## Accurate Computations and the big Package

This lesson discusses how Go ensures accurate computations in its program via the big package.

## WE'LL COVER THE FOLLOWING ^

- A major problem
- Solution provided by Go

## A major problem #

We know that programmatically performed floating-point computations are sometimes not accurate. If you use Go's *float64* type in floating-point numbers computation, the results are accurate to about **15** decimal digits, enough for most tasks. When computing with very big whole numbers, the range of the types *int64* or *uint64* might also be too small. In that case, *float32* or *float64* can be used if accuracy is not a concern, but if it is, we cannot use floating-point numbers because they are only represented in an approximated way in memory.

## Solution provided by Go #

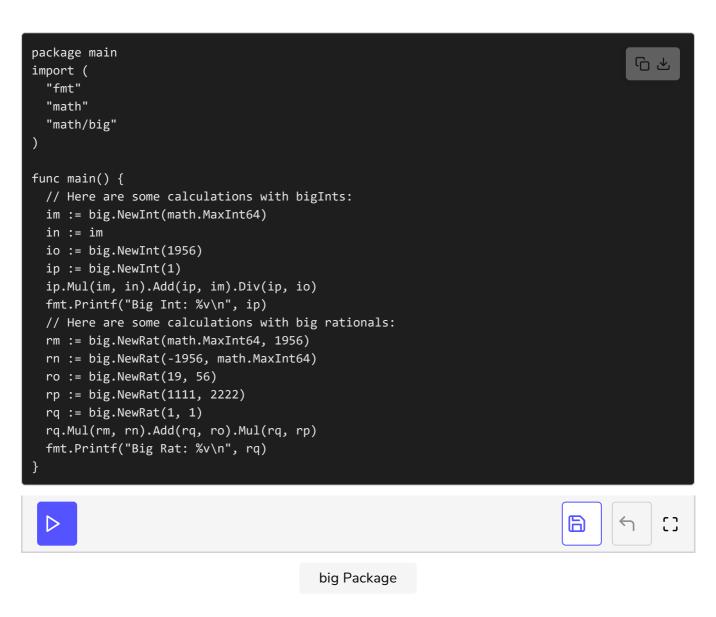
For performing perfectly accurate computations with integer values Go provides the <code>math/big</code> package contained in <code>math</code>: <code>big.Int</code> for integers, <code>big.Rat</code> for rational numbers (these are numbers than can be represented by a fraction like 2/5 or 3.1416 but not irrational numbers like <code>e</code> or  $\pi$ ) and <code>big.Float</code>. These types can hold an arbitrary number of digits, only limited by the machine's available memory. The downside is the bigger memory usage and the processing overhead, which means they are a lot slower to process than built-in integers.

A big integer is constructed with the function big.NewInt(n), where n is an int64. A big rational number with big.NewRat(n, d), where both n (the numerator) and d (the denominator) are of type int64. A big float number is

constructed with the function  $\frac{\text{big.NewFloat(x)}}{\text{big.NewFloat(x)}}$ , where x is a float64.

Because Go does not support operator overloading, all the methods of the big types have names like Add() for addition and Mul() for multiplication. They are methods (see Chapter 8) acting on the integer, rational or float as a receiver. In most cases, they modify their receiver and return the receiver as a result, so that the operations can be chained. This saves memory because no temporary big.Int variables have to be created to hold intermediate results.

We see this in action in the following program:



In the program above, outside the main function at line 5, we import the big package for accurate computations. In main, we perform some calculations with big integers and then big rationals. At line 10, we are returning an \*Int set to the value of the math.MaxInt64, which is equal to 9223372036854775807 and storing it in im. In the next line, we are declaring a new variable in and initializing it with im. At line 12, we are returning an \*Int set to the value of the 1956 and storing it in io. At line 13, we are returning an \*Int set to the

value of the 1 and storing it in ip.

Now at **line 14**, we are applying arithmetic functions as: <code>ip.Mul(im, in).Add(ip, im).Div(ip, io)</code>. First, the values of <code>im</code> and <code>in</code> will be multiplied and stored in <code>ip</code>. Then the value of <code>im</code> and updated value of <code>ip</code> will be added together, and the result will be stored again in <code>ip</code>. Now the updated value of <code>ip</code> will be divided with the value of <code>io</code>, and the final result will again be stored in <code>ip</code>. In the next line, we are printing <code>ip</code> to check the final value.

\*Rat set to the value of the division of numerator math.MaxInt64 and denominator of 1956 and storing it in rm. In the next line, we are returning \*Rat set to the value of the division of numerator -1956 and denominator of math.MaxInt64, and storing it in rn. At line 19, we are returning \*Rat set to the value of the division of numerator 19 and denominator of 56 and storing it in ro. At line 20, we are returning \*Rat set to the value of the division of numerator 1111 and denominator of 2222 and storing it in rp. In the next line, we are returning \*Rat set to the value of numerator 1 and denominator of 1 and storing it in rq.

Now at **line 22**, we are applying arithmetic functions on them as: rq.Mul(rm, rn).Add(rq, ro).Mul(rq, rp). First, the values of rm and rn will be multiplied and stored in rq. Then the value of ro and updated value of rq will be added together, and the result will be stored again in rq. Now the updated value of rq will be multiplied with the value of rp, and the final result will again be stored in rq. In the last line, we are printing rq to check the final value.

That's it about the big package and its working. In the next lesson, you'll see how you can make a package of your own.