More On Go

1. Janet Yellen offense stance.

What is a Hash

A hash is a mapping from an input set to a unique number. This is so that we have a high probability of a unique number for each different input. The output will have a fixed length.

There are a number of different kind of hashes.

1. md5 a deprecated hash - it has weaknesses and should not be used.
2. sha1 some weaknesses.
3. sha256
4. sha512
5. sha3

hashes are used to validate passwords. pbkdf and Ashley Madison.

Some hashes are specificly designed to be slow for passwords.

Some are build to be in hardware (sha1, sha256) or to not be in hardware (Keccak used in Etherum).

Bitcoin uses sha256.

Example:

Input	Output
Hi	c01a4cfa25cb895cdd0bb25181ba9c1622e93895a6de6f533a7299f70d6b0cfb
Bitcoin	deb10ca6fd85a5eba792ea8561da390635242f0c37c376f8eb7d7859adbffca9
war-and-peace.txt	6cbcc5ca5e590fb9ace161a5b93e4ecf280d7118104be0d63b686c004cfa70ae

Hashes for our purposes are unidirectional. You can take the same input and get the same output, but you can't derive the input from the output.

```
// HashStrings hash a set of strings and return in hex-strings form
func HashStrings(a ...string) string {
        h := sha256.New()
```

```
        for _, z := range a {
                h.Write([]byte(z))
        }
        return fmt.Sprintf("%x", (h.Sum(nil)))
}
```

One of your first Assignmetns was to build a 'ksum' that works like sha256sum or md5sum and reads a file and calculates the hash for that file.

Once you have a hash you can prove that the original file has not changed.

This is the basis of lots of Blockchain metadata applications.

Blockchain and Mining

What is Mining and How is it implemented.

1. More on Go

Maps do not synchronize automatically. So... Synchronization Primitives:

```
package main

import (
        "fmt"
        "sync"
        "time"
)

// SafeCounter is safe to use concurrently.
type SafeCounter struct {
        v   map[string]int
        mux sync.Mutex
}

// Inc increments the counter for the given key.
func (c *SafeCounter) Inc(key string) {
        c.mux.Lock()
        // Lock so only one goroutine at a time can access the map c.v.
        c.v[key]++
        c.mux.Unlock()
}

// Value returns the current value of the counter for the given key.
func (c *SafeCounter) Value(key string) int {
```

```
        c.mux.Lock()
        // Lock so only one goroutine at a time can access the map c.v.
        defer c.mux.Unlock()
        return c.v[key]
}

func main() {
        c := SafeCounter{v: make(map[string]int)}
        for i := 0; i < 1000; i++ {
                go c.Inc("somekey")
        }

        time.Sleep(time.Second)
        fmt.Println(c.Value("somekey"))
}
```

A Go Core/Panic

First the Code

```
        package main

        import "fmt"

        var mm map[string]int

        func main() {
                fmt.Println("vim-go")
                mm["bob"] = 3
        }
```

Then the bad output.

```
vim-go
panic: assignment to entry in nil map

goroutine 1 [running]:
main.main()
        /Users/philip/go/src/github.com/Univ-Wyo-Education/S21-4010/Lect/04/Lect-04_00.
exit status 2
```

Pseudo Code for Mining (Homework 02)

```
package mine

import "github.com/Univ-Wyo-Education/S21-4010/Assignments/02/block"

// TODO Replace above import with import below (commented out)
/*
import (
        "encoding/hex"
        "fmt"

        "github.com/Univ-Wyo-Education/S21-4010/Assignments/02/block"
        "github.com/Univ-Wyo-Education/S21-4010/Assignments/02/hash"
)
*/

// MineBlock implements a proof of work mining system where the first 4
// digits (2 bytes) of the hex value are 0.  This is set with the difficulty
// parameter.  So it could be more than 4 or less (Test with less).
// Difficulty can be increaesed by requiring more digits to be 0 or by
// requring some other pattern to apear in the resulting hash.
func MineBlock(bk *block.BlockType, difficulty string) {
        // Pseudo-Code:
        //
        // 1. Use an infinite loop to:
        //   1. Serialize the data from the block for hashing, Call
        //     `block.SearilizeForSeal` to do this.
        //   2. Calculate the hash of the data, Call `hash.HashOf` to do this.
        //      This is the slow part.  What would happen if we replaced the
        //      software with a hash calculator on a graphics card where you
        //      could run 4096 - 10240 hahes at once?  What would happen if we
        //      replaced the graphics card with an ASIC - so you had dedicated
        //              hardware to do the hash and you could run 4 billion hashes a
        //      second?
        //   3. Convert the hash (it is []byte) to a hex string.  Use the
        //      `hex.EncodeToString` standard go library function.
        //   4. `fmt.Printf("((Mining)) Hash for Block [%s] nonce [%8d]\r",
        //      theHashAsAString, bk.Nonce)` `\r` will overwrite the same line
        //      instead of advancing to the next. (You may want to skip on
        //      some windows systems)
        //   5. See if the first 4 characters of the hash are 0's. - if so we
        //      have met the work criteria.  In go this is
        //     `if theHashAsAString[0:n] == difficulty ("0000" for example) {`.
        //      This is create a slice, 4 long from character 0 with length of 4,
        //      then compare that to the string `difficulty`.
        //    -   Set the block's "Seal" to the hash
        //    -   `fmt.Printf("((Mining)) Hash for Block [%s] nonce [%8d]\n",
        //        theHashAsAString, bk.Nonce)` `\n` will overwrite the same
        //        *and then advance* to the next line.
```

```
        //    -   return
        //   5. Increment the Nonce in the block, and...
        //   6. Back to the top of the loop for another try at finding a seal
        //      for this block.
        //
        // For the genesis block, when I do this it requires 54586 trips through
        // the loop to calculate the proof of work (PoW).
        //

        // TODO: Start coding here.
}
```