

Benjamin Kataliko Viranga

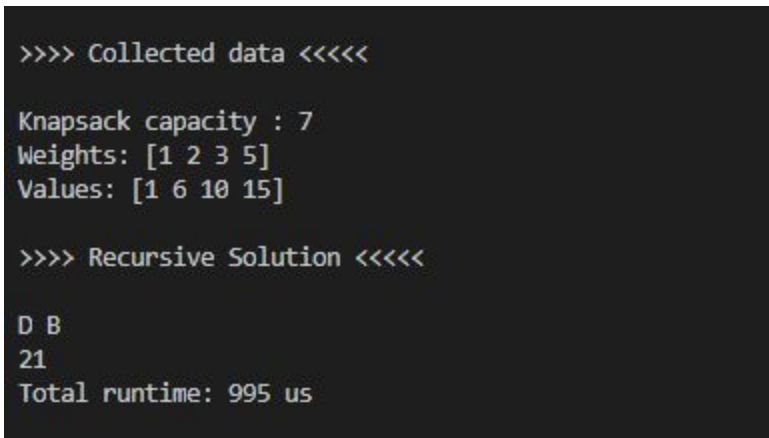
CSI2520 - 8842942

Projet Intégrateur - Partie Concurrente - GO

Explication de l'approche dans la conception de la solution optimal

- Commande exécutée : `go run KnapsackBruteForce.go p1.txt`
- Sorties du programme :

Deux solutions sont affichées sur le terminal : *"Recursive Solution"* et *"Concurrent Solution"*. La *"Recursive Solution"* correspond à la fonction récursive **KnapSackRec()** qui n'initialise aucune go routine pour trouver la solution optimale. Sa solution optimale ainsi que le temps d'exécution est montrée par l'image ci-dessous:



```
>>>> Collected data <<<<<

Knapsack capacity : 7
Weights: [1 2 3 5]
Values: [1 6 10 15]

>>>> Recursive Solution <<<<<

D B
21
Total runtime: 995 us
```

Figure : Collected data and recursive solution (runtime)

Quant à la solution concurrente exécutée par la fonction **KnapSackOptimal()**, son temps d'exécution varie selon le **seuil** assigné à la fonction concurrente. Le seuil (nombre d'items) est utilisé pour déterminer à partir de quel moment la solution concurrente n'est pas nécessaire pour analyser la quantité d'items proposées. Ainsi, lorsque le nombre d'items à analyser est inférieur ou égal au **seuil** correspondant, la **solution récursive** (ne générant aucune autre go routine) est appliquée pour cette quantité d'items tel que l'indique la portion de code ci-dessous:

```
if (len(wt) <= seuil){
    included := ""
    optimal_val_rec,repr_rec :=KnapSackRec(W,wt,val,included, availableItems)
    result <- optimal_val_rec
    result_characters <- repr_rec
    return // terminate the go routine}
```

Dans le programme principal, les valeurs de seuil testées sont :

```
// valeur du seuil pour l'execution en mode concurrent
seuils := []int{len(weights),len(weights)/2, len(weights)/3}
```

Considérer **len(weights)/4** dans le tableau de seuil pousserait le programme à bloquer à cause de la quantité de fils concurrents créés (dans le cas où l'input file contient plusieurs valeurs d'items comme dans le fichier 'autre_fichier.txt').

Ainsi, la sortie de la section concurrente est la suivante :

```
>>>> Concurrent Solution <<<<<

Valeur du seuil : 4
D B
21
Total runtime: 13965 us
Total number of created goroutines: 1
---

Valeur du seuil : 2
D B
21
Total runtime: 998 us
Total number of created goroutines: 6
---

Valeur du seuil : 1
D B
21
Total runtime: 986 us
Total number of created goroutines: 12
---

> Valeurs de seuils initialisées : [4 2 1]
> Durées d'exécution des routines collectées : [13.9654ms 998.2µs 986.5µs]
> Tableau des valeurs des goroutines créées pour chaque seuil: [1 6 12]
```

Figure : Section concurrente

Pour chaque valeur de seuil, le programme affiche le nombre de **goroutines créées**, la **solution optimale pour le Knapsack** ainsi que le **temps d'exécution** pris par la fonction **KnapSackOptimal()** pour chacune de ces valeurs. Par conséquent, le choix optimal est donné par l'image ci-dessous:

```

Valeur du seuil : 1
D B
21
Total runtime: 986 us
Total number of created goroutines: 12
---

> Valeurs de seuils initialisées : [4 2 1]
> Durées d'exécution des routines collectées : [13.9654ms 998.2µs 986.5µs]
> Tableau des valeurs des goroutines créées pour chaque seuil: [1 6 12]

> Le temps optimal correspond au seuil 1 avec une duree de 986.5µs et 12 goroutines créé(es).

> Initialisation de la solution optimal dans le fichier .sol correspondant <

>> File 'p1.sol' Updated Successfully.

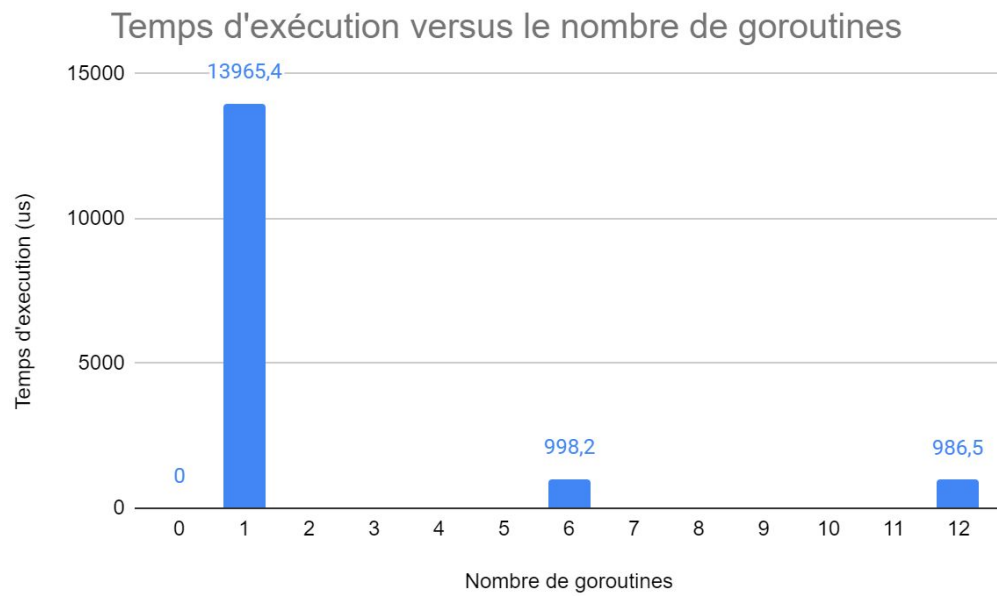
```

Figure: Conclusion sur le choix optimal

Ainsi, pour le fichier '**p1.txt**' correspondant à l'exécution représentée par la figure ci-haut, il faudrait une valeur de seuil de **1 item** et **12 goroutines** pour obtenir la solution optimale la plus rapide avec **986.5 us**.

À chaque nouvelle exécution, le programme propose la solution optimale en fonction du temps d'exécution mesuré.

Le graphique ci-dessous est un diagramme à colonnes pour les données de l'exécution précédente :



On peut noter qu'utiliser 1 seule go routine pousserait le programme à 13.9654 ms d'exécution au lieu de 986.5us d'exécution avec 12 go routines.

Ainsi, le nombre de goroutine ainsi que la valeur du seuil approprié améliore donc la vitesse d'exécution du programme.