

Queste de savoir

L'Advent of Code 2020 en Go, jours 1 à 5

6 décembre 2020

Table des matières

1.	Se construire quelques outils pour la suite	1
1.1.	Conventions	2
1.2.	Ouvrir le fichier passé en ligne de commande	2
1.3.	Lire un flux d'entrée de façon "bufferisée"	3
2.	Jour 1: Trouver N nombres dont la somme est 2020	4
3.	Jour 2: Valider une politique de mots de passe	4
3.1.	Partie 1, parser correctement les entrées	5
3.2.	Partie 2 : Ça y est, on peut factoriser	6
4.	Jour 3: Parcourir un tableau suivant une pente donnée	7
4.1.	Partie 1: Comprendre le parcours pour une pente donnée	7
4.2.	Partie 2: Généraliser à n'importe quelle pente	8
5.	Jour 4: Valider un JWT... enfin presque	8
5.1.	Modéliser les données et itérer dessus	9
5.2.	Valider le passeport	10
6.	Jour 5: Parser des nombres binaires de façon détournée	11
6.1.	S'aider de tests unitaires!	12
6.2.	Implémentation naïve de l'algorithme	13
6.3.	Et s'il y avait une méthode plus maligne?	13
6.4.	Quelle méthode est la plus performante?	14

Ho, ho, ho! Connaissez-vous l'[Advent of Code](#) ?

Comme son nom l'indique, il s'agit d'un calendrier de l'Avent un petit peu spécial: plutôt que d'ouvrir un chocolat par jour jusqu'à Noël, on résout des énigmes en programmant. Cette année, nous sommes plusieurs sur le serveur Discord de ZdS a avoir décidé d'y participer collectivement, chacun dans son langage de prédilection et selon son niveau en programmation.

En ce qui me concerne, j'y participe bien évidemment en Go, et je me propose de prendre ces exercices comme un prétexte pour illustrer la façon dont on travaille avec. Mon but est donc de résoudre ces exercices non pas *le plus rapidement possible*, ni même de la façon la plus élégante ou improbable, mais simplement de respecter au mieux la philosophie de Go, à savoir de produire le code final le plus banal et lisible possible sans toutefois faire n'importe quoi du point de vue des performances.

Et pour cela, je me propose d'écrire une série de billets en prenant les exercices 5 par 5. 🍊

1. Se construire quelques outils pour la suite

Bon, clairement, le tout premier jour n'est jamais un challenge: il est plutôt là pour s'assurer que l'on sait lire une consigne et faire manger des entrées à notre programme.

1. Se construire quelques outils pour la suite

Et ça tombe bien!

En effet, Go est à la fois un langage qui se veut **simple**¹, et qui n'a pas vocation à écrire des scripts. En ce sens, c'est un peu le contraire de Python qui permet de récupérer les entrées des programmes au moyen d'un générateur en intension. Nous allons donc devoir nous munir de fonctions utilitaires, dès le départ, pour mieux nous concentrer sur les problèmes par la suite. Ainsi, dans cette section, je vais surtout me focaliser sur les entrées-sorties.

1.1. Conventions

Les exercices de l'Advent of Code suivent tous le même principe:

- Le site vous génère un fichier texte de quelques centaines de lignes,
- Il vous pose deux questions, dont la réponse est systématiquement un nombre entier,
- Vous lui soumettez vos réponses en entrant simplement le nombre entier que vous aurez calculé pour votre fichier d'entrée.

Nous allons donc considérer que, chaque jour, nous allons:

- Télécharger un fichier texte,
- Le faire manger à notre programme...,
- ... qui va afficher les deux réponses attendues.

Créons donc un package pour ça, et nommons-le en faisant preuve du moins d'imagination possible: `utils`. Celui-ci proposera deux fonctions:

- `utils.ReadLines()` va ouvrir, lire, et retourner le contenu du fichier sous la forme d'un `[]string`: un élément par ligne.
- `utils.ReadInt()` va faire la même chose dans le cas où le fichier contient un nombre entier par ligne, et retourner le `[]int` correspondant.



Clairement, cette section va surtout aborder des considérations "système" assez courantes, qui n'ont pas grand chose à voir avec la résolution des problèmes algorithmiques de l'AoC. **Si vous débutez complètement en programmation et n'êtes pas plus curieux que ça sur les conventions du langage Go, vous pouvez sauter cette section, et vous rendre immédiatement à l'énoncé du premier jour.**

1.2. Ouvrir le fichier passé en ligne de commande

Nos programmes seront lancés en suivant ce modèle:

```
1 $ nom_du_programme input.txt
```

1. Vous le savez sûrement déjà, mais je le précise une nouvelle fois, juste au cas où: "simple" ne veut pas dire "facile". Typiquement: Go est simple, ce qui le rend plus difficile à utiliser dans certaines situations alors que Python est facile à utiliser, ce qui le rend assez complexe quand on regarde sous le capot.

1. Se construire quelques outils pour la suite

La fonction `mustOpen()` suivante se charge de lire le nom du fichier qui doit être passé en premier argument de la ligne de commande, et retourner ce fichier ouvert en lecture, ou bien quitte le programme avec un message d'erreur si quelque chose n'est pas normal (l'utilisateur n'a pas passé de fichier, ou bien il y a eu une erreur).



Convention de nommage

Vous observerez peut-être que la fonction du package `os` sur laquelle nous reposons s'appelle `Open` et retourne à la fois un fichier et une possible erreur. C'est ce que font la plupart des fonctions en Go.

Lorsque nous souhaitons écrire une fonction qui ne retourne jamais d'erreur, au risque de quitter le programme ou de "paniquer" en cas d'erreur, la façon idiomatique de s'y prendre est de préfixer le nom de la fonction par le terme `must` (ou `Must`).

La bibliothèque standard de Go regorge de fonctions de ce genre, qui viennent en deux versions: une version normale, et une version `Must`.

```
1 $ nom_du_programme input.txt
```

Si Go vous est étranger, vous tiquerez peut-être sur le formatage des noms: la fonction `mustOpen` a un nom en `lowerCamelCase` alors que la fonction `Fatal` est en `CamelCase`.

Il faut savoir que le fait d'utiliser l'une ou l'autre de ces conventions **veut dire quelque chose** en Go. En effet, la fonction `mustOpen` est *privée*, et ne pourra être appelée que depuis l'intérieur du package `utils`, alors que la fonction `Fatal` est *publique*, et pourra être appelée par les programmes qui importent notre package. C'est d'ailleurs la raison pour laquelle j'ai documenté cette dernière, mais pas `mustOpen`.

1.3. Lire un flux d'entrée de façon "bufferisée"

Maintenant que nous avons parsé correctement la ligne de commande et ouvert le fichier. Il est temps de le lire et d'implémenter nos fonctions `ReadLines` et `ReadInts`.

Dans les deux cas, nous avons besoin de lire le contenu du fichier *ligne par ligne*. Nous allons pour cela utiliser un objet de la bibliothèque standard qui s'appelle un `Scanner`, et dont le rôle est de lire un flux en remplissant un *buffer*. Par défaut, ce *buffer* sera rempli ligne par ligne. Dans le cas de la seconde fonction, nous utiliserons également la fonction `strconv.Atoi`² pour convertir des chaînes de caractères en entier.

Notez l'utilisation de `defer`, qui permet de nous assurer que le fichier sera fermé dès que nous sortirons du scope de la fonction. C'est un réflexe très courant en Go:

- J'ouvre un `truc`,
- J'écris immédiatement `defer truc.Close()` pour ne pas l'oublier.

2. Ce nom pourra sembler bizarre aux plus jeunes d'entre nous: sachez qu'il est inspiré de la bibliothèque standard du C, et qu'il signifie *Array To Integer*.

2. Jour 1: Trouver N nombres dont la somme est 2020

```
1 $ nom_du_programme input.txt
```

Ces deux fonctions vont nous faire gagner pas mal de temps pendant les 25 prochains jours. 🍊

2. Jour 1: Trouver N nombres dont la somme est 2020

Comme je le disais plus haut, [l'énigme du jour 1](#) est triviale. Cela revient à trouver les deux (ou trois) nombres dans une liste, dont la somme est égale à 2020, et retourner le produit de ces nombres.

Partant de là, ce problème se résout avec deux ou trois boucles `for` imbriquées, comme ceci:

```
1 $ nom_du_programme input.txt
```

On pourrait pinailler sur ce code en précisant qu'en toute rigueur, on devrait s'assurer que l'on n'essaye pas d'additionner un nombre à lui-même, même si l'ensemble d'entrée de l'exercice semble prendre bien soin d'éviter ce cas (en tout cas, le nombre 1010 était absent de mon *input*)...

Allez, faisons-le, juste histoire de préciser que la boucle `for` de Go, avec cette syntaxe (`for ... range`), va se comporter sur les slices de la même façon que la fonction `enumerate` en Python, c'est-à-dire qu'elle va itérer directement sur des paires (`indice, valeur`). Cela ne nous coûte rien de vérifier que les nombres `x` et `y` ont bien des indices différents dans la liste d'entrée:

```
1 $ nom_du_programme input.txt
```

Voilà, c'est à peu près tout ce qu'il y a à dire sur cet exercice en Go. Vous pouvez trouver ma solution complète [ici](#).

3. Jour 2: Valider une politique de mots de passe

[L'énigme du deuxième jour](#) était bien plus rigolotte que celle de la veille.

On nous confie une liste d'entrées ayant la forme suivante :

```
1 7-13 f: ftmwxpcsfxzqv
```

Ces entrées correspondent à des "mots de passe" en clair, chacun stocké avec une contrainte.

3. Jour 2: Valider une politique de mots de passe

3.1. Partie 1, parser correctement les entrées

Pour la partie 1, on nous explique que la contrainte de l'exemple plus haut doit être lue comme ceci:

Le mot de passe doit contenir entre 7 et 13 fois la lettre `f`.

On nous demande donc de retourner le nombre de mots de passe valides dans notre liste d'entrée. À ce stade de l'exercice, je pense qu'il est important de préciser que **l'on ne sait pas encore ce que l'on nous demandera dans la partie 2**. Ce sera probablement quelque chose lié aux mots de passe, mais là où je voudrais en venir, c'est qu'il est préférable de *ne pas essayer de deviner* la façon dont notre code devra être factorisé pour le moment.

On va se contenter d'écrire une fonction `isValid` qui prend une ligne en argument et retourne `true` si le mot de passe est valide. Cette fonction peut donc se découper en deux parties:

- Extraire les différents paramètres de la ligne,
- Effectuer la vérification elle-même.

Pour extraire les champs de la ligne, la façon la plus simple semble ici d'utiliser une **expression régulière**.

3.1.1. Quelques remarques à propos des expressions régulières de Go

Le moteur d'expressions régulières présent dans la bibliothèque standard de Go est légèrement différent de celui de la plupart des autres langages (Python, Java, Perl, Ruby, etc.). En effet, celui-ci a le bon goût d'utiliser un algorithme *sûr et performant*... D'accord, d'accord, dit comme ça, ça peut passer pour un tacle gratuit, mais laissez-moi vous expliquer et vous verrez que c'est à la fois rigoureusement vrai ET intéressant.

Pour évaluer des expressions régulières, on peut considérer grosso-modo qu'il existe deux méthodes populaires.

- La première est le célèbre moteur **PCRE** [↗](#), qui date de 1997, et qui nous vient des expressions régulières de Perl 5. C'est ce moteur qui est implémenté dans la plupart des langages de programmation;
- La seconde de ces deux méthodes est nommée *l'algorithme de Thompson*, car il a été publié en 1968 par **Ken Thompson** [↗](#) dans un article sobrement intitulé *Regular expression search algorithm* [↗](#). On le retrouve implémenté notamment dans la bibliothèque **re2** [↗](#).

Ce qui différencie ces deux moteurs de regexp, c'est principalement que **PCRE** implémente *plus* d'opérateurs que **re2**. On pourrait se dire que cela rend automatiquement **PCRE** "meilleur", mais en fait, c'est le résultat d'une différence fondamentale de philosophie, et cela se traduit par une différence d'autant plus énorme en termes de simplicité et de stabilité des performances. En effet, les opérateurs que **PCRE** propose "en plus", ce sont des opérateurs complexes comme celui de *lookahead*. Comme l'explique très bien Russ Cox [dans cet article](#) [↗](#), cet opérateur impose d'utiliser un algorithme de *backtracking* pour évaluer les **PCRE**. Le problème, voyez-vous, c'est que l'algorithme d'évaluation par *backtracking* est très instable, y compris sur des expressions pourtant "simples" mais pathologiques.

L'algorithme original de Ken Thompson, quant à lui, s'optimise en construisant un cache qui permet d'approcher les performances de l'algorithme optimal, c'est-à-dire un **AFD** comparable à

3. Jour 2: Valider une politique de mots de passe

celui de l'algorithme d'Aho-Corasick³ pour la recherche de mots-clés dans un texte. Partant de ce constat, le moteur `re2` fait le compromis de lâcher ces opérateurs de *lookahead* et *lookbehind* (qui rendent de toute façon les expressions régulières *encore plus difficiles à lire*), de manière à garder toutes les propriétés intéressantes de l'algo de Thompson.

Quant à Go, eh bien sachant que Ken Thompson et Russ Cox font tous les deux partie de ses créateurs, je vous laisse deviner quel algorithme il implémente. 🍊

3.1.2. Revenons à nos mots de passe

Bref, tout ça pour dire que le moteur d'expressions régulières de Go a un défaut de moins que la plupart des autres: on peut le considérer comme efficace, et dans un cas comme celui qui nous intéresse ici, il simplifie drôlement les choses.

Le format de nos entrées peut être décrit par la regexp suivante (les parenthèses sont "capturantes"):

```
^(\\d+)-(\\d+) ([a-z]): ([a-z]+)$
```

Nous n'avons plus qu'à l'utiliser pour *parser* nos lignes, et les valider dans la foulée:

```
1 7-13 f: ftmwxpcsfxzqv
```

Remarquez que ce code utilise la fonction `strconv.Atoi` que nous avons vu plus haut, ainsi que la fonction `regexp.MustCompile` qui suit la fameuse convention du préfixe `Must` sur les fonctions qui paniquent en cas d'erreur.

En dehors de cela, on peut remarquer que l'on convertit notre `password` en une chaîne d'octets (`[]byte`) pour itérer dessus. En effet, les chaînes de caractère en Go sont par défaut en UTF-8 (encore un truc inventé par Ken Thompson, tiens!), donc lorsque nous itérons dessus, nous itérons sur des caractères Unicode (`rune`) et non des octets. Ici, nous savons que le texte est forcément constitué de lettres minuscules ASCII, donc nous pouvons convertir notre chaîne Unicode en une chaîne d'octets sans craindre de casser l'encodage du mot de passe.

3.2. Partie 2 : Ça y est, on peut factoriser

Sans grande surprise, la partie 2 implémente une politique de validation des mots de passe différente de la première. On va donc devoir factoriser notre code.

Pour cet exercice, il me semble que le plus "sémantique" serait que la fonction `isValid` prenne deux arguments:

- L'entrée,
- Une "politique de validation", qui va se présenter sous la forme d'une fonction.

3. Je vous parlais d'ailleurs de l'algorithme d'Aho-Corasick [dans ce billet](#) ↗ .

4. Jour 3: Parcourir un tableau suivant une pente donnée

```
1 7-13 f: ftmwxpcsfxzqv
```

On pourrait donc découper le code de notre partie 1 comme ceci:

```
1 7-13 f: ftmwxpcsfxzqv
```

Valider un mot de passe se fera donc avec l'appel `isValid(entry, partOne)`. Il ne reste plus qu'à implémenter la fonction `partTwo`, ce qui ne représente aucune difficulté particulière. Vous trouverez la solution [ici](#) .

4. Jour 3: Parcourir un tableau suivant une pente donnée

L'énigme du troisième jour changeait un peu des précédentes. On nous donnait un tableau en deux dimensions avec plusieurs centaines de lignes de 31 caractères ressemblant à ceci:

```
1 .....##.....#...#..#.#.....#....
2 .....#...#...#..#.....##.....
3 #.#...#.....###.#.##..#.....
4 .....#.....##.#..##...##.##..
5 .#...#.#...##.....#.#.....
6 #.....##.....###.#...#.....#
7 .....#.....#.#.....#.#.....
8
9 ... etc., sur plus de 300 lignes ...
```

4.1. Partie 1: Comprendre le parcours pour une pente donnée

L'énoncé est simple: en partant du premier caractère de la première ligne (`input[0][0]`), nous allons nous déplacer jusqu'à la dernière ligne en suivant un mouvement de 3 cases vers la droite, et une case vers le bas. Lorsque l'on arrive au bord d'une ligne, disons à la position 29, le reste du mouvement vers la droite doit être reporté sur le début de la ligne suivante. On se retrouvera donc à la position $(29 + 3) \% 31 = 1$.

Le but de l'exercice est de compter le nombre de caractères # (représentant des sapins) que l'on croise en chemin.

```
1 .....##.....#...#..#.#.....#....
2 .....#...#...#..#.....##.....
3 #.#...#.....###.#.##..#.....
```

5. Jour 4: Valider un JWT... enfin presque

```
4 .....#.....##.#..##...##.##..
5 .#...#.#...##.....#.#.....
6 #.....##.....###.#...#.....#
7 .....#.....#.#.....#.#.....
8
9 ... etc., sur plus de 300 lignes ...
```

Tout l'intérêt de cette partie est d'aboutir à l'utilisation de l'opérateur modulo (%) pour reporter le mouvement sur le début de la ligne suivante.

4.2. Partie 2: Généraliser à n'importe quelle pente

La partie 2 revient à reproduire la même chose avec des pentes différentes, et donc de généraliser la fonction `countTrees` à d'autres mouvements que "1 vers le bas, 3 vers la droite". Vous pourrez trouver la solution [ici](#) .

Allez, ne perdons pas de temps là-dessus et passons à la suite.

5. Jour 4: Valider un JWT... enfin presque

L'exercice du jour 4 a connu un accueil mitigé sur notre serveur Discord.

Nombreux sont ceux qui l'ont trouvé ennuyeux et rébarbatif. En ce qui me concerne, je l'ai bien aimé car c'est celui qui, jusqu'à présent, s'est montré le plus proche d'une problématique courante *de la vraie vie*. 🍌

L'entrée est composée d'un certain nombre de "passeports" séparés par une ligne vide. Voici un exemple avec trois passeports différents:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Ces passeports sont composés d'un certain nombre de champs:

- `cid` : id du pays (ce champ est *optionnel*),
- `pid` : id du passeport,
- `eyr` : année d'expiration du passeport,
- `iyr` : année d'émission du passeport,
- `byr` : année de naissance,

5. Jour 4: Valider un *JWT*... enfin presque

- `hcl` : couleur des cheveux,
- `ecl` : couleur des yeux,
- `hgt` : taille, en centimètres (`cm`) ou en pouces (`in`),

Vu que nous sommes face à quelque chose qui commence à ressembler à *La Vraie Vie*TM, il y a une foule de choses que je pourrais vous montrer avec cet exercice, mais je vais me permettre de faire l'impasse sur la plupart d'entre elles (notamment l'écriture de tests), pour me concentrer sur les remarques les plus pertinentes. 😊

5.1. Modéliser les données et itérer dessus

Le but de la première partie est évidemment de *parser* ces passeports, et de commencer à valider ceux-ci. Comme je l'ai dit dans le titre, cet exercice ressemble furieusement à la validation des *claims* dans un *JWT* [↗](#). La première chose qui semble judicieuse à faire, donc, est de modéliser nos passeport de la même façon que les packages *JWT* de Go modélisent leurs claims, c'est-à-dire via une `map`:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Une `map` en Go, c'est plus ou moins la même chose qu'un dictionnaire en Python. Ici, notre `map` est contrainte à n'accepter que des `string` comme clés et comme valeurs.

Maintenant que le type est défini, voici comment je désire résoudre cette première partie:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Ce code fait intervenir deux fonctions:

- La fonction `iterPassports` va prendre en entrée deux arguments :
 - Les lignes de l'entrée,

5. Jour 4: Valider un *JWT*... enfin presque

- Une fonction qui accepte un passeport et qui ne retourne rien⁴.
- La fonction `isValid` prend un passeport en argument et retourne `true` si le passeport est valide.

Comme vous le constatez, la fonction que l'on passe à `iterPassports` est définie sur place. C'est une fonction *anonyme*. De plus, elle incrémente la variable `count` qui est définie dans le scope de la fonction `main`: il s'agit donc d'une *closure*. Ce faisant, `iterPassports` se comporte d'une façon qui ressemble énormément à une boucle `for` qui servirait à itérer sur les passeports au fur et à mesure que ceux-ci sont parsés.

Voici comment je l'ai implémentée:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Pour parser un passeport, le principe est simple: on isole d'abord les *champs* de la ligne, qui sont séparés par une espace (c'est ce que fait la fonction `strings.Fields`), puis on sépare chaque champ en une clé et une valeur, qui se trouvent de part et d'autre d'un `:`, avant de rajouter ce couple clé/valeur dans le passeport.

Lorsque l'on tombe sur une ligne vide, on appelle notre fonction de *callback* sur le passeport avant de le réinitialiser (pour passer au suivant).

Enfin, le fichier ne se termine pas par une ligne vide, donc on va appeler une toute dernière fois le *callback* avant de retourner.

5.2. Valider le passeport

Pour qu'un passeport soit valide dans la première partie, il suffit qu'il possède tous les champs obligatoires. Sautons directement à la seconde partie, dans laquelle il faut que ces champs suivent également le bon format:

- Les années sont contraintes par un intervalle particulier,
- La hauteur également, et les bornes acceptables diffèrent selon l'unité.
- Les autres champs ont un format bien précis à suivre.


Pour que tout cela reste visible, nous allons définir, pour chaque champ obligatoire, une fonction de validation que l'on appellera un `Validator`:

6. Jour 5: Parser des nombres binaires de façon détournée

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Valider un passeport consiste donc à vérifier que ces champs obligatoires sont présents, et validés par la fonction correspondante:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Je vous laisse découvrir l'implémentation des validateurs dans la solution que vous trouverez [ici](#) . Ceux-ci n'ont rien d'assez *nouveau* pour que ça vaille le coup de vous les détailler.

6. Jour 5: Parser des nombres binaires de façon détournée

Ouais, ce titre est un *spoiler*, mais pas plus que le [titre de l'exercice lui-même](#) .

Pour résumer l'énoncé, on peut dire que celui-ci consiste à faire une recherche dichotomique d'une cellule dans un tableau de 128 lignes et 8 colonnes (les sièges d'un avion). Autrement dit, on va vous fournir un code comme celui-ci:

BFFFBBFRRR

Vous pouvez considérer que ce code va jouer au *plus ou moins* en deux étapes avec vous:

- Les 7 premiers caractères servent à trouver le numéro de rangée de notre siège:
 - **B** nous indique que notre rangée sera entre le numéro 64 et le numéro 127
 - **F** nous indique que notre rangée sera entre le numéro 64 et le numéro 93
 - **F** nous indique que notre rangée sera entre le numéro 64 et le numéro 79
 - **F** nous indique que notre rangée sera entre le numéro 64 et le numéro 71
 - **B** nous indique que notre rangée sera entre le numéro 68 et le numéro 71

4. Bon, pour faire les choses vraiment bien elle devrait retourner une erreur, mais ce n'est pas nécessaire ici: s'il y a une erreur, on appelle **Fatal** et le programme est quitté.

6. Jour 5: Parser des nombres binaires de façon détournée

- B nous indique que notre rangée sera entre le numéro 70 et le numéro 71
- F nous indique que notre rangée est celle de numéro 70
- Les 3 derniers caractères permettent de trouver le siège dans une rangée de 8:
 - R nous indique que notre siège est entre le numéro 4 et le numéro 7
 - R nous indique que notre siège est entre le numéro 6 et le numéro 7
 - R nous indique que notre siège est le numéro 7

Cela se traduit par l'identifiant $70 * 8 + 7 = 567$

6.1. S'aider de tests unitaires !

Nous allons bien évidemment écrire une fonction `seatID` qui prend une `string` en entrée et retourne l'identifiant sous forme d'un `int`. Avant de nous jeter tête baissée dans son implémentation, ayons le nez creux: on est sur le genre de fonction où il est trop facile de faire des erreurs d'*off by one*, alors dotons-nous d'un petit test unitaire pour raccourcir notre boucle de *feedback*.

En Go, pour écrire des tests pour le package `main`, la façon canonique est de créer un fichier `main_test.go` qui va ressembler à ceci:

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Remarquez que j'ai écrit une fonction dont le nom est préfixé par `Test` et qui accepte un `*testing.T` en argument. Cet objet dispose de fonctions `Error`, `Errorf`, `Fatal` et `Fatalf` pour, respectivement:

- Indiquer que le test a échoué sans pour autant l'interrompre,
- Indiquer que le test a échoué, sans l'interrompre, en affichant un message formaté,
- Indiquer que le test a écouté et l'interrompre immédiatement,
- Indiquer que le test a écouté, et interrompre celui-ci en affichant un message formaté.

Par défaut, si le test arrive au bout sans qu'aucune de ces méthodes ne soit appelée, celui-ci est considéré comme valide.

Munis de ce fichier (dans `./day_05/main_test.go`), il suffit de lancer la commande `go test ./day_05 -v` pour tester notre code.

6. Jour 5: Parser des nombres binaires de façon détournée

6.2. Implémentation naïve de l'algorithme

Bien, implémentons l'algorithme que nous avons décrit plus haut. Ce n'est pas forcément très difficile, surtout que maintenant on peut détecter très vite, grâce au test, lorsque nous avons fait une erreur.

```
1 eyr:2029 byr:1931 hcl:z cid:128
2 ecl:amb hgt:150cm iyr:2015 pid:148714704
3
4 byr:2013 hgt:70cm pid:76982670 ecl:#4f9a1c
5 hcl:9e724b eyr:1981 iyr:2027
6
7 pid:261384974 iyr:2015
8 hgt:172cm eyr:2020
9 byr:2001 hcl:#59c2d9 ecl:amb cid:163
```

Testons:

```
1 $ go test ./day_05 -v
2 === RUN    TestSeatID
3 --- PASS: TestSeatID (0.00s)
4 PASS
5 ok        gitlab.com/neuware/aoc-2020/day_05    0.001s
```

Parfait! Le reste de l'exercice est trivial à coder.

6.3. Et s'il y avait une méthode plus maligne ?

Voyons voir. Ce code de 10 chiffres va engendrer un nombre compris entre 0 et... $127 * 8 + 7 = 1023$. Tiens, 1023, c'est assez proche de 2^{10} . Et si ce code n'était rien d'autre qu'une représentation binaire un peu bizarre où B et R vaudraient 1, et F et L vaudraient 0?

Essayons en convertissant nos trois cas de test:

- BFFFBBFRRR donnerait 1000110111 en binaire, soit 567,
- FFFBBBFRRR donnerait 0001110111 en binaire, soit 119,
- BBFFBBFRLR donnerait 1100110100 en binaire, soit 820.

Bon sang! Il suffit de parser ce code comme s'il s'agissait d'une représentation binaire!

Implémentons-le.

```
1 $ go test ./day_05 -v
2 === RUN    TestSeatID
3 --- PASS: TestSeatID (0.00s)
```

6. Jour 5: Parser des nombres binaires de façon détournée

```
4 PASS
5 ok      gitlab.com/neuware/aoc-2020/day_05      0.001s
```

Bien sûr, on peut soumettre ce code au même test que `seatID` pour vérifier qu'il fonctionne...

6.4. Quelle méthode est la plus performante ?

Vu le nombre d'opérations élémentaires que l'on réalise à chaque boucle sur des entiers, il ne serait pas surprenant que la seconde implémentation soit plus efficace que la première. Pour nous en convaincre, réalisons un benchmark!

Pour ce faire, ajoutons les deux fonctions suivantes à notre fichier de test:

```
1 $ go test ./day_05 -v
2 === RUN   TestSeatID
3 --- PASS: TestSeatID (0.00s)
4 PASS
5 ok      gitlab.com/neuware/aoc-2020/day_05      0.001s
```

Ces fonctions de benchmark vont itérer autant de fois qu'elles le peuvent. À chaque itération, on s'assure que l'on utilise une valeur d'entrée différente, et que le résultat de la fonction est *utilisé*, en le comparant à la valeur attendue, pour éviter que l'optimiseur de Go ne dégage purement et simplement la boucle.

On peut lancer le bench comme ceci:

```
1 $ go test ./day_05 -bench=. -cpu=1
2 goos: linux
3 goarch: amd64
4 pkg: gitlab.com/neuware/aoc-2020/day_05
5 BenchmarkSeatID      66955075      17.4 ns/op
6 BenchmarkSeatID2    88646050      13.1 ns/op
7 PASS
8 ok      gitlab.com/neuware/aoc-2020/day_05      2.363s
```

Pour comprendre ce qui est affiché, chaque fonction est ici exécutée en boucle pendant une durée *fixe* (une seconde). Le nombre qui apparaît immédiatement après le nom de la fonction, est le nombre de fois que la boucle a eu le temps de tourner. La colonne à droite nous donne le temps d'exécution moyen par itération.

Bon, ici, je ne suis pas certain que le peu de variété dans les données d'entrées n'introduise un biais dans l'évaluation des performances. Cela dit, on observe quand même que la seconde fonction est un peu plus efficace que la première, mais que cette différence est vraiment de l'ordre d'un pouillème (4 nanosecondes par opération) autant dire que c'est kif-kif.

6. Jour 5: Parser des nombres binaires de façon détournée

Mais au moins, vous savez maintenant comment on écrit des tests et des benchmarks en utilisant la *toolchain* standard de Go. 🍊

Comme d'hab, vous pourrez trouver ma solution finale [ici](#) 📄 .

C'est tout pour cette fois. À dans 5 jours! 🍊

Liste des abréviations

- AFD** Automate Fini Déterministe. 5
- JWT** JSON Web Token. 1, 8–10
- PCRE** Perl Compatible Regular Expressions. 5