Yogesh Sajanikar

# Writing recursive functions

In this recipe, we will be looking at writing recursive functions, and tail recursions which will allow efficient, infinite recursion.

We will take famous fibonacci number to implement our solution. We will then take this very simple solution and then convert it to a tail recursive efficient solution.

## Fibonacci Number

We know that, the first two fibonacci numbers are 0, and 1, and remaining numbers are calculated by a simple formula,

## Recursive solution

Now we can implement the solution. First we will start with declaration.

fib :: Int -> Int

Given an integer argument, n, we will return n'th fibonacci number.

### Base cases

The base cases are straight forward, we disallow negative arguments. And precalculate 0'th and 1'st fibonacci numbers.

fib n | n < 0 = error "Negative index"  
fib 0 = 0  
fib 1 = 1

### Remaining cases

All that remains is to calculate the remaining cases,

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

We have recursively called the same function.

### Is recursive solution efficient

We can run the soltion to check the efficiency. In GHCi, we load the function, and set the timing/memory stats to be printed after each execution.

λ> :set +s

And now we can run the function 'fib' with increasing values.

λ> fib 10  
55  
(0.00 secs, 0 bytes)  
λ> fib 20  
6765  
(0.05 secs, 15,287,704 bytes)  
λ> fib 30  
832040  
(3.69 secs, 1,262,480,024 bytes)

You can easily see that though the implementation is simple, and even though the change in the input argument linear, the timing has shot up exponentially.

This is because haskell is lazy (it will not evaluate the expression unless it is evaluated). And most importantly, the above implementation is not tail recursive. i.e. during the recursion, the recursive expression is not the function call itself. In the above implementation, "fib (n-1) + fib (n-2)" is the expression where recursion occurs. And since the expression actually is ( (+ ) (fib (n-1)) (fib (n-2))), one can see that the result expression contains an operation (+) which is required to be evaluated after calling the two recursive calls. This requires compiler to build up stack.

If on the other hand, if we can make the last expression, the function call itself, compiler can actually optimize the recursion in such a way that only function call is kept in the stack. This is called tail recursion.

### Implementing tail recursive function by worker pattern

Since the very simple recursive definition is not efficient, we will convert the definition to a tail recursive. We use worker pattern here. In this pattern, we create a worker function and pass an additional argument where we keep the result. Whenever we hit the base case, we return the result.

In case of fibonacci number, the number depends upon two previous number. Let's use previous numbers as argument to our worker function.

fibworker :: Int -> Int -> Int -> Int  
fibworker n p1 p2 = \_

Where p1 and p2 are previous fibonacci numbers respectively, and p1 > p2.

We then calculate, next number by adding two previous results, n1 and n2. Since this is a new number, we make following recursive call.

fibworker n p1 p2 = fib (n-1) (p1+p2) p1

Note that we have replaced previous fibonacci numbers p1 and p2 by (p1+p2) and p1 in the recursive calls. Also notice that we have reduced n to n-1 to indicate that we have done with a step in the calculation of fibonacci number.

The recursive call will soon hit '2' in the first argument. When that happens, we will be ready to spit out the result, by just adding two previous values.

Why '2', you might ask. Do remember in the worker we use two previous values. It means that we have to stop recursion at '2' when we already have two values.

fibworker 2 p1 p2 = p1 + p2

We can trivially complete the remaining cases, 0, and 1 by directly returning the result

fibworker 0 \_ \_ = 0  
fibworker 1 \_ \_ = 1

We now can wrap worker in another function, called 'fib1' so that we can provide initial values to the worker function. The initial values are second and first fibonacci numbers, i.e. 1 and 0 respectively.

fib1 n | n < 0 = error "Negative index"  
fib1 n = fibworker n 1 0

Also notice that the fibworker is an internal function, and need not be used elsewhere, and hence we can move the whole definition of fibworker as a where construct,

fib1 n | n < 0 = error "Negative index"  
fib1 n = fibworker n 1 0  
 where  
 fibworker :: Int -> Int -> Int -> Int  
 fibworker 0 \_ \_ = 0  
 fibworker 1 \_ \_ = 1  
 fibworker 2 p1 p2 = p1 + p2  
 fibworker n p1 p2 = fibworker (n-1) (p1+p2) p1

1. Running efficient version

* We can now test the efficient version in GHCi.
* λ> fib1 10  
  55  
  (0.00 secs, 0 bytes)  
  λ> fib1 20  
  6765  
  (0.00 secs, 0 bytes)  
  λ> fib1 30  
  832040  
  (0.00 secs, 0 bytes)
* You can see that the function execution is much more efficient, and memory usage is very low.

### Fibonacci Evaluator

The source for the fibonacci evaluator is given here.

module Fibonacci where  
  
-- Naive recursive fibonacci implementation.  
fib :: Int -> Int  
fib n | n < 0 = error "Negative index"  
fib 0 = 0  
fib 1 = 1  
fib n = fib (n-1) + fib (n-2)  
  
-- Efficient fibonacci implementation with worker pattern  
fib1 n | n < 0 = error "Negative index"  
fib1 n = fibworker n 1 0  
 where  
 fibworker :: Int -> Int -> Int -> Int  
 fibworker 0 \_ \_ = 0  
 fibworker 1 \_ \_ = 1  
 fibworker 2 p1 p2 = p1 + p2  
 fibworker n p1 p2 = fibworker (n-1) (p1+p2) p1