NOHAR

Zeste de savoir

L'Advent of Code 2020 en Go, jours 11 à 15

15 décembre 2020

Table des matières

Ho, ho, ho![1]

Nous voici au troisième épisode de cette série.

Si vous ne savez encore de quoi il retourne, voici la liste des épisodes précédent:

— Jours 1 à 5 , où l'on plante le décor.
— Jours 6 à 10 , où la difficulté augmente d'un cran.

1. Jour 11 : Un automate cellulaire qui respecte les gestes barrières

Connaissez-vous le jeu de la vie de Conway? L'exercice du jour 11 y ressemble beaucoup, en tout cas, puisqu'il s'agit d'un automate cellulaire en 2 dimensions.

La différence, c'est qu'au lieu de modéliser des cellules, on modélise le comportement de gens qui s'installent sur des chaises dans une *très grande* salle d'attente (9025 places, quand même!).

1. Non, je ne m'en lasse pas!

Prenons le temps de planter le sujet. Nous avons une salle comme la suivante:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

Ici, les `.` désignent le sol, les `L` des sièges vides et les `#` des sièges occupés[2]. Les gens vont venir s'asseoir aux places qui leur semblent confortables ou s'en aller quand il y a trop de monde autour. L'*énoncé nous précise* que l'automate va évoluer pendant un nombre fini de générations **jusqu'à arriver à un état stationnaire**. Où plus rien n'aura besoin de bouger.

1.1. Modélisation du problème

Ici, il est relativement aisé d'anticiper que ce qui va changer entre les deux parties de l'exercice, c'est *le jeu de règles* que les cellules vont suivre pour changer d'état. Cela veut dire que l'exo se prête assez bien à écrire une première modélisation un peu léchée avant même de commencer à réfléchir aux questions de l'énoncé. Et en réalité, *c'est précisément* ce que je trouve intéressant dans cet exercice.

1.1.1. Les fausses pistes à éviter

Nous pourrions partir dès le début sur une modélisation simple, où la grille n'est qu'un tableau à deux dimensions du style `[][]uint8`, que l'on muterait à chaque génération. Seulement, ça ne peut pas fonctionner, car les règles de mutation vont nous imposer d'examiner le voisinage de chaque cellule à l'instant T, donc si on modifie le tableau en même temps que l'on évalue les règles, ça va se mordre la queue.

Qu'à cela ne tienne, pourriez-vous répondre, *on n'a qu'à créer un nouveau tableau à chaque génération!*

C'est vrai, ça règlerait le problème, mais le fait d'allouer dynamiquement un nouveau tableau de cette taille à chaque génération n'est pas gratuit. Ça a un coût en performances qui n'est pas du tout négligeable, donc si on pouvait éviter dès le début, ce ne serait pas plus mal.

Et c'est ainsi que l'on arrive au **double buffering**, dont l'idée est de n'allouer que deux tableaux, que l'on permute à chaque nouvelle génération:

— Génération N-1 : on lit le tableau `A` et on écrit dans `B`;

2. Vous aussi, ça vous *trigger* d'imaginer tous ces gens entassés dans une salle d'attente sans respecter les distances de sécurité? Bienvenue en 2020!

— Génération N : on lit le tableau `B` et on écrit dans `A` ;
— Génération N+1 : on lit le tableau `A` et on écrit dans `B` ;
— etc.

Pour faire cela efficacement, nous allons donc manipuler des **pointeurs** sur nos grilles. Cela rend tout de suite l'utilisation de *slices* beaucoup moins pratique. Autant Go a tendance à déréférencer automatiquement les pointeurs dans de nombreux cas (et même faire l'opération inverse dans d'autres), autant sur des slices, c'est pas trop ça. Typiquement, le code suivant lèvera une erreur à la compilation:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

Entre cela, et le fait que les accès dans les tableaux 2D sont légèrement (~10% d'après mes observations) moins efficaces que dans une belle slice contiguë en mémoire, nous avons tout intérêt à modéliser notre grille par une jolie structure:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

1.1.2. Implémenter les méthodes du type `Grid`

En général, quand on implémente un type dont les membres sont *privés* comme ici (leur nom commence par une minuscule), il convient de respecter des conventions:

— Écrire une ou plusieurs fonctions publiques dont le nom commence par `New` pour l'instancier,
— Lui ajouter des accesseurs et des mutateurs au besoin.

J'ai implémenté tout ça dans le fichier `grid.go` . Voici un exemple qui résume les méthodes les plus importantes:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

1.1.3. Modéliser l'exécution

Nous savons que ce qui va changer sera très certainement les règles pour décider si un siège libre doit être occupé, ou si un siège occupé doit être libéré. Il est aisé de deviner la signature de telles fonctions: on leur passe la grille et les coordonnées de la cellule que l'on vise, et elles nous répondent par `true` ou `false`. Nous pouvons donc définir le type suivant:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

En nous promettant de revenir plus tard sur les implémentations concrètes.

Modélisons maintenant l'exécution d'une génération. Nous savons qu'une génération va prendre en entrée:

— Une `*Grid` dans laquelle lire l'état de la génération précédente,
— Une `*Grid` dans laquelle écrire l'état courant,
— Une `SeatEvalFunc` pour décider si un siège libre peut être occupé,
— Une `SeatEvalFunc` pour décider si un siège occupé doit être libéré.

Afin de détecter la stabilisation de l'automate, cette fonction retournera un `int`: le nombre de cellules modifiées à cette génération.

Voici comment je l'ai implémentée:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

Cela ne devrait pas vous poser de difficultés de lecture.

Le reste du code ”générique” se trouve dans le fichier `main.go` . Il n'y figure pas grand chose d'assez intéressant pour que je m'y attarde dans ce billet.

1.2. Modéliser les règles qui répondent aux questions

Bien, maintenant que notre environnement d'exécution est écrit, nous n'avons plus qu'à écrire les règles de notre automate cellulaire, à savoir nos quatre `SeatEvalFunc`.

1.2.1. Partie 1 : Des règles classiques ”pré-2020”

Dans la partie 1, le modèle est le suivant:

— Un siège libre à la génération `N` va devenir occupé à la génération `N+1` si aucun tous les sièges de son voisinage 8-connexe (haut-bas-gauche-droite + les diagonales) sont libres.
— Un siège occupé à la génération `N` va se libérer à la génération `N+1` si 4 sièges ou plus de son voisinage 8-connexe sont occupés.

Commençons par définir les coordonnées `X` et `Y` d'un *voisinage 8-connexe*:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

En toute rigueur, au lieu de boucler sur ces "directions", on pourrait écrire une simple boucle qui tourne autour des coordonnées d'un siège, mais ma boule de cristal me dicte que cette approche s'avérera payante dans la partie 2. 🍊

Bien, maintenant, écrivons la fonction `CanTakeSeatPart1`, qui indique si un siège de coordonnées `(x, y)` peut devenir occupé:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

Aucune difficulté particulière, on retourne `false` dès que l'on tombe sur un siège occupé. 🍊

Terminons avec cette partie avec la fonction `MustLeaveSeatPart1` qui nous indique si le voisinage immédiat d'un siège comporte 4 sièges occupés ou plus:

```
L.LL.LL.LL
LLLLLLL.LL
L.L.L..L..
LLLL.##.LL
L.LL.LL.LL
L.L#LLL.LL
..L.L.....
LLLLLLLLLL
L.L###LL.L
L.LLLLL.LL
```

1.2.2. Partie 2 : Appliquons les gestes barrières !

Nan, je déconne. C'est pas comme ça que le raconte l'énoncé, même si au final ça y ressemble un peu! 🍊

Dans la partie 2, une difficulté supplémentaire est ajoutée. On ne considère plus seulement le voisinage immédiat d'un siège, mais *le premier obstacle* que l'on rencontre quand on regarde dans une des 8 directions.

Autrement dit:

```
X voit un siège occupé à sa droite
X.....#LL

X voit un siège libre à sa droite
X.....L##
```

Autrement, les règles sont :

— Un siège libre peut devenir occupé si tous les autres sièges dans son champ de vision sont libres.
— Un siège occupé va se libérer si 5 sièges ou plus sont occupés dans son champ de vision.

Voici enfin les deux fonctions correspondantes:


```
X voit un siège occupé à sa droite
X.....#LL

X voit un siège libre à sa droite
X.....L##
```


Et voilà, problème résolu.

C'était long!

2. Jour 12 : L'art de ne pas tomber dans la *float*

Le douzième jour mélangeait 2 ingrédients plutôt sympa:

— Le but est d'écrire une VM pour un petit langage;
— Qui réalise (dans la partie 2) des transformations géométriques dans le plan.

L'entrée est une successions d'ordres que l'on donne au système de navigation d'un bateau, comme ceci:

```
N15
E30
L180
F10
R90
W5
S10
F30
```

Chaque commande est bien sûr composée d'une opération (la première lettre) et d'un argument (le nombre). La signification des opérations varie entre les deux parties.

2.1. Partie 1 : Un genre de module *turtle*...

Dans la première partie on considère que les opérations sont des ordres que l'on donne immédiatement au navire:

— `N`, `S`, `E`, `W` indiquent qu'il faut bouger de n cases vers le Nord/le Sud/l'Est/l'Ouest.
— `L` et `R` indiquent qu'il faut se tourner de n degrés vers la gauche ou la droite.
— `F` indiquent qu'il faut bouger de n cases dans la direction vers laquelle on est tourné.

Les angles donnés à `L` et `R` sont exprimés en degrés, et ce sont systématiquement des multiples de 90°. On considère que le navire est initialement tourné vers l'Est. Tout l'intérêt du problème repose sur la modélisation *des angles* pour savoir dans quelle direction le navire est tourné. J'ai personnellement opté pour le modèle suivant:

```
N15
E30
L180
F10
R90
W5
S10
F30
```

La table d'association `Move` ne devrait pas vous surprendre. C'est plutôt le tableau `Cardinal` sur lequel je pense qu'il faut que je m'attarde.

Dans mon modèle, la direction vers laquelle on est tourné ("l'angle") est représenté par un indice dans ce tableau:

— 0° -> Indice 0 -> on est tourné vers l'Est.
— 90° -> Indice 1 -> on est tourné vers le Nord.
— 180° -> Indice 2 -> on est tourné vers l'Ouest.
— 270° (ou -90°) -> Indice 3 -> on est tourné vers le Sud.

Ainsi:

— pour tourner de 90° vers la gauche, on ajoute 1 à cet indice (et on ajuste si on dépasse du tableau),
— pour tourner de 90° vers la droite, on soustrait 1 à cet indice (et on ajuste…).

Ce calcul est donné par la fonction suivante:

```
N15
E30
L180
F10
R90
W5
S10
```

```
F30
```

Cela nous permet de résoudre la partie 1 au moyen de cette fonction:

```
N15
E30
L180
F10
R90
W5
S10
F30
```

2.2. Partie 2 : Un système de navigation un peu plus réaliste

Dans la partie 2, l'histoire nous raconte que la feuille sur laquelle sont écrits les ordres de navigation contenait un manuel d'utilisation rédigé au verso…

En fait, il s'agit d'un système de navigation au moyen d'un *waypoint*. Un *waypoint* est un point qui s'exprime *dans le référentiel du navire*, et qui indique une destination immédiate. Dans ces conditions, les ordres signifient:

— `N`, `S`, `E`, `W` : déplacer *le waypoint* de n cases vers le Nord/Sud/Est/Ouest.
— `L` et `R` : faire tourner *le waypoint* autour du navire, de n degrés.
— `F` : se translater n fois dans la direction indiquée par le *waypoint*.

OK, on n'y coupera pas, on va devoir faire un peu de géométrie pour gérer ces rotations.

2.2.1. Comprendre la rotation du waypoint

La rotation du *waypoint* autour du navire est en réalité *une rotation autour de l'origine* puisque les coordonnées de celui-ci sont exprimées dans le référentiel du navire. Ça veut dire qu'on peut l'exprimer comme ceci (où θ est l'angle de rotation):

— $x\prime = \cos(\theta)x - \sin(\theta)y$
— $y\prime = \sin(\theta)x + \cos(\theta)y$

Ouaip, c'est de la trigo de lycée, mais attendez un peu avant de dégainer le module `math` et des calculs en arithmétique flottante.

Seul le navire a besoin de flotter dans cet exercice. **Pas nos virgules!**

Dans notre cas, on dispose d'une astuce assez cool pour éviter de se taper des calculs de cosinus et de sinus. Regardez notre tableau `Cardinal`!

On a vu un peu plus haut que celui-ci permettait d'obtenir une direction en fonction d'un angle d'orientation. Par exemple, si on a un angle de 90°, cela nous donne la direction Nord, dont les coordonnées sont $(0, 1)$. Remarquez que les coordonnées en x et en y de cette direction sont *très exactement* le cosinus et le sinus de 90°!

On pourrait s'amuser à démontrer que cette relation est valable sur les 4 directions cardinales, en nous servant des formules de calcul du cosinus et du sinus dans un triangle rectangle (vues au collège, si ma mémoire est bonne)… Faites-le si ça vous amuse, mais moi pendant ce temps-là je vais plutôt écrire la fonction qui calcule la rotation du *waypoint*. 🍊

```
N15
E30
L180
F10
R90
W5
S10
F30
```

Et voilà, le reste est plutôt direct:

```
N15
E30
L180
F10
R90
W5
S10
F30
```

C'était plutôt rigolo à programmer, une fois qu'on a trouvé l'astuce. Comme d'habitude, vous trouverez mon code complet ici .

3. Jour 13 : La prochaine fois je prendrai un Uber

Le jour 13 a battu le triste record du *problème qui m'a le plus gonflé ce mois-ci*. Et pourtant, il avait l'air inoffensif au départ!

On cherche à prendre une navette pour se rendre à l'aéroport. On sait qu'il y a N navettes, numérotées **selon leur période**. Par exemple, le bus 19 va passer toutes les 19 minutes, le bus 23 toutes les 23 minutes, etc.

On nous fournit l'entrée du problème sous la forme de 2 lignes, comme ceci:

```
939
7,13,x,x,59,x,31,19
```

La première ligne nous dit quelle heure il est, la seconde ligne nous donne l'ensemble des bus du service de navettes, sachant que les bus marqués d'un `x` sont hors-service.

Ayons le nez creux: ces `x` ne sont certainement pas là pour rien, remplaçons-les par des 0 lorsque nous allons parser nos entrées!


```
939
7,13,x,x,59,x,31,19
```


3.1. Partie 1 : Super ! Une question facile !

La première question 1 est triviale: *quel est le prochain bus qui va passer, et combien de temps allons-nous devoir l'attendre?*

Il suffit de dégainer l'opérateur modulo!

Par exemple, sachant que l'heure actuelle est 939, le bus N°13 est passé il y a `939 % 13 = 3` minutes, donc il repassera dans `13 - 939%13 = 10` minutes. Il nous suffit donc de faire une recherche de minimum dans la liste des navettes en circulation:


```
939
7,13,x,x,59,x,31,19
```


Allez, on soumet la réponse et on découvre la prochaine partie.

3.2. Partie 2 : Où l'on se retrouve téléporté sans consentement sur le Project Euler

Bon, là, désolé, mais j'ai un poids à évacuer!

> ! Si vous aussi, vous avez souffert sur cette question, je vous souhaite que les lignes qui suivent soient aussi cathartiques à lire pour vous qu'elles l'ont été à écrire pour moi. Sinon, vous pouvez sauter au paragraphe suivant.

Je vous disais la semaine dernière que les problèmes de dénombrement étaient "ma bête noire". En fait, pour être plus juste, il faudrait que je précise que ce que je **hais** vraiment, ce sont tous ces problèmes qui mélangent gratuitement de l'algorithmique et des maths pures et dures, juste

pour le plaisir de faire des maths. Vous savez, ce genre de maths dont on s'est toujours demandé à quoi ça nous servirait dans la vie, à part résoudre des exos sur le *Project Euler*…

En voici un beau spécimen dans toute son horrible gratuité. Même l'auteur de l'exo semble avoir abandonné l'idée de le rendre crédible ou amusant!

> The shuttle company is running a contest : one gold coin for anyone that can find the earliest timestamp such that the first bus ID departs at that time and each subsequent listed bus ID departs at that subsequent minute.

Là, donc, il n'est plus question de sauver le père Noël, d'aider un comptable, de réparer un système de navigation, ni même de respecter les gestes barrières dans une salle d'attente. Là, c'est *un concours pour gagner une pièce d'or*. Et pour cause, ce concours qui tombe comme un cheveu sur la soupe est bien la seule excuse qu'on puisse imaginer pour s'infliger un truc pareil qui n'a strictement **aucune application pratique** dans *La Vraie Vie™*.

Mais gardez votre pièce d'or et foutez-moi la paix, à la fin! Je m'en fous d'être pauvre, moi. Tout ce que je veux c'est continuer mon aventure pour arriver jusqu'à l'île de mes vacances!

Je me suis déjà tapé une descente en luge en me bouffant des arbres tous les 3 mètres. J'ai Interpol aux fesses parce que j'ai piraté le portique qui vérifie les passeports à la frontière le deuxième jour, tout ça pour voyager avec une compagnie aérienne qui frôle l'incompétence avec son système de placement dans l'avion, et ses consignes de sécurité qui m'obligent à fourrer 18 885 pochons en plastique de couleurs différentes dans ma valise. Il a fallu en plus que je cracke le système de divertissement à bord pour pouvoir mater un film et que je répare la *gameboy* du gosse assis à côté de moi. Et après ça j'ai encore dû débugger le système de navigation du ferry pour éviter de sombrer dans une tempête.

Mais qu'est-ce qui vous fait croire que j'ai envie de me coltiner un concours de maths, là, au juste ?!

3.2.1. OK, calmons-nous. Qu'est-ce qu'on cherche, en fait ?

Reprenons la seconde ligne de l'exemple plus haut:

```
7,13,x,x,59,x,31,19
```

Ce que l'on cherche, c'est le temps `T` auquel on aura:

— Le bus 7 qui part à l'heure `T`,
— Le bus 13 qui part à l'heure `T+1`,
— Le bus 59 qui part à l'heure `T+4`,
— Le bus 31 qui part à l'heure `T+6`,
— Le bus 19 qui part à l'heure `T+7`.

~~*Comme si quelqu'un en avait quelque chose à f*~~… D'accord, d'accord, je suis calme. Cherchons ça.

3.2.2. Est-ce qu'on peut trouver la solution en bourrinant ?

L'exo nous précise que la solution dépasse 100000000000000. Faisons un calcul sur un coin de table. En supposant qu'il nous faut environ ~70ns pour vérifier toutes les contraintes sur un nombre donné. Combien de temps on mettrait, au minimum, avant de tomber sur la solution en essayant tous les nombres?

$$\frac{100000000000000 \times 70 \times 10^{-9}}{3600 \times 24 \times 30} = 2.700617...$$

Deux-trois mois[3]... Bon, pas le choix. Je vais faire du café.

3.2.3. Comment réduire ce temps ?

Si le *brute force* en partant de 0 et en incrémentant notre nombre de 1 à chaque itération prendrait trop de temps, c'est parce qu'on essaye trop de possibilités. Essayons de diminuer le nombre de candidats à tester. Commençons par observer ces nombres.

La première remarque que l'on peut faire c'est que **les numéros des bus sont tous premiers**. Gardons cette propriété en tête, ça ne peut pas être un hasard!

Ensuite réfléchissons de façon incrémentale. Si on a trouvé un temps `T0` qui satisfait la contrainte pour l'ensemble de contraintes `[7]`, comment fait-on pour chercher `T1` qui satisfasse `[7, 13]` sans tester tous les nombres un par un? On sait déjà que `T1` devra être divisible par 7, donc on peut *incrémenter de 7* jusqu'à trouver un nombre *divisible par 13 lorsque l'on lui ajoute 1*. Facile! On peut considérer `T1` comme acquis.

Réfléchissons maintenant en termes de fréquences. `T1` est la conjonction de deux phénomènes:

— Un qui se produit tous les 7 cycles (le passage du bus 7).
— Un autre qui se produit tous les 13 cycles (le passage du bus 13).

Puisque 7 et 13 sont premiers, **on peut en déduire que ce phénomène ne se reproduira pas avant** $7 \times 13 = 91$ **cycles**, donc on peut ajuster notre pas pour tester les nombres en les incrémentant de 91 en 91 à partir de `T1`, jusqu'à tomber sur un nombre `T2` *qui soit divisible par 59 quand on lui ajoute 4*.

Maintenant que l'on a `T2`, nous savons que nous voulons reproduire ce phénomène qui est la superposition:

— D'un premier phénomène qui se produit tous les 91 cycles,
— D'un second phénomène qui se produit tous les 59 cycles.

Puisque 91 est le produit de deux nombres premiers différents de 59, on sait que l'on devra maintenant incrémenter nos candidats de $91 \times 59 = 5369$ pour tester uniquement les nombres qui satisfont les trois premières contraintes...

3. Tout ça pour parcourir 1cm du trajet final!

> L'idée, c'est donc de faire tomber les contraintes une par une: dès que l'on a satisfait une contrainte, on multiplie notre incrément par la période du bus, et on passe à la contrainte suivante.

Voici comment j'ai implémenté ma solution:

```
7,13,x,x,59,x,31,19
```

Sur ma machine et mon ensemble d'entrée, le résultat final est trouvé en 4µs. Je sais qu'on pourrait faire mieux en trouvant une équation qui calcule immédiatement le résultat, mais 4µs, ça me suffit **largement**!

FIGURE 3.1. – *sifflote*

Allez, oublions ce mauvais souvenir.

4. Jour 14 : Des bitmasks à la pelle

Le 14 , nous sommes revenus en terrain plus "civilisé".

Il s'agit une nouvelle fois d'implémenter une petite VM, qui réalise cette fois des opérations bit-à-bit sur ses entrées.

Concrètement, un programme peut ressembler à ceci:

```
mask = XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
mem[8] = 11
mem[7] = 101
mem[8] = 0
```

Nous allons travailler explicitement avec des entiers non-signés sur 36 bits, ce qui veut dire que nous allons manipuler explicitement des `uint64` pour contenir ces entiers et garder un code qui soit portable.

Nous avons donc deux types d'instructions:

- `mask ....` définit le bitmask courant du programme.
- `mem[X] = Y` écrit la variable `Y` à l'emplacement mémoire `X`. Ce qui se passe réellement pendant cette opération va changer entre les deux parties, et bien évidemment dépendre du masque.

Je ne parlerai pas de la partie "exécution" ici, car elle ne présente pas de difficulté particulière, surtout que des VM, on en a déjà implémenté deux autres depuis le début…

4.1. Modélisation

Toute la subtilité de l'exercice réside dans la façon dont le masque va se comporter, mais vous pouvez être sûrs qu'il va s'agir d'opérations bit-à-bit!

Pour commencer, parsons nos masques de manière à ce que ceux-ci soient représentés par 3 nombres:

— `zeros` est un nombre qui indique tous les bits du masque qui sont à `0`.
— `ones` est un nombre qui indique tous les bits du masque qui sont à `1`.
— `floats` (le nom aura du sens plus tard) est un nombre qui indique tous les bits du masque qui sont à `X`.

Ainsi, sur notre exemple plus haut:

```
Mask  : XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
zeros : 00000000000000000000000000000010
ones  : 00000000000000000000000001000000
floats: 11111111111111111111111110111101
```

Cela nous donne le code suivant pour construire un masque:

```
Mask  : XXXXXXXXXXXXXXXXXXXXXXXXXXXXX1XXXX0X
zeros : 00000000000000000000000000000010
ones  : 00000000000000000000000001000000
floats: 11111111111111111111111110111101
```

4.2. Partie 1 : Mettre les bits d'un masque à 0 ou à 1 dans un nombre

Dans la première partie, le but est d'appliquer plus ou moins directement le masque sur des nombres:

— On veut que les `1` du masque soient mis à `1` dans le nombre,
— On veut que les `0` du masque soient mis à `0` dans le nombre.

Je soupçonne cette première question d'être un simple "rappel" sur l'utilisation de *bitmasks*. Mais qu'à cela ne tienne, faisons-le sérieusement.

Prenons le masque ci-dessus, mais ramenons-le à 8 bits par commodité: `X1XXXX0X` et appliquons-le au nombre 11 (en binaire : `00001011`).

```
bits: 76543210
--------------
mask: X1XXXX0X
N   : 00001011
Res.: 01001001  (= 73 en base 10)
```

Comme vous le voyez, les bits marqués d'un X n'ont pas bougé, le bit 1 est passé à 0 et le bit 6 est passé à 1. Nous allons implémenter ce comportement dans la méthode `Apply` de notre type `Mask`, mais **avant de nous lancer là-dedans**, commençons par écrire un test, en reportant les exemples de l'exercice:

```
bits: 76543210
--------------
mask: X1XXXX0X
N   : 00001011
Res.: 01001001  (= 73 en base 10)
```

Comme vous le constaterez si vous allez lire les tests que j'ai écrits pour à peu près tous les exos jusqu'à aujourd'hui, ceux-ci suivent plus ou moins tout le temps la même forme.

— Je déclare une structure dans laquelle j'écris mes "cas",
— J'itère sur chaque cas, et je vérifie que la sortie produite est égale à celle attendue.

Il faut savoir qu'il existe des packages en Go qui permettent de faire ressembler nos tests un peu plus à ce à quoi les gens ont l'habitude (avec des assertions, des suites de tests, du *setup* et du *teardown*…), mais ici, je me contente d'utiliser la *toolchain* standard.

Maintenant que nous avons écrit ce test, écrivons notre méthode.

Comme je vous l'ai montré plus haut, j'ai séparé ces masques en 3 sous-masques. Il ne reste plus qu'à utiliser les opérateurs qui vont bien.

— Pour mettre un bit à 1, c'est simple, c'est le "OU" bit-à-bit (`|`).
— Pour mettre un bit à 0 sachant que l'on dispose d'un masque qui marque ce bit d'un 1, cela demande deux étapes:
  — On inverse le masque pour qu'il soit rempli de 1 par défaut, avec les bits que l'on veut changer à 0.
  — On applique le "ET" bit-à-bit (`&`).
  — … Ou bien on fait tout ça en n'utilisant qu'un seul opérateur: `&^`, le "XOR" bit-à-bit.

Cela nous donne la méthode suivante, beaucoup plus rapide à écrire qu'à décrire:

```
bits: 76543210
--------------
mask: X1XXXX0X
N   : 00001011
```

```
Res.: 01001001  (= 73 en base 10)
```

4.3. Partie 2 : générer toutes les combinaisons possibles sur les bits ”flottants”

Pour la partie 2, appliquer un masque revient à générer plusieurs nombres à partir d'une base donnée.

— Les bits à `1` dans le masque seront à 1 dans tous les nombres générés,
— Les bits à `0` dans le masque seront recopiés depuis le nombre de base,
— Les bits à `X` dans le masque sont dits ”flottants”: ce sont eux qui vont varier.

Le but du jeu est de générer toutes les combinaisons possibles sur les bits flottants.

Prenons par exemple le masque `00X1001X` et le nombre `42` (`101010`) comme base. Nous allons commencer par déterminer la base des nombres que nous allons générer.

```
n   : 00101010
mask: 00X1001X
base: 00-1101-
```

Comment calculer cette base?

— Pour les `1` c'est facile, il suffit encore une fois d'appliquer le ”OU” bit-à-bit.
— Pour les `0` remarquez que si l'on applique le masque `zeros` avec un ”ET” bit-à-bit, cela va remettre à 0 tous les bits qui ne font pas partie du masque `zeros` dans le nombre.

On obtient donc `base := n & m.zeros | m.ones`. Notez qu'en nous y prenant ainsi, les bits ”flottants” sont initialisés à 0 dans cette base.

C'est maintenant que les choses rigolottes commencent. Il faut que nous générions tous les nombres possibles à partir de cette base. Remarquez que cela revient à générer **tous les nombres à 2 bits** possibles, et de répartir ces bits dans les deux ”trous” de la base.

Supposons que nous ayons le nombre `i = ab` à répartir dans notre base (où `a` et `b` sont des bits arbitraires), et que nous désirons obtenir `00a1101b` à partir de là.

— On initialise notre nombre généré `x` à la base
— Commençons par isoler le bit 0 (`b`) du nombre `i`: `i & 1`.
— Puis appliquons-le à notre nombre: `x |= i&1`.
— On isole ensuite le bit 1 (`a`) du nombre `i`: `i > 1 & 1`.
— Puis on l'applique au bit 5 du nombre à générer: `x |= (i > 1 & 1) < 5`.

Dans le cas général, nous allons devoir itérer sur les bits du nombre `i`, ainsi que sur les bits qui sont à `1` dans le masque `floats`. C'est peut-être ces itérations qui sont les plus verbeuses dans la fonction qui suit. En dehors de ça, je ne fais qu'appliquer strictement ce que je viens de vous dire:

```
n   : 00101010
mask: 00X1001X
base: 00-1101-
```

Si vous avez suivi ces billets depuis le début, vous reconnaîtrez mon pattern `IterMachin`, qui permet d'itérer sur des résultats générés au fur et à mesure, au moyen d'un *callback*. Par exemple, pour retourner une slice qui contient tous les résultats, on l'utiliserait comme ça:


```
n   : 00101010
mask: 00X1001X
base: 00-1101-
```


Cela dit, je préfère exposer ma méthode `IterAddr` parce que ça me dérange de réaliser systématiquement une allocation dynamique dans une méthode qui sera appelée aussi souvent.

Ceci suffit à résoudre les principales difficultés de cet exercice. Vous pourrez trouver le reste ici .

4.4. Bonus : un sort de magie noire pour remplir les bits flottants

Je tiens à remercier @Berdes de m'avoir montré l'astuce, d'une élégance monstrueuse, qui suit.

Il existe une façon beaucoup moins laborieuse de générer nos valeurs, en créant un compteur qui va automatiquement faire bouger les bits dont on a besoin.

Supposons que notre masque soit `0X000X00` (donc `mask.floats == 01000100`). Ce que nous voulons c'est un compteur qui génère les 4 valeurs suivantes, que l'on n'aurait plus qu'à `|` avec notre nombre de base:

```
i = 00000000
i = 00000100
i = 01000000
i = 01000100
```

Voici comment on pourrait s'y prendre:

- On part de `i = 00000000`
  - On génère `base | i` et on l'envoie où on veut…
- Pour générer `00000100` :
  - `i |= ^mask.floats` (`i = 10111011`)
  - `i += 1` (`i = 10111100`)
  - `i &= mask.floats` (`i = 00000100`)
  - On génère `base | i` et on l'envoie où on veut…
- Pour générer `01000000` :
  - `i |= ^mask.floats` (`i = 10111111`)

— `i += 1` (`i = 11000000`)
— `i &= mask.floats` (`i = 01000000`)
— On génère `base | i` et on l'envoie où on veut…

Vous aurez certainement compris que ce qui fait toute la magie dans cette opération, c'est l'addition du milieu qui pousse la retenue jusqu'à boucher "le prochain trou"… Ce qui est encore plus joli, c'est qu'une fois que l'on a généré toutes les valeurs, on se retrouve à ajouter 1 au nombre `11111111`, donc provoquer un *overflow* qui remet notre compteur à zéro, et nous permet ainsi de détecter la fin de la boucle.

Le code de `IterAddr` devient **beaucoup** plus simple, tout à coup:

```
i = 00000000
i = 00000100
i = 01000000
i = 01000100
```

5. Jour 15 : La mémoire, ça coûte pas cher, mais c'est pas gratuit non plus !

L'exercice du 15e jour est un cas d'école assez intéressant, parce qu'il n'existe pas vraiment de solution idéale. Par contre, ça va être un bon prétexte pour comparer les deux conteneurs *builtin* de Go.

On part d'une suite de nombres:

```
0
3
6
```

Pour la compléter, on regarde le dernier nombre qui a été cité:

— Si le dernier nombre cité apparaît pour la première fois, alors le nouveau nombre est `0`
— S'il avait déjà été cité, alors on dit depuis combien de cycles.

C'est plus facile à comprendre *in situ*:

```
0
3
6
0  # car 6 est apparu pour la première fois
3  # car 0 avait déjà été cité trois cycles plus tôt
3  # car 3 avait déjà été cité trois cycles plus tôt
```

```
1  # car 3 venait d'être dit
0  # car 1 est apparu pour la première fois
4  # car 0 avait déjà été cité quatre cycles plus tôt
0  # car 4 est apparu pour la première fois
```

Le but de la première partie est de donner le 2020^{e} nombre de la suite ainsi formée. Celui de la seconde partie, le 30 000 000^{e}. 🍏

5.1. Choisissons une complexité linéaire dès le départ

Il est évident que l'algorithme naïf est en $O(N^2)$, puisque cela impliquerait, à chaque nouveau nombre, de regarder dans toute la liste des nombres déjà cités s'il ne s'y trouve pas. Et le pire, c'est que puisqu'il faut remonter cette liste de la fin vers le début, le cache du CPU sera incapable de nous aider à accélérer cette recherche de manière efficace.

Pour cette raison, nous allons nous rappatrier sur un algorithme en $O(N)$: nous allons garder en mémoire la dernière fois que nous avons dit chaque nombre, et rechercher à chaque itération le dernier nombre que nous avons cité.

Reste à savoir quelle structure utiliser pour garder ces nombres en mémoire: un `map` ou une `slice`?

Essayons les deux!

5.2. Implémentation au moyen d'une `map`

Intuitivement, on peut se dire que l'on n'aura certainement pas besoin de stocker 1000 nombres pour compléter la série des 2020 premières itérations. Notre structure en mémoire sera donc *éparse*, plutôt qu'un tableau dense (comme un *array* ou une *slice*). Partant de là, essayons de minimiser l'espace que nous allons occuper en RAM en utilisant une `map[int32]int32`.

```
0
3
6
0  # car 6 est apparu pour la première fois
3  # car 0 avait déjà été cité trois cycles plus tôt
3  # car 3 avait déjà été cité trois cycles plus tôt
1  # car 3 venait d'être dit
0  # car 1 est apparu pour la première fois
4  # car 0 avait déjà été cité quatre cycles plus tôt
0  # car 4 est apparu pour la première fois
```

Voyons ce que cela donne dans un benchmark:

```
name          time/op
Part1Hashmap  74.7µs ± 0%
Part2Hashmap   2.51s ± 0%

name          alloc/op
Part1Hashmap  12.8kB ± 0%
Part2Hashmap   200MB ± 0%

name          allocs/op
Part1Hashmap    31.0 ± 0%
Part2Hashmap    154k ± 0%
```

On répond à la partie 1 en 74µs et à la partie 2 en 2.5s. On remarque que dans la partie 2, les écritures dans la `map` nous obligent à allouer dynamiquement un total de 200MB de mémoire (même si cette mémoire est régulièrement libérée par le ramasse-miettes): cela nous donne ~154000 cycles allocations/recyclages de mémoire… Ça fait *vraiment beaucoup*, mais au final cela permet d'occuper à tout instant une quantité relativement faible de RAM.

5.3. Implémentation au moyen d'une slice

Puisqu'il n'y a aucun moyen de prévoir quels nombres seront cités à l'avance, nous sommes obligés de faire l'hypothèse que ces nombres vont aller de 0 à la borne haute (2020 ou 30 000 000) de notre problème. Cela signifie que si l'on veut stocker la dernière position de chaque nombre dans un tableau contigu, sa taille sera égale à la borne haute de notre problème.

Faisons un petit calcul sur un coin de table: pour stocker 30 000 000 entiers sur 32 bits (4 octets), nous avons besoin de… 120 Mo de mémoire. OK, c'est beaucoup, mais ça tient largement dans la RAM de ma machine. Essayons, ce n'est qu'une ligne à changer dans le code!

Voici les résultats:

```
name          time/op
Part1         3.57µs ± 0%
Part2          666ms ± 0%
Part1Hashmap  74.7µs ± 0%
Part2Hashmap   2.51s ± 0%

name          alloc/op
Part1         8.19kB ± 0%
Part2          120MB ± 0%
Part1Hashmap  12.8kB ± 0%
Part2Hashmap   200MB ± 0%

name          allocs/op
Part1           1.00 ± 0%
Part2           1.00 ± 0%
```

```
16 Part1Hashmap    31.0 ± 0%
17 Part2Hashmap    154k ± 0%
```

Cette nouvelle version est 20 fois plus rapide pour la partie 1, 4 fois plus rapide pour la partie 2, tout en allouant globalement *moins de mémoire* au total, même si la totalité de la mémoire allouée va rester mobilisée jusqu'à ce que nous ayons fini de calculer le résultat.

5.4. Que peut-on en conclure ?

Je pense que la conclusion qu'il faut tirer de cette expérience, c'est que dans le cas d'un exercice d'algorithmique comme celui-ci dont la durée est a priori courte, une `map` ne sera jamais un bon choix en termes de performances: au mieux, cela constitue un compromis pour éviter de consommer trop de RAM en même temps.

Attention, cela ne veut pas dire que les `map` ne servent à rien dans l'absolu! On peut notamment s'en servir dans des cas où nous avons besoin de garder des choses en RAM **pendant longtemps**, comme dans un serveur, par exemple. Mais d'une façon générale: si vos données sont indexées par des entiers et que l'espace des clés (le *keyspace*) est borné et suffisamment peu étendu pour que vous puissiez vous permettre d'allouer un *array* contigu, même si celui-ci ne sera criblé de trous, alors **cela vaudra toujours le coup** d'opter pour une *slice* du point de vue des performances.

Rétrospectivement, ces 5 exercices m'ont tous forcé à chercher une astuce, à part le 12 où j'ai cherché une astuce *parce que ça m'embêtait* de faire des calculs sur des flottants.

Ce que je remarque, c'est que chacun d'entre eux m'a poussé à me poser une question intéressante, dont je ne connaissais pas la réponse. Autrement dit, chacun de ces exercices m'a appris quelque chose à sa manière, et j'en suis ravi!