Recursive Functions

WE'LL COVER THE FOLLOWING

- Recursion
- Recursive functions in Go
  - Mutually recursive function

Recursion

Sometimes, the solution for bigger instances depends on the solution of smaller instances. A function calls itself for smaller instances until the problem is solved. This approach is called **recursion**.

Recursive functions in Go

A function that calls itself in its body is called a *recursive function*. The proverbial example is the calculation of the numbers of the *Fibonacci sequence*, in which each number is the sum of its two preceding numbers. The sequence starts with:

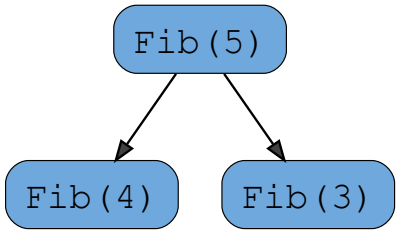
```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
 4181, 6765, 10946, ...
```

How about programming this problem? Consider a number **5**, let's see how recursive functions solve this problem. When we break the problem, we can visualize that:

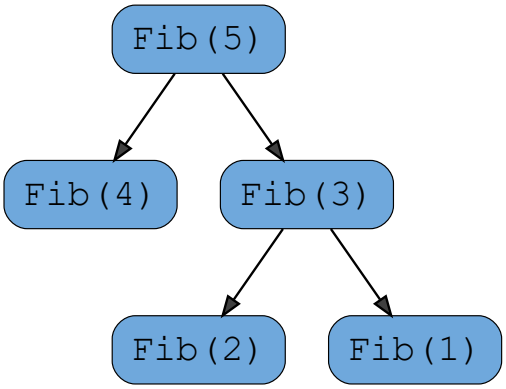
- Fi(5) = Fib(4)+Fib(3)
- Fi(4) = Fib(3)+Fib(2)
- Fib(3) = Fib(2)+Fib(1)
- Fib(2) = Fib(1)+Fib(0)

- Fib(1) = 1
- Fib(0) = 1

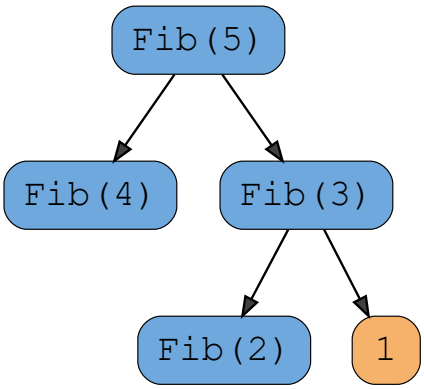
This is a recursive operation. Before coding, look at the illustration to understand in a better way.



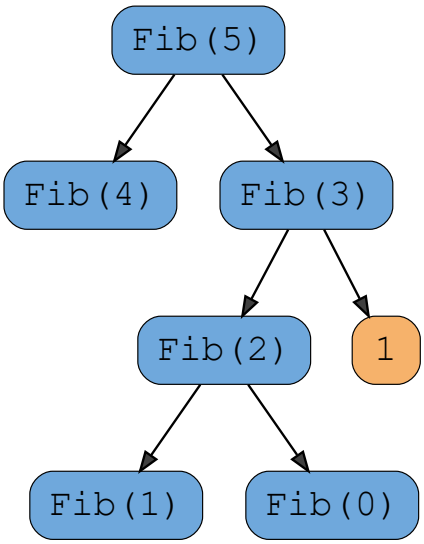
Calculating Fibonacci Sequence



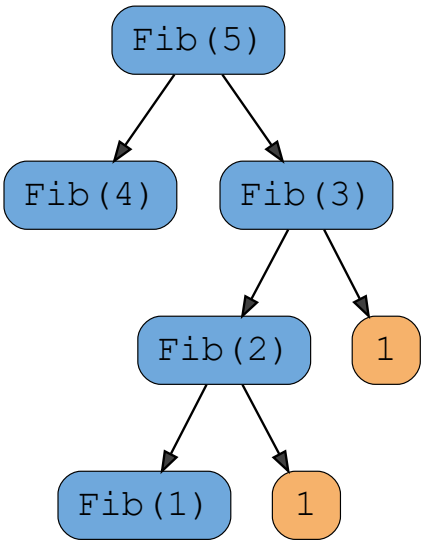
Calculating Fibonacci Sequence



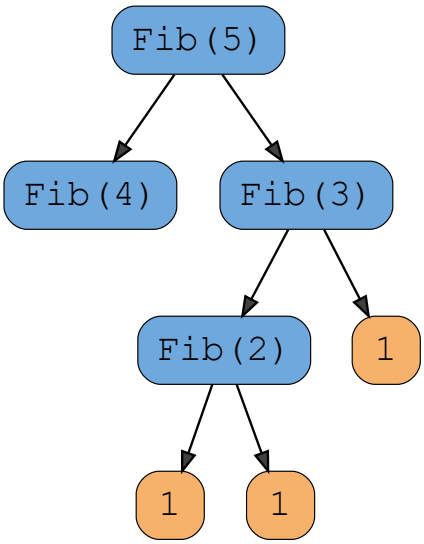
Calculating Fibonacci Sequence



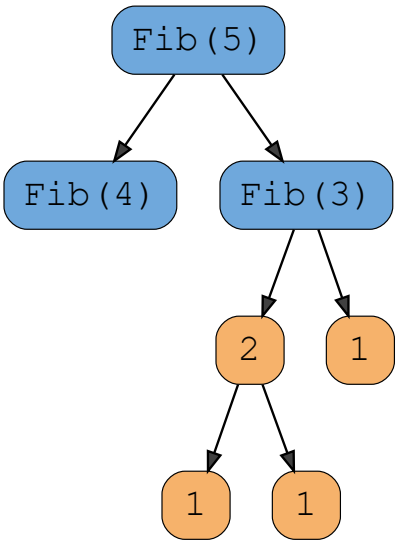
Calculating Fibonacci Sequence



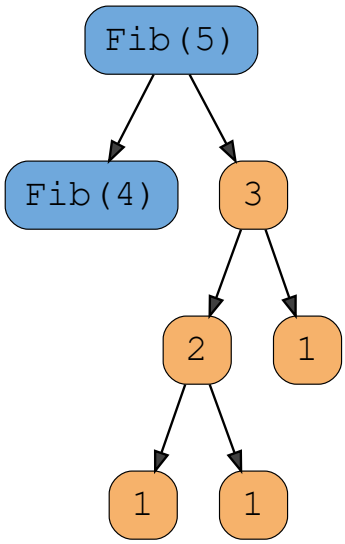
Calculating Fibonacci Sequence



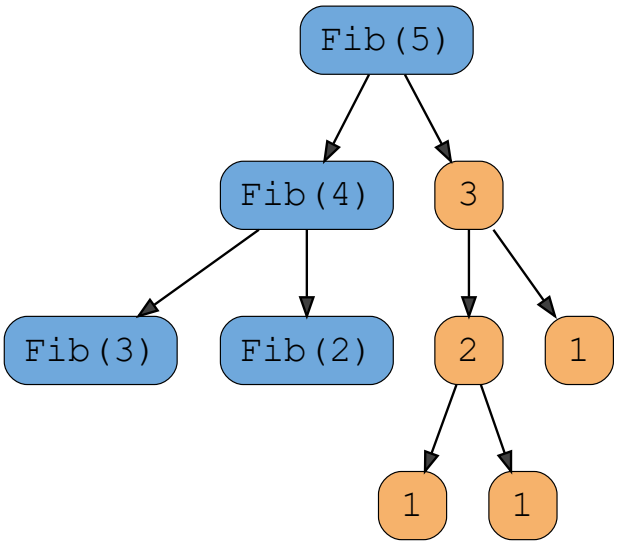
Calculating Fibonacci Sequence



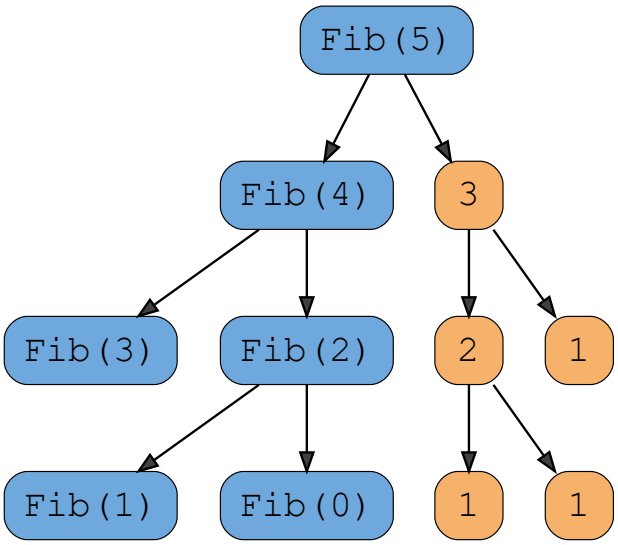
Calculating Fibonacci Sequence



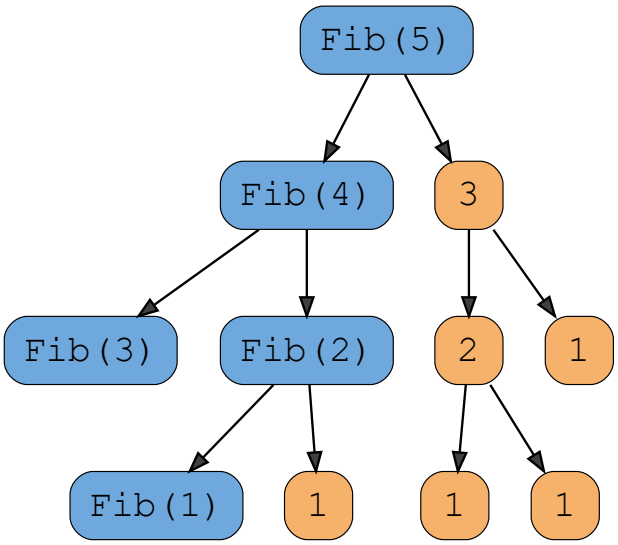
Calculating Fibonacci Sequence



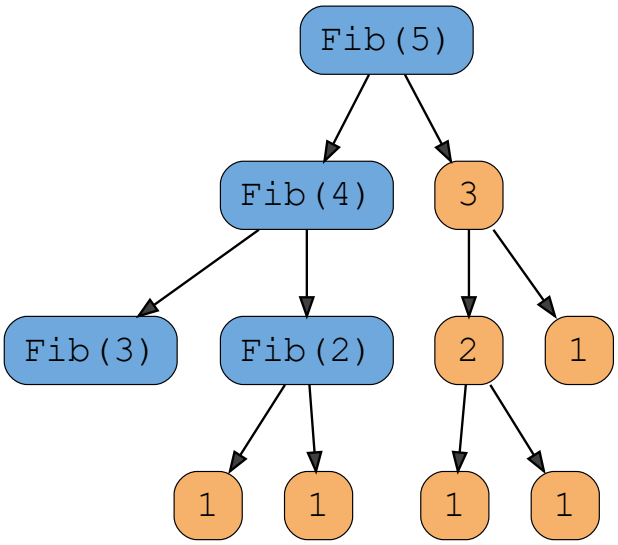
Calculating Fibonacci Sequence



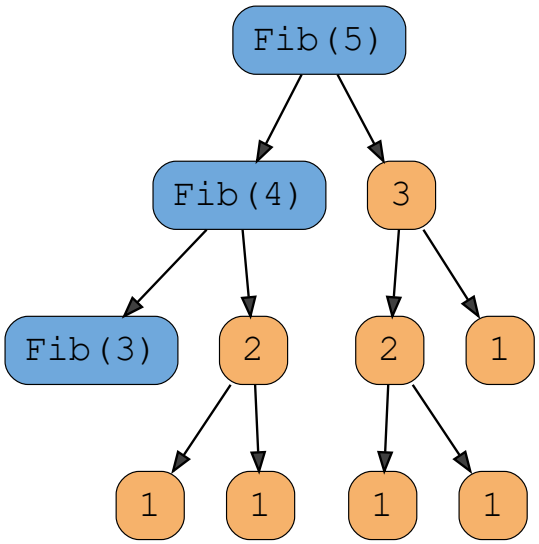
Calculating Fibonacci Sequence



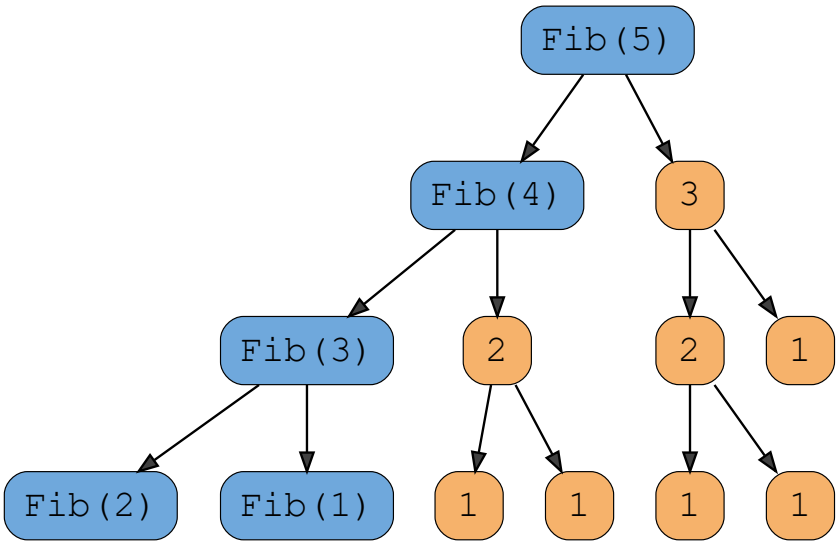
Calculating Fibonacci Sequence



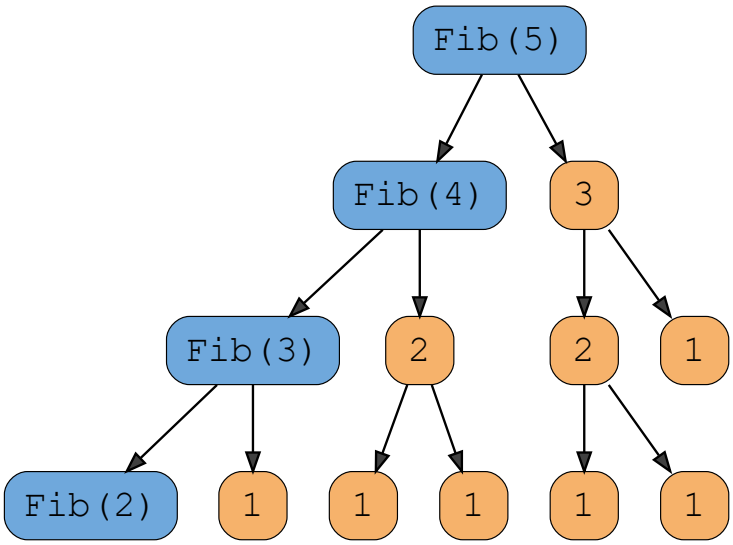
Calculating Fibonacci Sequence



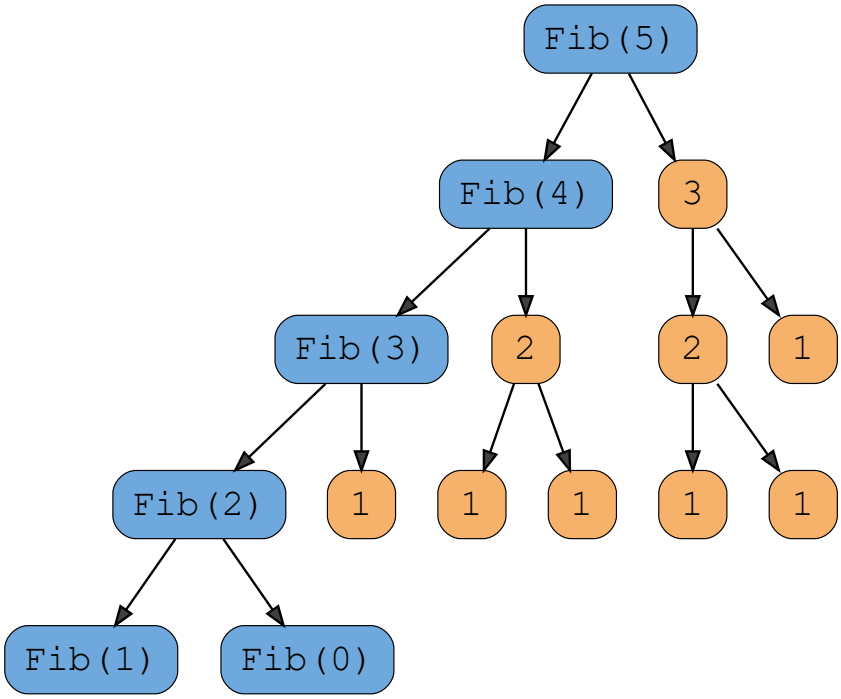
Calculating Fibonacci Sequence



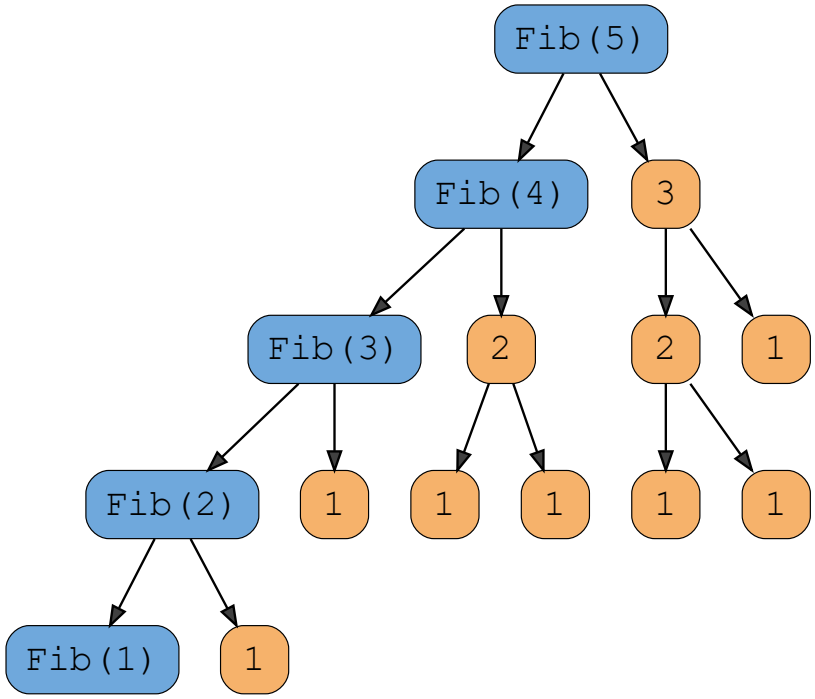
Calculating Fibonacci Sequence



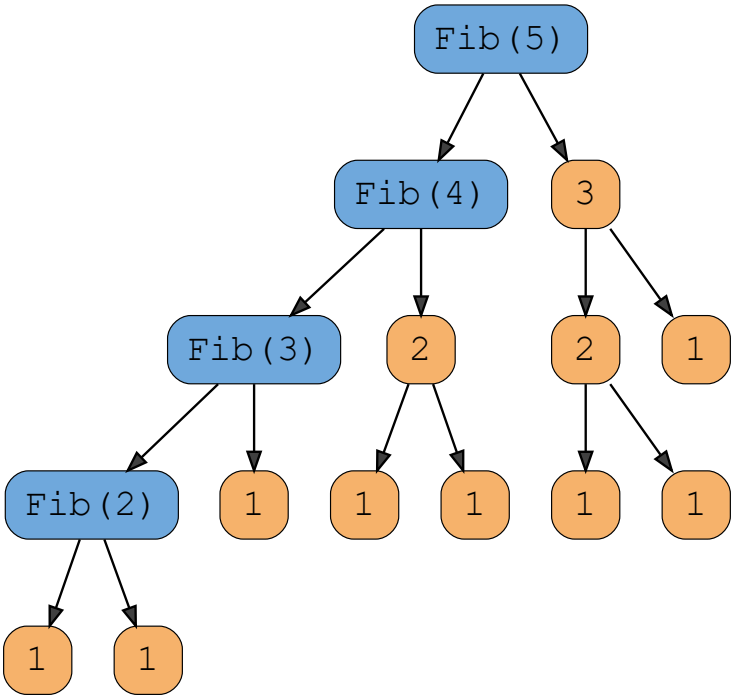
Calculating Fibonacci Sequence



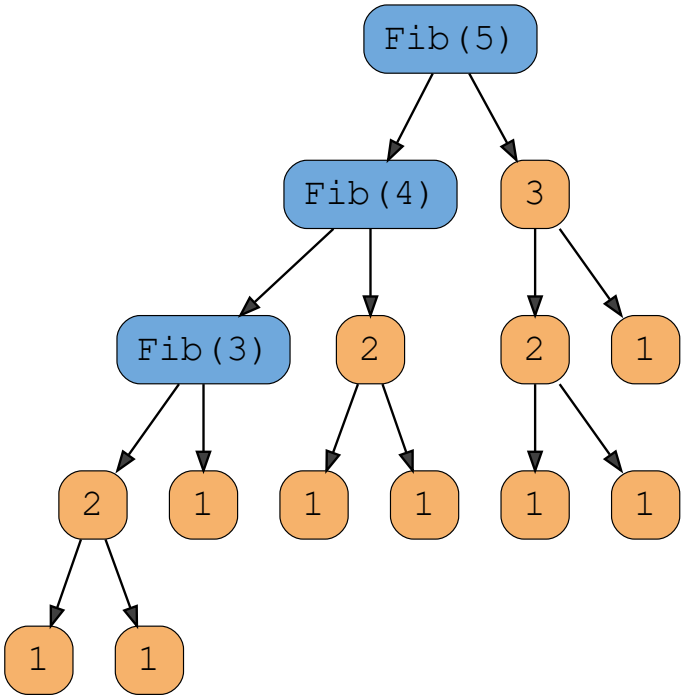
Calculating Fibonacci Sequence



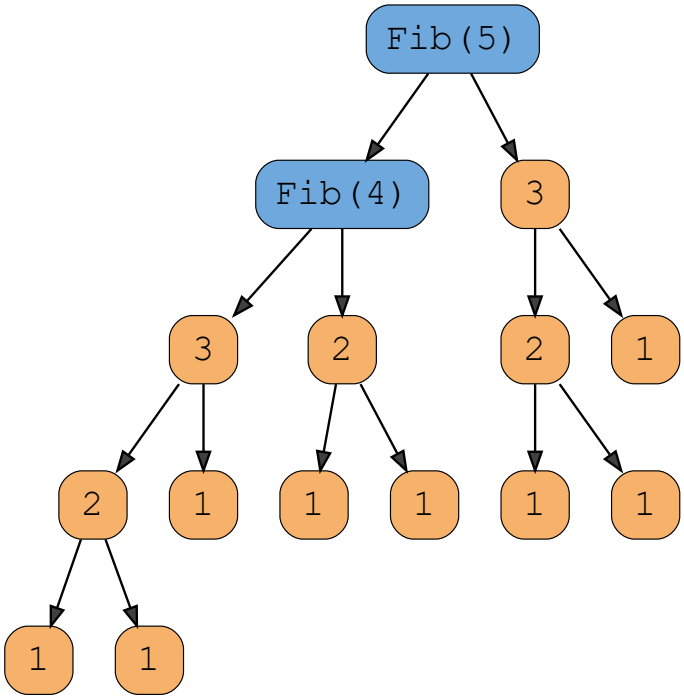
Calculating Fibonacci Sequence



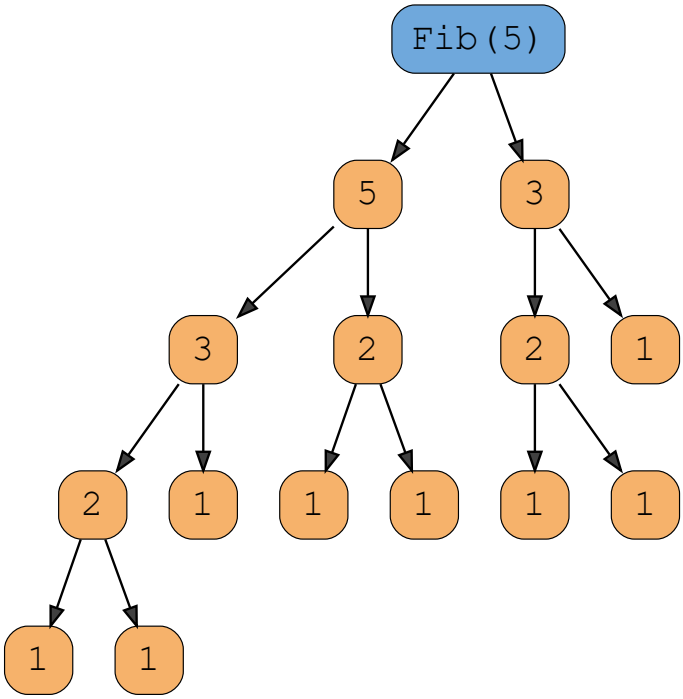
Calculating Fibonacci Sequence



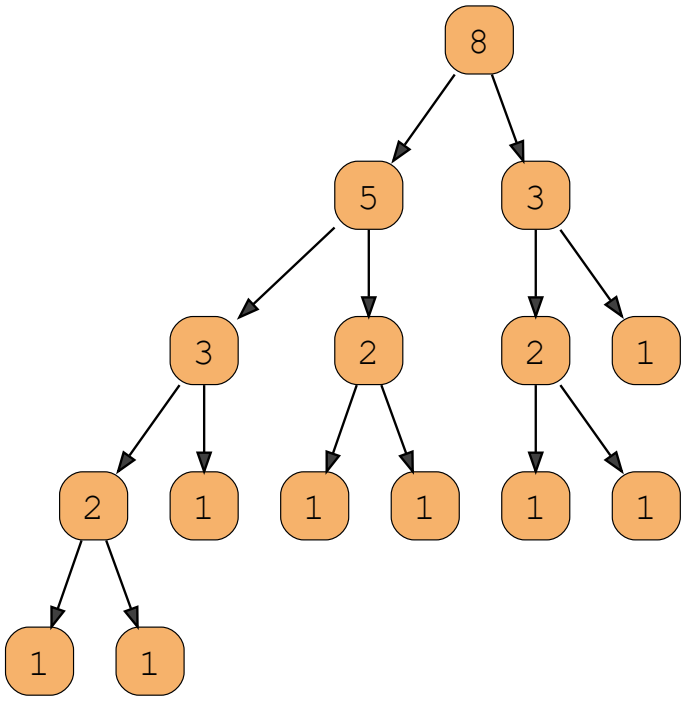
Calculating Fibonacci Sequence



Calculating Fibonacci Sequence



Calculating Fibonacci Sequence

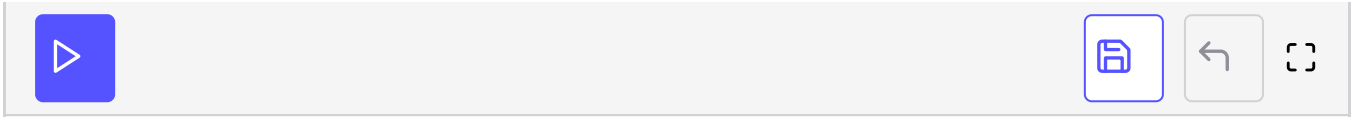


Calculating Fibonacci Sequence

Now, we are ready to write a program. See the following implementation.

```
package main
import "fmt"

func main() {
    result := 0
    for i := 0; i <= 10; i++{
        result = fibonacci(i) // function call
        fmt.Printf("fibonacci(%d) is: %d\n", i, result)
    }
}

// Function to calculate the nth fibonacci number
func fibonacci(n int)(res int) {
    if n <= 1 {
        res = 1
```

```
    } else {
        res = fibonacci(n - 1) + fibonacci(n - 2) // recursive call
    }

    return
}
```



Fibonacci Sequence

In `main`, we are calling function `fibonacci` for numbers less than and equal to **10** using a for loop at **line 6** and printing the `result` for every number. See the `fibonacci` function. We always need a *base* and a *recursive* case to implement any recursive function. From the illustration above, we know that when the number is *1* or *0*, we stop the calls and set the value to *1*. This is the base case. The following code implementats the base case. :

```
if n <= 1 {
    res = 1
}
```

Then, we have a recursive case. You may have noticed the pattern. The `nth` number in the Fibonacci sequence is equal to:

```
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

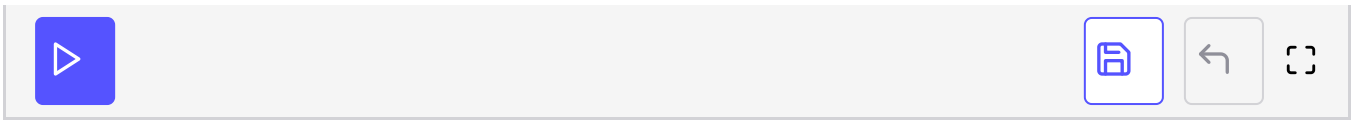
This pattern is implemented at **line 17**. We start solving this problem from smaller instances, which in turn solve problems for bigger instances.

An important problem when using recursive functions is *stack overflow*. This can occur when a large number of recursive calls are needed, and the program runs out of allocated stack memory. This can be solved by using a technique called **lazy evaluation**, implemented in Go with a channel, and a goroutine (see Chapter 12).

Mutually recursive function

Mutually recursive functions can also be used in Go. These are functions that call one another. Because of the Go compilation process, these functions may be declared in any order. Here is a simple example:

```go
package main
import (
    "fmt"

)

func main() {
    fmt.Printf("%d is even: is %t\n", 16, even(16)) // 16 is even: is true
    fmt.Printf("%d is odd: is %t\n", 17, odd(17)) // 17 is odd: is true
    fmt.Printf("%d is odd: is %t\n", 18, odd(18)) // 18 is odd: is false
}

func even(nr int) bool {
    if nr == 0 {
        return true
    }
    return odd(RevSign(nr) - 1) // even calls odd
}

func odd(nr int) bool {
    if nr == 0 {
        return false
    }
    return even(RevSign(nr) - 1) // odd calls even
}

func RevSign(nr int) int {
    if nr < 0 {
        return -nr
    }
    return nr
}
```



Mutually Recursive Functions

To find out whether a number is even or odd, we can use the modulus operator, but we are instead solving the problem through mutually recursive functions.

As you can see in the above code, at **line** 7 within `main`, we are calling a function `even`, and passing **16** as a parameter. In function `even`, we check that the number `nr` is *0* or not at **line 13**. If it's *0*, then we return `true`. Otherwise, we call the `RevSign` function. `RevSign` returns the *absolute value* of the number.

Control goes back to **line 16**. Now, we subtract **1** from the value returned by `RevSign`. The result will go as a parameter to the `odd` function now. In function `odd`, we check that the number `nr` is **0** or not at **line 20**. If it's **0**, then we return *false*. Otherwise, we call the `RevSign` function. `RevSign` checks that

if `nr` is less than **0**, it returns `-nr`. Otherwise, it returns `nr`.

Control goes back to **line 23**. Now, the result will go as a parameter to the `even` function. The same process will go on and on until `nr` is **0**. The moment `nr` is zero, we can make a decision whether the number was even or not.

Now that you are familiar with recursion in Go, attempt a challenge to evaluate your understanding.