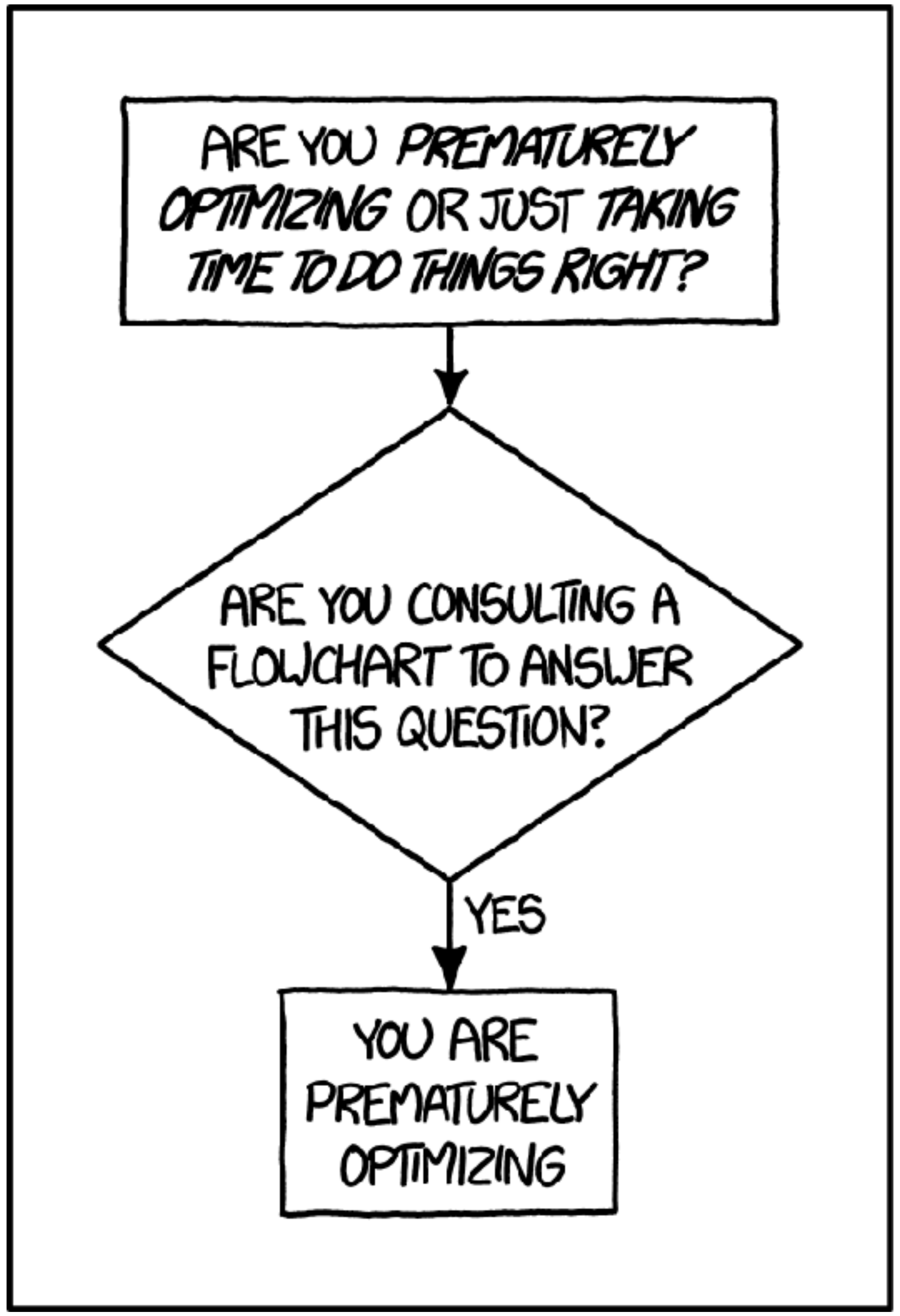
OPTIMIZATION

Excelsior


ARE YOU PREMATURELY OPTIMIZING OR JUST TAKING TIME TO DO THINGS RIGHT?
ARE YOU CONSULTING A FLOWCHART TO ANSWER THIS QUESTION?
YES
YOU ARE PREMATURELY OPTIMIZING

WHAT IS IT?

MAKING THE MOST OF THE RESOURCES YOU HAVE

- Time ← *We usually talk about this*
- Space
- Money
- Electricity
- Brain power
- etc.

but these are important too

CONSIDER THE CONSTRAINTS

Ask your interviewer

FIRST RULE OF OPTIMIZATION:

Don’t do it
Seriously
Don’t

…UNLESS YOU HAVE TO

use “benchmarking” to help you find out when it’s necessary

…OR YOU HAVE IMPORTANT INFO AHEAD OF TIME ABOUT HOW YOUR PROGRAM IS GOING TO BE USED

- input size
- rate of requests
- how many other things will rely on it
- etc.

…OR THERE ARE REALLY EASY WINS YOU CAN GET WITHOUT EXPENDING MUCH TIME OR EFFORT

SO…
HOW DO WE GO ABOUT THIS?

DECIDE WHAT YOU’RE OPTIMIZING FOR

YOU CAN'T OPTIMIZE ALL THE THINGS

- Time
- Space
- Money
- Electricity
- Brain power
- etc.

pick one (or two, but don't be greedy)

HOW DO WE DECIDE WHAT TO OPTIMIZE?

IDENTIFY THE BOTTLENECK

Think about the environment you’re developing for

Ask your interviewer

bottleneck

making this part wider
won’t help you pour faster

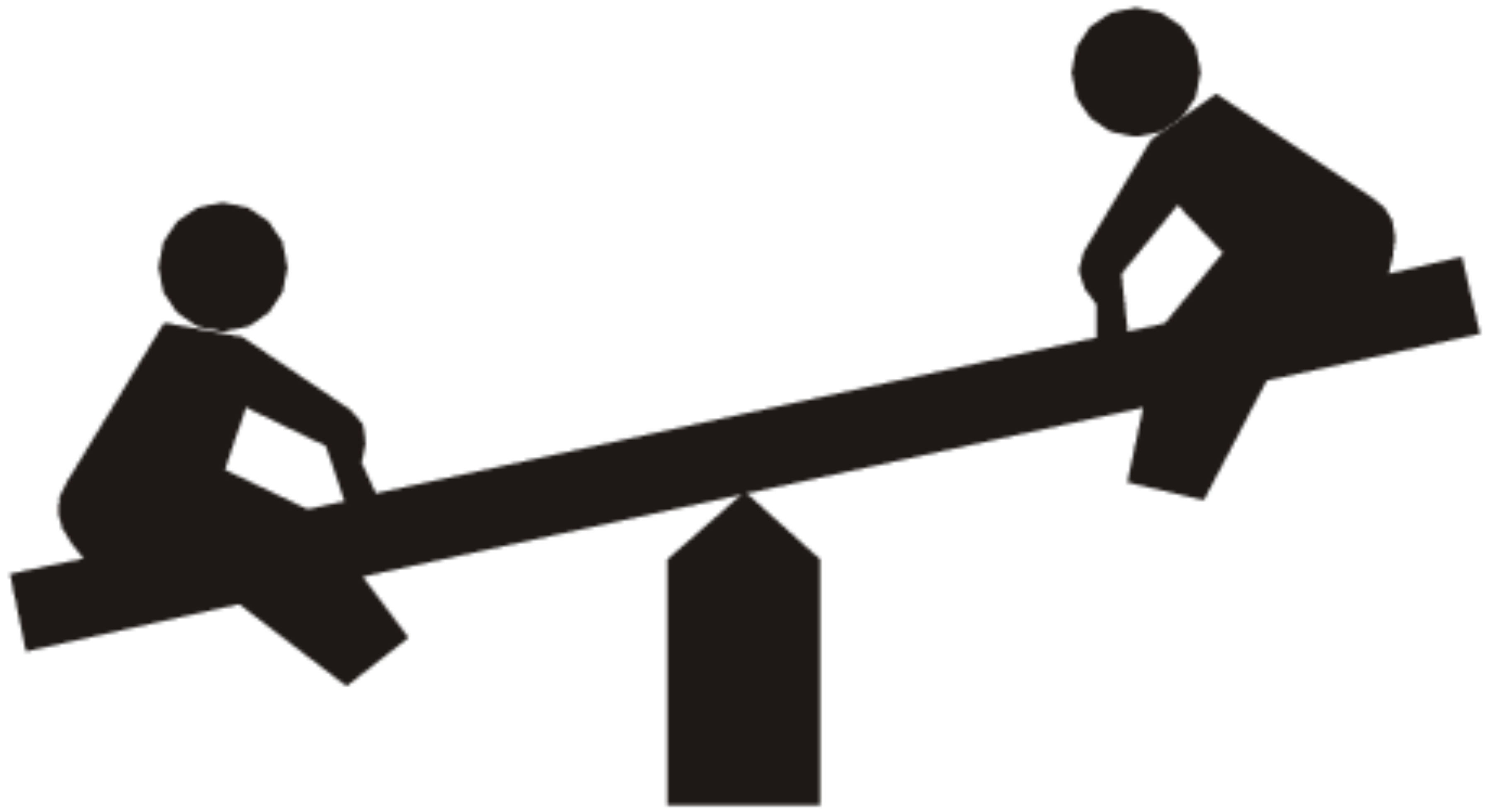
FOCUS ON WIDENING THE BOTTLENECK

- **apply this recursively**
  - “What’s our scarcest resource? Time? Space?”
  - “Time is our scarcest resource. Which part of our program is taking the most time?” (use benchmarking)
  - “This utility function is taking the most time. What’s the Big O? Which part of the function is taking the most time? Can it be improved?”
- **go around bottlenecks you don’t have much control over**
  - “Network latency is our bottleneck. Let’s try to minimize the size and frequency of our API calls.”

EXAMPLE PROBLEM #1:

Write a function that returns true if any permutation of a string is a palindrome.

OPTIMIZATION GENERALLY INVOLVES TRADE-OFFS



SPACE FOR TIME IS THE MOST COMMON TRADEOFF

HOW DO WE SAVE TIME AT THE EXPENSE OF SPACE?

DATA STRUCTURES! 🎉

PRO TIP #1:
USE A HASH TABLE

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

```
const couldBePalindrome = str => {
  const m = new Map()
  for (let i = 0; i < str.length; i++) {
    const ch = str[i]
    if (m.has(ch)) {
      m.delete(ch)
    } else {
      m.set(ch, true)
    }
  }
  return m.size <= 1
}
```

PRO TIP #2:
USE BINARY SEARCH

EXAMPLE PROBLEM #2:

Write a function which takes in a number and a sorted array of numbers. Return true if any 2 numbers could add up to the number passed in.

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
    const sum = sortedArr[left] + sortedArr[right]
    if (sum === n) {
      return true
    } else if (sum > n) {
      right--
    } else {
      left++
    }
  }
  return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
     const sum = sortedArr[left] + sortedArr[right]
     if (sum === n) {
       return true
     } else if (sum > n) {
       right--
     } else {
       left++
     }
   }
   return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
    const sum = sortedArr[left] + sortedArr[right]
    if (sum === n) {
      return true
    } else if (sum > n) {
      right--
    } else {
      left++
    }
  }
  return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
    const sum = sortedArr[left] + sortedArr[right]
    if (sum === n) {
      return true
    } else if (sum > n) {
      right--
    } else {
      left++
    }
  }
  return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
    const sum = sortedArr[left] + sortedArr[right]
    if (sum === n) {
      return true
    } else if (sum > n) {
      right--
    } else {
      left++
    }
  }
  return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
     const sum = sortedArr[left] + sortedArr[right]
     if (sum === n) {
       return true
     } else if (sum > n) {
       right--
     } else {
       left++
     }
   }
   return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
    const sum = sortedArr[left] + sortedArr[right]
    if (sum === n) {
      return true
    } else if (sum > n) {
      right--
    } else {
      left++
    }
  }
  return false
}
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
     const sum = sortedArr[left] + sortedArr[right]
     if (sum === n) {
       return true
     } else if (sum > n) {
       right--
     } else {
       left++
     }
   }
   return false
 }
```

```
const pairSum = (n, sortedArr) => {
  let left = 0
  let right = sortedArr.length - 1
  while (right !== left) {
     const sum = sortedArr[left] + sortedArr[right]
     if (sum === n) {
       return true
     } else if (sum > n) {
       right--
     } else {
       left++
     }
   }
   return false
 }
```

DYNAMIC PROGRAMMING

Breaking a big problem down into smaller sub-problems and solving those instead

Think recursion!

MEMOIZATION

Storing the results of previous function invocations for easy (fast) future access

EXAMPLE PROBLEM #3: FIBONACCI

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

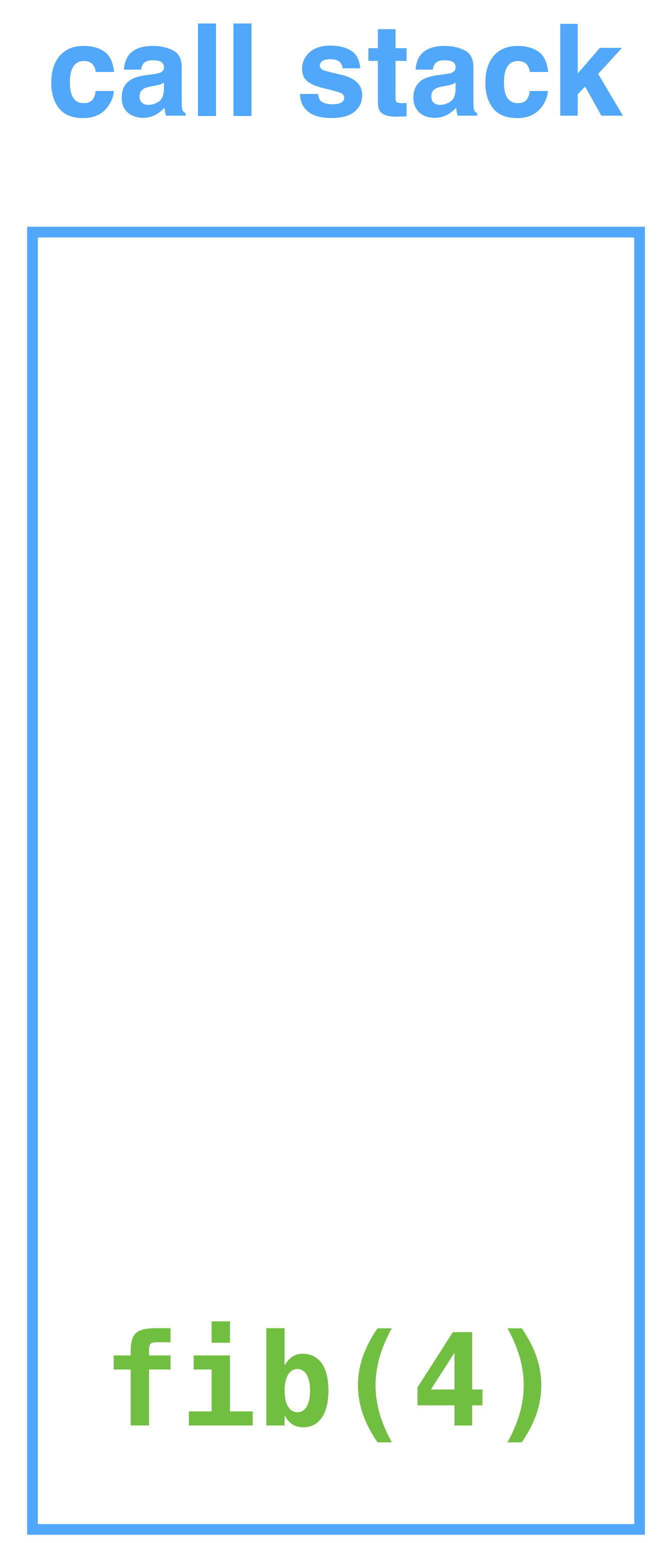
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

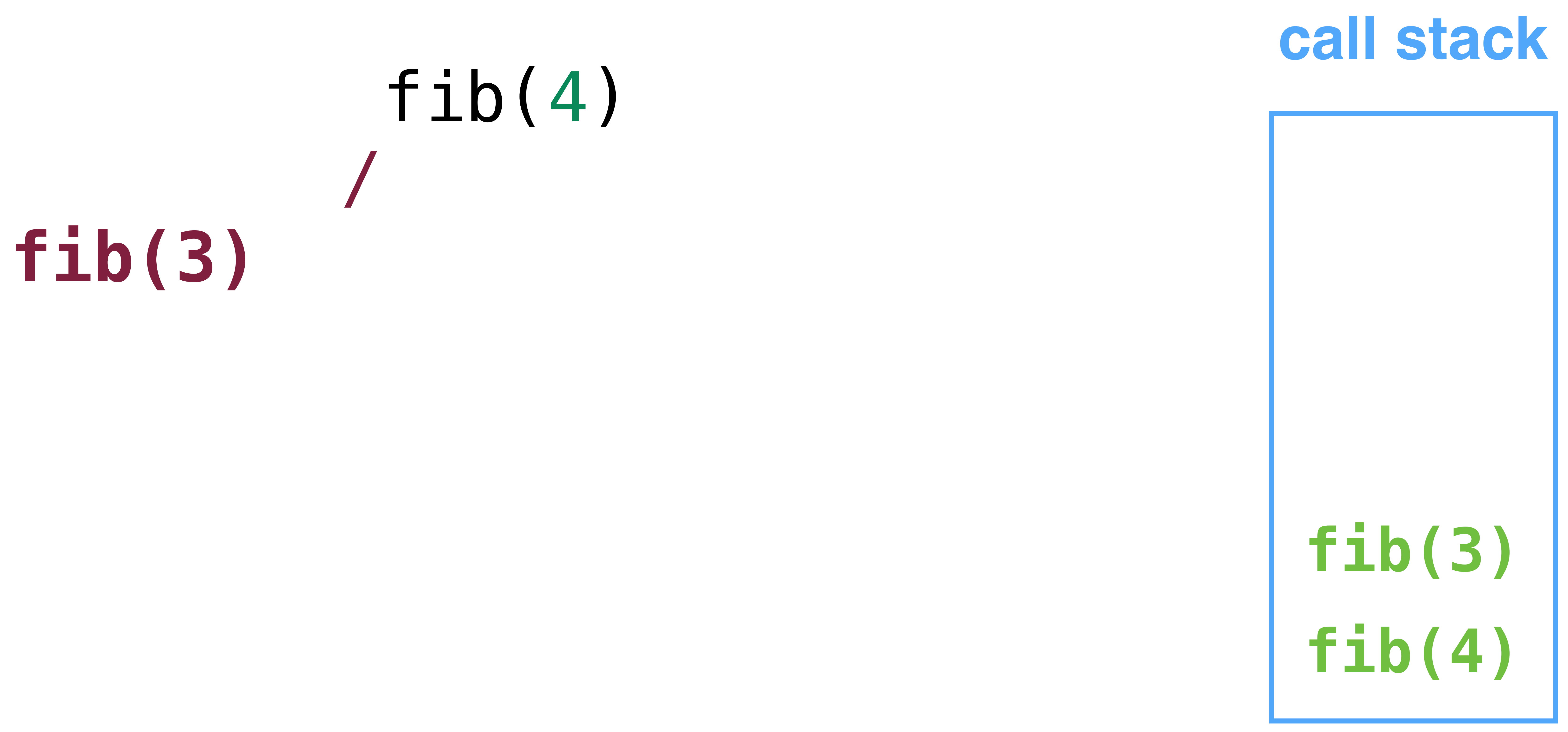
call stack

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

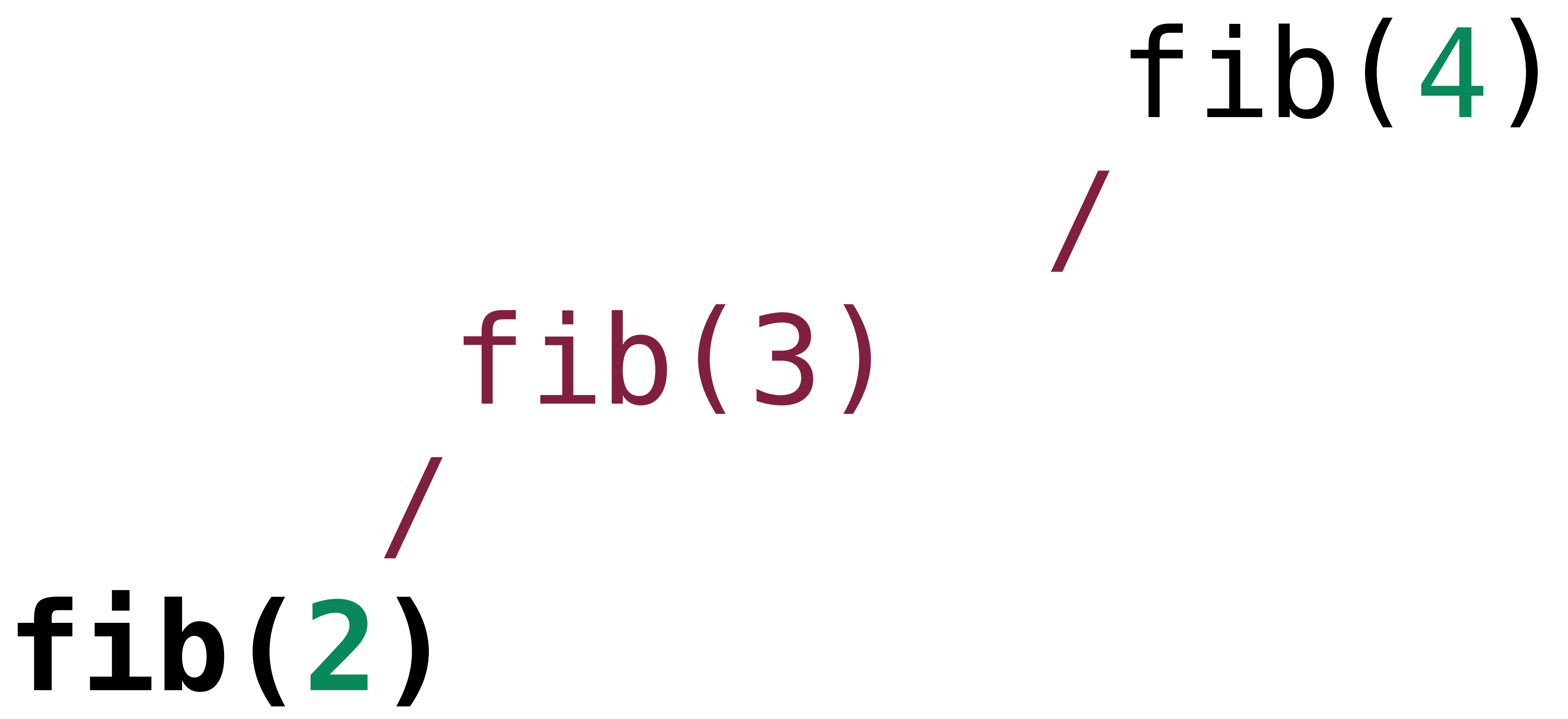
fib(4)

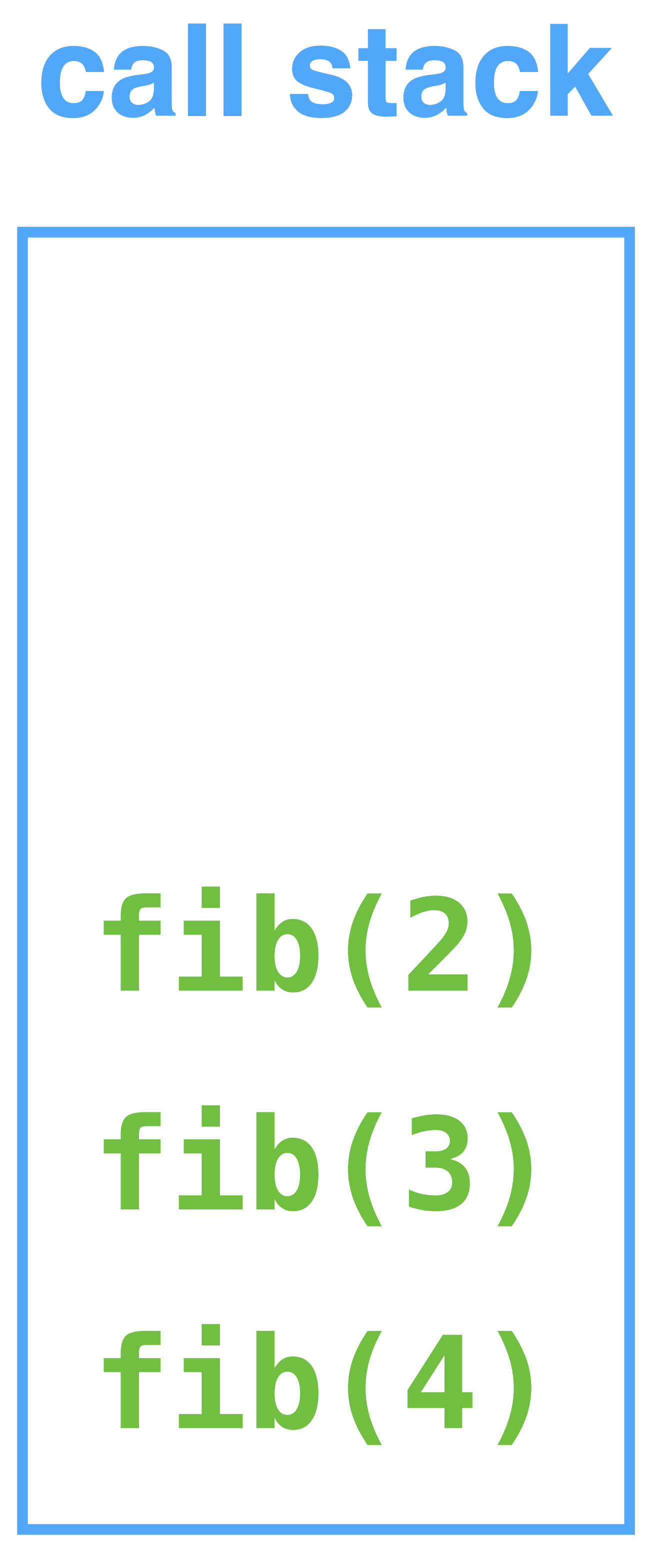


```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

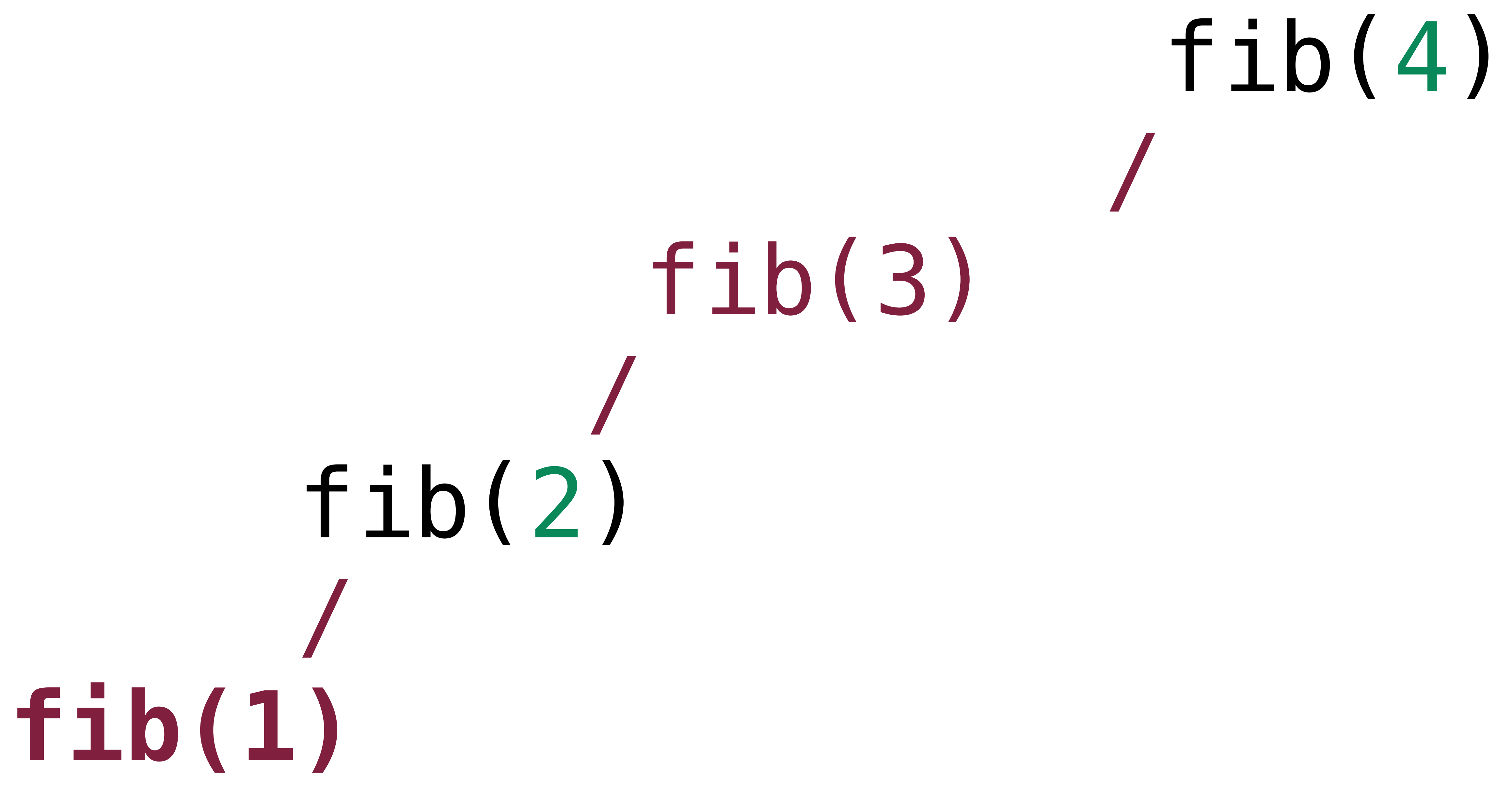


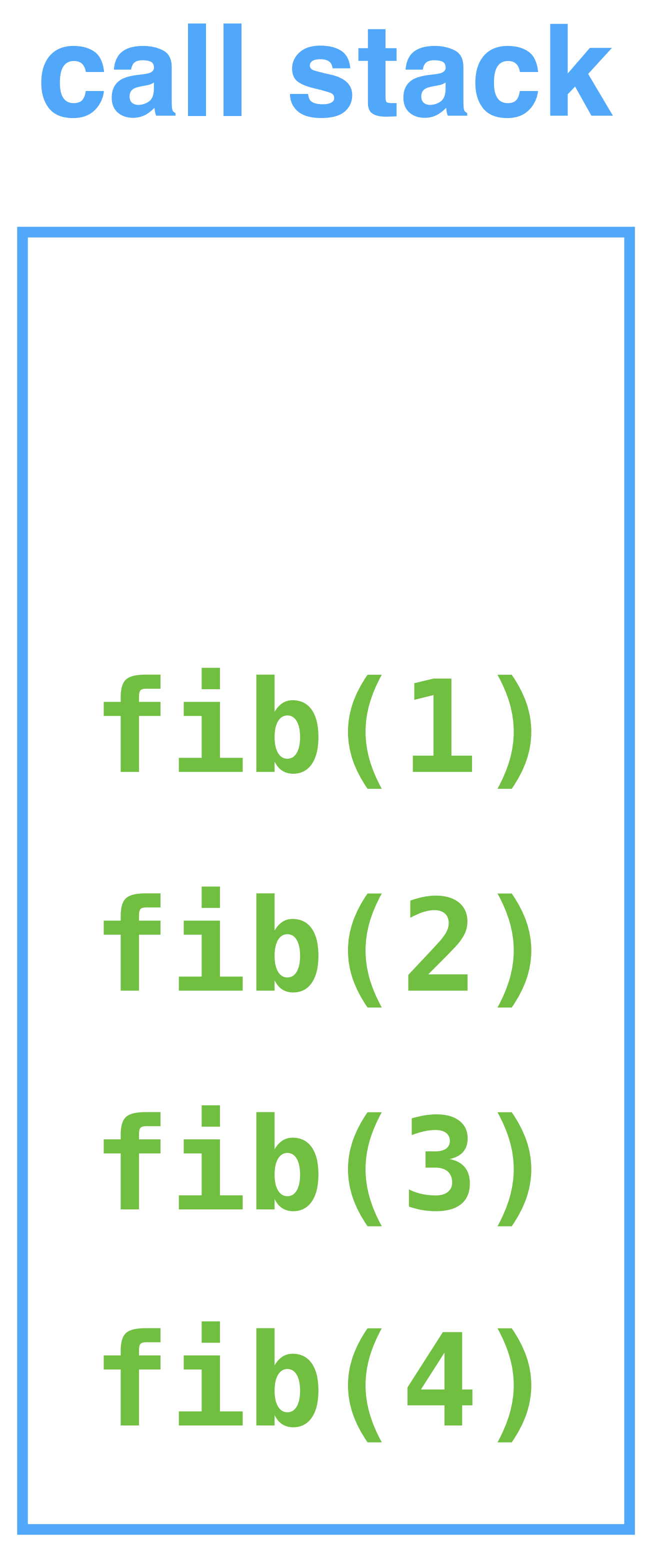
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```



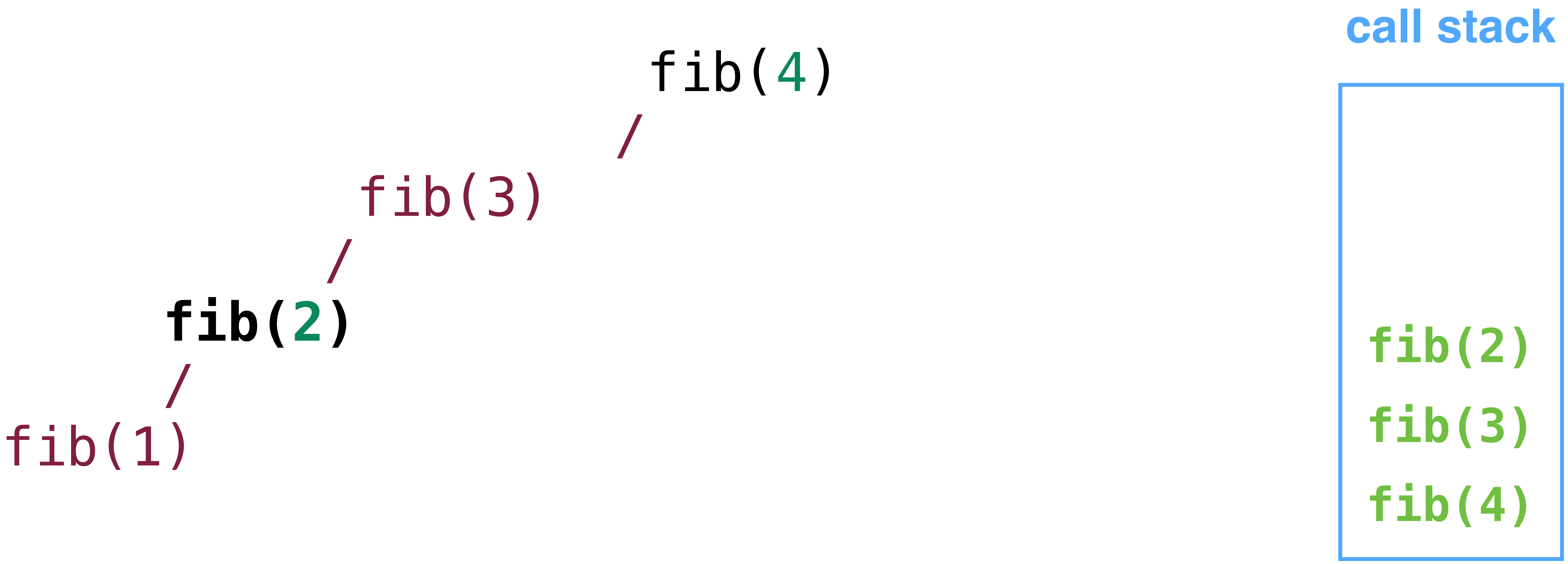


```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

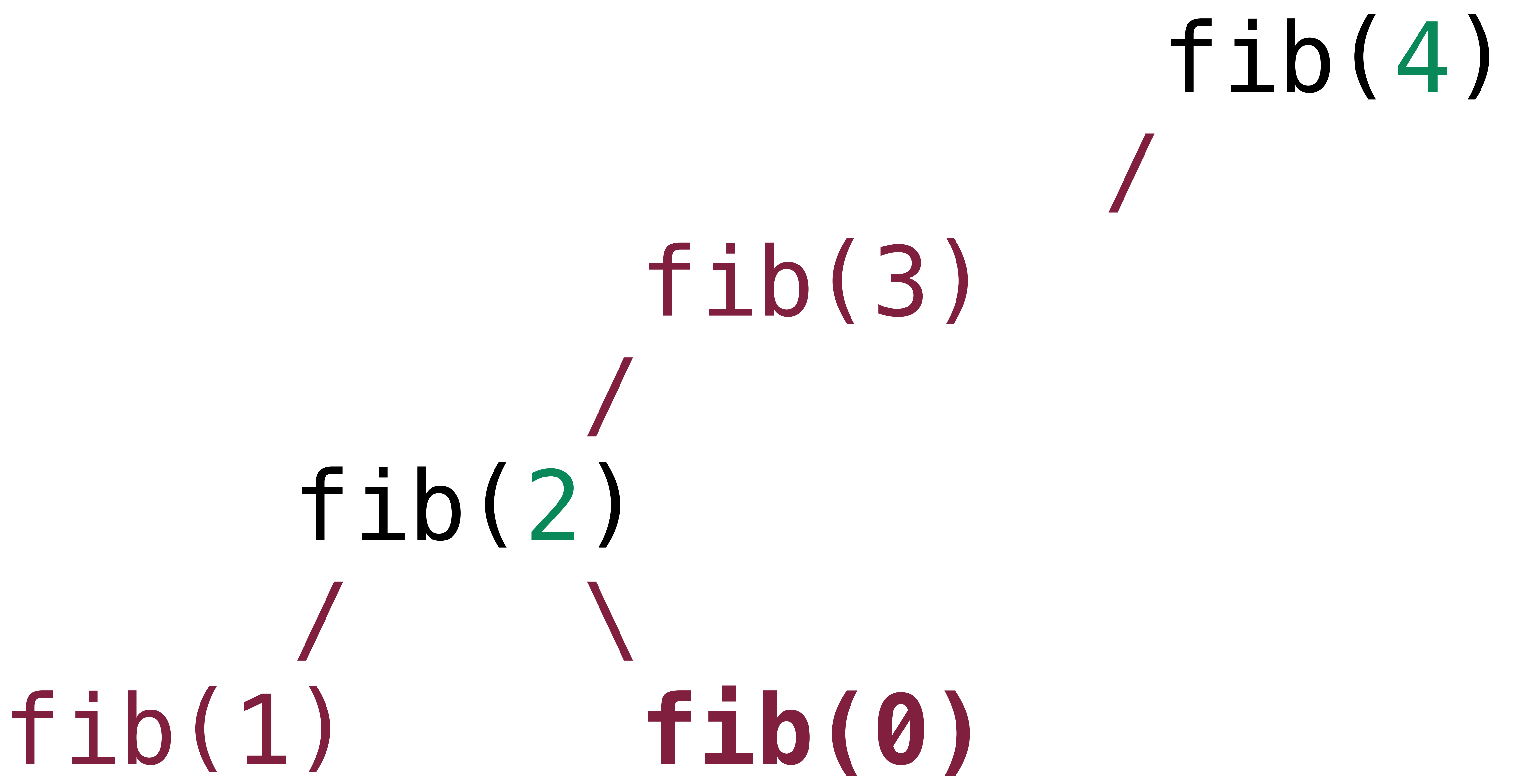




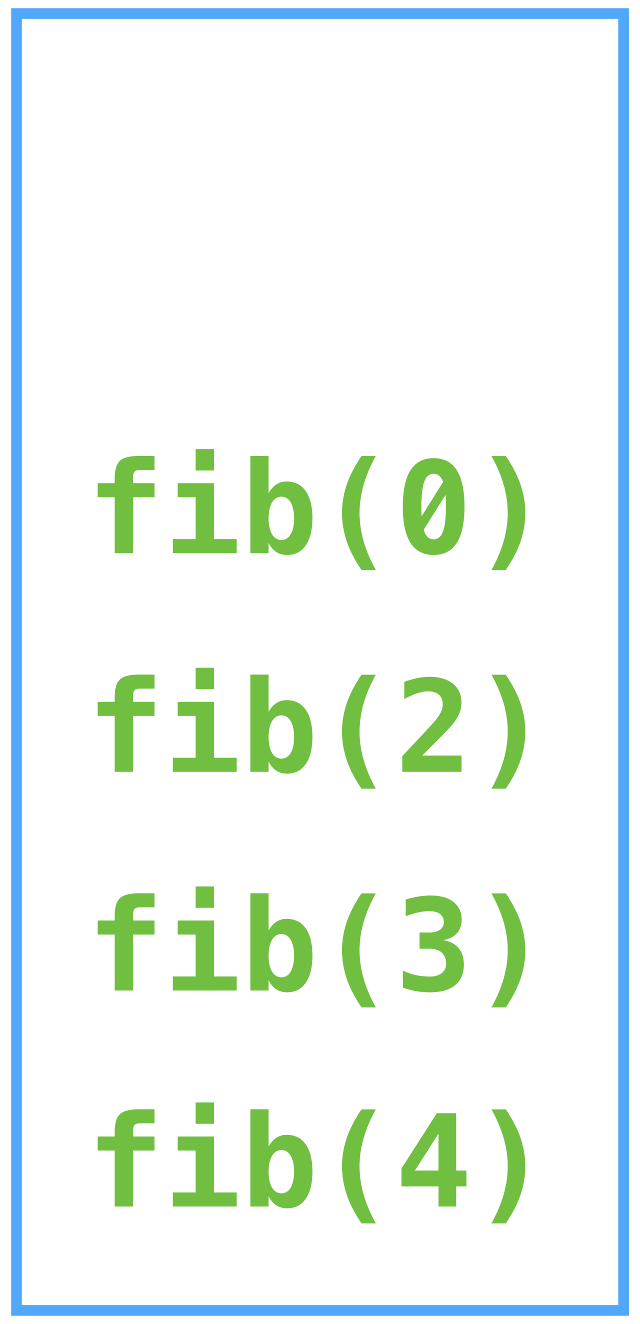
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```



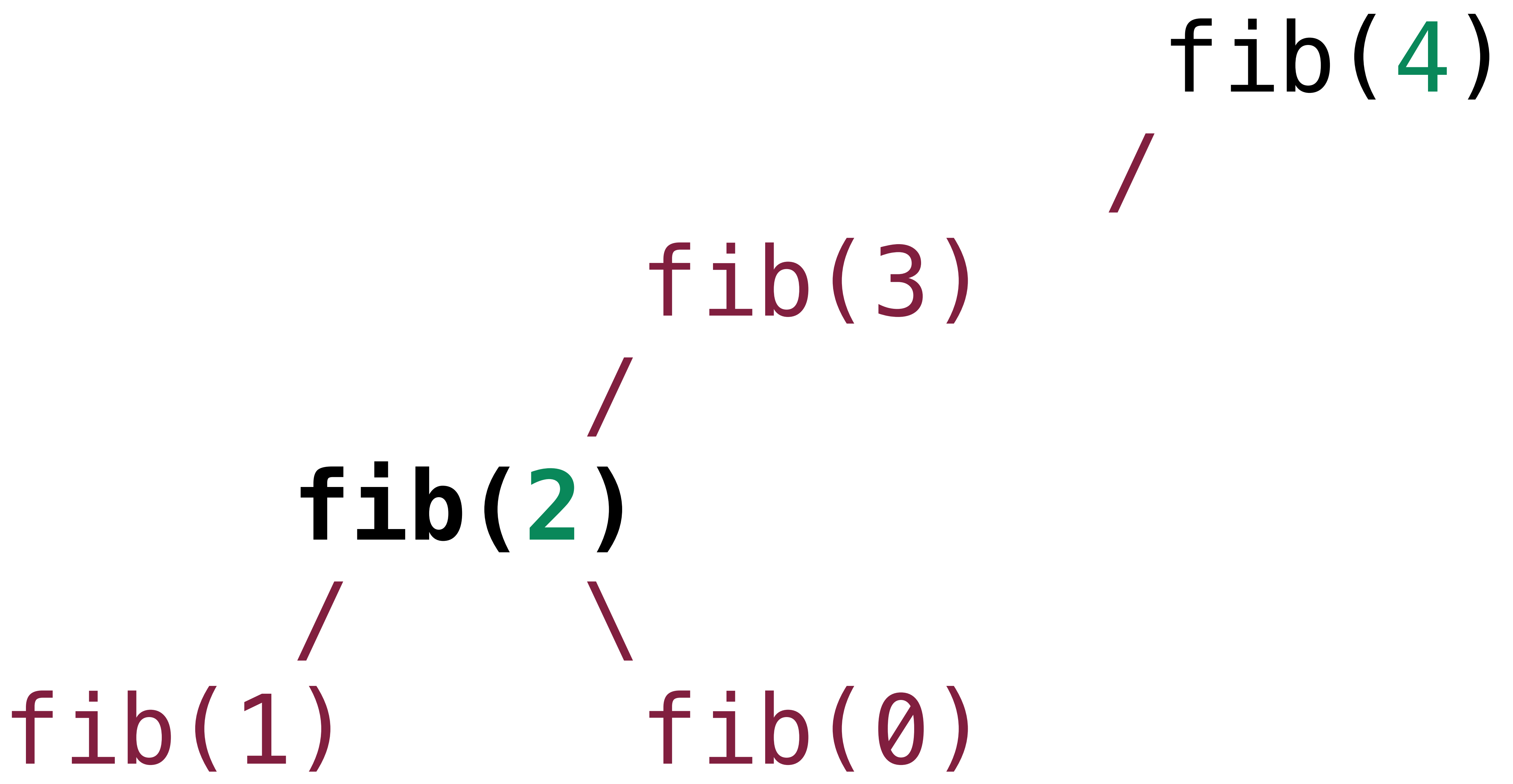
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

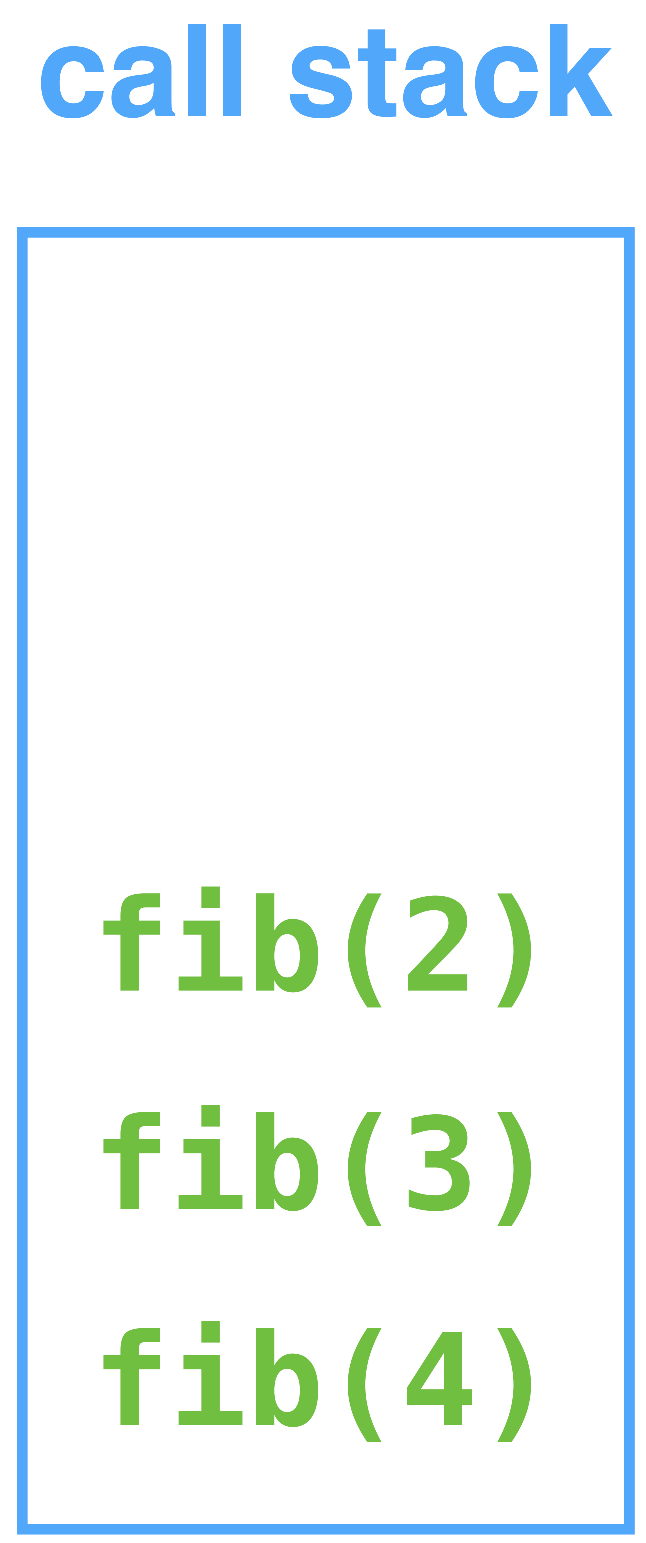


call stack

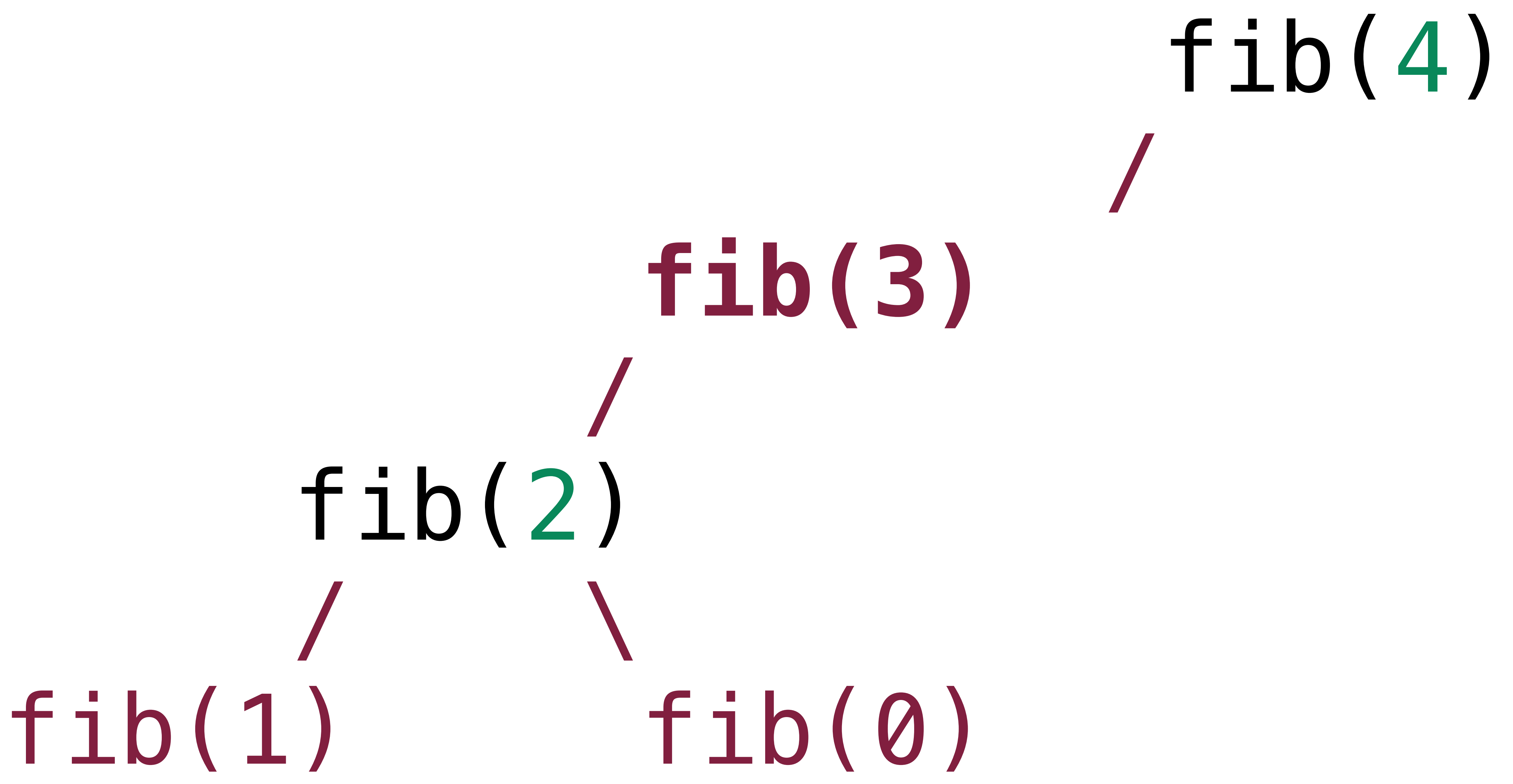


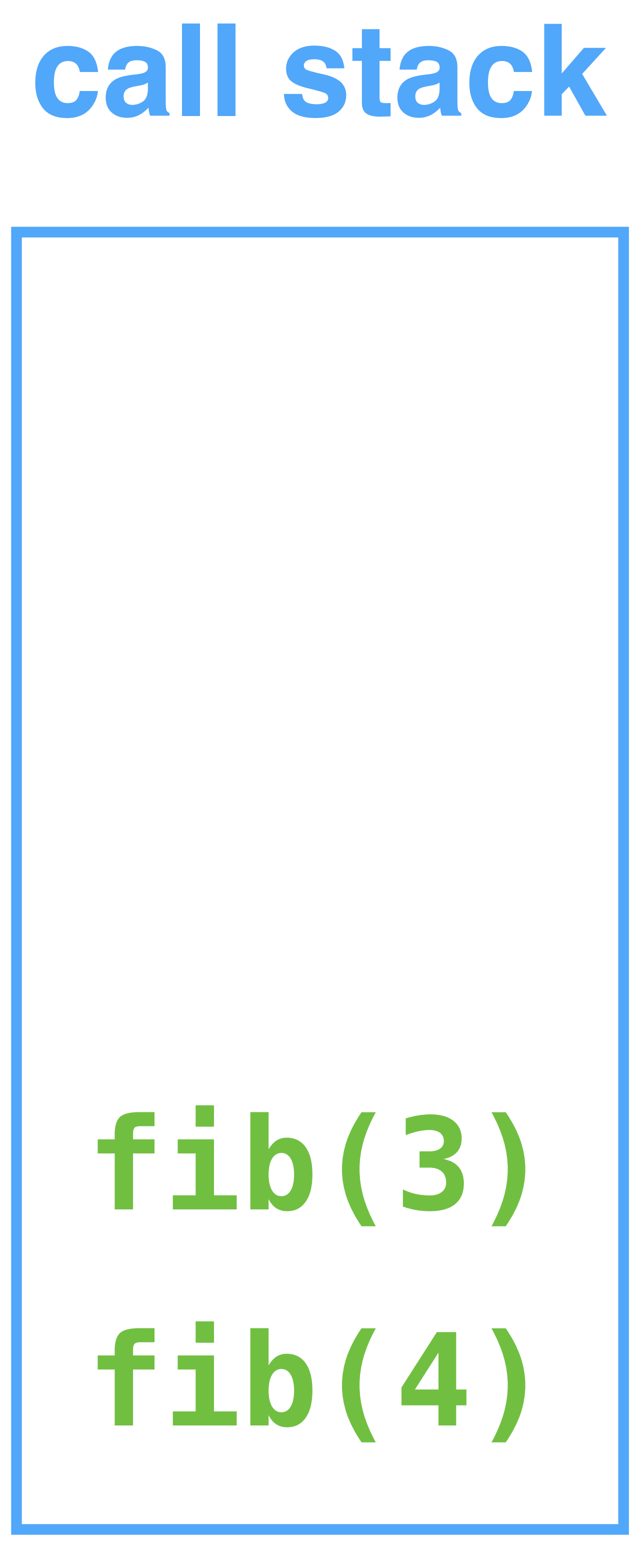
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```



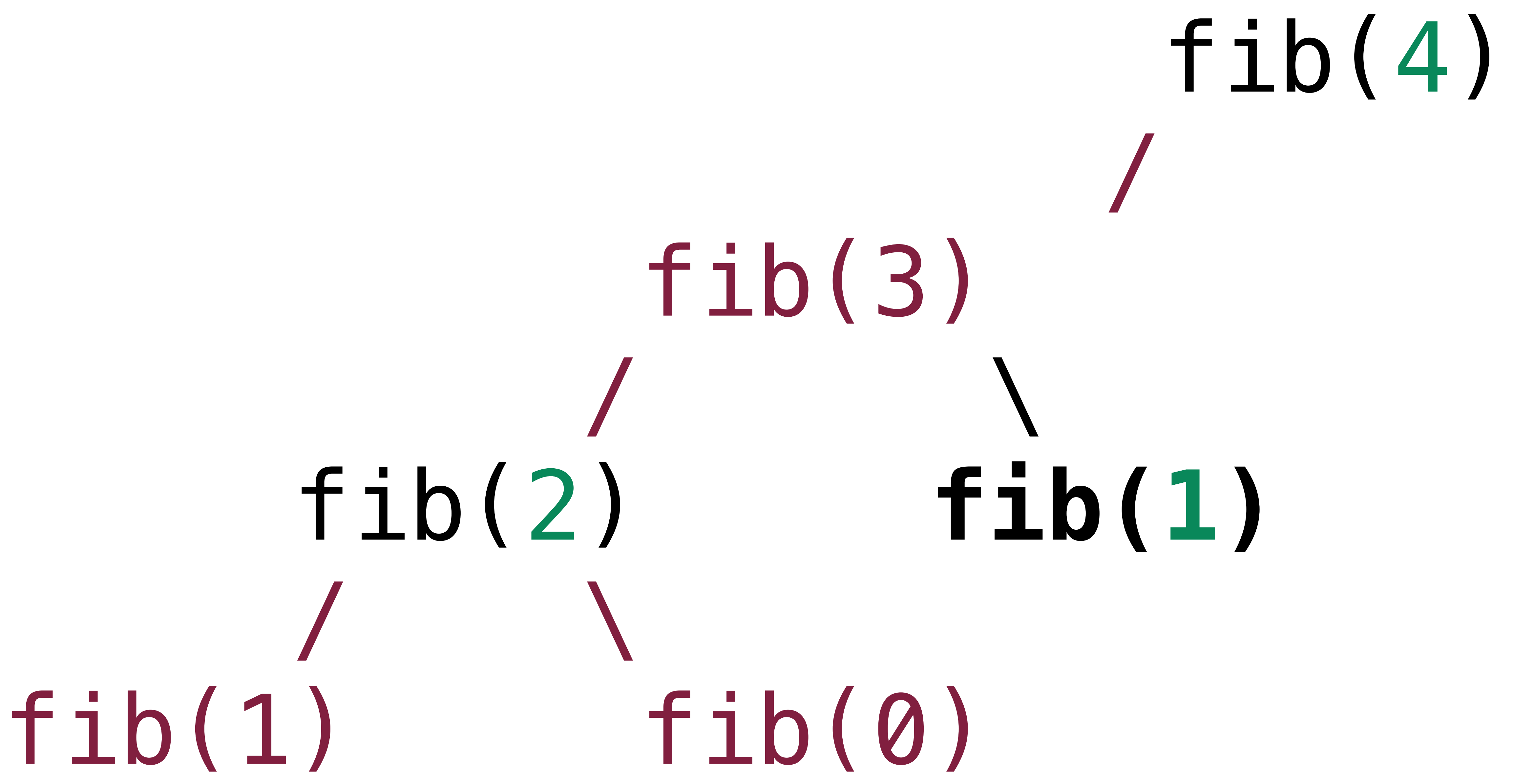


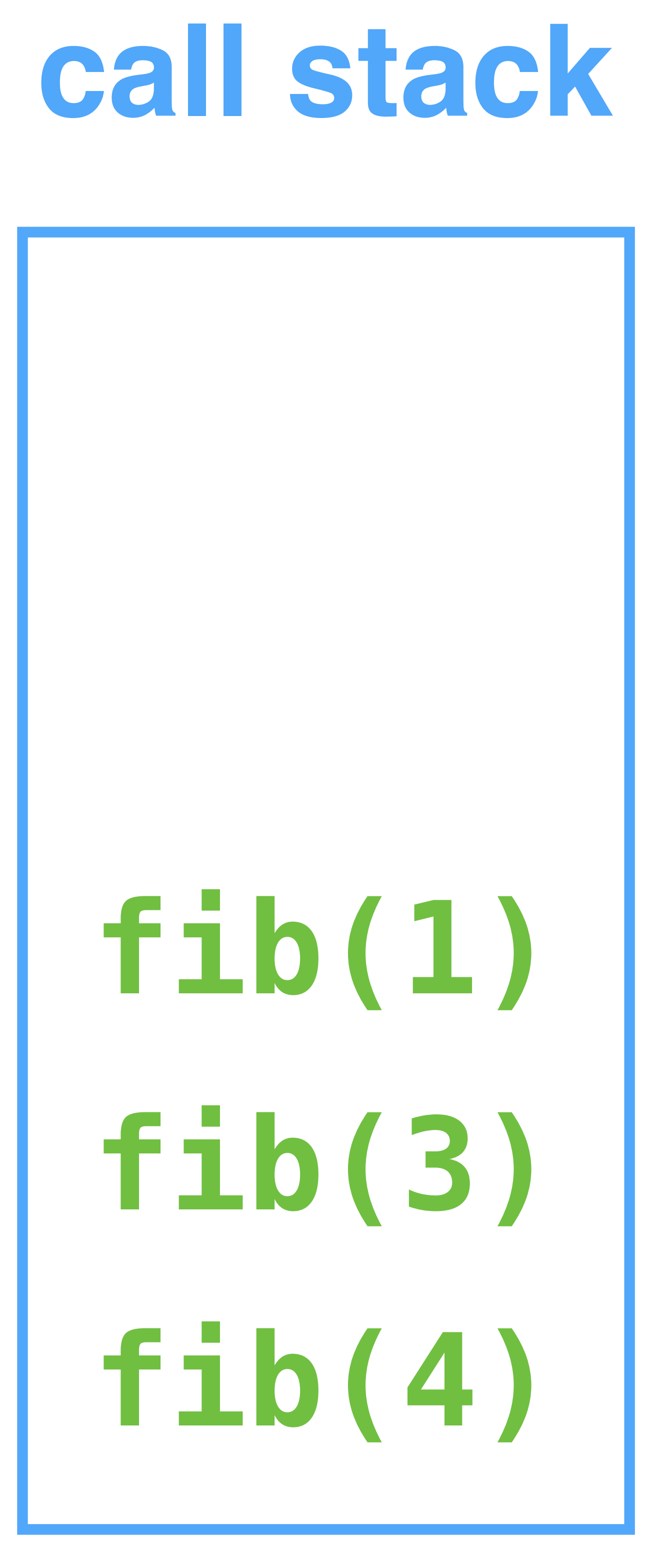
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```



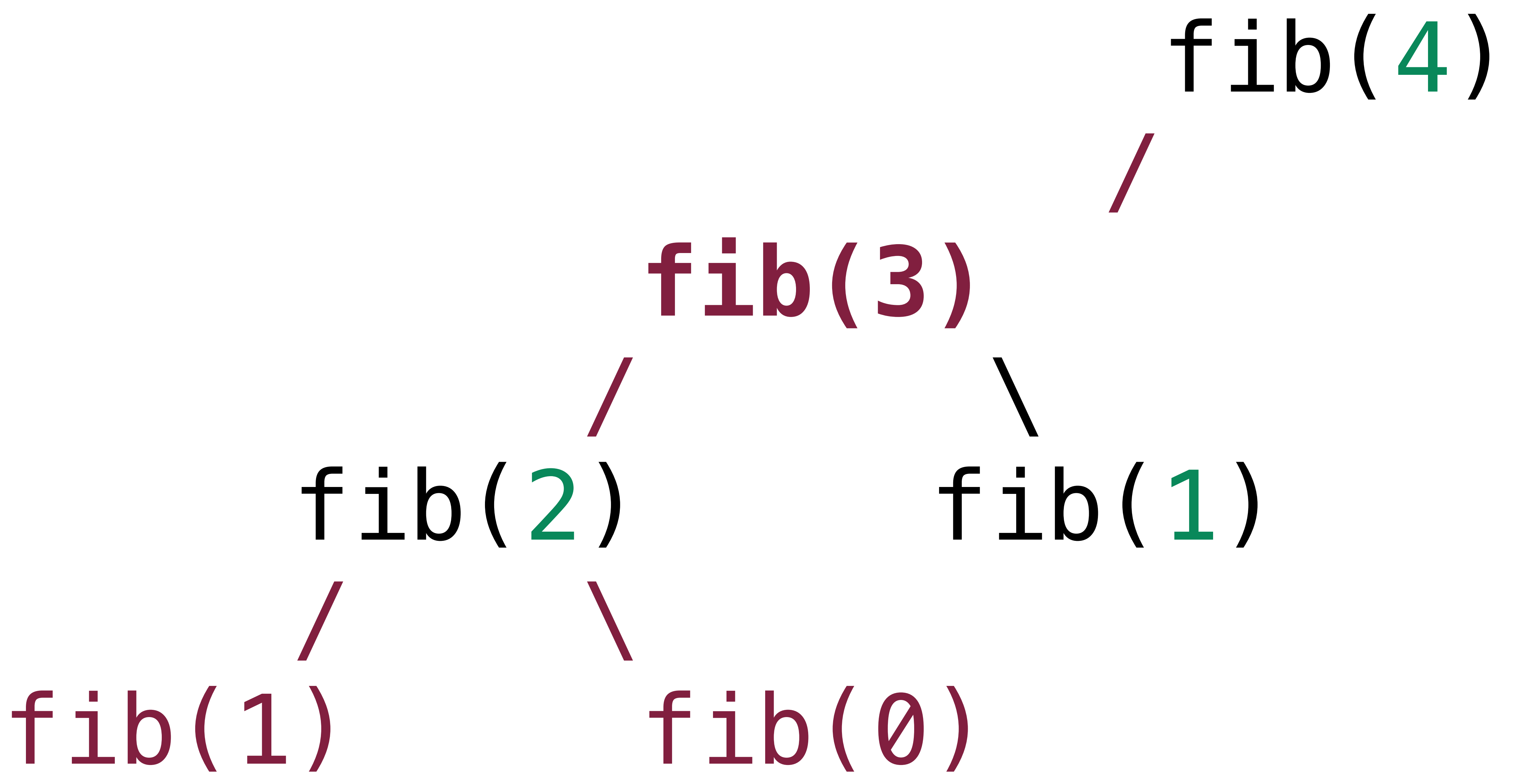


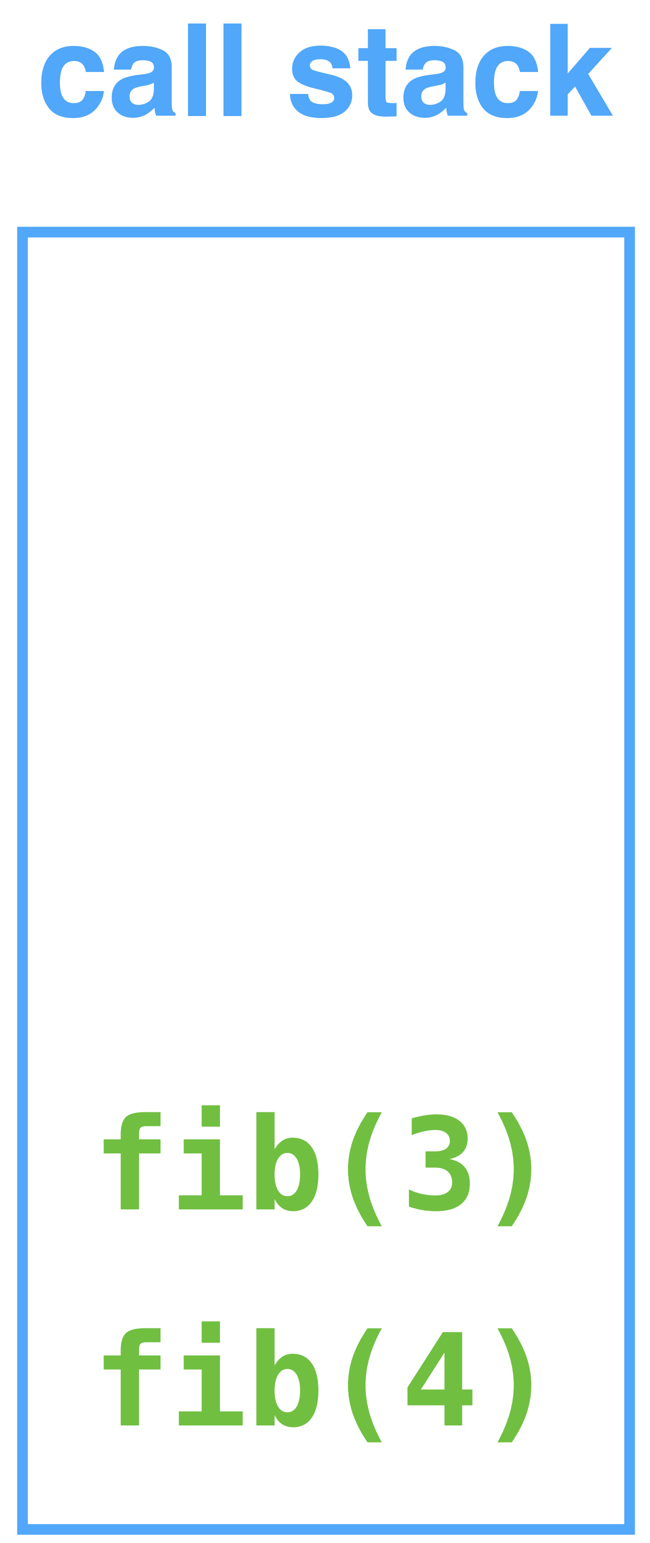
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```



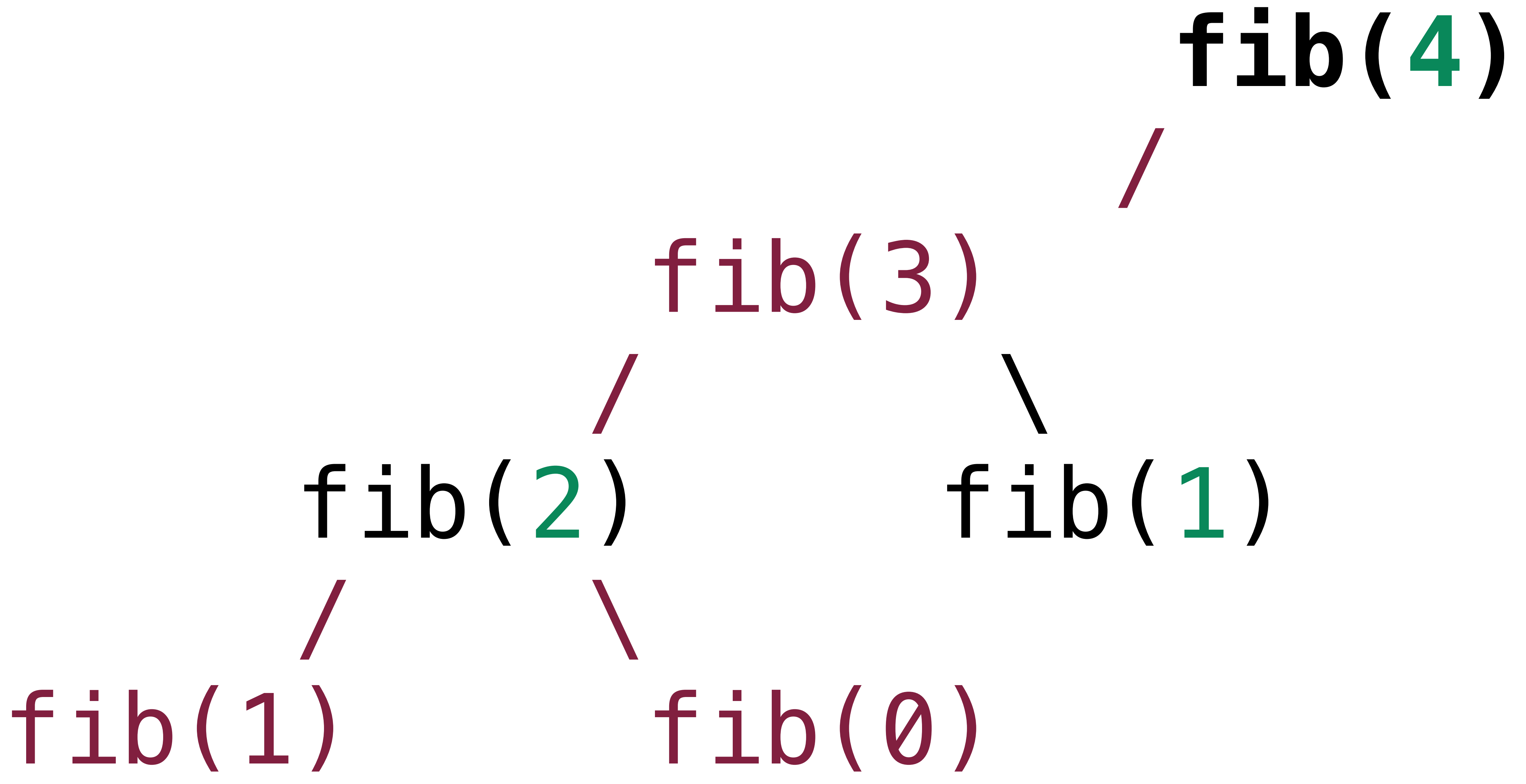


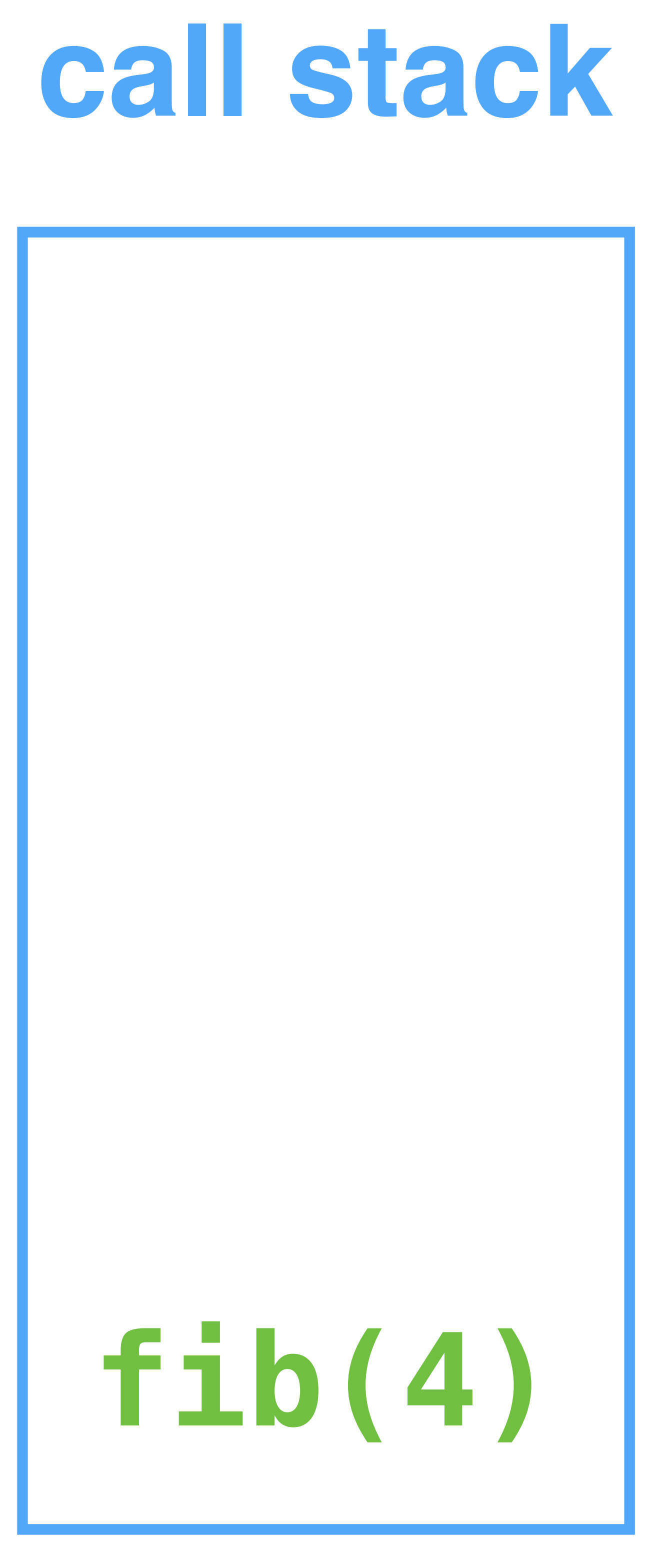
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```





```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```





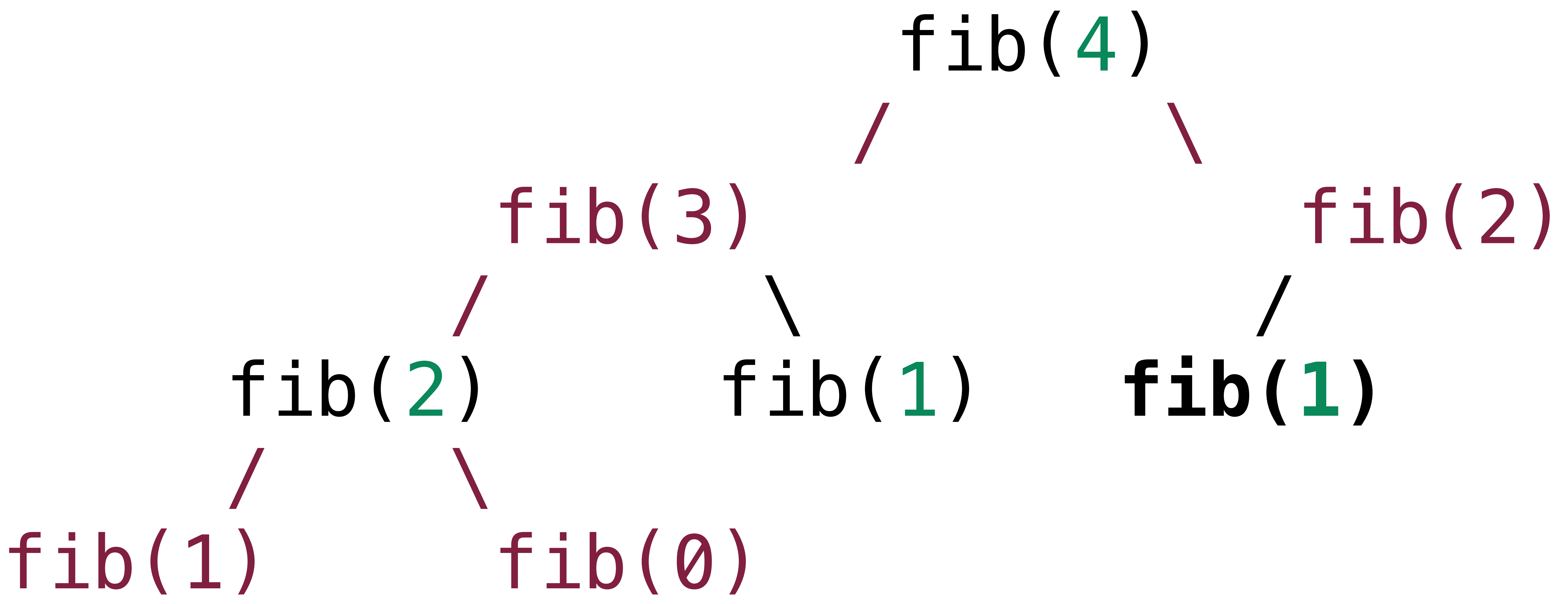
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

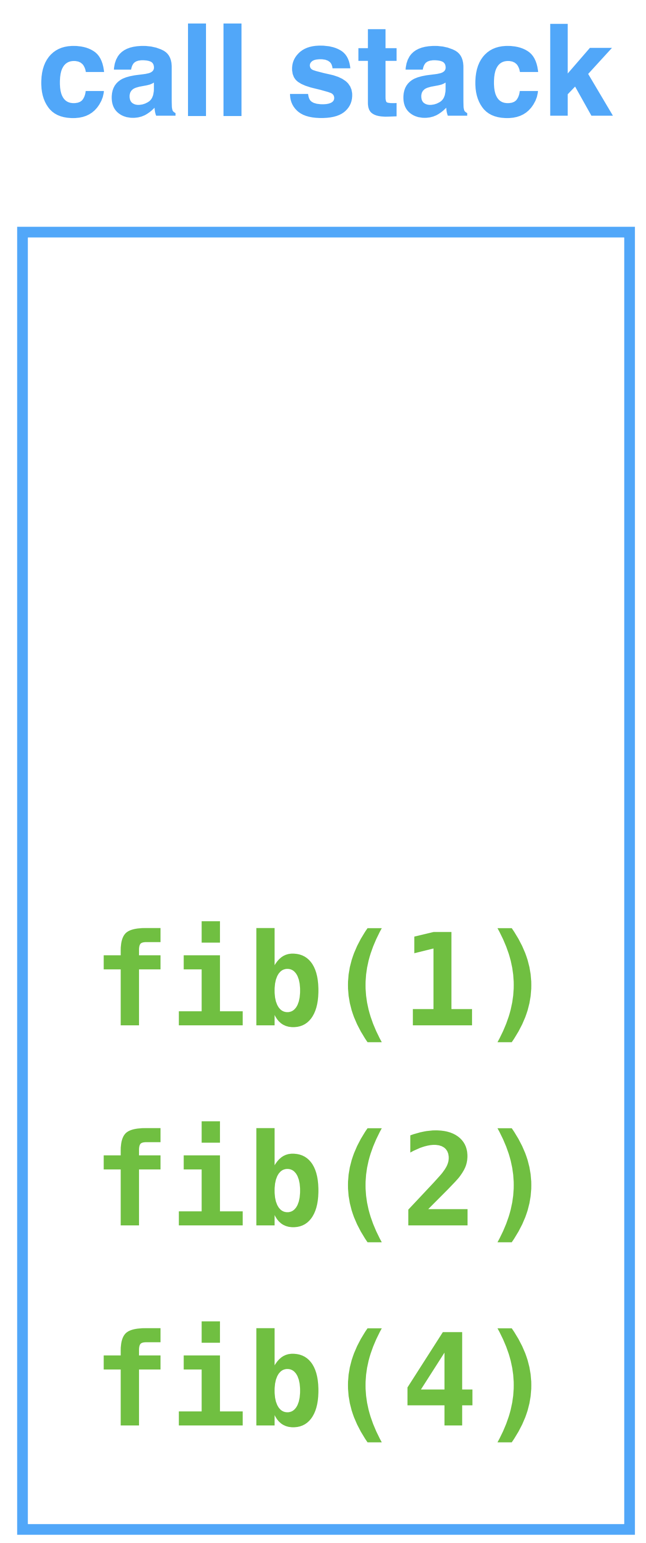
```
                             fib(4)
                            /      \
                    fib(3)              fib(2)
                   /      \
             fib(2)        fib(1)
            /      \
      fib(1)        fib(0)
```

call stack

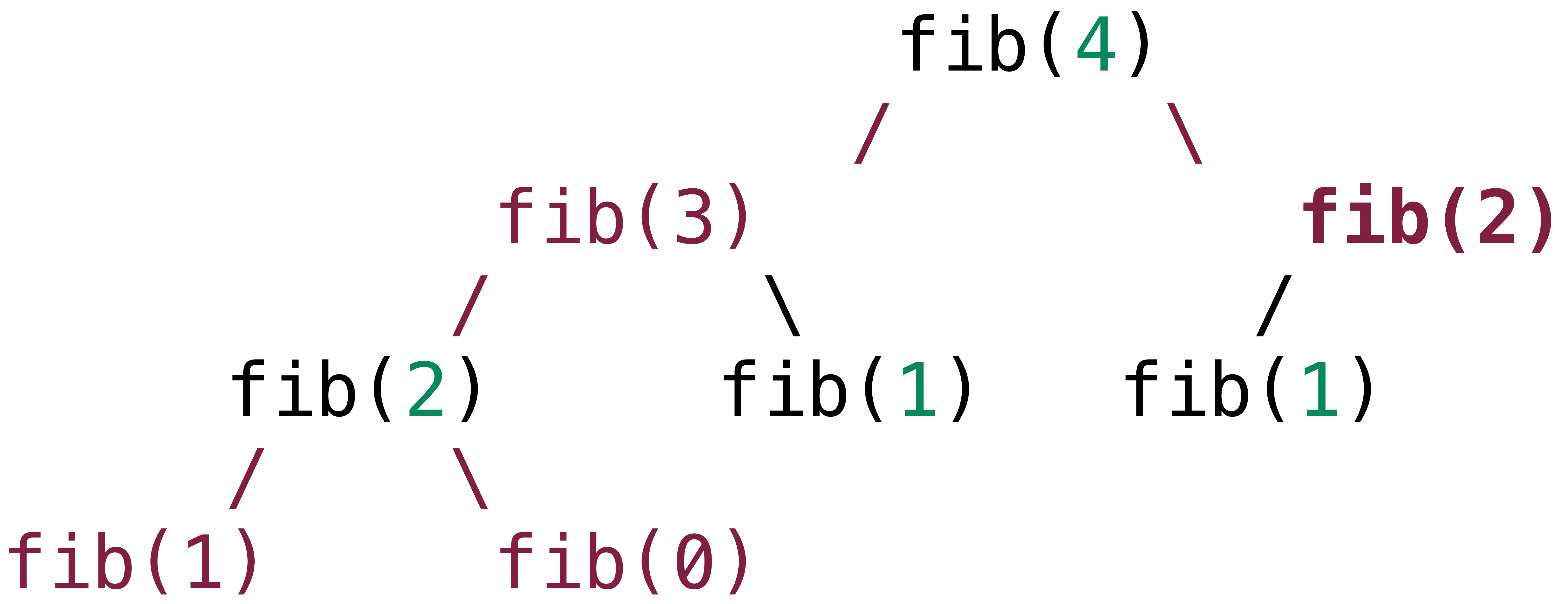
```
fib(2)
fib(4)
```

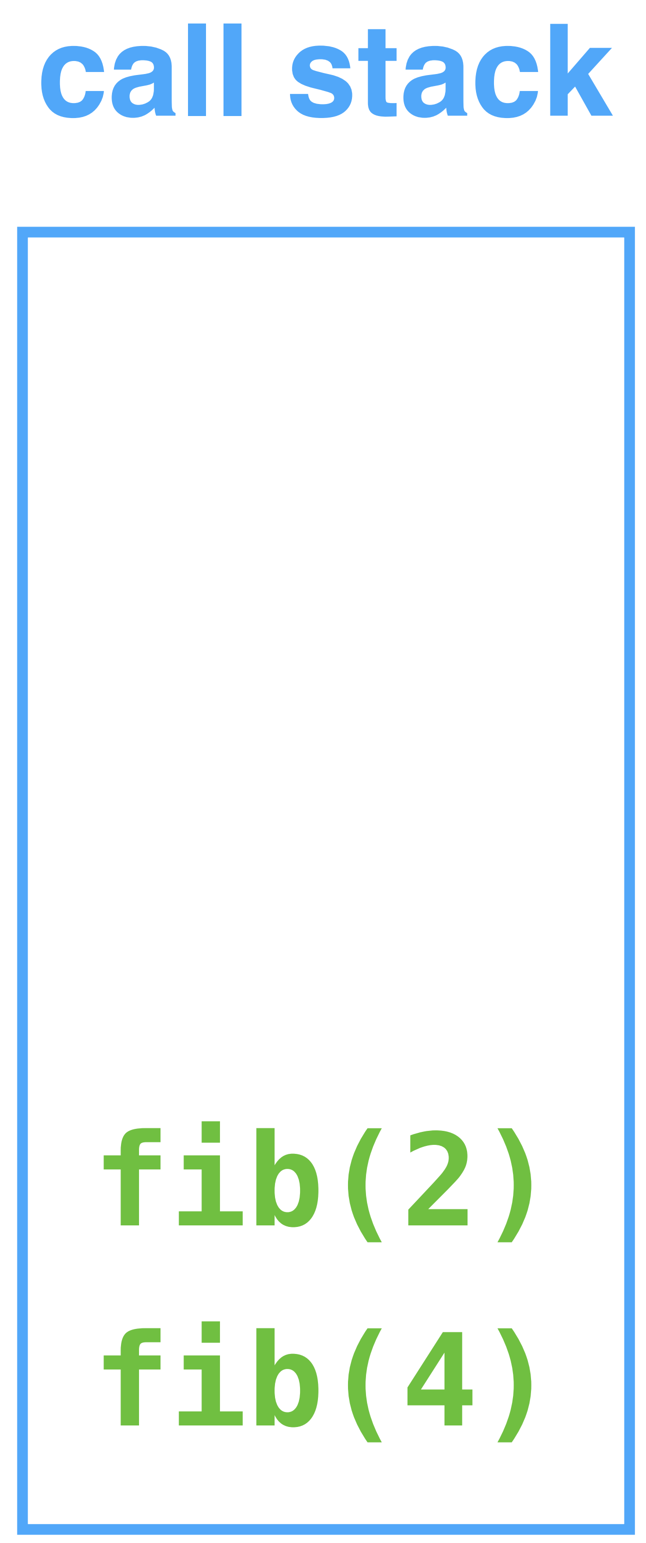
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```





```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```





```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                     fib(4)
                    /      \
              fib(3)              fib(2)
             /      \            /      \
        fib(2)     fib(1)    fib(1)     fib(0)
       /      \
   fib(1)     fib(0)
```

call stack

```
fib(0)
fib(2)
fib(4)
```

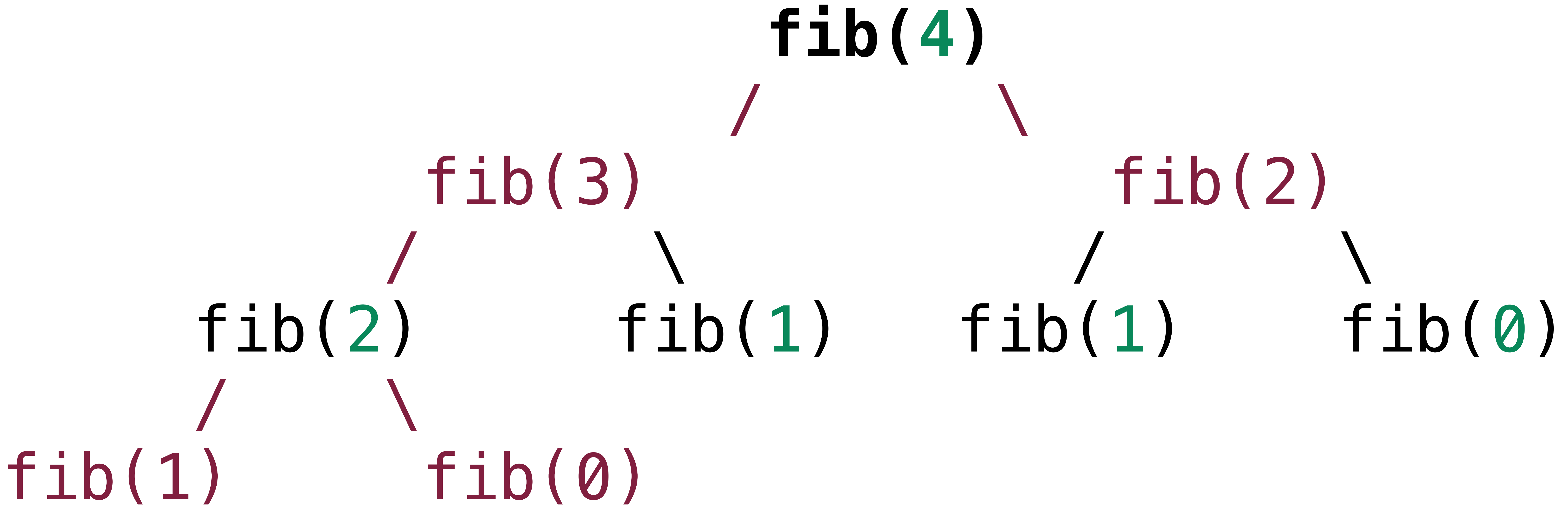
```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

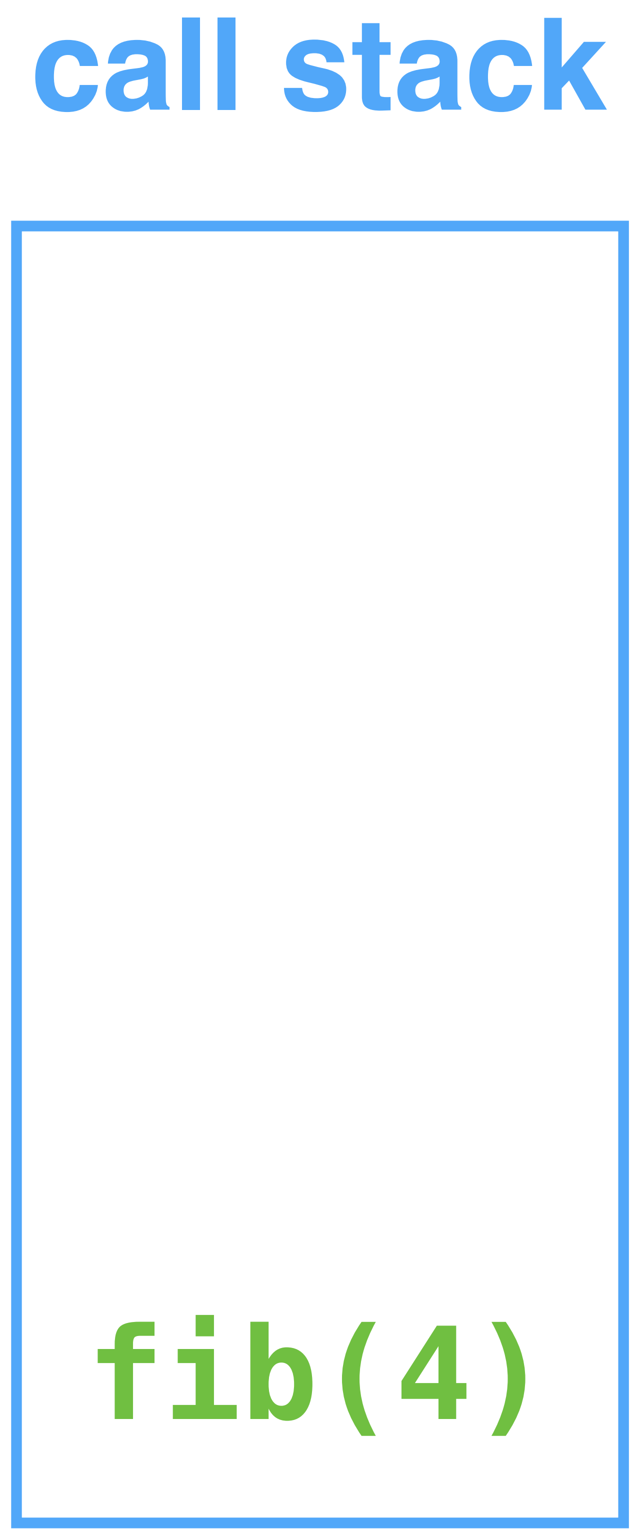
```
                         fib(4)
                        /      \
                fib(3)              fib(2)
               /      \            /      \
          fib(2)      fib(1)   fib(1)     fib(0)
         /     \
    fib(1)     fib(0)
```

call stack

```
fib(2)
fib(4)
```

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n – 1) + fib(n – 2);
}
```





```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

```
                    fib(4)
                  /        \
            fib(3)              fib(2)
           /      \            /      \
      fib(2)      fib(1)   fib(1)     fib(0)
     /      \
fib(1)      fib(0)
```

call stack

```
function fib (n) {
  if (n === 1 || n === 0) return n;
  else return fib(n - 1) + fib(n - 2);
}
```

our input is equal to 4: n = 4

we go four levels deep, so depth = n

we branch twice with each recursive call

therefore, runtime is O(2^n)!

```
                        fib(4)
                       /      \
              fib(3)                fib(2)
             /      \              /      \
       fib(2)        fib(1)    fib(1)      fib(0)
      /      \
fib(1)        fib(0)
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```javascript
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

call stack

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

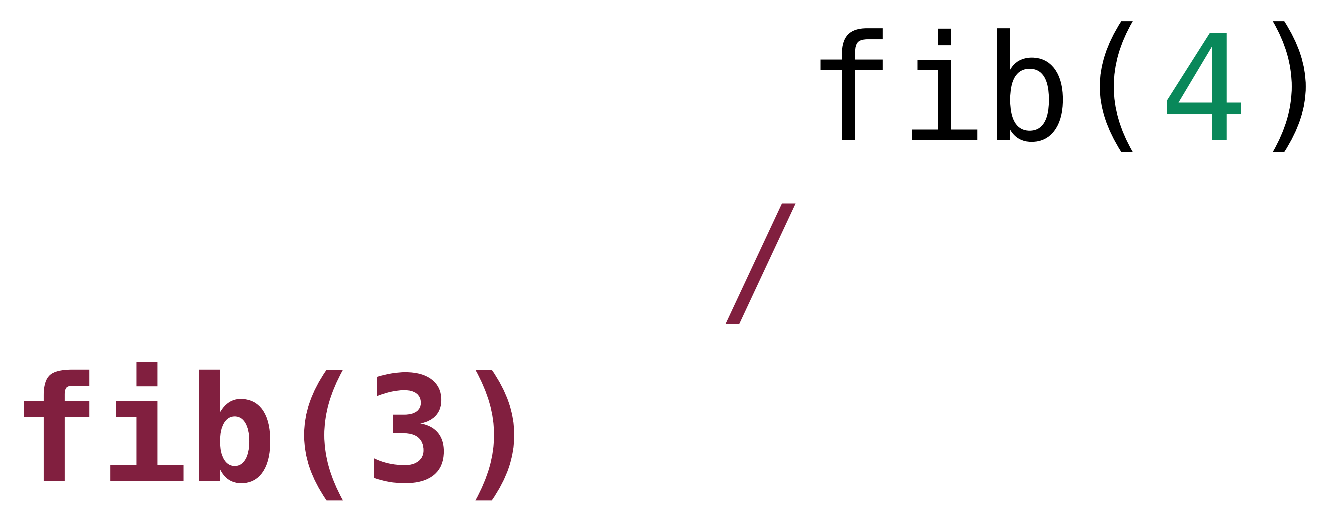
fib(4)

call stack

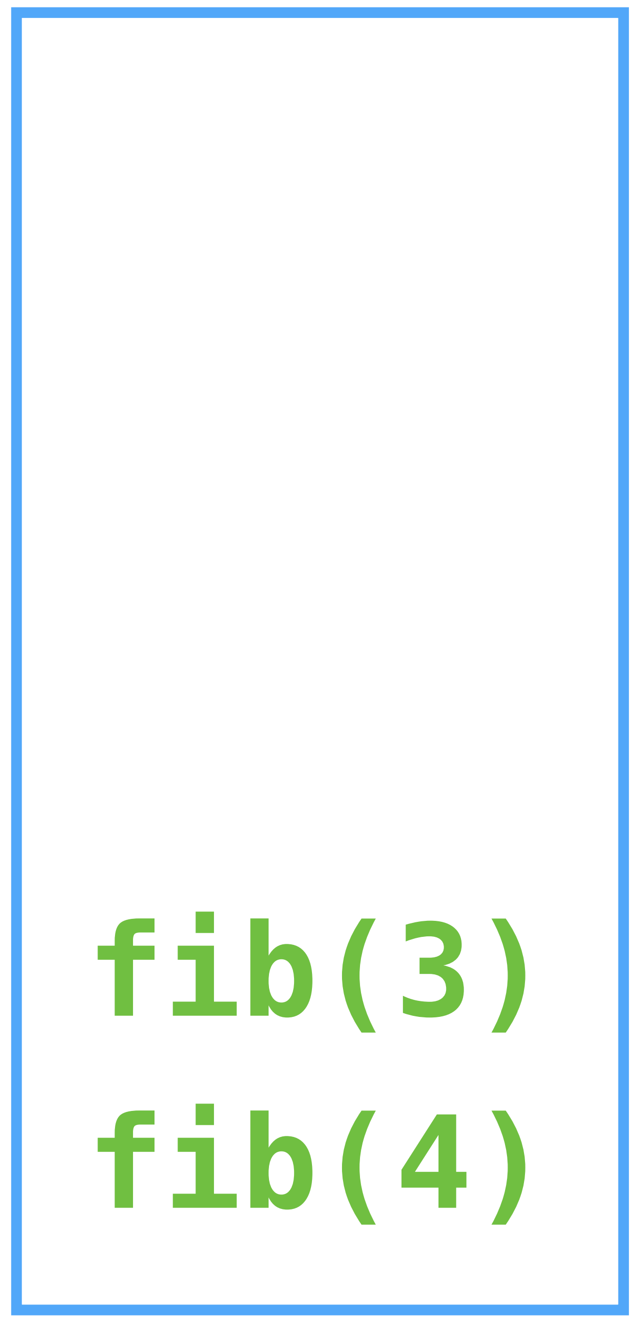
fib(4)

```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

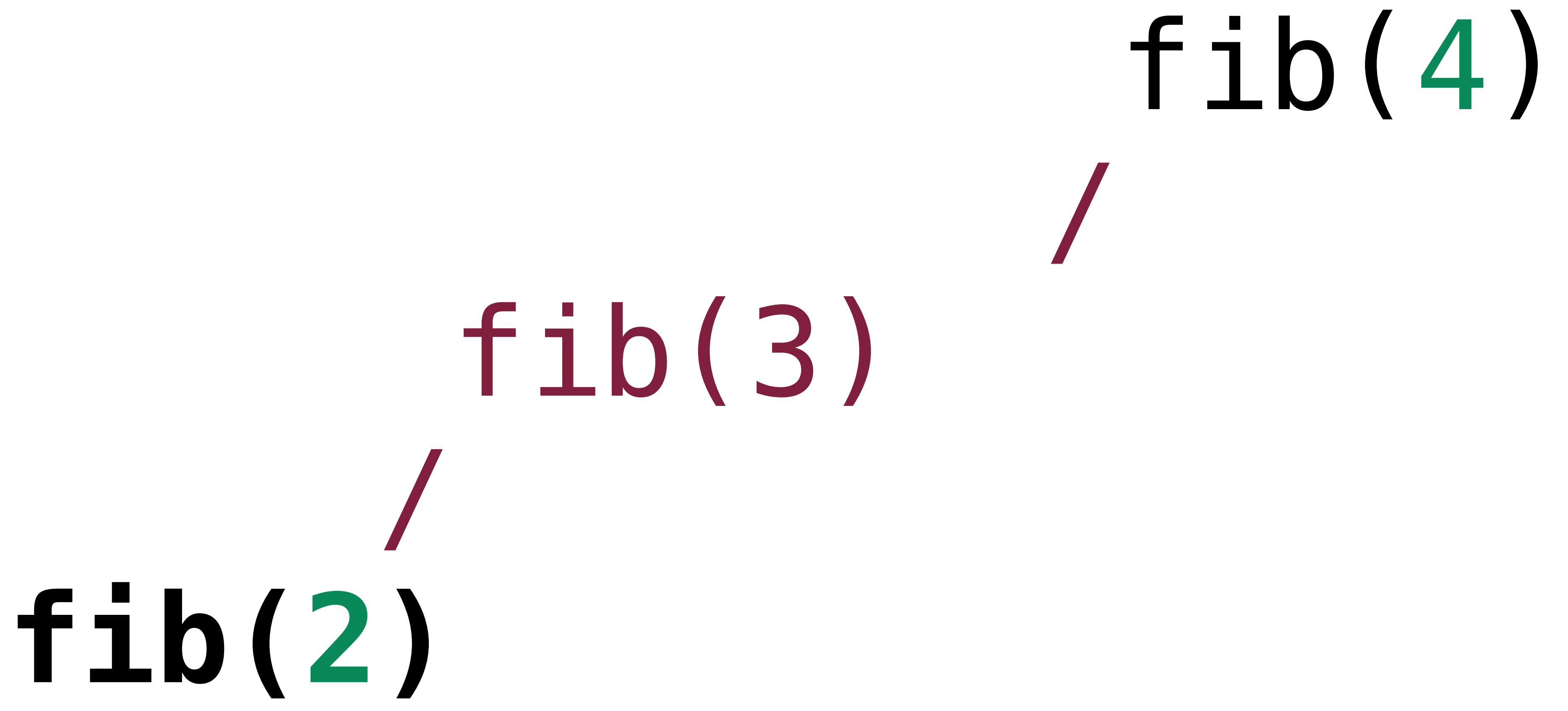


call stack

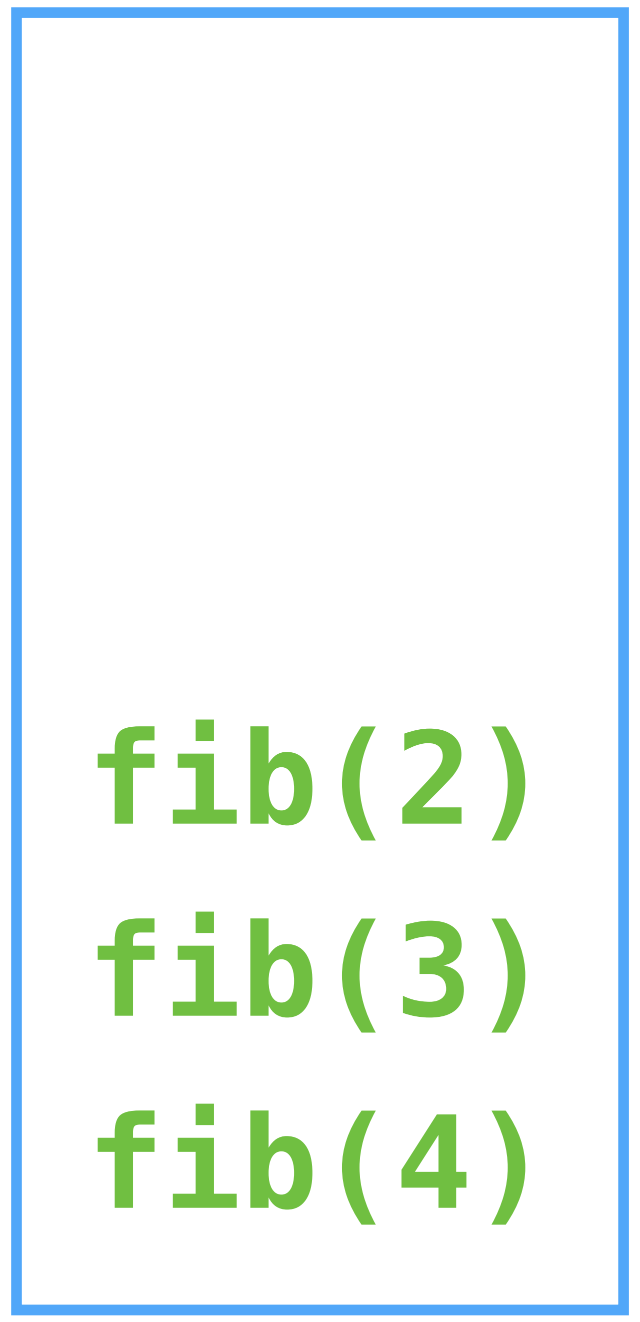


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

}
```

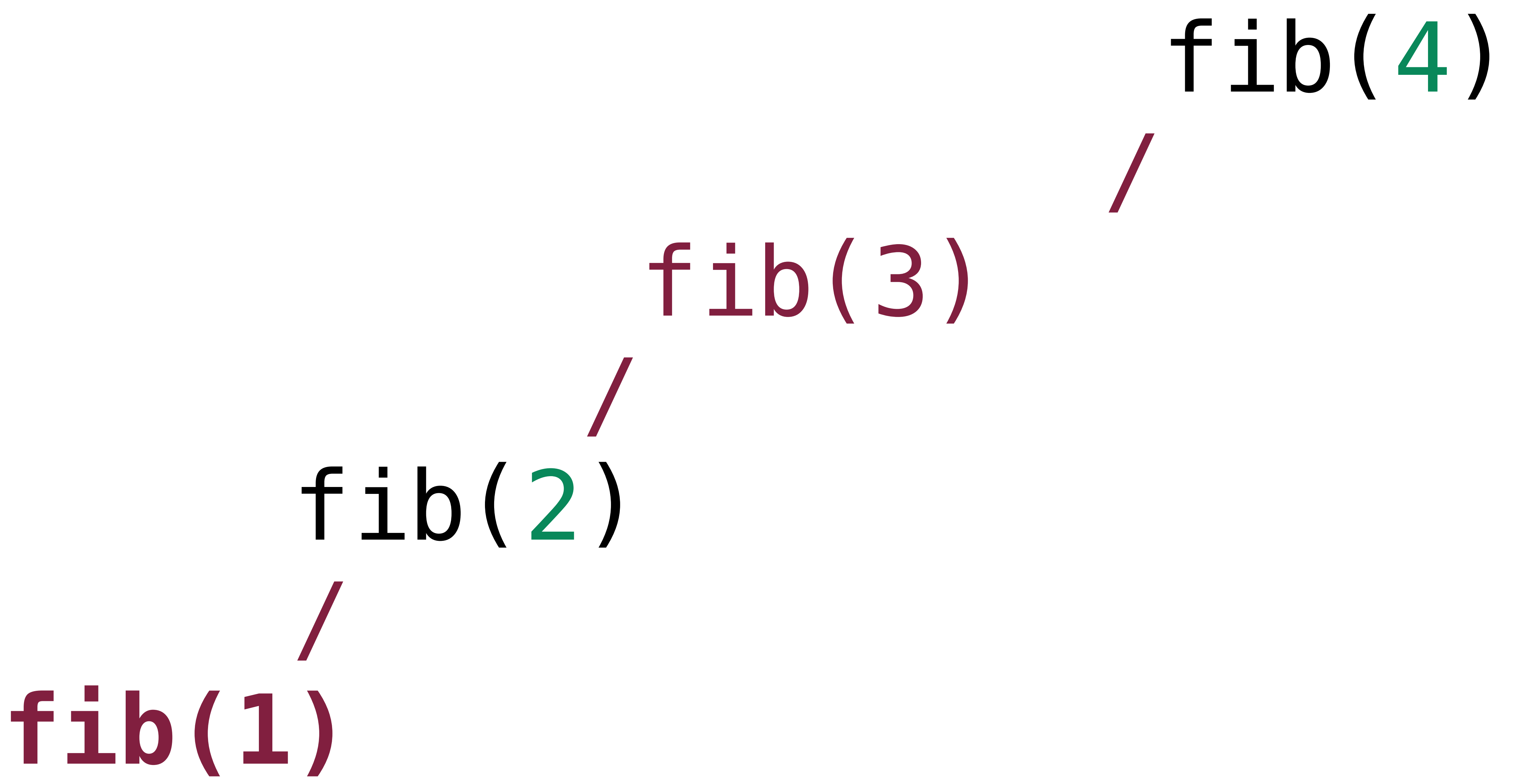


call stack

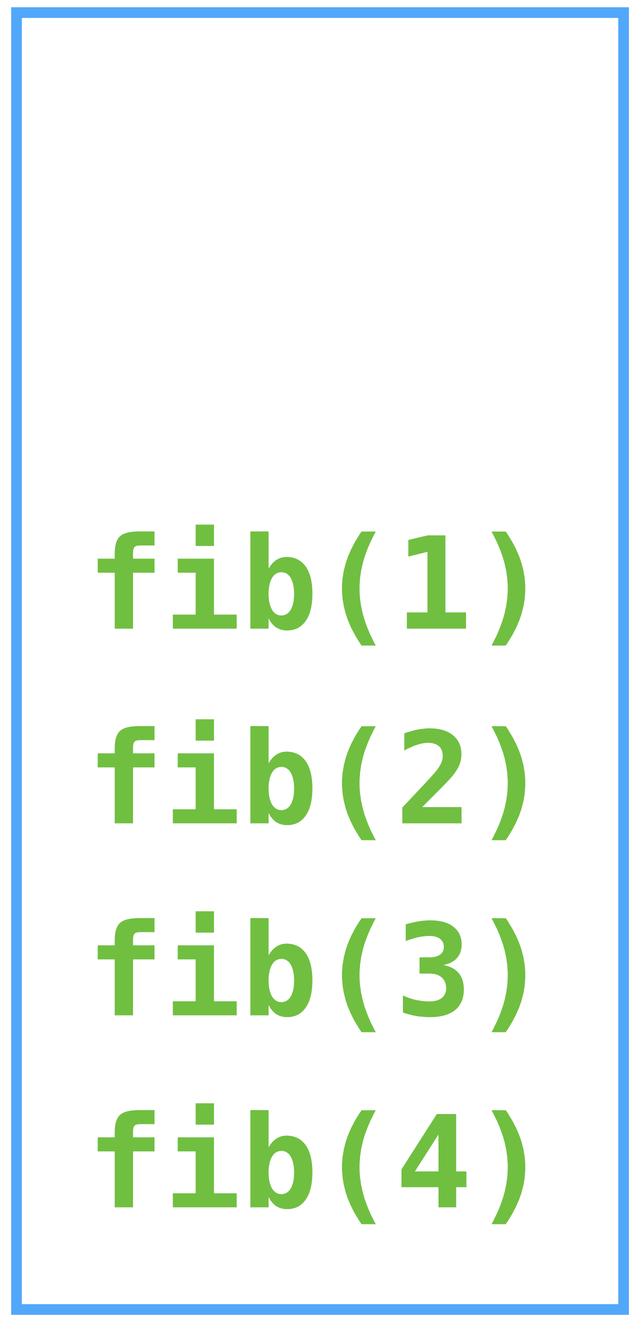


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

  1: 1


}
```

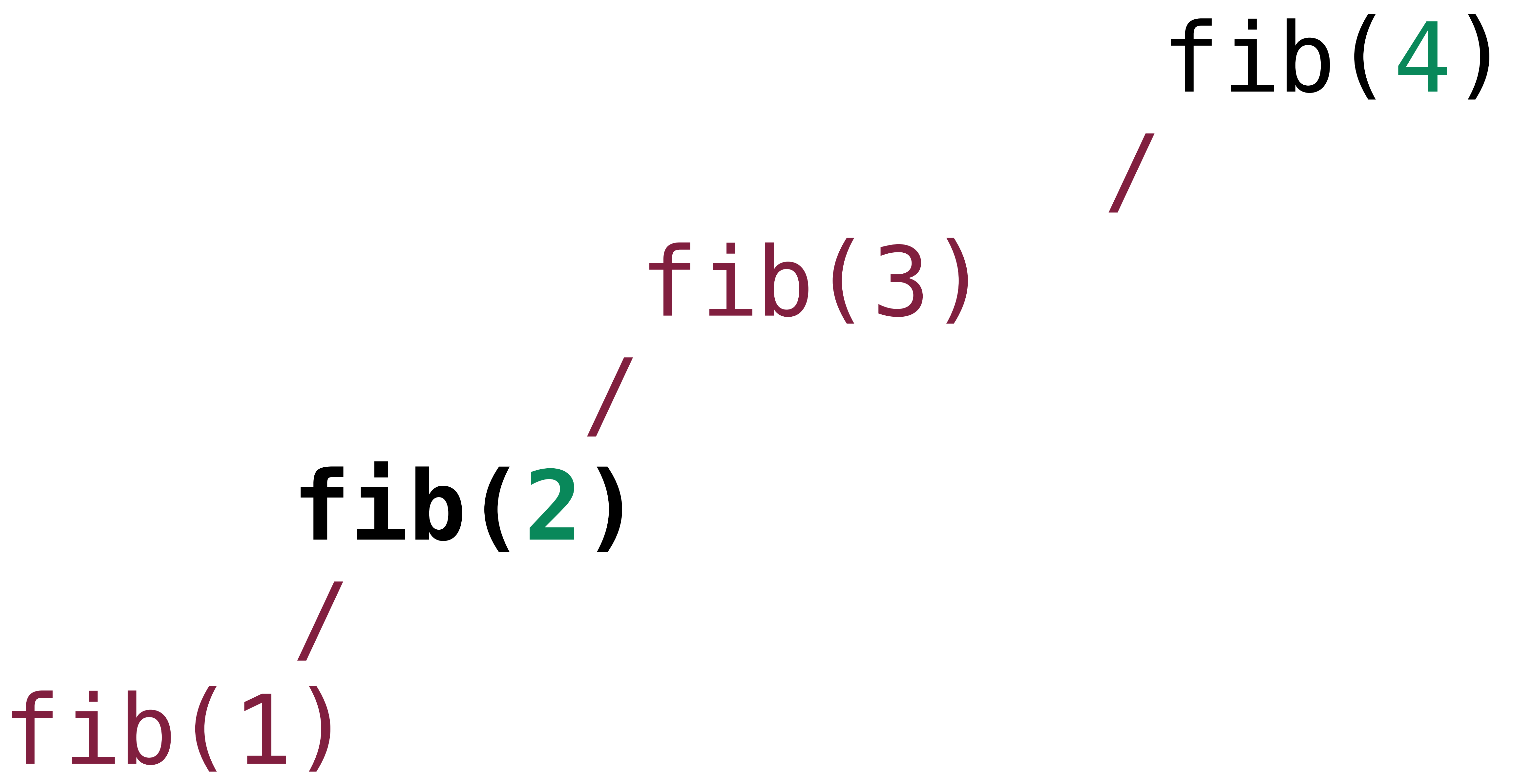


call stack

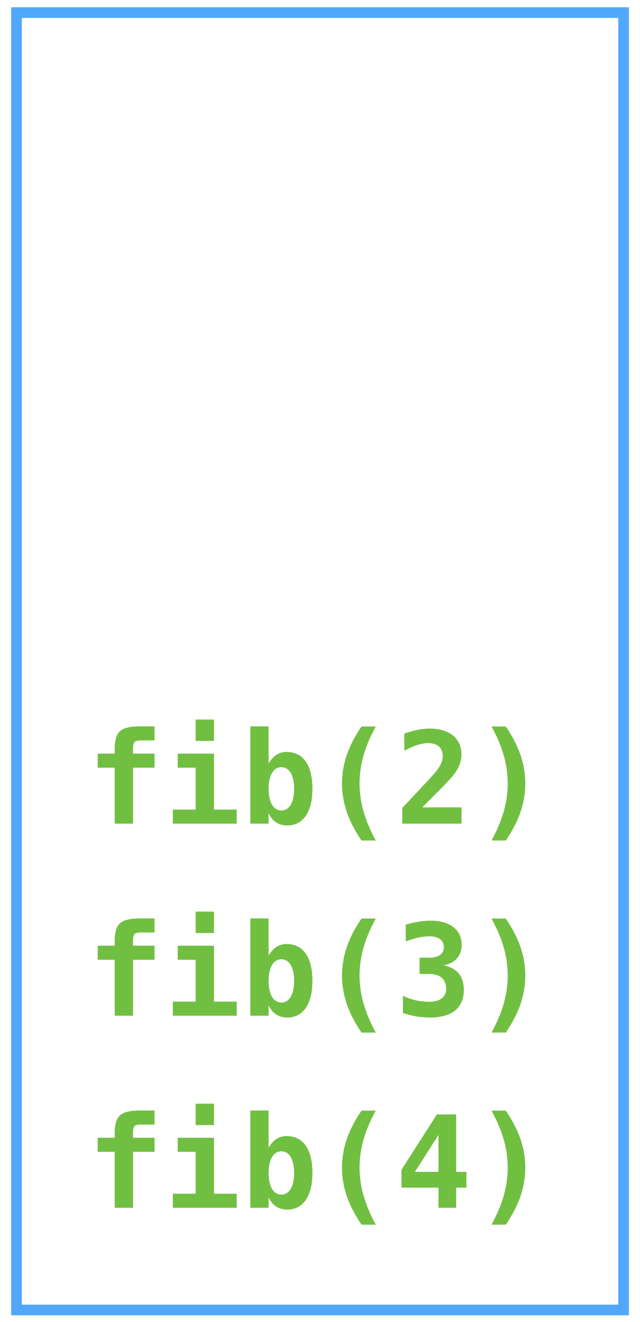


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {

  1: 1

}
```

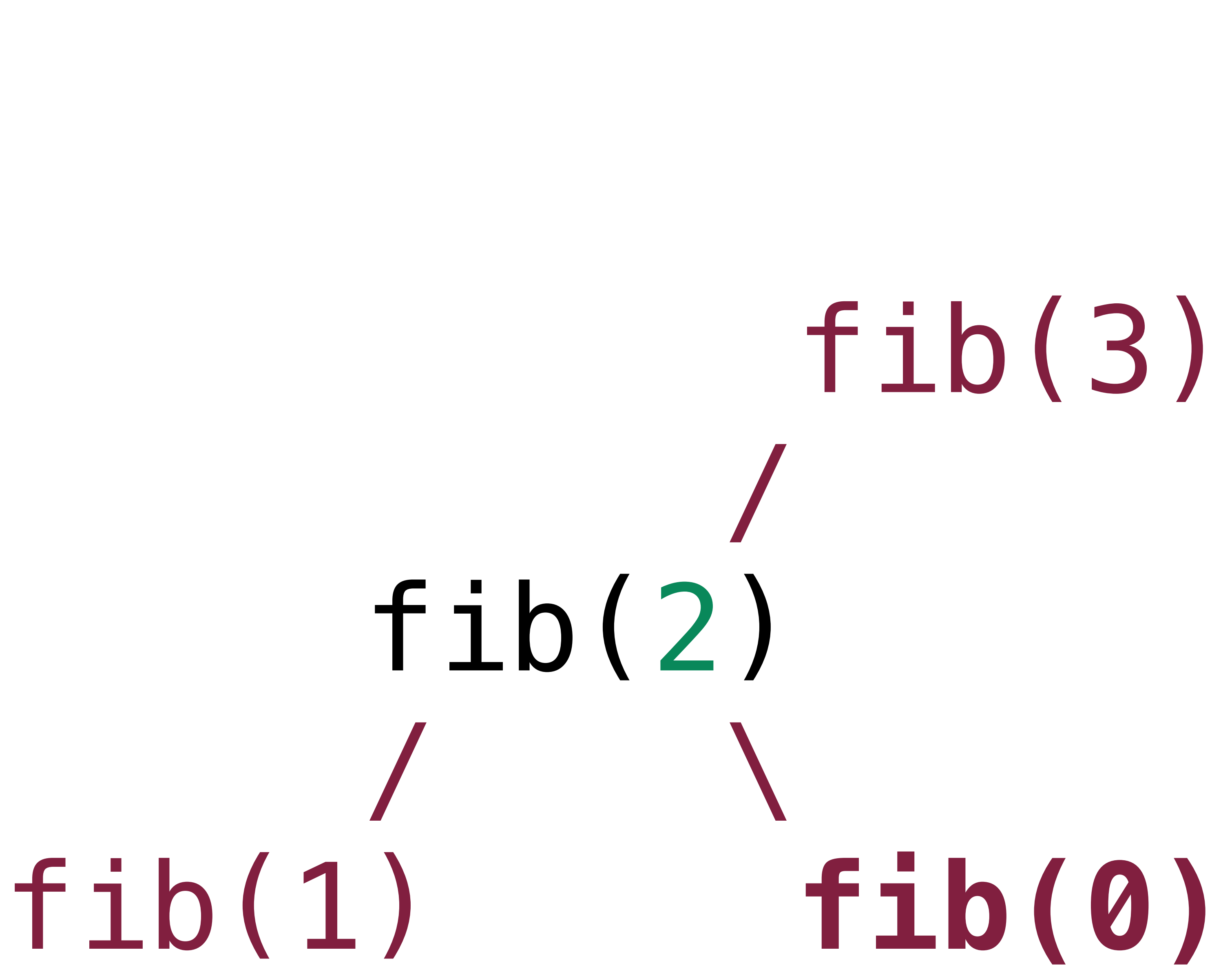


call stack

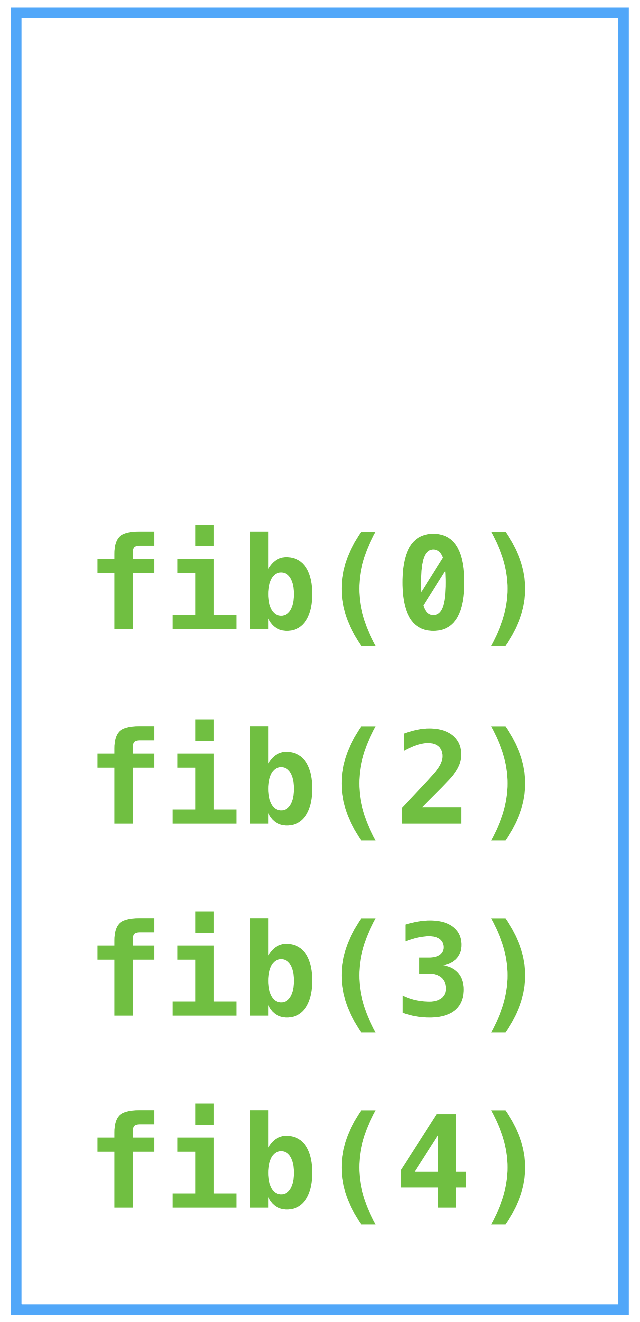


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,

}
```

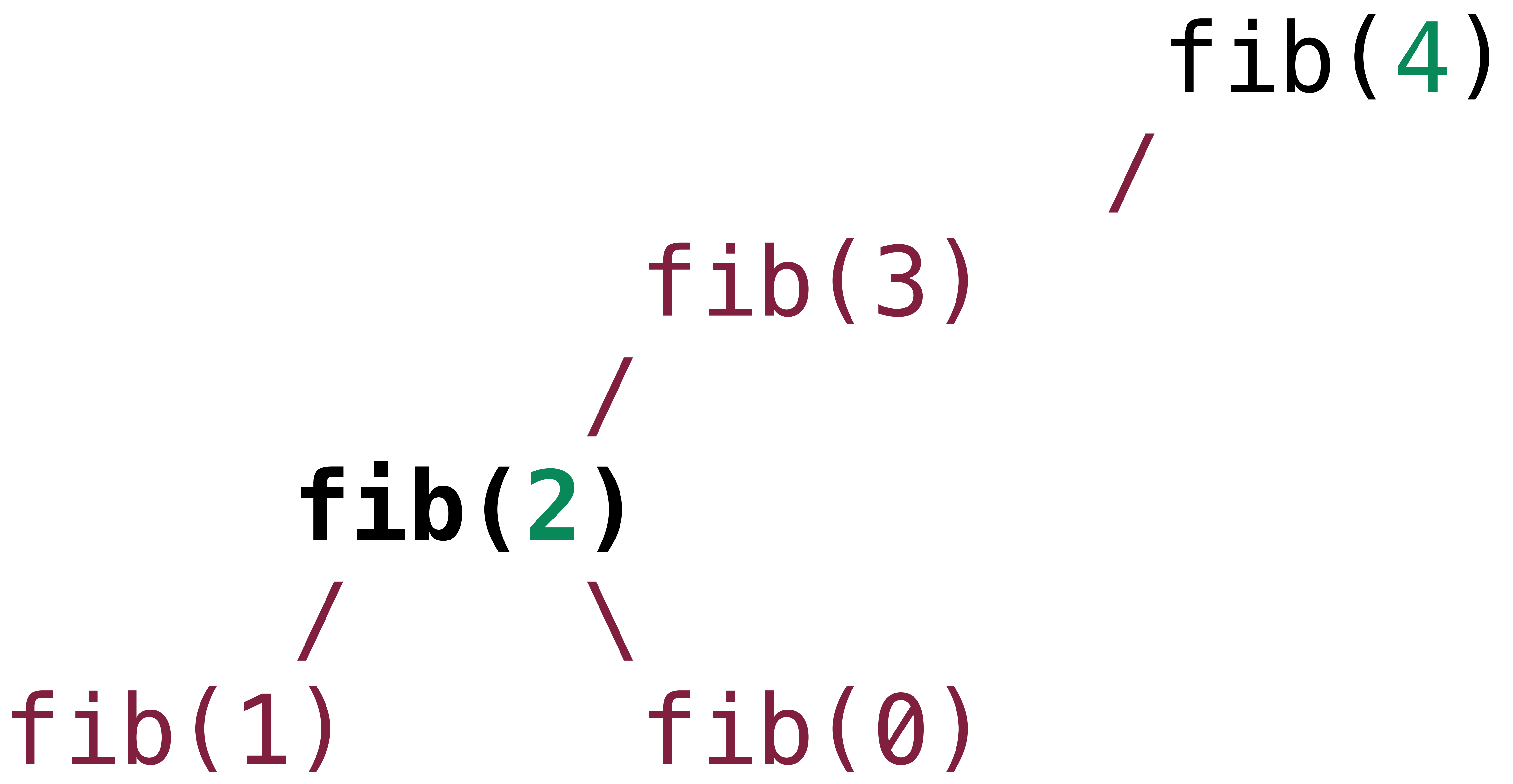


call stack

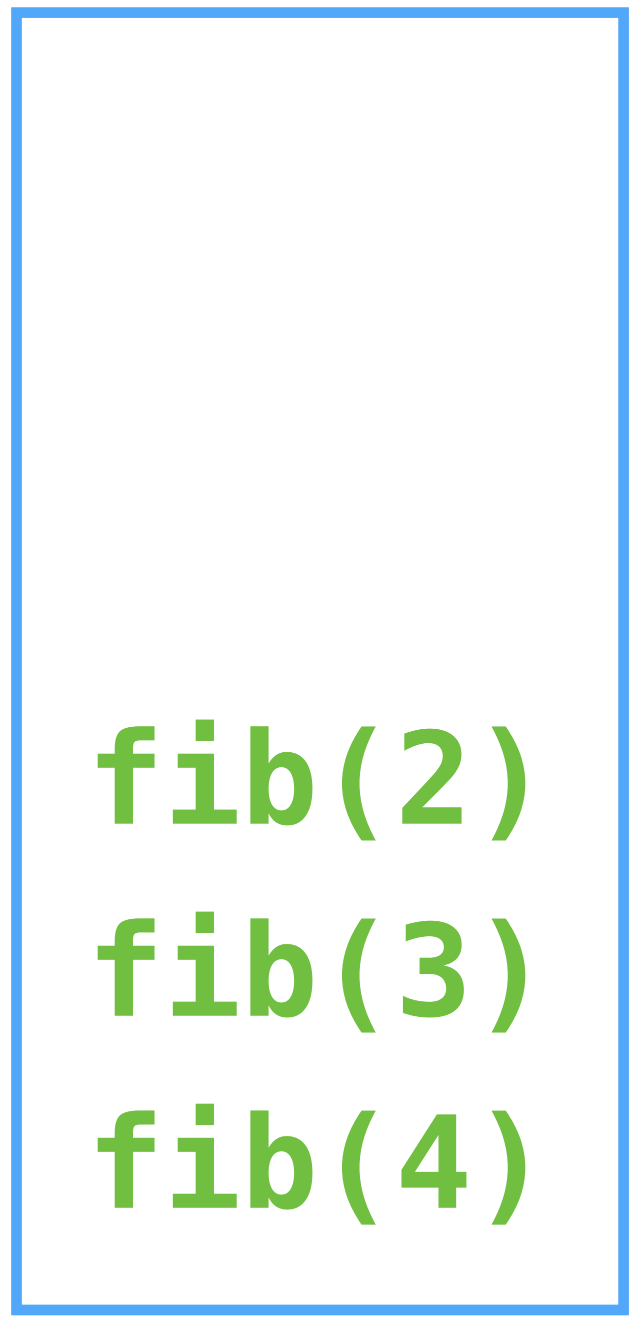


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,

}
```

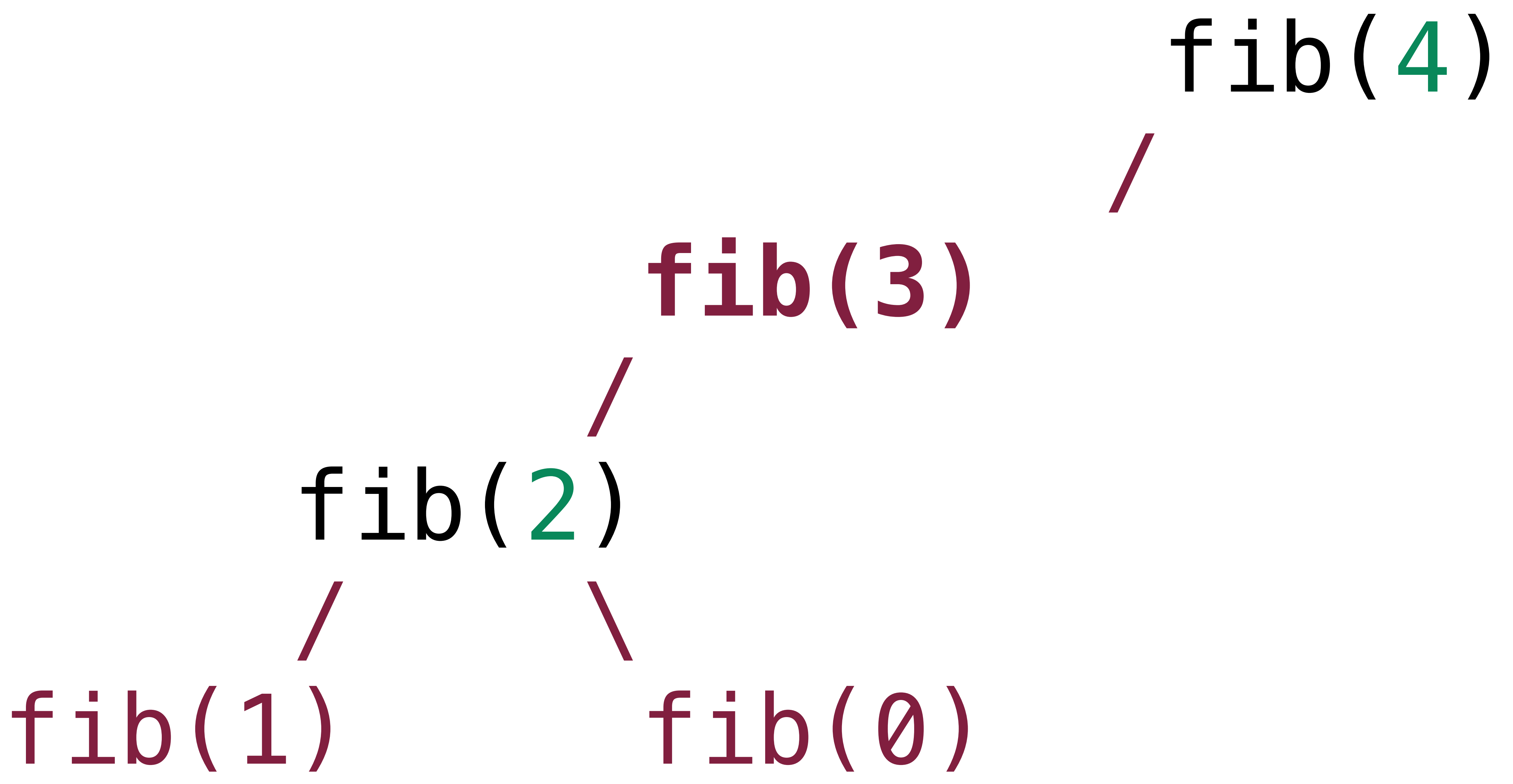


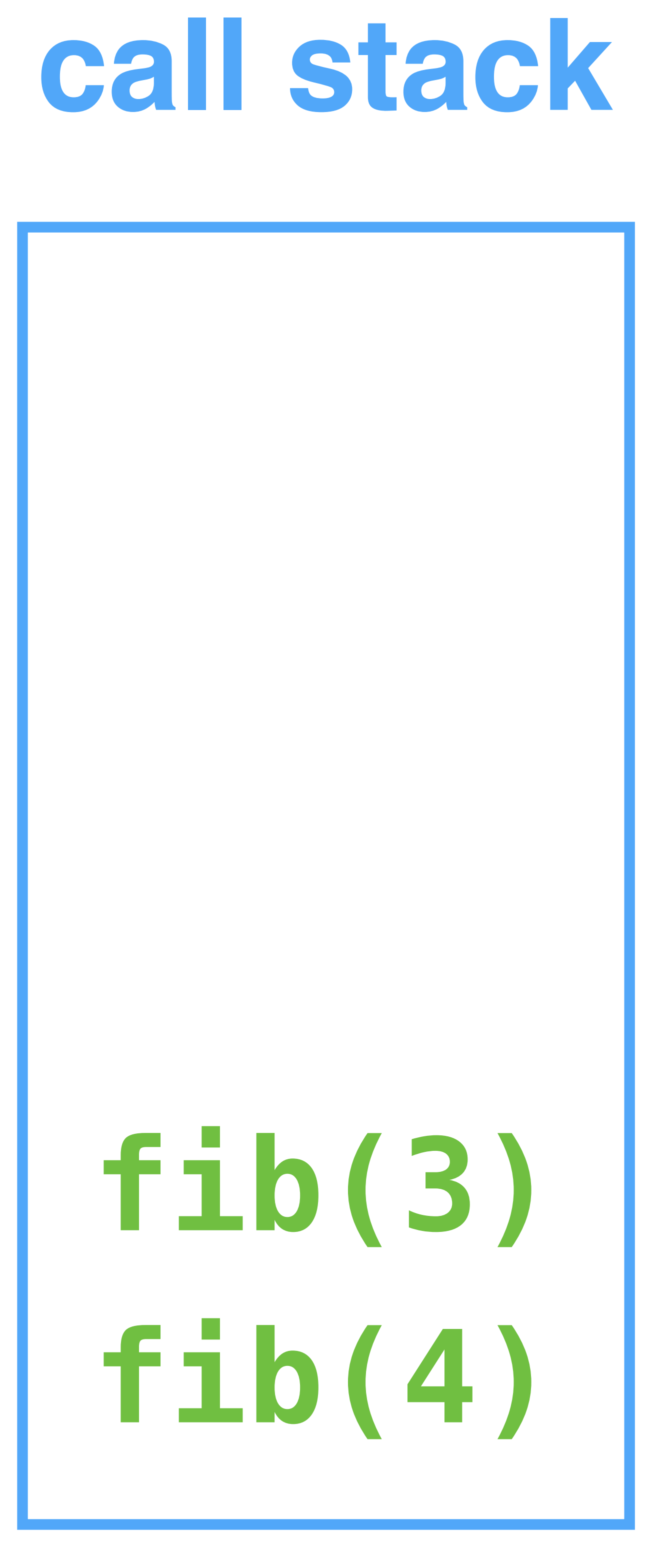
call stack



```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

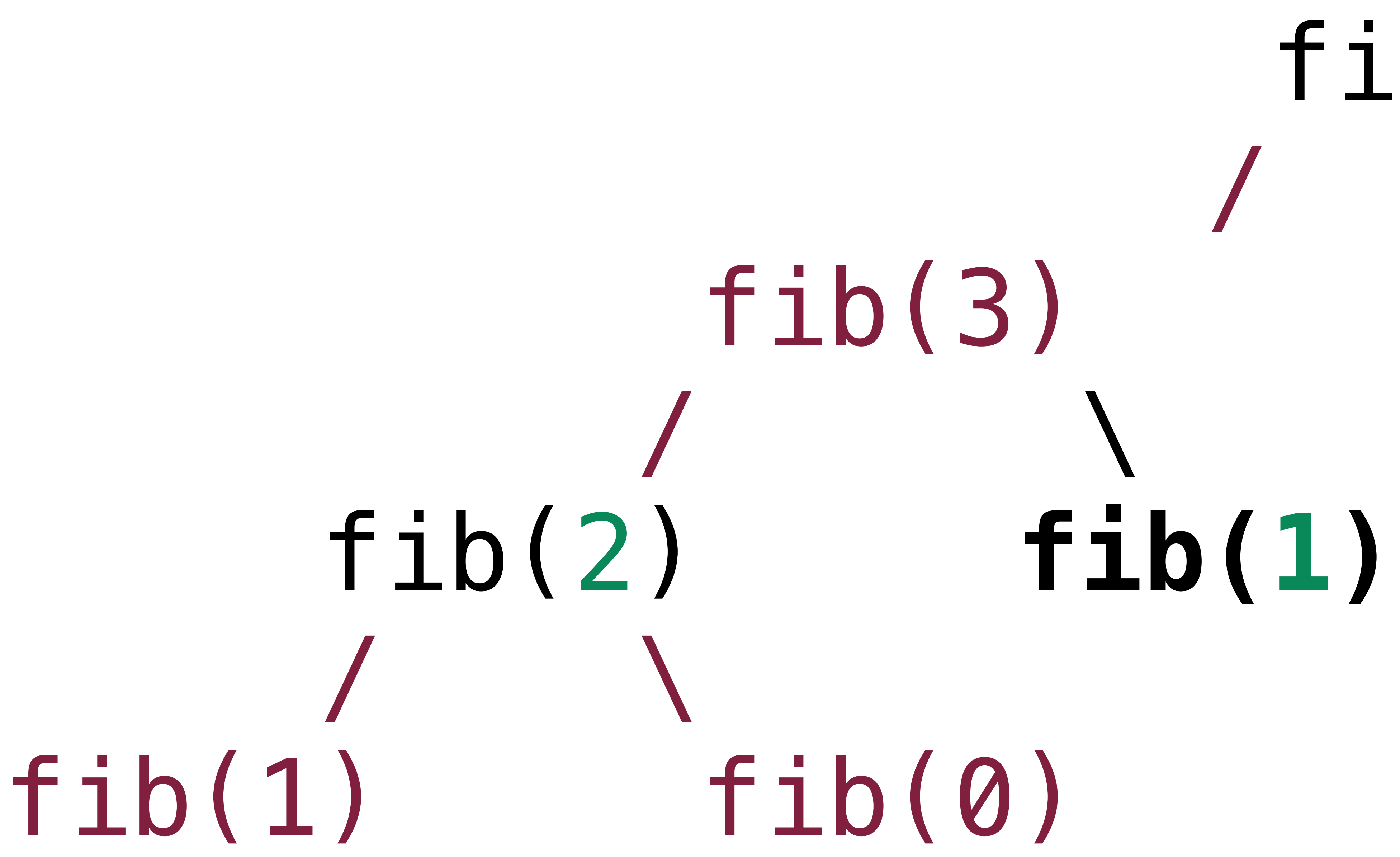
```
memo = {
  0: 0,
  1: 1
  2: 1,

}
```



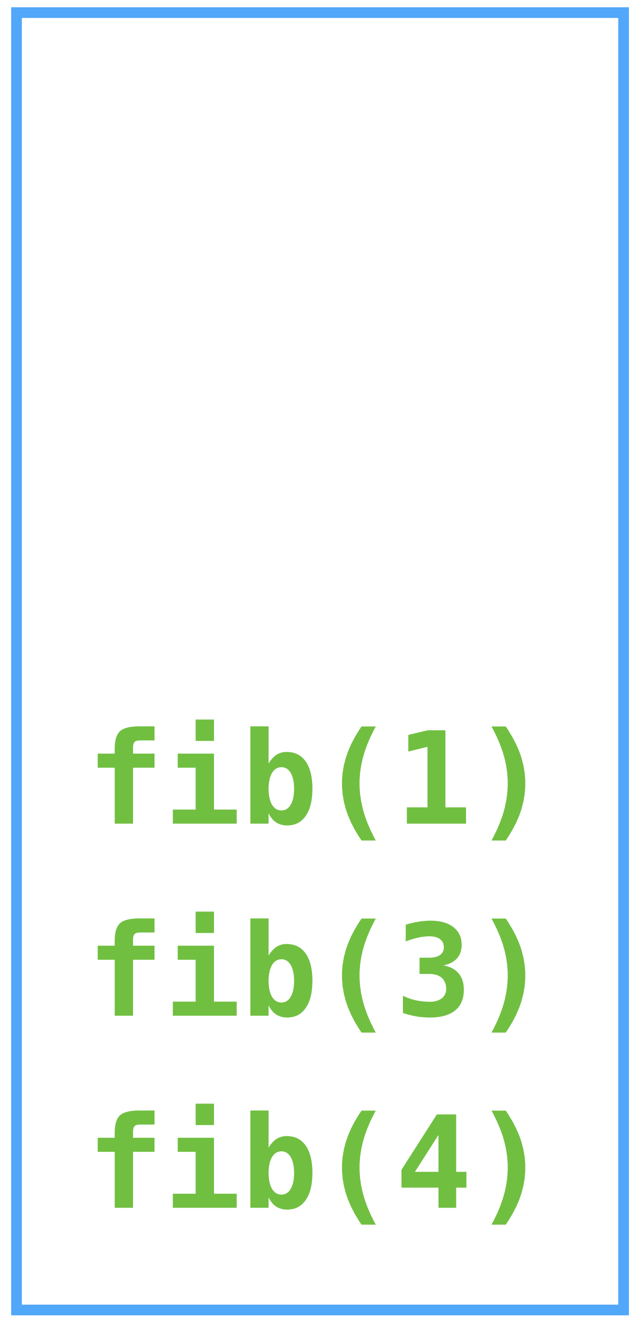


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```

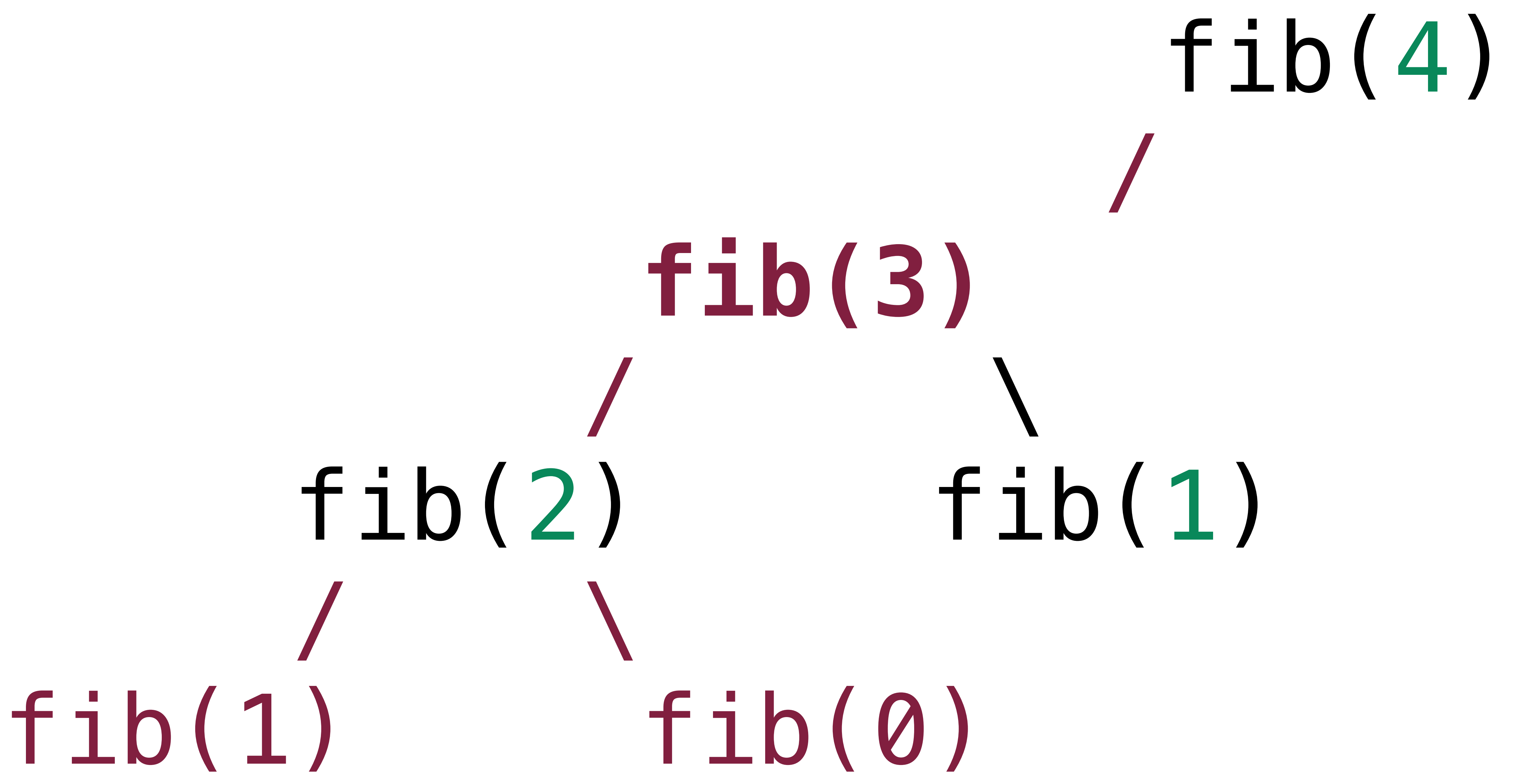


call stack

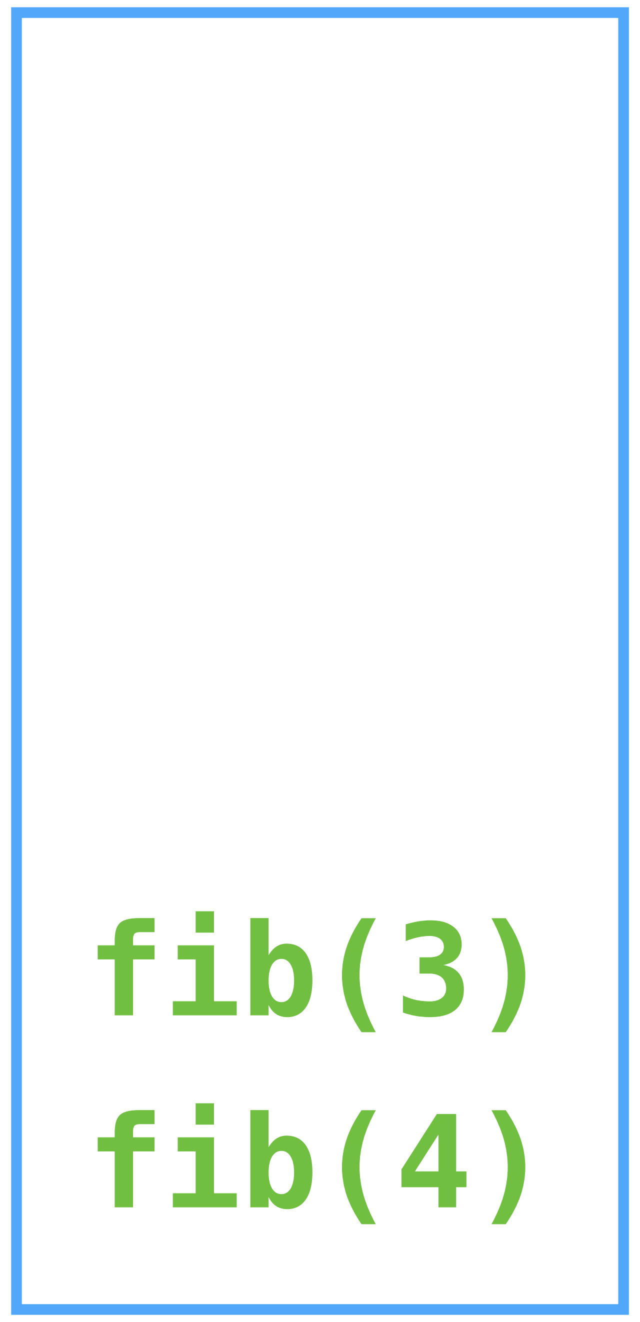


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```

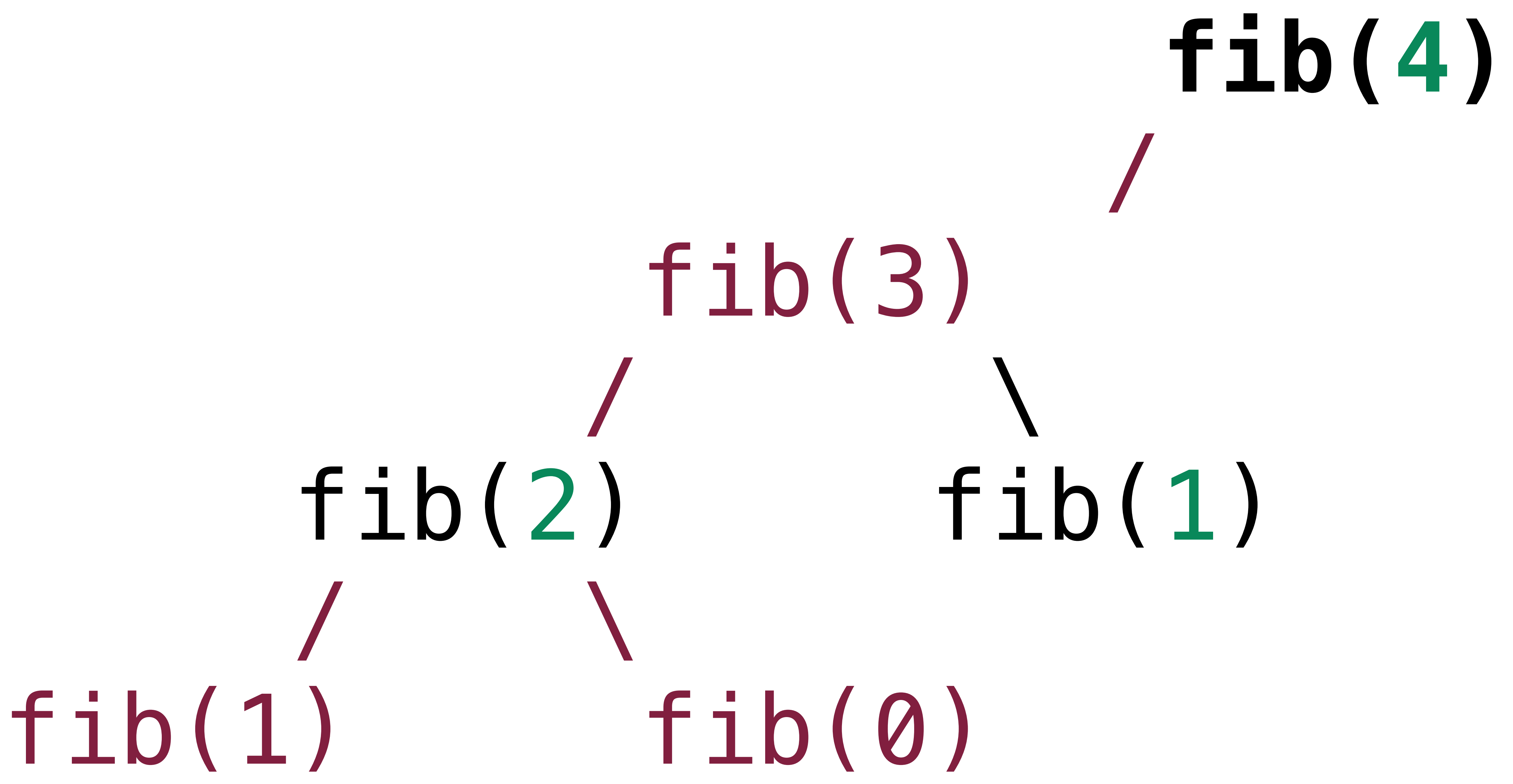


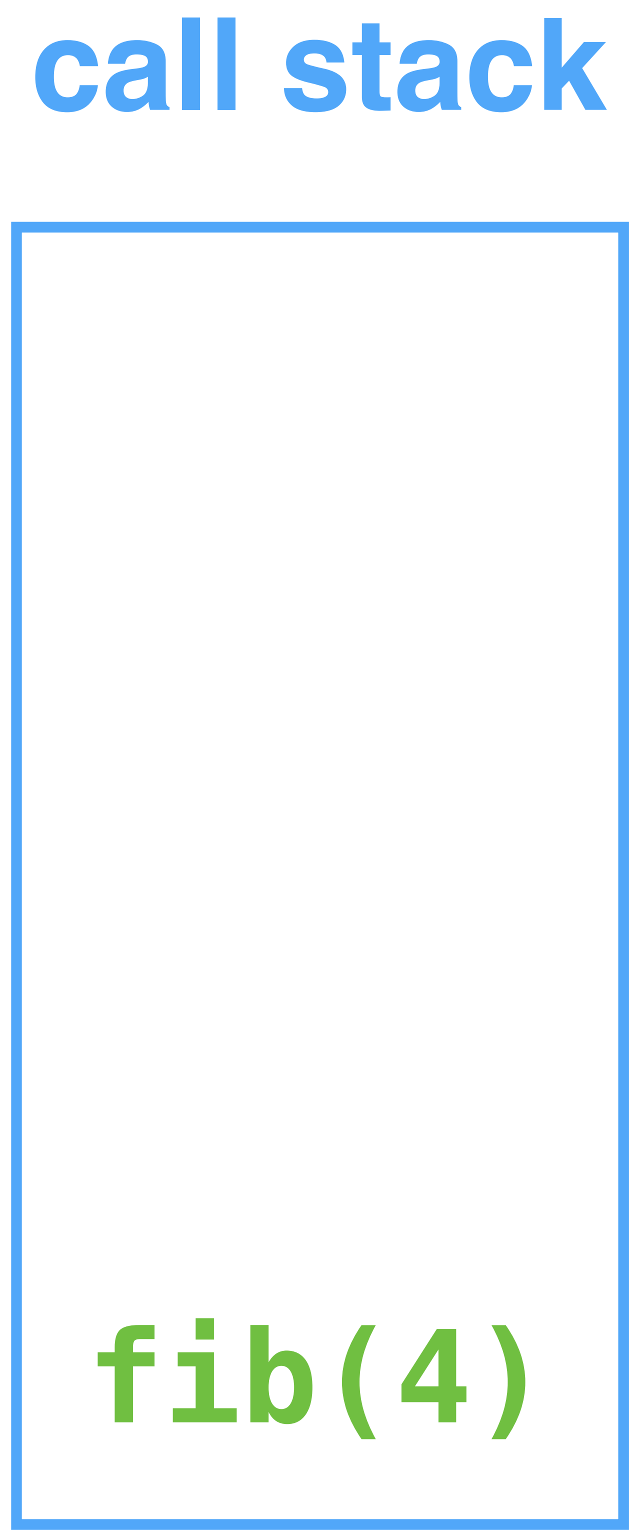
call stack



```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

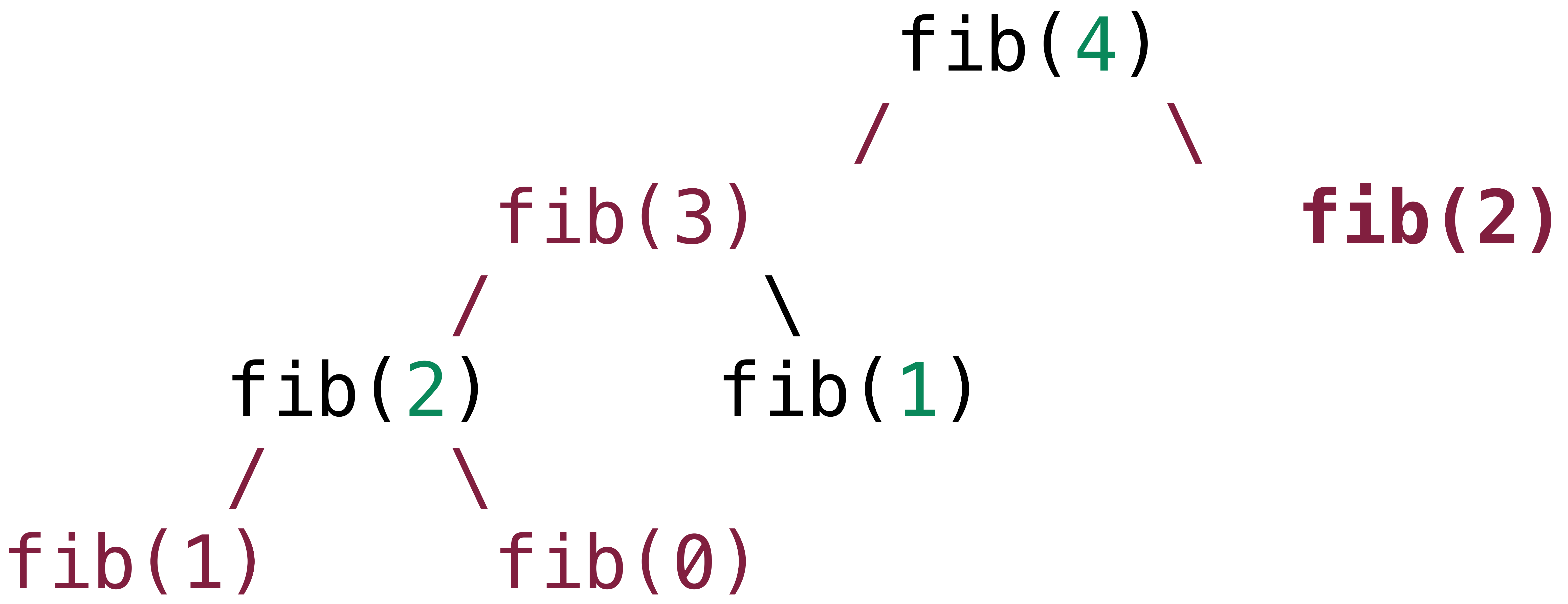
```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```



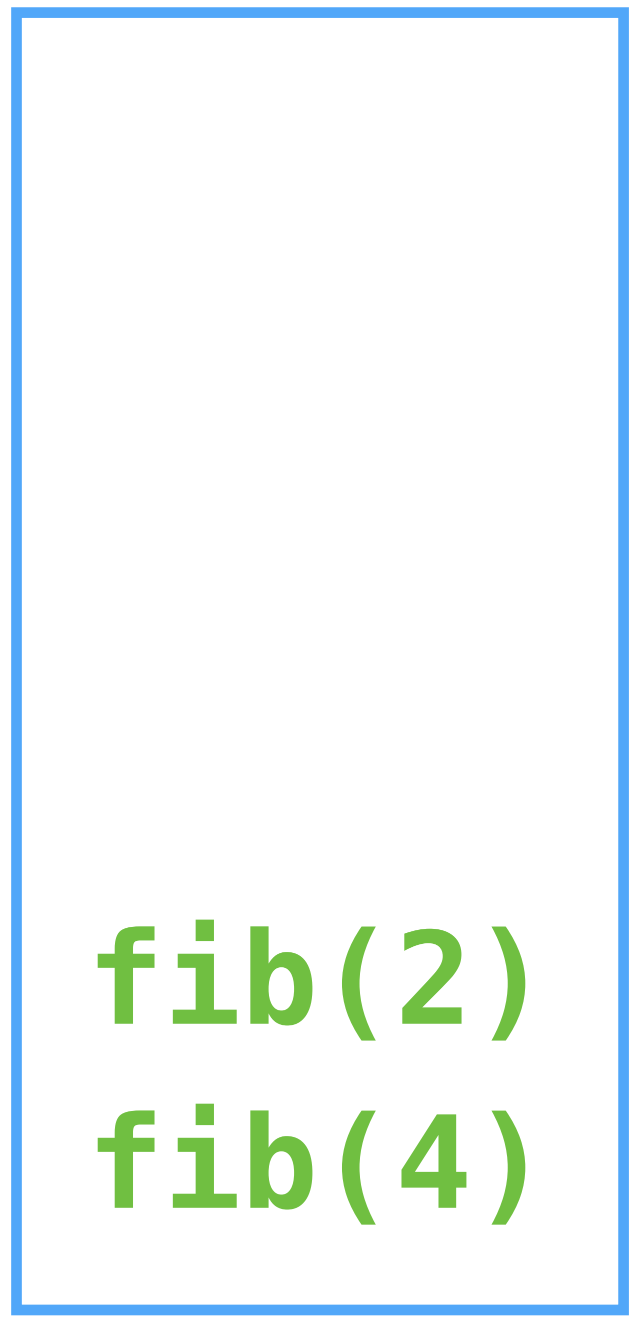


```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```

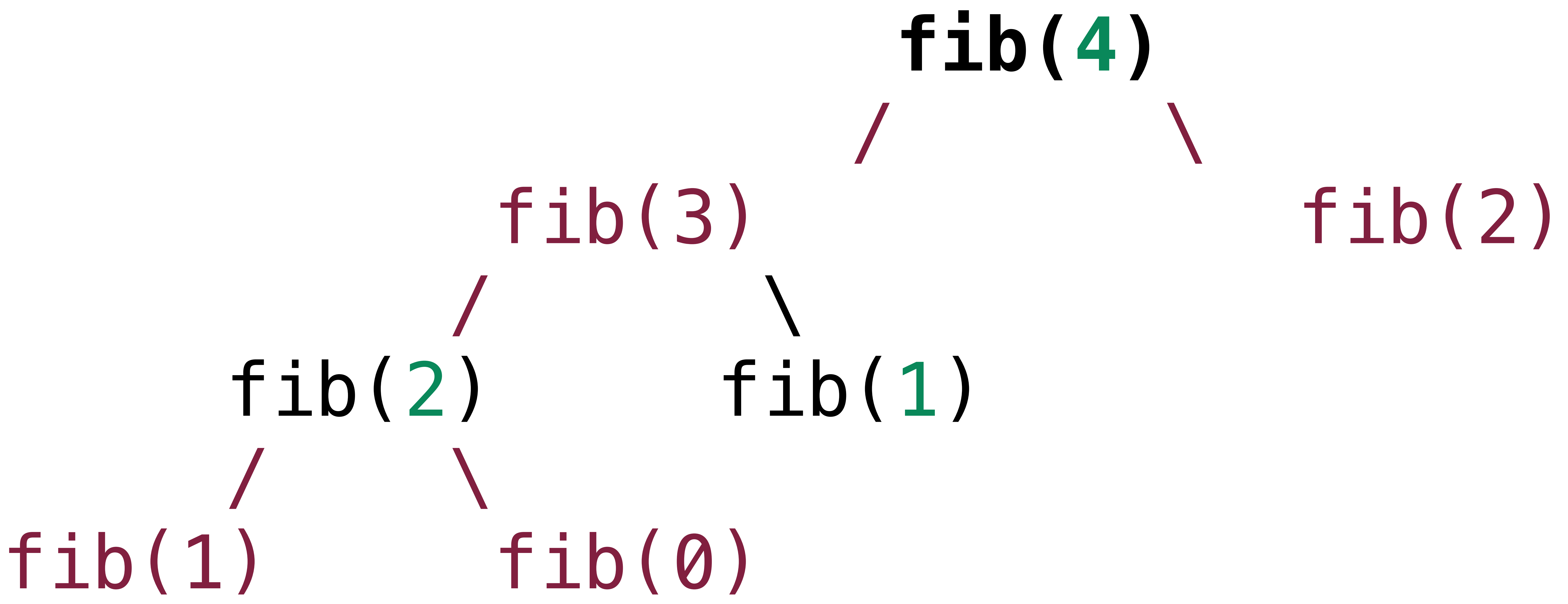


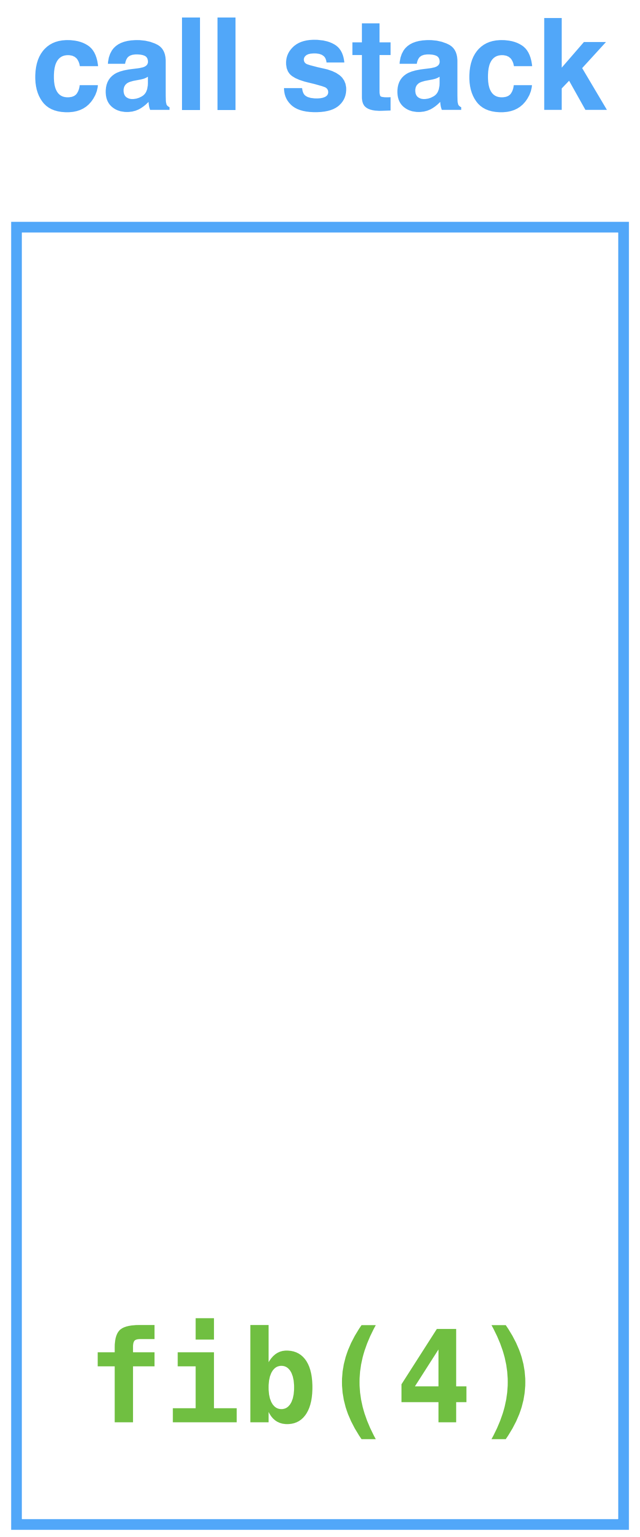
call stack



```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```





```
function fib (n, memo = {}) {
  if (n === 1 || n === 0) return n;
  else if (memo[n]) return memo[n];
  else memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
  return memo[n];
}
```

```
memo = {
  0: 0,
  1: 1
  2: 1,
  3: 2
}
```

```
                        fib(4)
                       /      \
                fib(3)             fib(2)
               /      \
          fib(2)      fib(1)
         /      \
    fib(1)      fib(0)
```

call stack

ASK QUESTIONS

- **Should I worry about optimization?**
- **What should I optimize for?**
  - What environment are we in? What are the constraints?
- **Is the array sorted?**
- **Will this only run one time or many times?**
  - Optimizing a one-off solution is different than optimizing the average for repeated executions
  - *Any* pure function can be memoized

OTHER TIPS:

- **Pay very careful attention to the details in the problem description**
  - Most problems won't contain irrelevant info (though it's not impossible)
  - Try to take advantage of EVERY piece of info given to you
- **Consider the best conceivable runtime**
- **See if you can do some pre-computation up front to save time later**
  - Boyer-Moore string search algorithm