Optimizing Programs

This lesson focuses on how to optimize a program.

WE'LL COVER THE FOLLOWING

- Timing a function
- Using memoization for performance

Timing a function

Sometimes it is interesting to know how long a certain computation took, e.g. for comparing and benchmarking calculations. A simple way to do this is to record the *start-time* before the calculation, and the *end-time* after the calculation by using the function `Now()` from the `time` package. The difference between them can then be calculated with `Sub()`. In code, this goes like:

```
package main
import "fmt"
import "time"

func Calculation(){
    for i := 0; i<10000; i++{
        //do something
    }
}
func main(){
    start := time.Now()
    Calculation()
    end := time.Now()
    delta := end.Sub(start)
    fmt.Printf("Calculation took this amount of time: %s\n", delta)
}
```

Timing a function

If you have optimized a piece of code, always time the former version and the optimized version to see that there is a significant (enough) advantage.

Using memoization for performance

When doing heavy computations, one thing that can be done to increase performance is not to repeat any calculation that has already been done and that must be reused. Instead, cache the calculated value in memory; this is called **memoization**. A great example of this is the Fibonacci program. To calculate the n-th Fibonacci number, you need the 2 preceding ones, which normally have already been calculated. If you do not stock the preceding results, every higher Fibonacci number results in an even greater avalanche of recalculations.

```go
package main
import "fmt"

func main() {

    // as the fabonacci for 0 and 1 is 1 so, setting variables expilcitly as 1
    result := 1
    a := 1
    b := 1

    // calculating fabonacci seq for first 10 numbers
    for i := 0;i <= 10;i++{
        if i>1 {            // no need to add preceding values for first two case
            result = a+b
        }
        fmt.Printf("fibonacci(%d) is: %d\n", i, result)

        // modifying 2 values for next iteration
        a = b       // a should have the fibonacci for i-1th number for next iteration
        b = result  // b should have the fibonacci for ith number for next iteration
    }
}
```

Fibonacci using Memoization

Memoization is useful for relatively expensive functions (not necessarily recursive as in the example) that are called lots of times with the same arguments. It can also only be applied to pure functions; these are functions that always produce the same result with the same arguments, and have no side-effects.

If you write a function keeping in view the timing and memoization, you can optimize the program in a good manner. That’s it about how to optimize programs using timing and memoization approaches. There is a quiz in the next lesson for you to solve.