Seminario de lenguajes opción GO

1) Realice un programa que haga dos usos con distintos tipos del siguiente tipo genérico:

```
type Map[K comparable, V any] map[K]V
```

Objetivo: tipos genéricos

2) Definir e implementar las operaciones de lista enlazada de la práctica anterior usando el siguiente tipo de datos, de una lista genérica:

```
type List[T any] struct {
   head, tail *element[T]
}

type element[T any] struct {
   next *element[T]
   val  T
}
```

Objetivo: tipos genéricos

3) Ejecute el siguiente programa:

```
package main

import "fmt"

func main() {
   fmt.Println("Inicia Goroutine del main")
   go hello()
   fmt.Println("Termina Goroutine del main")
}

func hello() {
   fmt.Println("Inicia Goroutine de hello")
```

```
    for i := 0; i < 3; i++ {
        fmt.Println(i, " Hello world")
    }
    fmt.Println("Termina Goroutine de hello")
}
```

a) ¿Cuántas veces se imprime Hello world?
b) ¿Cuántas Goroutines tiene el programa?
c) ¿Cómo cambiaría el programa (con la misma cantidad de Goroutines) para que imprima 3 veces Hello world?
    i) Hágalo usando time.Sleep
    ii) Hágalo usando Channel Synchronization

4) ¿Cómo podría hacer para garantizar que el siguiente programa imprima?

PING
PONG
PING
PONG
PING
PONG
PING
PONG
PING
PONG

```
package main

import (
    "fmt"
)

func pxng(m chan string, str string) {
    m <- str
}

func main() {
```

```
    messages := make(chan string)

    for i := 0; i < 5; i++ {
        go pxng(messages, "PING")
        go pxng(messages, "PONG")
    }
    for i := 0; i < 10; i++ {
        fmt.Println(<-messages)
    }
}
```

5) Realice un programa que tenga 2 productores y 2 consumidores. Cada productor y consumidor debe ser una Goroutine. Cada productor producirá 3 números y cada consumidor consumirá 3 números. Los productores deben esperar un tiempo aleatorio entre 0 y 1 segundo para producir un número aleatorio entre 0 y 100. Los consumidores deben consumirlos inmediatamente e imprimirlos por pantalla indicando cual es el consumidor que lo consumió.

   Objetivo: WaitGroups

6) Realice un programa que utilice select para recibir valores desde tres canales diferentes. Cada canal debe recibir una secuencia de números enteros. El programa debe recibir un valor de cada canal y mostrar el valor recibido. Debes usar select para determinar cuál canal tiene un valor disponible y recibir ese valor. El programa debe continuar hasta haber recibido todos los valores enviados a cada canal. Al final debe mostrar el total de valores recibidos desde cada canal.

   Objetivo: select

7) Realice un programa que envíe datos a dos canales desde dos goroutines y estos sean recibidos en el programa principal por un select. Los datos se deben recibir en uno de los canales por el lapso de 5 segundos y por el otro en el lapso de 10 segundos.

Objetivo: timeouts

8) Cree un programa que maneje una lista de contactos de manera concurrente. La lista de contactos debe permitir agregar, eliminar y buscar contactos de manera segura desde múltiples goroutines.

    a) Defina una estructura `Contact` que contenga campos como `Nombre`, `Apellido`, `CorreoElectronico`, y `Telefono`.
    b) Cree una estructura llamada `Agenda` que contenga un mapa de `Contact` con el correo electrónico como clave.
    c) Implemente los siguientes métodos para la estructura `Agenda`:
        i. `AgregarContacto(contacto Contact)`: Agrega un nuevo contacto a la agenda.
        ii. `EliminarContacto(correo string)`: Elimina un contacto de la agenda dado su correo electrónico.
        iii. `BuscarContacto(correo string) Contact`: Busca y devuelve un contacto dado su correo electrónico.
    d) Asegúrese de que las operaciones de agregar, eliminar y buscar contactos se realicen de manera concurrente y que la estructura `Agenda` sea segura para ser accedida desde múltiples goroutines.
    e) Cree una función `main()` que cree una agenda, inicie varias goroutines para agregar, eliminar y buscar contactos de manera simultánea, y luego imprima el contenido de la agenda después de un tiempo para verificar que las operaciones se hayan realizado correctamente.

Ejercicios obligatorios

1) Realice un programa que acepte un número entero positivo N como entrada desde la línea de comandos y calcule todos los números primos menores o iguales a N.
    a) Realice el programa con una única goroutine que muestre en pantalla la lista de números primos encontrados.
    b) Realice el programa utilizando más de una goroutine para dividir el trabajo de comprobación de primos entre múltiples goroutines en paralelo
        i) Cada goroutine debe recibir un rango de números a comprobar y devolver una lista de los números primos encontrados

ii) El programa principal debe recibir el número N y dividir el trabajo en goroutines, asignando a cada una un rango de números a comprobar
iii) Una vez que todas las goroutines hayan finalizado, el programa principal debe recopilar los resultados y mostrar en pantalla la lista de números primos encontrados.

c) Realice la ejecución con N igual a 1.000, 100.000 y 1.000.000 tanto del programa a) como del b). Para cada caso calcule el *speed-up* con la siguiente fórmula:

$$S(p) = T(1) / T(p)$$

donde, S(p) es el *speed-up*, T(1) es el tiempo que tarda la ejecución con una única goroutine y T(p) es el tiempo de ejecución con p goroutines.

2) Realice un programa que simule la atención de clientes en las cajas de un supermercado. La atención de cada cliente por parte del cajero se debe simular con un timer entre 0 y 1 segundo.
   a) Realice el programa haciendo esperar a los clientes en un única cola global y luego enviándolo a la caja para su atención cuando esta se encuentre disponible
   b) Realice el programa haciendo esperar a los clientes en colas individuales por caja asignando la caja para su atención con un algoritmo *round-robin*
   c) Realice el programa haciendo esperar a los clientes en colas individuales por caja asignando la caja para su atención a aquella que tenga la cola más corta
   d) Imprima los tiempos de ejecución de cada uno de los programas implementados en a), b) y c)

3) Desarrolla un programa que implemente un sistema de planificación (scheduler) utilizando 5 goroutines (el main y 4 más) . El programa debe generar una serie de números enteros aleatorios, cada uno con una prioridad aleatoria entre 0 y 3 (donde 0 es la prioridad más alta y 3 la más baja).

   El scheduler debe seguir las siguientes reglas:

   a) El scheduler debe procesar los datos en orden de prioridad, comenzando por los de prioridad 0, luego 1, 2 y 3.
   b) Mientras haya datos de prioridad 0, el scheduler debe procesarlos exclusivamente.
   c) Si no hay datos de prioridad 0 y hay goroutines disponibles, el scheduler puede asignarles datos de menor prioridad para su procesamiento.

d) Una vez que no haya datos de prioridad 0, el scheduler debe pasar a procesar los datos de prioridad 1, y así sucesivamente, utilizando las goroutines disponibles de forma dinámica.
e) El programa principal debe generar los datos numéricos aleatorios con sus respectivas prioridades aleatorias y distribuir el trabajo a las goroutines disponibles manteniendo el orden en el que llegan los datos por prioridad.
f) Con los datos se debe:
    i) Prioridad 0: Sumar los dígitos del número y almacenar el resultado en un archivo llamado "prioridad0.txt" en formato de par ordenado (0, resultado).
    ii) Prioridad 1: Invertir los dígitos del número y almacenar el resultado en un archivo llamado "prioridad1.txt" en formato de par ordenado (1, resultado).
    iii) Prioridad 2: Multiplicar el número por un valor constante (por ejemplo, 10) e imprimir el resultado en la consola.
    iv) Prioridad 3: Acumular los números y mostrar el valor acumulado en la consola cada vez que se procesa un dato.

Tip: Puedes utilizar la librería `math/rand` para generar números aleatorios.

Este último ejercicio es el que deben entregar.