Université d'Ottawa
Faculté de génie

École de science
d'informatique
et de génie électrique

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Electrical
Engineering
and Computer Science

CSI2520 Paradigmes de programmation

Hiver 2019

Devoir 1 (6%)

Date de remise: le 8 février avant 23:00 sur le campus virtuel

Question 1. Structures, Méthodes and Interfaces **[4 points]**

On vous demande de créer une petite simulation impliquant une compagnie basée à Ottawa livrant ses produit à Montréal et à Toronto.

1. Créer la structure `struct Trip` avec les attributs suivants :
   - `string` contenant le nom de la `destination` pour la livraison,
   - `float32` pour le poids `weight` de l'item à livrer,
   - `int` pour le temps requis de livraison `deadline` spécifié en nombre d'heures.
2. Créer les structures `Truck, Pickup` et `TrainCar` avec les attributs suivants:
   - `string` pour le type de `vehicle` (Truck, Pickup and TrainCar),
   - `string` pour le nom du véhicule,
   - `string` pour la `destination` (""),
   - `float32` pour la vitesse moyenne de déplacement `speed` (40, 60 ou 30),
   - `float32` pour la capacité en nombre d'items `capacity` (10, 2 ou 30),
   - `float32` pour la charge du véhicule `load` (0),
   - `Pickup` aura aussi un attribut `bool` appelé `isPrivate` (true) et `TrainCar` aura un attribut `string` appelé `railway` (CNR).
   - Vous devez utiliser les types embarqués afin de minimiser la duplication de code.
   - Définir les fonctions suivantes :

     `NewTruck, NewPickUp` and `NewTrainCar` retournant une de ces structures correctement initialisée.
3. Créer une interface `Transporter` avec les méthodes suivantes:
   - `addLoad` prenant un `Trip` en argument et retournant une `error` si le transporteur dépasse sa capacité, ou a une destination différente, ou ne peut livrer en temps. Si la destination courante du transporteur est vide, alors celle-ci doit être mise à jour avec la destination prévue pour l'item ajouté.
   - `print` afiichant le transporteur à la console

4. Définir les fonctions suivantes:
   - `NewTorontoTrip` avec les arguments poids et temps de livraison requis et retournant un pointeur à un `Trip` avec comme destination `"Toronto"`
   - `NewMontrealTrip` avec les arguments poids et temps de livraison requis et retournant un pointeur à un `Trip` avec comme destination `"Montreal"`
5. Définir les méthodes de l'interface `Transporter` avec comme récepteur un pointeur à un `Truck, PickUp` ou un `TrainCar`
6. Définir la fonction main créant 2 `Truck,` 3 `Pickup` et 1 `TrainCar.` Dans une boucle, demander à l'utilisateur de créer un `Trip` avec son délai de livraison et son poids. Attribuer le `Trip` au premier `Vehicle` dans la liste pouvant effectuer cette livraison, un transporteur pouvant livrer plusieurs items.

Exemple:

```
Destination: (t)oronto, (m)ontreal, else exit? Tor
Weight: 8
Deadline (in hours): 12
Destination: (t)oronto, (m)ontreal, else exit? mo
Weight: 8
Deadline (in hours): 20
Error: Other destination
Destination: (t)oronto, (m)ontreal, else exit? M
Weight: 8
Deadline (in hours): 12
Error: Other destination
Error: Out of capacity
Error: Out of capacity
Error: Out of capacity
Error: Out of capacity
Destination: (t)oronto, (m)ontreal, else exit? q
Not going to TO or Montreal, bye!
Trips: [{Toronto 8 12} {Montreal 8 20} {Montreal 8 12}]
Vehicles:
Truck A to Toronto with 8.000000 tons
Truck B to Montreal with 8.000000 tons
Pickup A to with 0.000000 tons (Private: true)
Pickup B to with 0.000000 tons (Private: true)
Pickup C to with 0.000000 tons (Private: true)
TrainCar A to Montreal with 8.000000 tons (CNR)
```

Question 2. Concurrence et communication **[3 points]**

Le programme à concevoir utilise un serveur appelé `ComputeServer`. Ce serveur reçoit des tâches et ces tâches sont résolues en effectuant un calcul. Les résultats obtenus sont ensuite affichés à la console en utilisant un `DisplayServer`.

Les communications se font par channels. Les tâches sont envoyées au channel retourné par `ComputeServer` et les résultats sont envoyés au channel de `DisplayServer` pour affichage.

Le programme doit donc procéder ainsi.

Les deux serveurs sont créés et chacun d'eux retourne un channel. Les tâches sont créées en demandant à un utilisateur de spécifier 2 nombres.

```
type Task struct {
     a, b float32  // les 2 nombres à additionner
     disp chan float32 // le channel via lequel l'affichage se fait
}
```

Une fois la tâche saisie, elle est envoyée à `ComputeServer` via son channel.

`ComputerServer` résout les tâches en créant des goroutine appelée `handleRequest`. Mais il ne peut s'exécuter qu'un maximum de 3 goroutines à la fois. Il faut donc utiliser un sémaphore ayant une capacité de 3 afin de contrôler la création des goroutines.

```
const (
     NumRoutines = 3
     NumRequests = 1000
)

// global semaphore monitoring the number of routines
var semRout = make(chan int, NumRoutines)
```

La fonction `handleRequest` appelle simplement la fonction `solve` afin de résoudre la tâche et la fonction `solve` calcule le résultat et l'envoie dans le `channel` spécifié dans la tâche.

Un autre sémaphore permet de s'assurer que les résultats sont affichés un à la fois et qu'il ne s'affiche pas pendant que l'utilisateur est en train d'entrer une nouvelle tâche.

`DisplayServer` affiche simplement à la console tous les résultats qu'il reçoit dans son channel.

```
// global semaphore monitoring console
var semDisp = make(chan int, 1)
```

Deux WaitGroup sont utilisés afin de s'assurer que le programme ne s'arrête pas avant d'avoir complété toutes les tâches et d'avoir affiché tous les résultats.

```
// Waitgroups to ensure that main does not exit until all done
var wgRout sync.WaitGroup
var wgDisp sync.WaitGroup
```

Voici les fonctions à réaliser:
`func solve(t *Task)` cette fonction dort pendant une durée de temps aléatoire variant entre 1 et 15 seconds, puis additionne les nombres `a` et `b` et envoie le résultat au channel spécifié dans la tâche (i.e. celui de `DisplayServer`).
`func handleReq(t *Task)` cette fonction est simplement un intermédiaire entre `ComputeServer` et `solve`. C'est elle qui signale au `WaitGroup wgRout` que la tâche a été complétée.
`func ComputeServer()(chan *Task)` la fonction qui créé le channel pour les tâches et qui se met à l'écoute de ce channel. Elle appelle les gorountines `handleReq` (pas plus de 3 à la fois) permettant de résoudre ces tâches.
`func DisplayServer() (chan float32)` la fonction qui créé le channel d'affichage et se met à l'écoute de ce channel. Chaque fois qu'un résultat est reçu, il est affiché à la console.

Voici le squelette du programme à concevoir :

```
func main() {
     dispChan := DisplayServer()
     reqChan := ComputeServer()
     for {
          var a, b float32
          // use semDisp to make sure a result will
          // not be displayed while a user is interacting
          fmt.Print("Enter two numbers: ")
          fmt.Scanf("%f %f \n", &a, &b)
          fmt.Printf("%f %f \n", a, b)
          if a == 0 && b == 0 {
               break
          }
          // Create task and send to ComputeServer
          // …
          time.Sleep( 1e9 )
     }
     // Don't exit until all is done
}
```

Et voici un exemple de sortie produit:

```
Enter two numbers: 2.4 3
2.400000 3.000000
Enter two numbers: 8.0 1.5
8.000000 1.500000
-------
Result: 5.400000
-------
Enter two numbers: 0 0
0.000000 0.000000
-------
Result: 9.500000
-------
```

Question 3. Traitement concurrent **[3 points]**

Pour cette question, on vous demande de traiter un tableau de Triangles. Ces triangles sont représentés par 3 points dans le plan Euclidien.

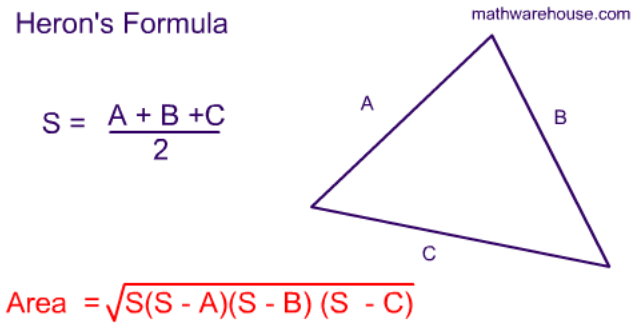
```
type Point struct {
     x float64
     y float64
}

type Triangle struct {
    A Point
    B Point
    C Point
}
```

Ces triangles se trouvent dans un tableau. Pour les fins de cet exercice, vous devez utiliser la fonction test suivante afin de générer ce tableau.

```
func triangles10000() ([10000]Triangle) {

    var tableau [10000]Triangle
    rand.Seed(2120)

    for i := 0; i < 10000; i++ {
        tableau[i].A= Point{rand.Float64()*100.,rand.Float64()*100.}
        tableau[i].B= Point{rand.Float64()*100.,rand.Float64()*100.}
        tableau[i].C= Point{rand.Float64()*100.,rand.Float64()*100.}
    }
    return tableau
}
```

Créer deux méthodes afin de calculer l'aire et le périmètre d'un triangle (utiliser la formule de Heron pour l'aire):

```
func (t Triangle) Perimeter() float64

func (t Triangle) Area() float64
```



On vous demande de créer une fonction Go permettant de trier un Slice de Triangles selon leur ratio périmètre / aire. Pour ce faire vous devez fournir deux piles à cette fonction. La première pile contiendra les triangles dont le ratio est supérieur à 1.0 et la seconde ceux dont le ratio est inférieur ou égale à 1.0

```
func classifyTriangle(highRatio *Stack, lowRatio *Stack,
                      ratioThreshold float64, triangles []Triangle,
// plus some channels or other mecanisms for synchronization?
```

Afin de rendre ce triage plus efficace, on vous demande de subdiviser le tableau de triangles en 10 paquets de 1000. Vous allez alors appeler 10 fois la fonction `classifyTriangles` afin d'effectuer un classement concurrent.

Vous devez aussi créer le type `Stack` qui devra être protégé contre les accès simultanés.

Une fois le traitement terminé, votre fonction main doit afficher le nombre d'éléments dans chacune des deux piles et aussi montrer l'élément sur le dessus de chaque pile.