NOHAR



Zeste de savoir

# L'Advent of Code 2020 en Go, jours 6 à 10

10 décembre 2020

# Table des matières



Ho, ho, ho!

Ce billet est la suite de mes aventures pendant l'Advent of Code 2020 .

## 1. Jour 6 : On n'a pas besoin de sets en Go...

Le sixième jour , nous étions dimanche, et je me suis permis de faire la grasse matinée.

### 1.1. Les sujets qui fâchent dès le mat... dès le midi

En me connectant à Discord, j'ai vu les autres participants parler d'opérations ensemblistes, et je me suis dit, "pas de bol, je vais encore devoir parler d'un truc qui fâche dans le billet !". Ce point de frottement, c'est l'absence de type `set` en Go.

Il y a deux raisons qui expliquent cette absence:

— La première, c'est qu'un `set` n'est jamais qu'un `map` dont on se moque des valeurs. Comme on dit, "qui peut le plus, peut le moins", et donc cette absence n'est *pas réellement handicapante* car il suffit bien souvent d'utiliser un `map` pour obtenir le comportement d'un ensemble sur ses clés.

— La seconde nous amène dans un débat plus large: on pourrait parfaitement imaginer implémenter nous-mêmes (ou ajouter dans la bibliothèque standard) un type `Set` tout à fait idiomatique **si Go proposait un mécanisme de généricité**.

Je ne souhaite pas spécialement entrer dans le débat sur la généricité en Go, car ce serait lui donner *bien plus d'importance* qu'il n'en a à mes yeux. Je me bornerai à préciser que:

— Le polymorphisme paramétrique va arriver un beau jour en Go, comme en témoigne ce *draft* (mis à jour le 25 novembre) qui est actuellement en cours d'expérimentation par la communauté. D'ailleurs, ce beau jour est *très bientôt*, car on peut déjà jouer avec cette fonctionnalité dans une version dédiée du go playground . Vous pourrez trouver un article détaillé et plus abordable que le *draft* ici . Bref, si Go n'a pas encore de généricité, c'est simplement que ses mainteneurs sont *extrêmement prudents* sur l'ajout de cette fonctionnalité, qui ne doit pas se faire à n'importe quel prix: la solution qui sera finalement intégrée ne doit représenter ni un surcroît de complexité dans le langage, ni un coût en temps ou en complexité à la compilation.

— Plus terre à terre: la philosophie de Go est extrêmement proche de celle de *The Pragmatic Programmer* . À ce titre, dans l'écosystème des Gophers, on trouve énormément d'outils qui reposent sur de la *génération automatique de code* (prenez par exemple gRPC et Protobuf, mais on pourrait aussi parler du SDK de Kubernetes…). Et ça, non seulement c'est rendu possible et facile grâce à la simplicité du langage cible (Go), mais dans les deux premiers exemples que j'ai cités, cela permet en plus de créer des passerelles de compatibilité avec tous les langages majeurs. Autrement dit, le fait que Go manque de *generics* n'est pas réellement handicapant sur la plupart des projets sérieux car on a des moyens très pratiques de s'en accommoder, même si leur introduction sera la bienvenue, notamment dans les nombreux cas où il est *overkill* de dégainer une étape supplémentaire de génération de code à la compilation.

## 1.2. Bon, on regarde l'exo, maintenant ?

Allez, fermons cette parenthèse. Après tout, on est là pour s'amuser et apprendre des trucs!

L'exo du jour présente une certaine familiarité avec celui du 4e jour dans la structure de ses entrées, et donc également dans la structure du programme qui va servir à le résoudre.

En effet, les entrées sont des blocs d'informations séparés par des lignes vides, comme ceci:

```
abc

a
b
c

ab
ac

a
a
```

```
a
a

b
```

Dans l'histoire associée, chaque ligne correspond à l'ensemble des questions d'un formulaire de douanes (numérotées de `a` à `z`) auxquelles les passagers de l'avion ont repondu "oui". Nous avons une ligne par personne, et les personnes sont regroupées (par famille, j'imagine).

L'exercice nous demande de calculer pour chaque groupe:

1. le nombre de questions auxquelles au moins un membre du groupe a répondu "oui".
2. le nombre de questions auxquelles tous les membres du groupe ont répondu "oui".

Du coup, effectivement, il s'agit d'unions et d'intersections ensemblistes, et la façon la plus intuitive d'implémenter la solution est effectivement d'utiliser un `map`.

```
abc

a
b
c

ab
ac

a
a
a
a

b
```

Ce type `Answers` va modéliser *les réponses d'un groupe*. À chaque "question" (de `a` à `z`), elle va associer le nombre de fois que quelqu'un du groupe a répondu "oui".

Vu que l'entrée semble vouloir nous faire compulser des paquets de données séparées par des lignes vides, on va reproduire le même *pattern* d'itération que dans l'exo du jour 4, avec une fonction `iterAnswers` qui appellera une fonction de callback chaque fois que les réponses d'un groupe sont prêtes. Notez que l'on passe également au callback une variable `size` indiquant la taille du groupe.

```
abc

a
b
```

```
c

ab
ac

a
a
a
a

b
```

Il ne nous reste plus qu'à compter:

1. Le nombre de clés de chaque groupe de réponses,
2. Le nombre de questions pour lesquelles on a autant de réponses positives que de personnes dans le groupe.

```
abc

a
b
c

ab
ac

a
a
a
a

b
```

Et voilà, problème résolu!

## 1.3. Mais… est-ce qu'on a vraiment besoin de tout ça ?

Vous avez sûrement remarqué que je n'ai pas résolu le problème directement dans la fonction `main`. En fait, c'est parce que j'ai envie de vous montrer *une autre solution* pour ce problème qui ne fait intervenir ni `map`, ni closure, ni rien d'aussi haut niveau que ça. 🍊

Remarquez que les questions sont numérotées de `a` à `z`: cela nous fait 26 questions en tout. Il existe un type de données tout bête, que vous connaissez forcément depuis le premier jour que vous programmez, et qui permet de calculer des opérations ensemblistes (union et intersection) de façon triviale avec des opérateurs que n'importe quel microprocesseur est capable d'exécuter

en une seule opération. Je veux bien sûr parler des **nombres entiers** et des **opérations bit à bit**.

Si l'on décide que les réponses d'une personne sont modélisées par les 26 premiers bits d'un entier non signé sur 32 bits, on peut calculer l'intersection entre deux ensembles de réponses avec le "et" bit-à-bit (`&`), et l'union avec le "ou" bit-à-bit (`|`). Il ne reste plus ensuite qu'à compter le nombre de `1` dans l'union et dans l'intersection pour répondre à l'exo (et on a d'ailleurs une fonction pour ça dans la bibliothèque standard).

Voici comment j'ai implémenté cette solution:

```
abc

a
b
c

ab
ac

a
a
a
a

b
```

Le code est beaucoup plus KISS, plus compact, et certes un peu moins intuitif à lire. Mais qu'en est-il de son efficacité?

## 1.4. Benchons!

Pour comparer ces solutions, nous allons les exécuter sur l'entrée complète du problème. Pour ce faire, il va falloir trouver un moyen de leur passer notre fichier d'entrée autrement que par la ligne de commande. C'est pourquoi, avec une petite modification , j'ai fait en sorte que nos fonctions d'input aillent chercher le chemin du fichier d'abord dans la variable d'environnement `INPUT_FILE`, et ne cherchent à lire la ligne de commande que si cette variable est vide.

Exécutons maintenant notre bench :

```
$ INPUT_FILE=$(pwd)/day_06/input.txt go test -bench=. -cpu=1
    -benchmem ./day_06
goos: linux
goarch: amd64
pkg: gitlab.com/neuware/aoc-2020/day_06
BenchmarkWithMaps           1470            745774 ns/op
    315918 B/op       2859 allocs/op
```

```
BenchmarkBitwise           36751             32398 ns/op
    0 B/op          0 allocs/op
PASS
ok      gitlab.com/neuware/aoc-2020/day_06      2.703s
```

Ici, j'ai ajouté quelques options à `go test` pour qu'il nous affiche une comparaison non seulement en temps CPU, mais également au niveau de la mémoire allouée, parce que c'est le nerf de la guerre ici.

En résumé:

— La solution intuitive avec les `maps` résout le problème (2178 lignes) en 746µs, en réalisant 2859 allocations dynamiques qui représentent un total de 316Ko de mémoire allouée,
— La solution avec les opérations bit-à-bit résout le même problème en 32µs, sans effectuer la moindre allocation dynamique.

Cette solution est donc 20 fois plus efficace que la première, littéralement.

**Vous voyez bien qu'on a pas besoin de sets en Go!** 🍊

## 2. Jour 7 : Représentation et parcours d'un graphe

Le septième jour , nous sommes tombés sur un problème d'un nouveau genre.

En entrée, nous avions plusieurs centaines de règles de ce style:

```
bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
    bags.
dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
    maroon bags, 1 bright beige bag, 1 drab white bag.
vibrant fuchsia bags contain 4 dark salmon bags.
muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
    4 plaid blue bags.
dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Chaque règle nous dit que «Les sacs de type X doivent contenir M sacs de type Y, N sacs de type Z». Les relations entre les types de sacs forment un graphe acyclique orienté. Un arbre de dépendances, si vous préférez.

Les questions sont:

1. Combien de types de sacs peuvent en contenir un de type `shiny gold`?
2. Si j'ai un sac de type `shiny gold`, combien de sacs devra-t'il contenir en tout?

Vu à quel point je me suis étalé sur l'exercice de la veille, je vais faire court ici. Cet exercice a deux composantes bien distinctes:

— Une intéressante: modéliser les données.

— Une *carrément reloue* car rébarbative et qui n'apporte rien de nouveau: parser les entrées.

Je ne m'attarderai pas sur le parsing des entrées, vous n'aurez qu'à lire ma solution si ça vous intéresse vraiment.

## 2.1. Modéliser les données

Rechercher un algorithme optimal ne m'intéresse pas spécialement ici. Je préfère profiter de cette occasion pour vous montrer comment modéliser le problème.

D'abord, les sacs ont un "type", caractérisés par une couleur, qui est en fait une chaîne de caractères:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Ensuite, chaque type de sac est associé à un certain nombre de règles. Une règle est l'association d'un type de sac, et du nombre de sacs de ce type qu'il faut contenir.


```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```


Enfin, l'arbre de dépendances peut être modélisé par un `map`, qui à chaque `Color` associe un certain nombre de `Rule`, comme ceci:


```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

## 2.2. Parcours récursif

La principale "difficulté" de cet exercice, pour les débutants, est bien sûr que le parcours d'un arbre est un procédé essentiellement récursif. Nous allons implémenter ici deux méthodes du type `RuleGraph`: une pour savoir si un sac d'une couleur donnée peut (transitivement) en contenir un autre, et une pour compter le nombre total de sacs qui doivent être contenus dans un sac d'une couleur donnée.

Voici donc la méthode `HasAncestor`:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

La logique du nom, c'est que si un sac `a` peut contenir un sac `c`, alors `a` **est un ancêtre** de `c` dans l'arbre.

Remarquez la syntaxe de la déclaration. Le graphe est passé par un paramètre spécial, qui *associe* (*bind*) la méthode au type `RuleGraph`. En général, cet argument est très souvent un pointeur, pour que la méthode puisse modifier l'état interne de l'objet auquel elle est associée. Ici, le `RuleGraph` est passé *par valeur*, car il s'agit d'un `map`, qu'un `map` se comporte de toute manière comme une référence sur ses données (donc que le passer par valeur ne représente pas une copie des données), et que de toute façon nous n'avons pas besoin de modifier son état interne.

Autrement, cette méthode est bien sûr récursive: pour savoir si un sac d'une couleur A peut en contenir d'une couleur B, on regarde toutes les couleurs qu'il peut contenir, et si B ne s'y trouve pas, on regarde, récursivement, si ces sacs peuvent contenir un sac de couleur B.

Passons à la deuxième méthode:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Pour savoir combien de sacs un sac de type `A` peut contenir au total, sachant qu'il peut contenir `N` sacs de type `B` :

— On compte les `N` sacs de type `B`,
— Puis on ajoute `N` fois le nombre total de sacs que peut contenir un sac de type `B`.

Avec ces deux méthodes, nous pouvons maintenant répondre à l'énoncé:

```
bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
    bags.
dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
    maroon bags, 1 bright beige bag, 1 drab white bag.
vibrant fuchsia bags contain 4 dark salmon bags.
muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
    4 plaid blue bags.
dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Allez, passons à la suite.

# 3. Jour 8 : Une petite machine virtuelle

Le huitième jour , j'ai laissé échapper une exclamation de pure joie avant même de commencer à coder: «c'est mon exo préféré jusqu'à maintenant!».

Et pour cause, puisqu'il s'agissait d'implémenter une VM minimaliste pour exécuter des petits programmes qui ressemblent à ceci:

```
bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
    bags.
dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
    maroon bags, 1 bright beige bag, 1 drab white bag.
vibrant fuchsia bags contain 4 dark salmon bags.
muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
    4 plaid blue bags.
dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Bien sûr, l'entrée réelle de l'exercice fait plus de 600 lignes…

## 3.1. Modélisation du problème

L'état du programme est un unique nombre entier: un *accumulateur* qui va être modifié tout au long de l'exécution et que j'appellerai ici `RESULT`. Les programmes sont composés de trois types possibles d'instructions. Chaque instruction ("ligne", "opcode"…) comprend:

— Une opération (`nop`, `acc` ou `jmp`),
— Un argument (nombre entier signé compris entre -999 et +999).

Pour exécuter le programme, la VM dispose d'un "pointeur d'instruction" (aussi appelé *program counter*) que j'appellerai `PC`. `PC` commence à 0 (la toute première instruction). Le comportement par défaut de la VM, c'est d'exécuter l'instruction qui se trouve à l'adresse `PC` puis d'incrémenter `PC` de 1 (pour passer à l'instruction suivante).

Les opérations ont le comportement suivant:

- `nop` ignore purement et simplement son argument et ne fait rien,
- `acc <arg>` exécute `RESULT += <arg>`,
- `jmp <arg>` réalise un "saut" dans le programme, c'est-à-dire `PC += <arg>` au lieu de `PC += 1`.

Commencons par modéliser ces éléments de façon à ce qu'ils soient faciles à exécuter en Go.

```
bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
    bags.
dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
    maroon bags, 1 bright beige bag, 1 drab white bag.
vibrant fuchsia bags contain 4 dark salmon bags.
muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
    4 plaid blue bags.
dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

La déclaration des constantes (au moyen de `iota`) est assez typique de Go. Lorsque l'on veut déclarer une énumération constituée d'une série de constate, Go accepte que l'on se contente de donner une valeur qui dépend de `iota` au premier élément, et va déduire les autres constantes automatiquement.

Comme vous pouvez le deviner, un programme sera simplement un `[]Opcode`. Je vous ferai grâce du code qui parse le programme: vous pourrez le retrouver ici dans la fonction load() . Ce n'est rien de bien différent que ce que nous avons vu jusqu'à présent.

## 3.2. Détecter la boucle infinie

*En temps normal*, un programme se termine dès que `PC` atteint la position juste après le dernier opcode. Mais ça, c'est "en temps normal".

Si l'on reprend le programme d'exemple plus haut pour le dérouler mentalement, on s'aperçoit que celui-ci entre dans une boucle infinie. Le premier objectif de l'énoncé est de réussir à exécuter un programme *en l'interrompant immédiatement* dès que l'on détecte une boucle. Autrement dit, il faut quitter l'exécution dès que l'on s'apprête à exécuter **une instruction que l'on a déjà rencontrée auparavant**.

Pour ce faire, j'ai implémenté une fonction `debug` dont voici la signature:

```
bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
    bags.
```

```
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Du reste, la fonction retourne la valeur de l'accumulateur auquel elle est parvenue (`result`), et un booléen (`ok`) pour indiquer si le programme s'est terminé normalement ou non.

Voici son implémentation:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Comme vous le voyez, je maintiens ici une *slice* de booléens que j'ai appelée `footprint`. La logique est simple: il s'agit de laisser une trace derrière soi, au fur et à mesure de l'exécution pour détecter quand nous sommes retombés sur nos propres pas, et ainsi ne pas entrer dans une boucle infinie.

Cette fonction suffit à répondre à la première partie.

## 3.3. Corriger automatiquement le programme

L'énoncé de la seconde partie nous explique que si notre programme d'entrée part en boucle infinie, c'est parce qu'il y a **une** et **une seule** instruction de ce programme qui a été corrompue.

— Soit c'est un `nop` qui s'est transformé en `jmp`,
— Soit c'est un `jmp` qui s'est transformé en `nop`.

Notre mission est bien sûr de trouver laquelle, la corriger, et de retourner le résultat du programme final, quand il se termine avec succès.

La première solution qui pourrait bien sûr fonctionner, est d'y aller par *force brute*, c'est-à-dire d'essayer de corriger les instructions `nop` et `jmp` une par une, jusqu'à tomber sur un programme qui termine. En considérant un programme de longueur N, cela nous donne une complexité en $O(N^2)$. C'est-à-dire que chaque instruction du programme finira par être exécutée au plus N fois. Ça fonctionne, mais ce n'est pas terrible.

Au lieu de cela, j'ai opté pour un algorithme de *backtracking*.

### 3.3.1. Le backtracking

L'idée du *backtracking*, c'est que chaque fois que l'on croise une opération "patchable" `nop` ou `jmp`, on sauvegarde l'état courant de l'exécution (`PC` et `ACC`) sur une pile avant de poursuivre l'exécution.

Lorsque l'on arrive à une situation d'échec:

— on dépile le dernier état sauvegardé (on "*backtracke*"),
— on corrige l'instruction pointée par `PC`,
— on reprend l'exécution avec l'instruction corrigée,
— en cas d'échec, on annule notre modification et on dépile le prochain état de la pile.

Sachant que l'on garde en mémoire les "traces" laissées par l'exécution (`footprint`), on sait que l'on n'exécutera chaque instruction (ou sa correction) qu'une fois maximum: c'est donc un algorithme en O(N).

Voici une petite animation qui montre une exécution du programme d'exemple plus haut, avec l'algorithme de *backtracking*:


### 3.3.2. Exécuter le programme à partir d'un état donné

Avant d'implémenter le *backtracking*, refactorisons un peu notre fonction `run()`, puisque nous allons maintenant avoir besoin de pouvoir démarrer le programme à partir d'un état arbitraire (un *checkpoint*, si vous préférez).

Pour cela, j'ai créé une fonction `eval()`, dont `run()` n'est qu'un cas particulier:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Pas de difficulté particulière ici: ce qui est surtout important à noter, c'est la signature de `eval()`, que nous allons reproduire dans la fonction qui va suivre.

### 3.3.3. Implémenter un backtracking de façon récursive

Dans l'animation que je vous montre plus haut, nous avons une *pile* qui reproduit l'état où l'on souhaite reprendre le programme. En réalité, il n'y a pas besoin d'implémenter nous-mêmes cette pile, car celle-ci est en fait redondante avec la *pile d'appel* des fonctions: lorsqu'une fonction `A()` appelle une fonction `B()`, l'état de la fonction `B()` est créé tout en haut de la *stack*, et lorsque `B()` retourne, son état est dépilé pour laisser la place à l'état de la fonction parente: `A()`. Dans ces conditions, "empiler" un état revient simplement à réaliser un appel récursif.

Voici la fonction `evalBacktrack` qui implémente notre algorithme:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

Il s'agit donc d'éxécuter normalement le programme, jusqu'à tomber sur une instruction "modifiable".

- Dès que l'on tombe sur un `nop` :
  - On appelle `evalBacktrack` récursivement sur l'instruction qui suit le `nop` ;
  - Si cela échoue, on interprète ce `nop` comme un `jmp` et on retourne le résultat de `eval()` sur sa cible.
- À l'inverse, dès que l'on tombe sur un `jmp` :
  - On appelle `evalBacktrack` récursivement sur la cible du `jmp` ;
  - Si cela échoue, on interprète ce `jmp` comme un `nop` et on returne le résultat de `eval()` sur l'instruction suivante.

Et voilà le travail! Il ne reste plus qu'à encapsuler le tout premier appel à `evalBacktrack` dans une jolie fonction `patch`, pour les utilisateurs:

```
1 bright indigo bags contain 4 shiny turquoise bags, 3 wavy yellow
      bags.
2 dotted turquoise bags contain 3 vibrant salmon bags, 2 dotted
      maroon bags, 1 bright beige bag, 1 drab white bag.
3 vibrant fuchsia bags contain 4 dark salmon bags.
4 muted cyan bags contain 2 light gold bags, 5 mirrored salmon bags,
      4 plaid blue bags.
5 dotted tomato bags contain 3 vibrant gold bags, 4 faded blue bags.
```

### 3.4. Et niveau performances ?

Juste pour le plaisir, voici un petit benchmark de la fonction `patch` sur le programme de plus de 600 lignes qui avait été généré pour moi. Pour comparaison, j'y ai aussi fait figurer une solution "par force brute" , dans laquelle je prends notamment soin d'éviter les réallocations intempestives.

```
BenchmarkPatch           1601563               753 ns/op
    704 B/op          1 allocs/op
BenchmarkBruteForce        42866             27934 ns/op
    1408 B/op          2 allocs/op
```

Sur ce programme d'entrée, le backtracking est donc 37 fois plus efficace que de tester toutes les modifications "à la zob". 🍑

## 4. Jour 9 : Récupérons avec un petit exo de boucles

J'ai trouvé l'exo du neuvième jour beaucoup plus banal que celui de la veille.

Toute la "difficulté" résidait dans le fait d'écrire des boucles qui dépendent les unes des autres. Passons rapidement dessus. L'entrée est un grand tableau d'entiers.

### 4.1. Partie 1 : Itérer sur les 25 précédents éléments

Dans cette partie, on nous demandait de trouver le premier nombre qui ne puisse pas s'écrire comme la somme de *deux nombres distincts parmi les 25 qui le précèdent.*

Voici comment j'ai implémenté cette fonction:

```
BenchmarkPatch           1601563               753 ns/op
    704 B/op          1 allocs/op
BenchmarkBruteForce        42866             27934 ns/op
    1408 B/op          2 allocs/op
```

Il y a deux-trois petites choses à dire sur ce code. D'abord, il fallait trouver les itérations, et notamment se débrouiller pour ne jamais avoir `j == k`. C'est pourquoi `k` itère systématiquement de `j+1` à `i`.

Ensuite, la syntaxe que j'ai utilisée peut surprendre. En effet, lorsque nous nous trouvons dans la boucle sur `k` imbriquée, si la condition est vérifiée, c'est à l'itération suivante **de la boucle sur `i`** que nous voulons sauter. Pour ce faire, Go propose un mécanisme emprunté à Perl, qui est d'utiliser une sorte de "`goto` intelligent".

> QUOI ?! UN `GOTO` ?! Mais tu débloques? C'est le MAL absolu!

Ouais, alors euh… non.

On a tous *appris* à un moment donné que `goto` était un mot-clé dangereux et un mécanisme à éviter. Et il y a une bonne raison à cela: son abus peut rendre le code parfaitement illisible.

En revanche, les `goto` de Go sont assez inoffensifs car ils obéissent à des règles plutôt strictes:

— On ne peut pas sauter sur un *label* qui n'est pas défini *dans la même fonction* que le `goto`.
— On peut utiliser un label pour nommer une boucle, comme ici, et préciser *à quelle boucle* appliquer un `break` ou un `continue`

C'est ce que j'ai fait ici. J'ai associé un label `SeachLoop` à la boucle sur `i`, et j'ai précisé dans mon `continue` que je souhaitais sauter à l'itération suivante de `SearchLoop`. On est typiquement dans le genre de cas où ce mécanisme rend le code bien plus clair.

### 4.2. Partie 2 : Trouver les N nombres consécutifs dont la somme est donnée

La seconde partie est elle aussi assez banale:

— il s'agit de trouver les N nombre consécutifs dont la somme est le résultat de la partie 1,
— et de retourner la somme entre le plus petit et le plus grand de ces nombres.

Là encore, l'exercice n'est pas particulièrement difficile, mais il demande par contre d'être à l'aise avec les boucles. Voici ma solution.

```
1 BenchmarkPatch            1601563               753 ns/op
      704 B/op          1 allocs/op
2 BenchmarkBruteForce         42866             27934 ns/op
      1408 B/op          2 allocs/op
```

Et voilà pour aujourd'hui.

## 5. Jour 10 : Un problème de dénombrement

Si l'exo de la VM est mon préféré, celui du 10e jour est définitivement celui qui m'a le plus soûlé jusqu'à présent, et de loin!

On nous donne en entrée une liste d'entiers. Il s'agit des "joltages" de sortie d'adaptateurs que l'on peut brancher les uns sur les autres. Chaque adaptateur peut s'accomoder d'une entrée allant `n-3` à `n-1` jolts, sachant qu'il a une sortie de précisément `n` jolts. En d'autres termes, si l'on trie cette liste, on obtient la propriété suivante: l'écart entre deux nombres successifs est inférieur ou égal à 3.

Dernière info importante: on veut chaîner ces adaptateurs entre une entrée à `0` jolts, et un appareil à `max+3` jolts, où `max` est ici le joltage max dont nous disposons dans la liste.

## 5.1. Partie 1 : Compter les écarts

Le but de la première question est assez simple: il faut trier la liste, et compter, entre la prise et l'appareil, le nombre d'écarts de 1 jolts et de 3 jolts que l'on trouve dans la chaîne ainsi formée.

Pour trier la liste, ne nous prenons pas la tête, la fonction `sort.Ints` de la bibliothèque standard fera parfaitement l'affaire. Une fois cette liste triée, la solution à la partie 1 est relativement directe, il faut juste ne pas oublier l'écart entre la prise d'entrée et notre premier adapteur, ni celui entre le dernier adaptateur et l'appareil. 🍊

```
1 BenchmarkPatch           1601563               753 ns/op
      704 B/op          1 allocs/op
2 BenchmarkBruteForce        42866             27934 ns/op
      1408 B/op          2 allocs/op
```

## 5.2. Partie 2 : Dénombrer tous les arrangements possibles

Le but de la partie 2 est de dénombrer toutes les chaînes d'adaptateurs possibles que l'on peut construire pour alimenter notre appareil à `max+3` jolts en partant d'une prise à `0` jolts.

En effet, chaque adaptateur peut s'accommoder d'une entrée jusqu'à `n-3` jolts. Cela veut dire que nous ne sommes pas obligés de les chaîner tous nos adaptateurs les uns derrière les autres et qu'il existe de très nombreuses façons de les combiner entre eux… Et donc on veut les compter.

C'est *exactement* le genre d'algo que j'ai depuis longtemps pris en grippe. Sérieusement, je **déteste** ça, c'est ma bête noire!

Après avoir passé beaucoup trop de temps à chercher une formule par récurrence qui permettrait de calculer le résultat sur ma liste d'entrée en un temps raisonnable (même jusqu'à une demi-heure ou une heure, s'il n'y avait que ça pour m'en débarrasser !), j'ai fini par trouver l'algorithme linéaire que voici:

— Notre liste est triée par ordre croissant, donc pour un adaptateur `n` jolts, nous devrions déjà connaître le nombre d'arrangements possibles pour construire une chaîne à `n-1`, `n-2` et `n-3` jolts.
— Si on ne dispose pas d'un adaptateur à `n` jolts, le nombre d'arrangements possibles est bien sûr 0.

Dans le cas général, maintenant. Supposons que nous pouvons avoir:

— une entrée à `n-3` jolts au moyen de `x` arrangements,
— une entrée à `n-2` jolts au moyen de `y` arrangements,
— une entrée à `n-1` jolts au moyen de `z` arrangements,

Dans ces conditions, si nous disposons d'un adaptateur à `n` jolts, alors il existe `x+y+z` arrangements pour obtenir une entrée à `n` jolts, puisqu'il suffit de le brancher par-dessus les arrangements précédents.

En gros, ça fait penser à un genre de suite de Fibonacci, mais avec une récurrence triple:

$$U_n = U_{n-1} + U_{n-2} + U_{n-3}$$

***Sauf*** qu'ici, $n$ ne suit pas un incrément de 1: les valeurs successives de $n$ sont les valeurs de notre liste d'entrée triée par ordre croissant. Autrement dit, il va y avoir des trous, jusqu'à deux valeurs manquantes successives.

On peut réaliser ce calcul en maintenant une fenêtre glissante, exactement comme une fonction qui calcule la suite de Fibonacci (`a, b = b, a+b`), mais en faisant simplement attention à bien combler les valeurs avec des 0 pour les joltages qui nous manquent.

```
1 BenchmarkPatch           1601563               753 ns/op
      704 B/op          1 allocs/op
2 BenchmarkBruteForce        42866             27934 ns/op
      1408 B/op          2 allocs/op
```

Voilà, ça, c'est fait. J'ai **beaucoup trop** galéré pour arriver à une solution aussi simple. Espérons que ça sera plus marrant demain.

---

Par rapport au précédent billet, on peut remarquer qu'on est carrément passé à la vitesse supérieure en termes de difficulté. À vrai dire, je ne sais pas encore si je dois me réjouir ou m'inquiéter du fait que nous n'en soyons même pas à la moitié! 🍊