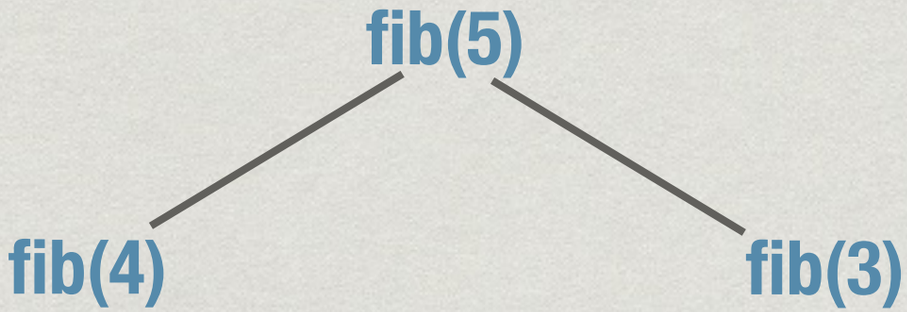
fib(5)

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

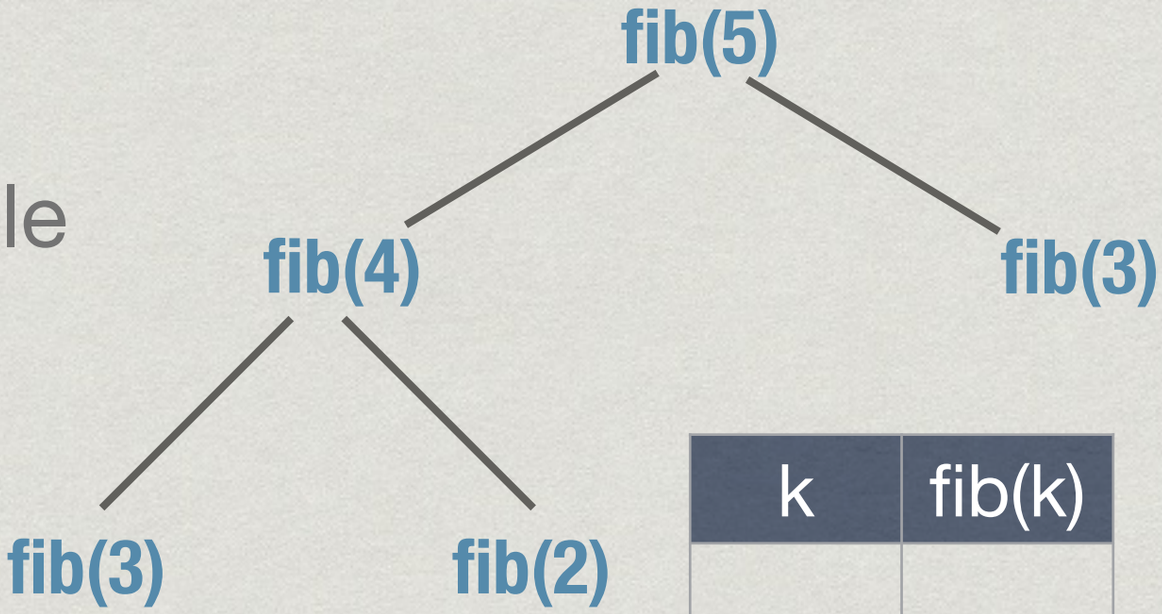


k	fib(k)

Memoized fib(5)

Memoization

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

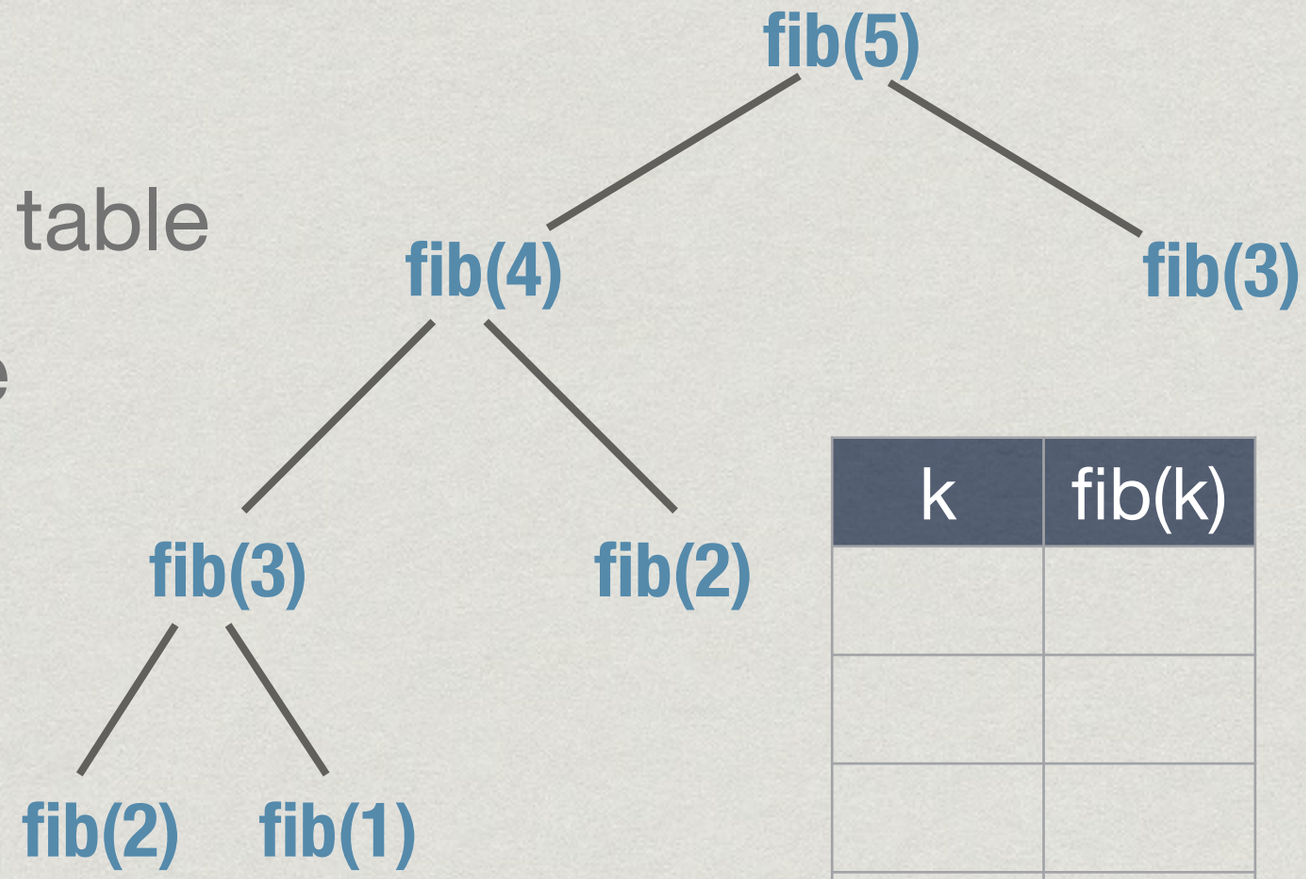


k	fib(k)

Memoized fib(5)

Memoization

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

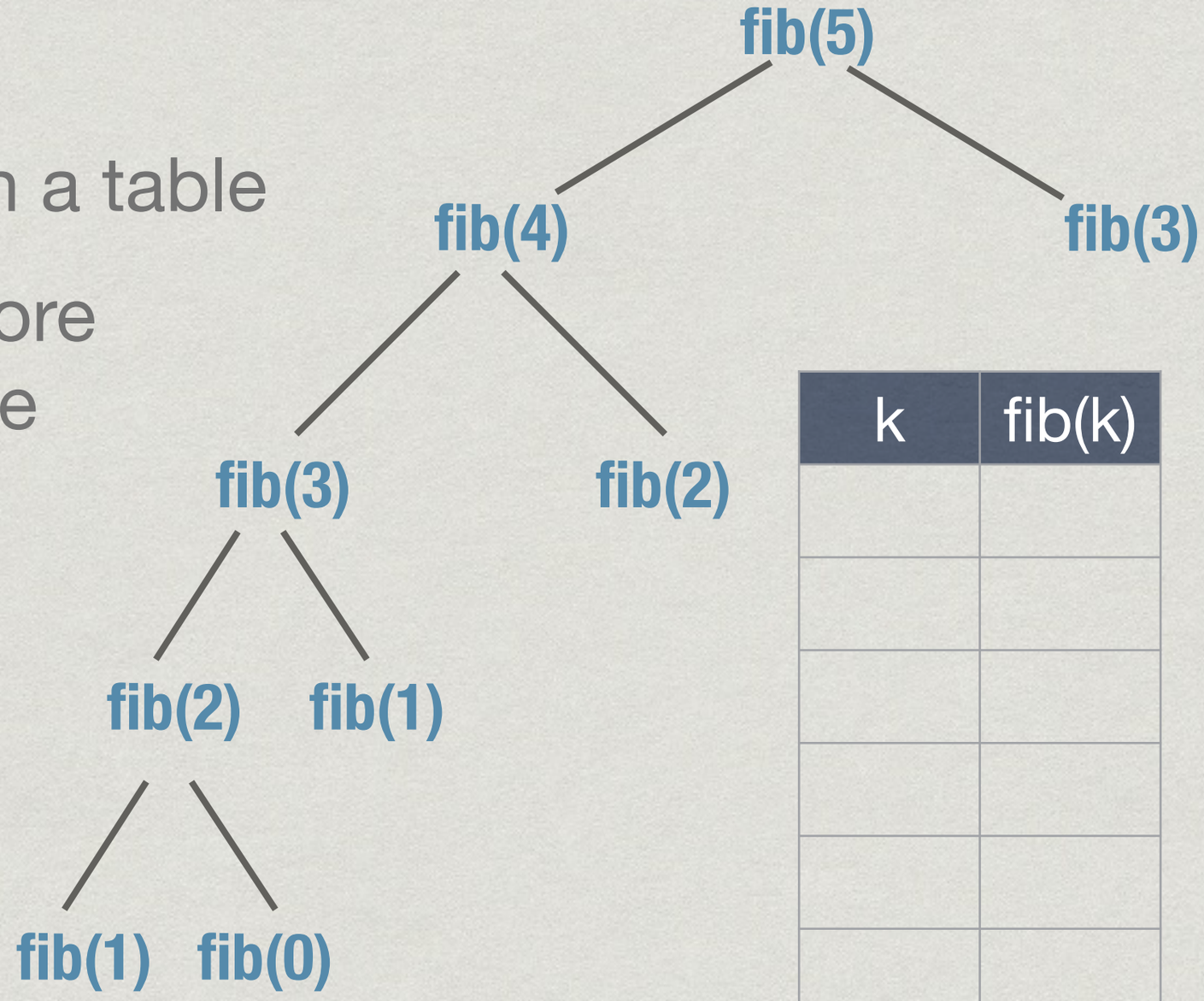


k	fib(k)

Memoized fib(5)

Memoization

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

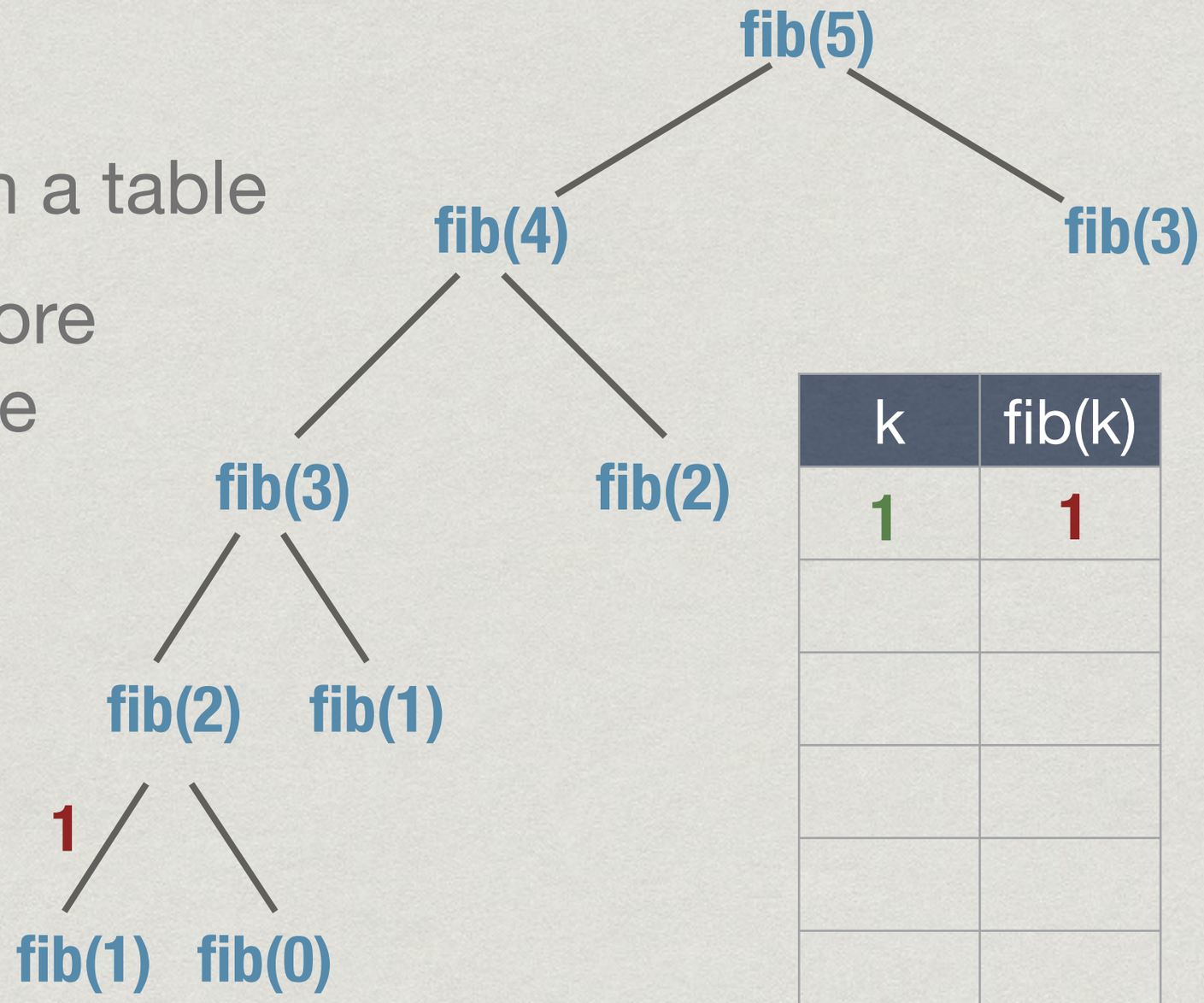


k	fib(k)

Memoized fib(5)

Memoization

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

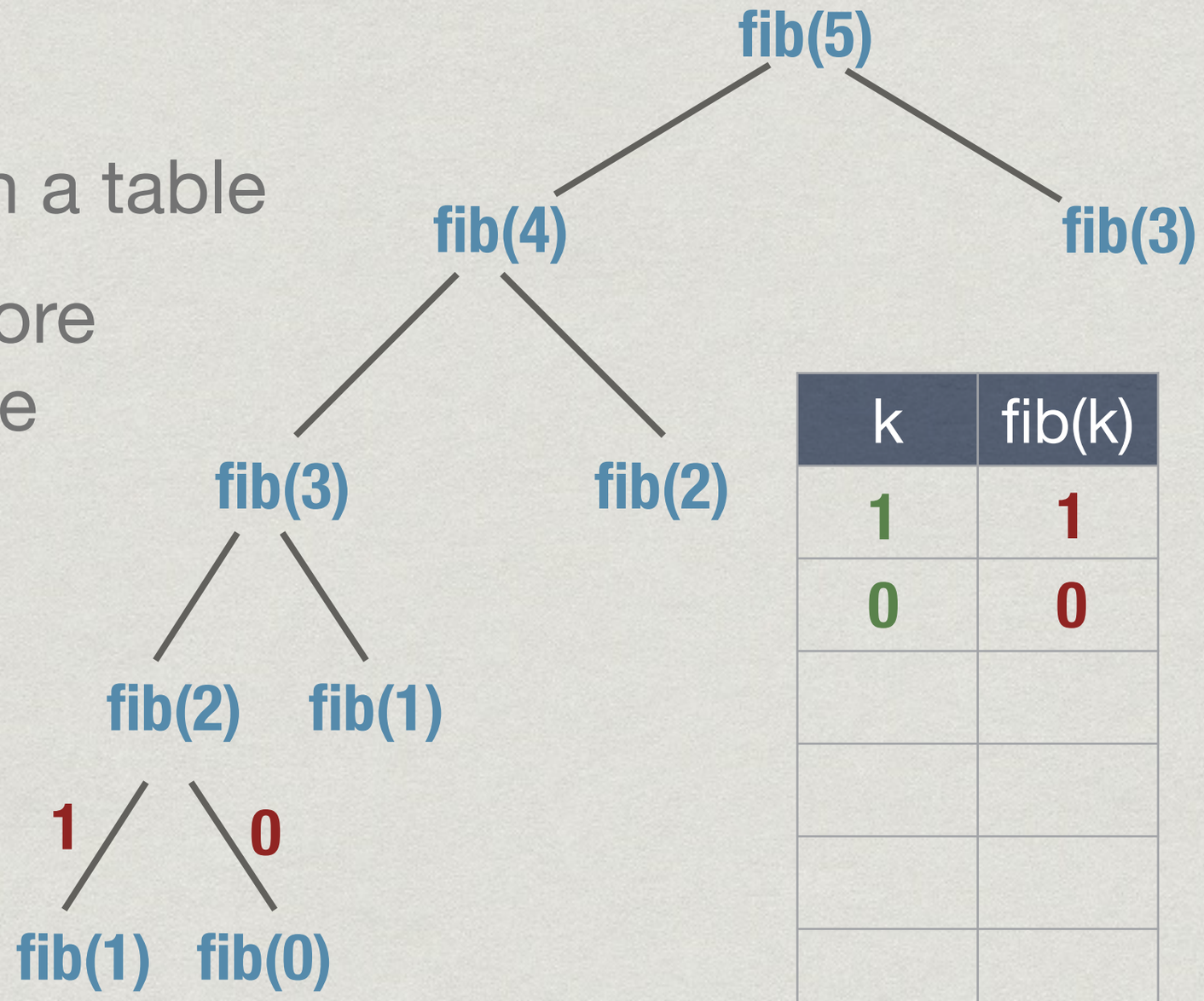


k	fib(k)
1	1

Memoized fib(5)

Memoization

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

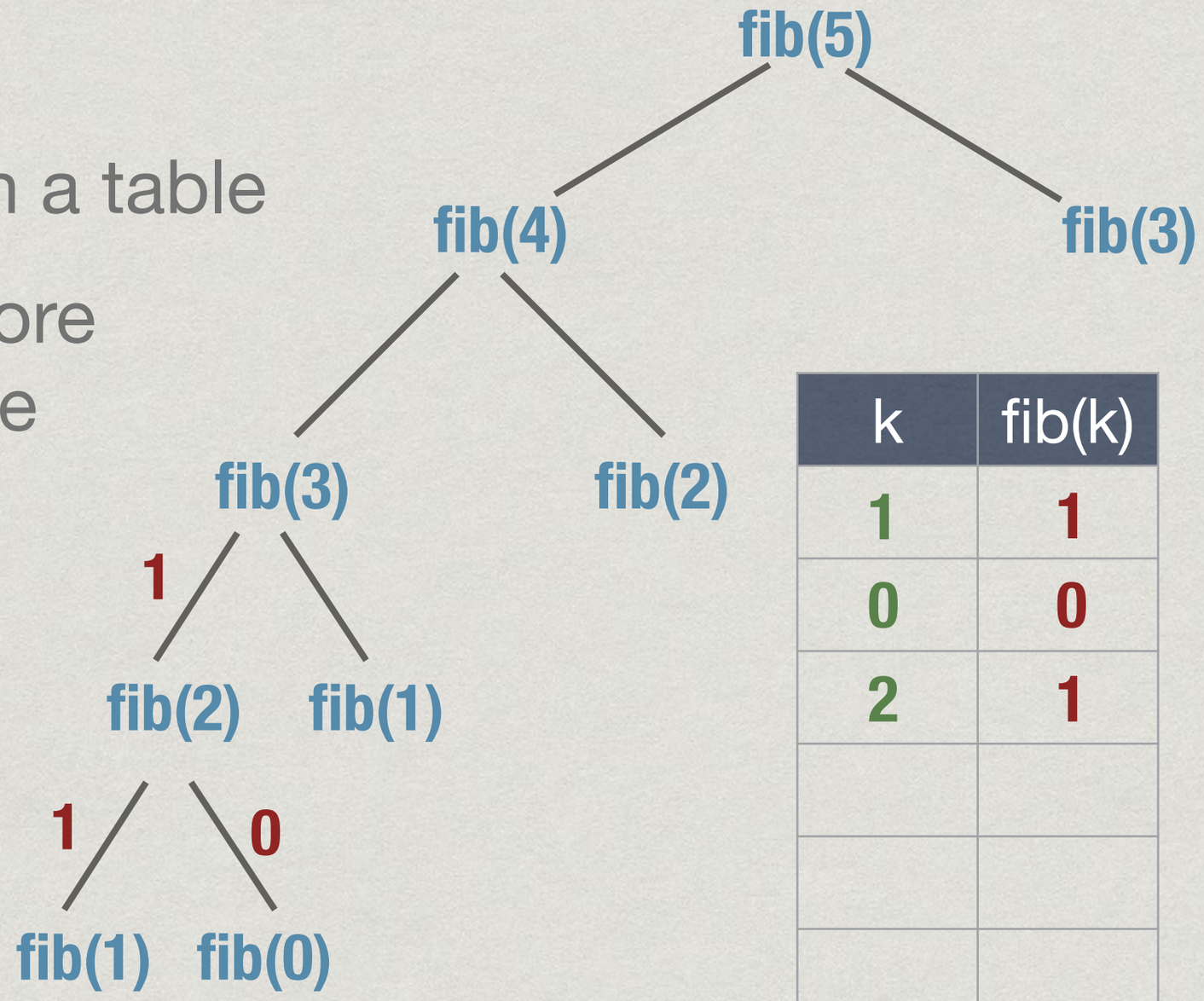


k	fib(k)
1	1
0	0		

Memoized fib(5)

Memoization

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

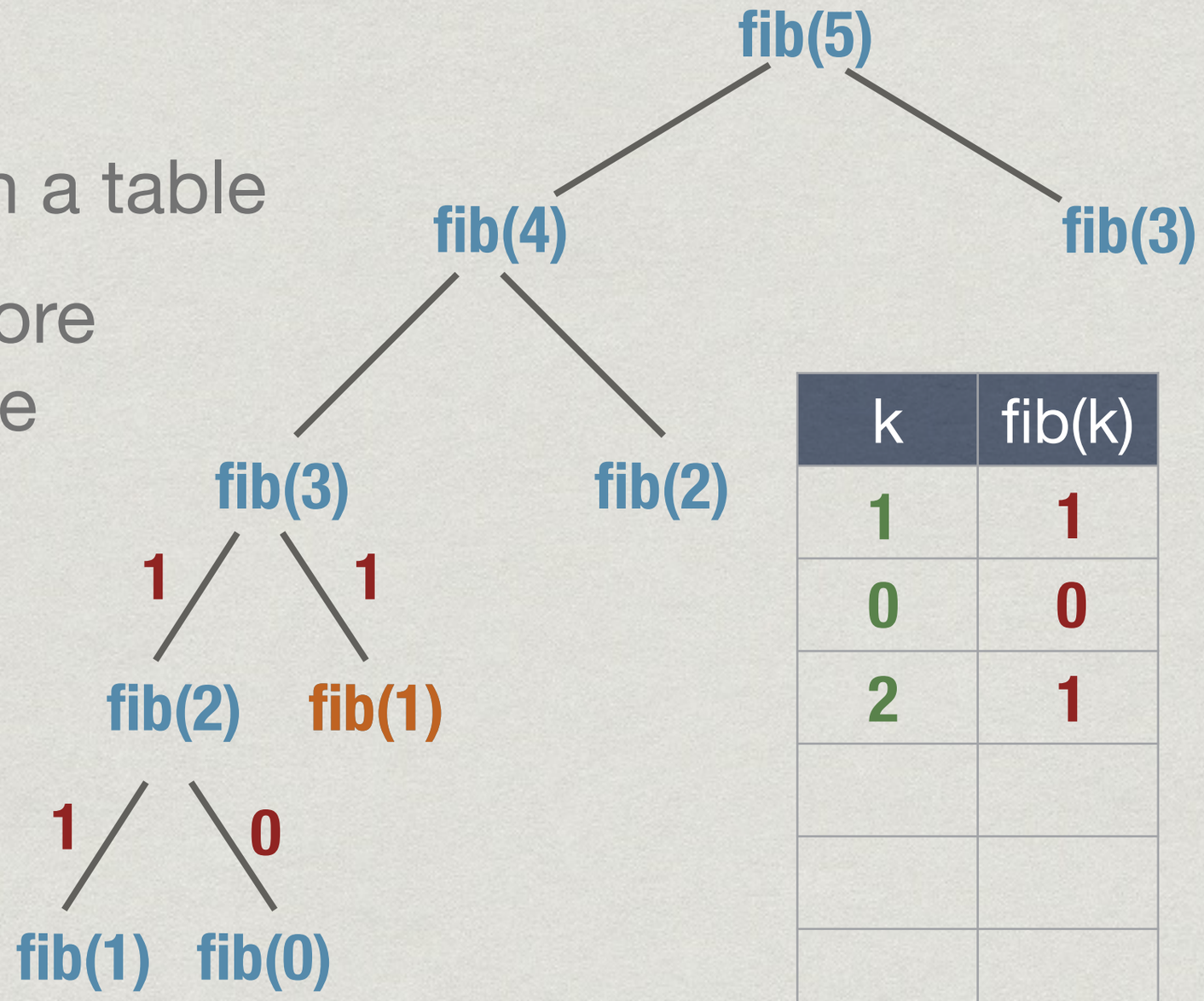


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

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

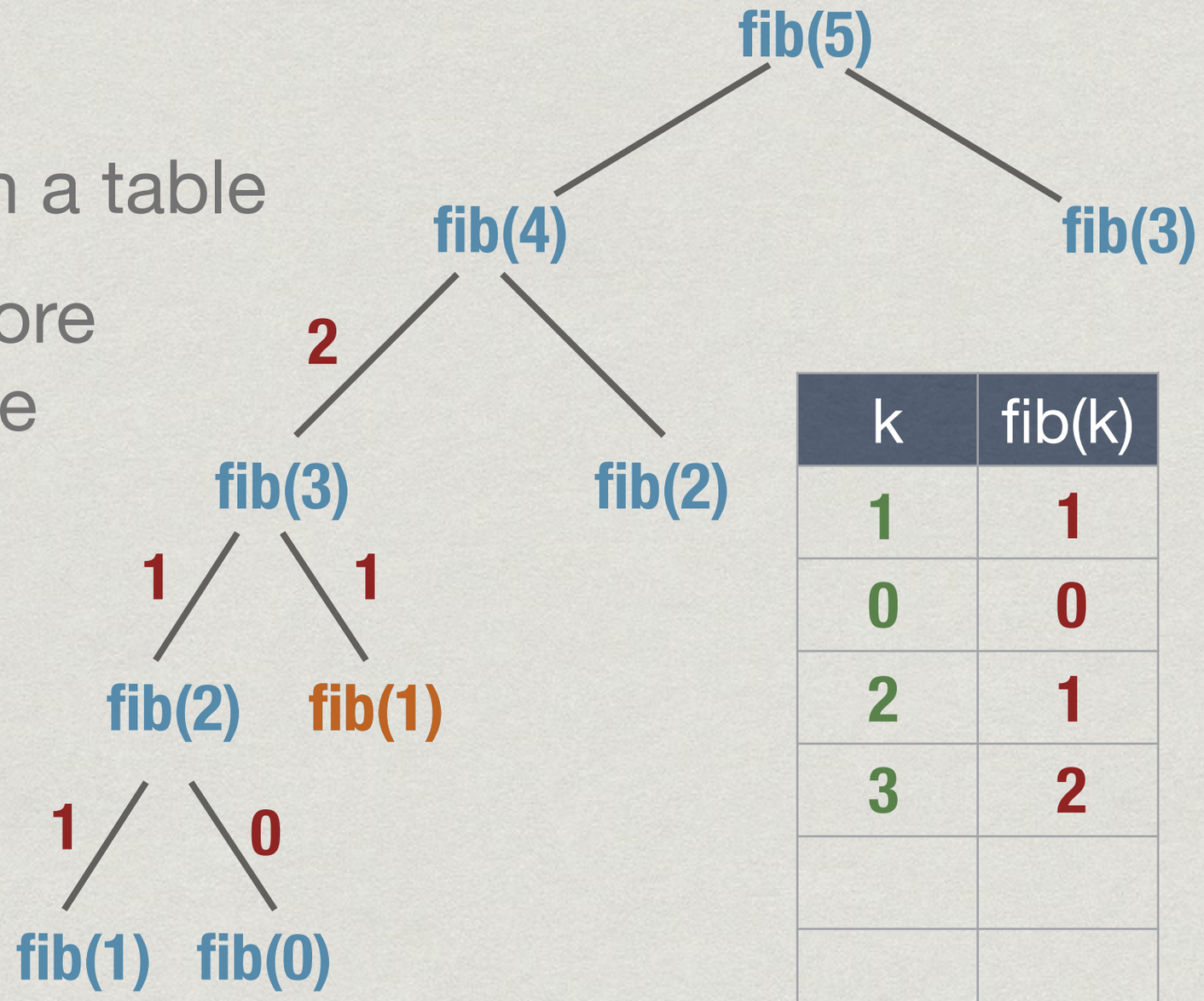


k	fib(k)
1	1
0	0
2	1

Memoized fib(5)

Memoization

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

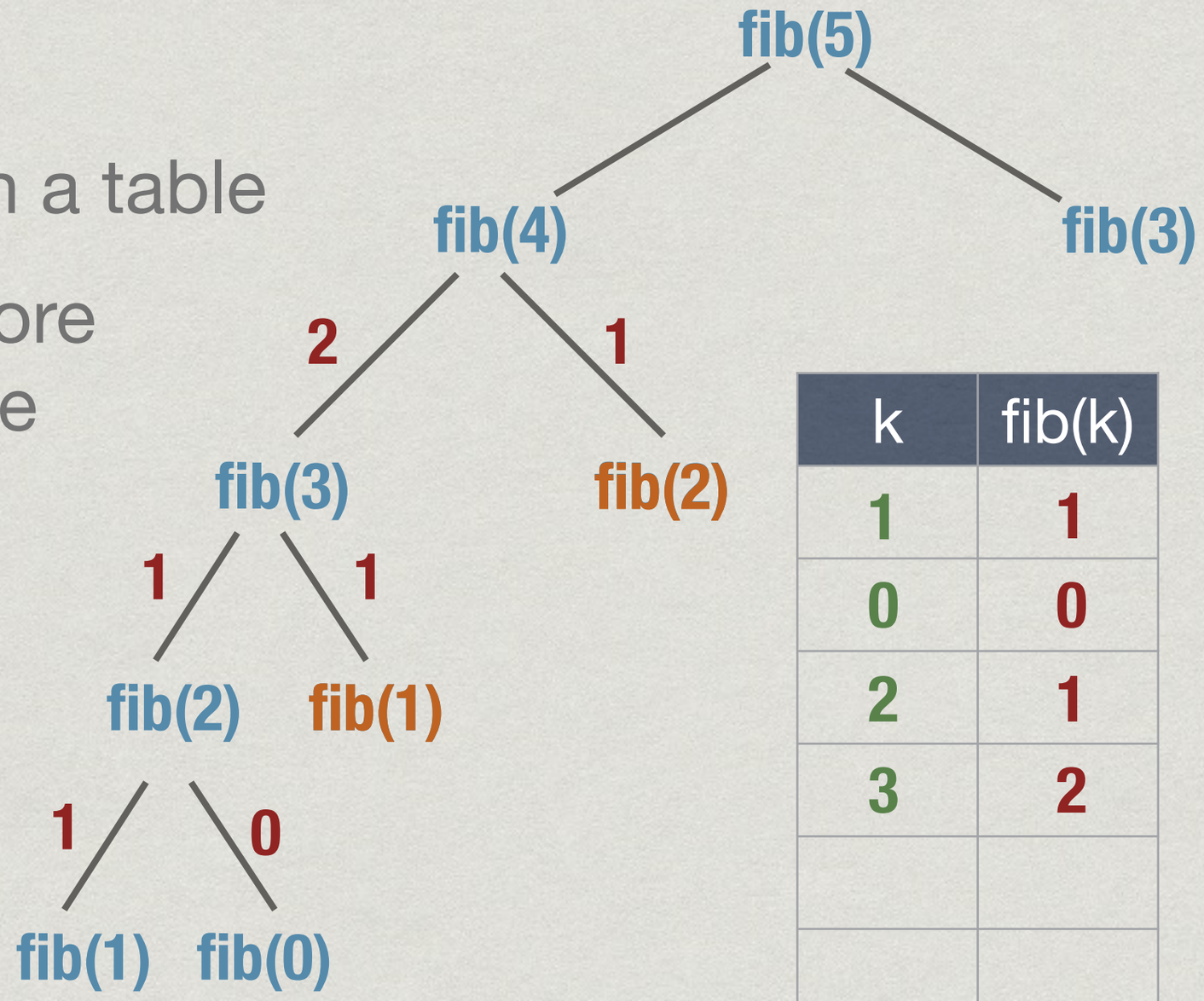


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

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

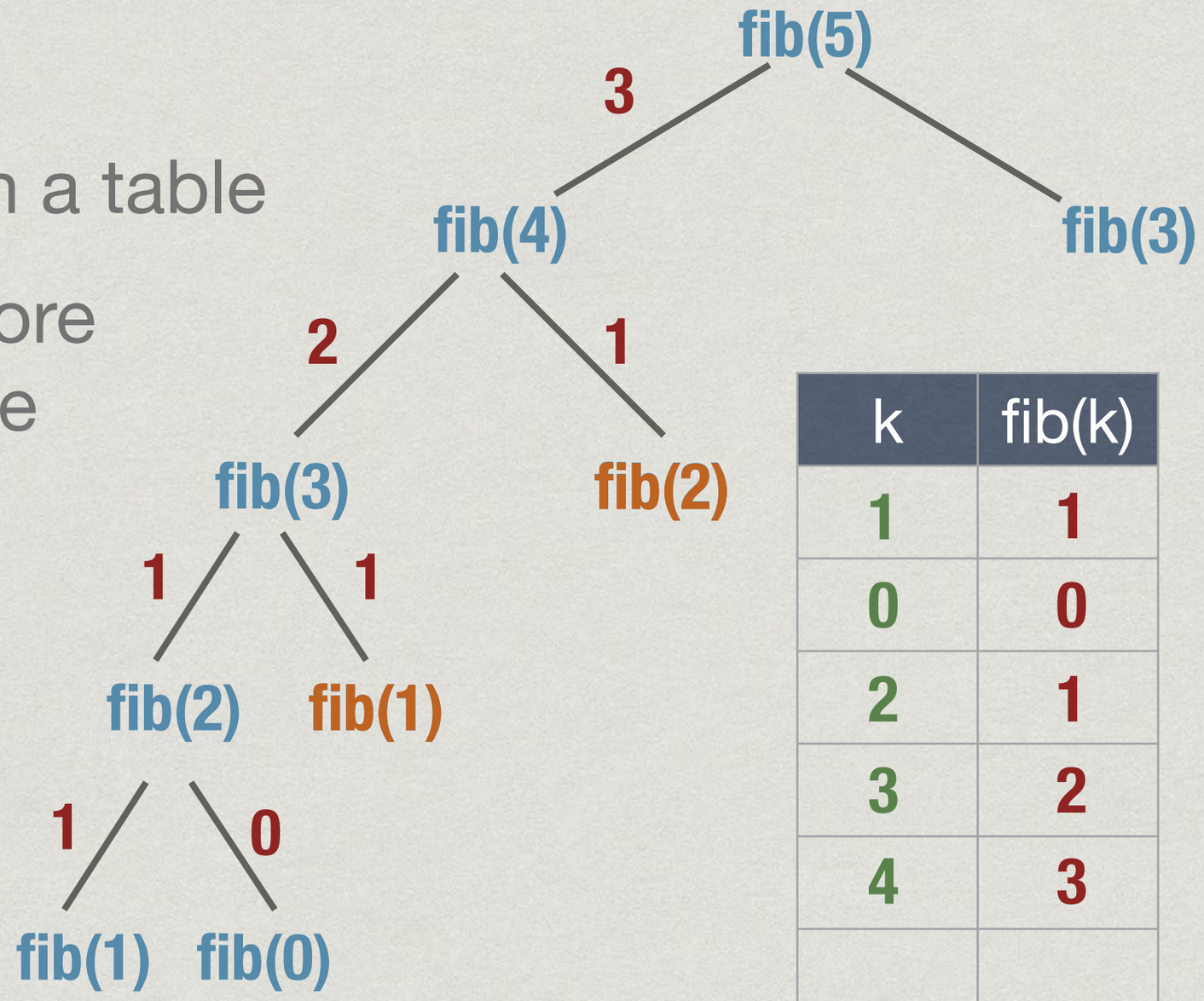


k	fib(k)
1	1
0	0
2	1
3	2

Memoized fib(5)

Memoization

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

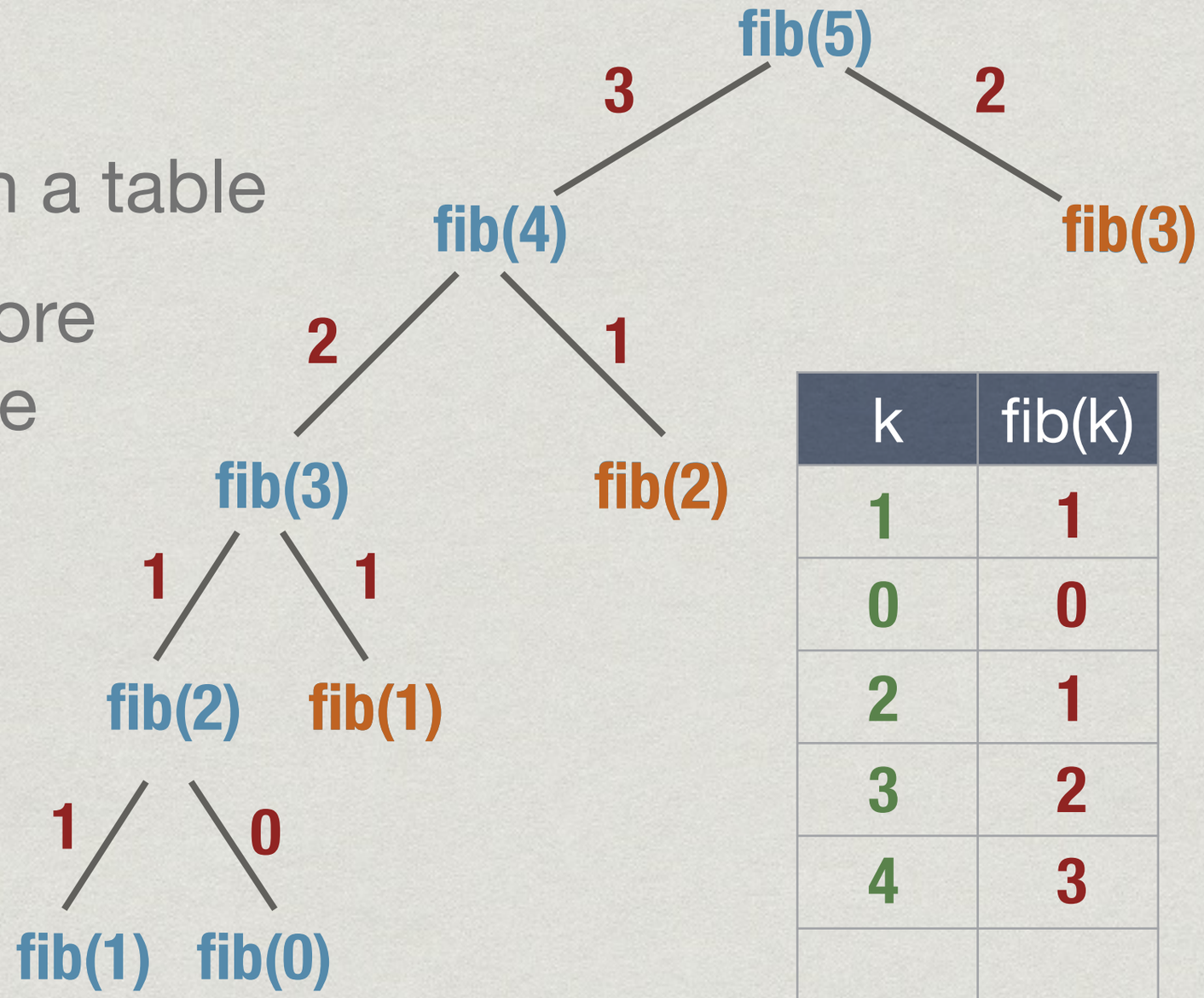


k	fib(k)
1	1
0	0
2	1
3	2
4	3

Memoized fib(5)

Memoization

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

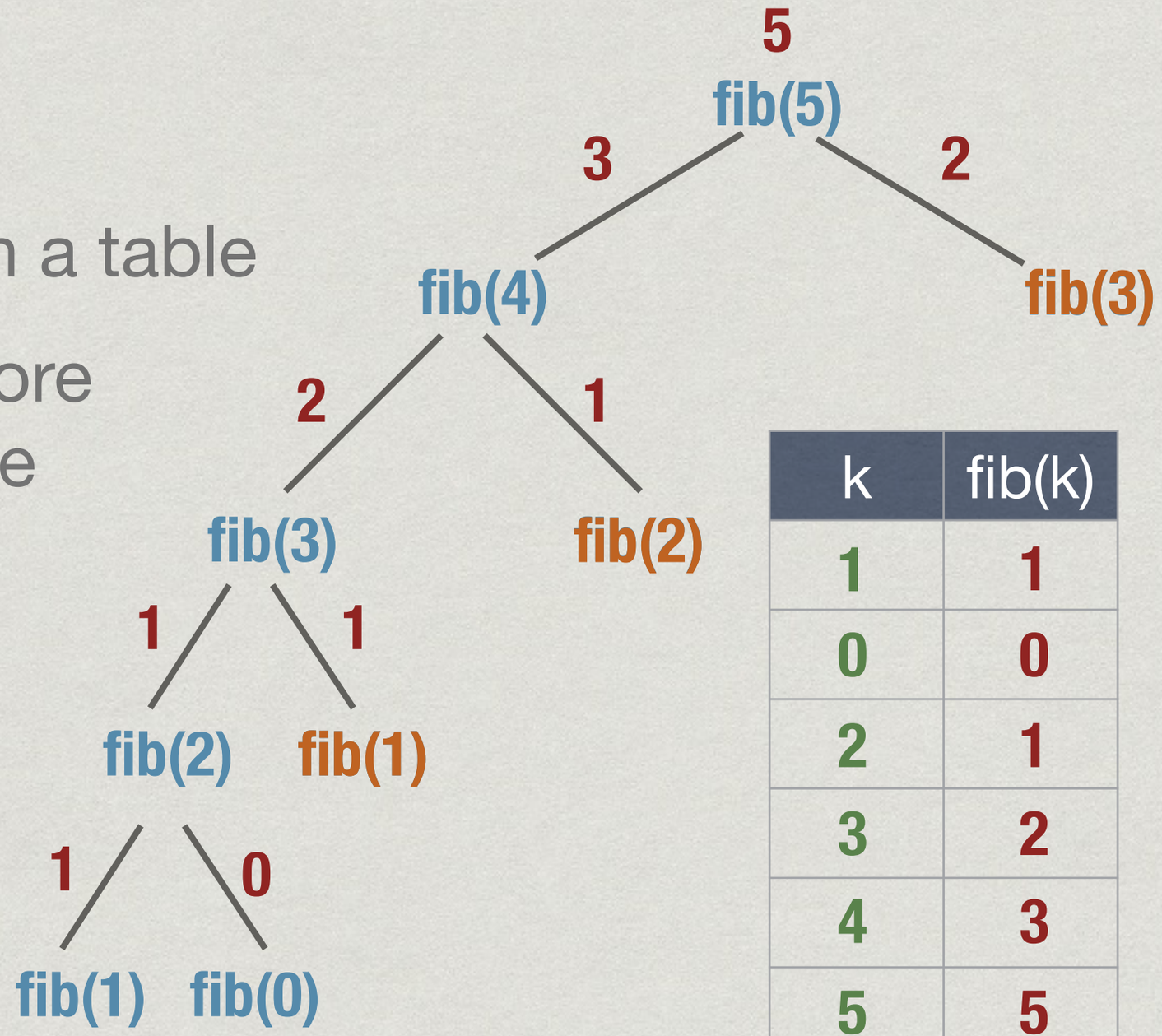


k	fib(k)
1	1
0	0
2	1
3	2
4	3

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

```
function fib(n):
  if fibtable[n]
    return(fibtable[n])
  if n == 0 or n == 1
    value = n
  else
    value = fib(n-1) + fib(n-2)
  fibtable[n] = value
  return(value)
```

In general

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

Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order

Dynamic programming

- Anticipate what the memory table looks like

  - Subproblems are known from problem structure

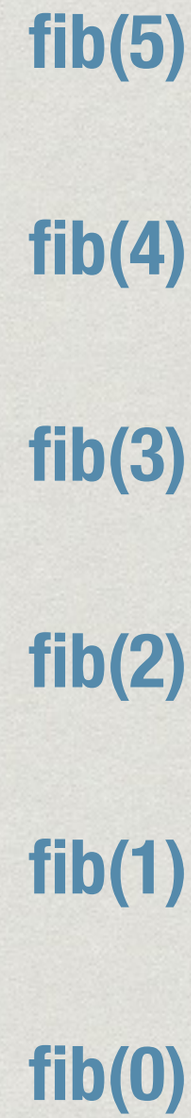
    - Dependencies form a dag

- Solve subproblems in topological order

fib(5)

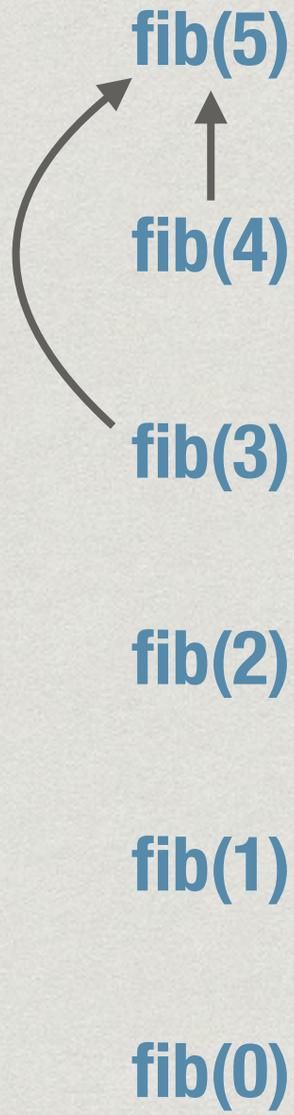
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
    - Dependencies form a dag
- Solve subproblems in topological order



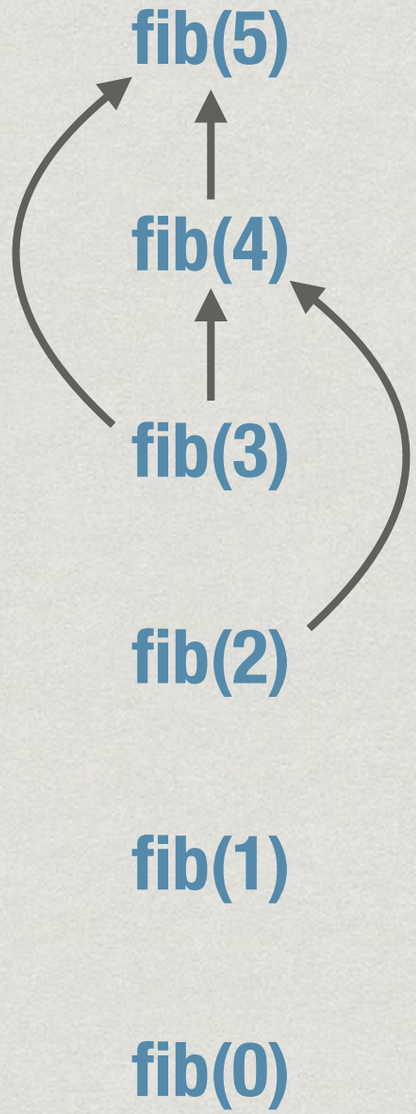
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



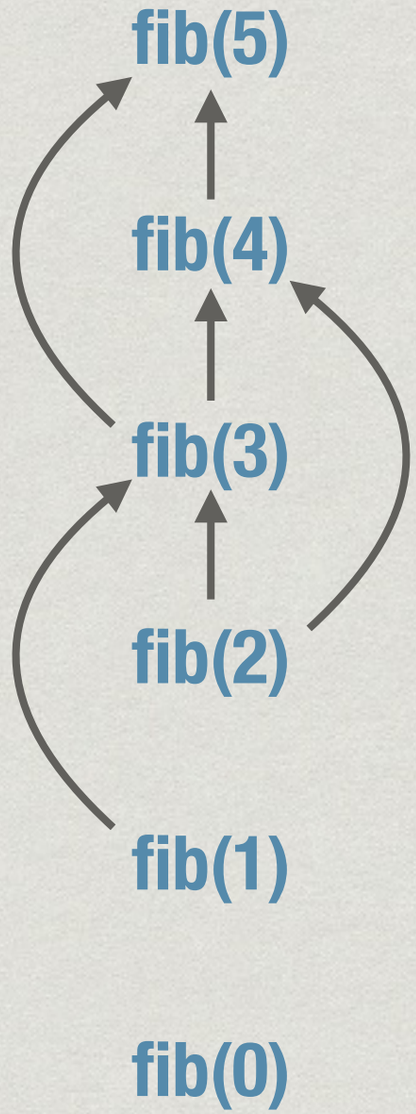
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



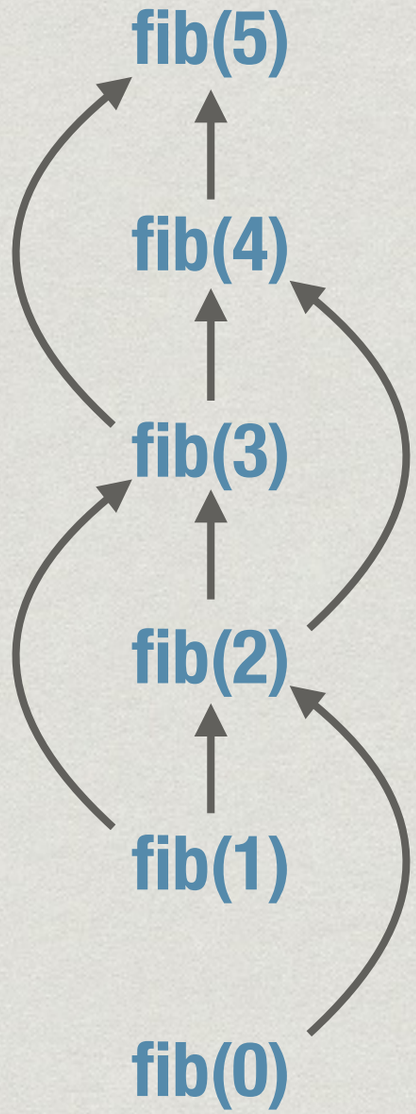
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



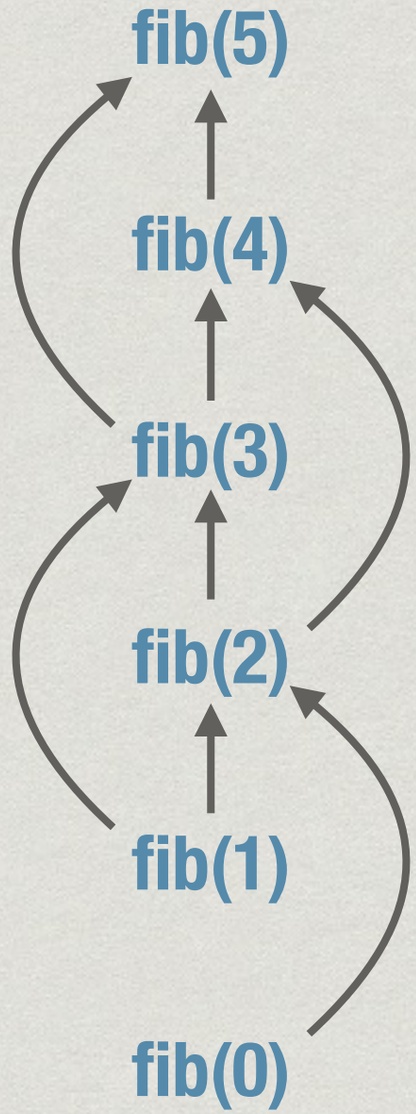
Dynamic programming

- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



Dynamic programming

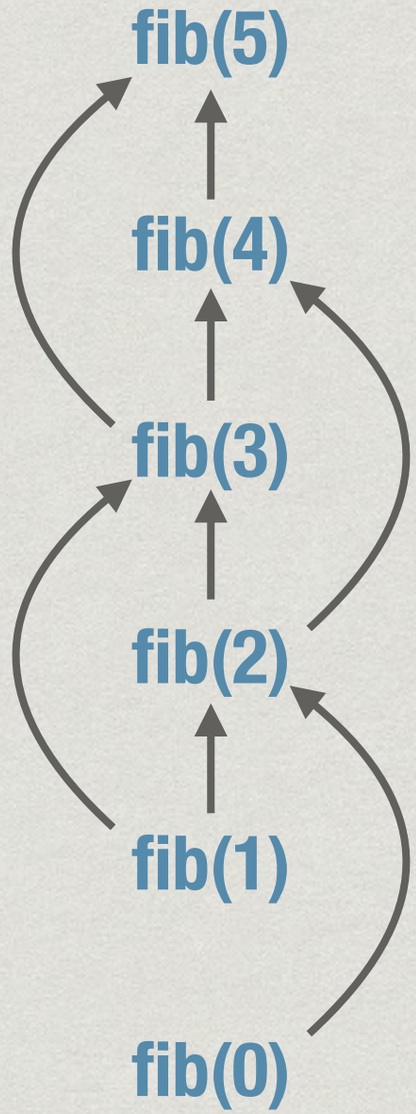
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



k	0	1	2	3	4	5
fib(k)						

Dynamic programming

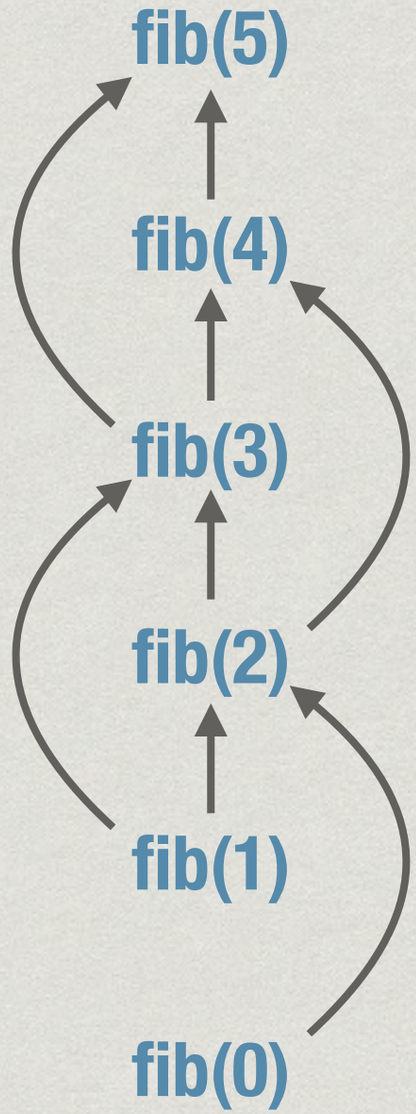
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



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

Dynamic programming

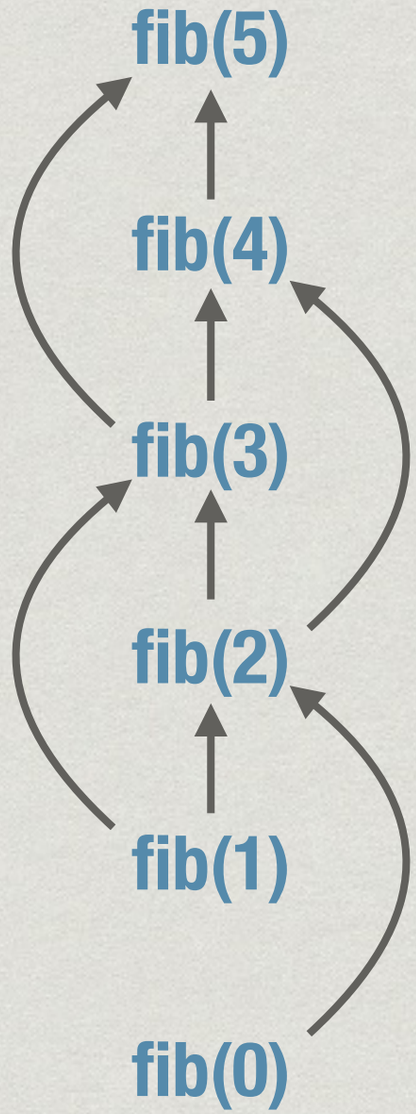
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



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

Dynamic programming

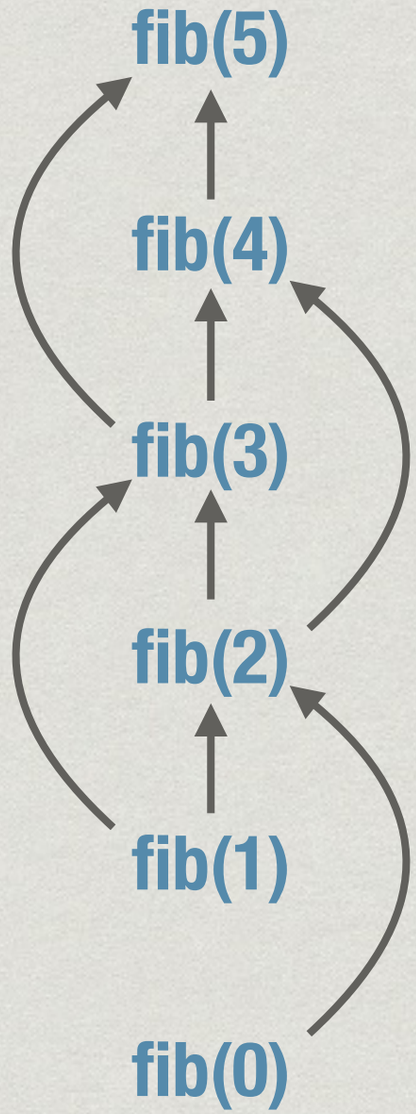
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



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

Dynamic programming

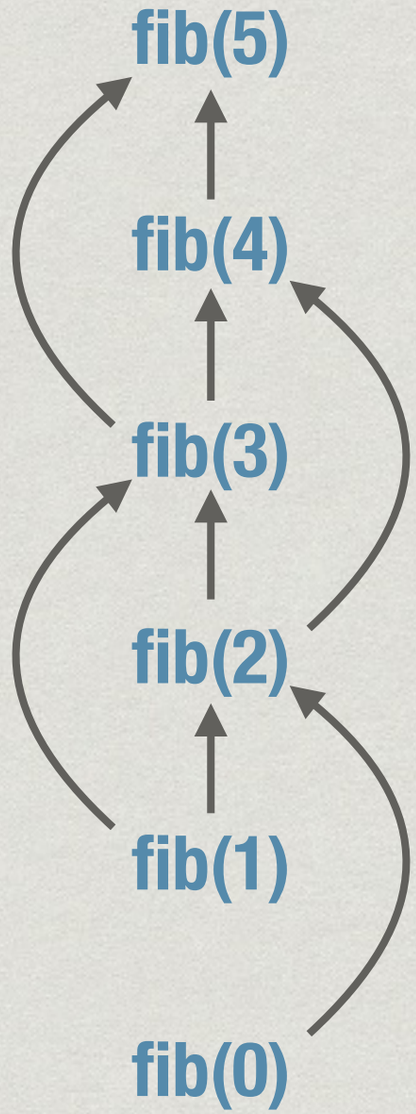
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



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

Dynamic programming

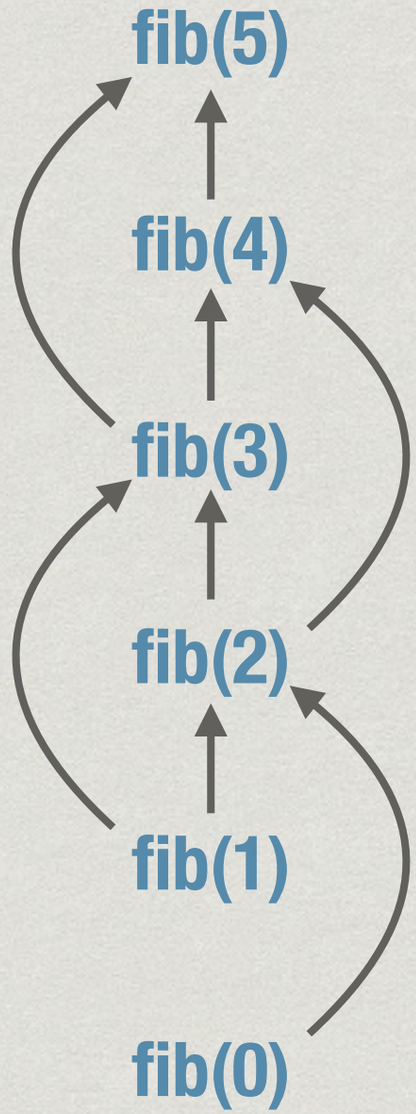
- Anticipate what the memory table looks like
  - Subproblems are known from problem structure
  - Dependencies form a dag
- Solve subproblems in topological order



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
  - Dependencies form a dag
- Solve subproblems in topological order



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

Dynamic programming fibonacci

```
function fib(n):
  fibtable[0] = 0
  fibtable[1] = 1
  for i = 2,3,..n
    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 topological order of dependency
  - Dependencies must form a dag (why?)
- Iterative evaluation