

Tail call optimization

tail call optimization for writing elegant recursive solutions without the performance tax of ES5

A tail call is a subroutine call performed as the final action of a procedure.
That is,

```
return myFunction()
```

It is important to understand that ES6 does not introduce new syntax for tail call optimization. It is just a different structure of code to make sure that it is efficient.

Let's calculate the Fibonacci using recursion:

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



Let's view the function calls in the form of a tree:

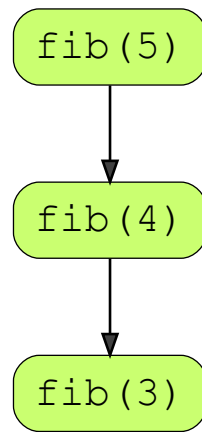
`fib(5)`

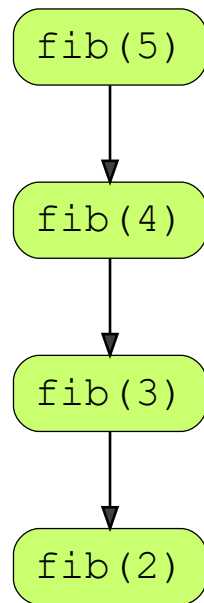
1 of 46

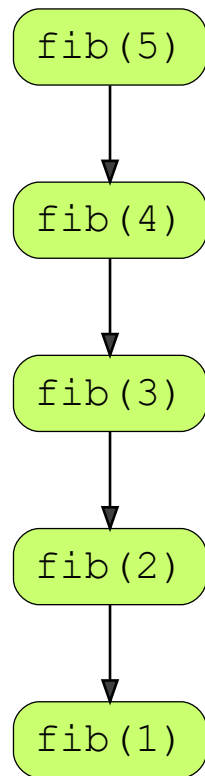
`fib(5)`

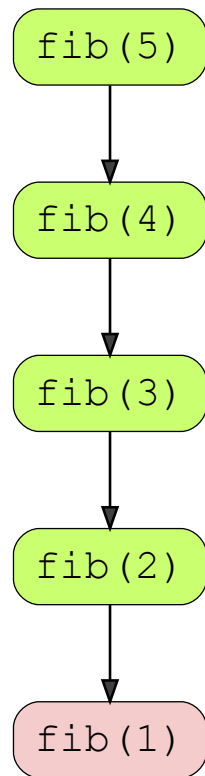


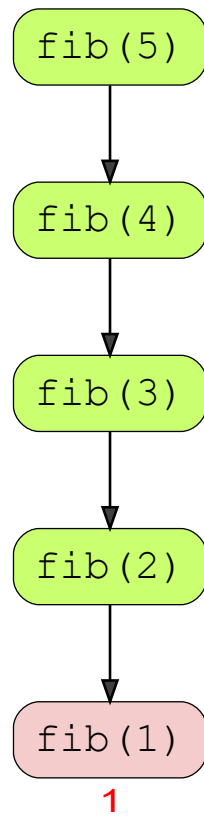
`fib(4)`

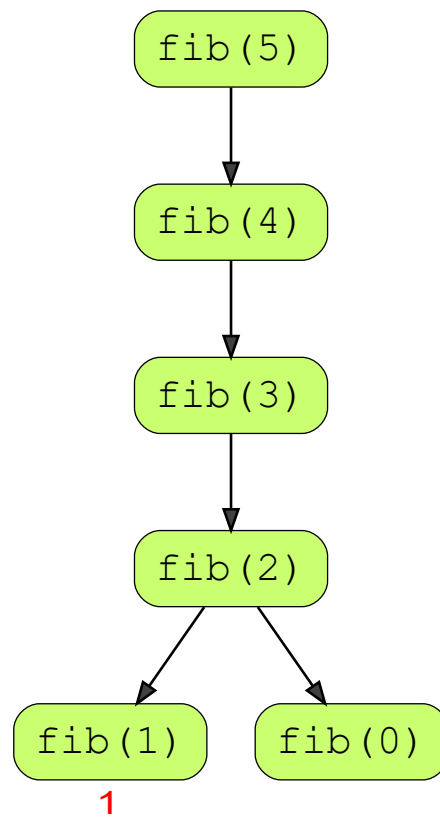


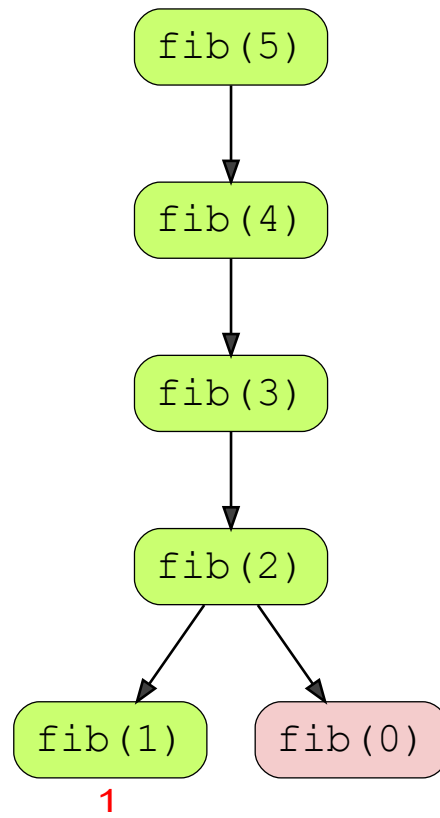


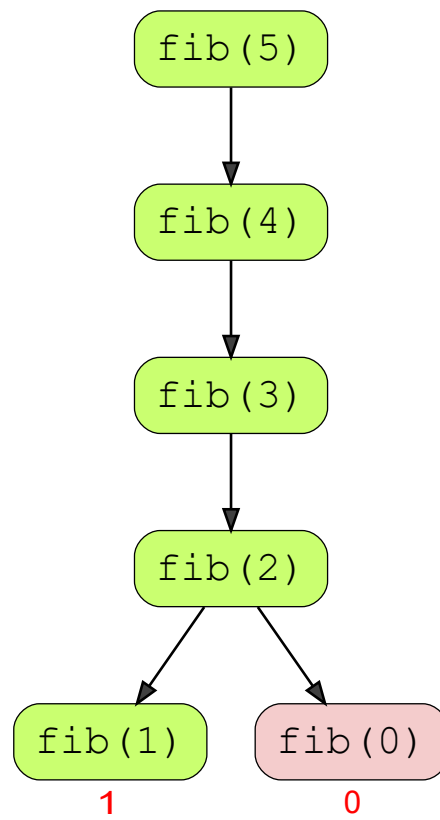


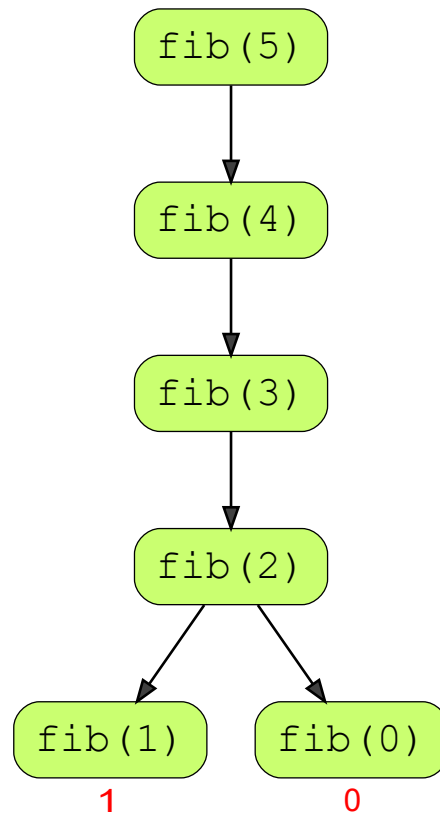


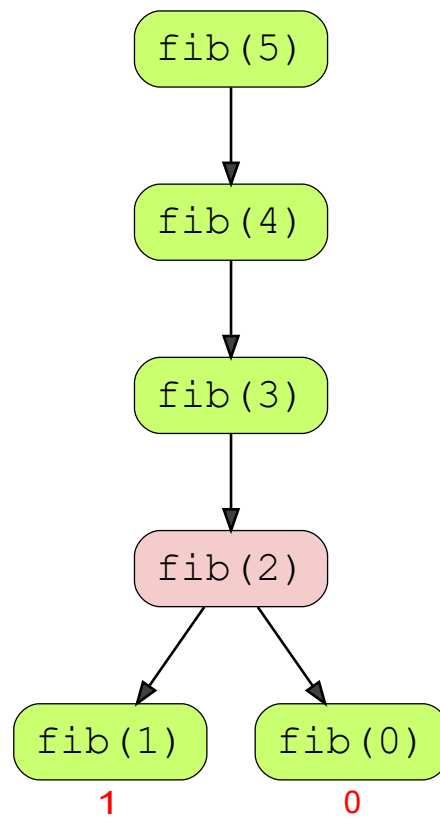


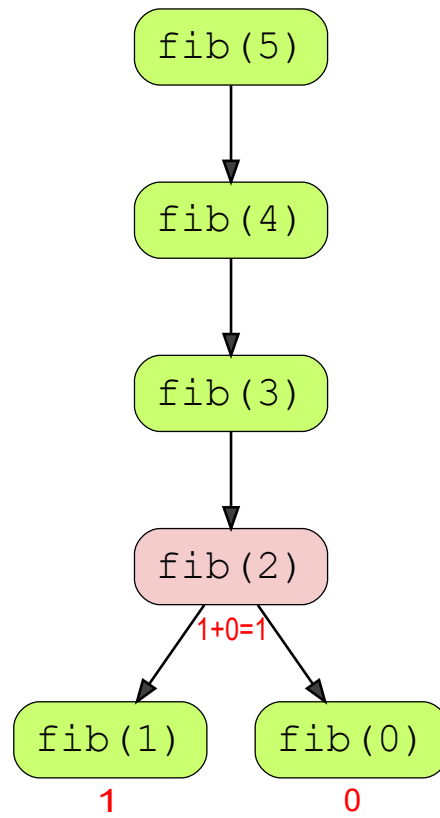


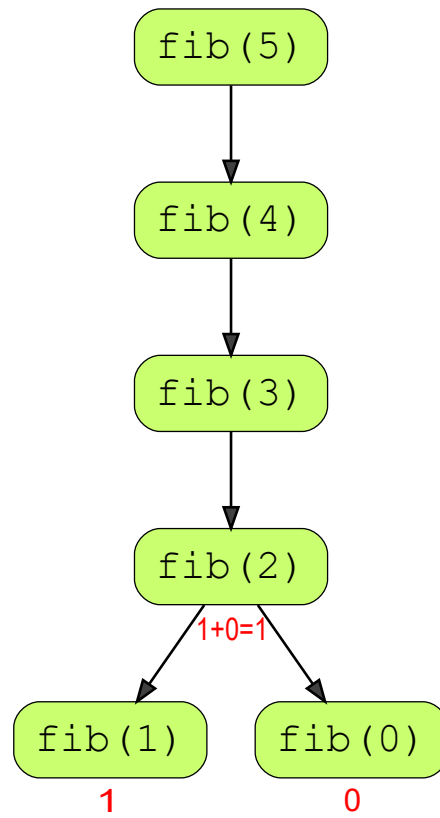


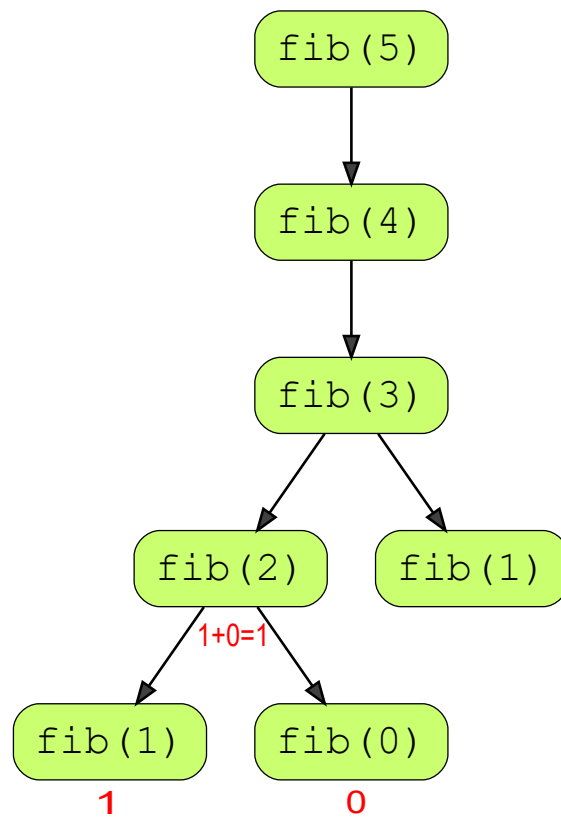


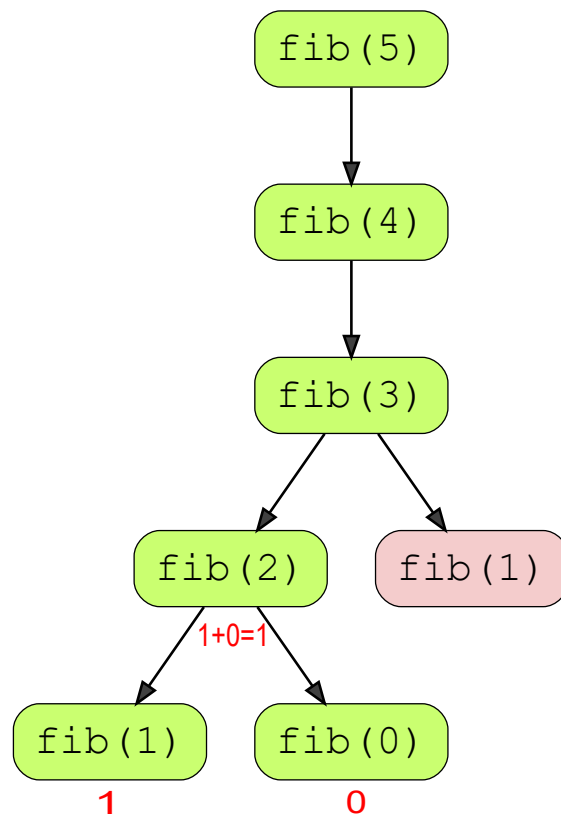


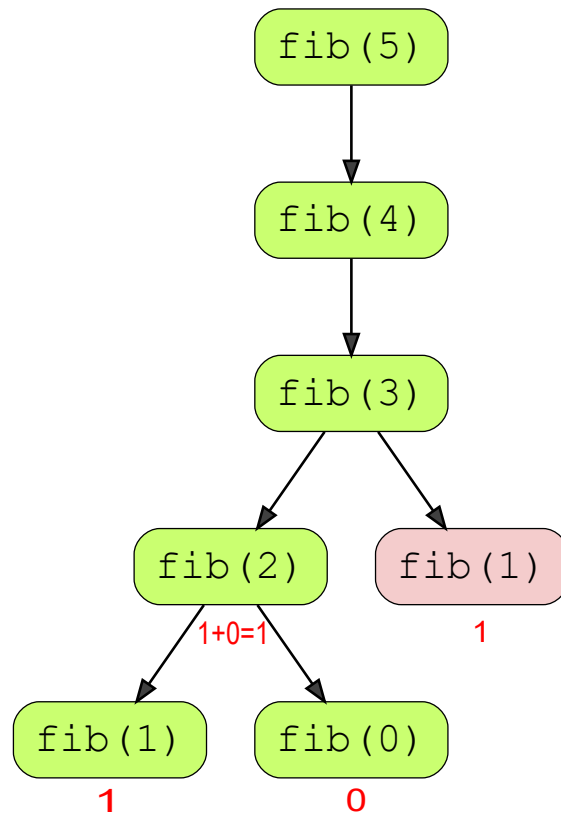


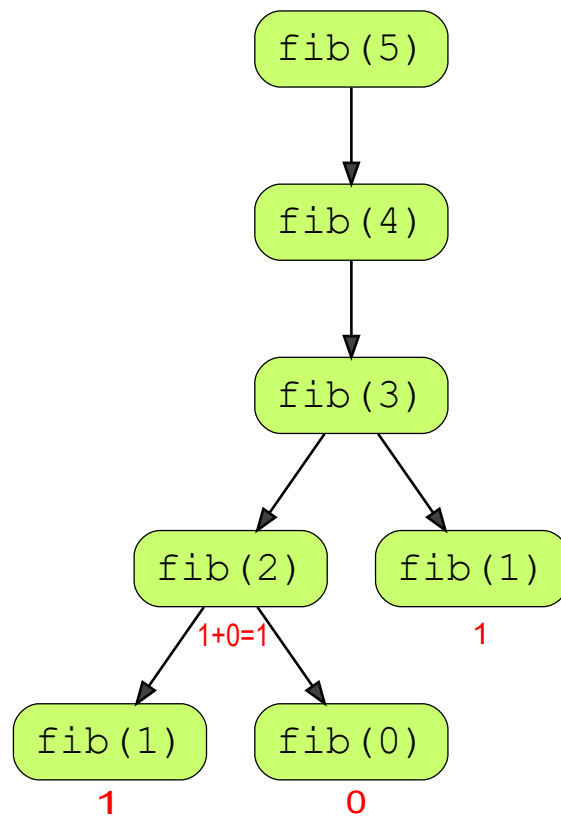


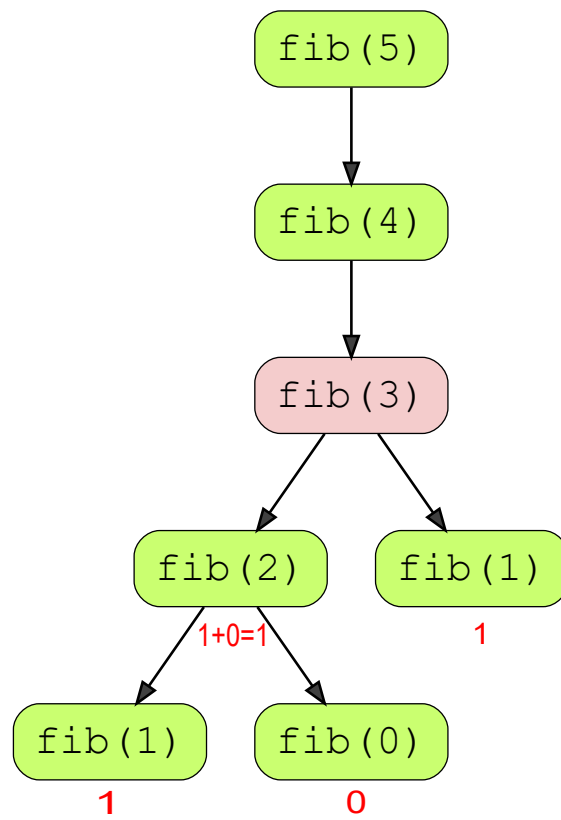


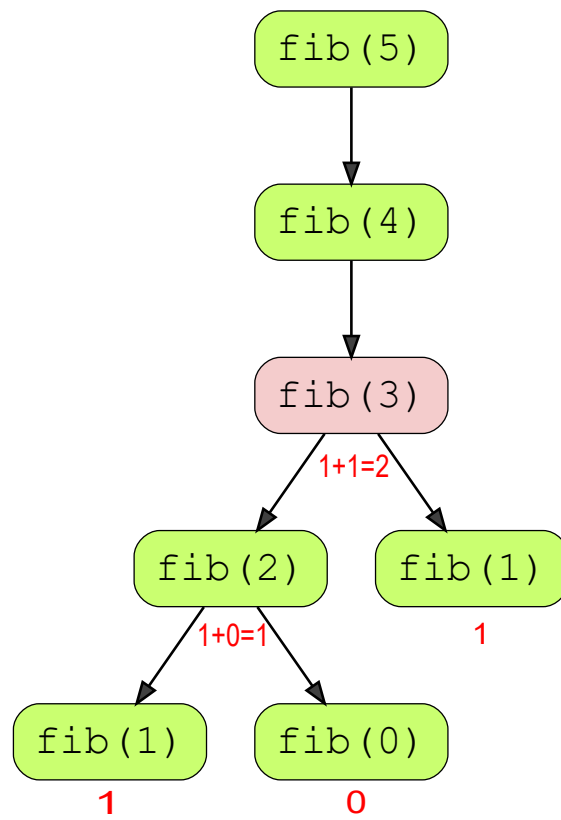


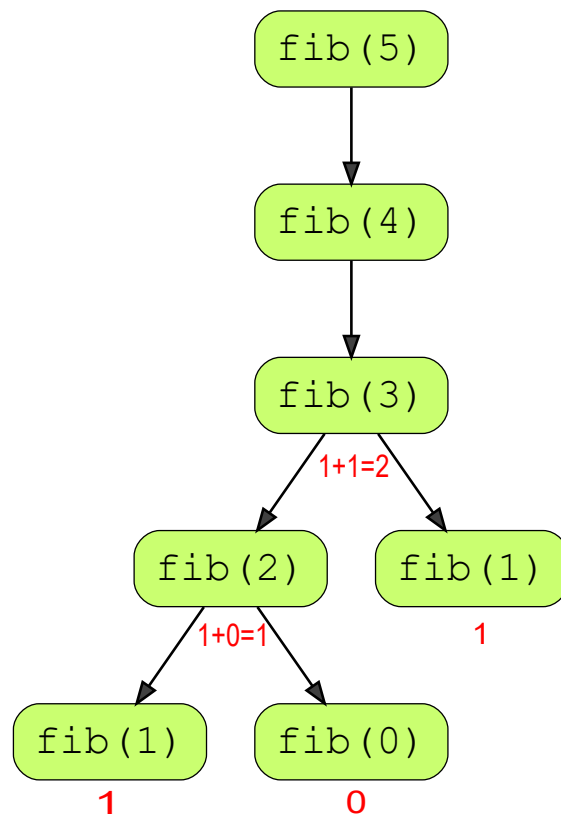


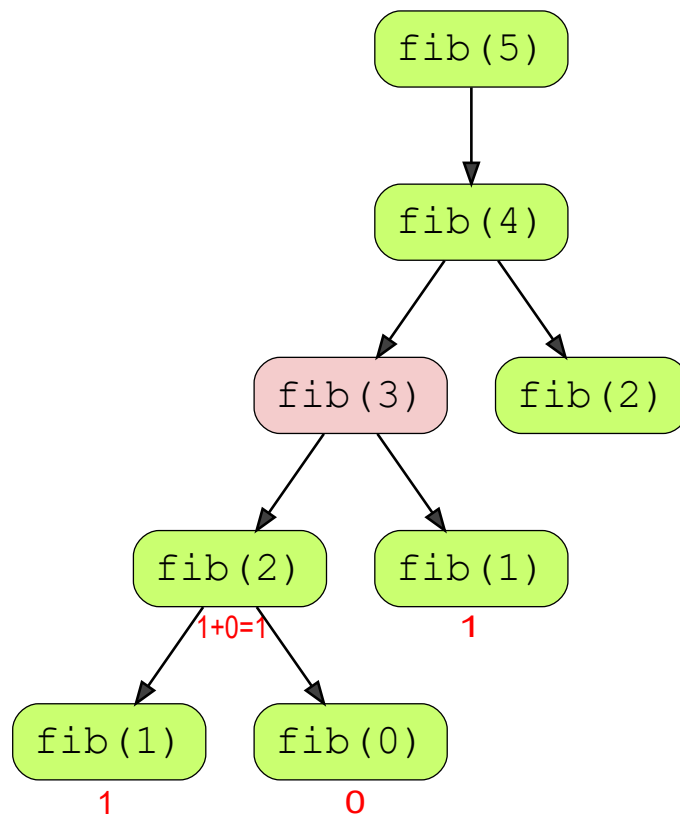


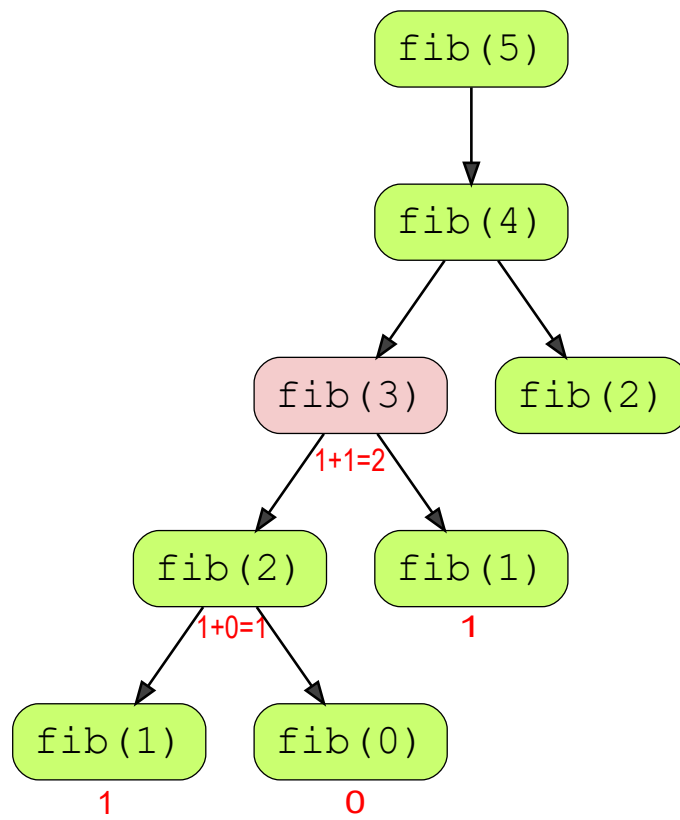


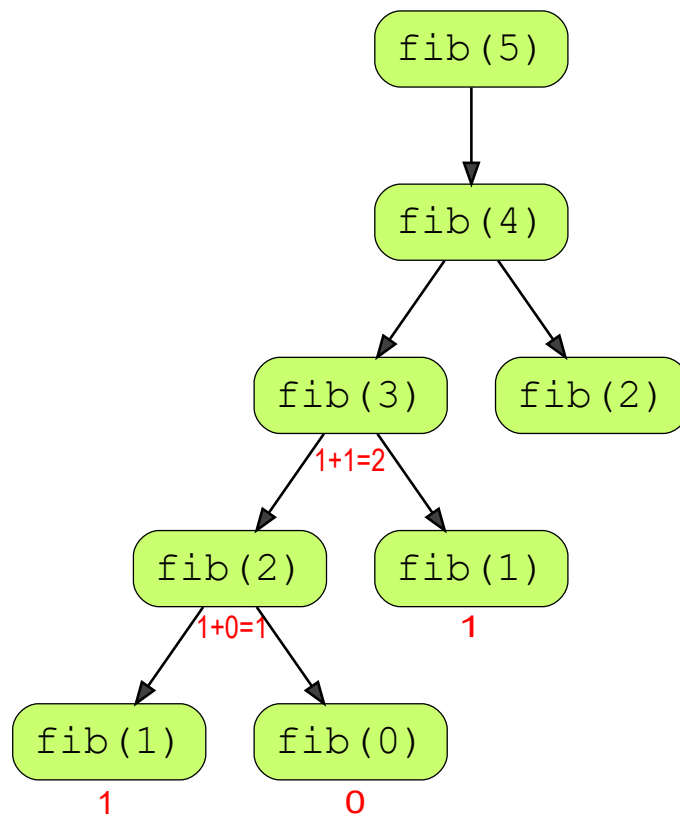


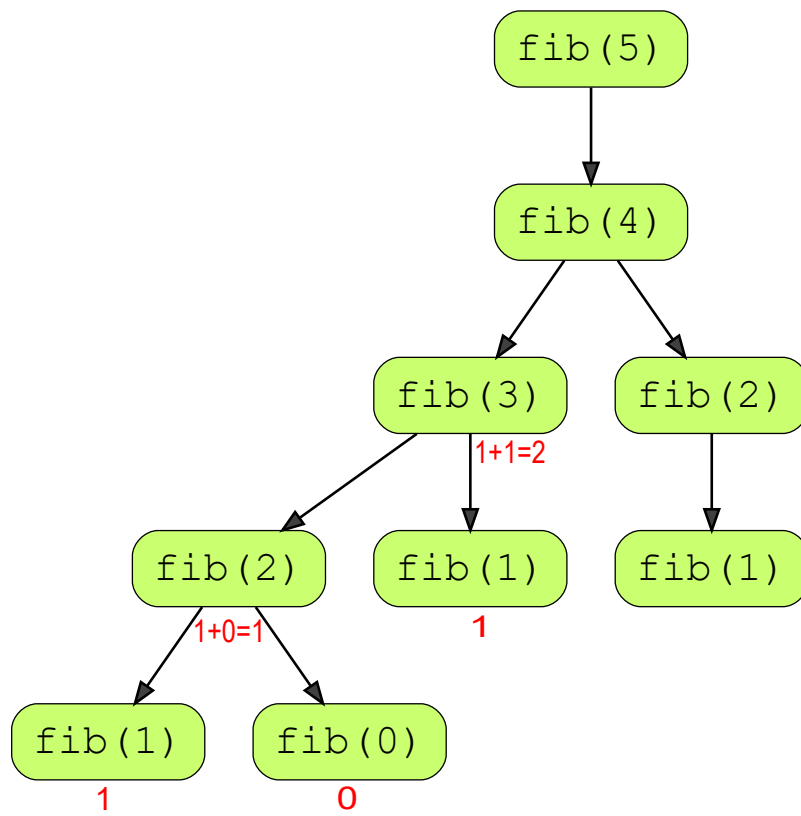


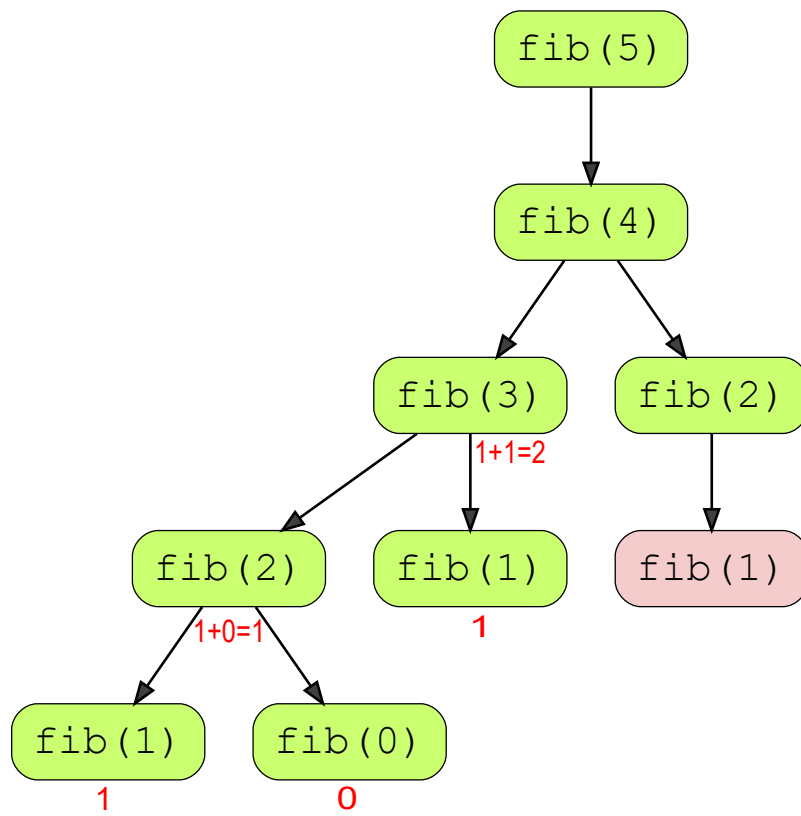


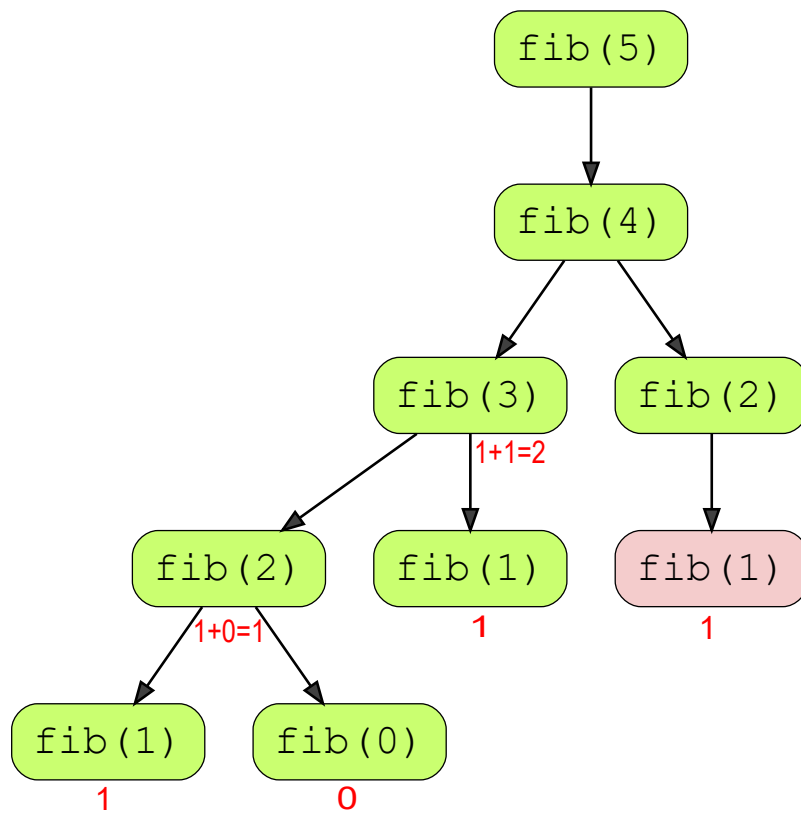


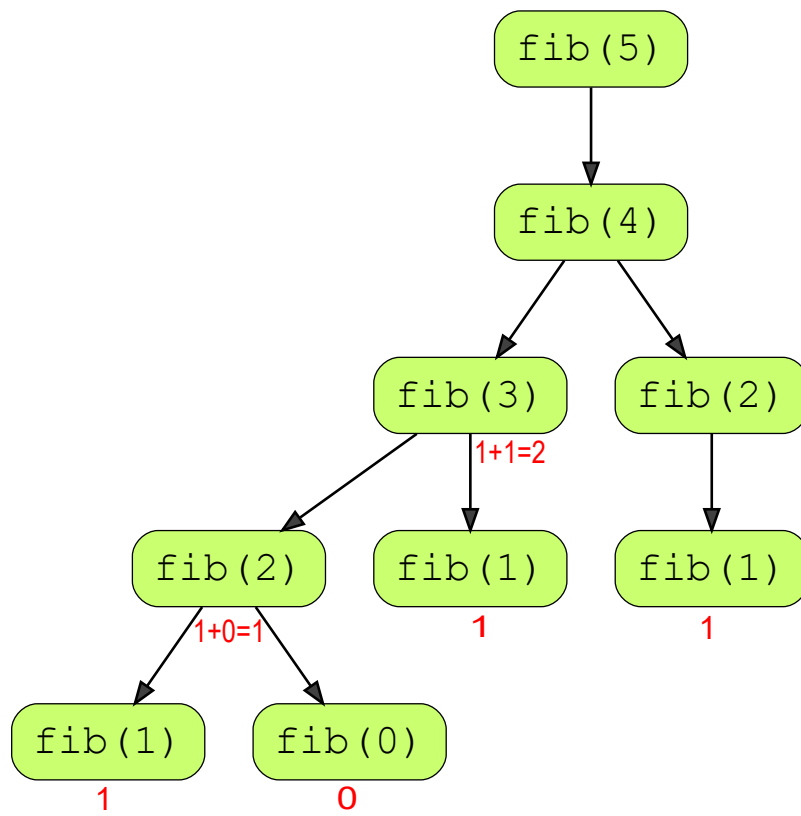


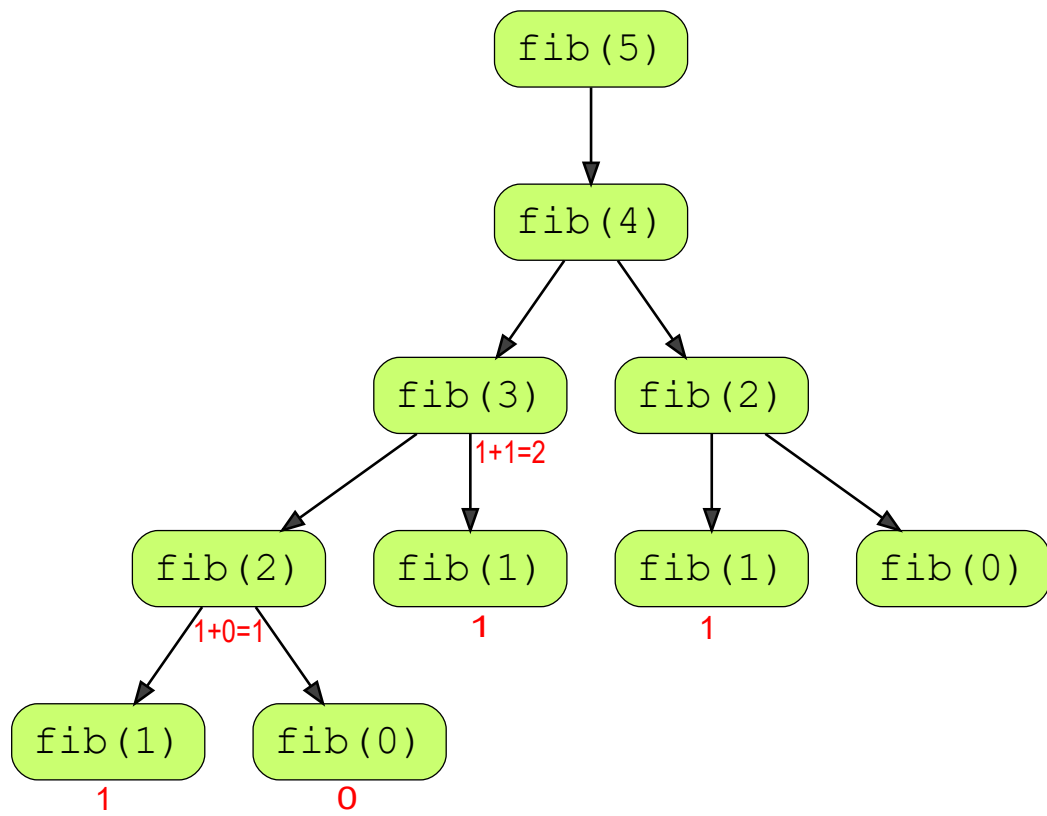


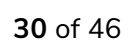


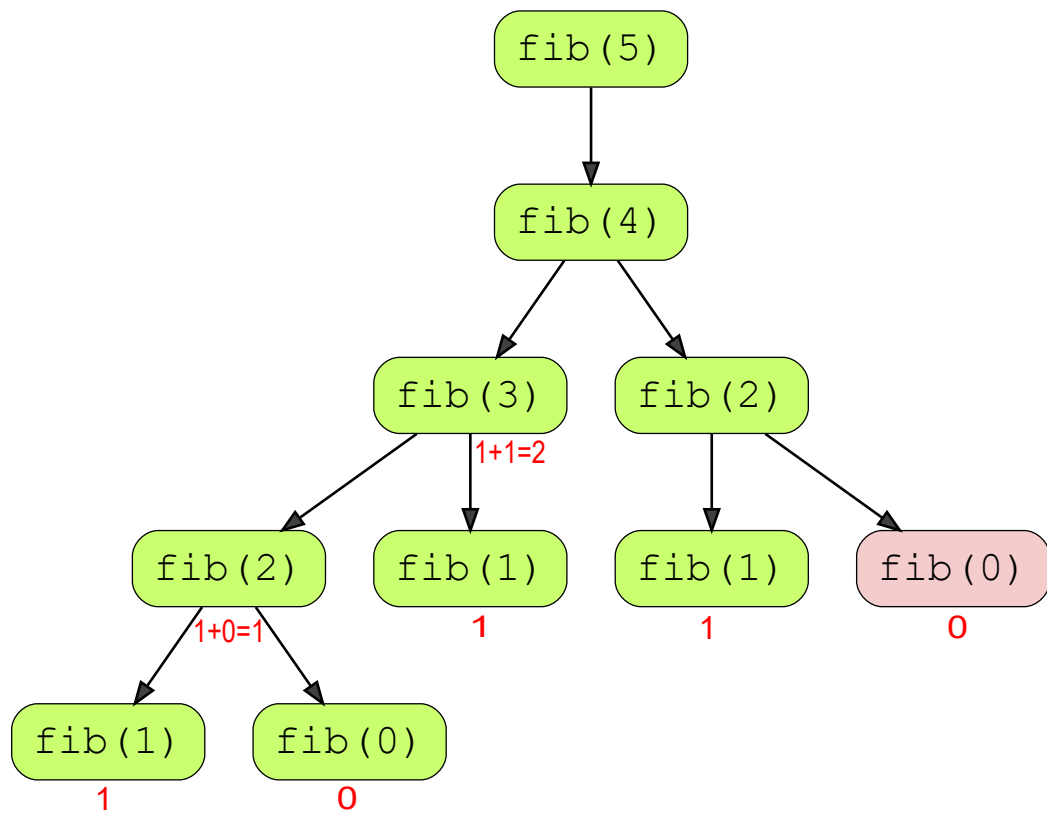


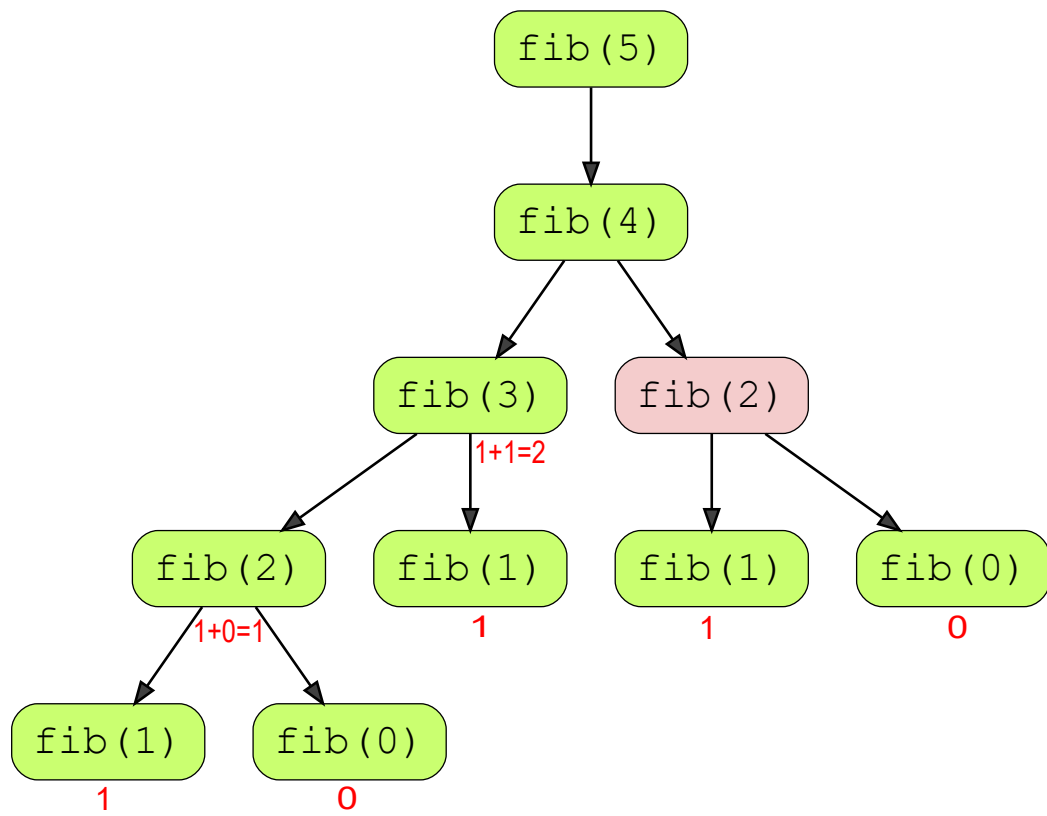


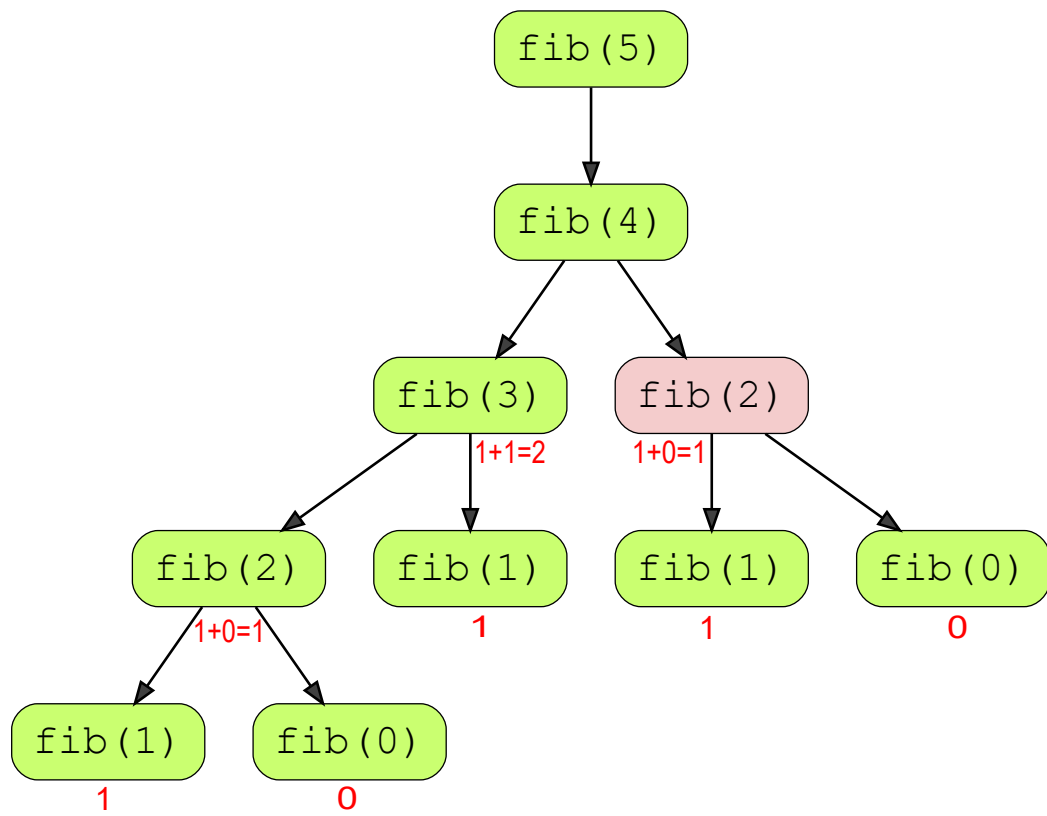


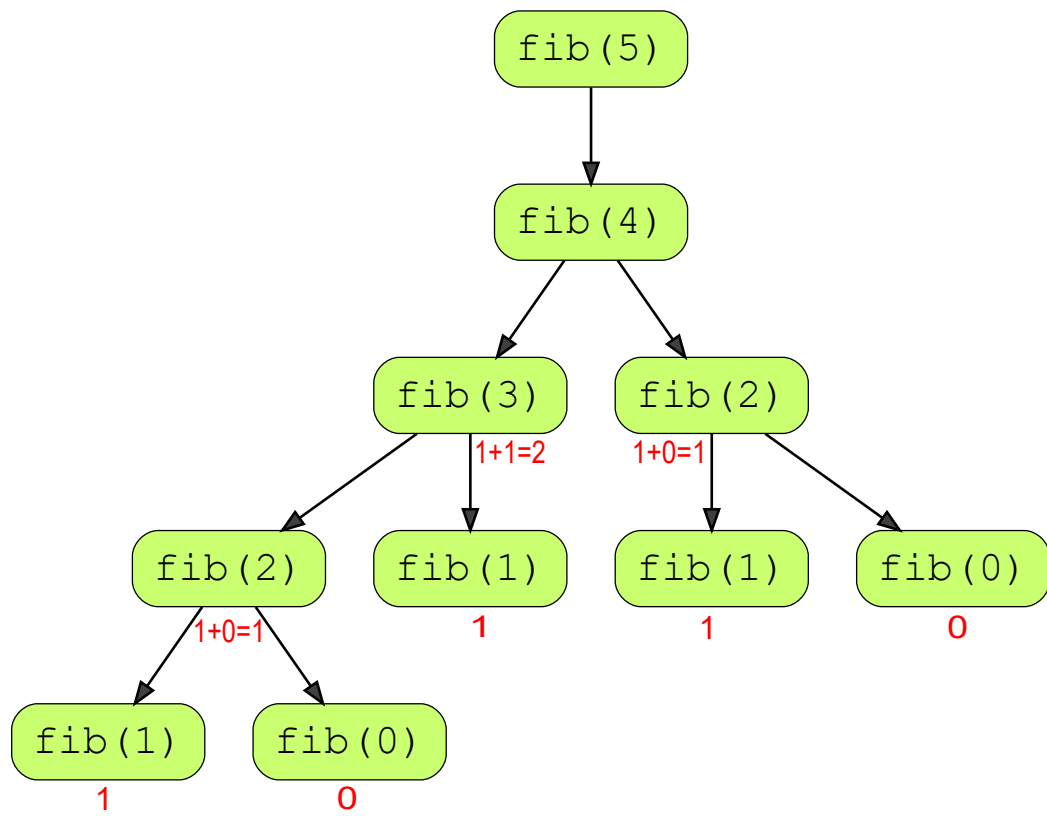


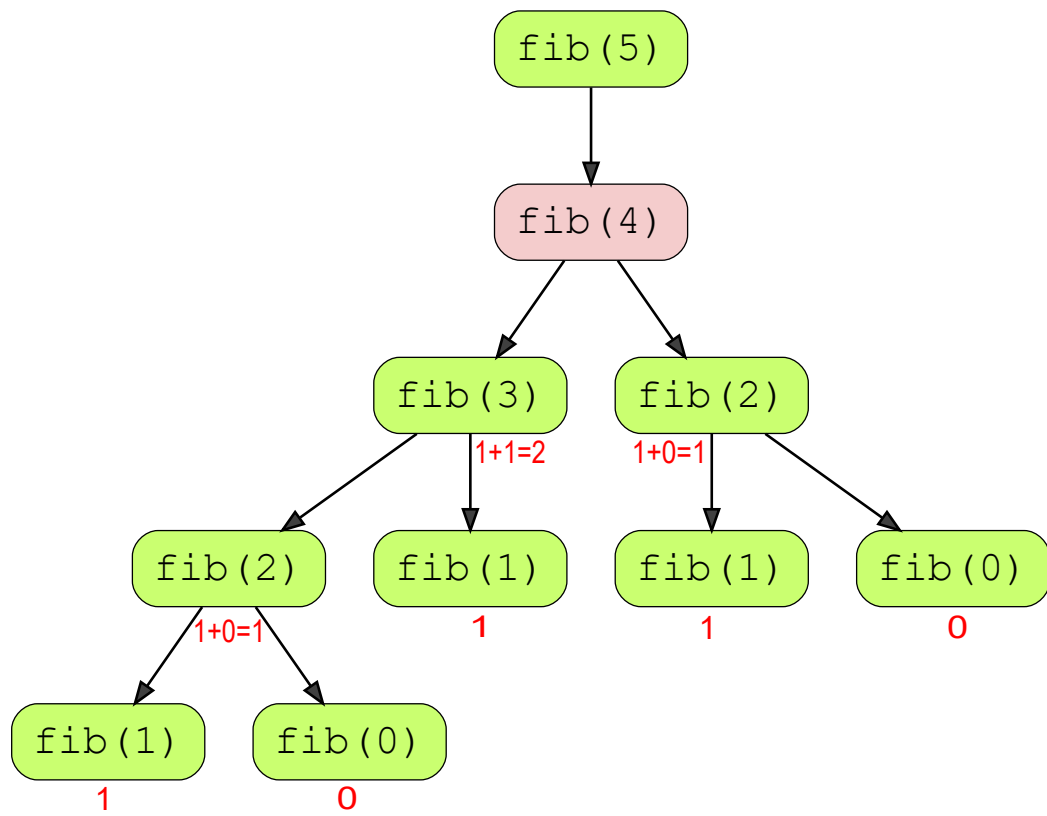


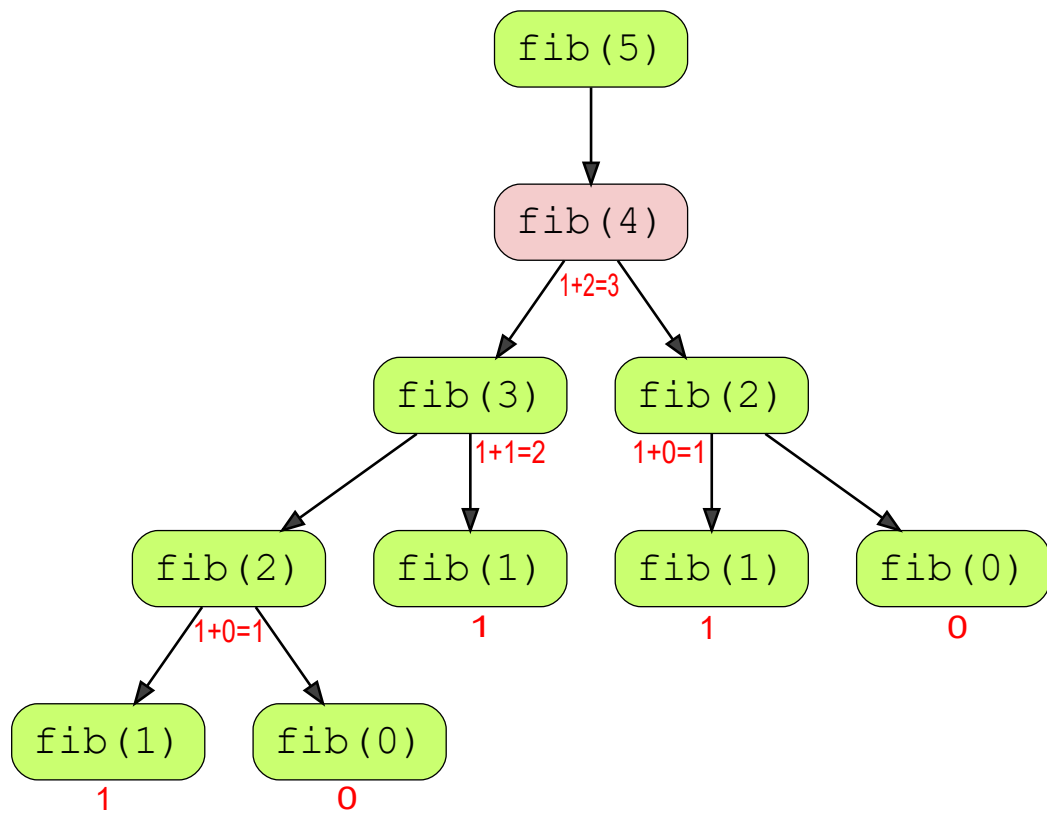


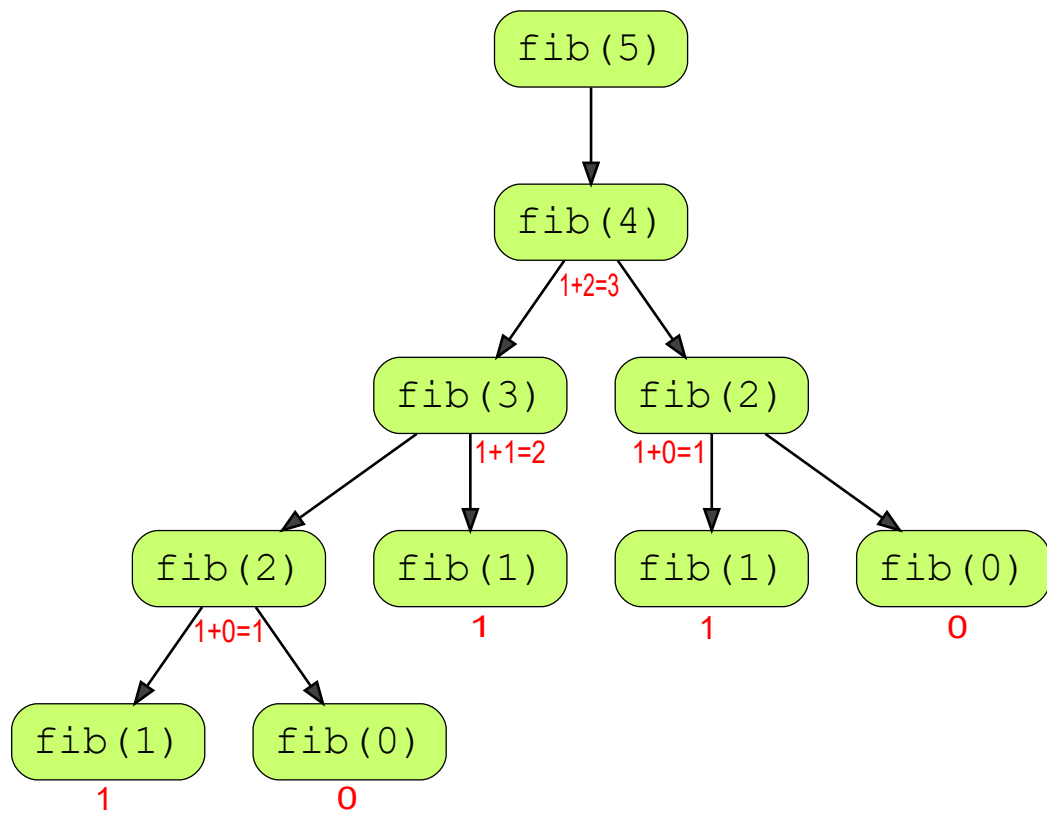


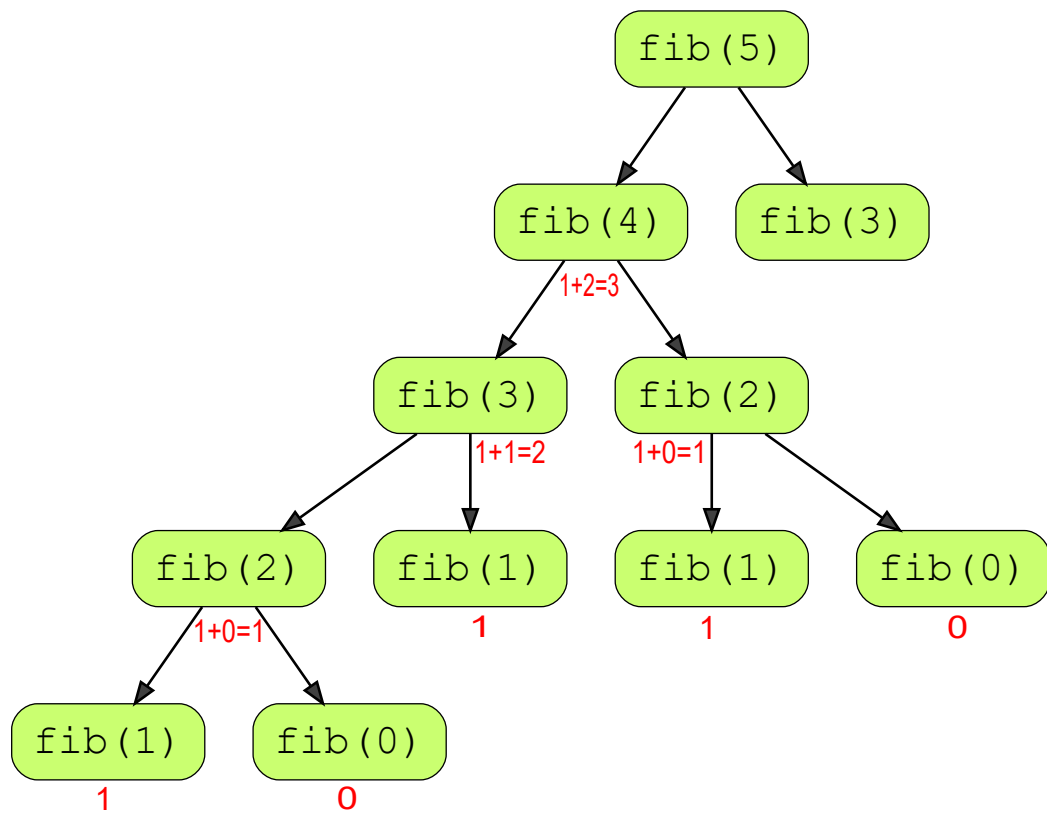


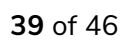


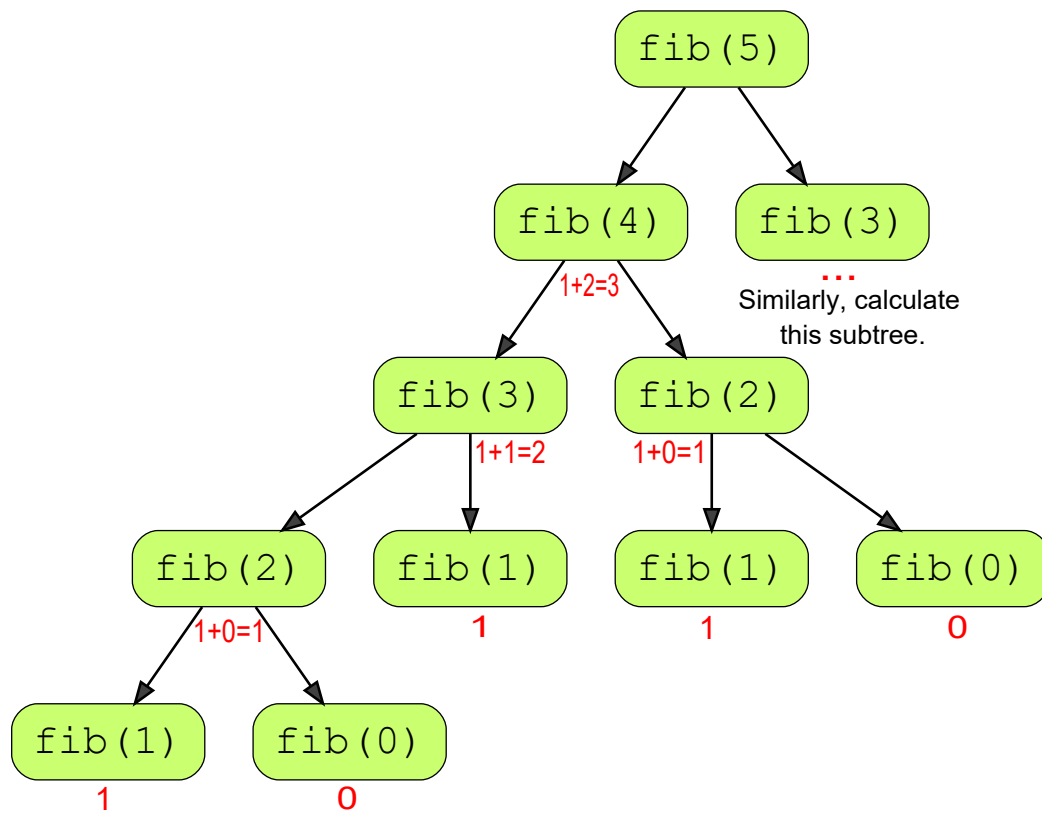


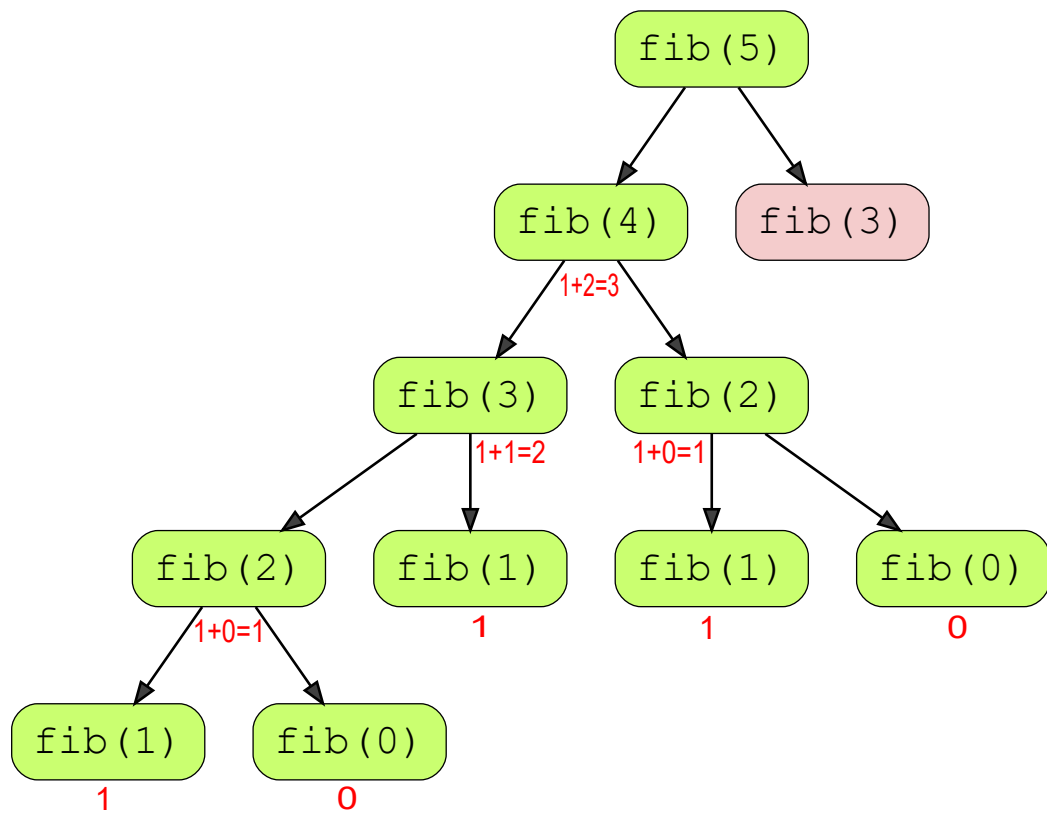


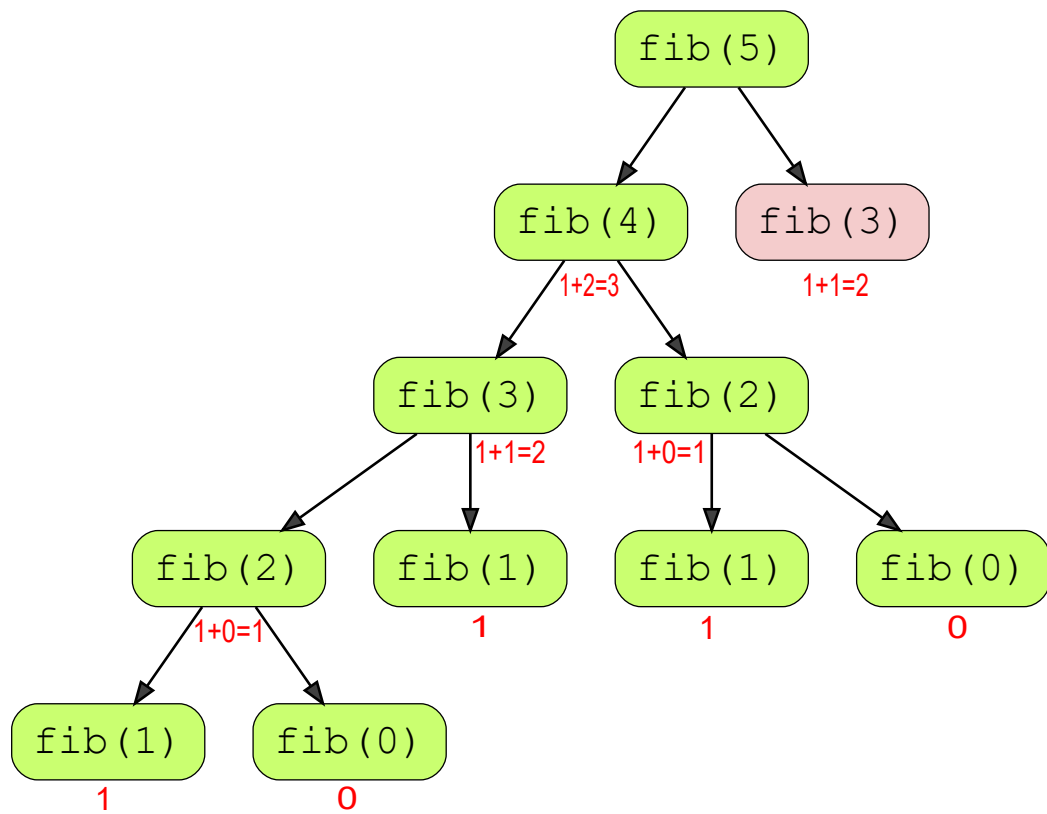


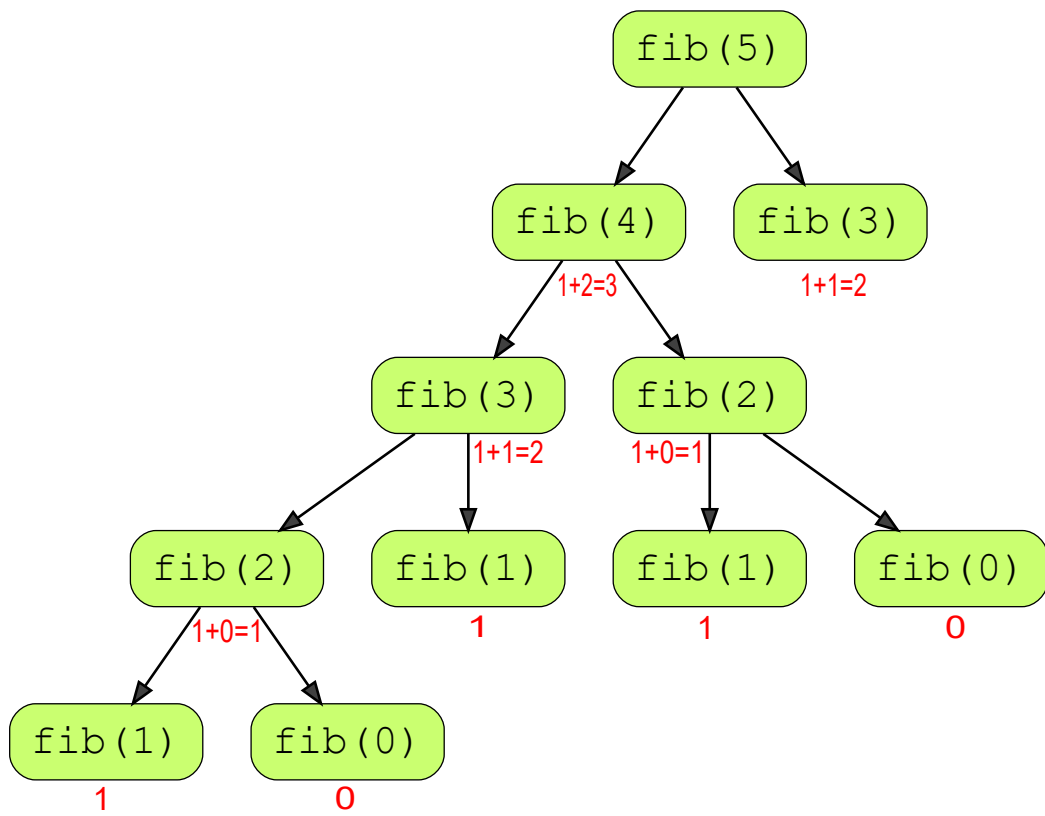


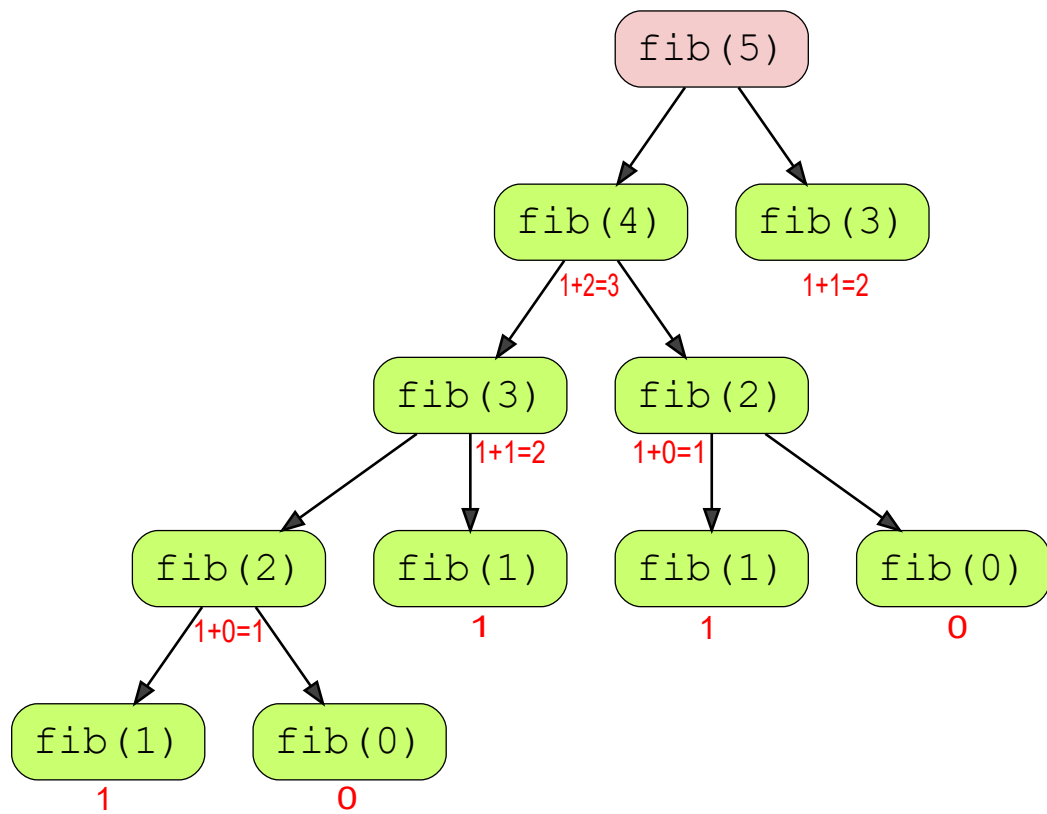


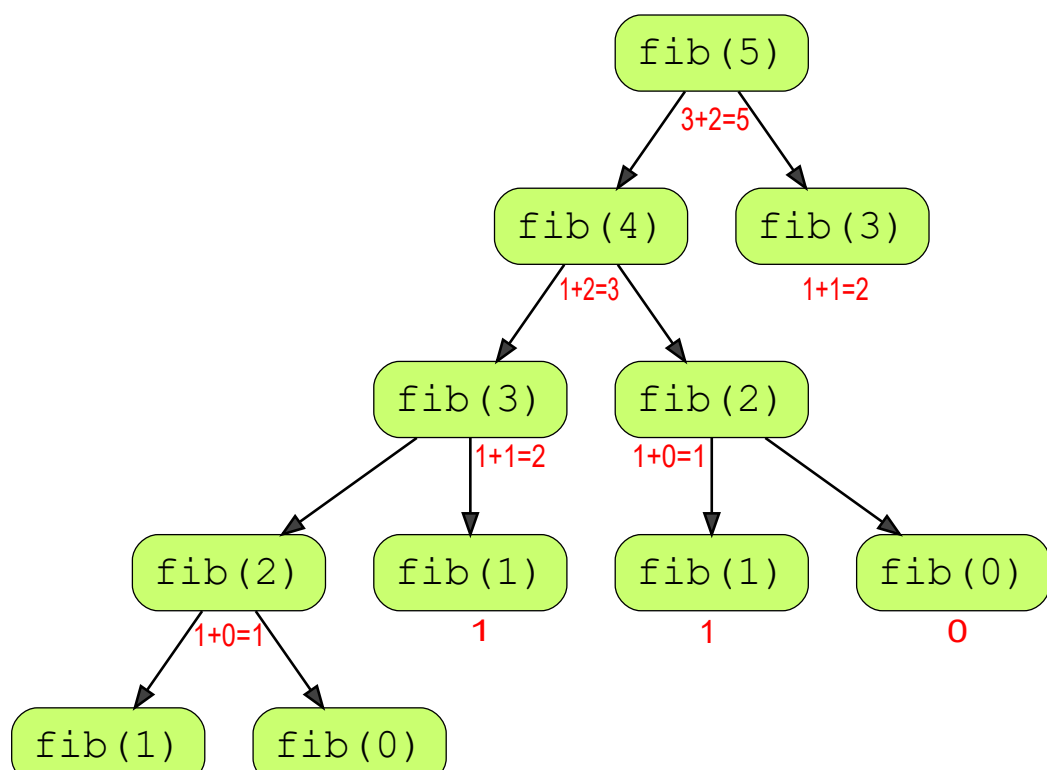
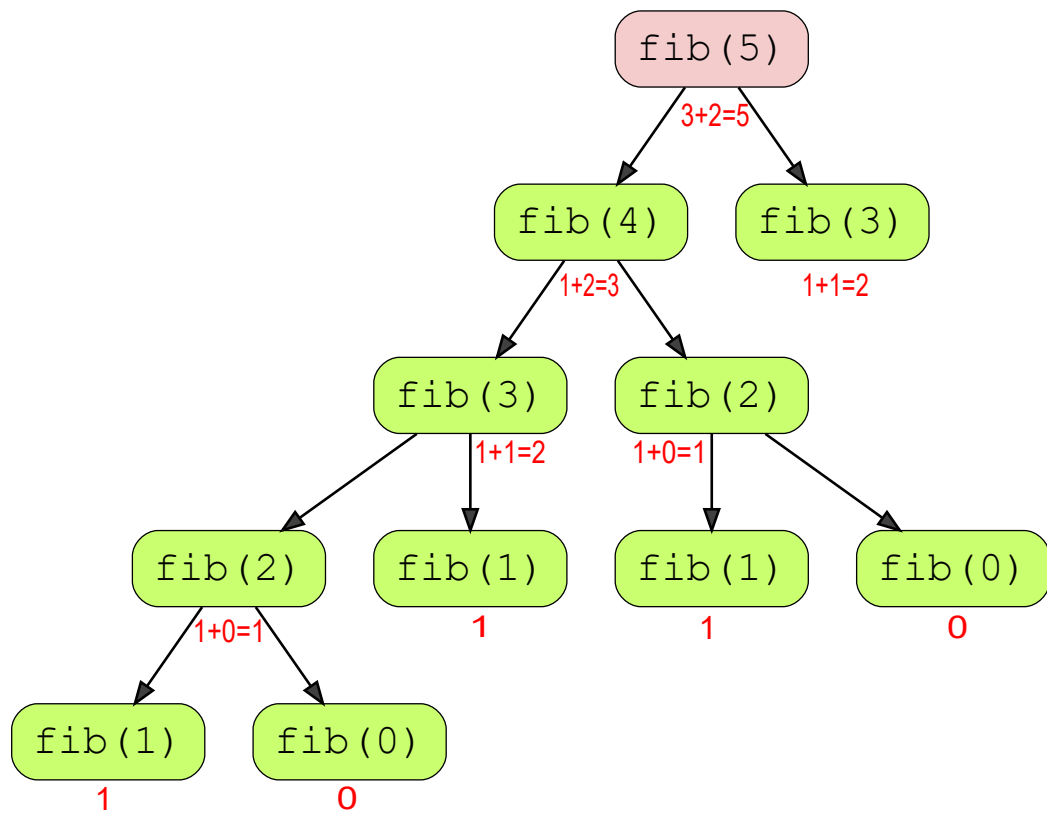














Notice that none of these have any clue that they are actually a part of a bigger process to calculate `fib(5)`. For instance, no `fib(1)` knows that it will be added to `fib(0)` and be used to calculate `fib(5)` at the end.

How do we know that `fib(3)` has to be added to `fib(2)` to make `fib(4)`, or `fib(2)` has to be added to `fib(1)` to make `fib(3)`? Where do we keep this information? All this information is in the call stack.

If at any point during the process, we were to pause and continue the process in a new environment with a clean stack, we would be unable to calculate `fib(5)` because the call stack has been wiped clean.

Think of a recursive process as a series of deferred operations, where there is information hidden to each recursive call - that hidden information is in the call stack.

Notice, that in the above code some functions, `fib(0)` and `fib(1)` for instance, are called multiple times. There is no memory of them ever being called before. As a result, the memory complexity is **O(n)**. If we run this program for large numbers, it throws an error.

```
function sumToN( n, sum = 0 ) {
  if ( n <= 1 ) return sum;
  let result = sum + n;
  return sumToN( n - 1, result );
};

console.time( 'recursion' );
console.log( sumToN( 1000000 ) ); //the code works if you replace 1000000 with a smaller number
console.timeEnd( 'recursion' );
```



RangeError: Maximum call stack size exceeded

Now we will write the same function using tail call optimization:

Now, we will write the same function using tail call optimization:

```
function fib(n, a, b){
  if (n === 0) {
    return b;
  } else {
    return fib(n-1, a + b, a);
  }
};
```



When we run this function, here is what the function calls will look like:

fib(5, 1, 0)

1 of 7

fib(5, 1, 0)
fib(4, 1, 1)

2 of 7

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

3 of 7

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

4 of 7

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

5 of 7

fib(5, 1, 0)
fib(4, 1, 1)
fib(3, 2, 1)
fib(2, 3, 2)
fib(1, 5, 3)
fib(0, 8, 5)

6 of 7

fib(5, 1, 0)
fib(4, 1, 1)
fib(3, 2, 1)
fib(2, 3, 2)
fib(1, 5, 3)
fib(0, 8, 5)

7 of 7

In this implementation, the entire state of the process is encapsulated in each function call. If we were to pause the process midway and start with a clean stack, we would still get the correct output. Moreover, the memory complexity is **$O(1)$** .

The current support for tail call optimization is currently very low. It will take time until you will be able to benefit from properly optimized tail calls. See the current [compatibility table](#).

On Google Chrome, the feature status can be seen [here](#). Currently, Safari is the only main browser that supports tail calls.

Once browser support is better, I will rewrite this section. Until then, know that tail call optimization exists.

We will now move on to the name property in ES6.