Maîtriser la programmation

Des tests à la performance en Go

Daniel Lemire

Table des matières

Introduction

Quand j'ai appris la programmation alors que j'étais un adolescent, j'ai rapidement eu l'impression d'en avoir fait le tour. Il y avait des boucles, les fonctions et ainsi de suite. Pourtant, à chaque fois que j'essayais de me lancer dans un projet de programmation ambitieux, je rencontrais des difficultés.
La programmation informatique est un activité riche qui peut nécessiter des années de pratique. Mon objectif dans cet ouvrage est d'amener le lecteur déjà familier avec la programmation informatique à mieux maîtriser le programmation dans son ensemble. En somme, le manuel vise à répondre en partie à la question suivante: comment pouvons-nous produire rapidement du code informatique qui est correct et efficace afin de résoudre des problèmes pertinent pour une organisation? En particulier, je souhaite que le lecteur puisse mieux faire le lien entre les préoccupations considérées comme étant purement techniques (bas niveau) comme le processeur et la mémoire, et les préoccupations plus abstraites comme la conception des algorithmes.

Les langages de programmation

La programmation informatique passe normalement par un langage de programmation. Je suppose que le lecteur est déjà familier avec un langage établi, comme le Java, le C# ou le C++. Je suppose aussi que le lecteur a des notions de base concernant l'organisation des ordinateurs et les algorithmes. Dans ce manuel, nous ne cherchons pas à vous amener

à maîtriser un langage de programmation en particulier. Les langages ont tous leurs forces et leurs faiblesses. On choisit souvent un langage particulier en fonction des gens avec qui on travaille, sur la base de notre familiarité ou en fonction de la tâche à accomplir. Je veux cet ouvrage accessible à tous les lecteurs, sans tenir compte de leur langage de programmation de prédilection. Néanmoins, je souhaite utiliser des langages de programmation concrets et pratiques afin de m'exprimer.

Au lieu d'utiliser un seul langage de programmation, je veux être capable d'en utiliser plusieurs en fonction des concepts abordés. Je choisis de présenter plusieurs de mes exemples dans un langage relativement neutre, le langage de programmation Go. Je ne prends pas pour acquis que le lecteur connaît le langage Go. En fait, je ne suppose pas que le lecteur choisira d'utiliser le language Go dans sa lecture. Il suffit d'en maîtriser la syntaxe de base.

Le lecteur curieux voudra peut-être faire le tutoriel A Tour of Go en ligne (https://tour.golang.org/). Les développeurs du langage Go mettent de l'avant un outil en ligne (https://go.dev/play/) qui vous permet d'écrire et d'exécuter des programmes en Go.

Pour commencer à utiliser Go sous Windows, visitez le site officiel de Go à l'adresse https://go.dev/dl/ et téléchargez le dernier programme d'installation pour Windows. Exécutez le programme d'installation, qui ajoutera le binaire Go au PATH de votre système. Pour confirmer l'installation, ouvrez une invite de commande ou PowerShell et tapez `go version` ; vous devriez voir la version installée, comme `go version go1.21.x windows/amd64`. Ensuite, créez un répertoire pour vos projets Go, comme `C:\NUsers\NVotreNom\Ngo`.

Pour commencer à utiliser Go sur macOS, le processus est similaire. Visitez le site officiel de Go à l'adresse https://go.dev/dl/ et téléchargez le dernier programme d'installation pour macOS. Pour confirmer l'installation, ouvrez un Terminal et tapez `go version` ; vous devriez voir la version installée. Ensuite, créez un répertoire pour vos projets Go, comme `/Users/YourName/go`.

En utilisant un éditeur de texte, créez un fichier nommé `hello.go` dans votre répertoire Go:

```
package main

import "fmt"

func main() {
    fmt.Println(« Hello, World ! »)
}
```

L'éditeur de texte par défaut sous Windows est Notepad. L'éditeur de texte par défaut sous macOS est TextEdit.

Par défaut, vous devez initialiser un module chaque fois que vous démarrez un nouveau projet. Initialiser un module en naviguant dans le dossier de votre projet et en exécutant `go mod init hello`, qui crée un fichier `go.mod` pour la gestion des dépendances. Dans votre cas, le fichier de module pourrait être aussi simple que:

```
module hello

go 1.24.2
```

Pour compiler votre code en un exécutable, utilisez `go build`, qui génère un fichier exécutable (par exemple, hello.exe). Pour démarrer un nouveau projet, vous pouvez créer un nouveau répertoire et répéter le même processus. La documentation Go à l'adresse https://go.dev/doc/ est une excellente ressource pour en savoir plus, et la commandego env‘ peut aider à résoudre les problèmes de configuration.

L'utilisation de Visual Studio Code (VS Code) comme éditeur peut vous rendre plus productif. Après avoir téléchargé et installé VS Code à partir de https://code.visualstudio.com/, installez l'extension Go à partir du marché des extensions. Cette extension offre des fonctionnalités telles que la complétion de code, le formatage et le débogage. Lors de l'installation,

VS Code peut vous demander d'installer des outils supplémentaires. Vous pouvez ouvrir votre dossier de projet Go dans VS Code, et l'éditeur reconnaîtra les fichiers .go, offrant une vérification des erreurs en ligne et des suggestions. Par exemple, l'écriture d'un fichier `hello.go` dans VS Code vous permet d'utiliser le terminal intégré pour exécuter `go run hello.go` ou mettre en place des configurations de débogage.

Vous pouvez aussi prendre en compte GoLand. GoLand est un outil de développement dédié au développement Go. Il fournit une complétion de code, des outils de refactorisation avancés, un débogage intégré et une intégration avec des outils spécifiques à Go. Vous pouvez acheter GoLand à l'adresse https://www.jetbrains.com/go/.

Voici rapidement quelques éléments de syntaxe concernant les variables et les structures de contrôle:

```
x := 1 // déclaration de la variable entière x
x = 2 // assignation d'une valeur '2' à la variable
if x > 2 {
  x -= 1
}
for i := 0; i < 10; i++ {
  x += i
}
```

La définition des fonctions en Go se fait avec le mot-clé `func` suivi du nom de la fonction. On déclare ensuite les paramètres de la fonction en mettant en séquence le nom de la variable et son type (ici `int` pour integer ou entier). La déclaration de la fonction se termine par un déclaration de type correspondant à la valeur retournée par la fonction. Nous procédons ensuite avec la définition de la fonction:

```
func pair(n int) bool {
  return n % 2 == 0
}
```

En Go, un paquetage est une unité organisationnelle fondamentale qui

regroupe du code apparenté. Chaque fichier source Go commence par une déclaration de paquetage, spécifiant le paquetage auquel il appartient, comme le paquetage main pour les programmes exécutables.

Avec ces quelques notions de syntaxe, vous devriez être capable de lire du code écrit en Go.

Un autre langage utile est le langage Python. Encore une fois, il n'est pas nécessaire que vous maîtrisiez le langage Python, il vous suffit d'être familier avec la syntaxe de base. Les langages comme JavaScript ou Go vous permettent d'indenter votre code pour une meilleure lisibilité: en Python, cette indentation est requise. Voici rapidement quelques éléments de syntaxe concernant les variables et les structures de contrôle:

```
x = 1 // déclaration de la variable entière x
if x > 2 :
  x -= 1
for i in range(10):
  x += i
```

La définition des fonctions en Go se fait avec le mot-clé `func` suivi du nom de la fonction. En Python, nous utilisons plutôt le mot-clé `def`. Nous procédons ensuite avec la définition de la fonction:

```
def pair(n) :
  return n % 2 == 0
```

Contrairement au langage de programmation Go, le type des variables n'est pas explicite en Python. Cependant, vous pouvez les déclarer si vous le souhaitez.

```
def pair(n: int) -> bool:
    return n % 2 == 0
```

Le lecteur intéressé trouvera sans mal des tutoriels et autres guides concernant la programmation Python. Un manuel recommandé est Programmation avec Python: des jeux au Web [1] par Godin et Lemire.

1. https://www.amazon.ca/dp/B0CVX9296P/

Pour installer Python sur Windows, visitez le site officiel de Python (python.org) et téléchargez le dernier programme d'installation de Python pour Windows. Exécutez le fichier exécutable en veillant à cocher la case «Add Python to PATH» (ajouter Python au PATH) au cours de la procédure d'installation. Suivez les instructions pour terminer l'installation. Une fois l'installation terminée, ouvrez l'invite de commande et tapez `python --version` pour vérifier l'installation. Si le numéro de version apparaît, Python est installé avec succès et prêt à être utilisé. Il est fourni avec le gestionnaire de paquets `pip`.

Pour installer Python sur macOS, le processus est similaire. Allez sur python.org et téléchargez le dernier programme d'installation de Python pour macOS. Ouvrez le fichier .pkg téléchargé et suivez les instructions du programme d'installation, qui impliquent généralement d'accepter la licence et de sélectionner l'emplacement d'installation (les paramètres par défaut conviennent généralement). Après l'installation, ouvrez le Terminal et tapez `python3 --version` pour confirmer que Python est installé correctement. Le gestionnaire de paquets correspondant `pip3` devrait également avoir été installé.

Remerciements

J'aimerais remercier Robert Godin et Wojciech Muła pour leurs commentaires critiques.

Chapitre 1

Notre objectif le plus important en écrivant du logiciel est qu'il soit correct. Le logiciel doit accomplir ce que le programmeur souhaite accomplir. Il doit rencontrer les besoins de l'utilisateur.

Dans le monde des affaires, la comptabilité par partie double [2] est l'idée selon laquelle les opérations sont inscrites dans deux comptes au moins (débit et crédit). Un des avantages à la comptabilité par partie double, en comparaison à une approche plus naïve, est qu'elle permet dans une certaine mesure de vérifier les comptes et de trouver les erreurs. Si on compare la comptabilité à la programmation logicielle, nous pourrions dire que la comptabilité par partie double et sa vérification subséquente équivalent à la mise en place de tests logiciels.

Pour un comptable, la conversion d'une comptabilité naïve en système par partie double est une tâche difficile en général. Dans bien des cas, il faudrait tout refaire. Dans le même sens, il peut être difficile d'ajouter des tests à une application de grande envergure qui a été développée entièrement sans des tests. Et c'est pour cette raison que je fais du premier chapitre de ce manuel un chapitre sur les tests.

Un programmeur pressé ou débutant peut écrire rapidement une routine, la compiler ou l'exécuter et se satisfaire du résultat. Un programmeur prudent ou expérimenté saura qu'il ne faut pas prendre pour acquis que cette routine est correcte.

2. https://fr.wikipedia.org/wiki/Comptabilite_en_partie_double

Les erreurs logicielles communes peuvent causer des problèmes allant du programme qui se termine de manière soudaine à la corruption d'une base de données. Les conséquences peuvent être coûteuses: un bogue logiciel causa l'explosion d'une fusée Ariane 5 en 1996 (Dowson, 1997 [3]). L'erreur fut causée par le conversion d'un nombre à virgule flottante vers un entier signé représenté avec 16 bits. Seules les petites valeurs entières pouvaient être représentées. Or comme la valeur ne pouvait pas être représentée, une erreur fut détectée et le logiciel s'arrêta car une telle erreur était imprévue. L'ironie est que la fonction qui a déclenchée l'erreur n'était pas requise: elle avait simplement été intégrée comme sous-système datant d'un modèle précédent de la fusée Ariane. En dollars Américains de 1996, le coût estimé de cette erreur est de presque 400 millions de dollars.

L'importance de produire du logiciel correct est comprise depuis longtemps. Les meilleurs scientifiques et ingénieurs tentent d'y arriver depuis des décennies.

Il existe plusieurs stratégies communes. Par exemple, s'il s'agit de faire un calcul scientifique complexe, nous pourrons alors demander à plusieurs équipes indépendantes de produire une réponse. Si toutes les équipes arrivent à la même réponse, nous pourrons alors conclure qu'elle est correcte. Une telle redondance est souvent utilisée pour prévenir les failles liées au matériel (Yeh, 1996 [4]). Malheureusement, il est peu pratique d'écrire plusieurs versions d'un logiciel logiciel en général.

Plusieurs des premiers programmeurs avaient une formation avancée en mathématiques. Ils espéraient que nous pourrions prouver qu'un logiciel est correct. En ignorant la possibilité que le matériel soit en cause, nous pourrions alors être certain de ne pas rencontrer d'erreurs. Et en effet, nous disposons aujourd'hui de logiciels sophistiqués qui nous permettent de parfois prouver qu'un programme est correct.

Considérons un exemple de vérification formelle afin d'illustrer notre propos. Nous pouvons utiliser la librairie z3 à partir de Python (De

3. https://doi.org/10.1145/251880.251992
4. https://doi.org/10.1109/AERO.1996.495891

Moura et Bjørner, 2008 [5]). Si vous n'êtes pas un utilisateur de Python, ne vous en faites pas: vous n'avez pas à l'être pour suivre l'exemple. On peut installer la librairie nécessaire avec la commande `pip install z3-solver` ou l'équivalent. Supposons que nous voulions être certain que l'inégalité `( 1 + y ) / 2 < y` est vérifiée pour tous les entiers de 32 bits. Nous pouvons utiliser le script suivant:

```
import z3
y = z3.BitVec("y", 32)
s = z3.Solver()
s.add( ( 1 + y ) / 2 >= y )
if(s.check() == z3.sat):
    model = s.model()
    print(model)
```

Nous construisons dans cet exemple un mot (*BitVec*) de 32 bits pour représenter notre exemple. Par défaut, la librairie z3 interprète les valeurs pouvant être représentés par une telle variable comme allant de -2147483648 à 2147483647 (de -2^{31} à $2^{31} - 1$ inclusivement). Nous saisissons l'inégalité contraire à celle que nous souhaitons montrer (`( 1 + y ) / 2 >= y`). Si z3 ne trouve pas de contre-exemple, nous saurons alors que l'inégalité `( 1 + y ) / 2 < y` est vérifiée.

En exécutant le script, Python affiche la valeur entière 2863038463 ce qui indique que z3 a trouvé un contre-exemple. La librairie z3 donne toujours un entier positif et c'est à nous de l'interpréter correctement. Le nombre 2147483648 devient -2147483648, le nombre 2147483649 devient -2147483647 et ainsi de suite. Cette représentation est souvent appelée le complément à deux [6]. Nous reviendrons sur la représentation des entiers au chapitre 3. Ainsi, le nombre 2863038463 est en fait interprété comme un nombre négatif. Sa valeur exacte est peu importante: ce qui importe est que notre inégalité (`( 1 + y ) / 2 < y`) est incorrecte lorsque la variable est négative. Nous pouvons le vérifier en donnant à la variable la valeur -1, nous obtenons alors `0 < -1`. Lorsque la variable prend la

5. https://doi.org/10.1007/978-3-540-78800-3_24
6. https://fr.wikipedia.org/wiki/Complement_a_deux

valeur 0, l'inégalité est aussi fausse (`0<0`). Nous pouvons aussi vérifier que l'inégalité est fausse lorsque la variable prend la valeur 1. Ajoutons donc comme condition que la variable est plus grande que 1 (`s.add( y > 1 )`):

```
import z3
y = z3.BitVec("y", 32)
s = z3.Solver()
s.add( ( 1 + y ) / 2 >= y )
s.add( y > 1 )

if(s.check() == z3.sat):
    model = s.model()
    print(model)
```

Puisque ce dernier script n'affiche rien à l'écran lors de son exécution, nous pouvons conclure que l'inégalité est satisfaite tant que la variable de variable est plus grande que 1.

Puisque nous avons montré que l'inégalité `( 1 + y ) / 2 < y` est vraie, peut-être que l'inégalité `( 1 + y )  < 2 * y` est vraie aussi? Essayons:

```
import z3
y = z3.BitVec("y", 32)
s = z3.Solver()
s.add( ( 1 + y ) >= 2 * y )
s.add( y > 1 )

if(s.check() == z3.sat):
    model = s.model()
    print(model)
```

Ce script affichera 1412098654, la moitié de 2824197308 qui est interprété par z3 comme une valeur négative. Pour éviter ce problème, ajoutons une nouvelle condition afin que le double de la variable puisse continuer d'être interprété comme une valeur positive:

```
import z3
y = z3.BitVec("y", 32)
s = z3.Solver()
s.add( ( 1 + y ) / 2 >= y )
s.add( y > 0 )
s.add( y <  2147483647/2)

if(s.check() == z3.sat):
    model = s.model()
    print(model)
```

Cette fois-ci, le résultat est vérifié. Comme vous pouvez le constater, une telle approche formelle exige beaucoup de travail, même dans des cas relativement simples. Il était peut-être possible d'être plus optimiste au début de l'informatique, mais dès les années 1970, des informaticiens comme Dijkstra émettaient des doutes:

> we see automatic program verifiers verifying toy programs and one observes the honest expectation that with faster machines with lots of concurrent processing, the life-size problems will come within reach as well. But, honest as these expectations may be, are they justified? I sometimes wonder... (Dijkstra, 1975 [7])

Il est peu pratique d'appliquer une telle méthode mathématique à grande échelle. Les erreurs peuvent prendre plusieurs formes, et toutes ces erreurs ne peuvent être présentées de manière concise sous une forme mathématique. Même quand cela est possible, même quand nous pouvons représenter précisément le problème sous une forme mathématique, rien ne nous permet de croire qu'un outil comme z3 pourra toujours trouver une solution: quand les problèmes deviennent difficiles, les temps de calcul peuvent devenir très longs. Une approche empirique est plus appropriée en général.

7. https://doi.org/10.1145/800027.808478

Au fil du temps, les programmeurs ont compris la nécessité de tester leur logiciel. Il n'est pas toujours nécessaire de tout tester: un prototype ou un exemple peut souvent être fourni sans plus de validation. Par contre, tout logiciel conçu dans un contexte professionnel et devant remplir une fonction importante devrait être au moins en partie testé. Les tests nous permettent de réduire la probabilité que nous devions faire face à une situation désastreuse.

On distingue généralement deux grandes catégories de tests.

— Il y a d'une part les tests unitaires. Ceux-ci visent à vérifier une composantes particulière d'un logiciel. Par exemple, un test unitaire peut porter sur une seule fonction. Le plus souvent, les tests unitaires sont automatisés: le programmeur peut les exécuter en pressant un bouton ou en tapant une commande. Au sein des tests unitaires, on évite souvent l'acquisition de ressources précieuses, comme la création de fichiers volumineux sur un disque ou des branchements au réseau. Lors d'un test unitaire, il n'y aura généralement pas une reconfiguration du système d'exploitation. Les tests unitaires sont souvent dérivés des attentes des programmeurs.
— Quant à eux, les tests d'intégration visent à valider une application complète. Ils requièrent souvent des accès aux réseaux et des accès à des données parfois volumineuses. Ils nécessitent parfois une intervention manuelle et une connaissance spécifique de l'application. Les tests d'intégration peuvent comprendre une reconfiguration du système d'exploitation et l'installation de logiciel. Ils peuvent aussi être automatisés, au moins en partie. Le plus souvent, les tests d'intégration prennent appui sur les tests unitaires qui servent de fondation. Ils peuvent valider plusieurs besoins tels quel la sécurité ou le respect des normes environnementales. En comparaison avec les tests unitaires, les tests d'intégration sont souvent plus proches des besoins des utilisateurs.

Les tests unitaires font souvent partie d'un processus continu d'intégration (Kaiser et al., 1989[8]). L'intégration continue exécute souvent

8. https://doi.org/10.1109%2FCMPSAC.1989.65147

automatiquement des tâches spécifiques comprenant les tests unitaires, des copies de sauvegarde, l'application de signatures cryptographiques, et ainsi de suite. L'intégration continue peut se faire à intervalles réguliers, ou à chaque fois qu'une modification est apportée au code.

Les tests unitaires permettent de structurer et de guider le développement logiciel. Les tests peuvent être écrits avant le code lui-même, nous parlons alors de *test-driven development*. Souvent, les tests sont écrits après avoir développé les fonctions. Les tests peuvent être écrits par d'autres programmeurs que ceux qui ont développé les fonctions. Il est parfois plus facile pour des développeurs indépendants de fournir des tests capables de mettre à jour les erreurs parce qu'ils ne partagent pas les mêmes a priori.

Il est possible d'intégrer les tests à même des fonctions ou une application. Par exemple, une application peut, lorsqu'elle se lance, effectuer quelques tests. Dans un tel cas, les tests feront partie du code diffusé. Néanmoins, il est plus commun de ne pas publier les tests unitaires. Ils représentent une composante réservée aux programmeurs et ils n'affectent pas le fonctionnement de l'application. En particulier, ils ne posent donc pas de risque de sécurité et ils ne nuisent pas à la performance de l'application.

Les programmeurs d'expérience considèrent souvent les tests comme ayant autant d'importance que le code original. Il n'est donc pas rare de passer la moitié de son temps sur l'écriture de tests. L'effet net est de réduire substantiellement la vitesse initiale de rédaction du code informatique. Par contre, cette perte de temps apparente permet souvent de gagner du temps à terme: la mise sur pied de tests est un investissement. Un logiciel peu testé est souvent plus difficile à mettre à jour. La présence des tests nous permet de faire des modifications ou des extensions avec moins d'incertitude.

Les tests devraient être lisibles, simples et ils devraient s'exécuter rapidement. Ils utilisent souvent peu de mémoire.

Malheureusement, il est difficile de définir exactement la qualité des tests. Il existe plusieurs mesures statistiques. Par exemple, nous pouvons

compter les lignes de code qui s'exécutent au cours de tests. Nous parlons alors de la couverture des tests. Une couverture de 100% implique que toutes les lignes de code sont concernés par les tests. En pratique, cette mesure de couverture peut être une piètre indication de la qualité des tests.

Considérons cet exemple [9]:

```
package main

import (
  "testing"
)

func Moyenne(x, y uint16) uint16 {
   return (x + y)/2
}

func TestMoyenne(t *testing.T) {
    if Moyenne(2,4) != 3 {
        t.Error(Moyenne(2,4))
    }
}
```

Notre test vérifie simplement que la moyenne de 2 et 4 est 3.

Dans le langage Go, nous pouvons exécuter les tests avec la commande `go test`. Il faut nommer les fichiers contenant des tests avec le suffixe `_test.go` (par ex., `mycode_test.go`), les functions de test doivent avoir le préfixe `Test` (par ex., `TestMoyenne`) qui prennent en paramètre une valeur de type `*testing.T`: en cas d'erreur, il faut appeler la fonction `t.Error` où `t` est de type `*testing.T`. Nous avons une fonction `Moyenne` accompagnée d'un test correspondant. Dans notre exemple, le test s'exécutera avec succès. Si vous avez nommé le fichier `moyenne_test.go`, vous pouvez exécuter les tests avec la commande `go test average_test.go`. La

9. https://play.golang.org/p/nwMUq2o_WlX

couverture est de 100%.

Malheureusement, la fonction `Moyenne` n'est peut-être pas aussi correcte que nous puissions l'espérer. En effet, si nous passons en paramètre les valeurs entières 40000 et 40000, nous pourrions nous attendre à ce que la valeur moyenne de 40000 soit retournée. Or l'entier 40000 ajouté à l'entier 40000 ne peut pas être représenté avec un entier de 16 bits (`uint16`): le résultat sera plutôt `(40000+4000)%65536=14464`. Ainsi donc la fonction retournera 7232 ce qui pourra être surprenant. Le test suivant échouera:

```
func TestMoyenne(t *testing.T) {
    if Moyenne(40000,40000) != 40000 {
        t.Error(Moyenne(40000,40000))
    }
}
```

Il est facile en Go d'obtenir automatiquement un rapport de couverture. Il est plus pratique de créer un module d'abord. Dans un nouveau dossier, créez les trois fichiers suivants.

go.mod:

```
module lemire.me/average

go 1.24.4
```

average.go:

```
package average

func Average(x, y uint16) uint16 {
  return (x + y) / 2
}

func Average2(x, y uint16) uint16 {
  return (x + y) / 2
}
```

average_test.go:

```
package average

import (
  "testing"
)

func TestAverage(t *testing.T) {
  if Average(2, 4) != 3 {
    t.Error(Average(2, 4))
  }
}
```

Alors que vous êtes dans le dossier nouvelle créé, tapez les commandes suivantes.

```
go test -coverprofile=coverage.out
go tool cover -func=coverage.out
```

Go vous informera alors que la couverture est de 50%. En effet, la fonction `Average2` n'est pas testée.

La couverture de code est utile pour identifier les parties d'un programme non atteintes par les tests. Dans le cas d'un code que vous ne connaissez pas, elle offre une vue d'ensemble sur ce qui se passe. Une absence de couverture peut signaler du code mort (qui ne sert à rien).

Quand cela est possible et rapide, nous pouvons tenter de tester de manière plus exhaustive le code, comme dans cet exemple [10] où nous incluons plusieurs valeurs:

```
package main

import (
  "testing"
```

10. https://play.golang.org/p/nlq_J_-Tw8F

```
)

func Moyenne(x, y uint16) uint16 {
   if y > x {
     return (y - x)/2 + x
   } else {
     return (x - y)/2 + y
   }
}

func TestMoyenne(t *testing.T) {
  for x := 0; x < 65536; x++ {
    for y := 0; y < 65536; y++ {
      m := int(Moyenne(uint16(x),uint16(y)))
      if x < y {
        if m < x || m > y {
          t.Error("erreur ", x, " ", y)
        }
      } else {
        if m < y || m > x {
          t.Error("erreur ", x, " ", y)
        }
      }
    }
  }
}
```

Avec un peu de patience, il est possible tester des milliards de cas. Par exemple, le test suivant vérifie qu'il est possible de convertir tous les entiers de `math.MinInt32` à `math.MaxInt32` en chaîne de caractères (avec `strconv.Itoa`) pour ensuite récupérer l'entier original à partir de la chaîne produire (`strconv.ParseInt`).

```
package int32test
```

```
import (
  "math"
  "strconv"
  "testing"
)

func TestInt32StringConversion(t *testing.T) {
  for i := int64(math.MinInt32); i <= math.MaxInt32; i++ {
    s := strconv.Itoa(int(i))
    parsed, err := strconv.ParseInt(s, 10, 32)
    if err != nil {
      t.Errorf("Erreur de parsing pour %d: %v", i, err)
      continue
    }
    if parsed != i {
      t.Errorf("Pour %d, obtenu %d après conversion",
        i, parsed)
    }
  }
}
```

Si vous nommez ce fichier `exhaustive_test.go`, vous devriez pouvoir exécuter `go test exhaustive_test.go` en quelques minutes.

En pratique, il est rare que nous puissions faire des tests exhaustifs. Nous pouvons utiliser plutôt des tests pseudo-aléatoires. Par exemple, nous pouvons générer des nombres pseudo-aléatoires et les utiliser comme paramètres. Nous reviendrons sur les nombres pseudo-aléatoires au chapitre 5. Dans le cas des tests aléatoires, il est important de faire en sorte que ceux-ci demeurent déterministes: il faut qu'à chaque fois que le test s'exécute, les mêmes valeurs sont testées. On peut obtenir ce résultat en fournissant une *semence* fixe au générateur de nombre aléatoires comme dans cet exemple [11]:

11. https://play.golang.org/p/XGoxJoxfiEJ

```
package main

import (
  "testing"
  "math/rand"
)

func Moyenne(x, y uint16) uint16 {
   if y > x {
     return (y - x)/2 + x
   } else {
     return (x - y)/2 + y
   }
}

func TestMoyenne(t *testing.T) {
  rand.Seed(1234)
  for test := 0; test < 1000; test++ {
    x := rand.Intn(65536)
    y := rand.Intn(65536)
    m := int(Moyenne(uint16(x),uint16(y)))
    if x < y {
      if m < x || m > y {
        t.Error("erreur ", x, " ", y)
      }
    } else {
      if m < y || m > x {
        t.Error("erreur ", x, " ", y)
      }
    }
  }
}
```

Les tests basés sur une exploration aléatoire relèvent d'une stratégie

souvent appelée *fuzzing* (Miller at al., 1990 [12]).

Le langage Go vous permet de faire du fuzzing automatiquement, sans avoir à générer vos propres valeurs aléatoires:

```go
// go test -fuzz=FuzzMoyenne
package main

import (
  "testing"
)

func Moyenne(x, y uint16) uint16 {
  if y > x {
    return (y-x)/2 + x
  } else {
    return (x-y)/2 + y
  }
}

func FuzzMoyenne(f *testing.F) {
  f.Fuzz(func(t *testing.T, x uint16, y uint16) {
    m := Moyenne(x, y)
    if x < y {
      if m < x || m > y {
        t.Errorf("error with x=%d, y=%d", x, y)
      }
    } else {
      if m < y || m > x {
        t.Errorf("error with x=%d, y=%d", x, y)
      }
    }
  })
}
```

12. https://doi.org/10.1145/96267.96279

Nous distinguons généralement deux types de tests. Les tests positifs visent à vérifier qu'une fonction ou une composante se comporte de manière convenue. Ainsi, le premier test de notre fonction `Moyenne` était un test positif. Les tests négatifs vérifient que le logiciel se comporte correctement même dans des situations inattendues. Nous pouvons produire des tests négatifs en fournissant à nos fonctions des données aléatoires (*fuzzing*). Notre second exemple peut être considéré comme un test négatif si le programmeur s'attendait à de petites valeurs entières.

Les tests devraient échouer lorsque le code est modifié (Budd et al., 1978 [13]). Sur cette base, nous pouvons aussi développer des mesures plus sophistiquées en testant des modifications aléatoire du code et en s'assurant que de telles modifications causent souvent un échec des tests.

Certains programmeurs choisissent de générer les tests automatiquement à partir du code. Dans un tel cas, une composante est testée et le résultat est capturé. Par exemple, dans notre exemple du calcul de la moyenne, nous aurions pu capturer le fait que `Moyenne(40000,40000)` a comme valeur 7232. Si un changement subséquent vient changer le résultat de l'opération, le test échouera. Une telle approche permet de gagner du temps puisque les tests sont produits automatiquement. Nous pouvons arriver rapidement et sans effort à une couverture de 100% du code. Par contre, de tels tests peuvent nous induire en erreur. Il est notamment possible de capturer un comportement incorrect. Par ailleurs, l'importance avec les tests n'est pas tant leur nombre que leur qualité. La présence de plusieurs tests qui ne contribuent pas à valider les fonctions essentielles de notre logiciel peuvent même devenir nuisible. Des tests impertinents peuvent faire perdre du temps aux programmeurs lors de révisions subséquentes.

Concernant les tests d'intégration, les nombreux systèmes externes dont ils dépendent souvent, tels que les bases de données, les services réseau ou les API tierces, peuvent fréquemment être simulés à l'aide de faux services (aussi appelé *mocks* ou *stubs*). Dans ce contexte, un faux service est une version simulée d'un composant externe qui expose la même API que le système réel, mais permet à l'auteur du test de contrôler

13. https://doi.org/10.1109/AFIPS.1978.195

directement son comportement au sein du test. Cela permet de tester les interactions de l'application avec ces systèmes sans dépendre de leur disponibilité réelle, ce qui peut être coûteux, lent ou sujet à des défaillances externes. La capacité à utiliser efficacement les faux services repose sur une architecture système bien conçue, en particulier une qui utilise l'injection de dépendances pour fournir des services plutôt que de coder en dur les dépendances, garantissant ainsi que les composants sont faiblement couplés et facilement remplaçables.

L'injection de dépendances est un patron de conception en ingénierie logicielle qui favorise un couplage faible entre les composants en passant les dépendances (services ou objets dont un composant a besoin) à ce composant, plutôt que de laisser le composant les créer ou y faire référence directement. Cette approche rend le système plus modulaire, testable et maintenable, car les dépendances peuvent être facilement échangées ou simulées pendant les tests sans modifier le code du composant. Au lieu de coder en dur les dépendances (en les intégrant), elles sont « injectées » de l'extérieur, généralement via des constructeurs, des setters ou des interfaces.

Par exemple, considérons une application qui s'intègre à un service de traitement des paiements. Dans un scénario réel, l'application envoie des requêtes à l'API du service pour traiter les transactions. Lors des tests d'intégration, se connecter au véritable service de paiement peut être peu pratique en raison des coûts, des problèmes de réseau ou du besoin de scénarios de test spécifiques (par exemple, simuler un paiement échoué). À la place, un faux service peut être créé pour imiter l'API du service de paiement. Le faux service répondrait aux appels d'API de manière contrôlée, comme défini par le test. Par exemple, un test pourrait configurer le faux service pour retourner une réponse « paiement réussi » pour une requête valide ou une erreur « paiement refusé » pour un numéro de carte invalide. Cela permet au test de vérifier comment l'application gère ces réponses sans jamais contacter le véritable service de paiement.

Supposons que nous ayons une application qui traite des commandes et

interagit avec un service de paiement. Nous voulons tester la logique de traitement des commandes sans appeler le véritable service de paiement, en utilisant un faux service à la place.

```
package main

import (
  "fmt"
)

type PaymentService interface {
  ProcessPayment(orderID string, amount float64)
  (string, error)
}

type RealPaymentService struct{}

func (r *RealPaymentService) ProcessPayment(orderID string,
  amount float64) (string, error) {
  // Simule l'appel à une passerelle de paiement externe.
  return "tx_12345", nil
}

type MockPaymentService struct {
  // Champs pour contrôler le comportement du mock.
  transactionID string
  err           error
}

func (m *MockPaymentService) ProcessPayment(orderID string,
  amount float64) (string, error) {
  // Retourne des valeurs prédéfinies pour les tests.
  return m.transactionID, m.err
}
```

```go
type OrderProcessor struct {
  paymentService PaymentService
}

func NewOrderProcessor(paymentService PaymentService)
  *OrderProcessor {
  return &OrderProcessor{
    paymentService: paymentService,
  }
}

func (op *OrderProcessor) ProcessOrder(orderID string,
  amount float64) (string, error) {
  transactionID, err
    := op.paymentService.ProcessPayment(orderID, amount)
  if err != nil {
    return "", fmt.Errorf("échec du paiement : %w", err)
  }
  return transactionID, nil
}

func main() {
  // Exemple d'utilisation dans une application réelle.
  realPaymentService := &RealPaymentService{}
  processor := NewOrderProcessor(realPaymentService)
  transactionID, err := processor.ProcessOrder("order_001",
    99.99)
  if err != nil {
    fmt.Printf("Erreur : %v\n", err)
    return
  }
  fmt.Printf("ID de transaction du service réel : %s\n",
    transactionID)
```

```
  mockPaymentService := &MockPaymentService{
    transactionID: "mock_tx_999",
    err:           nil,
  }
  processor = NewOrderProcessor(mockPaymentService)
  transactionID, err = processor.ProcessOrder("order_002",
    49.99)
  if err != nil {
    fmt.Printf("Erreur : %v\n", err)
    return
  }
  fmt.Printf("ID de transaction du service simulé : %s\n",
    transactionID)
}
```

Ce code Go illustre l'injection de dépendances pour faciliter les tests d'intégration en permettant à un composant, `OrderProcessor`, de fonctionner avec une implémentation réelle ou simulée d'un `PaymentService`. L'interface `PaymentService` définit une méthode `ProcessPayment`, qui est implémentée par deux structures : `RealPaymentService`, simulant une interaction avec une passerelle de paiement externe, et `MockPaymentService`, utilisée pour les tests avec des réponses configurables. La structure `OrderProcessor` dépend d'un `PaymentService`, qui est injecté via le constructeur `NewOrderProcessor`, garantissant un couplage faible. La méthode `ProcessOrder` appelle la méthode `ProcessPayment` du service injecté et gère sa réponse, retournant un ID de transaction ou une erreur. Cette conception permet à `OrderProcessor` de rester agnostique quant à savoir s'il interagit avec un service réel ou simulé, améliorant la modularité et la testabilité.

Dans la fonction `main`, le code montre à la fois des scénarios réels et de test. D'abord, il crée un `RealPaymentService`, l'injecte dans un `OrderProcessor`, et traite une commande, affichant un ID de transaction (par exemple, `tx_12345`). Ensuite, il démontre un scénario de test en injectant un `MockPaymentService`, configuré avec un `transactionID`

prédéfini (`mock_tx_999`) et sans erreur. Le `OrderProcessor` traite une autre commande, affichant l'identifiant de transaction du faux service. Les champs du faux service permettent aux auteurs de tests de simuler divers résultats (par exemple, succès ou échec) sans dépendre d'un système externe. Cette structure illustre comment l'injection de dépendances permet un échange fluide des dépendances, rendant les tests d'intégration plus rapides, plus fiables et indépendants des services externes.

Les tests basés sur des tables permettent de vérifier une fonction en utilisant un ensemble prédéfini d'entrées et de sorties attendues, regroupés dans une structure de données, généralement une tranche de structures. Cette approche est particulièrement efficace pour tester une fonction avec plusieurs cas de test, comme une chaîne vide, zéro ou une valeur maximale, en réduisant la duplication de code. Dans l'exemple suivant, nous testons une fonction qui calcule le carré d'un nombre entier, en validant son comportement avec des entrées variées, y compris une chaîne vide convertie en entier, zéro et une valeur qui excède la valeur maximale d'un entier.

```
package main

import (
  "strconv"
  "testing"
)

func square(n int64) int64 {
  return n * n
}

func TestSquare(t *testing.T) {
  tests := []struct {
    name     string
    input    string
    expected int64
    err      bool
```

```
}{
  {
    name:     "empty string",
    input:    "",
    expected: 0,
    err:      true,
  },
  {
    name:     "zero",
    input:    "0",
    expected: 0,
    err:      false,
  },
  {
    name:     "positive number",
    input:    "5",
    expected: 25,
    err:      false,
  },
  {
    name:     "negative number",
    input:    "-4",
    expected: 16,
    err:      false,
  },
  {
    name:     "max int64",
    input:    "9223372036854775808",
    expected: 0,
    err:      true,
  },
}

for _, tt := range tests {
```

```
    t.Run(tt.name, func(t *testing.T) {
      n, err := strconv.ParseInt(tt.input, 10, 64)
      if tt.err {
        if err == nil {
          t.Errorf("expected error for input %q, got none",
            tt.input)
        }
        return
      }
      if err != nil {
        t.Errorf("unexpected error for input %q: %v",
          tt.input, err)
        return
      }
      result := square(n)
      if result != tt.expected {
        t.Errorf("square(%d) = %d; want %d", n, result,
          tt.expected)
      }
    })
  }
}
```

À mesure que le code évolue, il est souhaitable d'exécuter les tests fréquemment, parfois même après chaque modification, aussi minime soit-elle. Une régression logicielle désigne un défaut où une fonctionnalité, précédemment opérationnelle, cesse de fonctionner correctement. Dans certains cas, il peut être nécessaire de lancer l'ensemble des tests, y compris les tests unitaires et d'intégration. Ce processus est communément appelé test de régression.

En terminant, nous pouvons passer en revue les bénéfices des tests: les tests nous aident dans l'organisation du travail, ils sont une mesure de qualité, ils nous aident à documenter le code, ils évitent la régression, ils aident au déboguage et ils peuvent produire du code plus efficace.

Organisation

La conception d'un logiciel sophistiqué peut exiger des semaines ou des mois de travail. Le plus souvent, le travail sera décomposé en unités distinctes. Il peut être difficile, avant d'avoir le produit définitif, de juger du résultat. L'écriture de tests en même temps que nous développons le logiciel permet d'organiser le travail. Par exemple, une composante donnée pourra être considérée comme complétée quand elle est écrite et testée. Sans le processus d'écriture de tests, il est plus difficile d'estimer l'état d'avancement d'un projet puisqu'une composante non testée peut être encore loin d'être terminée.

Qualité

Les tests servent aussi à rendre tangible le soin que le programmeur a mis dans son travail. Ils permettent aussi d'évaluer rapidement le soin apporté aux différentes fonctions et composantes d'un logiciel: la présence de tests soigneusement composés pourra être une indication que le code correspondant est fiable. En contrepartie, l'absence de tests concernant certains fonctions peut servir de mise en garde.

Certains langages de programmation sont assez stricts et ils comportent une phase de compilation qui valide le code. D'autres langages de programmation (Python, JavaScript) laissent plus de liberté au programmeur. Certains programmeurs considèrent que les tests peuvent aider à combler les limites des langages de programmation moins stricts en imposant au programmeur une rigueur que le langage n'exige pas.

Une approche connexe est l'utilisation de *linters*. Ce sont des outils qui examinent le code pour détecter des erreurs potentielles. Bien qu'ils ne constituent pas une forme de test à proprement parler, ils peuvent aider à identifier des erreurs. Le langage Go dispose d'un linter intégré que l'on peut invoquer avec la commande `go vet`. Considérez le code Go suivant :

```
package main
import (
```

```
"fmt"
)
func main() {
num := 42
str := string(num) // strconv.Itoa
fmt.Print(str)
name := "Alice"
fmt.Printf("%s %d\n", name)
}
```

Ce code contient probablement deux erreurs. La première erreur probable est que, pour convertir un nombre en chaîne, la fonction appropriée est `strconv.Itoa` et non `string`. L'expression `string(42)` retourne le caractère ASCII correspondant au point de code 42, qui est `*`. La deuxième erreur est que l'expression `fmt.Printf("%s %d\n", name)` devrait avoir un troisième paramètre de type entier. L'exécution de `go vet` dans un répertoire contenant ce code produit les avertissements suivants :

```
./bad.go:13:2: le format %d de fmt.Printf
attend un 2e argument,
mais l'appel n'en a qu'un
./bad.go:9:9: la conversion d'un int en
string produit une chaîne
d'une seule rune, pas une chaîne de chiffres
```

Il est courant d'exécuter automatiquement des linters comme `go vet` pour assurer un contrôle de qualité. Cependant, il faut être prudent, car ces outils ont tendance à produire des faux positifs (signaler des erreurs qui n'en sont pas).

Documentation

La programmation logicielle devrait généralement s'accompagner de la rédaction d'une documentation claire et complète. En pratique, la documentation est souvent partielle, imprécise, erronée ou même inexistante.

Les tests offrent donc souvent la seule spécification technique disponible. La lecture des tests permettent aux programmeurs d'ajuster leurs attentes face aux composantes et fonctions logicielles. Contrairement à la documentation, les tests sont généralement à jour, s'ils sont exécutés régulièrement, et ils sont précis dans la mesure où ils sont rédigés dans un langage de programmation. Les tests peuvent donc offrir de bons exemples de l'utilisation du code.

Par ailleurs, même dans le cas où nous souhaitons rédiger une documentation de grande qualité, les tests peuvent aussi jouer un rôle important. Pour illustrer du code informatique, on utilise souvent des exemples. Chaque exemple peut être transformé en test. Nous pouvons donc nous assurer que les exemples inclus dans la documentation sont fiables. Quand le code change, et qu'il nécessite de modifier les exemples, une procédure visant à tester nos exemples nous rappelera de mettre à jour notre documentation. Nous évitons donc ainsi aux lecteurs de notre documentation l'expérience frustrante consistant à prendre connaissance d'exemples qui ne sont plus fonctionnels.

Go prend en charge les exemples comme tests. Dans un fichier contenant des tests, vous pouvez ajouter une fonction commençant par la chaîne `Example` et incluant le nom de la fonction que nous souhaitons documenter. La fonction d'exemple ne doit prendre aucun paramètre. La fonction doit se terminer par un commentaire commençant par `Output:`. Elle devient automatiquement un test. Lors de l'exécution des tests, la fonction est exécutée et le résultat affiché par la fonction est comparé au commentaire commençant par `Output:`. Le code suivant illustre ce concept avec une fonction appelée `Sum` et une fonction d'exemple appelée `ExampleSum`. Lors de l'exécution des tests, la fonction `ExampleSum()` est appelée, elle affiche `3`, ce qui correspond au commentaire (`// Output: 3`), et ainsi, le test réussit.

```
package main

import (
  "fmt"
```

```
)

func Sum(x, y int) int {
  return x + y
}

func ExampleSum() {
  fmt.Println(Sum(1, 2))
  // Output: 3
}
```

Régression

Les programmeurs corrigent régulièrement des failles au sein de leurs logiciels. Il arrive souvent ensuite que le même problème se présente à nouveau. Un même problème peut revenir pour diverses raisons: parfois le problème initial n'a pas été complètement corrigé. Parfois, une autre modification ailleurs dans le code entraîne le retour de l'erreur. Il arrive que l'ajout d'une nouvelle fonction ou d'une optimisation logicielle cause le retour d'un bogue, ou l'ajout d'un nouveau bogue. Quand un logiciel acquiert une nouvelle faille, on parle alors d'une régression. Afin d'empêcher de telles régressions, il est important d'accompagner toute correction d'une erreur ou toute nouvelle fonction d'un test correspondant. De cette manière, nous pouvons prendre connaissance rapidement des régressions en exécutant les tests. Idéalement, la régression pourra être identifiée alors même que le code est modifié, nous évitons donc la régression. Afin de convertir un bogue en test simple et efficace, il est utile de le réduire à sa plus simple expression. Par exemple, dans notre exemple précédent avec `Moyenne(40000,40000)`, nous pouvons ajouter l'erreur détectée en test supplémentaire [14]:

14. https://play.golang.org/p/PH9y3ZqV2c9

```
package main

import (
  "testing"
)

func Moyenne(x, y uint16) uint16 {
   if y > x {
     return (y - x)/2 + x
   } else {
     return (x - y)/2 + y
   }
}

func TestMoyenne(t *testing.T) {
   if Moyenne(2,4) != 3 {
     t.Error("erreur 1")
   }
   if Moyenne(40000,40000) != 40000 {
     t.Error("erreur 2")
   }
}
```

Correction des bogues

En pratique, la présence d'une batterie de tests étendue permet d'identifier et de corriger les bogues plus rapidement. En effet, les tests permettent de réduire l'ampleur des erreurs et de fournir au programmeur plusieurs garanties. Dans une certaine mesure, le temps passé à écrire des tests permet de gagner du temps lorsque des erreurs sont trouvées tout en réduisant le nombre d'erreurs. En outre, une stratégie efficace pour identifier et corriger un bogue consiste à écrire de nouveaux tests. Cette stratégie peut s'avérer plus efficace à long terme que d'autres stratégies

de débogage telles que l'exploration du code. En effet, une fois votre session de débogage terminée, vous vous retrouvez avec de nouveaux tests unitaires en plus d'un bug corrigé.

Performance

La fonction principale des tests est de vérifier que les fonctions et composantes produisent les résultats attendus. Par contre, les programmeurs utilisent de plus en plus les tests afin de mesurer la performance des composantes. Par exemple, la vitesse d'exécution d'une fonction, la taille de l'exécutable ou l'utilisation de la mémoire peuvent être mesurées. Il est alors possible de détecter une perte de performance suite à une modification du code. Vous pouvez comparer les performances de votre code avec un code de référence et vérifier les différences à l'aide de tests statistiques.

Conclusion

Tous les systèmes informatiques ont des failles. Le matériel peut en tout temps avoir des défaillances. Et même lorsque le matériel est fiable, il est quasiment impossible pour un programmeur de prévoir toutes les conditions dans lesquelles le logiciel sera utilisé. Peu importe qui vous êtes, et peu importe l'attention que vous portez à votre travail, votre logiciel ne sera pas parfait. Néanmoins, vous devez au moins tenter d'écrire du code qui est généralement correct: il satisfait le plus souvent aux attentes des utilisateurs. Il est possible d'écrire du code correct sans rédiger des tests. Néanmoins, les bénéfices d'une batterie de tests sont tangibles dans le cadre de projets difficiles ou de grande échelle. Plusieurs programmeurs expérimentés refuseront d'utiliser une composante logicielle qui a été construite sans des tests. Certains programmeurs vous diront: "Si le code n'est pas testé, il ne fonctionne pas." L'habitude d'écrire des tests fait sans doute de vous un meilleur programmeur. Sur le plan psychologique, vous êtes davantage conscient de vos limites humaines si vous écrivez des tests. Quand vous interagissez avec d'autres programmeurs et avec des

utilisateurs, vous serez sans doute plus à même de prendre en compte leur rétroaction si avez une une batterie de tests. Dans ce chapitre, nous avons dit peu de choses sur les tests d'intégration. Ceux-ci dépendent souvent de manière plus spécifiques de l'application et des utilisateurs. Dans tous les cas, vous devez débuter par les tests unitaires.

Lectures suggérées

— James Whittaker, Jason Arbon, Jeff Carollo, How Google Tests Software, Addison-Wesley Professional; 1st edition (March 23 2012)
— Lisa Crispin, Janet Gregory, Agile Testing: A Practical Guide for Testers and Agile Teams, Addison-Wesley Professional; 1st edition (Dec 30 2008)
— Hoare, Charles Anthony Richard. "How did software get so reliable without proof?." International Symposium of Formal Methods Europe. Springer, Berlin, Heidelberg, 1996.

Exercices du chapitre 1

Question 1.

Écrivez une fonction qui convertit des nombres à virgule flottante (`float64` en Go) vers des entiers signé représenté avec 16 bits. Écrivez ensuite un test qui permet de vérifier que cette fonction est correcte: quand la valeur est belle et bien un entier, celui-ci est retournée, sinon une erreur est générée.

Question 2.

Écrivez une fonction qui additionne deux entiers non signés (`uint16`), mais qui produit zéro si le résultat ne peut pas être représenté sous la même forme (`uint16`). Écrivez ensuite un test correspondant.

Chapitre 2

En pratique, le code informatique est constamment transformé. Au début d'un projet, le code informatique prend souvent la forme d'esquisses qui sont peu à peu raffinées. Par la suite, le code pourra être optimisé ou corrigé, parfois pendant de nombreuses années.

Assez tôt, les programmeurs ont compris qu'ils avaient besoin non seulement de stocker des fichiers, mais aussi de garder une trace des différentes versions d'un fichier donné. Ce n'est pas un accident si nous sommes tous familiers avec le fait que le logiciel est souvent associé à des versions. Il est nécessaire de distinguer les différentes versions du code informatique afin de suivre les mises à jour.

Nous pourrions croire qu'après avoir avoir mis au point une nouvelle version d'un logiciel, les versions précédentes pourraient être mises au rancard. Pourtant, il est pratique de pouvoir conserver une copie de chaque version du code informatique pour plusieurs raisons:

1. Une modification du code qui nous semblait appropriée peut causer des problèmes: il faut alors pouvoir revenir en arrière rapidement.
2. Il arrive que différentes versions du code informatique soient utilisées en même temps et qu'il ne soit pas possible pour tous les utilisateurs de passer à la dernière version. Si une erreur est trouvée dans une version précédente du code informatique, il faut parfois que le programmeur revienne en arrière pour corriger cette erreur dans une version antérieure du code informatique sans modifier le code courant. Dans ce ce scénario, l'évolution du logiciel n'est

pas strictement linéaire. Il est donc possible de publier la version 1.0 d'un logiciel, suivi de la version 2.0, pour ensuite publier la version 1.1.

3. Il est parfois utile de pouvoir revenir dans le temps pour étudier l'évolution du code afin de comprendre la motivation derrière une section de code. Par exemple, une section de code peut avoir été ajoutée sans beaucoup de commentaires afin de corriger rapidement un nouveau bogue. Le programmeur attentif pourra mieux comprendre le code en retournant lire les modifications dans leur contexte.
4. Par ailleurs, le code informatique est souvent modifié par différents programmeurs travaillant en même temps. Dans un tel contexte social, il est souvent utile de pouvoir déterminer rapidement qui a fait quel changement et à quel moment. Par exemple, si un problème est causé par un segment de code, nous pouvons vouloir poser des questions au programmeur qui a travaillé sur ce segment pour la dernière fois.

De plus, lorsque plusieurs personnes travaillent sur le même code logiciel en même temps, il existe simultanément plusieurs versions différentes de ce même code. Peut-être que chaque programmeur d'une équipe possède sa propre copie locale, et celles-ci diffèrent légèrement.
Il est nécessaire de fusionner régulièrement le travail afin qu'il n'existe qu'une seule version définitive du code. Idéalement, nous souhaiterions que la synchronisation du code soit automatisée et sécurisée. En particulier, nous ne voulons pas perdre un travail précieux au cours de ce processus.

Systèmes de contrôle des version

Les programmeurs ont rapidement constaté qu'ils avaient besoin de systèmes de contrôle des version. Les fonctions de bases que permet un système de contrôle de version sont le retour en arrière, l'ajout d'une modification et la présence d'un historique des modifications effectuées. Avec le temps, le concept s'est répandu. Il y a même plusieurs variantes

destinées au grand public tel que DropBox où divers fichiers, et non seulement du code informatique, sont entreposés.

L'histoire des outils de contrôle de versions logicielles remonte aux années 1970s (Rochkind, 1975 [15]). En 1972, Rochkind développa le SCCS (Source Code Control System) au sein des laboratoires Bell. Ce système permettait de créer, de mettre à jour et de faire le suivi des changements au sein d'un projet logiciel. SCCS est demeuré une référence à compter de la fin des années 1970s jusque dans les années 1980s. Une des contraintes du SCCS est qu'il ne permet pas le travail collaboratif: une seule personne peut modifier un fichier donné à un instant précis.

Au début des années 1980s, Tichy propose le RCS (Revision Control System) qui innove par rapport au SCCS en stockant les différences entre les différentes versions d'un fichier en commençant par la dernière version. Au contraire, SCCS stocke les différences entre les différentes versions en commençant par la première version. Le RCS est donc plus rapide lors d'une utilisation typique où nous chargeons la dernière version du fichier.

En programmation, nous stockons généralement le code informatique au sein de fichiers texte. Les fichiers texte utilisent le plus souvent un encodage ASCII ou Unicode (UTF-8 ou UTF-16). Les lignes sont séparées par une séquence de caractères spéciaux identifiant la fin d'une ligne et le début d'une nouvelle ligne. Deux caractères sont souvent utilisés à cette fin: "carriage return" (CR) et "line feed" (LF). En ASCII et en UTF-8, ces caractères sont représentés avec l'octet ayant la valeur 13 et l'octet ayant la valeur 10 respectivement. Sous Windows, la séquence est composée du caractère CR suivi du caractère LF alors que sous Linux et macOS, on n'utilise que le caractère LF. Dans la plupart des langages de programmation, nous pouvons représenter ces deux caractères avec les séquences d'échappement `\r` et `\n` respectivement. Ainsi donc la chaîne de caractères `"a\nb\nc"` comporte trois lignes dans la plupart des langages de programmation sous Linux ou macOS: les lignes "a", "b" et "c".

Quand le fichier texte est modifié, le plus souvent une petite fraction

15. http://doi.org/10.1109/TSE.1975.6312866

de toutes les lignes sont modifiées. Certaines lignes peuvent aussi être insérées ou supprimées. On cherche à décrire les différences de manière aussi succinctes que possible en identifier les nouvelles lignes, les lignes supprimées et les lignes modifiées.

Le calcul des différences entre deux fichiers texte se fait souvent d'abord en décomposant les fichiers texte en lignes. Nous traitons ensuite un fichier texte comme une liste de lignes. Étant donné deux versions du même fichier, nous souhaitons associer un maximum de lignes de la première version avec une ligne identique dans la seconde version. Nous supposons aussi que l'ordre des lignes n'est pas inversé.

Nous pouvons formaliser ce problème en cherchant la plus longue sous-séquence commune. Étant donné une liste, une sous-séquence reprend simplement une partie de la liste, en excluant quelques éléments. Par exemple, (a,b,d) est une sous-séquence de la liste (a,b,c,d,e). Étant donné deux listes, on peut trouver une sous-séquence commune, par exemple (a,b,d) est une sous-séquence de la liste (a,b,c,d,e) et de la liste (z,a,b,d). La plus longue sous-séquence commune entre deux listes de lignes de texte représente la liste de lignes qui n'ont pas été modifiées entre les deux versions d'un fichier texte. Il pourrait être difficile de résoudre ce programme en utilisant la force brute. Heureusement, nous pouvons calculer la plus longue sous-séquence commune par programmation dynamique. En effet, nous pouvons faire les observations suivantes.

1. Si nous avons deux chaînes ayant une plus longue sous-séquence de longueur k, et que nous ajoutons à la fin de chacune des deux chaînes le même caractère, les nouvelles chaînes auront une plus longue sous-séquence de longueur k+1.
2. Si nous avons deux chaînes de caractères de longueurs m et n, se terminant par des caractères distincts (par exemple, “abc” et “abd”), alors la plus longue sous-séquence des deux chaînes est la plus longue sous-séquence des deux chaînes après avoir éliminé le dernier caractère d'une deux chaînes. En d'autres mots, pour déterminer la longueur de la plus longue sous-séquence entre deux chaînes, on peut prendre le maximum de la longueur de la sous-

séquence après avoir amputé un caractère à la première chaîne tout en gardant la seconde inchangée, et de la longueur de la sous-séquence après avoir amputé un caractère à la seconde chaîne tout en gardant la première inchangée. Ces deux observations suffisent à permettre un calcul efficace de la longueur de la plus longue sous-séquence commune. Il suffit de commencer avec des chaînes de caractères ne comprenant que le premier caractère et d'ajouter progressivement les caractères suivants. De cette manière, on peut calculer toutes les plus longues sous-séquences communes entre les chaînes tronquées. Il est ensuite possible d'inverser ce processus pour construire la plus longue sous-séquence en partant de la fin. Si deux chaînes se terminent par le même caractère, on sait que le dernière caractère fera partie de la plus longue sous-séquence. Autrement, on ampute l'une des deux chaînes de son dernière caractère, en faisant notre choix de telle manière à maximiser la longueur de la plus long sous-séquence commune.

La fonction suivante illustre une solution possible à ce problème. Étant donné deux tableaux de chaînes de caractères, la fonction retourne la plus longue sous-séquence commune. Si la première chaîne a comme longueur `m` et la second `n`, alors l'algorithme s'exécute en temps `O(m*n)`.

```
func longest_subsequence(file1, file2 []string) []string {
  m, n := len(file1), len(file2)
  P := make([]uint, (m+1)*(n+1))
  for i := 1; i <= m; i++ {
    for j := 1; j <= n; j++ {
      if file1[i-1] == file2[j-1] {
        P[i*(n+1)+j] = 1 + P[(i-1)*(n+1)+(j-1)]
      } else {
        P[i*(n+1)+j] = max(P[i*(n+1)+(j-1)],
         P[(i-1)*(n+1)+j])
      }
    }
  }
```

```
  longest := P[m*(n+1)+n]
  i, j := m, n
  subsequence := make([]string, longest)
  for k := longest; k > 0; {
    if P[i*(n+1)+j] == P[i*(n+1)+(j-1)] {
      j-- // the two strings end with the same char
    } else if P[i*(n+1)+j] == P[(i-1)*(n+1)+j] {
      i--
    } else if P[i*(n+1)+j] == 1+P[(i-1)*(n+1)+(j-1)] {
      subsequence[k-1] = file1[i-1]
      k--; i--; j--
    }
  }
  return subsequence
}
```

Une fois la sous-séquence calculée, on peut calculer rapidement une description de la différence entre les deux fichiers textes. Il suffit d'avancer dans chacun des fichiers texte, ligne par ligne en stoppant dès que l'on atteint une position correspondant à un élément de la plus longue sous-séquence. Les lignes ne correspondant pas à la sous-séquence dans le premier fichier sont considérées comme ayant été effacées alors que les lignes ne correspondant pas à la sous-séquence dans le second fichier sont considérées comme ayant été ajoutées. La fonction suivante illustre une solution possible.

```
func difference(file1, file2 []string) []string {
  subsequence := longest_subsequence(file1, file2)
  i, j, k := 0, 0, 0
  answer := make([]string, 0)
  for i < len(file1) && k < len(file2) {
    if file2[k] == subsequence[j] &&
     file1[i] == subsequence[j] {
      answer = append(answer, "'"+file2[k]+"'\n")
      i++; j++; k++
```

```
    } else {
      if file1[i] != subsequence[j] {
        answer = append(answer,
         "deleted: '"+file1[i]+"'\n")
        i++
      }
      if file2[k] != subsequence[j] {
        answer = append(answer, "added: '"+file2[k]+"'\n")
        k++
      }
    }
  }
  for ; i < len(file1); i++ {
    answer = append(answer, "deleted: '"+file1[i]+"'\n")
  }
  for ; k < len(file2); k++ {
    answer = append(answer, "added: '"+file2[k]+" \n")
  }
  return answer
}
```

La fonction que nous proposons à titre d'illustration pour le calcul de la plus longue sous-séquence utilise `O(m*n)` éléments de mémoire. Il est possible de réduire l'utilisation de mémoire de cette fonction et de la simplifier (Hirschberg, 1975 [16]). Plusieurs autres améliorations sont possibles en pratique (Miller and Myers, 1985 [17]). Nous pouvons ensuite représenter les modifications entre les deux fichiers de manière concise.

Lecture suggérée: article Diff (wikipedia) [18]

Tout comme SCCS, le RCS ne permet pas à plusieurs programmeurs de travailler sur le même fichier au même moment. La nécessité de

16. https://doi.org/10.1145/360825.360861
17. https://doi.org/10.1002/spe.4380151102
18. https://en.wikipedia.org/wiki/Diff

s'approprier un fichier, à l'exclusion de tous les autres programmeurs, pendant le travail pouvait sembler alors une contrainte raisonnable mais elle peut considérablement alourdir le travail au sein d'une équipe de programmeurs.

En 1986, Grune développa le CVS (Concurrent Versions System). Contrairement aux systèmes précédents, CVS permet à plusieurs programmeurs de travailler simultanément sur le même fichier. Il adopte aussi un modèle client-serveur qui permet d'avoir un seul répertoire présent sur un réseau, accessible par plusieurs programmeurs simultanément. Le programmeur peut travailler sur un fichier localement, mais tant qu'il n'a pas transmis sa version au serveur, celle-ci demeure invisible pour les autres développeurs.

Le serveur distant sert aussi dans les faits de copie de sauvegarde pour les programmeurs. Même si les ordinateurs de tous les programmeurs sont détruits, il est possible de repartir du code présent sur le serveur distant.

Dans un système de contrôle de versions, il y a habituellement toujours une dernière version unique. Tous les programmeurs apportent des modifications à cette dernière version. Cependant une telle approche linéaire a des limites. Une innovation importante que CVS a mis à jour est le concept de branche. Une branche permet d'organiser des ensembles de versions capables d'évoluer en parallèle. Dans ce modèle, un même fichier est virtuellement dupliqué. Il existe alors deux versions du fichier (ou plus que deux) capables d'évoluer en parallèle. Par convention, il y a généralement une branche principale qui est utilisée par default, accompagnée de plusieurs branches secondaires. Les programmeurs peuvent créer de nouvelles branches quand bon leur semble. On peut ensuite refusionner des branches: si une branche A est divisée en deux branches (A et B) qui sont modifiées, il est ensuite possible de ramener toutes les modifications dans une seule branche (fusion de A et de B). Le concept de branche est utile dans plusieurs contextes:

1. Certains développements logiciels sont spéculatifs. Par exemple, un programmeur peut explorer une nouvelle approche sans être certain qu'elle soit viable. Dans un tel cas, il peut être préférable

de travailler dans une branche distincte et de fusionner avec la branche principale seulement en cas de succès.
2. La branche principale peut être limitée à certains programmeurs pour des raisons de sécurité. Dans un tel cas, les programmeurs avec un accès réduit peuvent être limités à des branches distinctes. Un programmeur avec un accès privilégié pourra ensuite fusionner la branche secondaire après une inspection du code.
3. Une branche peut être utilisée pour explorer un bogue particulier et sa correction.
4. Une branche peut être utilisée pour y mettre à jour une version antérieure du code. Une telle version peut être maintenue à jour parce que certains utilisateurs dépendent de cette version antérieure tout en souhaitant recevoir certains correctifs. Dans un tel cas, la branche secondaire ne sera peut-être jamais intégrée à la branche principale.

S'il est relativement facile de définir algorithmiquement quelles lignes ont changé entre les révisions successives d'un même fichier, différentes modifications simultanées sur le même fichier rendent le problème plus difficile. En pratique, ces modifications simultanées peut conduire à des *conflits* qui obligent les utilisateurs à prendre des décisions. Par exemple, supposons que nous commencions avec un fichier contenant une seule fonction:

```
func f1() int {
  return 1
}
```

L'utilisateur A ajoute une fonction `fa`...

```
func f1() int {
  retour 1
}

func fa() int {
  retour 2
```

```
}
```

L'utilisateur B ajoute une fonction `fb` dans une branche différente...

```
func f1() int {
  retour 1
}

func fb() int {
  retour 2
}
```

La fusion de ces changements est un choix suggestif. Devrait-il y avoir trois fonctions (`f1`, `fa` et `fb`)? Ou bien les deux utilisateurs ajoutent-ils la même fonction, mais avec des noms différents? Dans ce cas, nous ne devrions avoir que deux fonctions dans le fichier.

Ainsi, en pratique, il peut être difficile d'avoir différentes branches modifiant les mêmes fichiers, car la fusion des résultats pourrait nécessiter de plus en plus d'intervention humaine.

Un des inconvénients du CVS est une piètre performance lorsque les projets comprennent plusieurs fichiers et plusieurs versions. En 2000, Subversion (SVN) fut proposé comme une solution de rechange à CVS qui répond aux mêmes besoins, mais avec une meilleure performance.

CVS et Subversion bénéficient d'une approche client-serveur, ce qui permet à plusieurs programmeurs de travailler simultanément avec un même répertoire de versions. Par contre, les programmeurs souhaitent souvent pouvoir utiliser plusieurs répertoires distants distincts.

Systèmes de contrôle de version distribués

Afin de répondre à ces besoins, divers systèmes de contrôle de version distribués ou *distributed version control system* (DVCS) ont vu le jour. Le plus populaire étant sans doute le système Git mis au point par Torvalds

(2005). Torvalds cherchait à résoudre un problème de gestion du code source Linux. Git est devenu l'outil de gestion de versions dominant. Il a été adopté notamment par Google, Microsoft, etc. Il s'agit de logiciel libre.

Dans un modèle distribué, un programmeur disposant d'une copie locale du code pourra la synchroniser soit avec un répertoire, soit avec un autre. Il peut facilement créer une nouvelle copie du répertoire distant sur un nouveau serveur. Une telle flexibilité est considérée comme essentielle au sein de plusieurs projets complexes comme le noyau du système d'exploitation Linux.

Plusieurs entreprises offrent des services basés sur Git dont GitHub. Fondé en 2008, GitHub compte des dizaines de millions d'utilisateurs. En 2018, Microsoft a acquis GitHub pour 7.5 milliards de dollars.

Pour CVS et Subversion, il n'y a qu'un ensemble de versions du logiciel. Avec une approche distribuée, plusieurs ensembles peuvent coexister sur des serveurs distincts. Le résultat net est qu'un projet logiciel peut évoluer différemment, sous la responsabilité de différentes équipes, avec une possible future réconciliation.

En ce sens, Git est distribué. Bien que plusieurs utilisateurs s'en remettent à GitHub (par exemple), votre copie locale peut être rattachée à n'importe quel répertoire distant et on peut même le rattacher à plusieurs répertoires distants. On utilise parfois le verbe *cloner* pour décrire la récupération d'un projet Git localement puisqu'il s'agit alors d'une copie intégrale de tous les fichiers, de toutes les modifications et de toutes les branches.

Si on rattache une copie du projet à un autre répertoire distant, on parle alors d'une fourche (fork). On distingue souvent les branches des fourches. Une branche appartient toujours au projet principal. La fourche est à l'origine une copie intégrale du projet, incluant toutes les branches. Il est possible pour une fourche de réintégrer le projet principal, mais ce n'est pas essentiel.

Étant donné un répertoire Git publiquement disponible, n'importe qui peut le cloner et commencer à y travailler et y contribuer. On peut

créer une nouvelle fourche. À partir d'une fourche, on peut soumettre un *pull request* qui invite les gens à intégrer nos changements. Cela permet une forme d'innovation sans permission. En effet, il devient possible de récupérer le code, de le modifier et de proposer une nouvelle version sans jamais devoir interagir directement avec les auteurs.

Les systèmes comme CVS et subversion pouvaient devenir inefficaces et prendre plusieurs minutes pour effectuer certaines opérations. Quant à lui, Git est généralement efficace et rapide, même pour des projets énormes. Git est robuste et il ne se *corrompt* pas facilement. Il n'est cependant pas recommandé d'utiliser Git pour des fichiers énormes comme du contenu multimédia: la force de Git repose sur des fichiers texte. Il faut préciser que la mise en œuvre de Git s'est améliorée avec le temps et qu'elle comprend des techniques d'indexation sophistiquées.

On utilise souvent Git en ligne de commande. Il est possible d'utiliser des clients graphiques. Des services comme GitHub rendent Git un peu plus facile.

L'unité de base logique du Git est le `commit` qui est un ensemble de modifications sur plusieurs fichiers. Un `commit` comprend notamment une référence à au moins un parent, à l'exception du premier `commit` qui n'a pas de parent. Un même `commit` peut être le parent de plusieurs enfants: on peut créer plusieurs branches à partir d'un `commit` et chaque `commit` subséquent devient un *enfant* du `commit` initial. Par ailleurs, lorsque plusieurs branches sont fusionnées, le `commit` qui est résulte aura plusieurs parents. En ce sens, les `commit` forment un *graphe dirigé acyclique*.

Avec Git, on souhaite pouvoir faire référence à un `commit` de manière aisée, en utilisant un identifiant unique. C'est-à-dire que nous souhaitons disposer d'une courte valeur numérique qui correspond à un `commit` et un seul `commit`. On pourrait assigner à chaque `commit` un numéro de version (1.0, 2.0, etc.). Malheureusement, une telle approche est difficile à réconcilier avec le fait que les `commit`s ne forment pas une chaîne linéaire où un `commit` n'ayant qu'un et un seul parent. Comme solution de rechange, nous utilisons une fonction de hachage pour calculer l'identifiant

unique. Une fonction de hachage prend des éléments en paramètre et calcule une valeur numérique (valeur de hachage). Il existe plusieurs fonctions de hachage simple. Par exemple, on peut itérer sur les octets contenus dans un message à partir d'une valeur de départ `h`, en calculant `h = 31 * h + b` où `b` est la valeur de l'octet. Par exemple, un message contenant les octets 3 et 4 pourra avoir une valeur de hachage de `31 * (31 * 3) + 4` si on démarre `h = 0`. Une telle approche simple est efficace dans certains cas, mais elle permet à des utilisateurs malveillants de créer des collisions: il serait possible de créer un faux `commit` qui a la même valeur de hachage et de créer ainsi des failles de sécurité. Pour cette raison, Git utilise des techniques de hachage plus sophistiquées (SHA-1, SHA-256) développés par des spécialistes en cryptographie. Les `commit` sont identifiés en utilisant une valeur de hachage (par exemple, la valeur numérique hexadécimale 921103db8259eb9de72f42db8b939895f5651489) qui est calculée à partir de la date et de l'heure, du commentaire laissé par le programmeur, par le nom de l'utilisateur, par les parents et par la nature du changement. En théorie, deux `commit` pourraient avoir la même valeur de hachage, mais c'est un événement improbable étant donné les fonctions de hachage utilisés par Git.

Il peut sembler que Git s'appuie sur le hasard en supposant que deux `commit` au sein d'un même projet ne peuvent avoir la même valeur de hachage. Dans un certain sens, c'est le cas, mais la probabilité de collision est si faible qu'elle peut être ignorée. En effet, les systèmes informatiques ne sont de toute façon jamais parfaits. Nous sommes constamment bombardés par des rayons cosmiques qui peuvent changer des bits dans nos systèmes de mémoire. Nous sommes sujets à des défaillances matérielles aléatoires. Nous acceptons des probabilités d'échec algorithmiques qui sont astronomiquement plus faibles que les probabilités d'échec liées au matériel.

Il n'est pas toujours pratique de faire référence à un code hexadécimal. Pour faciliter le travail, Git permet d'identifier un commit à l'aide d'une étiquette (par ex. v1.0.0). La commande suivante suffira: `git tag -a v1.0.0 -m "version 1.0.0"`. La commande crée une balise annotée nommée `v1.0.0` dans un dépôt git, marquant le `commit` courant avec une

étiquette de version. L'option `-m "version 1.0.0"` fournit un message descriptif pour le tag, indiquant ici la version.

Les étiquettes peuvent utiliser n'importe quelle chaîne de caractères, mais elles sont néanmoins souvent composées de séries de nombres représentant une version du logiciel. Il n'y a pas de norme universelle concernant l'attribution de nombres aux versions d'un logiciel. Néanmoins, les étiquettes prennent parfois la forme de trois nombres séparés par des points: MAJOR.MINOR.PATCH (par exemple, 1.2.3). À chaque nouvelle version, on ajoute 1 à au moins un des trois nombres. Le premier nombre débute souvent à 1 tandis que les deux suivants débutent à 0. - Le premier nombre (MAJOR) doit être augmenté lorsque vous faites des modifications majeures au code. Les deux autres nombres (MINOR et PATCH) sont souvent remis à zéro. Ainsi, on peut passer de la version 1.2.3 à la version 2.0.0. - Le second nombre (MINOR) est augmenté lors de modifications mineures (par exemple, l'ajout d'une fonction). Lorsqu'on augmente le second nombre, on garde généralement le premier nombre (MAJOR) inchangé et on remet le dernier nombre (PATCH) à zéro. - Le dernier nombre (PATCH) est augmenté lors de la correction de bogues. Les deux autres nombres ne sont pas augmentés. Il existe des versions plus fines de cette convention comme le *semantic versioning* [19].

Avec Git, le programmeur peut disposer d'une copie locale du graphe des `commit`s. Il peut ajouter de nouveaux `commit`s. Dans une étape subséquente, le programmeur doit *pousser* ses modifications vers un répertoire distant afin que les modifications deviennent visibles pour les autres programmeurs. Les autres programmeurs peuvent aller chercher les modifications l'aide d'un `pull`.

Git offre un ensemble de fonctionnalités avancées qui facilitent le travail collaboratif au sein d'équipes de développement, en permettant une traçabilité précise et une coordination efficace des modifications apportées au code. La commande `git blame` est l'un des outils les plus emblématiques à cet égard. Elle permet d'analyser un fichier ligne par ligne pour identifier l'auteur de la dernière modification de chaque ligne, la date de cette

19. https://semver.org

modification et le commit associé. Par exemple, en exécutant `git blame nom_du_fichier`, Git affiche chaque ligne du fichier avec des informations telles que l'identifiant du commit (hash), le nom de l'auteur et la date, offrant ainsi une vue détaillée de l'historique des contributions. Cette fonctionnalité est particulièrement utile dans plusieurs contextes. Tout d'abord, elle aide à comprendre l'origine d'un bug ou d'un comportement inattendu dans le code. Si une erreur est détectée, un programmeur peut utiliser `git blame` pour repérer qui a modifié la ligne problématique en dernier, ce qui permet de poser des questions spécifiques à cette personne ou d'examiner le contexte de la modification via le message du commit. Ensuite, `git blame` favorise la responsabilité et la transparence dans les projets collaboratifs, car chaque modification est rattachée à un contributeur. Cela peut également être utile lors de revues de code pour évaluer les contributions individuelles ou pour documenter l'évolution d'un projet. Au-delà de `git blame`, Git propose d'autres outils collaboratifs puissants. Par exemple, la commande `git log` permet de consulter l'historique complet des commits, avec des options comme `--author` pour filtrer les contributions d'un programmeur spécifique ou `--grep` pour rechercher des commits par mot-clé dans leurs messages.

Commits atomiques

Un commit doit encapsuler un changement unique et cohérent dans la base de code, facilitant la compréhension, le suivi et la gestion des modifications au fil du temps. L'idée est de maintenir chaque commit centré sur un aspect spécifique du code, en évitant de regrouper des changements non liés. Cette approche améliore la clarté, simplifie le débogage et facilite la collaboration en s'assurant que chaque commit représente une unité de travail logique et autonome. On appelle parfois ces commits *atomiques*.

Un commit atomique doit traiter une tâche, une fonctionnalité ou une correction spécifique. Il doit inclure toutes les modifications pertinentes pour accomplir cette tâche (par exemple, le code, les tests et la documentation). Il doit éviter de mélanger des changements non liés, comme combiner une correction de bogue avec une mise à jour du style de code

ou une nouvelle fonctionnalité.

En gardant les commits atomiques, les développeurs peuvent plus facilement examiner l'historique des changements pour comprendre pourquoi et comment des modifications spécifiques ont été effectuées. Ils peuvent mieux annuler ou sélectionner des changements individuels sans affecter d'autres parties du code. Ils peuvent identifier la source des bogues ou des problèmes en isolant les changements dans des commits spécifiques.

Supposons que vous travaillez sur un projet et que vous devez ajouter une nouvelle fonctionnalité pour calculer le carré d'un nombre dans un programme Go, reformater du code pour plus de cohérence et mettre à jour la documentation. Au lieu de regrouper toutes ces modifications dans un seul commit, vous créeriez des commits séparés pour chaque tâche. Considérez le point de départ suivant.

```
func add(a, b int) int {
return a + b
}
```

Vous remarquez que le code manque de cohérence dans l'indentation et le formatage. Vous exécutez un formateur (par exemple, `gofmt`) pour standardiser le style. Vous commitez ce changement séparément.

Message de commit : Reformater le code pour un style cohérent

```
func add(a, b int) int {
    return a + b
}
```

Ce commit ne concerne que le formatage du code et n'inclut pas de changements fonctionnels.

Vous ajoutez une nouvelle fonction `square` pour calculer le carré d'un nombre. C'est une fonctionnalité distincte, elle obtient donc son propre commit.

Message de commit : Implémenter la fonction square

```
func add(a, b int) int {
    return a + b
}

func square(n int) int {
    return n * n
}
```

Ce commit se concentre uniquement sur l'ajout de la nouvelle fonction et son implémentation.

Vous mettez à jour la documentation du projet (par exemple, un README ou des commentaires de code) pour expliquer la nouvelle fonction `square`. C'est une tâche distincte, elle est donc committée indépendamment.

Message de commit : Documenter la fonction square dans les commentaires de code

```
// add retourne la somme de deux entiers.
func add(a, b int) int {
    return a + b
}

// square retourne le carré d'un entier.
func square(n int) int {
    return n * n
}
```

Ce commit ne concerne que l'ajout de documentation, en la gardant distincte de l'implémentation.

Si toutes ces modifications étaient regroupées dans un seul commit avec un message vague comme « Mettre à jour le code », il serait plus difficile de comprendre l'objectif de chaque changement lors de l'examen de l'historique des commits. En revanche, les commits atomiques rendent l'historique clair :

```
$ git log --oneline
c3f2e1d Documenter la fonction square
  dans les commentaires de code
a7b9c4f Implémenter la fonction square
f1a2b3c Reformater le code pour un style cohérent
```

Chaque commit a un objectif spécifique, et les messages décrivent clairement ce qui a été fait. Si un bogue est détecté dans la fonction `square`, vous pouvez utiliser `git blame` ou `git bisect` pour localiser le commit `a7b9c4f` et l'examiner ou l'annuler sans affecter les changements de formatage ou de documentation.

Un mauvais commit pourrait ressembler à l'exemple suivant.

Message de commit : Ajouter la fonction square et corriger des choses

```go
// add retourne la somme de deux entiers
func add(a, b int) int {
    return a + b
}

// square retourne le carré d'un entier
func square(n int) int {
    return n * n
}
```

Ce commit combine le formatage, l'implémentation de la fonctionnalité et la documentation en un seul. Si vous devez annuler la fonctionnalité tout en conservant la documentation, cela devient difficile car les changements sont entrelacés. De plus, le message vague ne précise pas ce que signifie « corriger des choses », ce qui réduit la traçabilité.

En utilisant des commits atomiques, vous créez un historique de versions plus maintenable et transparent, rendant la collaboration et le débogage plus efficaces.

Les branches dans Git

Dans Git, une innovation majeure par rapport aux systèmes de contrôle de version précédents comme CVS ou Subversion est le concept de branches. Une branche dans Git représente une ligne de développement indépendante qui permet à plusieurs versions d'un même projet d'évoluer en parallèle. Contrairement à un modèle linéaire où toutes les modifications sont appliquées à une seule version principale, les branches permettent aux programmeurs de travailler sur des copies distinctes du code sans affecter la branche principale, souvent appelée `main` ou `master`. Une branche peut être créée avec la commande `git branch <nom-de-la-branche>` pour définir une nouvelle branche, suivie de `git switch <nom-de-la-branche>` pour basculer vers celle-ci. Une fois sur une branche, les modifications effectuées sont isolées des autres branches, permettant ainsi des expérimentations ou des corrections sans risque pour le code principal.

Un programmeur peut travailler sur une fonctionnalité expérimentale sans garantie de succès. En utilisant une branche dédiée, il peut tester de nouvelles idées et ne fusionner avec la branche principale que si le résultat est satisfaisant. Dans certains projets, l'accès à la branche principale peut être restreint à un groupe spécifique de programmeurs. Les autres contributeurs travaillent sur des branches secondaires, et leurs modifications sont fusionnées après une revue de code par un responsable. Une branche peut être créée pour isoler et corriger un bug spécifique, en particulier dans une version antérieure du logiciel encore utilisée par certains utilisateurs. Une branche peut être dédiée à la maintenance d'une version précédente du code, recevant des mises à jour ou des correctifs sans être fusionnée avec la branche principale.

La fusion des branches, réalisée avec la commande `git merge <nom-de-la-branche>`, permet d'intégrer les modifications d'une branche dans une autre. Par exemple, si une branche `feature` contient une nouvelle fonctionnalité, elle peut être fusionnée dans la branche `main` une fois terminée. Cependant, des conflits peuvent survenir si des modifications concurrentes ont été apportées au même fichier dans différentes branches. Par exemple, si deux programmeurs modifient la

même ligne de code différemment, Git signalera un conflit qui devra être résolu manuellement en éditant le fichier pour choisir ou combiner les modifications.

Voici un exemple simple pour illustrer un conflit potentiel. Supposons un fichier initial contenant le code suivant.

```
func f1() int {
    return 1
}
```

Un programmeur A, sur une branche `branche-a`, ajoute une fonction `fa`.

```
func f1() int {
    return 1
}
func fa() int {
    return 2
}
```

Un programmeur B, sur une branche `branche-b`, ajoute une fonction `fb`.

```
func f1() int {
    return 1
}
func fb() int {
    return 2
}
```

Lors de la fusion de `branche-a` et `branche-b` dans `main`, Git peut généralement combiner les modifications automatiquement, car elles affectent des parties différentes du fichier. Le résultat serait le suivant.

```
func f1() int {
    return 1
}
func fa() int {
    return 2
```

```
}
func fb() int {
    return 2
}
```

Cependant, si les deux programmeurs modifient la même ligne, Git signalera un conflit, et le programmeur devra intervenir pour résoudre le problème en choisissant quelle version conserver ou en combinant les modification.

Les branches sont souvent utilisées en combinaison avec des services comme GitHub, où les programmeurs peuvent soumettre des *pull requests* pour proposer l'intégration de leurs modifications. Cela facilite la collaboration à grande échelle, car n'importe qui peut cloner un dépôt public, créer une branche, effectuer des modifications et proposer leur intégration sans interaction directe avec les mainteneurs du projet.

Pour énumérer toutes les branches d'un dépôt, on utilise `git branch`. Pour supprimer une branche devenue inutile, on peut exécuter `git branch -d <nom-de-la-branche>`. Git garantit également que les branches supprimées localement n'affectent pas les dépôts distants, offrant ainsi une sécurité supplémentaire.

Conclusion

Le contrôle de versions en informatique est une approche sophistiquée, qui a bénéficié de plusieurs années de travail. Il possible de stocker à moindre coût plusieurs versions d'un même fichier et de naviguer d'une version à l'autre rapidement.

Si vous développez du code sans utiliser un outil de contrôle des versions comme Git ou l'équivalent, vous passez outre à des pratiques éprouvées. Il est probable que si vous souhaitez travailler sur des projets complexes, avec plusieurs programmeurs, votre productivité sera bien moindre sans contrôle des versions.

Exercices du chapitre 2

Question 1.

Concevez une fonction qui représente la différence entre deux fichiers de manière concise. Par exemple, si seulement deux lignes sont modifiées dans un grand fichier, il devrait être possible de stocker approximativement seulement deux lignes de texte.

Question 2.

Donnez votre solution au premier problème. Ecrivez une fonction qui peut prendre le fichier original et votre description concise et reconstruire le second fichier.

Question 3.

Lisez le tutoriel Git [20]. Installez Git sur votre machine, créez un nouveau répertoire, modifiez-le, créez une nouvelle branche, modifiez-la et fusionnez les deux branches.

20. https://git-scm.com/docs/gittutorial

Chapitre 3

La programmation informatique commence par l'organisation des données en structures de données. Dans presque tous les cas, nous travaillons avec des chaînes de caractères ou des nombres. Il est essentiel de comprendre ces éléments de base pour devenir un programmeur expert.

Mots

Nous organisons souvent les données en utilisant des blocs fixes de mémoire. Lorsque ces blocs sont relativement petits (par exemple, 8 bits, 16 bits, 32 bits, 64 bits), nous les appelons communément *mots*.

La notion de *mot* est importante car les processeurs n'opèrent pas sur des types de données arbitraires. Pour des raisons pratiques, les processeurs s'attendent à ce que les données tiennent dans des registres matériels de taille fixe (généralement des registres de 64 bits). La plupart des processeurs modernes traitent des mots de 8, 16, 32 et 64 bits avec des instructions rapides. Il est courant que la granularité des accès à la mémoire ne soit pas inférieure à l'*octet* (8 bits), de sorte que les octets sont, en un sens, les plus petits mots pratiques.

Les structures de données de longueur variable, comme les chaînes de caractères, peuvent être composées d'un nombre variable de mots. Historiquement, les chaînes de caractères ont été constituées de listes d'octets, mais d'autres possibilités sont courantes (par exemple, des mots de 16 ou 32 bits).

Valeurs booléennes

Le type le plus simple est probablement le type booléen. Une valeur booléenne peut prendre la valeur faux ou vrai. Bien qu'un seul bit suffise pour représenter une valeur booléenne, il est courant d'utiliser un octet entier (ou plus). On peut inverser une valeur booléenne: la valeur vraie devient la valeur fausse, et inversement. Il existe également des opérations binaires:

— Le résultat de l'opération OU entre deux valeurs booléennes est faux si et seulement si les deux entrées sont fausses. L'opération OU est souvent notée `|`. Par exemple, `1 | 0 == 1` où nous utilisons la convention que le symbole `==` indique l'égalité entre deux valeurs.
— Le résultat de l'opération ET entre deux valeurs booléennes est vrai si et seulement si les deux entrées sont vraies. L'opération ET est souvent notée `&`. Par exemple, `1 & 1 == 1`.
— Le résultat de l'opération XOR est vrai si et seulement si les deux entrées ont une valeur différente. L'opération XOR est souvent notée `^`. Par exemple, `1 ^ 1 == 0`.
— Le résultat de l'opération AND NOT entre deux valeurs booléennes est vrai si et seulement si la première valeur booléenne est vraie et la seconde fausse.

Entiers

Les types de données représentant des entiers sont probablement les plus largement pris en charge dans les logiciels et le matériel, après les types booléens. Nous représentons souvent les nombres entiers à l'aide de chiffres. Par exemple, le nombre entier 1234 comporte 4 chiffres décimaux. Par extension, nous utilisons des chiffres *binaires*, appelés bits, dans les ordinateurs. Nous écrivons souvent un nombre entier en utilisant la notation binaire avec le préfixe `0b`. Par exemple, l'entier `0b10` vaut deux, l'entier `0b10110` est égal à `2^1+2^2+2^4` ou 22. Après le préfixe `0b`, on énumère les valeurs des bits, en commençant par le bit le plus significatif. Nous pouvons également utiliser la notation hexadécimale (base 16) avec le

préfixe `0x`: dans ce cas, nous utilisons 16 chiffres différents dans la liste `0, 1, 2, 3,..., 9, A, B, C, D, E, F`. Ces chiffres ont les valeurs `0, 1, 2, 3,..., 9, 10, 11, 12, 13, 14, 15`. Pour les chiffres représentés avec des lettres, nous pouvons utiliser les minuscules ou les majuscules. Ainsi le nombre `0x5A` est égal à `5 * 16 + 10` ou 90 en décimal. La notation hexadécimale est pratique pour travailler avec des valeurs binaires: un seul chiffre représente une valeur de 4 bits, deux chiffres représentent une valeur de 8 bits, et ainsi de suite.

On peut compter le nombre de chiffres d'un entier en utilisant la formule `ceil(log(x+1))` où le logarithme est dans la base qui nous intéresse (par exemple, la base 2) et où `ceil` est la fonction *plafond*: `ceil(x)` renvoie le plus petit entier non inférieur à `x`. Le produit entre un entier ayant `d1` chiffres et un entier ayant `d2` chiffres a soit `d1+d2-1` chiffres, soit `d1+d2` chiffres. À titre d'exemple, considérons le produit de deux nombres entiers à trois chiffres. En base 10, le plus petit produit est 100 fois 100, soit 10 000, et nécessite donc 5 chiffres. Le plus grand produit est 999 fois 999 ou 998 001, donc 6 chiffres.

Pour des raisons de rapidité ou de commodité, nous pouvons utiliser un nombre fixe de chiffres. Étant donné que nous travaillons avec des ordinateurs binaires, il est probable que nous utilisions des chiffres binaires. Nous avons également besoin d'un moyen de représenter le signe d'un nombre (négatif et positif).

Entiers non signés

Le type de nombre le plus simple est probablement l'entier non signé, où un nombre fixe de bits est utilisé pour représenter les entiers non négatifs. La plupart des processeurs prennent en charge les opérations arithmétiques sur les nombres entiers non signés. Dans ce cas, le terme *non signé* est équivalent à *non négatif*: les entiers peuvent être nuls ou positifs.

Nous pouvons effectuer des opérations sur des entiers binaires en utilisant des opérations logiques par bit. Par exemple, le ET par bit (noté avec

le symbole &) entre `0b101` et `0b1100` est `0b100`: `0b101 & 0b1100 == 0b100`. L'opération OU par bit (avec le symbole |) est `0b1101`. Le XOR (OU exclusif) par bit est `0b1001`, noté avec le symbole ^: `0b101 ^ 0b1100 == 0b1001`.

Les puissances de deux (1, 2, 4, 8,...) sont les seuls nombres ayant un seul bit 1 dans leur représentation binaire (`0b1`, `0b10`, `0b100`, `0b1000`, etc.). Les nombres précédant les puissances de deux (1, 3, 7,...) sont les nombres composés de bits 1 consécutifs dans les positions les moins significatives (`0b1`, `0b11`, `0b111`, `0b1111`, etc.). Une caractéristique unique des puissances de deux est que leur ET bit à bit avec l'entier précédent est égal à zéro: par exemple, 4 ET 3 est égal à zéro, 8 ET 7 est égal à zéro, et ainsi de suite.

Dans le langage de programmation Go, par exemple, nous disposons de types de nombres entiers non signés de 8 bits, 16 bits, 32 bits et 64 bits: `uint8`, `uint16`, `uint32`, `uint64`. Ils peuvent représenter tous les nombres de 0 à (mais pas inclus) 2 à la puissance 8, 16, 32 et 64. Par exemple, un entier non signé de 8 bits peut représenter tous les entiers de 0 à 255 inclusivement.

Comme nous choisissons d'utiliser un nombre fixe de bits, nous ne pouvons donc représenter qu'une plage d'entiers. Le résultat d'une opération arithmétique peut dépasser cette plage: il y a un *débordement*. Par exemple, 255 plus 2 est égal à 257: bien que les deux entrées (255 et 2) puissent être représentées à l'aide d'entiers non signés de 8 bits, le résultat dépasse la plage.

En ce qui concerne les multiplications, le produit de deux entiers non signés de 8 bits est au maximum de 65025, ce qui peut être représenté par un entier non signé de 16 bits. Il est toujours vrai que le produit de deux entiers de `n` bits peut être représenté en utilisant `2n` bits. L'inverse n'est pas vrai: un entier de `2n` bits donné n'est pas le produit de deux entiers de `n` bits. Lorsque `n` devient grand, seule une petite fraction de tous les entiers de `2n`-bits peut être écrite comme le produit de deux entiers de `n`-bits, un résultat prouvé pour la première fois par Erdős.

En général, les opérations arithmétiques sont *modulo* la puissance de deux. C'est-à-dire que tout se passe comme si on faisait le calcul en utilisant des entiers de précision infinie et qu'on ne gardait ensuite que le reste (positif) de la division par la puissance de deux.

Développons cette idée. Étant donné deux entiers `a` et `b` (`b` étant non nul), il existe des entiers uniques `d` et `r` où `r` est dans `[0,b)` tels que `a = d * b + r`. L'entier `r` est le reste et l'entier `d` est le quotient. En Go, le quotient est obtenu avec l'expression `a/b` alors que le reste est donné par `a%b`.

Le lemme d'Euclide nous dit que le quotient et le reste existent et sont uniques. Nous pouvons vérifier l'unicité. Supposons qu'il existe une autre paire d'entiers (`d'` et `r'`), `a = d' * b + r'`. On peut vérifier que si `d'` est égal à `d`, alors on doit avoir que `r'` est égal à `r`, et inversement, si `r'` est égal à `r`, alors `d'` est égal à `d`. Supposons que `r'` soit supérieur à `r` (sinon, il suffit d'inverser l'argument). Alors, par soustraction, nous avons que `0 = (d'-d)*b + (r'-r)`. On doit avoir que `r'-r` est dans `[0,b)`. Si `d'-d` est négatif, alors on a que `(d-d')*b = (r'-r)`, mais c'est impossible car `r'-r` est dans `[0,b)` alors que `(d-d')*b` est supérieur ou égal à `b`. Un argument similaire fonctionne lorsque `d'-d` est positif.

Dans notre cas, le diviseur (`b`) est une puissance de deux. Lorsque le numérateur (`a`) est positif, alors le reste équivaut à une sélection des bits les moins significatifs. Par exemple, le reste de la division de 65 (ou `0b1000001`) par 64 est 1.

Lorsque l'on considère l'arithmétique non signée, il est souvent utile de penser que l'on ne garde que les bits les moins significatifs (8, 16, 32 ou 64) du résultat final. Ainsi, si on prend 255 et qu'on ajoute 2, on obtient 257, mais en tant qu'entier non signé de 8 bits, on obtient le chiffre 1. Ainsi, en utilisant des entiers du type `uint8`, nous avons que `255 + 2` est 1 (`255 + 2 == 1`). La puissance de deux est elle-même égale à zéro: 256 est égal à zéro en tant qu'entier `uint8`. Si l'on soustrait deux nombres et que la valeur est négative, on fait le tour: 10 - 20 en arithmétique `uint8` est le reste positif de (-10) divisé par 256, soit 246. Une autre façon de penser aux nombres négatifs est que nous pouvons ajouter la puissance de

deux (disons 256) autant de fois que nécessaire (sa taille est effectivement nulle) jusqu'à ce que nous obtenions une valeur qui se situe entre 0 et la puissance de deux. Ainsi, si nous devons évaluer `1-5*250` comme un entier de 8 bits, nous prenons le résultat (-1249) et nous ajoutons 256 autant de fois que nécessaire: nous avons que `-1249+5*256` est 31, un nombre entre 0 et 256. Ainsi, `1-5*250` est 31 en tant que nombre non signé sur 8 bits.

Nous avons que `0-1`, en tant que nombre de 8 bits, est 255 ou `0b11111111`. Nous avons que `0-2` est 254, nous avons que `0-3` est 253 et ainsi de suite. Considérons l'ensemble des nombres entiers.

```
-1024, -1023, ..., -513, -512, -511, ...,
 -1, 0, 1, ..., 255, 256, 257, ...
```

En tant qu'entiers de 8 bits, ils sont convertis de la manière suivante.

```
0, 255, ... , 255, 0, 1, ..., 255, 0, 1,
 ..., 255, 0, 1,...
```

La multiplication par une puissance de deux revient à décaler les bits vers la gauche, en perdant éventuellement les bits les plus à gauche. Par exemple, 17 est égal à `0b10001`. En le multipliant par 4, on obtient `0b1000100` ou 68. Si nous multiplions 68 par 4, nous obtenons `0b100010000` ou, en tant qu'entier de 8 bits, `0b10000`. En d'autres termes, en tant qu'entiers non signés de 8 bits, nous avons que `17 * 16` est 16. Nous avons donc que `17 * 16 == 1 * 16`.

Le produit de deux entiers non nuls peut être nul. Par exemple, `16*16` est nul en tant qu'entier de 8 bits. Cela n'arrive que lorsque les deux entiers sont divisibles par deux. Le produit de deux nombres entiers impairs doit toujours être impair.

On dit que deux nombres sont copremiers si leur plus grand diviseur commun est 1. Les nombres entiers impairs sont copremiers avec les puissances de deux. Les nombres entiers pairs ne sont jamais copremiers avec une puissance de deux.

Lorsqu'on multiplie un nombre entier non nul par un nombre entier impair en utilisant une arithmétique ayant un nombre limité de bits, on n'obtient jamais zéro. Ainsi, par exemple, `3 * x` en tant qu'entier de 8 bits est nul si et seulement si `x` est nul lorsqu'on utilise des entiers non signés ayant un nombre de bits fixe. Cela signifie que `3 * x` est égal à `3 * y` si et seulement si `x` et `y` sont égaux. Ainsi, nous avons que le code Go suivant imprimera toutes les valeurs de 0 à 255, sans répétition:

```
for i:=uint8(1) ; i != 0 ; i++ {
  fmt.Println(3*i)
}
```

En mathématiques, une permutation est une bijection d'un ensemble sur lui-même, c'est-à-dire une fonction qui réarrange les éléments de cet ensemble sans en ajouter ni en supprimer. Lorsque vous mélangez les cartes d'un jeu, vous les permutez : aucune nouvelle carte n'est ajoutée et aucune carte n'est perdue. Multiplier des entiers par un entier impair en utilisant une arithmétique à bits finis les permute.

Si vous considérez les puissances d'un nombre entier impair, vous n'obtiendrez jamais un résultat nul. Cependant, vous pouvez éventuellement obtenir que la puissance soit égale à un. Par exemple, en tant qu'entier non signé de 8 bits, 3 à la puissance 64 est égal à 1. Ce nombre (64) est parfois appelé *ordre* de 3. Puisqu'il s'agit du plus petit exposant pour que le résultat soit égal à 1, nous avons que les 63 puissances précédentes donnent des résultats distincts. Nous pouvons montrer ce résultat comme suit. Supposons que 3 élevé à la puissance `p` est égal à 3 élevé à la puissance `q`, et supposons sans perte de généralité que `p>q`, alors nous avons que `3` à la puissance de `p-q` doit être 1, par inspection. Et si `p` et `q` sont tous deux inférieurs à 64, alors `p-q` doit l'être aussi, ce qui est une contradiction. De plus, on peut vérifier que les puissances d'un entier impair se répètent une fois l'ordre atteint: on a que 3 à la puissance 64 est 1, 3 à la puissance 65 est 3, 3 à la puissance 66 est 9, et ainsi de suite. Il s'ensuit que l'ordre de tout entier impair doit diviser la puissance de deux (par exemple, 256).

Quelle peut être la taille de l'ordre d'un nombre entier impair? Nous pouvons vérifier que toutes les puissances d'un nombre entier impair doivent être des nombres entiers impairs et qu'il n'existe que 128 nombres entiers 8 bits distincts. Ainsi, l'ordre d'un entier impair de 8 bits peut être au maximum de 128. Inversement, le théorème d'Euler nous dit que tout entier impair à la puissance du nombre d'entiers impairs (par exemple, 3 à la puissance 128) doit être un. Étant donné que les valeurs de la puissance d'un nombre entier impair se répètent de manière cyclique une fois l'ordre atteint, nous avons que l'ordre de tout nombre entier impair doit diviser 128 pour les entiers non signés de 8 bits. En général, quelle que soit le nombre de bits, l'ordre d'un entier impair doit être une puissance de deux.

Étant donné deux entiers non signés non nuls, `a` et `b`, on s'attendrait à ce que `a+b>max(a,b)` mais ce n'est vrai que s'il n'y a pas de débordement. Quand et seulement quand il y a un débordement, nous avons que `a+b<min(a,b)` en utilisant l'arithmétique non signée de largeur finie. On peut vérifier s'il y a un débordement avec l'une ou l'autre des conditions: `a+b<a` et `a+b<b`.

Typiquement, l'une des opérations les plus coûteuses qu'un ordinateur puisse effectuer avec deux entiers est de les diviser. Une division peut nécessiter plusieurs fois plus de cycles qu'une multiplication, et une multiplication est à son tour souvent plusieurs fois plus coûteuse qu'une simple addition ou soustraction. Cependant, la division par une puissance de deux et la multiplication par une puissance de deux sont peu coûteuses: nous pouvons calculer le quotient entier de la division d'un entier non signé en décalant les bits *à droite*. Par exemple, l'entier 7 (0b111) divisé par 2 est 0b011 ou 3. On peut encore diviser 7 (0b111) par 4 pour obtenir 0b001 ou 1. Le reste du nombre entier est donné en sélectionnant les bits qui seraient décalés: le reste de 7 divisé par 4 est 7 ET 0b11 ou 0b11. Le reste de la division par deux est juste le bit le moins significatif. Les nombres entiers pairs sont caractérisés par le fait que le bit le moins significatif est zéro. De même, la multiplication par une puissance de deux est juste un décalage vers la gauche: l'entier 7 (0b111) multiplié par deux est 14 (0b1110). Plus généralement, un compilateur optimisant peut

produire un code efficace pour le calcul du reste et du quotient lorsque le diviseur est fixe. Typiquement, [cela implique au moins une multiplication et un décalage] (https://arxiv.org/abs/1902.01961).

Étant donné un entier `x`, on dit que `y` est son inverse multiplicatif si `x * y == 1`. Nous savons que tout entier impair a un inverse multiplicatif car la multiplication par un entier crée une permutation de tous les entiers. Nous pouvons calculer cet inverse multiplicatif en utilisant la méthode de Newton. C'est-à-dire que nous commençons par une estimation et, à partir de cette estimation, nous en obtenons une meilleure, et ainsi de suite, jusqu'à ce que nous convergions naturellement vers la bonne valeur. Nous avons donc besoin d'une formule `f(y)`, afin de pouvoir répéter `y = f(y)` jusqu'à ce que `y` converge. Une formule de récurrence utile est `f(y) = y * (2 - y * x)`. Vous pouvez vérifier que si `y` est l'inverse multiplicatif de `x`, alors `f(y) = y`. Supposons que `y` n'est pas tout à fait l'inverse, supposons que `x * y = 1 + z * p` pour un certain nombre d'entiers impairs `z` et une certaine puissance de deux `p`. Si la puissance de deux est (disons) 8, cela vous indique que `y` est l'inverse multiplicatif sur les trois premiers bits. On obtient `x * f(y) = x * y * (2 - y * x) = 2 + 2 * z * p - (1 - 2 * z * p + z * z * p * p) = 1 - z * z * p * p`. On peut voir à partir de ce résultat que si `y` est l'inverse multiplicatif sur les `n` premiers bits, alors f(y) est l'inverse multiplicatif sur `2n` bits. Autrement dit, si `y` est l'inverse *des `n` premiers bits*, alors `f(y)` est l'inverse *des `2n` premiers bits*. Nous doublons la précision à chaque fois que nous appelons la formule de récurrence. Cela signifie que nous pouvons converger rapidement vers l'inverse.

Quelle devrait être notre estimation initiale pour `y`? Si on utilise des mots de 3 bits, alors chaque nombre est son inverse. Donc, en commençant par `y = x`, on obtient une précision de trois bits, mais on peut faire mieux: `( 3 * x ) ^ 2` donne 5 bits de précision. Le programme Go suivant vérifie cette affirmation:

```
package main

import fmt
```

```
func main() {
  for x := 1 ; x < 32 ; x += 2 {
    y := (3 * x) ^ 2
    if (x*y)&0b11111 != 1 {
      fmt.Println("erreur")
    }
  }
  fmt.Println("terminé")
}
```

Observez comment nous capturons les 5 bits les moins significatifs à l'aide de l'expression `&0b11111`: il s'agit d'une opération logique ET par bit.

En partant de 5 bits, le premier appel à la formule de récurrence donne 10 bits, puis 20 bits pour le second appel, puis 40 bits, puis 80 bits. Ainsi, nous devons appeler notre formule de récurrence 2 fois pour des valeurs de 16 bits, 3 fois pour des valeurs de 32 bits et 4 fois pour des valeurs de 64 bits. La fonction `FindInverse64` calcule l'inverse multiplicatif sur 64 bits d'un nombre entier impair:

```
func f64(x, y uint64) uint64 {
  return y * (2 - y*x)
}

func FindInverse64(x uint64) uint64 {
  y := (3 * x) ^ 2 // 5 bits
  y = f64(x, y) // 10 bits
  y = f64(x, y) // 20 bits
  y = f64(x, y) // 40 bits
  y = f64(x, y) // 80 bits
  return y
}
```

Nous constatons que `FindInverse64(271) * 271 == 1`. Il est important de noter qu'il échoue si l'entier fourni est pair.

Nous pouvons utiliser les inverses multiplicatifs pour remplacer la division par un entier impair par une multiplication. C'est-à-dire que si vous calculez au préalable `FindInverse64(3)`, alors vous pouvez calculer la division par trois pour tout multiple de trois en calculant le produit: par exemple, `FindInverse64(3) * 15 == 5`.

Lorsque nous stockons des valeurs multi-octets telles que des entiers non signés dans des tableaux d'octets, nous pouvons utiliser l'une des deux conventions suivantes: *little endian* et *big endian*. Les variantes *little endian* et *big endian* ne diffèrent que par l'ordre des octets: on commence soit par les octets les moins significatifs (*little endian*), soit par les octets les plus significatifs (*big endian*). Considérons le nombre entier 12345. En valeur hexadécimale, il vaut 0x3039. Si nous le stockons sous forme de deux octets, nous pouvons soit le stocker sous la forme de la valeur d'octet 0x30 suivie de la valeur d'octet 0x39 (*big endian*), soit l'inverse (0x39 suivi de 0x30). La plupart des systèmes modernes utilisent par défaut la convention *little endian*, et il existe relativement peu de systèmes *big endian*. En pratique, nous devons rarement nous préoccuper de l'endiannité de notre système.

Les entiers signés et le complément à deux

Étant donné les entiers non signés, comment représenter des entiers signés? À première vue, il est tentant de réserver un bit pour le signe. Ainsi, si nous disposons de 32 bits, nous pourrions utiliser un bit pour indiquer si la valeur est positive ou négative, puis nous pourrions utiliser 31 bits pour stocker la valeur absolue de l'entier.

Bien que cette approche par bit de signe soit viable, elle présente des inconvénients. Le premier inconvénient évident est qu'il existe deux valeurs zéro possibles: `+0` et `-0`. L'autre inconvénient est qu'elle fait des entiers signés des valeurs totalement distinctes de celles des entiers non signés: idéalement, nous voudrions que les instructions matérielles qui fonctionnent sur des entiers non signés *fonctionnent directement* sur des entiers signés.

C'est pourquoi les ordinateurs modernes utilisent la notation du complément à deux pour représenter les entiers signés. Pour simplifier l'exposé, nous considérons des entiers de 8 bits. Nous représentons tous les entiers positifs jusqu'à la moitié de l'intervalle (127 pour les mots de 8 bits) de la même manière, que nous utilisions des entiers signés ou non signés. Ce n'est que lorsque le bit le plus significatif est activé que nous différons: pour les entiers signés, c'est comme si la valeur non signée dérivée de tous les bits sauf le bit le plus significatif était soustraite de la moitié de la plage (128). Par exemple, en tant que valeur signée de 8 bits, 0b1111111111 est -1. En effet, en ignorant le bit le plus significatif, on a 0b1111111 ou 127, et en soustrayant 128, on obtient -1.

Binaire	non signé	signé
0b00000000	0	0
0b00000001	1	1
0b00000010	2	2
0b01111111	127	127
0b10000000	128	-128
0b10000001	129	-127
0b11111110	254	-2
0b11111111	255	-1

Observez comment les entiers impairs sont mis en correspondance avec les entiers impairs et vice versa. En effet, 254 est interprété comme -2, 255 comme -1.

En Go, vous pouvez convertir des entiers non signés en entiers signés, et vice versa: Go laisse les valeurs binaires inchangées, mais il réinterprète simplement la valeur en entiers non signés et signés. Si nous exécutons le code suivant, nous avons que `x==z`:

```
x := uint16(52429)
y := int16(x)
z := uint16(y)
```

De manière pratique, que l'on calcule la multiplication, l'addition ou la soustraction entre deux valeurs, le résultat est le même (en binaire), que l'on interprète les bits comme une valeur signée ou non signée. Cette particularité permet de réutiliser les mêmes circuits matériels.

Un inconvénient de la notation du complément à deux est que l'on ne peut pas changer le signe de la plus petite valeur négative (-128 dans le cas de 8 bits). En effet, le nombre 128 ne peut pas être représenté à l'aide d'entiers signés sur 8 bits. Cette asymétrie est inévitable car nous avons trois types de nombres: le zéro, les valeurs négatives et les valeurs positives. Or, nous avons un nombre pair de valeurs binaires.

Comme pour les entiers non signés, nous pouvons décaler (à droite et à gauche) les entiers signés. Le décalage à gauche fonctionne comme pour les entiers non signés au niveau du bit. Nous avons donc

```
x := int8(1)
(x << 1) == 2
(x << 7) == -128
```

Cependant, le décalage à droite fonctionne différemment pour les entiers signés et non signés. Pour les entiers non signés, on décale les zéros par la gauche ; pour les entiers signés, on décale soit les zéros (si l'entier est positif ou nul), soit les uns (si l'entier et négatif). Nous illustrons ce comportement par un exemple:

```
  x := int8(-1)
  (x >> 1) == -1
  y := uint8(x)
  y == 255
  (y >> 1) == 127
```

Lorsqu'un nombre entier signé est positif, la division par une puissance de deux ou le décalage vers la droite donne le même résultat (`10/4 == (10>>2)`). Cependant, lorsque l'entier est négatif, cela n'est vrai que si l'entier négatif est divisible par une puissance de deux. Lorsque le nombre entier négatif n'est pas divisible par une puissance de deux, alors

le décalage est plus petit d'une unité que la division, comme l'illustre le code suivant:

```
  x := int8(-10)
  (x / 4) == -2
  (x >> 2) == -3
```

En Go, le reste de la division entre un entier négatif et un entier positif est nul ou négatif. En effet, on a que `(-a)/b == - (a/b)` et `(-a)%b == -(a%b)`. Par exemple, on a `(-5)//2 == - (5//2)` et `-1%2 == -(1%2) == -1`. Pour un entier positif, on peut vérifier s'il est impair avec l'expression `x%2 == 1`, mais cette expression ne fonctionne plus avec les entiers négatifs puisque le reste de la division sera négatif: `-3%2 == -1%`. On peut vérifier si un nombre entier est impair en vérifiant la valeur du bit le moins significatif: `x&1==1` pour les nombres impairs, qu'ils soient négatifs ou positifs.

Pour les entiers positifs ou non signés, on a que `(2*x + 1)/2==x`. Cependant, pour les entiers négatifs, c'est faux: `(2*-1 + 1)/2 == (-1)/2 == 0`. Plus généralement, on a que `(2*x - 1)/2 == x` lorsque `x` est négatif, et `(2*x + 1)/2 == x` lorsque `x` est positif. Le programme suivant illustre cette observation:

```
package main

import fmt

func Sign(a int) int {
  switch {
  Si a < 0 :
    return -1
  cas a > 0 :
    return +1
  }
  return 0
}
```

```
func main() {
  for x := -10 ; x < 10 ; x++ {
    fmt.Println(x, (2*x+Sign(x))/2)
  }
}
```

Cependant, nous avons l'identité `2*(x>>1)+(x&1)` que `x` soit positif ou négatif.

Nombres à virgule flottante

Au sein des ordinateurs, les nombres réels sont généralement modélisés par des nombres binaires à virgule flottante: un entier de largeur fixe `m` (la *significande*) multiplié par 2 élevé à un exposant entier `p`: `m * 2**p` où `2**p` représente le nombre deux élevé à la puissance `p`. Un bit signé est ajouté pour que les zéros positifs et négatifs soient disponibles. La plupart des systèmes actuels suivent la norme IEEE 754, ce qui signifie que vous pouvez obtenir des résultats cohérents quels que soient les langages de programmation et les systèmes d'exploitation. Par conséquent, il importe peu que vous implémentiez votre logiciel en C++ sous Linux alors que quelqu'un d'autre l'implémente en C# sous Windows: si vous disposez tous deux de systèmes récents, vous pouvez vous attendre à des résultats numériques identiques lors des opérations arithmétiques et de racine carrée de base.

Un nombre à virgule flottante positif *normal* à double précision est un nombre à virgule flottante binaire où l'entier de 53 bits `m` est dans l'intervalle `[2**52,2**53)` tout en étant interprété comme un nombre dans `[1,2)` en le divisant virtuellement par `2**52`, et où l'exposant de 11 bits `p` est compris entre de `-1022` à `+1023`. Nous pouvons donc représenter toutes les valeurs comprises entre `2**-1022` et jusqu'à `2**1024` non compris. Certaines valeurs inférieures à `2**-1022` peuvent être représentées comme des valeurs *subnormales*: elles utilisent un code spécial d'exposant qui a la valeur `2**-1022` et la significande est alors interprétée comme

une valeur dans l'intervalle $[0, 1)$.

Dans le langage Go, un nombre de type `float64` peut représenter tous les nombres décimaux composés d'une significande de 15 chiffres, de `-1,8 * 10**308` à `1,8 *10**308` environ. L'inverse n'est pas vrai: il ne suffit pas d'avoir 15 chiffres de précision pour distinguer deux nombres à virgule flottante quelconques: on peut avoir besoin de jusqu'à 17 chiffres.

Le type `float32` est similaire. Il permet de représenter tous les nombres compris entre `2**-126` et, sans les inclure, `2**128` ; avec une gestion spéciale pour certains nombres plus petits que `2**-126` (subnormaux). Le type `float32` peut représenter exactement tous les nombres décimaux composés d'une significande décimale à 6 chiffres, mais 9 chiffres sont nécessaires en général pour identifier de manière unique un nombre.

Les nombres à virgule flottante comprennent également l'infini positif et négatif, ainsi qu'une valeur spéciale non numérique. Ils sont identifiés par une valeur d'exposant réservée.

Les nombres sont généralement sérialisés sous forme de nombres décimaux dans des chaînes de caractères, puis analysés par le récepteur. Cependant, il est généralement impossible de convertir des nombres décimaux en nombres binaires à virgule flottante: le nombre 1/5 n'a pas de représentation exacte en tant que nombre binaire à virgule flottante. Cependant, vous devez vous attendre à ce que le système choisisse la meilleure approximation possible: `7205759403792794 * 2**-55` comme nombre `float64` (ou environ `0,20000000000000001110`). Si le nombre initial était un `float64` (par exemple), vous devriez vous attendre à ce que la valeur exacte soit préservée: elle fonctionnera comme prévu dans Go.

Les zéros positifs et négatifs peuvent être difficiles à distinguer car ils sont considérés comme des valeurs égales. Heureusement, la fonction `math.Signbit` ne renvoie `true` que lorsque la valeur est négative. Considérez le programme suivant:

```
package main

import (
  "fmt"
  "math"
)

func main() {
  var zero float64 = 0.0
  var negzero float64 = -zero
  fmt.Println(zero == negzero)
  fmt.Println(math.Signbit(zero))
  fmt.Println(math.Signbit(negzero))
  fmt.Println(1 / zero)
  fmt.Println(1 / negzero)
}
```

Il produira:

```
true
false
true
+Inf
-Inf
```

On pourrait s'attendre à ce que la multiplication d'une valeur par zéro donne la valeur zéro, et que l'ajout de zéro ne change jamais le signe d'une valeur. Malheureusement, nous avons que `0 * Inf` est `NaN`(not-a-number), une valeur spéciale qui peut être considérée comme un code d'erreur. Nous avons que `-0.0 + 0.0` est `0.0` et ajouter `0.0` à `-0.0` change son signe. Le programme suivant illustre ces identités:

```
package main

import (
  "fmt"
```

```
  "math"
)

func main() {
  var zero float64 = 0.0
  var negzero float64 = -zero

  fmt.Println(math.Signbit(negzero + zero))
  fmt.Println((1 / zéro) * zéro)
}
```

Il peut être gênant de représenter des nombres binaires à virgule flottante sous forme de valeurs décimales, car il n'existe pas de conversion exacte en général. À cette fin, Go et d'autres langages de programmation nous permettent d'imprimer des nombres au format hexadécimal à virgule flottante. La notation hexadécimale flottante est prise en charge par les langages de programmation C (C99), C++ (C++17), Swift, Java, Julia et Go. Comme dans la notation hexadécimale habituelle pour les nombres entiers, nous commençons la chaîne par 0x suivi de la significande sous forme hexadécimale. Chaque caractère hexadécimal (0-9, A-F) représente 4 bits (un quartet). Nous ajoutons le suffixe p suivi de l'exposant (par exemple, p4 ou p-4). En option, nous pouvons ajouter un point hexadécimal dans le significande. Avec un point décimal, nous interprétons la fraction décimale en la divisant par la puissance de dix appropriée. Par exemple, on écrit 1,45 = 145/100. Le point hexadécimal fonctionne de la même manière. Ainsi, 0x1.FCp17 signifie 0x1FC/256 fois deux à la puissance 17 ou 260096 où nous divisons 0x1FC par 256 parce qu'il y a deux quartets (16 * 16) après le point binaire. Chaque quartet après le point nécessite une division par 16. Lorsque la valeur est un nombre normal à virgule flottante de 64 bits, le significande peut être exprimé comme un 1 suivi d'un maximum de 52 bits après le point, soit 13 caractères hexadécimaux. Ainsi, 9 000 000 000 000 000 peut être écrit comme 0x1.ff973cafa8p+52. Sans surprise, le chiffre 1 s'écrit 0x1p0. Le nombre 1/5 (0,2) demande plus de précautions. Nous pouvons

l'approximer le plus possible comme un nombre à virgule flottante de 64 bits en divisant 7205759403792794 par 2 à la puissance 55. Le résultat est alors légèrement supérieur à 0,2: il est exactement égal à 0,200000000000000038857805861880478914827108383178710937 5. Il n'est pas possible de représenter exactement à 0,2 en utilisant des nombres binaires à virgule flottante. Le nombre 7205759403792794 est 0x19999999999a en hexadécimal, de sorte que 0,2 est environ 0x1,999999999999ap-03. Le programme suivant imprime 0x1.999999999999ap-03 et 0x1p+00. En effet, la fonction `strconv.FormatFloat` peut traduire n'importe quel nombre en un nombre hexadécimal à virgule flottante: dans ce cas, nous passons les paramètres 'x' pour l'hexadécimal, -1 pour indiquer que nous voulons une précision complète et 64 pour indiquer que nous utilisons une représentation de 64 bits. Il est important de noter que la valeur 0x1.999999999999ap-03 est une représentation exacte de ce qui est stocké dans votre logiciel lorsque vous entrez 0,2 et que vous le représentez comme un nombre binaire standard à virgule flottante.

```
package main

import (
  "fmt"
  "strconv"
)

func main() {
  fmt.Println(strconv.FormatFloat(0.2, 'x', -1, 64))
  fmt.Println(strconv.FormatFloat(1, 'x', -1, 64))

}
```

Tableaux

En Go comme dans la plupart des langages de programmation, on peut organiser les données en tableau. Par exemple, on peut créer un tableau composé de 3 entiers, comme ceci:

```
  y := [3]int{1, 2, 3}
```

Dans ce cas précis, on fournit à Go le contenu du tableau avec une liste d'entiers. Si on omet cette liste d'entiers, les éléments seront initialisés avec la valeur zéro. On peut créer des tableaux avec différents types. Tant que le tableau est relativement petit et que sa taille est connue au moment d'écrire le code, on peut sans problème utiliser la syntaxe de déclaration contenant une taille fixe.

En Go, le passage des valeurs à une fonction se fait généralement *par valeur* ce qui signifie que Go fait une copie. La fonction suivante fait une copie des trois valeurs entières à chaque appel:

```
func g(z [3]int) int {
  return z[1]
}
```

En Go, on préfère donc utiliser la notion de *slice*: une *slice* s'utilise comme un tableau, mais dont la taille n'est pas fixée. Au sein de la mémoire, il ne s'agit que d'une référence à une section d'un tableau qui peut être implicite. On peut créer un tableau avec une taille fixée lors de l'exécution avec la fonction `make`:

```
func f2(n int) int {
  y := make([]int, n)
  return y[0]
}
```

La fonction `make` initialise les valeurs avec un zéro. Un tableau *caché* est créé. Lorsqu'on passe une *slice* à une fonction, les données elles-mêmes ne sont pas copiées, on ne fait que copier la référence:

```
func g2(z []int) int {
  return z[1]
}
```

La fonction `append` permet d'ajouter des éléments à une *slice*. Cette fonction ajoute la valeur '1' et retourne la nouvelle *slice* comprenant la valeur supplémentaire:

```
func g3(z []int) []int {
  return append(z, 1)
}
```

L'appel de la fonction `g3` ne modifie pas la *slice* initiale. On peut donc avoir plusieurs *slice* partageant les mêmes données. Par exemple, le tableau suivant créé une *slice* `z` contenant un seul élément (la valeur `3`):

```
  y := make([]int, 3)
  y[2] = 3
  z := y[2:3]
```

Chaînes de caractères

L'une des premières normes pour représenter les chaînes de caractères est la norme ASCII: elle a été spécifiée pour la première fois au début des années 1960. La norme ASCII reste populaire. Chaque caractère est un octet, dont le bit le plus significatif est mis à zéro. Il n'existe donc que 128 caractères ASCII distincts. C'est souvent suffisant pour des tâches simples comme la programmation. Malheureusement, la norme ASCII ne peut représenter que 128 caractères au maximum, soit beaucoup moins que nécessaire.

Le programme suivant affiche tous les caractères ASCII imprimables.

```
package main

import (
```

```
  "fmt"
)

func main() {
  for i := 32; i < 127; i++ {
    var buffer []byte = make([]byte, 1)
    buffer[0] = byte(i)
    fmt.Println(i, "'"+string(buffer)+"'")
  }
}
```

Le programme contient une boucle for, qui itère de 32 à 126 (inclus). Nous déclarons une variable tampon, qui est une tranche d'octets. Une tranche est une séquence d'éléments de même type dont la taille est dynamique. Un octet est un alias de uint8, qui est un entier non signé de 8 bits. La fonction make crée une tranche avec une longueur et une capacité données. Dans le cas présent, la longueur et la capacité sont toutes deux égales à 1, ce qui signifie que la tranche ne comporte qu'un seul élément. Nous attribuons la valeur de i convertie en octet au premier élément de la tranche de tampon. La fonction byte convertit un entier en octet, en le tronquant si nécessaire. La huitième ligne imprime la valeur de i, suivie d'un guillemet simple, suivi de la représentation sous forme de chaîne de la tranche de tampon, suivie d'un autre guillemet simple, suivi d'une nouvelle ligne. La fonction fmt.Println imprime ses arguments sur la sortie standard, en les séparant par des espaces et en les terminant par un retour à la ligne. La fonction string convertit une tranche d'octets en une chaîne de caractères, en interprétant chaque octet comme un caractère encodé UTF-8. La sortie du programme est un tableau des codes ASCII et des caractères correspondants, de 32 (espace) à 126 (tilde). Il existe des caractères de 0 à 32, mais il s'agit de caractères de contrôle utilisés pour les fins de ligne et à d'autres fins spéciales. Le dernier caractère ASCII (127) est également un caractère de contrôle (DEL).

De nombreuses normes divergentes sont apparues pour représenter les caractères dans les logiciels. L'existence de multiples formats incompatibles

a rendu difficile la production de logiciels localisés interopérables.

Les ingénieurs ont développé Unicode à la fin des années 1980 pour tenter de fournir une norme universelle. Au départ, on pensait que l'utilisation de 16 bits par caractère serait suffisante, mais cette conviction était erronée. La norme Unicode a été étendue pour inclure jusqu'à 1 114 112 caractères. Seule une petite fraction de tous les caractères possibles a été attribuée, mais d'autres le sont au fil du temps, à chaque révision d'Unicode. La norme Unicode est une extension de la norme ASCII: les 128 premiers caractères Unicode correspondent aux caractères ASCII.

En raison de l'attente initiale qu'Unicode tienne dans un espace de 16 bits, un format basé sur des mots de 16 bits (UTF-16) a été publié en 1996. Il peut utiliser soit 16 bits, soit 32 bits par caractère. Le format UTF-16 a été adopté par des langages de programmation tels que Java, et est devenu une valeur par défaut sous Windows. Malheureusement, l'UTF-16 n'est pas rétrocompatible avec l'ASCII en tant que format binaire. Un format compatible avec l'ASCII a été proposé et officialisé en 2003: UTF-8. Au fil du temps, l'UTF-8 est devenu largement utilisé pour les formats d'échange de texte tels que JSON, HTML ou XML. Les langages de programmation tels que Go, Rust et Swift utilisent UTF-8 par défaut. Les deux formats (UTF-8 et UTF-16) nécessitent une validation: tous les tableaux d'octets ne sont pas valides. Le format UTF-8 est plus coûteux à valider.

Les caractères ASCII nécessitent un octet en UTF-8 et deux octets en UTF-16. Le format UTF-16 peut représenter tous les caractères, à l'exception des caractères supplémentaires tels que les emojis, en utilisant deux octets. Le format UTF-8 utilise deux octets pour les alphabets latin, hébreu et arabe, trois octets pour les caractères asiatiques et quatre octets pour les caractères supplémentaires.

L'UTF-8 code les valeurs en séquences de un à quatre octets. Le premier octet d'une séquence est appelé octet de tête ; le bit le plus significatif de l'octet de tête indique la longueur de la séquence:

— Si le bit le plus significatif est zéro, nous avons une séquence d'un

octet (ASCII).
— Si les trois bits les plus significatifs sont 0b110, nous avons une séquence de deux octets.
— Si les quatre bits les plus significatifs sont 0b1110, nous avons une séquence de trois octets.
— Enfin, si les cinq bits les plus significatifs sont 0b11110, nous avons une séquence de quatre octets.

Tous les octets qui suivent l'octet de tête dans une séquence sont des octets de continuation et leurs bits les plus significatifs doivent être 0b10. À l'exception des bits les plus significatifs requis, la valeur numérique du caractère (entre 0 et 1 114 112) est stockée en commençant par les bits les plus significatifs (dans l'octet de tête) suivis des bits les moins significatifs dans les autres octets de continuation.

Le programme suivant en Go va afficher tous les caractères Unicode qui utilisent deux octets en UTF-8:

```
package main

import (
  "fmt"
)

func main() {
  for i := 0b00010; i <= 0b11111; i++ {
    for j := 0; j <= 0b111111; j++ {
      var buffer []byte = make([]byte, 2)
      buffer[0] = byte(i | 0b11000000)
      buffer[1] = byte(j | 0b10000000)
      fmt.Println((i<<6)+j, "'"+string(buffer)+"'")
    }
  }
}
```

Ce programme Go génère et affiche tous les caractères Unicode encodés

en UTF-8 sur deux octets, correspondant aux points de code de U+0080 à U+07FF. Il utilise deux boucles imbriquées : la boucle externe (`i`) parcourt les valeurs binaires de 0b00010 à 0b11111 (2 à 31), représentant les 5 bits significatifs du premier octet, et la boucle interne (`j`) parcourt les valeurs de 0 à 0b111111 (0 à 63), représentant les 6 bits du second octet. Pour chaque combinaison, un tampon de deux octets est créé. Le premier octet est formé en combinant `i` avec le préfixe 0b11000000 (conformément au format UTF-8 pour deux octets, 110xxxxx), et le second octet combine `j` avec 0b10000000 (préfixe de continuation, 10xxxxxx). Le programme affiche ensuite le point de code Unicode calculé comme `(i<<6)+j` (décalage de 6 bits pour `i` plus `j`) et le caractère correspondant obtenu en convertissant le tampon en chaîne. Cela permet d'afficher les 1 920 caractères UTF-8 à deux octets.

Le programme suivant illustrate le fait qu'au sein d'une même chaîne de caractères UTF-8, certains caractères ne nécessitent qu'un octet ('L') alors que d'autres en utilisent deux ('é') :

```
package main

import (
  "fmt"
)

func main() {
  var str string
  str = "L'été est arrivé"
  fmt.Println([]byte(str))
  str2 := string(str[0:1])
  fmt.Println(str2)
  str3 := string(str[1:2])
  fmt.Println(str3)
  str4 := string(str[2:4])
  fmt.Println(str4)
}
```

Dans le format UTF-16, les caractères compris entre 0x0000 et 0xD7FF et entre 0xE000 et 0xFFFF sont stockés sous forme de mots simples de 16 bits. Les caractères compris entre 0x010000 et 0x10FFFF nécessitent deux mots de 16 bits appelés paire de substituts. Le premier mot de la paire est compris entre 0xd800 et 0xdbff tandis que le second mot est compris entre 0xdc00 et 0xdfff. La valeur du caractère est constituée des 10 bits les moins significatifs des deux mots, en utilisant le deuxième mot comme moins significatif, et en ajoutant 0x10000 au résultat. Il existe deux types de format UTF-16. Dans la variante *little-endian*, chaque mot de 16 bits est stocké en utilisant les bits les moins significatifs du premier octet. L'inverse est vrai dans la variante *big-endian*.

En utilisant l'ASCII, il est relativement facile d'accéder aux caractères dans un ordre aléatoire. Pour l'UTF-16, c'est possible si l'on suppose qu'il n'y a pas de caractères supplémentaires, mais comme certains caractères peuvent nécessiter 4 octets et d'autres 2 octets, il n'est donc pas possible d'aller directement à un caractère par son index sans accéder au contenu précédent. De même, l'UTF-8 n'est pas accessible de manière aléatoire en général.

Les logiciels dépendent souvent des paramètres régionaux: par exemple, l'anglais américain, le français canadien, et ainsi de suite. Le tri des chaînes de caractères dépend des paramètres régionaux. Il n'est généralement pas possible de trier des chaînes de caractères sans connaître les paramètres régionaux. Cependant, il est possible de trier les chaînes de manière lexicographique sous forme de séquences d'octets (UTF-8) ou de séquences de mots de 16 bits (UTF-16). Dans le cas du format UTF-8, le résultat est alors un tri de chaînes de caractères basé sur la valeur numérique des caractères.

Pointeurs

Au moins conceptuellement, les données dans un logiciel sont stockées à un emplacement dans la mémoire. Ainsi, chaque variable, chaque élément d'un tableau est associé à une adresse mémoire. Cette abstraction est utile

mais n'est pas tout à fait exacte en pratique. Par exemple, un pointeur peut contenir une adresse invalide. Néanmoins, un pointeur est un concept utile en programmation logicielle, lorsqu'il est utilisé avec précaution.

La syntaxe générale consiste à placer le caractère esperluette avant une valeur (`&`) pour obtenir un pointeur, et le type de pointeur résultant est créé en utilisant le préfixe astérisque `*`. À l'inverse, le préfixe astérisque permet d'accéder à la valeur pointée. Le programme suivant affiche `1`:

```
package main
import (
    "fmt"
)
func f(v *int) {
    // f reçoit un pointeur vers une valeur entière
    *v = 1 // on définit la valeur pointée à 1
}
func main() {
    x := make([]int, 10)
    f(&x[0]) // passe un pointeur vers le premier élément
    fmt.Println(x[0])
}
```

Dans le programme, la fonction `f` modifie la valeur de `x[0]` en utilisant un pointeur, ce qui évite de passer une copie de la valeur. Cela est particulièrement utile pour les structures de données volumineuses, comme une structure contenant de nombreux champs, où copier l'ensemble des données serait coûteux en termes de performances. Par exemple, en Go, une structure comme `type Person struct { Name string; Age int }` peut être passée par pointeur à une fonction pour modifier ses champs sans dupliquer la structure entière.

Cependant, les pointeurs introduisent des risques, notamment les pointeurs nuls. En Go, un pointeur non initialisé a la valeur `nil`, ce qui signifie qu'il ne pointe vers aucune adresse mémoire valide. Si une fonction tente de déréférencer un pointeur `nil`, comme dans `*ptr = 5` où `ptr` est `nil`,

le programme plantera avec une erreur d'exécution (panic). Pour éviter cela, il est courant de vérifier si un pointeur est `nil` avant de l'utiliser. Par exemple, une fonction manipulant un pointeur pourrait inclure une vérification comme `if v == nil { return }` pour gérer les cas où aucun pointeur valide n'est fourni. Par exemple, le programme suivant se terminera avec une erreur fatale.

```
package main

import "fmt"

func main() {
  var i *int
  fmt.Println(*i)
}
```

Pour illustrer une autre difficulté, nous allons créer une situation où un pointeur fait référence à une valeur qui n'est plus accessible, en raison de la portée des variables et de la gestion de la mémoire en Go.

```
package main

import "fmt"

func createPointer() *int {
    x := 42
    return &x
}

func main() {
    ptr := createPointer()
    fmt.Println(ptr)
    fmt.Println(*ptr)
}
```

Dans ce programme, la fonction `createPointer` déclare une variable

locale `x` et retourne son adresse sous forme de pointeur. Cependant, comme `x` est une variable locale, elle est allouée sur la pile (stack) et sa portée se limite à la fonction `createPointer`. Une fois que la fonction se termine, la mémoire associée à `x` peut être réutilisée ou considérée comme invalide par le ramasse-miettes de Go. Le pointeur `ptr` dans `main` devient un pointeur pendant, car il fait référence à une adresse mémoire qui n'est plus garantie d'être valide.

En Go, tenter de déréférencer `ptr` avec `*ptr` dans `main` peut provoquer un comportement indéfini ou, dans certains cas, une erreur d'exécution (panic), car la mémoire pointée n'est plus accessible. Vous remarquerez peut-être que, bien que le programme soit incorrect, il peut s'exécuter sans erreur apparente. C'est pourquoi les erreurs produites par une mauvaise utilisation d'un pointeur sont si difficiles. Un plantage peut se produire de manière aléatoire.

Pour illustrer un scénario plus pratique, considérons un cas où une fonction retourne un pointeur vers une structure allouée localement.

```
package main

import "fmt"

type Person struct {
    Nom string
    Age int
}

func createPerson() *Person {
    p := Person{Nom: "Alice", Age: 30}
    return &p
}

func main() {
    ptr := createPerson()
    fmt.Println(ptr)
```

```
    fmt.Println(ptr.Nom) // Comportement indéfini ou panic
}
```

Ici, `createPerson` crée une structure `Person` localement et retourne un pointeur vers celle-ci. Comme `p` est une variable locale, sa mémoire peut être libérée ou réutilisée après la fin de la fonction. Lorsque `main` tente d'accéder à `ptr.Nom`, le programme risque de planter ou d'afficher des données corrompues, car l'adresse pointée par `ptr` n'est plus garantie d'être valide.

Pour éviter les pointeurs pendants, une bonne pratique en Go consiste à allouer la mémoire sur le tas (heap) en utilisant `new` ou en créant des objets qui persisteront au-delà de la portée de la fonction. Voici une version corrigée du deuxième exemple.

```
package main

import "fmt"

type Person struct {
    Nom string
    Age int
}

func createPerson() *Person {
    p := new(Person)
    p.Nom = "Alice"
    p.Age = 30
    return p
}

func main() {
    ptr := createPerson()
    fmt.Println(ptr)
    fmt.Println(ptr.Nom)
```

```
}
```

Dans cette version, `new(Person)` alloue la structure sur le tas, et le ramasse-miettes de Go garantit que la mémoire reste valide tant que `ptr` y fait référence. Cela élimine le risque de pointeur pendant, car la mémoire n'est pas libérée prématurément.

Structures, interfaces et méthodes

En Go, les structures de données complexes sont souvent construites à l'aide de structures (`struct`), d'interfaces et de méthodes, qui offrent une approche flexible et modulaire pour organiser et manipuler les données. Une structure permet de regrouper plusieurs champs de différents types sous une entité unique. Par exemple, pour représenter un nœud dans une liste chaînée, on pourrait définir :

```
package main

import "fmt"

type Node struct {
    Valeur int
    Suivant *Node
}

func main() {
    n := Node{Valeur: 10, Suivant: nil}
    fmt.Println(n.Valeur)
}
```

Ici, `Node` est une structure représentant un nœud avec une valeur entière et un pointeur vers le nœud suivant. Les structures sont particulièrement utiles pour les structures de données comme les listes chaînées, les arbres ou les graphes, où les relations entre les éléments sont définies par des pointeurs.

Les méthodes en Go permettent d'associer un comportement à une structure. Une méthode est une fonction avec un récepteur spécial, qui peut être une structure ou un pointeur vers une structure. Par exemple, pour ajouter une méthode à `Node` qui modifie sa valeur :

```go
package main

import "fmt"

type Node struct {
    Valeur int
    Suivant *Node
}

func (n *Node) UpdateValeur(nouvelleValeur int) {
    n.Valeur = nouvelleValeur
}

func main() {
    n := Node{Valeur: 10, Suivant: nil}
    n.UpdateValeur(20)
    fmt.Println(n.Valeur) // Affiche 20
}
```

Dans cet exemple, `UpdateValeur` utilise un récepteur pointeur (`*Node`) pour modifier directement la structure. Si le récepteur était `n Node`, la méthode recevrait une copie de la structure, et les modifications ne seraient pas répercutées sur l'original. Les récepteurs pointeurs sont courants dans les structures de données pour éviter les copies coûteuses et permettre des modifications en place, comme dans une liste chaînée où un nœud doit être mis à jour.

Les interfaces en Go jouent un rôle clé dans la création de structures de données génériques et réutilisables. Une interface définit un ensemble de méthodes qu'un type doit implémenter, sans spécifier comment. Cela permet de travailler avec différentes structures de données de manière

uniforme. Par exemple, pour une structure de données comme une pile, on pourrait définir l'interface suivante.

```
package main

import "fmt"

type Pile interface {
    Push(valeur int)
    Pop() (int, bool)
}

type PileTableau struct {
    elements []int
}

func (p *PileTableau) Push(valeur int) {
    p.elements = append(p.elements, valeur)
}

func (p *PileTableau) Pop() (int, bool) {
    if len(p.elements) == 0 {
        return 0, false
    }
    valeur := p.elements[len(p.elements)-1]
    p.elements = p.elements[:len(p.elements)-1]
    return valeur, true
}

func main() {
    var pile Pile = &PileTableau{}
    pile.Push(5)
    pile.Push(10)
    if valeur, ok := pile.Pop(); ok {
        fmt.Println("Valeur dépilée :", valeur)
```

```
    }
}
```

Dans cet exemple, l'interface `Pile` définit les méthodes `Push` et `Pop`, et `PileTableau` implémente cette interface en utilisant un tableau comme structure sous-jacente. Une autre mise en œuvre pourrait utiliser une liste chaînée, mais tant qu'elle implémente `Push` et `Pop`, elle serait compatible avec l'interface `Pile`. Cela permet de manipuler différentes implémentations de piles de manière interchangeable, un concept essentiel pour des structures de données comme les files, les arbres ou les tables de hachage.

Les interfaces favorisent également la composition. Par exemple, une structure de données comme un arbre binaire de recherche pourrait mettre en œuvre plusieurs interfaces, comme une pour la recherche et une pour l'insertion, permettant une réutilisation modulaire du code. De plus, les interfaces combinées avec des pointeurs permettent des modifications efficaces des structures de données, car elles évitent les copies tout en offrant une abstraction claire.

Exercices du chapitre 3

Question 1

Écrivez une fonction Go qui calcule l'inverse multiplicatif d'un entier non signé de 16 bits. Vérifiez que votre fonction est correcte en la testant sur tous les entiers de 16 bits.

Question 2

Étant donné l'inverse multiplicatif de `5` sur 16 bits, écrivez un programme qui calcule `inverse(5) * x` pour tous les entiers non signés de 16 bits `x`. En examinant le résultat, trouvez un moyen de tester si `x` est un multiple de `5` en utilisant une seule multiplication et une seule comparaison.

Question 3

Transformez l'inverse multiplicatif non signé de 16 bits de `5` en un entier signé de 16 bits et appelez le résultat `z`. Écrivez un programme qui calcule `z * x` pour tous les entiers signés de 16 bits `x`. Expliquez le résultat.

Question 4

Étant donné une chaîne UTF-8 potentiellement tronquée sous forme de tableau d'octets (`[]byte`), écrivez une fonction qui trouve l'emplacement du dernier caractère valide.

Chapitre 4

À un niveau fondamental, un programmeur doit manipuler des bits. Les processeurs modernes traitent les données en les chargeant dans des "registres" et non dans des bits individuels. Ainsi, un programmeur doit savoir comment manipuler les bits dans un registre. En général, nous pouvons le faire en programmant avec des entiers de 8, 16, 32 et 64 bits. Par exemple, supposons que je veuille mettre un bit individuel à la valeur 1. Prenons le bit à l'index 12 dans un mot de 64 bits. Le mot avec juste le bit à l'index 12 est `1<<12`: le nombre 1 décalé vers la gauche 12 fois, ou 4096. En Go, nous formatons les nombres en utilisant la fonction `fmt.Printf`: nous utilisons une chaîne de caractères avec des instructions de formatage suivies des valeurs que nous voulons imprimer. On commence une séquence de formatage par la lettre `%` qui a une signification particulière (si on veut imprimer `%`, on doit utiliser la chaîne `%%`). Elle peut être suivie par la lettre `b` qui signifie binaire, la lettre `d` (pour décimal) ou `x` (pour hexadécimal). Parfois, nous voulons spécifier la longueur minimale (en caractères) de la sortie, et nous le faisons par un nombre de tête: par exemple, `fmt.Printf("%100d", 4096)` imprime une chaîne de 100 caractères qui se termine par 4096 et commence par des espaces. Nous pouvons spécifier le zéro comme caractère de remplissage plutôt que l'espace en l'ajoutant comme préfixe (par exemple, `"%0100d"`). En Go, nous pouvons ainsi imprimer les bits individuels dans un mot comme dans l'exemple suivant:

```
package main
```

```
import "fmt"

func main() {
  var x uint64 = 1 << 12
  fmt.Printf("%064b", x)
}
```

En exécutant ce programme, nous obtenons une chaîne binaire représentant `1<<12`:

```
package main

import "fmt"

func main() {
  var x int64 = -2
  fmt.Printf("%064b", uint64(x))
}
```

La convention générale pour l'impression des nombres est que les chiffres les plus significatifs sont imprimés en premier, suivis des chiffres les moins significatifs: par exemple, nous écrivons 1234 quand nous voulons dire `1000 + 200 + 30 + 4`. De même, Go imprime les bits les plus significatifs en premier, et donc le nombre `1<<12` a `64-13=51` zéros de tête suivis d'un `1` avec 12 zéros de queue.

Il peut être intéressant de revoir comment Go représente les nombres entiers négatifs. Prenons l'entier de 64 bits `-2`. En utilisant la notation du complément à deux, le nombre devrait être représenté comme le nombre non signé `(1<<64)-2` qui devrait être un mot fait entièrement de uns, sauf pour l'avant-dernier bit. Nous pouvons utiliser le fait qu'une opération *cast* en Go (par exemple, `uint64(x)`) préserve la représentation binaire:

```
package main

import "fmt"
```

```go
func main() {
  var x int64 = -2
  fmt.Printf("%064b", uint64(x))
}
```

Ce programme imprimera `11...10` comme prévu.

Go possède quelques opérateurs binaires pertinents que nous utilisons souvent pour manipuler les bits:

```
&    bitwise AND
|    bitwise OR
^    bitwise XOR
&^   bitwise AND NOT
```

De plus, le symbole `^` est également utilisé pour inverser tous les bits d'un mot lorsqu'il est utilisé comme une opération unaire: `a ^ b` calcule le XOR bit à bit de `a` et `b` alors que `^a` inverse tous les bits de `a`. Nous pouvons vérifier que nous avons `a|b == (a^b) | (a&b) == (a^b) + (a&b)`.

Il existe d'autres identités utiles. Par exemple, étant donné deux entiers `a` et `b`, nous avons que `a+b = (a^b) + 2*(a&b)`. Dans l'identité, `2*(a&b)` représente les retenues alors que `a^b` représente l'addition sans les retenues. Considérons par exemple `0b1001 + 0b10001`. Nous avons que `0b1 + 0b1 == 0b10` et ce résultat correspond à `2*(a&b)`, alors que `0b1000 + 0b10000 == 0b11000` est capturé par `a^b`. Nous avons que `2*(a|b) = 2*(a&b) + 2*(a^b)`, donc `a+b = (a^b) + 2*(a&b)` devient `a+b = 2*(a|b) - (a^b)`. Ces relations sont valables que l'on considère des entiers non signés ou signés, puisque les opérations (logique bit à bit, addition et soustraction) sont identiques au niveau des bits.

Nous vous invitons à prendre en compte ces identités:

- `((a & b) + (a | b)) == (a + b)`
- `((a & b) + (a ^ b)) == (a | b)`
- `((a | b) - (a ^ b)) == (a & b)`

— `((a + b) - (a | b)) == (a & b)`
— `((a + b) - (a & b)) == (a | b)`
— `a + b == 2(a & b) + (a ^ b)`
— `(a - (a | b)) == ((a & b) - b)`

Mise en place, effacement et inversion des bits

Nous savons comment créer un mot de 64 bits avec un seul bit à 1 (par exemple, `1<<12`). Inversement, nous pouvons aussi créer un mot qui est fait de 1 à l'exception d'un 0 à l'indice de bit 12 en inversant tous les bits: `^uint64(1<<12)`. Avant d'inverser tous les bits d'une expression, il est parfois nécessaire de préciser son type (en prenant `uint64` ou `uint32`) afin que le résultat soit sans ambiguïté.

Nous pouvons ensuite utiliser ces mots pour affecter un mot existant:

1. Si nous voulons mettre le douzième bit du mot `w` à un: `w |= 1<<12`.
2. Si nous voulons effacer (mettre à zéro) le douzième bit du mot `w`: `w &^= 1<<12` (ce qui est équivalent à `w = w & ^uint64(1<<12)`).
3. Si nous voulons juste retourner (envoyer les zéros en uns et les uns en zéros) le douzième bit: `w ^= 1<<12`.

Nous pouvons également affecter une plage de bits. Par exemple, nous savons que le mot `(1<<12)-1` a tous les bits sauf les 11 bits les moins significatifs mis à zéro, et les 11 bits les moins significatifs mis à un. Nous pouvons donc faire les opérations suivantes.

1. Si nous voulons que les 11 bits les moins significatifs du mot `w` soient des uns: `w |= (1<<12)-1`.
2. Si l'on veut effacer (mettre à zéro) les 11 bits les moins significatifs du mot `w`: `w &^= (1<<12)-1`.
3. Si l'on veut inverser les 11 bits les moins significatifs: `w ^= (1<<12)-1`.

L'expression `(1<<12)-1` est générale dans le sens où si on veut sélectionner les 60 bits les moins significatifs, on peut faire `(1<<60)-1`. Elle fonctionne même avec 64 bits: `(1<<64)-1` a tous les bits mis à 1.

Nous pouvons également générer un mot dont la plage de bits est arbitraire: le mot `((1<<13)-1) ^ ((1<<2)-1)` a les bits de l'index 2 à l'index 12 (inclusivement) mis à 1, les autres bits sont mis à 0. Avec une telle construction, nous pouvons mettre à 1, effacer ou inverser une gamme arbitraire de bits dans un mot, efficacement.

Nous pouvons activer tous les bits que nous voulons dans un mot. Mais qu'en est-il de l'interrogation des jeux de bits? Nous pouvons vérifier que le douzième bit est activé dans le mot `u` en vérifiant si `w & (1<<12)` est non nul. En effet, l'expression `w & (1<<12)` a pour valeur `1<<12` si le douzième bit est activé dans `w` et, sinon, elle a pour valeur zéro. Nous pouvons étendre une telle vérification: nous pouvons vérifier si l'un des bits de l'indice 2 à l'indice 12 (inclusivement) est mis à 1 en calculant `w & ((1<<13)-1) ^ ((1<<2)-1)`. Le résultat est zéro si et seulement si aucun bit de la plage spécifiée n'est à un.

Opérations efficaces et sûres sur les entiers

En pensant aux valeurs en termes de leur représentation binaire, nous pouvons écrire un code plus efficace ou, ce qui est équivalent, avoir une meilleure appréciation de ce à quoi pourrait ressembler un code binaire optimisé. Considérons le problème consistant à vérifier si deux nombres ont le même signe: nous voulons savoir s'ils sont tous deux inférieurs à zéro, ou tous deux supérieurs ou égaux à zéro. Une mise en œuvre naïve pourrait ressembler à ceci:

```
func SlowSameSign(x, y int64) bool {
  return ((x < 0) && (y < 0)) || ((x >= 0) && (y >= 0))
}
```

Cependant, réfléchissons à ce qui distingue les entiers négatifs des autres entiers: leur dernier bit est activé. C'est-à-dire que leur bit le plus signifi-

catif en tant que valeur non signée est un. Si nous prenons le ou exclusif (xor) de deux entiers, le résultat aura son dernier bit à zéro si leur signe est le même. Autrement dit, le résultat est positif (ou nul) si et seulement si les signes concordent. On peut donc préférer la fonction suivante pour déterminer si deux entiers ont le même signe:

```
func SameSign(x, y int64) bool {
  return (x ^ y) >= 0
}
```

Supposons que nous voulions vérifier si `x` et `y` diffèrent d'au plus 1. `x` est peut-être plus petit que `y`, mais il pourrait être plus grand.

Considérons le problème du calcul de la moyenne de deux entiers. Nous disposons de la fonction correcte suivante:

```
func Average(x, y uint16) uint16 {
  si y > x {
    return (y-x)/2 + x
  } else {
    return (x-y)/2 + y
  }
}
```

Avec une meilleure connaissance de la représentation des entiers, nous pouvons faire mieux.

Nous avons une autre identité pertinente: `x == 2*(x>>1) + (x&1)`. Cela signifie que `x/2` est compris dans `[(x>>1), (x>>1)+1)`. C'est-à-dire que `x>>1` est le plus grand entier non supérieur à `x/2`. Inversement, on a que `(x+(x&1))>>1` est le plus petit entier non inférieur à `x/2`.

Nous avons que `x+y = (x^y) + 2*(x&y)`. Nous avons donc que `(x+y)>>1 == ((x^y)>>1) + (x&y)` (en ignorant les débordements dans `x+y`). Par conséquent, `((x^y)>>1) + (x&y)` est le plus grand nombre entier non supérieur à `(x+y)/2`. Nous avons aussi que `x+y = 2*(x|y) - (x^y)` ou `x+y + (x^y)&1= 2*(x|y) - (x^y) + (x^y)&1` et donc `(x+y+(x^y)&1)>>1 == (x|y) - ((x^y)>>1)` (en ignorant les déborde-

ments dans `x+y+(x^y)&1`). Il s'ensuit que `(x|y) - ((x^y)>>1)` est le plus petit entier non inférieur à `(x+y)/2`. La différence entre `(x|y) - ((x^y)>>1)` et `((x^y)>>1) + (x&y)` est `(x^y)&1`. Par conséquent, nous avons les deux fonctions rapides suivantes:

```
func FastAverage1(x, y uint16) uint16 {
  return (x|y) - ((x^y)>>1)
}
```

```
func FastAverage2(x, y uint16) uint16 {
  return ((x^y)>>1) + (x&y)
}
```

Bien que nous utilisions le type `uint16`, cela fonctionne quelle que soit la taille de l'entier (`uint8`, `uint16`, `uint32`, `uint64`) et cela s'applique également aux entiers signés (`int8`, `int16`, `int32`, `int64`).

Traitement efficace de l'Unicode

En UTF-16, nous pouvons avoir des paires de substituts: le premier mot de la paire est dans la plage `0xd800` à `0xdbff` alors que le second mot est dans la plage de `0xdc00` à `0xdfff`. Comment pouvons-nous détecter les paires de substituts d'efficacité? Si les valeurs sont stockées en utilisant un type `uint16`, alors il semblerait que nous puissions détecter une valeur faisant partie d'une paire de substituts avec deux comparaisons: `(x>=0xd800) && (x<=0xdfff)`. Cependant, il peut s'avérer plus efficace d'utiliser le fait que les soustractions *enveloppent* naturellement: `0-0xd800==0x2800`. Ainsi, `x-0xd800` sera compris entre 0 et `0xdfff-0xd800` inclusivement chaque fois que nous aurons une valeur qui fait partie d'une paire de substituts. Cependant, toute autre valeur sera plus grande que `0xdfff-0xd800=0x7ff`. Une seule comparaison est donc nécessaire: `(x-0xd800)<=0x7ff`. Une fois que nous avons déterminé que nous avons une valeur qui pourrait correspondre à une paire de substituts, nous pouvons vérifier que la première valeur `x1` est valide (dans l'intervalle `0xd800` à `0xdbff`) avec la condition `(x1-0xd800)<=0x3ff`, et de même pour la seconde valeur

`x2: (x2-0xdc00)<=0x3ff`. Les deux vérifications peuvent être combinées en une seule expression: `((x1-0xd800)|(x2-0xdc00)) <=0x3ff`. Nous pouvons alors reconstruire le point de code avec l'expression `(1<<20) + ((x-0xd800)<<10) + x-0xdc00`. En pratique, vous n'aurez peut-être pas à vous préoccuper d'une telle optimisation puisque votre compilateur pourrait le faire pour vous. Néanmoins, il est important de garder à l'esprit que ce qui pourrait sembler être des comparaisons multiples pourrait en fait être implémenté comme une seule comparaison.

SWAR de base

Les processeurs modernes possèdent des instructions spécialisées capables d'opérer sur plusieurs unités de données avec une seule instruction (appelées SIMD pour Single Instruction Multiple Data). Nous pouvons effectuer plusieurs opérations en utilisant une seule instruction (ou quelques instructions) avec une technique appelée SWAR (*SIMD dans un registre*) (Lamport, 1975 [21]). Typiquement, on nous donne un mot de 64 bits `w` (`uint64`) et nous voulons le traiter comme un vecteur de huit mots de 8 bits (`uint8`).

Étant donné une valeur d'octet (`uint8`), nous pouvons la répliquer sur tous les octets d'un mot avec une seule multiplication: `x * uint64(0x0101010101010101)`. Par exemple, on a `0x12 * uint64(0x0101010101010101) == 0x1212121212121212`.
Cette approche peut être généralisée de différentes manières. Par exemple, nous avons que `0x7 * uint64(0x1101011101110101) == 0x7707077707770707`.

Pour des raisons de commodité, définissons `b80` comme étant le `uint64` égal à `0x8080808080808080` et `b01` comme étant le `uint64` égal à `0x0101010101010101`. Nous pouvons vérifier si tous les octets sont inférieurs à 128. On réplique d'abord la valeur de l'octet en mettant à zéro tous les bits sauf le plus significatif (`0x80 * b01` ou `b80`), puis on calcule le ET au sens du bit avec notre mot de 64 bits et on vérifie si le

21. https://doi.org/10.1145/360933.360994

résultat est zéro: `(w & b80)) == 0`. Cela peut se compiler en deux ou trois instructions sur un processeur.

Nous pouvons donc vérifier si une séquence d'octets est composée de caractères ASCII en chargeant des mots de 8 octets. La fonction IsAscii du programme suivant prend en argument une tranche d'octets et renvoie une valeur booléenne indiquant si la tranche d'octets contient uniquement des caractères ASCII. Elle déclare une variable x de type uint64 et l'initialise à zéro. Cette variable stockera le OU bit à bit de tous les octets de la tranche. La variable est indexée pour itérer sur la tranche d'octets. Elle utilise une boucle for pour traiter la tranche d'octets par morceaux de 8 octets à la fois. Pour chaque morceau, il appelle la fonction binary.LittleEndian.Uint64 pour convertir les 8 octets en une valeur uint64, puis effectue une opération OU bit à bit avec x. De cette façon, x aura un bit à 1 si l'un des octets de la tranche a ce bit à 1. Il utilise une autre boucle for pour traiter les octets restants de la tranche, le cas échéant. Pour chaque octet, il le convertit en une valeur uint64 et effectue une opération OU bit à bit avec x. Il effectue une opération ET bit à bit entre x et la constante 0x8080808080808080. Le résultat de cette opération sera zéro si et seulement si aucun des octets de la tranche n'a le bit le plus significatif à 1, ce qui signifie qu'il s'agit de caractères ASCII. Il renvoie le résultat de la comparaison de l'opération bitwise AND avec zéro.

```
package main

import (
  "encoding/binary"
  "fmt"
)

func IsAscii(b []byte) bool {
  var x uint64
  i := 0
  for ; i+8 < len(b); i += 8 {
    x |= binary.LittleEndian.Uint64(b[i : i+8])
```

```
  }
  for ; i < len(b); i++ {
    x |= uint64(b[i])
  }
  x &= 0x8080808080808080
  return x == 0
}

func main() {
  var str string
  str = " My name is Bond,   James Bond."
  fmt.Println(IsAscii([]byte(str)))
  str = " My name is Bondé,  James Bond."
  fmt.Println(IsAscii([]byte(str)))
}
```

Nous pouvons vérifier si un octet est égal à zéro, en supposant que nous avons vérifié qu'il est inférieur à 128, avec une expression telle que `((w - b01) & b80) == 0`: si cette expression est fausse, il se trouve au moins un octet nul. Si nous ne sommes pas sûrs qu'ils sont plus petits que 128, nous pouvons simplement ajouter une opération: `(((w - b01)|w) & b80) == 0`. Vérifier qu'un octet est nul nous permet de vérifier si deux mots, `w1` et `w2`, ont une valeur d'octet correspondante puisque, lorsque cela se produit, `w1^w2` a une valeur d'octet nulle.

Nous pouvons également concevoir des opérations plus complexes si nous supposons que toutes les valeurs d'octets ne sont pas supérieures à 128. Par exemple, nous pouvons vérifier que toutes les valeurs d'octets ne sont pas plus grandes qu'une valeur de 7 bits (`t`) par l'expression suivante: `((w + (0x80 - t) * b01) & b80) == 0`. Si la valeur `t` est une constante, alors la multiplication sera évaluée au moment de la compilation et devrait être à peine plus coûteuse que de vérifier si tous les octets sont plus petits que 128. En Go, nous vérifions qu'aucune valeur d'octet n'est supérieure à 77, en supposant que toutes les valeurs d'octet sont inférieures à 128, en vérifiant que `b80 & (w+(128-77) * b01)` est

zéro. De même, nous pouvons vérifier que toutes les valeurs d'octets sont au moins aussi grandes que le `t` de 7 bits, en supposant qu'elles sont également toutes plus petites que 128: `((b80 - w) + t * b01) & b80) == 0`. Nous pouvons généraliser davantage. Supposons que nous voulions vérifier que tous les octets sont au moins aussi grands que la valeur 7 bits `a` et pas plus grands que la valeur 7 bits `b`. Il suffit de vérifier que `((w + b80 - a * b01) ^ (w + b80 - b * b01)) & b80 == 0`.

Rotation et inversion de l'ordre des bits

On dit que l'on fait une rotation des bits si l'on décale les bits vers la gauche ou la droite, tout en replaçant les bits restants au début. Pour illustrer le concept, supposons que l'on nous donne l'entier de 8 bits `0b1111000` et que nous voulons le faire tourner de 3 bits vers la gauche. Le langage Go fournit une fonction à cet effet (`bits.RotateLeft8` du paquet `math/bits`): nous obtenons `0b10000111`. En Go, il n'y a pas d'opération *rotation vers la droite*. Cependant, tourner à gauche de 3 bits est la même chose que tourner à droite de 5 bits quand on traite des entiers de 8 bits. Go fournit des fonctions de rotation pour les entiers de 8 bits, 16 bits, 32 bits et 64 bits.

Supposons que vous souhaitiez savoir si deux mots de 64 bits (`w1` et `w2`) ont des valeurs d'octets identiques, indépendamment de leur ordre. Nous savons comment vérifier efficacement qu'ils ont des valeurs d'octets ordonnées correspondantes (par exemple, `(((w1^w2 - b01)|(w1^w2)) & b80) == 0`). Pour comparer tous les octets avec tous les autres octets, nous pouvons répéter la même opération autant de fois qu'il y a d'octets dans un mot (huit fois pour les entiers de 64 bits), à chaque fois, nous faisons tourner un des mots de 8 bits:

```
(((w1^w2 - b01)|(w1^w2)) & b80) == 0
w1 = bits.RotateLeft64(w1,8)
(((w1^w2 - b01)|(w1^w2)) & b80) == 0
w1 = bits.RotateLeft64(w1,8)
...
```

Nous rappelons que les mots peuvent être interprétés comme little-endian ou big-endian selon que les premiers octets sont les moins significatifs ou les plus significatifs. Go vous permet d'inverser l'ordre des octets dans un mot de 64 bits avec la fonction `bits.ReverseBytes64` du paquet `math/bits`. Il existe des fonctions similaires pour les mots de 16 et 32 bits. Nous avons que `bits.ReverseBytes16(0xcc00) == 0x00cc`. Inverser les octets d'un mot de 16 bits, et faire une rotation de 8 bits, sont des opérations équivalentes.

On peut aussi inverser l'ordre des bits. Nous avons que `bits.Reverse16(0b1111001101010101) == 0b10101010110011`.
Go possède des fonctions pour inverser les bits pour les mots de 8, 16, 32 et 64 bits. De nombreux processeurs ont des instructions rapides pour inverser l'ordre des bits, et cela peut être une opération rapide.

Comptage rapide des bits

Il peut être utile de compter le nombre de bits à 1 dans un mot. Cette opération est parfois appelée *comptage de population*. Go possède des fonctions rapides à cet effet dans le paquet `math/bits` pour des mots de 8 bits, 16 bits, 32 bits et 64 bits. Ainsi, nous avons que `bits.OnesCount16(0b1111001101010101) == 10`.

De même, nous voulons parfois compter le nombre de zéros de queue ou de tête. Le nombre de zéros de queue est le nombre de bits zéro consécutifs apparaissant dans les positions les moins significatives. Par exemple, le mot "0b1" n'a pas de zéro de queue, alors que le mot "0b100" a deux zéros de queue. Lorsque l'entrée est une puissance de deux, le nombre de zéros de queue est le logarithme en base deux. Nous pouvons utiliser les fonctions Go `bits.TrailingZeros8`, `bits.TrailingZeros16` et ainsi de suite pour calculer le nombre de zéros de fin. Le nombre de zéros de tête est similaire, mais nous commençons par les positions les plus significatives. Ainsi, le nombre entier de 8 bits `0b10000000` n'a pas de zéros de tête, alors que le nombre entier `0b00100000` a deux zéros de tête. Nous pouvons utiliser les fonctions `bits.LeadingZeros8`,

`bits.LeadingZeros16` et ainsi de suite.

Étant donné seulement le nombre d'une fonction de comptage de population, nous pouvons compter le nombre de zéros de queue efficacement en observant que `^x & (x-1)` est le mot où tous les zéros de queue deviennent des uns, et les autres bits sont mis à zéro. Par conséquent, un comptage de population sur `^x & (x-1)` est équivalent au nombre de zéros de queue.

Alors que le nombre de zéros de queue donne directement le logarithme des puissances de deux, on peut utiliser le nombre de zéros de tête pour calculer le logarithme de n'importe quel entier, arrondi à l'entier supérieur. Pour les entiers de 32 bits, la fonction suivante fournit le résultat correct:

```
func Log2Up(x uint32) int {
  return 31 - bits.LeadingZeros32(x|1)
}
```

Nous pouvons également calculer d'autres logarithmes. Intuitivement, cela devrait être possible car si $\log_b$ est le logarithme en base b, alors $\log_b(x) = \log_2(x)/\log_2(b)$. En d'autres termes, tous les logarithmes diffèrent par un facteur constant (par exemple, $1/\log_2(b)$).

Par exemple, nous pouvons nous intéresser au nombre de chiffres décimaux nécessaires pour représenter un nombre entier (par exemple, le nombre entier " 100 " nécessite trois chiffres). La formule générale est `ceil(log(x+1))` où le logarithme doit être pris en base 10. Nous pouvons montrer que la fonction suivante (conçue par un ingénieur appelé Kendall Willets) calcule le nombre de chiffres souhaité pour les entiers de 32 bits:

```
func DigitCount(x uint32) uint32 {
  var table = []uint64{
    4294967296, 8589934582, 8589934582,
    8589934582, 12884901788, 12884901788,
    12884901788, 17179868184, 17179868184,
    17179868184, 21474826480, 21474826480,
    21474826480, 21474826480, 25769703776,
```

```
    25769703776, 25769703776, 30063771072,
    30063771072, 30063771072, 34349738368,
    34349738368, 34349738368, 34349738368,
    38554705664, 38554705664, 38554705664,
    41949672960, 41949672960, 41949672960,
    42949672960, 42949672960}
  return uint32((uint64(x) + table[Log2Up(x)]) >> 32)
}
```

Bien que la fonction soit un peu mystérieuse, son calcul consiste principalement à calculer le nombre de zéros de fin et à utiliser le résultat pour rechercher une valeur dans une table. Elle se traduit par quelques instructions CPU seulement et est efficace.

Indexation des bits

Étant donné un mot, il est parfois utile de calculer la position des bits ayant la valeur 1. Par exemple, étant donné le mot `0b11000111`, nous voudrions avoir les index 0, 1, 2, 6, 7 correspondant aux 5 bits de valeur 1. Nous pouvons déterminer efficacement combien d'index nous devons produire grâce aux fonctions `bits.OnesCount`. Les fonctions `bits.TrailingZeros` peuvent servir à identifier la position d'un bit. Nous pouvons aussi utiliser le fait que `x & (x-1)` met à zéro le bit de poids faible de `x`. La fonction Go suivante génère un tableau d'index:

```
func Indexes(x uint64) []int {
  var ind = make([]int, bits.OnesCount64(x))
  pos := 0
  for x != 0 {
    ind[pos] = bits.TrailingZeros64(x)
    x &= x - 1
    pos += 1
  }
  return ind
```

```
}
```

Étant donné `0b11000111`, il produit le tableau `0, 1, 2, 6, 7`:

```
var x = uint64(0b11000111)
for _, v := range Indexes(x) {
  fmt.Println(v)
}
```

Si nous voulons calculer les bits dans l'ordre inverse (`7, 6, 2, 1, 0`), nous pouvons le faire avec une fonction d'inversion de bits, comme suit:

```
for _, v := range Indexes(bits.Reverse64(x)) {
  fmt.Println(63 - v)
}
```

Conclusion

En tant que programmeur, vous pouvez accéder, définir, copier ou déplacer des valeurs binaires individuelles de manière efficace. Avec un peu d'attention, vous pouvez éviter les débordements arithmétiques sans que les performances en pâtissent beaucoup. Avec SWAR, vous pouvez utiliser un seul mot comme s'il était composé de plusieurs sous-mots. Bien que la plupart de ces opérations ne soient que rarement nécessaires, il est important de savoir qu'elles sont disponibles.

Exercices du chapitre 4

Question 1

Ecrivez une expression Go efficace qui inverse les bits de l'index 3 à l'index 15 inclusivement dans un mot de 64 bits. C'est-à-dire qu'elle doit laisser les 3 bits les moins significatifs inchangés, retourner les 13 bits suivants et laisser les 48 bits les plus significatifs inchangés.

Question 2

Ecrivez une fonction Go efficace qui calcule la moyenne de deux entiers, en arrondissant le résultat à l'entier supérieur. Indice: dans le cas d'un entier impair `x` et d'un entier pair `y`, une solution est `x/2 + y/2 + 1`.

Question 3

Étant donné un tableau d'entiers de 16 bits (`uint16`), vérifiez qu'il s'agit d'une séquence UTF-16 valide.

Question 4

Ecrivez une fonction qui vérifie si deux entiers de 64 bits ont une valeur d'octet correspondante, sans supposer que les valeurs d'octet sont plus petites que 128. Votre solution ne devrait pas nécessiter plus d'une poignée d'opérations et aucun branchement (if ou for) ne devrait être nécessaire.

Question 5

Écrivez une procédure SWAR qui additionne les octets individuels de deux mots de 64 bits. Utilisez l'arithmétique sans débordement ($255 + 255 = 254$).

Chapitre 5

Les logiciels informatiques sont typiquement déterministes sur le papier: si vous exécutez deux fois le même programme avec les mêmes entrées, vous devriez obtenir les mêmes sorties. En pratique, la complexité de l'informatique moderne fait qu'il est peu probable que vous puissiez exécuter deux fois le même programme et obtenir exactement le même résultat, jusqu'au même temps d'exécution. Par exemple, les systèmes d'exploitation modernes randomisent les adresses de la mémoire par mesure de sécurité: une technique appelée *randomisation de la disposition de l'espace d'adressage*. Ainsi, si vous exécutez un programme deux fois, vous ne pouvez pas être sûr que la mémoire est stockée aux mêmes adresses. En Go, vous pouvez imprimer l'adresse d'un pointeur avec la directive `%p`. Le programme suivant allouera un petit tableau d'entiers, et imprimera l'adresse correspondante, en utilisant un pointeur sur la première valeur. Si vous exécutez ce programme plusieurs fois, vous pouvez obtenir des adresses différentes.

```
package main

import (
    "fmt"
)

func main() {
  x := make([]int, 3)
    fmt.Printf("Hello %p", &x[0])
```

```
}
```

Ainsi, dans un certain sens, les programmes logiciels sont déjà *aléatoires*, que cela nous plaise ou non. La randomisation peut rendre la programmation plus difficile. Par exemple, un mauvais programme peut se comporter correctement la plupart du temps et n'échouer que par intermittence. Un tel comportement imprévisible est un défi pour un programmeur.

Néanmoins, nous pouvons utiliser l'aléatoire pour produire de meilleurs logiciels: par exemple en testant notre code avec des entrées aléatoires. En outre, l'aléatoire est un élément clé des routines de sécurité.

Bien que le caractère aléatoire soit une notion intuitive, sa définition nécessite plus de soin. Le caractère aléatoire est généralement lié à un manque d'informations. Par exemple, il peut être mesuré par notre incapacité à prédire un résultat. Peut-être générez-vous des nombres, un par seconde, et après avoir regardé les derniers nombres que vous avez générés, je ne peux toujours pas prédire le prochain nombre que vous allez générer. Cela ne signifie pas que l'approche que vous utilisez pour générer des nombres est magique. Peut-être appliquez-vous une routine mathématique parfaitement prévisible. Le caractère aléatoire est donc relatif à l'observateur et à ses connaissances.

Dans les logiciels, nous faisons la distinction entre le pseudo-aléatoire et l'aléatoire. Si j'exécute une routine mathématique qui génère des nombres d'apparence aléatoire, mais que ces nombres sont parfaitement déterminés, je dirai qu'ils sont *pseudo-aléatoires*. La notion d'apparence aléatoire est subjective et le concept de pseudo-aléatoire l'est également.

Il est possible, sur un ordinateur, de produire des nombres qui ne peuvent être prédits par le programmeur. Par exemple, vous pouvez utiliser un capteur de température dans votre processeur pour capturer le *bruit* physique qui peut servir d'entrée aléatoire. Vous pouvez utiliser l'heure à laquelle un programme a été lancé comme entrée *aléatoire*. Ces valeurs sont souvent qualifiées d'aléatoires (par opposition à pseudo-aléatoires). Nous les considérons comme aléatoires dans le sens où, même en principe,

il n'est pas possible pour le logiciel de les prédire: elles sont produites par un processus extérieur au système logiciel.

Hachage

Le hachage est le processus par lequel nous concevons une fonction qui prend diverses entrées (par exemple des chaînes de longueur variable) et produit une valeur pratique, souvent une valeur entière. Étant donné que le hachage implique une fonction, nous obtenons toujours la même sortie à partir d'une même entrée. En règle générale, les fonctions de hachage produisent un nombre fixe de bits: 32 bits, 64 bits, etc.

L'une des applications du hachage est liée à la sécurité: lorsqu'un fichier est récupéré sur le réseau, il est possible de calculer une valeur de hachage à partir de ce fichier. Vous pouvez ensuite la comparer à la valeur de hachage fournie par le serveur. Si les deux valeurs de hachage correspondent, il est probable que le fichier que vous avez récupéré corresponde au fichier du serveur. Des systèmes tels que git reposent sur cette stratégie.

Le hachage peut également être utilisé pour construire des structures de données utiles. Par exemple, vous pouvez créer une table de hachage: étant donné un ensemble de valeurs clés, vous calculez une valeur de hachage représentant un index dans un tableau. Vous pouvez ensuite stocker la clé et la valeur correspondante à l'index donné, ou à proximité. Lorsqu'une clé est fournie, vous pouvez la hacher, rechercher l'adresse dans le tableau et trouver la valeur correspondante. Si le hachage est aléatoire, vous devriez pouvoir hacher `N` objets dans un tableau de `M` éléments pour `M` légèrement supérieur à `N`, de sorte que quelques objets soient hachés au même endroit dans le tableau. Il est difficile de s'assurer que deux objets ne sont jamais en collision: il faut que `M` soit beaucoup, beaucoup plus grand que `N`. Pour `M` beaucoup plus grand que `N`, la probabilité d'une collision est d'environ `1 - exp(-N*N/(2*M))`. Bien que cette probabilité tombe à zéro lorsque M devient grand, il faut que `M` soit beaucoup plus grand que `N` pour qu'elle soit pratiquement nulle. En résolvant pour `p` dans `1 - exp(-N*N/(2*M)) = p`, nous obtenons `M = -1/2 N*N / ln(1-p)`.

En d'autres termes, pour maintenir une probabilité `p`, `M` doit croître quadratiquement (proportionnellement à `N*N`) par rapport à `N`. Il faut donc s'attendre à des collisions dans une table de hachage, même si la fonction de hachage semble aléatoire. Nous pouvons gérer les collisions de différentes manières. Par exemple, vous pouvez utiliser le chaînage: chaque élément du tableau stocke une référence à un conteneur dynamique qui peut contenir plusieurs clés.

Vous pouvez également utiliser la recherche linéaire. Lorsqu'une collision survient, c'est-à-dire lorsque deux clés produisent le même indice après application de la fonction de hachage, la recherche linéaire consiste à examiner les emplacements suivants dans le tableau jusqu'à trouver une position libre pour stocker la clé ou jusqu'à localiser la clé recherchée. Supposons une table de hachage de taille 10, avec une fonction de hachage simple : `h(clé) = clé % 10`. Il existe plusieurs variations de la recherche linéaire pour optimiser les performances ou gérer les collisions différemment. Au lieu d'avancer d'un pas à la fois (comme dans la recherche linéaire), la recherche quadratique utilise un saut qui augmente de manière quadratique. Par exemple, si une collision survient à l'indice `h(clé)`, on essaie les emplacements `h(clé) + 1²`, `h(clé) + 2²`, `h(clé) + 3²`, etc. La recherche quadratique peut aider à répartir les clés de manière plus uniforme. Nous pouvons aussi utiliser une une deuxième fonction de hachage pour déterminer le pas de saut en cas de collision. Si `h1(clé)` donne l'indice initial et qu'il est occupé, on calcule un pas avec une deuxième fonction `h2(clé)`, et on essaie les emplacements `h1(clé) + h2(clé)`, `h1(clé) + 2 * h2(clé)`, etc.

Une fonction de hachage pourrait prendre chaque entrée possible et l'affecter à une valeur purement aléatoire donnée par un Oracle. Malheureusement, de telles fonctions de hachage sont souvent peu pratiques. Elles nécessitent le stockage de grandes tables de valeurs d'entrée et de valeurs aléatoires correspondantes. En pratique, nous cherchons à produire des fonctions de hachage qui se comportent comme si elles étaient purement aléatoires tout en étant faciles à mettre en œuvre de manière efficace.

La fonction murmur est un exemple raisonnable de hachage de valeurs

entières non nulles. La fonction murmur consiste en deux multiplications et trois opérations shift/xor. Le programme Go suivant affichera des entiers 64 bits d'apparence aléatoire, en utilisant la fonction murmur:

```
package main

import (
    "fmt"
    "math/bits"
)

func murmur64(h uint64) uint64 {
    h ^= h >> 33
    h *= 0xff51afd7ed558ccd
    h ^= h >> 33
    h *= 0xc4ceb9fe1a85ec53
    h ^= h >> 33
    return h
}

func main() {

    for i := 0; i < 10; i++ {
      fmt.Println(i, murmur64(uint64(i)))
    }
}
```

C'est une fonction raisonnablement rapide. L'inconvénient de la fonction `murmur64` est que zéro est mis en correspondance avec zéro, il faut donc faire attention.

En pratique, vos valeurs peuvent ne pas être des entiers. Si vous voulez hacher une chaîne de caractères, vous pouvez utiliser une *fonction récursive*. Vous traitez la chaîne caractère par caractère. À chaque caractère, vous combinez la valeur du caractère avec la valeur de hachage calculée jusqu'à présent, générant ainsi une nouvelle valeur de hachage. Une fois

la fonction terminée, vous pouvez appliquer murmur au résultat:

```
package main

import (
    "fmt"
)

func murmur64(h uint64) uint64 {
    h ^= h >> 33
    h *= 0xff51afd7ed558ccd
    h ^= h >> 33
    h *= 0xc4ceb9fe1a85ec53
    h ^= h >> 33
    return h
}

func hash(s string) (v uint64) {
    v = uint64(0)
    for _, c := range s {
      v = uint64(c) + 31*v
    }
    return murmur64(v)
}

func main() {
    fmt.Print(hash("la vie"), hash("Daniel"))
}
```

Il existe des fonctions de hachage plus performantes et plus rapides, mais le résultat d'un hachage récursif avec un finalisateur murmur est raisonnable.

Il est important de noter qu'il est relativement facile de générer deux chaînes de caractères qui correspondent aux mêmes valeurs, c'est-à-dire de créer une collision. Par exemple, vous pouvez vérifier que les chaînes

`"Ace"`, `"BDe"`, `"AdF"`, `"BEF"` ont toutes la même valeur de hachage:

```
fmt.Print(hash("Ace"), hash("BDe"),
 hash("AdF"), hash("BEF"))
```

Lors du hachage de chaînes de caractères arbitrairement longues, des collisions sont toujours possibles. Toutefois, nous pouvons utiliser des fonctions de hachage plus sophistiquées (et plus coûteuses en termes de calcul) pour réduire la probabilité de rencontrer un problème.

Étant donné une longue chaîne de caractères, vous pouvez vouloir hacher toutes les séquences de N caractères. Une approche naïve pourrait être la suivante:

```
for(size_t i = 0; i < len-N; i++) {
  uint32_t hash = 0;
  for(size_t j = 0; j < N; j++) {
    hash = hash * B + data[i+j];
  }
  //...
}
```

Vous accédez à la plupart des valeurs des caractères N fois. Si N est grand, cela est inefficace.

Vous pouvez faire mieux en utilisant une fonction de hachage roulant: au lieu de recalculer la fonction de hachage à chaque fois, vous la mettez simplement à jour. Il est possible de n'accéder à chaque caractère que deux fois (au lieu de N fois).

```
func rollinghash(s string, N int) {
    rev := uint64(1)
    for i := 0; i < N; i++ {
        rev *= 31
    }
    v := uint64(0)
    for i := 0; i < N; i++ {
        v = uint64(s[i]) + 31*v
```

```
    }
    fmt.Println(v)
    for i := N; i < len(s); i++ {
        v = uint64(s[i]) + 31*v - rev*uint64(s[i-N])
        fmt.Println(v)
    }
}
```

Une caractéristique intéressante de la fonction `murmur64` est qu'elle est inversible. Si vous considérez les étapes, vous avez deux multiplications par des entiers impairs. Une multiplication par un entier impair est toujours inversible: l'inverse multiplicatif de 0xff51afd7ed558ccd est 0x4f74430c22a54005 et l'inverse multiplicatif de 0xc4ceb9fe185ec53 est 0x9cb4b2f8129337db, en tant qu'entiers non signés de 64 bits. Il peut être un peu moins évident que `h ^= h >> 33` soit inversible. Mais si `h` est un entier de 64 bits, nous avons que `h` et `h ^ (h >> 33)` sont identiques dans leurs 33 bits les plus significatifs, par inspection. Ainsi, si l'on nous donne `z = h ^ (h >> 33)`, nous avons `z >> (64-33) == h >> (64-33)`. En d'autres termes, nous avons identifié les 33 bits les plus significatifs de `h` à partir de `h ^ (h >> 33)`. En étendant ce raisonnement, nous obtenons que `g` est l'inverse de `f` dans le code suivant, dans le sens où `g(f(i)) == i`.

```
func f(h uint64) uint64 {
    return h ^ (h >> 33)
}

func g(z uint64) uint64 {
    h := z & 0xffffffff80000000
    h = (h >> 33) ^ z
    return h
}
```

Nous avons souvent besoin que les valeurs de hachage s'inscrivent dans un intervalle commençant à zéro. Par exemple, si vous souhaitez obtenir

une valeur de hachage dans `[0,max)`, vous pouvez utiliser la fonction suivante:

```
func toIntervalBias(random uint64, max uint64) uint64 {
  hi,_ := bits.Mul64(random, max)
  return hi
}
```

Cette fonction produit une valeur dans `[0,max)` en utilisant une seule multiplication. Il existe des alternatives telles que `random % max`, mais une opération de reste d'entier peut se compiler en une instruction de division, et une division est typiquement plus coûteuse qu'une multiplication. Dans la mesure du possible, il est préférable d'éviter les instructions de division lorsque les performances sont en jeu.

Il est important de noter que la fonction `toIntervalBias` introduit un léger biais: nous commençons avec 2^{64} valeurs distinctes et nous les convertissons en `N` valeurs distinctes. Cela signifie que sur 2^{64} valeurs originales, environ $2^{64}/N$ valeurs correspondent à chaque valeur de sortie. Soit $\lceil x \rceil$ le plus petit entier non inférieur à x et $\lfloor x \rfloor$ le plus grand entier non supérieur à x. Lorsque $2^{64}/N$ n'est pas un entier, certaines valeurs de sortie correspondent aux valeurs originales de $\lceil 2^{64}/N \rceil$, tandis que d'autres correspondent aux valeurs originales de $\lfloor 2^{64}/N \rfloor$. Lorsque N est petit, il peut être négligeable, mais lorsque N augmente, le biais est relativement plus important. Dans un certain sens, il s'agit du plus petit biais possible si l'on part de valeurs originales uniformément distribuées sur un ensemble de 2^{64} valeurs possibles.

Pour résumer, le programme suivant convertit une chaîne de caractères en une valeur comprise dans l'intervalle `[0,10]`.

```
package main

import (
    "fmt"
    "math/bits"
)
```

```
func murmur64(h uint64) uint64 {
    h ^= h >> 33
    h *= 0xff51afd7ed558ccd
    h ^= h >> 33
    h *= 0xc4ceb9fe1a85ec53
    h ^= h >> 33
    return h
}

func hash(s string) (v uint64) {
    v = uint64(0)
    for _, c := range s {
      v = uint64(c) + 31*v
    }
    return murmur64(v)
}

func toIntervalBias(random uint64, max uint64) uint64 {
  hi,_ := bits.Mul64(random, max)
  return hi
}

func main() {
    fmt.Print(toIntervalBias(hash("la vie"),10))
}
```

Bien que la fonction `toIntervalBias` soit généralement efficace, elle est inutilement coûteuse lorsque l'intervalle est une puissance de deux. Si `max` est une puissance de deux (par exemple, 32), alors `random % max = random & (max-1)`. Un ET bit à bit avec le maximum décrémenté est plus rapide qu'une simple multiplication, typiquement. La fonction suivante est donc préférable.

```
func toIntervalPowerOfTwo(random uint64,
  max uint64) uint64 {
  return random & (max-1)
}
```

Estimation de la cardinalité

L'un des cas d'utilisation du hachage est l'estimation de la cardinalité des valeurs d'un tableau ou d'un flux de valeurs. Supposons que votre logiciel reçoive des milliards d'identifiants. Combien y a-t-il d'identifiants distincts? Vous pourriez créer une base de données de tous les identificateurs, mais cela utiliserait beaucoup de mémoire et serait relativement coûteux. Parfois, vous ne souhaitez qu'une approximation grossière, mais vous voulez la calculer rapidement.

Il existe de nombreuses techniques pour estimer les cardinalités à l'aide du hachage: comptage probabiliste (Flajolet-Martin), comptage probabiliste LOGLOG, etc. Nous pouvons expliquer l'idée principale et même produire une fonction utile sans avoir recours à des mathématiques avancées.

Supposons qu'il y a m valeurs distinctes. Si vous appliquez une fonction de hachage sur ces m valeurs distinctes, et que les valeurs valeurs de hachages sont dans l'interval des entiers $[0, n)$, c'est comme si vous aviez choisi m valeurs au hasard dans l'interval $[0, n)$. Si vous choisissez au hasard m emplacements dans un tableau de taille n, la densité attendue (la fraction des éléments du tableau sélectionnés) est $(1 - (1 - 1/n)^m)$. Si nous mesurons cette densité D, nous pouvons calculer m à partir de $(1 - (1 - 1/n)^m) = D$: $m = \log(1 - D)/\log(1 - 1/n)$. Si n est choisi suffisamment grand, on devrait avoir que $\log(1 - D)/\log(1 - 1/n)$ est une bonne estimation pour le nombre de valeurs distinctes. La fonction suivante applique cette formule pour estimer la cardinalité:

```
func estimateCardinality(values []uint64) int {
    words := 32768
    volume := words * 64
```

```
    b := make([]uint64, words) // 256 kB
    for i := 0; i < len(values); i++ {
      num := murmur64(values[i])
      num = num & uint64(volume-1)
      b[num/64] |= 1 << (num % 64)
    }
    x := 0
    for i := 0; i < words; i++ {
      x += bits.OnesCount64(b[i])
    }
    if x == volume {
      return -1
    }
    m := math.Log(1-float64(x)/float64(volume))
    / math.Log((float64(volume)-1)/float64(volume))
    return int(m)
}
```

Nous pouvons appliquer notre fonction dans le programme suivant. L'approximation est assez grossière, mais elle peut être suffisante dans certains cas pratiques tant que le nombre d'éléments distincts est inférieur à 20 millions. Nous utilisons environ 0,1 bit par valeur distincte. Si vous prévoyez des ensembles plus importants, vous devez utiliser des techniques plus avancées ou augmenter la mémoire allouée à la fonction `estimateCardinality`. Elle est capable d'estimer le nombre d'éléments distincts (19,5 millions) dans un ensemble d'un milliard d'éléments en une ou deux secondes.

```
package main

import (
    "fmt"
    "math"
    "math/bits"
)
```

```
func murmur64(h uint64) uint64 {
    h ^= h >> 33
    h *= 0xff51afd7ed558ccd
    h ^= h >> 33
    h *= 0xc4ceb9fe1a85ec53
    h ^= h >> 33
    return h
}

func fillArray(arr []uint64, howmany int) {
    for i := 0; i < len(arr); i++ {
      arr[i] = 1 + uint64(i%howmany)
    }
}

// returns -1 if the cardinality is too high
func estimateCardinality(values []uint64) int {
    words := 32768
    volume := words * 64
    b := make([]uint64, words) // 256 kB
    for i := 0; i < len(values); i++ {
      num := murmur64(values[i])
      num = num & uint64(volume-1)
      b[num/64] |= 1 << (num % 64)
    }
    x := 0
    for i := 0; i < words; i++ {
      x += bits.OnesCount64(b[i])
    }
    if x == volume {
      return -1
    }
    m := math.Log(1-float64(x)/float64(volume))
    / math.Log((float64(volume)-1)/float64(volume))
```

```
    return int(m)
}

func main() {
    values := make([]uint64, 1_000_000_000) // 1 B
    distinct := 19_500_000          // 19.5 M
    fillArray(values, distinct)
    c := estimateCardinality(values)
    fmt.Println("estimated: ", c,
     "actual: ", distinct, " margin : ",
    float64(c)/float64(distinct))
}
```

L'algorithme Flajolet-Martin (Flajolet and Martin, 1985 [22]) est une méthode probabiliste pour estimer le nombre d'éléments distincts dans un flux de données. Il utilise une fonction de hachage pour transformer chaque élément en une valeur binaire, puis observe la position du bit 1 le plus à droite (r). L'estimation est basée sur la valeur maximale de r; le nombre d'éléments distincts est approximativement $2^{\max(r)}$. Plus précisément, l'estimation de la cardinalité est donnée par $2^{\max(r)}/\phi$, où $\phi \approx 0.77351$ est une constante de correction pour améliorer la précision. Cet algorithme est particulièrement adapté aux flux de données massifs, car il ne nécessite qu'une petite quantité de mémoire pour stocker la valeur maximale de r. Le code suivant illustre l'algorithme de Flajolet-Martin.

```
package main

import (
  "fmt"
  "math"
  "math/bits"
)
```

22. https://doi.org/10.1016/0022-0000(85)90041-8

```
func murmur64(h uint64) uint64 {
  h ^= h >> 33
  h *= 0xff51afd7ed558ccd
  h ^= h >> 33
  h *= 0xc4ceb9fe1a85ec53
  h ^= h >> 33
  return h
}

func fillArray(arr []uint64, howmany int) {
  for i := 0; i < len(arr); i++ {
    arr[i] = 1 + uint64(i%howmany)
  }
}

func flajoletMartin(values []uint64) int {
  maxR := 0
  for _, val := range values {
    hash := murmur64(val)
    r := bits.LeadingZeros64(hash)
    if r > maxR {
      maxR = r
    }
  }
  phi := 0.77351
  return int(math.Pow(2, float64(maxR)) / phi)
}

func main() {
  values := make([]uint64, 1_000_000_000)
  distinct := 19_500_000
  fillArray(values, distinct)
  fm := flajoletMartin(values)
  fmt.Println("estimated: ", fm, "actual: ",
```

```
    distinct, " margin : ", float64(fm)/float64(distinct))

}
```

Entiers

Il existe de nombreuses façons de générer des entiers aléatoires, mais une approche particulièrement simple consiste à s'appuyer sur le hachage. Par exemple, nous pourrions partir d'un entier (par exemple, 10) et renvoyer l'entier aléatoire `murmur64(10)`, puis incrémenter l'entier (par exemple, jusqu'à 11) et renvoyer ensuite l'entier `murmur64(10)`.

Steele et al. (2014) [23] proposent une stratégie similaire qu'ils appellent SplitMix: elle fait partie de la bibliothèque standard Java. Le fonctionnement est similaire à celui que nous venons de décrire, mais au lieu d'incrémenter le compteur de un, ils l'incrémentent d'un grand entier impair. Ils utilisent également une version légèrement différente de la version `murmur64`. La fonction suivante imprime 10 valeurs aléatoires différentes, en suivant la formule SplitMix:

```
package main

import "fmt"

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}
```

23. https://doi.org/10.1145/2714064.2660195

```
func main() {
    seed := uint64(1234)
    for z := 0; z < 10; z++ {
      r := splitmix64(&seed)
      fmt.Println(r)
    }
}
```

Chaque fois que la fonction `splitmix64` est appelée, la variable cachée `seed` est avancée d'une constante (`0x9E3779B97F4A7C15`). Si vous partez de la même semence, vous obtiendrez toujours les mêmes valeurs aléatoires.

La fonction effectue ensuite une série d'opérations bit à bit sur z. Tout d'abord, elle effectue une opération XOR entre z et z décalé vers la droite de 30 bits. Elle multiplie ensuite le résultat par la valeur constante 0xBF58476D1CE4E5B9. Ensuite, il effectue une autre opération XOR entre le résultat et le résultat décalé vers la droite de 27 bits. Enfin, il multiplie le résultat par la valeur constante 0x94D049BB133111EB et renvoie le résultat XORé avec le résultat décalé vers la droite de 31 bits.

Il produit des nombres entiers en utilisant toute la plage de 64 bits. Si l'on a besoin d'un entier aléatoire dans un intervalle (par exemple, `[0,N)`), il faut travailler davantage. Si la taille de l'intervalle est une puissance de deux (par exemple, `[0,32)`), nous pouvons simplement utiliser la même technique que pour le hachage:

```
// randomInPowerOfTwo -> [0,max)
func randomInPowerOfTwo(seed *uint64, max uint64) uint64 {
    r := splitmix64(seed)
    return r & (max-1)
}
```

Cependant, lorsque la borne est arbitraire (pas une puissance de deux) et que nous voulons éviter les biais, un algorithme légèrement plus compliqué est nécessaire. En effet, si nous supposons que les entiers de 64 bits sont

réellement aléatoires, alors toutes les valeurs ont la même probabilité. Cependant, si nous ne faisons pas attention, nous pouvons introduire un biais lors de la conversion des entiers de 64 bits en valeurs dans `[0,N)`. Ce n'est pas un problème lorsque `N` est une puissance de deux, mais cela le devient lorsque `N` est arbitraire. Une routine rapide a été décrite par Lemire (2019) [24] pour résoudre ce problème:

```
func toIntervalUnbiased(seed *uint64, max uint64) uint64 {
    x := splitmix64(seed)
    hi, lo := bits.Mul64(x, max)
    if lo < max {
      t := (-max) % max // division!!!
      for lo < t {
          x := splitmix64(seed)
          hi, lo = bits.Mul64(x, max)
      }
    }
    return hi
}
```

La fonction `toIntervalUnbiased` prend deux arguments: un pointeur sur un entier non signé de 64 bits (`seed`) et un entier non signé de 64 bits (`max`). Elle renvoie un entier non signé de 64 bits. La fonction appelle d'abord la fonction `splitmix64` avec le pointeur `seed` comme argument pour générer un entier aléatoire non signé de 64 bits `x`. Elle multiplie ensuite `x` avec `max` à l'aide de la fonction `bits.Mul64`, qui renvoie le produit de deux entiers non signés de 64 bits sous la forme de deux entiers non signés de 64 bits. Les 64 bits supérieurs du produit sont stockés dans la variable `hi` et les 64 bits inférieurs sont stockés dans la variable `lo`. Si `lo` est inférieur à `max`, la fonction entre dans une boucle qui génère de nouveaux nombres aléatoires à l'aide de `splitmix64` et recalcule le produit de `x` et de `max` jusqu'à ce que `lo` soit supérieur ou égal à `-max % max`. Cet algorithme permet de s'assurer que la distribution des nombres aléatoires n'est pas biaisée.

24. https://arxiv.org/abs/1805.10941

La stratégie générale utilisée par cette fonction est appelée méthode de rejet: nous essayons à plusieurs reprises de générer un entier aléatoire jusqu'à ce que nous puissions produire un résultat non biaisé. Cependant, lorsque l'intervalle est beaucoup plus petit que 2^{64} (un cas courant), il est très peu probable que nous utilisions la méthode de rejet ou que nous devions même calculer un reste entier. La plupart du temps, la fonction n'entre jamais dans la boucle de rejet.

Tester qu'un générateur aléatoire semble aléatoire est un défi. Nous pouvons utiliser de nombreuses stratégies de test, et chacune d'entre elles peut être plus ou moins étendue. Heureusement, il n'est pas difficile de penser à certains tests que nous pouvons appliquer. Par exemple, nous voulons que la distribution des valeurs soit uniforme: la probabilité qu'une valeur soit générée doit être égale à 1 sur le nombre de valeurs possibles. Lorsque l'on génère 2 des 64 valeurs possibles, il est techniquement difficile de tester l'uniformité. Cependant, nous pouvons commodément restreindre la taille de la sortie avec une fonction telle que `toInterval`.

Le programme suivant calcule l'écart-type relatif d'un histogramme de fréquence basé sur 100 millions de valeurs. L'écart-type relatif est bien inférieur à 1 % (0,05655 %), ce qui suggère que la distribution est uniforme.

```
package main

import (
    "fmt"
    "math"
    "math/bits"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
```

```
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

func toIntervalUnbiased(seed *uint64, max uint64) uint64 {
    x := splitmix64(seed)
    hi, lo := bits.Mul64(x, max)
    if lo < max {
      t := (-max) % max // division!!!
      for lo < t {
          x := splitmix64(seed)
          hi, lo = bits.Mul64(x, max)
      }
    }
    return hi
}

func meanAndStdDev(arr []int) (float64, float64) {
    var sum, sumSq float64
    for _, val := range arr {
      sum += float64(val)
      sumSq += math.Pow(float64(val), 2)
    }
    n := float64(len(arr))
    mean := sum / n
    stdDev := math.Sqrt((sumSq / n) - math.Pow(mean, 2))
    return mean, stdDev
}

func main() {
    seed := uint64(1234)
    const window = 30
    var counter [window]int
```

```
    for z := 0; z < 100000000; z++ {
      counter[toIntervalUnbiased(&seed, window)] += 1
    }
    moyenne, ecart := meanAndStdDev(counter[:])
    fmt.Println("relative std ", ecart/moyenne*100, "%")
}
```

Mélange aléatoire

Il arrive que l'on vous donne un tableau que vous souhaitez mélanger de manière aléatoire. Un algorithme élégant décrit par Knuth est l'approche standard. L'algorithme fonctionne en itérant sur le tableau du dernier élément au premier élément. À chaque itération, il sélectionne un indice aléatoire entre 0 et l'indice courant (inclus) et échange l'élément à l'indice courant avec l'élément à l'indice généré aléatoirement.

Nous pouvons prouver qu'il fournit un mélange aléatoire équitable par un argument d'induction. Il existe $N!$ permutations possibles d'un tableau de taille N et nous voulons un algorithme qui produise une de ces $N!$ permutations au hasard. Lorsque le tableau est de longueur 2, nous pouvons vérifier qu'il conserve l'ordre par défaut ou qu'il permute les deux valeurs: chaque possibilité a une probabilité de 50%. Supposons que vous souhaitiez mélanger aléatoirement un tableau de taille N, mais que vous sachiez comment mélanger aléatoirement un tableau de taille N-1. Vous commencez par mélanger les premiers ou les derniers N-1 éléments du tableau de taille N. Il y a $(N-1)!$ de telles permutations, et vous supposez qu'elles sont également probables par votre argument d'induction. Ensuite, vous permutez l'unique élément non mélangé avec n'importe quel autre élément choisi (y compris lui-même) au hasard (uniformément). Cela crée $N!$ permutations, toutes également probables. L'algorithme est donc correct. Vous pouvez penser qu'il est facile de concevoir d'autres algorithmes de mélange aléatoire, mais nous n'en connaissons que très peu.

Le programme suivant mélange aléatoirement un tableau en fonction d'une semence. Le changement de la semence modifie l'ordre du tableau. Pour les grands tableaux, le nombre de permutations possibles est susceptible de dépasser le nombre de semences possibles: cela implique que toutes les permutations possibles ne sont pas possibles avec un tel algorithme utilisant une simple semence de longueur fixe.

```
package main

import (
    "fmt"
    "math/bits"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

func toIntervalUnbiased(seed *uint64, max uint64) uint64 {
    x := splitmix64(seed)
    hi, lo := bits.Mul64(x, max)
    if lo < max {
      t := (-max) % max // division!!!
      for lo < t {
          x := splitmix64(seed)
          hi, lo = bits.Mul64(x, max)
      }
    }
    return hi
```

```
}

func shuffle(seed *uint64, arr []int) {
    for i := len(arr)-1; i >= 1; i-- {
      j := toIntervalUnbiased(seed, uint64(i+1))
      arr[i], arr[j] = arr[j], arr[i]
    }
}

func main() {
    seed := uint64(1234)
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    shuffle(&seed, numbers)
    fmt.Println(numbers)
}
```

Échantillonnage de réservoir

Il arrive que l'on veuille choisir au hasard k éléments distincts dans un ensemble de N éléments. Une solution évidente consisterait à mélanger aléatoirement les N éléments et à choisir les k premiers ou les k derniers éléments. Malheureusement, un tel algorithme nécessite beaucoup d'espace de stockage (N éléments). Une approche plus efficace consiste à utiliser une variante du mélange aléatoire standard qui ne conserve que les k premiers éléments en mémoire. Le résultat est appelé échantillonnage par réservoir.

L'algorithme fonctionne également par itération sur le tableau. Comme pour le brassage aléatoire, à chaque itération, nous sélectionnons un indice aléatoire entre 0 et l'indice courant (inclus) et nous échangeons virtuellement l'élément à l'indice courant avec l'élément à l'indice généré aléatoirement. Cependant, nous n'avons besoin de procéder à une permutation physique que si l'index généré aléatoirement se situe dans les k premiers éléments.

Le code ressemble beaucoup au mélange aléatoire. Une caractéristique intéressante de cet algorithme est qu'il nécessite le moins de mémoire possible. Malheureusement, si le tableau d'entrée est grand, ce n'est peut-être pas le meilleur algorithme car il nécessite de parcourir l'ensemble du tableau.

```
package main

import (
    "fmt"
    "math/bits"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

func toIntervalUnbiased(seed *uint64, max uint64) uint64 {
    x := splitmix64(seed)
    hi, lo := bits.Mul64(x, max)
    if lo < max {
      t := (-max) % max // division!!!
      for lo < t {
          x := splitmix64(seed)
          hi, lo = bits.Mul64(x, max)
      }
    }
    return hi
}
```

```
func sample(seed *uint64, arr []int, k int) []int {
    answer := arr[:k]
    for i := k; i < len(arr); i++ {
      j := int(toIntervalUnbiased(seed, uint64(i+1)))
      if j < k {
          answer[j] = arr[i]
      }
    }
    return answer
}

func main() {
    seed := uint64(1234)
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    for t := 0; t < 30; t++ {
      s := sample(&seed, numbers, 3)
      fmt.Println(s)
    }
}
```

Dans le cas où vous souhaitez sélectionner quelques éléments parmi un grand nombre d'éléments, vous pouvez simplement continuer à sélectionner un index au hasard et l'ajouter à un ensemble d'index, jusqu'à ce que vous ayez atteint k index distincts. Il existe plusieurs façons d'implémenter une structure de données d'ensemble, mais la plus simple est un tableau, qui peut bien fonctionner si vous souhaitez sélectionner quelques éléments. Une table de hachage ou une autre structure de données de ce type peut s'avérer plus efficace pour les ensembles plus importants. Il est également possible de résoudre ce problème en utilisant une structure de données de type bitset. Cependant, pour de nombreux cas d'utilisation, le code simple suivant devrait s'avérer utile:

```
package main
```

```
import (
    "fmt"
    "math/bits"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

func toIntervalUnbiased(seed *uint64, max uint64) uint64 {
    x := splitmix64(seed)
    hi, lo := bits.Mul64(x, max)
    if lo < max {
      t := (-max) % max // division!!!
      for lo < t {
          x := splitmix64(seed)
          hi, lo = bits.Mul64(x, max)
      }
    }
    return hi
}

func isInSlice(arr []int, x int) bool {
    for _, y := range arr {
      if x == y {
          return true
      }
    }
```

```
    return false
}

func sample(seed *uint64, arr []int, k int) []int {
    idx := make([]int, 0, k)
    for len(idx) < k {
      j := int(toIntervalUnbiased(seed, uint64(len(arr))))
      if !isInSlice(idx, j) {
          idx = append(idx, j)
      }
    }
    answer := make([]int, k)
    for i, j := range idx {
      answer[i] = arr[j]
    }
    return answer
}

func main() {
    seed := uint64(1234)
    numbers := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    for t := 0; t < 30; t++ {
      s := sample(&seed, numbers, 3)
      fmt.Println(s)
    }
}
```

Nombres à virgule flottante

Il est souvent nécessaire de générer des nombres flottants aléatoires. Les systèmes logiciels utilisent généralement les nombres à virgule flottante IEEE 754.

Pour générer des nombres à virgule flottante de 32 bits dans l'intervalle

$[0, 1)$, il peut sembler que nous pourrions générer un entier de 32 bits (dans $[0, 2^{32})$) et le diviser par 2^{32} pour obtenir une valeur aléatoire à virgule flottante dans $[0, 1)$. C'est certainement *approximativement vrai*, mais nous commettons une erreur en faisant cela. Quelle est l'ampleur de l'erreur?

Les nombres à virgule flottante (normaux*) sont représentés par un bit de signe, une mantisse et un exposant comme suit:

— Il y a un seul bit de signe. Comme nous ne nous intéressons qu'aux nombres positifs, ce bit est fixe et peut être ignoré.
— La mantisse d'un nombre à virgule flottante de 32 bits est de 23 bits. Elle est implicitement précédée du chiffre 1.
— Huit bits sont consacrés à l'exposant. Pour les nombres normaux, l'exposant est compris entre -126 et 127. Pour représenter zéro, il faut un exposant de -127 et une mantisse de zéro.

Combien y a-t-il donc de nombres normaux non nuls entre 0 et 1? Les exposants négatifs vont de -1 à -126. Dans chaque cas, nous avons 2^{23} de nombres flottants distincts car la mantisse est composée de 23 bits. Nous avons donc 126 x 2^{23} nombres flottants normaux dans $[0, 1)$. Si vous n'avez pas de calculatrice à portée de main, cela fait 1 056 964 608. Si l'on veut additionner les nombres 0 et 1, cela fait $126 \times 2^{23} + 2$ un peu plus d'un milliard de nombres distincts. + 2$, soit un peu plus d'un milliard de valeurs distinctes. Il y a 2^{32} mots de 32 bits, soit un peu plus de 4 milliards, ce qui signifie qu'environ un quart d'entre eux se trouve dans l'intervalle `[0,1]`. De tous les nombres à virgule flottante que votre ordinateur peut représenter, un quart se trouve dans l'intervalle `[0,1]`. Par extension, la moitié des nombres à virgule flottante se trouvent dans l'intervalle `[-1,1]`.

Le nombre 2^{32} n'est pas divisible par 126 $fois 2^{23} + 2$. + 2$, donc nous ne pouvons pas prendre un entier non négatif de 32 bits, le diviser par 2^{32} et espérer que cela génère un nombre dans `[0,1]` ou $[0, 1)$ d'une manière impartiale.

Nous pouvons utiliser le fait que la mantisse utilise 23 bits. Cela signifie

en particulier que si vous choisissez n'importe quel entier dans $[0, 2^{24})$, et que vous le divisez par 2^{24}, vous pouvez récupérer votre entier original en multipliant à nouveau le résultat par 2^{24}. Cela fonctionne avec 2^{24} mais pas avec 2^{25} ou tout autre nombre plus grand. Pour les nombres à virgule flottante de 64 bits, la précision est plus grande car vous pouvez remplacer 24 par 53.

Vous pouvez donc choisir un entier aléatoire dans $[0, 2^{24})$, le diviser par 2^{24} et vous obtiendrez un nombre aléatoire dans $[0, 1)$ sans biais, ce qui signifie que pour chaque entier dans `[0,2^{24})`, il y a un et un seul nombre dans $[0, 1)$. De plus, la distribution est uniforme dans le sens où les nombres flottants possibles sont régulièrement espacés (la distance entre eux est égale à 2^{-24}).

Ainsi, même si les nombres flottants à simple précision utilisent des mots de 32 bits et même si votre ordinateur peut représenter environ 230 nombres flottants distincts et normaux dans l'intervalle $[0, 1)$, il y a de bonnes chances que votre générateur aléatoire ne produise que 2^{24} de nombres flottants 32 bits distincts dans l'intervalle $[0, 1)$, et seulement 2^{53} de nombres flottants 64 bits distincts.

Une façon courante de générer des nombres entiers aléatoires dans un intervalle `[0,N)` est de générer d'abord un nombre flottant aléatoire $[0, 1)$ puis de multiplier le résultat par `N`. Si `N` dépasse 2^{24} (ou 2^{53}), alors vous n'êtes pas en mesure de générer tous les entiers dans l'intervalle `[0,N)`. De même, pour générer des nombres dans `[a,b)`, vous devez générer un nombre flottant aléatoire $[0, 1)$, puis multiplier le résultat par `b-a` et ajouter `a`. Le résultat n'est peut-être pas idéal en général, mais il est pratique.

Le programme suivant génère des nombres flottants aléatoires:

```
package main

import (
    "fmt"
)
```

```
func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

// toFloat32 -> [0,1)
func toFloat32(seed *uint64) float32 {
    x := splitmix64(seed)
    x &= 0xffffff // %2**24
    return float32(x)/float32(0xffffff)
}

// toFloat64 -> [0,1)
func toFloat64(seed *uint64) float64 {
    x := splitmix64(seed)
    x &= 0x1fffffffffffff // %2**53
    return float64(x)/float64(0x1fffffffffffff)
}

func main() {
    seed := uint64(1231114)
    fmt.Println(toFloat32(&seed))
    fmt.Println(toFloat64(&seed))
}
```

Vous pouvez préférer générer des nombres à virgule flottante dans l'intervalle `(0,1]`. Vous pouvez modifier la fonction en conséquence, par exemple:

```
// toFloat64 -> (0,1]
func toFloat64Alt(seed *uint64) float64 {
    x := splitmix64(seed)
    x &= 0x1fffffffffffff
    x += 1
    return float64(x)/float64(0x1fffffffffffff)
}
```

Dans certains cas, vous pouvez vouloir sacrifier le fait que les nombres sont uniformément distribués à des intervalles fixes afin de générer plus de valeurs en virgule flottante dans l'intervalle. Cawley (2023) [25] propose une stratégie. L'idée générale est de générer des nombres dans l'intervalle `[0.5,1]` avec une probabilité de 0.5 et un écart entre les nombres possibles de 2^{-54}, puis des nombres dans l'intervalle `[0.25,0.5]` avec une probabilité de 0.25 et un écart entre les nombres possibles de 2^{-55}, et ainsi de suite. Cawley propose la fonction suivante:

```
// toFloat64Cawley -> (0,1]
func toFloat64Cawley(seed *uint64) float64 {
  x := splitmix64(seed)
  e := bits.TrailingZeros64(x) - 11
  if e >= 0 {
    e = bits.TrailingZeros64(splitmix64(seed))
  }
  x = (((x >> 11) + 1) >> 1)
    - ((uint64(int64(e)) - 1011) << 52)
  return math.Float64frombits(x)
}
```

Une application amusante de la virgule flottante est l'estimation de la valeur de π. Si nous générons deux nombres à virgule flottante x, y dans $[0, 1), [0, 1)$, alors sur une surface de 1 (l'unité carrée), la surface était `x*x+y*y <= 1` devrait être $\pi/4$. Le programme suivant imprime une estimation de la valeur de π.

25. https://www.corsix.org/content/higher-quality-random-floats

```
package main

import (
    "fmt"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

// toFloat64 -> [0,1)
func toFloat64(seed *uint64) float64 {
    x := splitmix64(seed)
    x &= 0x1fffffffffffff // %2**53
    return float64(x) / float64(0x1fffffffffffff)
}

func main() {
    seed := uint64(1231114)
    N := 100000000
    circle := 0
    for i := 0; i < N; i++ {
      x := toFloat64(&seed)
      y := toFloat64(&seed)
      if x*x+y*y <= 1 {
          circle += 1
      }
```

```
    }
    fmt.Println(4 * float64(circle)/float64(N))
}
```

Bien entendu, les algorithmes pratiques peuvent nécessiter d'autres distributions telles que la distribution normale. Nous pouvons générer des valeurs en virgule flottante normalement distribuées de haute qualité à grande vitesse en utilisant la méthode Ziggurat [Marsaglia & Tsang, 2000] (http://www.jstatsoft.org/v05/i08/paper). La mise en œuvre n'est pas difficile, mais elle est technique. En particulier, elle nécessite une table précalculée. Généralement, nous générons des valeurs normalement distribuées avec une moyenne de zéro et un écart-type de un: nous multiplions souvent le résultat par le carré de l'écart-type. Nous multiplions souvent le résultat par la racine carrée de l'écart type souhaité, et nous ajoutons la moyenne souhaitée.

Distributions discrètes

Il arrive que l'on nous donne un ensemble de valeurs possibles et que chaque valeur ait une probabilité correspondante. Par exemple, nous pouvons choisir au hasard l'une des trois couleurs (rouge, bleu, vert) avec les probabilités correspondantes (20 %, 40 %, 40 %). Si ces valeurs sont peu nombreuses (par exemple trois), une approche standard consiste à choisir une roulette. Nous divisons l'intervalle de 0 à 1 en trois composantes distinctes, une pour chaque couleur: de 0 à 0,2, nous choisissons le rouge, de 0,2 à 0,6, nous choisissons le bleu, de 0,6 à 1,0, nous choisissons le vert.

Le programme suivant illustre cet algorithme:

```
package main

import (
    "fmt"
    "math/rand"
```

```
    "time"
)

func splitmix64(seed *uint64) uint64 {
    *seed += 0x9E3779B97F4A7C15
    z := *seed
    z = (z ^ (z >> 30))
    z *= (0xBF58476D1CE4E5B9)
    z = (z ^ (z >> 27))
    z *= (0x94D049BB133111EB)
    return z ^ (z >> 31)
}

func toFloat64(seed *uint64) float64 {
    x := splitmix64(seed)
    x &= 0x1fffffffffffff // %2**53
    return float64(x) / float64(0x1fffffffffffff)
}

func roulette(seed *uint64, colors []string,
 probabilities []float64) string {
    rand.Seed(time.Now().UnixNano())
    cumulProb := make([]float64, len(probabilities))
    cumulProb[0] = probabilities[0]
    for i := 1; i < len(probabilities); i++ {
      cumulProb[i] = cumulProb[i-1] + probabilities[i]
    }

    randomNumber := toFloat64(seed)

    if randomNumber < cumulProb[0] {
      return colors[0]
    }
    for i := 1; i < len(cumulProb); i++ {
```

```
        if randomNumber >= cumulProb[i-1]
        && randomNumber < cumulProb[i] {
            return colors[i]
        }
    }

    return colors[len(colors)-1]
}

func main() {
    seed := uint64(1231114)

    colors := []string{"red", "blue", "green"}
    probabilities := []float64{0.2, 0.4, 0.4}

    fmt.Println(roulette(&seed, colors, probabilities))
}
```

Si vous devez choisir une valeur parmi un grand nombre de valeurs, la méthode de la roulette peut s'avérer inefficace. Dans ce cas, nous pouvons utiliser la [méthode des alias](https://en.wikipedia.org/wiki/Alias_method).

Hachage cryptographique et nombres aléatoires

En règle générale, nous ne réimplémentons pas les fonctions cryptographiques. Il est préférable d'utiliser des implémentations bien testées. Elles sont généralement réservées aux cas où la sécurité est une préoccupation, car elles utilisent souvent plus de ressources.

Le hachage cryptographique des chaînes de caractères est conçu de manière à ce qu'il soit difficile de trouver deux chaînes qui entrent en collision (qui ont la même valeur de hachage). Ainsi, si vous recevez un message dont la

valeur de hachage vous a été communiquée à l'avance et que vous vérifiez que la valeur de hachage transmise et la valeur de hachage calculée à partir du message transmis correspondent, il y a de bonnes chances que le message n'ait pas été corrompu. Il est difficile (mais pas impossible) pour un attaquant de produire un message correspondant à la valeur de hachage qui vous a été communiquée. Pour hacher cryptographiquement une chaîne de caractères en Go, vous pouvez utiliser le code suivant:

```
package main

import (
    "crypto/sha256"
    "fmt"
)

func main() {
    message := "Hello, world!"
    hash := sha256.Sum256([]byte(message))
    fmt.Printf("Message: %s\nHash: %x\n", message, hash)
}
```

De même, vous pouvez vouloir générer des nombres aléatoires de manière cryptographique: dans ce cas, les nombres aléatoires produits sont difficilement prévisibles. Même si je vous donnais les dix derniers numéros, il serait difficile de prédire le suivant. Si vous deviez mettre en œuvre un logiciel pour un casino en ligne, vous devriez probablement utiliser des nombres aléatoires cryptographiques.

```
package main

import (
    "crypto/rand"
    "fmt"
    "math/big"
)
```

```
func main() {
    nBig, err := rand.Int(rand.Reader, big.NewInt(100))
    if err != nil {
      panic(err)
    }
    n := nBig.Int64()
    fmt.Printf("Here is a random %T between 0 and 99: %d\n",
   n, n)
}
```

Exercices du chapitre 5

Question 1

J'ai 100 éléments et je voudrais les hacher en 100 valeurs de hachage, et je veux être sûr que la probabilité que deux valeurs de hachage entrent en collision est faible. En supposant que vous disposiez de valeurs de hachage idéales (parfaitement) aléatoires, quelle est la probabilité de collision si je produis des valeurs de hachage de 32 bits? Et si j'utilise des valeurs de hachage de 128 bits?

Question 2

Etant donné la fonction suivante

```
func murmur64(h uint64) uint64 {
  h ^= h >> 33
  h *= 0xff51afd7ed558ccd
  h ^= h >> 33
  h *= 0xc4ceb9fe185ec53
  h ^= h >> 33
  return h
}
```

Trouvez son inverse.

Question 2

Ecrivez un programme qui teste l'uniformité du générateur aléatoire SplitMix, pour différentes semences. Vous pouvez générer des valeurs dans un intervalle. Certaines semences sont-elles meilleures que d'autres? Certains intervalles sont-ils meilleurs ou pires?

Question 3

La fonction suivante génère un seul nombre aléatoire de 32 bits en virgule flottante. Modifiez la fonction pour qu'elle génère deux nombres flottants de 32 bits à partir d'un seul appel à `splitmix64`.

```
func toFloat32(seed *uint64) float32 {
  x := splitmix64(seed)
  x &= 0xffffff // %2**24
  return float32(x)/float32(0xffffff)
}
```

Question 4

Modifiez la fonction `estimateCardinality` de façon à ce que lorsque la densité est de 1.0, elle réexécute l'analyse en utilisant deux fois plus de mémoire, et ainsi de suite, jusqu'à ce qu'elle puisse mesurer une densité non unitaire.

Chapitre 6

Lorsque nous programmons des logiciels, nous travaillons sur une abstraction d'un système. Le matériel informatique peut ne pas connaître vos fonctions, vos variables et vos données. Il ne voit que des bits et des instructions. Pourtant, pour écrire des logiciels efficaces, le programmeur doit connaître les caractéristiques du système sous-jacent. Heureusement, nous pouvons également utiliser le logiciel lui-même pour observer le comportement du système par le biais d'expériences.

Entre le logiciel et le matériel, il existe plusieurs couches telles que les compilateurs, le système d'exploitation et le matériel. Un bon programmeur doit tenir compte de ces couches si nécessaire. Un bon programmeur doit également comprendre le comportement de son logiciel en fonction de ces couches.

Benchmarks en Go

Pour mesurer les performances, nous mesurons souvent le temps nécessaire à l'exécution d'une fonction. La plupart des fonctions étant rapides, il peut être difficile de mesurer précisément le temps nécessaire à l'exécution d'une fonction si nous ne l'exécutons qu'une seule fois. Au lieu de cela, nous pouvons exécuter la fonction plusieurs fois et enregistrer le temps total. Nous pouvons ensuite diviser le temps total par le nombre d'exécutions. Il peut être difficile de décider combien de fois nous devons exécuter la fonction: cela dépend en partie de la rapidité de la fonction. Si une

fonction prend 6 secondes à s'exécuter, il n'est peut-être pas souhaitable ou nécessaire de l'exécuter trop souvent. Une stratégie plus simple consiste à spécifier une durée minimale et à appeler la fonction de manière répétée jusqu'à ce que la durée minimale soit atteinte ou dépassée.

Lorsque la fonction a un temps d'exécution court, nous appelons souvent le benchmark un microbenchmark. Nous utilisons les microbenchmarks pour comparer différentes implémentations de la même fonctionnalité ou pour mieux comprendre le système ou le problème. Nous devons toujours garder à l'esprit qu'un microbenchmark ne peut pas être utilisé seul pour justifier une optimisation logicielle. Les performances réelles dépendent de multiples facteurs qu'il est difficile de représenter dans un microbenchmark.

Il est important de noter que tous les benchmarks sont affectés par des erreurs de mesure et par des interférences provenant du système. Pour aggraver les choses, la distribution des temps peut ne pas suivre une distribution normale.

Tous les langages de programmation permettent d'exécuter des tests de performance. En Go, les outils facilitent l'écriture des tests. Vous pouvez importer le paquetage testing et créer une fonction avec le préfixe Benchmark et un paramètre de type pointeur `testing.B`. Par exemple, le programme suivant évalue le temps nécessaire pour calculer la factorielle de 10 en tant qu'entier:

```
package main

import (
    "fmt"
    "testing"
)

var fact int

func BenchmarkFactorial(b *testing.B) {
    for n := 0; n < b.N; n++ {
```

```
        fact = 1
        for i := 1; i <= 10; i++ {
            fact *= i
        }
    }
}

func main() {
    res := testing.Benchmark(BenchmarkFactorial)
    fmt.Println("BenchmarkFactorial", res)
}
```

Si vous placez des fonctions avec une telle signature (`BenchmarkSomething(b *testing.B)`) dans vos tests dans un projet, vous pouvez les exécuter avec la commande `go test -bench .` où le point (`.`) est une référence au paquetage courant. Pour n'en exécuter qu'un seul, vous pouvez spécifier un motif tel que `go test -bench Factorial` qui n'exécutera que les fonctions de benchmark contenant le mot `Factorial`.

Le champ `b.N` indique combien de fois la fonction de benchmark s'exécute. Le package de test ajuste cette valeur en l'augmentant jusqu'à ce que le benchmark s'exécute pendant au moins une seconde.

Mesurer les allocations de mémoire

En Go, chaque fonction possède sa propre *mémoire de pile*. Comme son nom l'indique, la mémoire de pile est allouée et désallouée dans l'ordre du dernier entré, premier sorti (LIFO). Cette mémoire n'est généralement utilisable qu'au sein de la fonction, et sa taille est souvent limitée. L'autre type de mémoire qu'un programme Go peut utiliser est la mémoire de tas. La mémoire de tas est allouée et désallouée dans un ordre aléatoire. Il n'y a qu'un seul tas partagé par toutes les fonctions.

Avec la mémoire de pile, il n'y a pas de risque que la mémoire soit perdue

ou mal utilisée puisqu'elle appartient à une fonction spécifique et peut être récupérée à la fin de la fonction. La mémoire de tas pose davantage de problèmes: le moment où la mémoire doit être récupérée n'est pas toujours clair. Les langages de programmation comme Go s'appuient sur un ramasse-miettes pour résoudre ce problème. Par exemple, lorsque nous créons une nouvelle tranche avec la fonction `make`, nous n'avons pas à nous soucier de récupérer la mémoire. Go la récupère automatiquement. Cependant, allouer et désallouer constamment de la mémoire peut s'avérer néfaste pour les performances. Dans de nombreux systèmes réels, la gestion de la mémoire devient un goulot d'étranglement pour les performances.

Il est donc parfois intéressant d'inclure l'utilisation de la mémoire dans le benchmark. Le paquetage de test Go vous permet de mesurer le nombre d'allocations de mémoire effectuées. Typiquement, en Go, cela correspond approximativement au nombre d'appels à `make` et au nombre d'objets que le ramasse miette doit gérer. Le programme étendu suivant calcule la factorielle en stockant son calcul dans des tranches allouées dynamiquement:

```
package main

import (
    "fmt"
    "testing"
)

var fact int

func BenchmarkFactorial(b *testing.B) {
    for n := 0; n < b.N; n++ {
      fact = 1
      for i := 1; i <= 10; i++ {
          fact *= i
      }
    }
}
```

```
func BenchmarkFactorialBuffer(b *testing.B) {
    for n := 0; n < b.N; n++ {
      buffer := make([]int, 11)
      buffer[0] = 1
      for i := 1; i <= 10; i++ {
          buffer[i] = i * buffer[i-1]
      }
    }
    b.ReportAllocs()
}

func BenchmarkFactorialBufferLarge(b *testing.B) {
    for n := 0; n < b.N; n++ {
      buffer := make([]int, 100001)
      buffer[0] = 1
      for i := 1; i <= 100000; i++ {
          buffer[i] = i * buffer[i-1]
      }
    }
    b.ReportAllocs()
}

func main() {
    res := testing.Benchmark(BenchmarkFactorial)
    fmt.Println("BenchmarkFactorial", res)
    resmem := testing.Benchmark(BenchmarkFactorialBuffer)
    fmt.Println("BenchmarkFactorialBuffer",
   resmem, resmem.MemString())
    resmem = testing.Benchmark
  (BenchmarkFactorialBufferLarge)
    fmt.Println("BenchmarkFactorialBufferLarge",
   resmem, resmem.MemString())
}
```

If you run such a Go program, you might get the following result:

```
BenchmarkFactorial 90887572       14.10 ns/op
BenchmarkFactorialBuffer 88609930         11.96 ns/op
0 B/op        0 allocs/op
BenchmarkFactorialBufferLarge   4408    249263 ns/op
802816 B/op     1 allocs/op
```

La dernière fonction alloue 802816 octets par opération, contrairement aux deux premières. Dans ce cas, si Go détermine que les données ne sont pas référencées après le retour de la fonction (un processus appelé *escape analysis*), et si la quantité de mémoire utilisée est suffisamment faible, il évitera d'allouer la mémoire au tas, préférant la mémoire de la pile. Dans le cas de la dernière fonction, l'utilisation de la mémoire est trop importante, d'où l'allocation sur le tas plutôt que sur la pile.

Mesurer l'utilisation de la mémoire

Votre système d'exploitation fournit de la mémoire à un processus en cours d'exécution en unités de pages. Le système d'exploitation ne peut pas fournir de la mémoire dans des unités plus petites qu'une page. Par conséquent, si vous allouez de la mémoire dans un programme, cela peut soit ne pas coûter de mémoire supplémentaire s'il y a déjà suffisamment de pages, soit obliger le système d'exploitation à fournir davantage de pages.

La taille d'une page dépend du système d'exploitation et de sa configuration. Elle varie souvent entre 4 et 16 kilo-octets, bien que des pages beaucoup plus grandes soient également possibles (par exemple, 1 giga-octet).

Une page est un tableau contigu d'adresses de mémoire virtuelle. Une page peut également représenter de la mémoire physique réelle. Toutefois, les systèmes d'exploitation ont tendance à ne faire correspondre que les pages utilisées à la mémoire physique. Un système d'exploitation peut fournir à un processus un nombre presque infini de pages, sans jamais les

affecter à la mémoire physique. Il n'est donc pas simple de déterminer la quantité de mémoire utilisée par un programme. Un programme peut sembler utiliser beaucoup de mémoire (virtuelle) alors qu'il n'utilise pas beaucoup de mémoire physique, et inversement.

La taille des pages influe à la fois sur les performances et sur l'utilisation de la mémoire. L'allocation de pages à un processus n'est pas gratuite, elle demande un certain effort. Entre autres, le système d'exploitation ne peut pas réutiliser telle quelle une page de mémoire provenant d'un autre processus. Cela constituerait une menace pour la sécurité, car vous pourriez avoir un accès indirect aux données stockées en mémoire par un autre processus. Cet autre processus pourrait avoir gardé en mémoire vos mots de passe ou d'autres informations sensibles. En règle générale, un système d'exploitation doit initialiser (par exemple, mettre à zéro) une page nouvellement attribuée. En outre, l'association des pages mémoire virtuelles (utilisées par les programmes) à la mémoire physique réelle (le matériel) est un processus complexe qui prend du temps. Chaque fois qu'un programme accède à une adresse mémoire virtuelle, le système doit traduire cette adresse en une adresse physique correspondante. Ce processus, appelé mappage des pages, repose sur des tables de pages maintenues par le système d'exploitation. Pour accélérer ce mappage, les processeurs modernes utilisent un mécanisme appelé *translation lookaside buffer* (TLB), ou mémoire cache de traduction d'adresses. Le TLB est une petite mémoire cache intégrée au processeur qui stocke les correspondances récentes entre adresses virtuelles et physiques. Lorsqu'une adresse virtuelle doit être traduite, le processeur consulte d'abord le TLB. Si la correspondance est trouvée, la traduction est rapide. Cependant, si l'adresse n'est pas dans le TLB, le processeur doit accéder aux tables de pages en mémoire, ce qui est un processus beaucoup plus lent, car il implique généralement des accès à la RAM et des calculs supplémentaires. Le TLB a une capacité limitée, ce qui signifie qu'il peut se remplir rapidement, surtout dans les systèmes exécutant de nombreux processus ou manipulant de grandes quantités de données. Lorsqu'il n'y a plus de place dans le TLB, il faut recalculer le mappage des pages. Cette opération peut impliquer plusieurs accès mémoire et des calculs complexes, ce qui

ralentit considérablement les performances du système. Les grandes pages peuvent donc améliorer les performances de certains programmes parce qu'elles réduisent le nombre de pages nécessaires. Cependant, les grandes pages obligent le système d'exploitation à fournir de la mémoire à un processus en morceaux plus importants, ce qui risque de gaspiller une mémoire précieuse. Vous pouvez écrire un programme Go qui affiche la taille des pages de votre système:

```
import (
    "fmt"
    "os"
)

func main() {
    pageSize := os.Getpagesize()
    fmt.Printf("Page size: %d bytes (%d KB)\n",
   pageSize, pageSize/1024)
}
```

Avec Go, il est relativement facile de mesurer le nombre de pages allouées à un programme par le système d'exploitation. Néanmoins, il convient d'être prudent. Comme Go utilise un ramasse-miettes pour libérer la mémoire allouée, il peut y avoir un délai entre le moment où vous n'avez plus besoin de mémoire et la libération effective de la mémoire. Vous pouvez forcer Go à appeler immédiatement le ramasse miette avec l'appel de fonction `runtime.GC()`. En pratique, vous devriez rarement invoquer délibérément le ramasse miette, mais pour nos besoins (mesurer l'utilisation de la mémoire), c'est utile.

Il existe plusieurs mesures de la mémoire. En Go, les plus utiles sont `HeapSys` et `HeapAlloc`. La première indique combien de mémoire (en octets) a été donnée au programme par le système d'exploitation. La seconde valeur, qui est généralement inférieure, indique la quantité de mémoire activement utilisée par le programme.

Le programme suivant alloue des tranches de plus en plus grandes, puis

de plus en plus petites. En théorie, l'utilisation de la mémoire devrait d'abord augmenter, puis diminuer:

```
package main

import (
    "fmt"
    "os"
    "runtime"
)

func main() {
    pageSize := os.Getpagesize()
    var m runtime.MemStats
    runtime.GC()
    runtime.ReadMemStats(&m)
    fmt.Printf(
      "Sys = %.3f MiB, Alloc =  %.3f MiB,  %.3f pages\n",
      float64(m.HeapSys)/1024.0/1024.0,
      float64(m.HeapAlloc)/1024.0/1024.0,
      float64(m.HeapSys)/float64(pageSize),
    )
    i := 100
    for ; i < 1000000000; i *= 10 {
      runtime.GC()
      s := make([]byte, i)
      runtime.ReadMemStats(&m)
      fmt.Printf(
          "%.3f MiB, Sys = %.3f MiB, Alloc =  %.3f MiB,"
      +"  %.3f pages\n",
          float64(len(s))/1024.0/1024.0,
          float64(m.HeapSys)/1024.0/1024.0,
          float64(m.HeapAlloc)/1024.0/1024.0,
          float64(m.HeapSys)/float64(pageSize),
      )
```

```
    }
    for ; i >= 100; i /= 10 {
      runtime.GC()
      s := make([]byte, i)
      runtime.ReadMemStats(&m)
      fmt.Printf(
          "%.3f MiB, Sys = %.3f MiB, Alloc =  %.3f MiB,"
      +"  %.3f pages\n",
          float64(len(s))/1024.0/1024.0,
          float64(m.HeapSys)/1024.0/1024.0,
          float64(m.HeapAlloc)/1024.0/1024.0,
          float64(m.HeapSys)/float64(pageSize),
      )
    }
    runtime.GC()
    runtime.ReadMemStats(&m)
    fmt.Printf(
      "Sys = %.3f MiB, Alloc =  %.3f MiB,  %.3f pages\n",
      float64(m.HeapSys)/1024.0/1024.0,
      float64(m.HeapAlloc)/1024.0/1024.0,
      float64(m.HeapSys)/float64(pageSize),
    )
}
```

Le programme appelle `os.Getpagesize()` pour obtenir la taille de la page mémoire du système sous-jacent en octets sous la forme d'un entier, et l'affecte à une variable `pageSize`. Il déclare une variable `m` de type `runtime.MemStats`, qui est une structure contenant diverses statistiques sur l'allocateur de mémoire et le ramasse-miettes. Le programme appelle à plusieurs reprises `runtime.GC()` pour déclencher manuellement un cycle de ramassage des ordures, ce qui peut libérer de la mémoire et la rendre disponible pour la libération. Il appelle `runtime.ReadMemStats(&m)` pour remplir la variable `m` avec les statistiques actuelles de la mémoire. Nous pouvons réutiliser la même variable `m` d'un appel à l'autre. Le but de

ce programme est de démontrer comment l'utilisation de la mémoire d'un programme Go change en fonction de la taille et de la fréquence des allocations et désallocations de mémoire, et comment le ramasse miette et le runtime affectent la libération de la mémoire. Le programme affiche l'utilisation de la mémoire avant et après chaque allocation, et montre comment les valeurs `m.HeapSys`, `m.HeapAlloc`, et `m.HeapSys / pageSize` augmentent et diminuent en conséquence.

Si vous exécutez ce programme, vous constaterez peut-être qu'un programme a tendance à conserver la mémoire que vous avez allouée et libérée par la suite. Il s'agit en partie d'une question d'optimisation: l'acquisition de mémoire prend du temps et nous souhaitons éviter de rendre des pages pour les redemander ensuite. Cela montre qu'il peut être difficile de déterminer la quantité de mémoire utilisée par un programme.

Le programme peut imprimer quelque chose comme ce qui suit:

```
$ go run mem.go
HeapSys = 3.719 MiB, HeapAlloc =  0.367 MiB,
238.000 pages
0.000 MiB, HeapSys = 3.719 MiB, HeapAlloc =  0.367 MiB,
238.000 pages
0.001 MiB, HeapSys = 3.719 MiB, HeapAlloc =  0.383 MiB,
238.000 pages
0.010 MiB, HeapSys = 3.688 MiB, HeapAlloc =  0.414 MiB,
236.000 pages
0.095 MiB, HeapSys = 3.688 MiB, HeapAlloc =  0.477 MiB,
236.000 pages
0.954 MiB, HeapSys = 3.688 MiB, HeapAlloc =  1.336 MiB,
236.000 pages
9.537 MiB, HeapSys = 15.688 MiB, HeapAlloc =  9.914 MiB,
1004.000 pages
95.367 MiB, HeapSys = 111.688 MiB, HeapAlloc =  95.750 MiB,
7148.000 pages
953.674 MiB, HeapSys = 1067.688 MiB,
HeapAlloc =  954.055 MiB,
```

```
68332.000 pages
95.367 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  95.750 MiB,
68332.000 pages
9.537 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  9.914 MiB,
68332.000 pages
0.954 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  1.336 MiB,
68332.000 pages
0.095 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  0.477 MiB,
68332.000 pages
0.010 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  0.414 MiB,
68332.000 pages
0.001 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  0.383 MiB,
68332.000 pages
0.000 MiB, HeapSys = 1067.688 MiB, HeapAlloc =  0.375 MiB,
68332.000 pages
HeapSys = 1067.688 MiB, HeapAlloc =  0.375 MiB,
68332.000 pages
```

Observez comment, au tout début et à la toute fin, plus d'un tiers de mégaoctet de mémoire (238 pages) est répété comme étant utilisé. En outre, plus de 68 000 pages restent allouées au programme à la toute fin, même si aucune structure de données ne reste dans le champ d'application de la fonction `main`.

Incorporation

L'une des techniques d'optimisation les plus puissantes qu'un compilateur puisse mettre en œuvre est l'incorporation de fonctions: le compilateur intègre certaines des fonctions appelées directement dans les fonctions appelantes.

Avec Go, il est facile de savoir quelles fonctions sont intégrées. On peut aussi facilement demander à ce que le compilateur ne fasse pas d'incorporation en ajoutant la ligne `//go:noinline` juste avant une fonction.

Considérons ce programme qui contient deux benchmarks où nous addi-

tionnons tous les entiers impairs dans un intervalle.

```
package main

import (
    "fmt"
    "testing"
)

func IsOdd(i int) bool {
    return i%2 == 1
}

//go:noinline
func IsOddNoInline(i int) bool {
    return i%2 == 1
}

func BenchmarkCountOddInline(b *testing.B) {
    for n := 0; n < b.N; n++ {
      sum := 0
      for i := 1; i < 1000; i++ {
          if IsOdd(i) {
            sum += i
          }
      }
    }
}

func BenchmarkCountOddNoinline(b *testing.B) {
    for n := 0; n < b.N; n++ {
      sum := 0
      for i := 1; i < 1000; i++ {
          if IsOddNoInline(i) {
            sum += i
```

```
            }
        }
    }
}

func main() {
    res1 := testing.Benchmark(BenchmarkCountOddInline)
    fmt.Println("BenchmarkCountOddInline", res1)
    res2 := testing.Benchmark(BenchmarkCountOddNoinline)
    fmt.Println("BenchmarkCountOddNoinline", res2)
}
```

En Go, le drapeau `-gcflags=-m` indique au compilateur de rapporter les principales optimisations qu'il effectue. Si vous appelez ce programme `simpleinline.go` et que vous le compilez avec la commande `go build -gcflags=-m simpleinline.go`, vous verrez ce qui suit:

```
$ go build -gcflags=-m simpleinline.go
./simpleinline.go:8:6: can inline IsOdd
./simpleinline.go:21:12: inlining call to IsOdd
...
```

Si vous exécutez le test de référence, vous devriez constater que la version intégrée est beaucoup plus rapide:

```
$ go run simpleinline.go
BenchmarkCountOddInline   3716786        294.6 ns/op
BenchmarkCountOddNoinline  1388792         864.8 ns/op
```

L'intégration n'est pas toujours bénéfique: dans certains cas, il peut générer des binaires volumineux et même ralentir le logiciel. Les processeurs modernes disposent généralement d'un cache séparé pour le code, mais sa taille est limitée et les échecs de cache peuvent annuler les avantages de l'intégration. Cependant, lorsqu'il est applicable, l'intégration peut avoir un effet bénéfique important.

Go s'efforce autant que possible d'intégrer les fonctions, mais il a ses

limites. Par exemple, les compilateurs ont souvent du mal à intégrer des fonctions récursives. Prenons l'exemple de deux fonctions factorielles, l'une récursive et l'autre non.

```
package main

import (
    "fmt"
    "testing"
)

var array = make([]int, 1000)

func Factorial(n int) int {
    if n < 0 {
      return 0
    }
    if n == 0 {
      return 1
    }
    return n * Factorial(n-1)
}

func FactorialLoop(n int) int {
    result := 1
    for i := 1; i <= n; i++ {
      result *= i
    }
    return result
}

func BenchmarkFillNoinline(b *testing.B) {
    for n := 0; n < b.N; n++ {
      for i := 1; i < 1000; i++ {
          array[i] = Factorial(i)
```

```
      }
    }
}

func BenchmarkFillInline(b *testing.B) {
    for n := 0; n < b.N; n++ {
      for i := 1; i < 1000; i++ {
          array[i] = FactorialLoop(i)
      }
    }
}

func main() {
    res1 := testing.Benchmark(BenchmarkFillNoinline)
    fmt.Println("BenchmarkFillNoinline", res1)
    res2 := testing.Benchmark(BenchmarkFillInline)
    fmt.Println("BenchmarkFillInline", res2)
    fmt.Println(float64(res1.NsPerOp())
  / float64(res2.NsPerOp()))
}
```

Bien que `FactorialLoop` et `Factorial` soient tous deux équivalents, en exécutant ce programme, vous devriez constater que la fonction non récursive (`FactorialLoop`) est beaucoup plus rapide. Le résultat possible de ce programme est le suivant. Dans ce cas, la fonction non récursive est plus de trois fois plus rapide.

```
BenchmarkFillNoinline     1165     1040019 ns/op
BenchmarkFillInline      3808      304275 ns/op
```

Préchargeurs matériels

Le chargement des données à partir de la mémoire prend souvent plusieurs nanosecondes. Pendant que le processeur attend les données, il peut être

contraint d'attendre sans faire de travail utile.

Les préchargeurs matériels des processeurs modernes anticipent les accès mémoire en chargeant les données dans le cache avant leur demande, optimisant ainsi les performances. Leur efficacité varie selon le motif d'accès: les lectures séquentielles profitent d'un préchargement efficace, contrairement aux accès aléatoires.

Pour tester l'impact des préchargeurs, nous proposons un programme en Go qui utilise une unique fonction d'accès à un tableau, avec un tableau d'indices configuré pour des accès séquentiels ou aléatoires. Le temps d'exécution est mesuré pour comparer les performances. Le programme suivant initialise un grand tableau et effectue des accès en utilisant un tableau d'indices, soit séquentiels, soit aléatoires, avec des mesures répétées pour plus de fiabilité.

```
package main

import (
  "fmt"
  "math/rand"
  "time"
)

const (
  arraySize = 1 << 26
  runs      = 10
)

func access(arr []int32, indices []int) int64 {
  var sum int64
  for _, i := range indices {
    sum += int64(arr[indices[i]])
  }
  return sum
}
```

```
func main() {
  arr := make([]int32, arraySize)
  for i := range arr {
    arr[i] = int32(i)
  }

  sequentialIndices := make([]int, arraySize)
  for i := range sequentialIndices {
    sequentialIndices[i] = i
  }

  randomIndices := make([]int, arraySize)
  for i := range randomIndices {
    randomIndices[i] = i
  }
  rand.Shuffle(len(randomIndices), func(i, j int) {
    randomIndices[i], randomIndices[j]
      = randomIndices[j], randomIndices[i]
  })

  sequentialTimes := make([]float64, runs)
  for i := 0; i < runs; i++ {
    start := time.Now()
    _ = access(arr, sequentialIndices)
    sequentialTimes[i] = time.Since(start).Seconds() * 1000
  }

  randomTimes := make([]float64, runs)
  for i := 0; i < runs; i++ {
    start := time.Now()
    _ = access(arr, randomIndices)
    randomTimes[i] = time.Since(start).Seconds() * 1000
  }
```

```
  var seqMin, seqMax, seqAvg, randMin, randMax,
    randAvg float64
  for i := 0; i < runs; i++ {
    if i == 0 || sequentialTimes[i] < seqMin {
      seqMin = sequentialTimes[i]
    }
    if i == 0 || sequentialTimes[i] > seqMax {
      seqMax = sequentialTimes[i]
    }
    seqAvg += sequentialTimes[i]
    if i == 0 || randomTimes[i] < randMin {
      randMin = randomTimes[i]
    }
    if i == 0 || randomTimes[i] > randMax {
      randMax = randomTimes[i]
    }
    randAvg += randomTimes[i]
  }
  seqAvg /= runs
  randAvg /= runs

  fmt.Printf("Seq (ms) : min=%.2f, max=%.2f, mean=%.2f\n",
    seqMin, seqMax, seqAvg)
  fmt.Printf("Alea (ms) : min=%.2f, max=%.2f, mean=%.2f\n",
    randMin, randMax, randAvg)
}
```

Ce programme utilise une seule fonction d'accès prenant un tableau d'indices, qui est configuré pour être soit séquentiel, soit aléatoire. Les accès séquentiels devraient être plus rapides grâce aux préchargeurs matériels, tandis que les accès aléatoires, moins prévisibles, seront plus lents. Les temps d'exécution sont mesurés sur 10 itérations, avec affichage des valeurs minimales, maximales et moyennes en millisecondes. Les résultats dépendent de l'architecture du processeur et des caractéristiques

du cache. Voici un résultat possible.

```
Seq (ms) : min=15.80, max=41.84, mean=18.86
Alea (ms) : min=219.73, max=225.37, mean=222.56
```

Dans ce cas, nous constatons que l'approche avec accès séquentiel est environ dix fois plus rapide que l'approche avec accès aléatoire.

Ligne de cache

Nos ordinateurs lisent et écrivent la mémoire en utilisant de petits blocs de mémoire appelés *lignes de cache*. La taille de la ligne de cache est généralement fixe et petite (par exemple, 64 ou 128 octets). Pour tenter de mesurer la taille de la ligne de cache, nous pouvons utiliser une copie stridée. À partir d'un grand tableau, nous copions tous les N octets dans un autre grand tableau. Nous répétons ce processus N fois. Ainsi, si le tableau original contient 1000 octets, nous copions toujours 1024 octets, que $N = 1$, $N = 2$, $N = 4$, ou $N = 8$.

Lorsque N est suffisamment grand (disons $N = 16$), le problème devrait être essentiellement lié à la mémoire. pas limitées par le nombre d'instructions, mais par la capacité du système à charger et à stocker des lignes de cache. Si N est plus grand que deux fois la ligne de cache, alors nous pouvons effectivement sauter une ligne de cache sur deux. On s'attend à ce qu'une grande bande soit beaucoup plus rapide parce qu'on saute de nombreuses lignes de cache. Si N est plus petit que la ligne de cache, il faut accéder à chaque ligne de cache. Par conséquent, si N est plus petit que la ligne de cache, la vitesse ne devrait pas être sensible à la valeur exacte de N.

L'une des limites de cette approche est que les processeurs peuvent aller chercher plus de lignes de cache que nécessaire, de sorte que nous pouvons surestimer la taille de la ligne de cache. Cependant, à moins que la bande passante de la mémoire ne soit excessivement abondante, nous devrions nous attendre à ce que les processeurs essaient de limiter le nombre de lignes de cache extraites.

Faisons une expérience. Pour chaque taille de ligne, nous répétons l'expérience 10 fois et enregistrons le maximum, le minimum et la moyenne. Considérons le programme suivant.

```
package main

import (
    "fmt"
    "time"
)

const size = 33554432 // 32 MB
func Cpy(arr1 []uint8, arr2 []uint8, slice int) {
    for i := 0; i < len(arr1); i += slice {
      arr2[i] = arr1[i]
    }
}

func AverageMinMax(f func() float64) (float64, float64,
float64) {
    var sum float64
    var minimum float64
    var maximum float64

    for i := 0; i < 10; i++ {
      arr1 = make([]uint8, size)
      arr2 = make([]uint8, size)

      v := f()
      sum += v
      if i == 0 || v < minimum {
          minimum = v
      }
      if i == 0 || v > maximum {
          maximum = v
```

```
      }
    }
    return sum / 10, minimum, maximum
}

var arr1 []uint8
var arr2 []uint8

func run(size int, slice int) float64 {
    start := time.Now()
    times := 10
    for i := 0; i < times*slice; i++ {
      Cpy(arr1, arr2, slice)
    }
    end := time.Now()
    dur := float64(end.Sub(start)) / float64(times*slice)
    return dur
}

func main() {
    for slice := 16; slice <= 4096; slice *= 2 {
      a, m, M := AverageMinMax(
    func() float64 { return run(size, slice-1) })
      fmt.Printf("%10d: %10.1f GB/s [%4.1f - %4.1f]\n",
      slice-1, float64(size)/a,
    float64(size)/M, float64(size)/m)
    }
}
```

Nous pouvons obtenir le résultat suivant:

```
$ go run cacheline.go                    1
    15:      23.6 GB/s [21.3 - 24.4]
    31:      24.3 GB/s [23.8 - 24.5]
    63:      24.2 GB/s [23.6 - 24.6]
```

```
  127:     26.9 GB/s [23.8 - 27.9]
  255:     40.8 GB/s [37.8 - 43.6]
  511:    162.0 GB/s [130.4 - 203.4]
 1023:    710.0 GB/s [652.0 - 744.4]
 2047:    976.1 GB/s [967.1 - 983.8]
 4095:   1247.4 GB/s [1147.7 - 1267.0]
```

Nous constatons que les performances augmentent considérablement lorsque le pas passe de 127 à 255. Cela suggère que la ligne de cache a 128 octets. Si vous exécutez ce même benchmark sur votre propre système, vous obtiendrez peut-être un résultat différent.

Les résultats doivent être interprétés avec précaution: nous ne mesurons pas une vitesse de copie de 1247,4 Go/s. Nous pouvons plutôt copier de grands tableaux de données. Nous pouvons plutôt copier de grands tableaux à une telle vitesse si nous ne copions qu'un octet sur 4095 octets.

Cache de l'unité centrale

Lorsque nous programmons, nous ne pensons souvent pas directement à la mémoire. Lorsque nous considérons que nos données utilisent de la mémoire, nous l'envisageons souvent comme homogène: la mémoire est comme une grande toile uniforme sur laquelle l'ordinateur écrit et lit ses données. Cependant, votre mémoire principale (RAM) est généralement mise en mémoire tampon à l'aide d'une petite quantité de mémoire qui réside à proximité du cœur du processeur (cache du CPU). Nous avons souvent plusieurs couches de mémoire cache (par exemple, L1, L2, L3): L1 est généralement petite mais très rapide, tandis que, par exemple, L3 est plus grande mais plus lente.

Il est possible de mesurer empiriquement l'effet de la mémoire cache. Si vous prenez un petit tableau et que vous le mélangez de manière aléatoire, les données seront principalement déplacées dans le cache de l'unité centrale, qui est rapide. Si vous prenez un tableau plus grand, vous déplacerez les données en mémoire sans l'aide du cache, un processus beaucoup plus lent. Le fait de mélanger des tableaux de plus en plus

grands est donc un moyen de déterminer la taille de votre cache. Il peut s'avérer difficile de savoir exactement combien de couches de cache vous avez et quelle est la taille de chaque couche. Cependant, il est généralement possible de savoir si votre tableau est beaucoup plus grand que le cache de l'unité centrale.

Nous allons écrire une fonction de mélange aléatoire: `Shuffle(arr []uint32)`. Elle utilise un algorithme appelé Fisher-Yates shuffle, qui consiste à parcourir le tableau à l'envers et à échanger chaque élément avec un autre choisi aléatoirement parmi ceux qui le précèdent. La fonction utilise une variable de départ pour générer des nombres aléatoires à partir d'une formule mathématique. Pour nos besoins, nous utilisons un générateur de nombres simpliste: nous multiplions la semence par l'index. La fonction bits.Mul64 calcule le produit de deux nombres de 64 bits et renvoie le résultat sous la forme de deux nombres de 32 bits: le plus significatif (hi) et le moins significatif. La valeur la plus significative est nécessairement comprise entre 0 et i (inclusivement). Nous utilisons cette valeur la plus significative comme index aléatoire. La fonction échange ensuite les éléments en utilisant l'assignation multiple. Nous appelons cette fonction de mélange plusieurs fois, sur des entrées de tailles différentes. Nous indiquons le temps normalisé par la taille de l'entrée.

```
package main

import (
    "fmt"
    "math/bits"
    "time"
)

func Shuffle(arr []uint32) {
    seed := uint64(1234)
    for i := len(arr) - 1; i > 0; i-- {
      seed += 0x9E3779B97F4A7C15
      hi, _ := bits.Mul64(seed, uint64(i+1))
      j := int(hi)
```

```
      arr[i], arr[j] = arr[j], arr[i]
    }
}

func AverageMinMax(f func() float64) (float64, float64,
float64) {
    var sum float64
    var minimum float64
    var maximum float64

    for i := 0; i < 10; i++ {
      v := f()
      sum += v
      if i == 0 || v < minimum {
          minimum = v
      }
      if i == 0 || v > maximum {
          maximum = v
      }
    }
    return sum / 10, minimum, maximum
}

func run(size int) float64 {
    arr := make([]uint32, size)

    for i := range arr {
      arr[i] = uint32(i + 1)
    }
    start := time.Now()
    end := time.Now()
    times := 0
    for ; end.Sub(start) < 100_000_000; times++ {
      Shuffle(arr)
```

```
      end = time.Now()
    }
    dur := float64(end.Sub(start)) / float64(times)
    return dur / float64(size)
}

func main() {
    for size := 4096; size <= 33554432; size *= 2 {
      fmt.Printf("%20d KB ", size/1024*4)
      a, m, M :=
    AverageMinMax(func() float64 { return run(size) })
      fmt.Printf(" %.2f [%.2f, %.2f]\n", a, m, M)
    }
}
```

Un résultat possible de l'exécution de ce programme pourrait être le suivant:

```
go run cache.go
          16 KB  0.70 [0.66, 0.93]
          32 KB  0.65 [0.64, 0.66]
          64 KB  0.64 [0.64, 0.66]
         128 KB  0.64 [0.64, 0.67]
         256 KB  0.65 [0.64, 0.66]
         512 KB  0.70 [0.70, 0.71]
        1024 KB  0.77 [0.76, 0.79]
        2048 KB  0.83 [0.82, 0.84]
        4096 KB  0.87 [0.86, 0.90]
        8192 KB  0.92 [0.91, 0.95]
       16384 KB  1.10 [1.06, 1.24]
       32768 KB  2.34 [2.28, 2.52]
       65536 KB  3.90 [3.70, 4.25]
      131072 KB  5.66 [4.80, 9.78]
```

Nous constatons qu'entre 16 Ko et 16384 Ko, le temps par élément

mélangé n'augmente pas beaucoup, même si nous doublons à plusieurs reprises la taille de l'entrée. Cependant, entre 16384 Ko et 32768 Ko, le temps par élément double. Il double ensuite systématiquement chaque fois que la taille du tableau est doublée. Cela suggère que la taille du cache de l'unité centrale est d'environ 16384 Ko dans ce cas.

Bande passante de la mémoire

Vous ne pouvez lire et écrire dans la mémoire que jusqu'à une vitesse maximale. Il peut être difficile de mesurer ces limites. En particulier, vous pouvez avoir besoin de plusieurs cœurs dans un système multicœur pour obtenir la meilleure mémoire possible. Pour simplifier, considérons la mémoire de lecture maximale.

De nombreux grands systèmes ne disposent pas d'un bande passante unique. Par exemple, de nombreux grands systèmes reposent sur le système NUMA: NUMA signifie Non-Uniform Memory Access (accès non uniforme à la mémoire). Dans un système NUMA, chaque processeur dispose de sa propre mémoire locale, à laquelle il peut accéder plus rapidement que la mémoire des autres processeurs.

La bande passante dépend également, dans une certaine mesure, de la quantité de mémoire demandée. Si la mémoire tient dans la mémoire cache du processeur, seul le premier accès peut être coûteux. Une très grande zone de mémoire peut ne pas tenir dans la RAM et nécessiter un stockage sur disque. Même si elle tient dans la RAM, une région de mémoire trop grande peut nécessiter de nombreuses pages de mémoire, et l'accès à toutes ces pages peut provoquer un saut de page en raison des limites de la mémoire tampon de translation.

Si la mémoire est accédée à des endroits aléatoires, il peut être difficile pour le système de maintenir une bande passante maximale, car le système ne peut pas prédire facilement où se produira la prochaine charge de mémoire. Pour obtenir la meilleure bande passante possible, il est préférable d'accéder à la mémoire de manière linéaire ou selon un modèle prévisible.

Considérons le code suivant:

```
package main

import (
    "fmt"
    "time"
)

func run() float64 {
    bestbandwidth := 0.0
    arr := make([]uint8, 2*1024*1024*1024) // 4 GB
    for i := 0; i < len(arr); i++ {
      arr[i] = 1
    }
    for t := 0; t < 20; t++ {
      start := time.Now()
      acc := 0
      for i := 0; i < len(arr); i += 64 {
          acc += int(arr[i])
      }
      end := time.Now()
      if acc != len(arr)/64 {
          panic("!!!")
      }
      bandwidth := float64(len(arr))
      / end.Sub(start).Seconds() / 1024 / 1024 / 1024
      if bandwidth > bestbandwidth {
          bestbandwidth = bandwidth
      }
    }
    return bestbandwidth
}

func main() {
```

```
    for i := 0; i < 10; i++ {
      fmt.Printf(" %.2f GB/s\n", run())
    }
}
```

Le code définit deux fonctions: run et main. La fonction main est le point d'entrée du programme et appelle la fonction run 10 fois, en imprimant le résultat à chaque fois. La fonction run est une fonction personnalisée qui mesure la bande passante de la mémoire du système. Pour ce faire, elle exécute les étapes suivantes:

Elle déclare une variable appelée bestbandwidth et l'initialise à 0,0. Cette variable stocke la valeur de bande passante la plus élevée obtenue lors de l'exécution de la fonction. Elle crée une tranche d'octets (uint8) appelée arr, d'une longueur équivalente à 4 Go. La tranche est initialisée avec 1s. La boucle n'accède qu'à chaque 64e élément de la tranche, en sautant le reste. Étant donné que la plupart des systèmes ont une taille de ligne de cache de 64 octets ou plus, il suffit de toucher chaque ligne de cache. Le code calcule la bande passante en divisant la taille de la tranche (en octets) par la différence entre les heures de fin et de début (en secondes), puis en divisant trois fois par 1024 pour convertir le résultat en gigaoctets par seconde (Go/s). Le code répète la mesure 20 fois et renvoie le meilleur résultat, afin de tenir compte des variations possibles des performances du système. Le code imprime le résultat 10 fois, pour montrer la cohérence de la mesure.

Latence de la mémoire et parallélisme

La latence est souvent décrite comme le délai entre le début d'une requête et le moment où vous êtes servi. Ainsi, si vous allez au restaurant, le temps de latence qui vous intéresse est le temps qui s'écoule avant que vous puissiez commencer à manger. La latence est distincte du débit: un restaurant peut être en mesure de servir des centaines de clients à la fois, mais avoir une latence élevée (longs délais pour chaque client). Si

vous mettez beaucoup de données sur un très grand disque, vous pouvez mettre ce disque dans un camion et le conduire entre deux villes. Cela pourrait représenter une bande passante élevée (beaucoup de données sont déplacées par unité de temps), mais le temps de latence pourrait être assez faible (des heures). De la même manière, vous pouvez envoyer un laser à votre partenaire lorsque le dîner est prêt: l'information peut arriver sans délai même si vous êtes très loin, mais vous communiquez peu d'informations (faible débit). La loi de Little permet d'exprimer ce compromis entre latence et débit: $L = \lambda W$ où L est le nombre moyen d'éléments dans le système, λ est le débit (taux d'arrivée moyen à long terme de nouveaux éléments) et W est la latence, c'est-à-dire le temps moyen d'attente des éléments. Ainsi, si vous souhaitez avoir L clients à tout moment dans votre restaurant, et que moins de clients arrivent, vous devriez servir les clients avec des délais plus longs. Et ainsi de suite. La loi de Little s'applique également à nos sous-systèmes de mémoire: les ordinateurs peuvent supporter un nombre maximal de demandes de mémoire, chaque demande de mémoire a un temps de latence et il existe une bande passante globale. Si le temps de latence ne s'améliore pas, nous pouvons encore améliorer la bande passante ou le débit en augmentant le nombre de requêtes qui peuvent être supportées simultanément. Malheureusement, les concepteurs de systèmes sont souvent contraints de faire ce choix, et il n'est donc pas courant de voir les temps de latence de la mémoire stagner ou s'aggraver malgré l'amélioration rapide de la bande passante. Une illustration courante du concept de latence de la mémoire est la traversée d'une liste chaînée. En informatique, une liste chaînée est une structure de données composée de nœuds, et chaque nœud est *lié* (par un pointeur) au nœud suivant. Les nœuds peuvent ne pas être disposés consécutivement dans la mémoire, mais même s'ils le sont, l'accès à chaque nœud successif nécessite au moins un petit délai. Sur les processeurs actuels, il faut souvent au moins 3 cycles pour charger les données de la mémoire, même si la mémoire est en cache. Ainsi, déterminer la longueur de la liste en parcourant l'ensemble de la liste liée peut prendre du temps, et la majeure partie de ce temps n'est constituée que des délais successifs. Le code suivant évalue le temps nécessaire pour parcourir une liste chaînée composée d'un million de nœuds. Bien que le

temps varie en fonction de votre système, il peut représenter une fraction non négligeable de milliseconde.

```
package main

import (
    "fmt"
    "testing"
)

type Node struct {
    data int
    next *Node
}

func build(volume int) *Node {
    var head *Node
    for i := 0; i < volume; i++ {
      head = &Node{i, head}
    }
    return head
}

var list *Node
var N int

func BenchmarkLen(b *testing.B) {
    for n := 0; n < b.N; n++ {
      len := 0
      for p := list; p != nil; p = p.next {
          len++
      }
      if len != N {
          b.Fatalf("invalid length: %d", len)
      }
```

```
    }
}

func main() {
    N = 1000000
    list = build(N)
    res := testing.Benchmark(BenchmarkLen)
    fmt.Println("milliseconds: ",
   float64(res.NsPerOp())/1e6)

    fmt.Println("nanoseconds per el.",
   float64(res.NsPerOp())/float64(N))
}
```

Dans ce code, une structure Node est définie avec deux champs: `data` est un entier représentant la valeur stockée dans le nœud, `next` est un pointeur sur le nœud suivant dans la liste chaînée. Nous pourrions également ajouter un pointeur sur le nœud précédent, mais ce n'est pas nécessaire dans notre cas. La fonction build crée une liste chaînée de nœuds. Elle initialise une liste chaînée vide (`head` est initialement `nil`). Elle itère de `0` à `volume-1`, en créant un nouveau nœud avec la valeur `i` et en pointant son suivant vers la tête actuelle. Le nouveau nœud devient la nouvelle tête. La fonction renvoie la tête finale de la liste chaînée. La fonction principale initialise deux variables globales (`list` et `N`) stockant respectivement la tête de la liste et la longueur attendue. Ces valeurs sont utilisées par la fonction `BenchmarkLen`. Ce code montre comment créer une liste chaînée, calculer sa longueur et évaluer les performances du calcul de la longueur. Notre calcul de longueur est presque entièrement limité par la latence de la mémoire, c'est-à-dire le temps nécessaire pour accéder à la mémoire. Les calculs que nous effectuons (comparaisons, incréments) n'ont pas d'importance pour les performances. Pour illustrer notre observation, nous pouvons essayer de parcourir deux listes chaînées simultanément, comme dans cet exemple:

```
package main

import (
    "fmt"
    "testing"
)

type Node struct {
    data int
    next *Node
}

func build(volume int) *Node {
    var head *Node
    for i := 0; i < volume; i++ {
      head = &Node{i, head}
    }
    return head
}

var list1 *Node
var list2 *Node

var N int

func BenchmarkLen(b *testing.B) {
    for n := 0; n < b.N; n++ {
      len := 0
      for p1, p2 := list1, list2;
      p1 != nil && p2 != nil; p1, p2 = p1.next, p2.next {
          len++
      }
      if len != N {
          b.Fatalf("invalid length: %d", len)
```

```
        }
    }
}

func main() {
    N = 1000000
    list1 = build(N)
    list2 = build(N)

    res := testing.Benchmark(BenchmarkLen)
    fmt.Println("milliseconds: ",
   float64(res.NsPerOp())/1e6)

    fmt.Println("nanoseconds per el.",
   float64(res.NsPerOp())/float64(N))
}
```

Si vous exécutez ce nouveau code, vous constaterez peut-être que les résultats du benchmark sont proches de ceux de la liste unique. Ce n'est pas surprenant: le processeur ne fait qu'attendre le nœud suivant, et l'attente de deux nœuds n'est pas beaucoup plus coûteuse. C'est pourquoi, lorsque vous programmez, vous devez limiter autant que possible les accès à la mémoire. Utilisez autant que possible des tableaux simples plutôt que des listes chaînées ou des structures arborescentes basées sur des nœuds. Nous aimerions travailler avec des structures de données de taille arbitraire, afin de pouvoir solliciter l'accès à la mémoire en dehors de la mémoire cache. L'algorithme de Sattolo est une variante du célèbre random shuffle qui génère une permutation cyclique aléatoire d'un tableau ou d'une liste. L'algorithme de Sattolo garantit que les données sont permutées en utilisant un seul cycle. En d'autres termes, en partant d'un élément d'une liste de taille n, nous constatons que cet élément est déplacé vers une autre position, qui est elle-même déplacée vers une autre position, et ainsi de suite, jusqu'à ce qu'après `n` déplacements, nous nous retrouvions à notre position initiale. Pour appliquer l'algorithme de Sattolo, étant

donné un tableau ou une liste d'éléments, nous commençons par un indice `i` compris entre 0 et `n-1`, où n est la longueur du tableau. Pour chaque indice `i`, nous choisissons un indice aléatoire `j` tel que `i < j < n`. Nous échangeons les éléments aux indices `i` et `j`. Par exemple, supposons que nous ayons un tableau [0, 1, 2, 3, 4]. L'algorithme peut produire une permutation cyclique comme [2, 0, 3, 1, 4]. Avec cet algorithme, nous pouvons visiter toutes les valeurs d'un tableau exactement une fois dans un ordre aléatoire. À partir d'un tableau contenant les indices 0 à `n-1` permutés avec l'algorithme de Sattolo, nous chargeons d'abord le premier élément, lisons sa valeur, passons à l'indice correspondant, et ainsi de suite. Après `n` opérations, nous devrions revenir à la position initiale. Comme chaque opération implique un chargement de la mémoire, elle est limitée par la latence de la mémoire. Nous pouvons essayer d'aller plus vite grâce au *parallélisme au niveau de la mémoire*: nous pouvons choisir k positions réparties dans le cycle et nous déplacer à partir de ces k positions initiales n/k fois dans le cycle. Comme les ordinateurs peuvent charger de nombreuses valeurs en parallèle, cet algorithme devrait être plus rapide pour des valeurs plus grandes de k. Cependant, au fur et à mesure que k augmente, les gains peuvent être de moins en moins importants car les systèmes ont un parallélisme et une bande passante limités au niveau de la mémoire. Le programme suivant met en œuvre cette idée.

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

func makeCycle(length int) ([]uint64, []uint64) {
    array := make([]uint64, length)
    index := make([]uint64, length)
    for i := 0; i < length; i++ {
```

```
      array[i] = uint64(i)
    }

    for i := 0; i+1 < length; i++ {
      swapIdx := rand.Intn(length-i-1) + i + 1
      array[i], array[swapIdx] = array[swapIdx], array[i]
    }

    total := 0
    cur := uint64(0)
    for cur != 0 {
      index[total] = cur
      total++
      cur = array[cur]
    }
    return array, index
}

func setupPointers(index []uint64, length,
  mlp int) []uint64 {
    sp := make([]uint64, mlp)
    sp[0] = 0

    totalInc := 0
    for m := 1; m < mlp; m++ {
      totalInc += length / mlp
      sp[m] = index[totalInc]
    }
    return sp
}

func runBench(array []uint64,
 index []uint64, mlp int) time.Duration {
    length := len(array)
```

```
    sp := setupPointers(index, length, mlp)
    hits := length / mlp
    before := time.Now()
    for i := 0; i < hits; i++ {
      for m := 0; m < mlp; m++ {
          sp[m] = array[sp[m]]
      }
    }
    after := time.Now()
    return after.Sub(before)
}

func main() {
    const length = 100000000
    array, index := makeCycle(length)
    fmt.Println("Length:", length*8/1024/1024, "MB")
    base := runBench(array, index, 1)
    fmt.Println("Lanes:", 1, "Time:", base)

    for mlp := 2; mlp <= 40; mlp++ {
      t := runBench(array, index, mlp)
      fmt.Println("Lanes:", mlp,
       "Speedup:", fmt.Sprintf("%.1f",
      float64(base)/float64(t)))
    }
}
```

La fonction makeCycle crée un cycle d'une longueur spécifiée à partir de l'élément 0. Elle initialise deux tranches: array et index, toutes deux de type []uint64. La tranche de tableau représente les éléments du cycle. La tranche d'index stocke les indices des éléments du cycle, ce qui permet d'accéder plus facilement à une position dans le cycle. La fonction exécute les étapes suivantes. Elle initialise le tableau avec des valeurs comprises entre 0 et 1. Elle applique l'algorithme de mélange de Sattolo au tableau

pour créer une permutation aléatoire. La fonction renvoie le tableau et l'index. La fonction setupPointers: la fonction calcule la valeur d'incrémentation (totalInc) en fonction de la longueur et du nombre de voies (mlp). Elle attribue les indices de index à sp sur la base des incréments calculés. La fonction runBench évalue le temps d'exécution pour un nombre donné de voies (mlp). Elle initialise une tranche sp à l'aide de setupPointers. La fonction itère à travers les pointeurs dans sp et les met à jour en suivant les indices dans `array`. Elle mesure le temps d'exécution et le renvoie sous la forme d'une instance time.Duration. La fonction principale calcule d'abord le temps d'exécution pour une voie, puis indique les gains obtenus lors de l'utilisation de plusieurs voies. Dans l'ensemble, ce code génère un cycle d'une longueur spécifiée, met en place des pointeurs et évalue le temps d'exécution pour différents nombres de couloirs. L'objectif principal semble être d'explorer la parallélisation en utilisant plusieurs voies. La fonction runBench simule l'exécution parallèle en mettant à jour les pointeurs simultanément. L'accélération est calculée en comparant le temps d'exécution pour différents nombres de voies. Plus l'accélération est importante, plus l'exécution parallèle au niveau de la mémoire est efficace. Le principe général est qu'il est souvent possible d'améliorer les performances d'un système confronté à des temps de latence élevés en brisant les dépendances entre les données. Au lieu de placer toutes vos données dans une longue chaîne, essayez de ne pas avoir de chaîne du tout ou, si vous devez avoir des chaînes, utilisez plusieurs chaînes plus petites.

Superscalarité et dépendance des données

La plupart des processeurs actuels sont superscalaires (par opposition à *scalaires*), ce qui signifie qu'ils peuvent exécuter et retirer plusieurs instructions par cycle de CPU. En d'autres termes, même si vous disposez d'un seul cœur de processeur, le parallélisme est important. Certains processeurs peuvent retirer 8 instructions par cycle ou plus. Toutes les routines de code ne bénéficient pas de la même manière de l'exécution superscalaire. Plusieurs facteurs peuvent limiter vos processeurs à quelques

instructions par cycle. Le fait de devoir attendre les accès à la mémoire est l'un de ces facteurs. Un autre facteur courant est la dépendance des données: lorsque l'instruction suivante dépend du résultat d'une instruction précédente, il se peut qu'elle doive attendre avant de commencer à s'exécuter. Pour illustrer ce phénomène, considérons les fonctions qui calculent les différences successives entre les éléments d'un tableau (par exemple, étant donné 5,7,6, vous pourriez obtenir la valeur initiale 5 suivie de 2 et -1), et l'opération inverse qui additionne toutes les différences pour retrouver la valeur originale. Vous pouvez implémenter ces fonctions comme suit:

```
func successiveDifferences(arr []int) {
    base := arr[0]
    for i := 1; i < len(arr); i++ {
      base, arr[i] = arr[i], arr[i]-base
    }
}

func prefixSum(arr []int) {
    for i := 1; i < len(arr); i++ {
      arr[i] = arr[i] + arr[i-1]
    }
}
```

En supposant que le compilateur n'optimise pas ces fonctions d'une manière non triviale (par exemple, en utilisant des instructions SIMD), nous pouvons raisonner relativement simplement sur les performances. Pour les différences successives, nous avons besoin d'environ une soustraction par élément du tableau. Pour la somme préfixe, il faut environ une addition par élément du tableau. Cela semble assez similaire à première vue. Cependant, la dépendance des données est différente. Pour calculer la différence entre deux valeurs du tableau, il n'est pas nécessaire d'avoir calculé les différences précédentes. Cependant, le préfixe somme, tel que nous l'avons implémenté, exige que nous ayons calculé toutes les sommes précédentes avant de pouvoir calculer la suivante. Ecrivons un

petit programme d'évaluation pour tester la différence de performance:

```
package main

import (
    "fmt"
    "math/rand"
    "testing"
)

func successiveDifferences(arr []int) {
    base := arr[0]
    for i := 1; i < len(arr); i++ {
      base, arr[i] = arr[i], arr[i]-base
    }
}

func prefixSum(arr []int) {
    for i := 1; i < len(arr); i++ {
      arr[i] = arr[i] + arr[i-1]
    }
}

var array []int

func BenchmarkPrefixSum(b *testing.B) {
    for n := 0; n < b.N; n++ {
      prefixSum(array)
    }
}

func BenchmarkSuccessiveDifferences(b *testing.B) {
    for n := 0; n < b.N; n++ {
      successiveDifferences(array)
    }
```

```
}

func main() {
    array = make([]int, 100)
    for i := range array {
      array[i] = rand.Int()
    }
    res2 :=
  testing.Benchmark(BenchmarkSuccessiveDifferences)
    fmt.Println("BenchmarkSuccessiveDifferences", res2)
    res1 := testing.Benchmark(BenchmarkPrefixSum)
    fmt.Println("BenchmarkPrefixSum", res1)

}
```

Votre résultat variera en fonction de votre système. Cependant, il ne faut pas s'étonner si la somme des préfixes prend plus de temps. Sur un système Apple, nous obtenons les résultats suivants:

```
BenchmarkSuccessiveDifferences 39742334     30.04 ns/op
BenchmarkPrefixSum  8307944      142.8 ns/op
```

La somme des préfixes peut être plusieurs fois plus lente, même s'il semble à première vue qu'elle devrait utiliser un nombre comparable d'instructions.

Prenons un autre exemple. Nous pouvons calculer le plus grand commun diviseur entre deux entiers en utilisant l'algorithme d'Euclide. Une implémentation raisonnable en Go est la suivante:

```
func gcd(a, b uint) uint {
    for b != 0 {
      quotient := a / b
      a, b = b, a-quotient*b
    }
    return a
```

```
}
```

Il existe une extension de cet algorithme qui calcule non seulement le plus grand diviseur commun, mais aussi les coefficients de Bézout. Autrement dit, étant donné deux entiers `a` et `b`, nous souhaitons trouver des entiers `s` et `t` tels que `a * s + b * t = pgcd(a,b)`: `s` et `t` sont appelés coefficients de Bézout. Une application des coefficients de Bézout est le calcul d'un inverse multiplicatif: supposons que `a` et `b` sont tels que `gcd(a,b)=1` alors nous avons que `s` est l'inverse multiplicatif de `a` modulo `b`: `(a * s) % b = 1`. L'algorithme étendu est assez similaire à l'algorithme euclidien normal:

```
func extended_gcd(a, b uint) (uint, uint, uint) {
    s1, s2 := uint(1), uint(0)
    t1, t2 := uint(0), uint(1)
    for b != 0 {
      quotient := a / b
      a, b = b, a-quotient*b
      s1, s2 = s2, s1-quotient*s2
      t1, t2 = t2, t1-quotient*t2
    }
    return a, s1, t1
}
```

Cette fonction renvoie d'abord le plus grand diviseur commun, puis les deux coefficients de Bézout. À première vue, l'algorithme d'Euclide étendu semble trois fois plus lent que l'algorithme d'Euclide normal. Il semble en effet effectuer trois fois plus de calculs. Nous pouvons écrire un petit benchmark pour tester cette hypothèse:

```
package main

import (
    "fmt"
    "testing"
)
```

```
//go:noinline
func gcd(a, b uint) uint {
    for b != 0 {
      quotient := a / b
      a, b = b, a-quotient*b
    }
    return a
}

//go:noinline
func extended_gcd(a, b uint) (uint, uint, uint) {
    s1, s2 := uint(1), uint(0)
    t1, t2 := uint(0), uint(1)
    for b != 0 {
      quotient := a / b
      a, b = b, a-quotient*b
      s1, s2 = s2, s1-quotient*s2
      t1, t2 = t2, t1-quotient*t2
    }
    return a, s1, t1
}

var count uint

func BenchmarkGCD(b *testing.B) {
    for n := 0; n < b.N; n++ {
      for i := uint(0); i < 10000; i++ {
          count += gcd(i+3111, i+1777)
      }
    }
}

func BenchmarkEGCD(b *testing.B) {
    for n := 0; n < b.N; n++ {
```

```
        for i := uint(0); i < 10000; i++ {
            g, _, _ := extended_gcd(i+3111, i+1777)
            count += g
        }
    }
}

func main() {
    res1 := testing.Benchmark(BenchmarkGCD)
    fmt.Println("GCD", res1)
    res2 := testing.Benchmark(BenchmarkEGCD)
    fmt.Println("EGCD", res2)
}
```

Observez que nous demandons à Go de ne pas intégrer les fonctions: il s'agit de fonctions relativement simples et l'intégration pourrait déclencher des optimisations qui compliqueraient notre analyse. Si vous exécutez ce test, vous constaterez probablement que l'algorithme d'Euclide étendu est plus lent, mais que la différence est minime. Lorsque nous avons exécuté ce test, nous avons constaté que l'algorithme euclidien étendu était environ 35 % plus lent. Il s'agit d'une différence minime par rapport à l'attente d'un algorithme trois fois plus lent.

```
GCD   12838      91481 ns/op
EGCD   8866      124004 ns/op
```

En général, il ne faut pas se fier à une analyse hâtive. Ce n'est pas parce que deux fonctions semblent effectuer une quantité de travail similaire qu'elles ont les mêmes performances. De même, une fonction qui semble effectuer plus de travail peut ne pas être beaucoup plus lente. Plusieurs facteurs doivent être pris en compte, notamment les dépendances des données.

Prédiction de branche

En partie parce que les processeurs sont multiscalaires, ils ont été conçus pour s'exécuter de manière spéculative: lorsqu'il est confronté à une branche, le processeur essaie de deviner la direction qui sera prise et commence le calcul de manière optimiste. Lorsque le processeur fait la bonne prédiction, il améliore généralement les performances, parfois de manière importante. Toutefois, lorsque le processeur n'est pas en mesure de prédire avec précision l'embranchement, la prédiction de l'embranchement peut devenir un élément négatif net. En effet, lorsque la branche est mal prédite, le processeur peut avoir à recommencer le calcul à partir du point où il a fait la mauvaise prédiction, un processus coûteux qui peut gaspiller plusieurs cycles de l'unité centrale. Pour illustrer notre propos, considérons tout d'abord une fonction qui copie le contenu d'une tranche dans une autre tranche de même taille:

```
func Copy(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      dest[i] = v
    }
}
```

Une fonction plus sophistiquée peut ne copier que les éléments impairs:

```
func CopyOdd(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      if v&1 == 1 {
          dest[i] = v
      }
    }
```

```
}
```

Nous pouvons essayer de copier un tableau qui contient des entiers aléatoires (pairs et impairs), uniquement des entiers impairs ou uniquement des entiers pairs. Le programme suivant l'illustre:

```
package main

import (
    "fmt"
    "math/rand"
    "testing"
)

func Copy(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      dest[i] = v
    }
}

func CopyOdd(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      if v&1 == 1 {
          dest[i] = v
      }
    }
}

var array []uint
```

```
var dest []uint

func BenchmarkCopyOdd(b *testing.B) {
    for n := 0; n < b.N; n++ {
      CopyOdd(dest, array)
    }
}

func BenchmarkCopy(b *testing.B) {
    for n := 0; n < b.N; n++ {
      Copy(dest, array)
    }
}

func main() {
    array = make([]uint, 10000)
    dest = make([]uint, len(array))

    for i := range array {
      array[i] = uint(rand.Uint32())
    }
    res0 := testing.Benchmark(BenchmarkCopy)
    fmt.Println("BenchmarkCopy (random)", res0)
    res1 := testing.Benchmark(BenchmarkCopyOdd)
    fmt.Println("BenchmarkCopyOdd (random)", res1)
    for i := range array {
      array[i] = uint(rand.Uint32()) | 1
    }
    res2 := testing.Benchmark(BenchmarkCopyOdd)
    fmt.Println("BenchmarkCopyOdd (odd data)", res2)
    for i := range array {
      array[i] = uint(rand.Uint32()) &^ 1
    }
    res3 := testing.Benchmark(BenchmarkCopyOdd)
```

```
    fmt.Println("BenchmarkCopyOdd (even data)", res3)
}
```

Sur un système Apple, nous avons obtenu les résultats suivants:

```
BenchmarkCopy (random)    414158      2936 ns/op
BenchmarkCopyOdd (random)  55408        19518 ns/op
BenchmarkCopyOdd (odd data)   402670       2975 ns/op
BenchmarkCopyOdd (even data)   402738      2896 ns/op
```

Les trois derniers temps impliquent la même fonction, seules les données d'entrée diffèrent. Nous constatons que tous les temps sont similaires dans ce cas, sauf pour le benchmark qui copie des données aléatoires: il est plusieurs fois plus lent dans nos tests. Le temps d'exécution beaucoup plus long est dû à la présence d'une branche imprévisible dans notre boucle interne. Observez que la même fonction, soumise au même volume de données, peut avoir des caractéristiques de performance très différentes, même si la complexité de calcul de la fonction ne change pas: dans tous les cas, nous avons une complexité temporelle linéaire. Si nous nous attendons à ce que nos données conduisent à une mauvaise prédiction des branches, nous pouvons réduire le nombre de branches dans le code. Le code résultant peut être presque sans branche ou sans branche. Par exemple, nous pouvons utiliser une expression arithmétique et logique pour remplacer une copie de condition:

```
func CopyOddBranchless(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      dest[i] ^= uint(-(v & 1)) & (v ^ dest[i])
    }
}
```

Passons en revue l'expression compliquée:

— `v & 1`: Cette opération vérifie si le bit de poids faible de v est

défini (c'est-à-dire si v est impair).
— `-(v & 1)`: Cette opération annule le résultat de l'opération précédente. Si v est impair, il devient -1 ; sinon, il devient 0. Cependant, `-1` en tant qu'entier non signé devient la valeur maximale, celle dont tous les bits sont à 1.
— `v ^ dest[i]`: Cette fonction effectue un XOR entre la valeur de v et l'élément correspondant de la tranche dest.
— `uint(-(v & 1)) & (v ^ dest[i])`: Si v est impair, il retourne le XOR de v avec dest[i] ; sinon, il retourne 0.
— Enfin, `dest[i] ^= uint(-(v & 1)) & (v ^ dest[i])` laisse `dest[i]` inchangé si v est pair, sinon il le remplace par `v` en utilisant le fait que `dest[i] ^ (v ^ dest[i]) == v`.

Nous pouvons utiliser cette fonction à bon escient dans un benchmark:

```
package main

import (
    "fmt"
    "math/rand"
    "testing"
)

func CopyOdd(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      if v&1 == 1 {
          dest[i] = v
      }
    }
}

func CopyOddBranchless(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
```

```
        panic("dest is too small")
    }
    for i, v := range arr {
        dest[i] ^= uint(-(v & 1)) & (v ^ dest[i])
    }
}

var array []uint
var dest []uint

func BenchmarkCopyOdd(b *testing.B) {
    for n := 0; n < b.N; n++ {
        CopyOdd(dest, array)
    }
}

func BenchmarkCopyOddBranchless(b *testing.B) {
    for n := 0; n < b.N; n++ {
        CopyOddBranchless(dest, array)
    }
}
func main() {
    array = make([]uint, 10000)
    dest = make([]uint, len(array))
    for i := range array {
        array[i] = uint(rand.Uint32())
    }
    res1 := testing.Benchmark(BenchmarkCopyOdd)
    fmt.Println("BenchmarkCopyOdd (random)", res1)
    res2 := testing.Benchmark(BenchmarkCopyOddBranchless)
    fmt.Println("BenchmarkCopyOddBranchless (random)", res2)
}
```

Sur un système Apple, nous avons obtenu:

```
BenchmarkCopyOdd (random)  60782       19254 ns/op
BenchmarkCopyOddBranchless (random)   166863      7124 ns/op
```

Dans ce test, l'approche sans branche est beaucoup plus rapide. Il convient de souligner que le code sans branche n'est pas toujours plus rapide. En fait, nous observons que dans les résultats globaux de nos tests, la fonction sans branche est significativement plus lente que l'originale lorsque les résultats sont prévisibles (par exemple, 2896 ns/op contre 7124 ns/op). Dans un logiciel réel, vous devriez essayer de reconnaître les branches mal prédites et agir dans ces cas pour voir si une approche sans branche pourrait être plus rapide. Heureusement, la plupart des branches sont bien prédites en pratique dans la plupart des projets.

Exercices du chapitre 6

Question 1

Ecrivez un programme qui alloue une tranche contenant `b` octets, pour `b` allant de 10,000,000 à 10,000,100. Votre programme doit estimer l'utilisation de la mémoire d'une tranche de `b` octets nouvellement allouée.

Question 2

Modifiez votre programme de la question précédente pour essayer de déterminer la taille d'une page mémoire en utilisant `runtime.MemStats.HeapInuse`.

Question 3

Ecrivez un programme qui calcule la longueur de k listes chaînées différentes pour un entier arbitraire $k > 0$.

Chapitre 7

En programmation, une structure de données est une manière spécifique d'organiser et de stocker des données dans un ordinateur afin d'y accéder et de les utiliser efficacement.

En menuiserie ou en métallurgie, un gabarit maintient une pièce et guide les outils qui travaillent dessus. Il permet de produire des résultats cohérents. Le gabarit le plus simple est probablement la règle droite. Par exemple, une barre métallique droite peut vous aider à couper du bois en suivant une ligne droite. Un autre gabarit simple est la boîte à onglets, qui nous aide à couper du bois à des angles fixes.

Les structures de données peuvent être considérées comme des gabarits pour les programmeurs. Tout comme un gabarit guide l'outil pour réaliser des coupes ou des formes précises, les structures de données en programmation fournissent un cadre pour l'organisation et l'accès aux données. Tout comme un gabarit guide l'outil, les structures de données indiquent comment les données doivent être stockées, accédées et manipulées. Les structures de données nous aident à garantir que les opérations sur les données sont effectuées de manière prévisible et efficace.

Les cours d'informatique classiques présentent souvent un large éventail de structures de données: listes chaînées, arbres rouge-noir, etc. Pourtant, dans la pratique, l'écrasante majorité des problèmes de programmation peuvent être résolus en utilisant à peine plus que des tableaux, le type composite standard (`struct`) et une table de hachage occasionnelle. L'analogie avec le gabarit de menuisier tient toujours: les gabarits de

menuisier les plus populaires sont si simples que nous oublions souvent qu'il s'agit d'un véritable outil.

Tableaux

Certaines structures de données sont particulièrement utiles en raison de leur simplicité et de leur efficacité. La structure de données la plus simple est le tableau. Les tableaux sont des collections d'éléments de taille fixe. Par exemple, en Go, vous pouvez déclarer un tableau de 5 entiers comme suit:

```
var myArray [5]int
```

En règle générale, les tableaux sont stockés de manière contiguë dans la mémoire. Les tableaux sont donc idéaux si vous devez parcourir tous les éléments aussi rapidement que possible: vous accédez à la mémoire d'une manière prévisible avec peu de frais généraux. Pour accéder à un élément donné d'un tableau, il suffit généralement de prendre l'index et de le convertir en adresse mémoire. Cela peut ne nécessiter que quelques opérations peu coûteuses.

D'une manière générale, les tableaux génèrent un code rapide, en partie parce que les compilateurs trouvent qu'il est facile d'optimiser les fonctions. Par exemple, en Go, les accès aux tableaux sont généralement soumis à un vérificateur de limites: Go empêchera l'accès à un tableau à l'index 10 si la taille du tableau est de 5. Cependant, le compilateur peut souvent optimiser de telles vérifications. Par exemple, considérons la fonction suivante dans laquelle nous additionnons les cinq éléments d'un tableau. Elle devrait être compilée en un code efficace ne comportant pratiquement aucun surcoût (par exemple, la vérification des limites).

```
func Sum(x [5]int) int {
      sum := 0
      for i := 0; i < 5; i++ {
            sum += x[i]
      }
```

```
        return sum
}
```

Un tableau peut contenir différentes valeurs (y compris des tableaux !). Il est même possible de stocker des valeurs booléennes (vrai/faux) dans un tableau, bien que dans ce cas, il faille au moins un octet par élément. Dans ce cas, vous pouvez remplacer un tableau par un ensemble de bits. Vous pouvez implémenter un ensemble de bits sur un tableau comme suit (en utilisant 128 bits comme exemple).

```
type BitSet struct {
        bytes [16]byte
}


func (bs *BitSet) Set(index int) {
    bs.bytes[index/8] |= 1 << uint(7-(index%8))
}

func (bs *BitSet) Clear(index int) {
    bs.bytes[index/8] &^= 1 << uint(7-(index%8))
}

func (bs *BitSet) Get(index int) bool {
    return bs.bytes[index/8]&(1<<uint(7-(index%8))) != 0
}
```

Une extension naturelle du tableau est le tableau multidimensionnel. Même si le langage Go supporte nativement les tableaux multidimensionnels, nous pouvons toujours les implémenter à partir de zéro en utilisant un tableau standard. Typiquement, nous utilisons une implémentation majeure en ligne. Par exemple, nous pouvons implémenter une simple matrice 3x3 en utilisant un tableau à 9 éléments. Les trois premiers éléments du tableau représentent la première ligne, les trois éléments suivants représentent la deuxième ligne, et ainsi de suite.

```
type Matrix struct {
	data [9]float64
}

func (m *Matrix) Set(i, j int, val float64) {
	if i < 0 || i > 2 || j < 0 || j > 2 {
		panic("Index out of bounds for a 3x3 matrix")
	}
	m.data[i*3+j] = val
}

func (m *Matrix) Get(i, j int) float64 {
	if i < 0 || i > 2 || j < 0 || j > 2 {
		panic("Index out of bounds for a 3x3 matrix")
	}
	return m.data[i*3+j]
}

func (m *Matrix) String() string {
	var str string
	for i := 0; i < 3; i++ {
		for j := 0; j < 3; j++ {
			str += fmt.Sprintf("%6.2f ", m.Get(i, j))
		}
		str += "\n"
	}
	return str
}
```

Go, comme beaucoup de langages de programmation, vous permet de créer des structures de données composites (`struct`) efficaces. En particulier, vous pouvez placer des tableaux à l'intérieur de ces structures de données. Supposons, par exemple, que nous voulions représenter des points dans l'espace, nous pourrions le faire comme suit:

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

Nous pouvons placer ces points dans un tableau de dix éléments:

```
var list [10]Point
```

Ce modèle est souvent appelé *tableau de structures*. Nous pouvons également représenter les mêmes données à l'aide d'une *structure de tableaux*:

```
type Points struct {
      x [10]float64
      y [10]float64
}
```

Bien que les tableaux de structures et les structures de tableaux puissent contenir les mêmes informations, ils présentent des caractéristiques de performance différentes. Par exemple, si vous avez besoin d'additionner les valeurs x, la structure de tableaux peut conduire à un code plus rapide.

Tableaux dynamiques et tranches

Le tableau dynamique est une structure de données légèrement plus sophistiquée. Un tableau dynamique est un tableau dont la taille peut augmenter ou diminuer. En Go, ils sont implémentés sous forme de tranches. Ils constituent l'une des structures de données les plus fondamentales de Go.

```
var mySlice []int
mySlice = append(mySlice, 1, 2, 3)
```

On accède aux tableaux par des index entiers. Par exemple, vous accédez à un tableau de taille 3 avec les indices 0, 1, 2. Chaque indice vous donne accès à une valeur. Le tableau constitue donc une structure de données clé-valeur où les clés sont les index.

Comme la taille de la tranche est dynamique, Go fournit une fonction pour l'interroger: `len` (length en anglais). Par exemple, `len(mySlice)` peut être 3.

En Go, plusieurs tranches peuvent partager le même tableau sous-jacent. Nous utilisons l'opérateur `[begin:end]` pour créer une nouvelle tranche à partir d'une tranche existante. Il est important de noter que cette opération ne copie pas les données sous-jacentes.

Dans l'exemple suivant, la deuxième tranche sera de taille 1 et contiendra le deuxième élément de la première tranche.

```
var mySlice []int
mySlice = append(mySlice, 1, 2, 3)
secondSlice := mySlice[1:2]
```

Go fournit quelques notations abrégées. Au lieu d'écrire `mySlice[2:len(mySlice)]`, vous pouvez écrire `mySlice[2:]` et au lieu d'écrire `mySlice[0:1]`, vous pouvez écrire `mySlice[:1]`. Pour créer une copie de la tranche, il suffit d'écrire `mySlice[:]`.

Les tranches partagent de nombreuses caractéristiques avec les tableaux. Par exemple, nous pouvons écrire des fonctions qui prennent des tranches en paramètre et compilent dans un code aussi efficace, comme dans cet exemple où nous additionnons les 5 premiers éléments de la tranche:

```
func Sum(x []int) int {
      sum := 0
  if len(x) < 5 {
    return -1
  }
      for i := 0; i < 5; i++ {
            sum += x[i]
      }
      return sum
}
```

En fait, les tableaux dynamiques, c'est-à-dire les tranches, sont géné-

ralement mis en œuvre sous la forme d'une fine enveloppe au-dessus d'un tableau. La tranche pointe sur une région à l'intérieur d'un tableau. Lorsque nous réduisons ou étendons la région pointée par la tranche, le tableau sous-jacent peut rester le même.

Généralement, la capacité de stockage du tableau sous-jacent est supérieure à la taille de la tranche. Vous pouvez demander la taille du tableau sous-jacent avec la fonction `cap` (capacité). Il peut sembler inutile d'avoir plus de capacité que nécessaire. Cependant, considérons le cas où nous ajoutons régulièrement des éléments à une tranche donnée, et où la taille du tableau sous-jacent est réglée pour correspondre exactement à la taille de la tranche. Dans ce cas, chaque ajout (append) à la tranche peut nécessiter l'allocation d'un nouveau tableau et la copie des éléments. En ne comptant que les copies d'éléments, nous obtenons que nous avons besoin de 1 + 2 + 3 + 4 + ... + n - 1 = (n-1)n/2$ copies pour remplir une tranche avec n valeurs. Au lieu de cela, nous augmentons généralement la taille du tableau sous-jacent par étapes exponentielles. Par exemple, lorsque la capacité devient insuffisante, nous pouvons allouer de la mémoire à la puissance deux suivante: nous allouons 16 éléments de capacité si nous avons besoin de 7 éléments, nous allouons 1024 éléments si nous avons besoin de 600 éléments, et ainsi de suite. Nous pouvons vérifier qu'avec une telle stratégie, si nous devons ajouter de manière répétée un élément à une tranche, nous n'aurons jamais besoin de plus de $2n$ copies pour créer une tranche avec n valeurs: $2n$ est beaucoup plus petit que $(n-1)n/2$ lorsque n est grand. Par défaut, Go ne récupère pas la capacité excédentaire en réduisant la taille de la tranche (par exemple, `s = s[:newlength]`). Cependant, vous pouvez utiliser la syntaxe étendue `s[low:high:max]` pour demander à Go de réduire la taille du tableau sous-jacent à `max-low`. Par exemple, `s = s[::len(s)]` créerait une copie de la tranche tout en ajustant la capacité sous-jacente à `len(s)`.

Le programme suivant ajoute des éléments à une tranche et affiche la capacité (taille du tableau sous-jacent) et la taille de la tranche à chaque incrément. À la fin du programme, nous montrons que nous pouvons réduire la tranche avec ou sans ajustement de la capacité.

```go
package main

import "fmt"

func main() {
	var mySlice []int
	for i := 0; i < 100; i++ {
		mySlice = append(mySlice, 1)
		fmt.Println(cap(mySlice), " : ", len(mySlice))
	}
	mySlice = mySlice[:10]
	fmt.Println(cap(mySlice), " : ", len(mySlice))
	mySlice = mySlice[:10:10]
	fmt.Println(cap(mySlice), " : ", len(mySlice))
}
```

Les résultats suivants sont possibles :

```
1  :  1
2  :  2
4  :  3
4  :  4
8  :  5
8  :  6
8  :  7
8  :  8
16  :  9
...
128  :  99
128  :  100
128  :  10
10  :  10
9  :  2
```

Il est parfois pratique de stocker les données dans un ordre trié à l'intérieur

d'un tableau. Nous pouvons alors utiliser le tableau comme une structure de données efficace, même si le tableau est de grande taille. Lors de la recherche d'une valeur, nous pouvons utiliser une recherche binaire. L'algorithme est relativement simple: étant donné la valeur recherchée, nous la comparons d'abord à la valeur du milieu, c'est-à-dire la valeur médiane. Si la valeur recherchée est supérieure à la valeur médiane, nous cherchons dans la dernière moitié du tableau, sinon nous cherchons dans la première moitié du tableau. Nous répétons la division de manière récursive jusqu'à ce que nous trouvions la valeur recherchée ou que nous déterminions qu'elle est introuvable. Il faut environ $log_2(N)$ itérations pour compléter la recherche sur un tableau de taille N, et la fonction logarithmique croît très lentement (par exemple, $log_2(10^6) \approx 20$).

```
func Search(n int, array []int) (int, bool) {
      low, high := 0, len(array)-1
      for low <= high {
            mid := (low + high) / 2
            if array[mid] == n {
                  return mid, true
            } else if array[mid] < n {
                  low = mid + 1
            } else {
                  high = mid - 1
            }
      }
      return -1, false
}
```

Parfois, nous voulons seulement pouvoir accéder à la plus petite ou à la plus grande valeur. Supposons par exemple que vous ayez un flux de demandes à traiter et que vous souhaitiez toujours sélectionner la demande qui a attendu le plus longtemps. Vous pourriez balayer le tableau de manière répétée ou le trier de manière répétée. Cependant, si vous ajoutez et retirez constamment des données du tableau, cela pourrait devenir coûteux. Au lieu de cela, nous pouvons ordonner les valeurs du

tableau en fonction d'un *troupeau binaire*. La racine du tas est stockée à l'index 0, et c'est soit la plus petite, soit la plus grande valeur, selon le type de tas dont vous avez besoin. Nous avons donc soit un tas max, soit un tas min. À l'intérieur du tableau, les valeurs sont ensuite organisées sous forme d'arbre. Pour tout nœud à l'index i, son enfant de gauche est à $2 * i + 1$ et son enfant de droite est à $2 * i + 2$. À la fin du tableau, certains nœuds peuvent n'avoir qu'un enfant à gauche, tandis que d'autres peuvent n'avoir aucun enfant. Si nous avons un tas maximal, chaque nœud ne doit pas être plus petit que ses enfants. Dans le cas d'un tas minimal, chaque nœud ne doit pas être plus grand que ses enfants. Les opérations telles que l'insertion et la suppression sont effectuées en manipulant les éléments du tableau tout en conservant la propriété de tas. Pour insérer une nouvelle valeur, vous placez le nouvel élément à l'extrémité du tableau. Il devient alors l'enfant d'un nœud existant si le tableau n'est pas vide: si vous le placez à l'index i, vous pouvez calculer l'index du parent comme l'entier $(i - 1)/2$. Vous le permutez ensuite avec son parent pour préserver la propriété du tas (par exemple, si nous avons un tas maximal, chaque nœud ne doit pas être plus petit que ses enfants). Il peut alors être nécessaire de vérifier le parent une fois de plus et ainsi de suite jusqu'à ce que vous atteigniez le sommet du tas. La fonction suivante modifie une tranche selon cet algorithme en utilisant la propriété max-heap.

```
func Insert(heap *[]int, value int) {
      *heap = append(*heap, value)
      heapSize := len(*heap)

      i := heapSize - 1

      for i > 0 {
            parent := (i - 1) / 2

            if (*heap)[i] > (*heap)[parent] {
                  (*heap)[i], (*heap)[parent]
          = (*heap)[parent], (*heap)[i]
```

```
                i = parent
            } else {
                break
            }
        }
}
```

Lorsque l'on supprime l'élément supérieur, on le remplace par le dernier élément, puis on propage cet élément vers le bas en l'échangeant avec l'enfant le plus grand (ou le plus petit) s'il ne respecte pas la propriété du tas. Nous pouvons l'implémenter de la manière suivante pour un tas maximal:

```
func RemoveTop(heap *[]int) int {
    if len(*heap) == 0 {
        return 0
    }

    top := (*heap)[0]

    heapSize := len(*heap)
    (*heap)[0] = (*heap)[heapSize-1]
    *heap = (*heap)[:heapSize-1]

    i := 0
    for {
        leftChild := 2*i + 1
        rightChild := 2*i + 2
        largest := i

        if leftChild < len(*heap)
    && (*heap)[leftChild] >
            (*heap)[largest] {
            largest = leftChild
        }
```

```
            if rightChild < len(*heap)
      && (*heap)[rightChild] >
                  (*heap)[largest] {
                  largest = rightChild
            }

            if largest != i {
                  (*heap)[i],
                        (*heap)[largest] = (*heap)[largest],
                        (*heap)[i]
                  i = largest
            } else {
                  break
            }
      }

      return top
}
```

Ainsi, avec seulement deux fonctions relativement simples, nous pouvons maintenir un tas binaire. Ces fonctions nécessitent au plus $\lceil \log_2 N \rceil$ comparaisons où N est le nombre d'éléments dans le tableau. Il est intéressant de noter qu'un tas binaire fournit un algorithme sensé pour trier un tableau, parfois appelé *heapsort*: insérer tous les éléments dans un tas binaire, puis retirer de manière répétée la valeur supérieure. Le résultat est un algorithme $O(N \log N)$.

Tout comme nous pouvons implémenter un ensemble de bits sur un tableau, nous pouvons implémenter un ensemble de bits dynamique sur un tableau dynamique. L'approche est sensiblement la même, mais le tableau est remplacé par une tranche. Il est important de vérifier s'il est nécessaire d'étendre l'ensemble de bits. Lorsque le besoin s'en fait sentir, nous agrandissons le jeu de bits en allouant un tableau plus grand et en copiant nos données existantes. Observez que nous avons choisi

d'augmenter la tranche de deux fois la quantité nécessaire: cela permet d'éviter les cas dégénérés où l'utilisateur pourrait accéder aux bits un par un (0, 1, ...), ce qui entraînerait des copies et des allocations répétées.

```
type BitSet struct {
	bytes []byte
}

func (bs *BitSet) ensureCapacity(index int) {
	requiredBytes := (index + 1 + 7) / 8
	if len(bs.bytes) < requiredBytes {
		newBytes := make([]byte, requiredBytes*2)
		copy(newBytes, bs.bytes)
		bs.bytes = newBytes
	}
}

func (bs *BitSet) Set(index int) {
	bs.ensureCapacity(index)
	bs.bytes[index/8] |= 1 << uint(7-(index%8))
}

func (bs *BitSet) Clear(index int) {
	bs.ensureCapacity(index)
	bs.bytes[index/8] &^= 1 << uint(7-(index%8))
}

func (bs *BitSet) Get(index int) bool {
	bs.ensureCapacity(index)
	return bs.bytes[index/8]&(1<<uint(7-(index%8))) != 0
}
```

Tables de hachage et dictionnaires

Il est fréquent que vous ayez besoin d'une structure de données dans laquelle vous pouvez accéder à des valeurs en utilisant des entiers non séquentiels (par exemple, 10, 1000, 100000) ou d'autres types de valeurs telles qu'une chaîne de caractères. Par exemple, vous souhaitez peut-être établir une correspondance entre les noms et les numéros de téléphone. définit le conception de dictionnaire en informatique. En informatique, une telle correspondance forme un dictionnaire (ou tableau associatif). Chaque clé est unique et associée à une valeur spécifique, permettant un accès rapide aux données via la clé.

L'une des structures de données les plus utiles est la table de hachage. Les tables de hachage s'appuient sur les tableaux, mais au lieu d'utiliser des valeurs entières consécutives comme index, elles peuvent prendre presque n'importe quel type de valeur (par exemple, une chaîne) comme clé.

L'astuce essentielle de la table de hachage consiste à utiliser une fonction de hachage qui associe des clés arbitraires à des valeurs entières. Les valeurs entières sont utilisées comme index à l'intérieur d'un tableau. Le tableau est généralement choisi pour avoir une taille comparable au nombre de clés distinctes. Une table de hachage est donc une généralisation du tableau, mais dans laquelle les accès simples sont remplacés par une fonction de hachage. Une fonction de hachage prend une clé et la convertit en un index du tableau de la table de hachage.

Nous espérons généralement que la fonction de hachage distribue les clés de manière uniforme dans le tableau afin de minimiser les collisions. Lorsque deux clés sont associées au même index dans le tableau, nous devons résoudre le problème. Une possibilité consiste à créer un tableau plus grand. Mais dans la pratique, cela entraînerait une croissance trop rapide des tableaux. Nous devons donc gérer les collisions de manière efficace. Il existe deux grandes stratégies. La première consiste à pouvoir stocker plusieurs paires clé-valeur dans chaque élément du tableau. C'est ce que l'on appelle souvent l'approche du seau. Une autre possibilité consiste à stocker les paires clé-valeur qui entrent en collision ailleurs: par

exemple, vous pouvez les stocker dans le prochain emplacement disponible dans le tableau.

Il existe de nombreuses implémentations de tables de hachage aux propriétés variées. Go fournit un type de dictionnaire intégré qui met en œuvre une table de hachage. Par exemple, pour créer une table de correspondance entre des chaînes de caractères et des entiers, nous pourrions procéder comme suit:

```
myMap := make(map[string]int)
myMap["key"] = 10
```

Go utilise une approche de type seau (*bucket*): chaque élément du tableau peut stocker plusieurs valeurs. Dans la mise en œuvre du dictionnaire de Go, chaque seau contient plusieurs paires clé-valeur. Le nombre exact de paires par seau peut varier, mais il est conçu pour gérer efficacement un petit nombre d'entrées. Par exemple, nous pourrions utiliser ce type de structure de données avec des tranches:

```
type bucket struct {
      keys    []string
      values  []int
}
```

Lorsque deux clés ont le même index, elles sont placées dans le même seau.

Lors de la recherche d'une clé, nous calculons le hachage de la clé, nous utilisons le hachage pour trouver le bon seau, nous vérifions chaque entrée du panier pour trouver la clé correspondante. Tant que les seaux sont suffisamment petits, les performances sont acceptables.

Nous calculons souvent le nombre de clés stockées dans la table de hachage et le divisons par la taille du tableau. Le résultat s'appelle le facteur de charge. Lorsqu'il y a trop de clés par rapport à la taille du tableau sous-jacent, le tableau est agrandi et la table de hachage est reconstruite. Cela permet généralement de réduire la taille moyenne des seaux. De même, nous pouvons réduire la taille du tableau s'il reste trop peu de

clés.

Prenons un exemple complet:

```
package main

import (
	"errors"
	"fmt"
)

type Bucket struct {
	keys   []string
	values []int
}

func (b *Bucket) Add(key string, value int) {
	b.keys = append(b.keys, key)
	b.values = append(b.values, value)
}

func (b *Bucket) Find(key string) (int, error) {
	for i := 0; i < len(b.keys); i++ {
		if key == b.keys[i] {
			return b.values[i], nil
		}
	}
	return 0, errors.New("Not found")
}

type HashTable struct {
	array []Bucket
}

func NewHashTable(size int) *HashTable {
	return &HashTable{
```

```
            make([]Bucket, size),
      }
}

func (ht *HashTable) hash(key string) int {
      hash := 0
      for i := 0; i < len(key); i++ {
            hash += 31 * int(key[i])
      }
      return hash % len(ht.array)
}

func (ht *HashTable) Get(key string) (int, error) {
      return ht.array[ht.hash(key)].Find(key)
}

func (ht *HashTable) Set(key string, value int) error {
      _, e := ht.Get(key)
      if e == nil {
            return errors.New("Key already present")
      }
      ht.array[ht.hash(key)].Add(key, value)
      return nil
}

func main() {
      ht := NewHashTable(10)
      ht.Set("apple", 1)
      ht.Set("banana", 3)
      fmt.Println(ht.Get("apple"))
}
```

Notre exemple est simplifié. Cependant, il illustre les principes de base du fonctionnement des tables de hachage.

L'approche que nous avons décrite, avec les seaux, est relativement simple à mettre en œuvre, mais elle n'offre pas toujours les meilleures performances en raison de la surcharge ayant trait à la gestion des seaux en tant que tableaux dynamiques distincts. À la place, nous pouvons envisager une variante dans laquelle nous avons une valeur-clé par emplacement dans le tableau. Ce modèle est parfois appelé adressage ouvert. Lorsque deux clés se trouvent dans le même emplacement, nous pouvons déplacer l'une des paires clé-valeur vers un autre emplacement disponible ou augmenter la taille du tableau sous-jacent. Au moment de la requête, nous commençons par rechercher la clé-valeur en fonction de la valeur de hachage de la clé. Si une autre valeur-clé est trouvée, nous visitons l'emplacement suivant, jusqu'à ce que nous trouvions la valeur recherchée ou un emplacement vide. Tant que le tableau sous-jacent est suffisamment grand pour qu'une fraction importante des emplacements soit vide, les performances sont acceptables.

Le code suivant illustre l'idée principale de l'adressage ouvert, à l'exception du fait que le tableau sous-jacent n'augmente pas au fur et à mesure que des valeurs sont ajoutées. Pour ajouter cette fonctionnalité, nous devrions suivre le nombre de clés ajoutées et faire croître le tableau si nécessaire. Comme *sentinelle* pour indiquer un emplacement disponible, nous utilisons la clé vide. En pratique, nous devrions ajouter des contrôles supplémentaires pour nous assurer que l'utilisateur n'essaie pas d'ajouter une clé avec une chaîne vide, et pour gérer le scénario de manière appropriée.

```
type Item struct {
      key   string
      value int
}

type HashTable struct {
      items      []Item
      emptyValue Item
}
```

```
func NewHashTable(size int) *HashTable {
	ht := &HashTable{
		items:      make([]Item, size),
		emptyValue: Item{key: "", value: -1},
	}
	for i := 0; i < size; i++ {
		ht.items[i] = ht.emptyValue
	}
	return ht
}

func hash(key string) int {
	hash := 0
	for i := 0; i < len(key); i++ {
		hash += 31 * int(key[i])
	}
	return hash
}

func (ht *HashTable) Put(key string, value int) {
	hashValue := hash(key) % len(ht.items)
	for {
		if ht.items[hashValue] == ht.emptyValue {
			ht.items[hashValue]
		= Item{key: key, value: value}
			return
		}
		if ht.items[hashValue].key == key {
			ht.items[hashValue].value = value
			return
		}
		hashValue = (hashValue + 1) % len(ht.items)
	}
}
```

```
func (ht *HashTable) Get(key string) (int, bool) {
      hashValue := hash(key) % len(ht.items)
      for i := 0; i < len(ht.items); i++ {
            if ht.items[hashValue].key == key {
                  return ht.items[hashValue].value, true
            }
            if ht.items[hashValue] == ht.emptyValue {
                  return -1, false
            }
            hashValue = (hashValue + 1) % len(ht.items)
      }
      return -1, false
}
```

Il existe de nombreuses solutions de rechange qui pourraient être encore plus rapides que l'adressage ouvert dans certains cas, comme le hachage Cuckoo. Dans la pratique, les programmeurs mettent rarement en œuvre leurs propres tables de hachage, mais ils doivent savoir qu'il existe différentes mises en œuvre présentant divers avantages.

Une table de hachage qui ne possède que des clés peut être utilisée pour mettre en œuvre une structure de données ensembliste. En d'autres termes, il n'est pas toujours nécessaire d'avoir des valeurs. Une variante simple qui mérite d'être envisagée consiste à permettre aux clés d'être associées à plusieurs valeurs différentes. Cette variante est parfois appelée *multimap*.

Conclusion

Bien qu'il existe d'innombrables structures de données en informatique, les tableaux et quelques fonctions simples permettent de réaliser beaucoup de choses. Lorsque vous organisez vos données, vous devriez d'abord essayer d'utiliser des tableaux. Dans certains cas, une table de hachage peut être nécessaire si vous avez un ensemble ou une carte. Cela devrait couvrir la

grande majorité de vos structures de données.

Pour illustrer cette idée, prenons l'exemple de JSON. JSON (JavaScript Object Notation) est un format standard d'échange de données textuelles couramment utilisé pour échanger des données en ligne. Il comporte des valeurs primitives (chaînes, nombres, booléens, valeur nulle) et des types composites (tableaux et objets). Les objets sont des dictionnaires de chaînes de caractères vers des valeurs primitives ou vers d'autres types composites (tableaux et objets). Nous représentons les objets sous forme de paires clé-valeur séparées par des virgules entre accolades `{}`, en utilisant les deux points comme séparateur entre la clé et la valeur. Nous utilisons des crochets `[]` pour les tableaux, en séparant les valeurs par des virgules. Prenons l'exemple suivant, qui représente une liste d'employés.

```
{
   "employees":[
      {
         "name":"Richard",
         "salary":56000,
         "function":"CEO"
      },
      {
         "name":"Lisa",
         "salary":55000,
         "function":"accountant"
      }
   ]
}
```

D'un point de vue empirique, JSON s'avère suffisant pour représenter la plupart des données, malgré sa simplicité. Une idée clé est qu'en combinant des tableaux et des dictionnaires, avec quelques types standard pour les nombres et les chaînes de caractères, vous pouvez résoudre la plupart des problèmes.

Dans certains cas, des structures de données spécialisées peuvent offrir

des performances supérieures. Cependant, vous devez garder à l'esprit qu'il est souvent plus facile d'optimiser le code lorsque la structure de données sous-jacente est plus simple.

Exercices du chapitre 7

Question 1

Implémentez à partir de zéro une fonction `Insert` et `RemoveTop` pour un min-heap de chaînes de caractères.

Question 2

Nous savons comment implémenter une recherche binaire. Elle divise les tableaux triés en deux à chaque itération, après une seule comparaison. Nous pouvons généraliser la recherche binaire à une recherche k-aire où, à chaque itération, nous divisons le tableau en k sections. Combien de comparaisons devons-nous effectuer à chaque itération?

Question 3

Ecrivez une fonction en Go qui trouve, étant donné un entier positif `n`, la plus petite puissance de deux qui est au moins aussi grande que `n`.

Chapitre 8

Dans la pratique, les logiciels que nous écrivons fonctionnent sur plusieurs processeurs. Malheureusement, la plupart des choses que nous tenons pour acquises sur un seul processeur deviennent fausses lorsqu'il y a plus d'un processeur. Par exemple, si deux processeurs modifient le même morceau de mémoire, quel est l'état de la mémoire après les modifications? Il est difficile de le savoir en général. Il est possible que la modification d'un processeur écrase toute modification effectuée par l'autre processeur. L'inverse pourrait être vrai: la modification effectuée par l'autre processeur pourrait l'emporter. Il se peut aussi que les deux processeurs tentent la modification et que le résultat soit un état confus qui ne correspond à rien de ce que nous voudrions voir. Nous appelons ces accès une *course aux données*: une situation dans laquelle deux ou plusieurs processeurs d'un programme accèdent simultanément au même emplacement de mémoire, et au moins l'un de ces accès est une opération d'écriture, sans synchronisation appropriée. Les choses se compliquent lorsque vous souhaitez que deux ou plusieurs processeurs modifient de manière significative la même mémoire. Supposons par exemple que vous ayez une variable qui compte le nombre de produits vendus. Il se peut que plusieurs processeurs incrémentent cette variable.

Threads et goroutines

Un thread est la plus petite unité d'exécution au sein d'un processus qui peut être programmé et exécuté indépendamment par le système d'exploi-

tation d'un ordinateur. Il représente une séquence unique d'instructions que le processeur exécute, ce qui permet à un programme d'effectuer plusieurs tâches simultanément au sein d'un même processus. Un thread existe au sein d'une entité plus large appelée processus, qui est essentiellement un programme en cours d'exécution disposant de son propre espace mémoire, de ses propres ressources et de son propre état. Un processus peut contenir un ou plusieurs threads, qui partagent tous la même mémoire et les mêmes ressources (comme les fs ouverts ou les variables globales) allouées à ce processus. Il existe une limite au nombre de threads qu'un programme peut gérer efficacement. Pour permettre encore plus de parallélisme, le langage de programmation Go a son propre concept de thread, appelé goroutine. Bien qu'une goroutine ne soit pas un thread au sens traditionnel du terme, elle s'apparente à des threads conventionnels sous le capot. Le moteur d'exécution de Go utilise un planificateur pour faire correspondre de nombreuses goroutines à un plus petit nombre de threads. Ces threads sont les véritables threads reconnus par le système d'exploitation - des entités au niveau du noyau avec leur propre pile et leur propre contexte d'exécution, comme décrit dans l'informatique générale. Un seul thread dans Go peut exécuter plusieurs goroutines en passant de l'une à l'autre de manière efficace. Cela rend les goroutines beaucoup moins coûteuses que les threads du système d'exploitation: il est possible de créer des milliers, voire des millions de goroutines, alors que la création d'autant de threads épuiserait les ressources du système en raison de leur empreinte mémoire plus importante. D'une certaine manière, Go brouille la distinction entre la simultanéité et le parallélisme. La simultanéité consiste à gérer plusieurs tâches de manière à ce qu'elles puissent progresser indépendamment les unes des autres. Le parallélisme, quant à lui, implique l'exécution simultanée de plusieurs tâches sur plusieurs ressources. Alors que la simultanéité se concentre sur la conception logicielle pour la coordination des tâches et peut fonctionner avec ou sans plusieurs cœurs, le parallélisme s'appuie sur le matériel pour réaliser une véritable exécution simultanée, et les deux peuvent se combiner lorsqu'un système simultané exploite des ressources parallèles pour plus d'efficacité. Pour lancer une goroutine, il suffit de taper le mot-clé 'go' suivi d'une fonction:

```
go func() {
  fmt.Println("Canada")
}()
```

Cela crée une goroutine, mais le runtime Go décide sur quel thread elle s'exécute, partageant potentiellement ce thread avec d'autres goroutines. Malheureusement, un programme composé uniquement de cette goroutine pourrait être décevant:

```
package main

import (
  "fmt"
)

func main() {
  go func() {
    fmt.Println("Canada")
  }()
}
```

Le problème est que la fonction principale peut se terminer avant que la goroutine ne puisse se terminer. En Go, les goroutines s'exécutent simultanément, et la fonction principale (qui est la goroutine principale) n'attend pas automatiquement que les autres goroutines se terminent. Si la goroutine principale se termine, le programme se termine, potentiellement avant que les autres goroutines ne se terminent. Pour s'assurer qu'une goroutine se termine avant le programme, l'approche la plus simple consiste à synchroniser la goroutine principale avec la goroutine créée en utilisant un mécanisme tel qu'un canal ou un WaitGroup. En Go, un canal est une construction intégrée qui permet aux goroutines (fonctions concurrentes) de communiquer entre elles et de synchroniser leur exécution. Un canal a un type et il est créé avec la fonction `make`:

```
ch := make(chan int)
```

Le mot-clé `chan` est le mot-clé pour déclarer un canal. Le type qui suit `chan` (par exemple, `int`) définit le type de données que le canal peut transporter.

Nous utilisons l'opérateur `<-` pour envoyer une valeur dans un canal.

```
ch <- 42
```

Nous utilisons l'opérateur `<-` pour recevoir une valeur d'un canal.

```
value := <-ch
```

Nous utilisons la fonction close pour indiquer que les données ne seront plus envoyées: `close(ch)`. L'envoi à un canal fermé provoque une panique. Le programme suivant imprimerait `Canada`:

```
package main

import "fmt"

func main() {
  ch := make(chan string)

  go func() {
    ch <- "Canada"
  }()

  msg := <-ch
  fmt.Println(msg)
}
```

Le programme suivant illustre comment nous pouvons utiliser un canal pour attendre la fin d'une goroutine:

```
package main
```

```
import (
  "fmt"
)

func main() {
  channel := make(chan bool)

  go func() {
    fmt.Println("Canada")
    channel <- true
  }()

  <-channel
}
```

La goroutine envoie une valeur (true) au canal lorsqu'elle se termine. La fonction principale se bloque à `<-done`, attendant de recevoir du canal, s'assurant de ne pas sortir avant que la goroutine ne se termine. Par défaut, un canal n'est pas mis en mémoire tampon: il ne peut contenir qu'une seule valeur. Ainsi, si vous essayez de lui écrire plus d'une valeur, il se bloquera jusqu'à ce qu'au moins une valeur soit lue.

```
ch := make(chan int, 2)
ch <- 1
ch <- 2
ch <- 3
```

En Go, vous pouvez passer plusieurs canaux à une fonction comme n'importe quel autre argument. Les canaux sont des valeurs de première classe en Go, ce qui signifie qu'ils peuvent être passés en tant que paramètres, renvoyés par des fonctions ou stockés dans des variables. Lorsque vous passez plusieurs canaux à une fonction, il vous suffit de les inclure dans la liste des paramètres de la fonction, en spécifiant leur type. Prenons un exemple où nous accédons à deux URL:

```go
package main

import (
  "fmt"
  "net/http"
  "time"
)

type Response struct {
  url  string
  status string
  err  error
}

func fetchURL(url string, ch chan Response) {
  client := &http.Client{
    Timeout: 10 * time.Second,
  }

  resp, err := client.Get(url)
  if err != nil {
    ch <- Response{url, "", err}
    return
  }
  defer resp.Body.Close()

  ch <- Response{url, resp.Status, nil}
}

func main() {
  startTime := time.Now()
  ch := make(chan Response)

  urls := []string{
```

```
    "https://www.google.com",
    "https://www.github.com",
  }

  for _, url := range urls {
    go fetchURL(url, ch)
  }

  for i := 0; i < len(urls); i++ {
    resp := <-ch
    if resp.err != nil {
      fmt.Printf("Error fetching %s: %v\n",
     resp.url, resp.err)
    } else {
      fmt.Printf("Successfully fetched %s: %s\n",
     resp.url, resp.status)
    }
  }

  close(ch)

  elapsed := time.Since(startTime)

  fmt.Printf("\nTotal time taken: %s\n", elapsed)
}
```

Ce programme définit une structure Response pour contenir l'URL, son statut et toute erreur survenue. Il implémente une fonction fetchURL qui prend en paramètre une URL et un canal, utilise un client HTTP avec un délai d'attente de 10 secondes, effectue une requête GET à l'URL et envoie le résultat via le canal. Elle utilise defer pour s'assurer que le corps de la réponse est fermé. Dans ce cas, le canal peut être écrit ou lu dans la fonction: pour s'assurer qu'il ne peut être qu'écrit, nous pourrions le déclarer comme `ch chan<- Response` au lieu de `ch chan`

`Response` lorsque nous le transmettons. Dans la fonction principale, nous créons un canal pour recevoir les réponses, nous définissons deux URL à récupérer, nous lançons une goroutine pour chaque URL, nous collectons les réponses du canal et nous imprimons les résultats. Lorsque nous exécutons ce programme, il récupère les deux URL simultanément en utilisant des goroutines distinctes, il utilise des canaux pour communiquer les résultats à la goroutine principale et il affiche le statut (comme `200 OK`) ou les éventuelles erreurs pour chaque URL. Nous pouvons réécrire ce programme de manière plus simple, sans goroutines, comme suit:

```
package main

import (
  "fmt"
  "net/http"
  "time"
)

type Response struct {
  url  string
  status string
  err  error
}

func fetchURLSynchro(url string) Response {
  client := &http.Client{
    Timeout: 10 * time.Second,
  }

  resp, err := client.Get(url)
  if err != nil {
    return Response{url, "", err}
  }
  defer resp.Body.Close()
```

```
  return Response{url, resp.Status, nil}
}

func main() {
  urls := []string{
    "https://www.google.com",
    "https://www.github.com",
  }
  startTime := time.Now()

  for i := 0; i < len(urls); i++ {
    resp := fetchURLSynchro(urls[i])
    if resp.err != nil {
      fmt.Printf("Error fetching %s: %v\n",
     resp.url, resp.err)
    } else {
      fmt.Printf("Successfully fetched %s: %s\n",
     resp.url, resp.status)
    }
  }
  elapsed := time.Since(startTime)
  fmt.Printf("\nTotal time taken: %s\n", elapsed)
}
```

Les deux programmes effectuent le même travail, mais l'un utilise deux goroutines (en plus de la goroutine principale) tandis que l'autre n'utilise que la goroutine principale. En testant ces programmes, vous constaterez peut-être que celui qui utilise deux goroutines s'exécute plus rapidement: les accès au réseau sont généralement coûteux et facilement parallélisables. En d'autres termes, les deux tâches peuvent être effectuées presque indépendamment sur votre ordinateur, même si elles sont exécutées simultanément. Ainsi, vous pouvez constater que nous pouvons interroger deux URL à l'aide de requêtes HTTP en 250 ms, alors que 400 ms sont nécessaires si les requêtes sont consécutives, à l'aide d'une seule goroutine.

Cependant, il ne faut pas croire que l'utilisation d'un plus grand nombre de goroutines accélère toujours l'exécution d'un logiciel. Ce n'est souvent pas le cas. En outre, l'ajout de goroutines peut entraîner l'utilisation de processeurs supplémentaires, ce qui augmente le coût ou la consommation d'énergie de votre logiciel. L'ajout de goroutines rend votre logiciel plus compliqué, plus difficile à maintenir et à déboguer. Pour illustrer ce point, considérons le cas où nous additionnons toutes les valeurs d'un tableau. Nous considérons deux cas, d'abord un petit tableau (100k éléments) et ensuite un grand tableau avec des millions d'éléments. Dans les deux cas, nous pouvons utiliser une fonction simple (avec une goroutine) ou une fonction qui utilise plusieurs goroutines. Pour maximiser le parallélisme, nous fixons le nombre de goroutines au nombre de processeurs détectés sur le système par Go (`runtime.NumCPU()`).

```
package main

import (
  "fmt"
  "runtime"
  "testing"
)

func sequentialSum(numbers []int) int {
  sum := 0
  for _, n := range numbers {
    sum += n
  }
  return sum
}

func goroutineSumWithChannels(numbers []int) int {
  numGoroutines := runtime.NumCPU()
  chunkSize := (len(numbers) + numGoroutines - 1)
  / numGoroutines
  resultChan := make(chan int, numGoroutines)
```

```
  activeGoroutines := 0
  for i := 0; i < numGoroutines; i++ {
    start := i * chunkSize
    end := start + chunkSize
    if end > len(numbers) {
      end = len(numbers)
    }
    if start >= end {
      break
    }

    go func(slice []int) {
      partialSum := 0
      for _, n := range slice {
        partialSum += n
      }
      resultChan <- partialSum
    }(numbers[start:end])
    activeGoroutines++
  }

  total := 0
  for i := 0; i < activeGoroutines; i++ {
    total += <-resultChan
  }
  close(resultChan)

  return total
}

func BenchmarkSequentialSum(b *testing.B) {
  numbers := make([]int, 100000)
  for i := range numbers {
    numbers[i] = i
```

```
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    sequentialSum(numbers)
  }
}

func BenchmarkGoroutineSumWithChannels(b *testing.B) {
  numbers := make([]int, 100000)
  for i := range numbers {
    numbers[i] = i
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    goroutineSumWithChannels(numbers)
  }
}

func BenchmarkSequentialSumLarge(b *testing.B) {
  numbers := make([]int, 10000000)
  for i := range numbers {
    numbers[i] = i
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    sequentialSum(numbers)
  }
}

func BenchmarkGoroutineSumWithChannelsLarge(b *testing.B) {
  numbers := make([]int, 10000000)
```

```
  for i := range numbers {
    numbers[i] = i
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    goroutineSumWithChannels(numbers)
  }
}

func main() {
  fmt.Println("Number of CPU cores: ", runtime.NumCPU())

  res :=
  testing.Benchmark(BenchmarkGoroutineSumWithChannels)
  fmt.Println("BenchmarkGoroutineSumWithChannels", res)
  ress := testing.Benchmark(BenchmarkSequentialSum)
  fmt.Println("BenchmarkSequentialSum", ress)

  resl :=
  testing.Benchmark(BenchmarkGoroutineSumWithChannelsLarge)
  fmt.Println("BenchmarkGoroutineSumWithChannelsLarge",
   resl)
  ressl := testing.Benchmark(BenchmarkSequentialSumLarge)
  fmt.Println("BenchmarkSequentialSumLarge", ressl)
}
```

Sur un système doté d'un grand nombre de processeurs, nous pourrions obtenir le résultat suivant:

```
Number of CPU cores: 128
BenchmarkGoroutineSumWithChannels    4048     258798 ns/op
BenchmarkSequentialSum  23756     50516 ns/op
BenchmarkGoroutineSumWithChannelsLarge     744
     1414114 ns/op
```

```
BenchmarkSequentialSumLarge     237      5030224 ns/op
```

Nous constatons qu'en additionnant le modeste tableau, nous obtenons que l'approche utilisant 128 goroutines prend cinq fois plus de temps. Si l'on finit par utiliser 128 processeurs, l'efficacité pourrait être `128 * 5 = 640` fois moindre! La leçon à retenir est que si la tâche est suffisamment peu coûteuse, comme la somme de milliers d'entiers, il ne faut pas utiliser plus d'une goroutine. Dans le cas où nous additionnons 10 millions d'entiers, la tâche parallélisée est plus intéressante: elle est 3,6 fois plus rapide. Là encore, l'approche à routine unique est probablement beaucoup plus efficace: un seul processeur prend 3,6 fois plus de temps que plus d'une centaine de goroutines. Le problème d'une simple somme est qu'elle est pilotée par des accès à la mémoire et qu'elle n'est pas particulièrement calculatoire. Et si nous considérions une tâche plus coûteuse? Faisons la somme du sinus des valeurs d'un tableau en utilisant différents nombres de goroutines (1, 2, ...). Nous utilisons un million de valeurs dans le tableau.

```go
package main

import (
  "fmt"
  "math"
  "runtime"
  "testing"
)

func computeSineSum(numbers []int) float64 {
  sum := 0.0
  for _, n := range numbers {
    sum += math.Sin(float64(n))
  }
  return sum
}

func computeSineSumWithGoroutines(numbers []int,
```

```
 numGoroutines int) float64 {
  chunkSize := (len(numbers) + numGoroutines - 1)
  / numGoroutines
  resultChan := make(chan float64, numGoroutines)

  for i := 0; i < numGoroutines; i++ {
    start := i * chunkSize
    end := start + chunkSize
    if end > len(numbers) {
      end = len(numbers)
    }
    if start >= end {
      break
    }

    go func(slice []int) {
      partialSum := 0.0
      for _, n := range slice {
        partialSum += math.Sin(float64(n))
      }
      resultChan <- partialSum
    }(numbers[start:end])
  }

  total := 0.0
  activeGoroutines := (len(numbers) + chunkSize - 1)
  / chunkSize
  for i := 0; i < activeGoroutines; i++ {
    total += <-resultChan
  }
  close(resultChan)
  return total
}
```

```go
func BenchmarkSequential(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSum(numbers)
  }
}

func Benchmark1Goroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSumWithGoroutines(numbers, 1)
  }
}

func Benchmark2Goroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSumWithGoroutines(numbers, 2)
  }
```

```
}

func Benchmark4Goroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSumWithGoroutines(numbers, 4)
  }
}

func Benchmark8Goroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSumWithGoroutines(numbers, 8)
  }
}

func Benchmark16Goroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
```

```
    computeSineSumWithGoroutines(numbers, 16)
  }
}

func BenchmarkMaxGoroutines(b *testing.B) {
  numbers := make([]int, 1000000)
  for i := range numbers {
    numbers[i] = i % 1000
  }

  b.ResetTimer()
  for i := 0; i < b.N; i++ {
    computeSineSumWithGoroutines(numbers, runtime.NumCPU())
  }
}

func main() {

  fmt.Printf("CPU cores: %d\n", runtime.NumCPU())
  res1 := testing.Benchmark(BenchmarkSequential)
  fmt.Println("BenchmarkSequential", res1)
  res11 := testing.Benchmark(Benchmark1Goroutines)
  fmt.Println("Benchmark1Goroutines", res11)
  res2 := testing.Benchmark(Benchmark2Goroutines)
  fmt.Println("Benchmark2Goroutines", res2)
  res4 := testing.Benchmark(Benchmark4Goroutines)
  fmt.Println("Benchmark4Goroutines", res4)
  res8 := testing.Benchmark(Benchmark8Goroutines)
  fmt.Println("Benchmark8Goroutines", res8)
  res16 := testing.Benchmark(Benchmark16Goroutines)
  fmt.Println("Benchmark16Goroutines", res16)
  resmax := testing.Benchmark(BenchmarkMaxGoroutines)
  fmt.Println("BenchmarkMaxGoroutines", resmax)
}
```

Sur une machine puissante dotée de nombreux cœurs, nous pourrions obtenir les résultats suivants:

```
CPU cores: 128
Benchmark1Goroutines     114    13701908 ns/op
Benchmark2Goroutines     134     8913817 ns/op
Benchmark4Goroutines     253     4648170 ns/op
Benchmark8Goroutines     472     2272842 ns/op
Benchmark16Goroutines    835     1227975 ns/op
BenchmarkMaxGoroutines   916     1189217 ns/op
```

Le passage d'une goroutine à deux goroutines améliore la vitesse d'un facteur 1,5. Passer d'une goroutine à 16 goroutines améliore la vitesse d'un facteur 11. Augmenter le nombre de goroutines au-delà de 16 n'apporte aucun gain supplémentaire. Ce schéma de gains sous-linéaires avec une limite supérieure est assez typique.

Pourtant, les goroutines et les canaux peuvent être remarquablement efficaces en eux-mêmes. Créons une chaîne de canaux. Chaque goroutine possède un canal d'entrée et un canal de sortie. Dès que des données sont reçues dans le canal d'entrée, elles sont écrites dans le canal d'entrée. Nous lions des centaines de goroutines dans une chaîne de canaux d'entrée et de sortie:

```
package main

import (
  "fmt"
  "time"
)

func relay(input <-chan int, output chan<- int) {
  value := <-input
  output <- value
}

func main() {
```

```
  const chainLength = 10000
  channels := make([]chan int, chainLength+1)
  for i := range channels {
    channels[i] = make(chan int)
  }

  startTime := time.Now()

  for i := 0; i < chainLength; i++ {
    go relay(channels[i], channels[i+1])
  }

  go func() {
    channels[0] <- 42
  }()

  result := <-channels[chainLength]

  elapsed := time.Since(startTime)

  fmt.Printf("Value %d successfully passed "
  +"through %d goroutines\n",
   result, chainLength)
  fmt.Printf("Time taken: %v\n", elapsed)
  fmt.Printf("Average time per hop: %v\n",
   elapsed/time.Duration(chainLength))
}
```

En exécutant ce programme, vous obtiendrez peut-être le résultat suivant:

```
Value 42 successfully passed through 10000 goroutines
Time taken: 13.987416ms
Average time per hop: 1.398µs
```

De cette manière, il est possible de parcourir près d'un million de gorou-

tines par seconde.

Les canaux sont fréquemment utilisés dans le contexte de l'accès aux fichiers et aux réseaux. Utiliser plusieurs goroutines est souvent avantageux dans ce contexte, parce que le processeur peut exécuter simultanément différentes tâches, améliorant ainsi l'efficacité et la réactivité du programme.

Le programme suivant crée dix fichiers texte, y écrit le mot `"Canada"`, puis lit chaque fichier pour compter le nombre d'octets qu'il contient, en affichant les résultats. Les opérations sur les fichiers sont effectuées à l'aide de la bibliothèque standard de Go. Tout d'abord, la fonction `os.WriteFile` est utilisée pour créer ou remplacer chaque fichier spécifié dans la liste `files` (f1.txt à f10.txt) et y écrire la chaîne "Canada" sous forme de bytes, avec des permissions de fichier définies à 0644 (lecture/écriture pour le propriétaire, lecture pour les autres). Ensuite, la fonction `countBytes` ouvre chaque fichier avec `os.Open`, qui établit une connexion au fichier pour permettre sa lecture. Si l'ouverture échoue (par exemple, si le fichier n'existe pas), une erreur est renvoyée. La fonction `defer file.Close()` garantit que le fichier est fermé après usage, libérant ainsi les ressources système. Ensuite, `io.ReadAll` lit tout le contenu du fichier ouvert en mémoire sous forme de tableau de bytes, et la longueur de ce tableau (obtenue avec `len(data)`) donne le nombre d'octets du fichier. Ces opérations sont exécutées en parallèle pour chaque fichier à l'aide de goroutines, et les résultats (nom du fichier et nombre d'octets) sont collectés via un canal (`results`) pour être affichés.

```
package main

import (
  "fmt"
  "io"
  "os"
)

type FileResult struct {
  Filename string
```

```
  Bytes    int64
}

func countBytes(filename string) (int64, error) {
  file, err := os.Open(filename)
  if err != nil {
    return 0, err
  }
  defer file.Close()

  data, err := io.ReadAll(file)
  if err != nil {
    return 0, err
  }
  return int64(len(data)), nil
}

func main() {
  files := []string{
    "f1.txt", "f2.txt", "f3.txt", "f4.txt", "f5.txt",
    "f6.txt", "f7.txt", "f8.txt", "f9.txt", "f10.txt",
  }

  for _, filename := range files {
    os.WriteFile(filename, []byte("Canada"), 0644)
  }

  results := make(chan FileResult, len(files))

  for _, file := range files {
    go func(filename string) {
      count, _ := countBytes(filename)
      results <- FileResult{Filename: filename,
        Bytes: count}
```

```
    }(file)
  }
  for i := 0; i < len(files); i++ {
    r := <-results
    fmt.Printf("%s : %d B\n", r.Filename, r.Bytes)
  }
}
```

Groupes d'attente ou *wait groups*

Une autre approche courante pour gérer plusieurs goroutines est l'utilisation de `sync.WaitGroup`. Avant d'explorer un exemple, il est utile de comprendre le rôle des *wait groups* dans la programmation en Go. Un *wait group* est un mécanisme de synchronisation fourni par le paquet `sync`, qui permet à un programme d'attendre que toutes les goroutines lancées aient terminé leur exécution avant de poursuivre. En pratique, un *wait group* maintient un compteur interne. Chaque goroutine incrémente ce compteur au démarrage et le décrémente à la fin de son exécution. La fonction principale peut alors attendre que le compteur revienne à zéro, garantissant ainsi que toutes les tâches asynchrones sont achevées.

Pour utiliser un *wait group*, trois fonctions clés sont employées : `wg.Add`, `wg.Done` et `wg.Wait`. La fonction `wg.Add(n)` incrémente le compteur du *wait group* de `n`, généralement appelée avant de lancer les goroutines pour indiquer le nombre de tâches à attendre. La fonction `wg.Done()` décrémente le compteur de 1, signalant qu'une goroutine a terminé son exécution. Enfin, `wg.Wait()` bloque l'exécution du programme jusqu'à ce que le compteur atteigne zéro, indiquant que toutes les goroutines associées sont terminées. Ces fonctions, utilisées ensemble, assurent une synchronisation robuste et prévisible.

Pour s'asssurer que `wg.Done()` soit appelé, même en cas d'erreur, nous utilisons `defer`. En Go, le mot-clé `defer` est utilisé pour planifier l'exécution d'une fonction juste avant que la fonction englobante (celle contenant

l'instruction `defer`) ne se termine. Cette approche est particulièrement utile pour s'assurer que des opérations de nettoyage, comme la fermeture d'un f ou la décrémentation d'un compteur de *wait group*, sont effectuées même en cas d'erreur ou de retour anticipé. Considérons l'exemple suivant où une division par zéro provoque une panique, mais où `defer` garantit l'exécution d'une instruction avant la terminaison.

```
package main
import "fmt"
func f(x int) int {
    defer fmt.Println("!!!")
    return 1 / x
}
func main() {
    f(0)
}
```

Dans cet exemple, lorsque `f(0)` est appelé, une division par zéro provoque une panique. Cependant, grâce à `defer`, l'instruction `fmt.Println("!!!")` est exécutée avant que le programme ne s'arrête, illustrant comment `defer` garantit l'exécution de code même en cas d'erreur fatale.

L'utilisation de `defer` devient particulièrement pertinente lorsqu'une erreur peut survenir dans une goroutine, nécessitant un nettoyage systématique. Considérons l'exemple suivant où une erreur est générée, mais où `defer` garantit que le compteur du *wait group* est décrémentéé

```
package main

import (
  "fmt"
  "sync"
)

func divide(x int) (int, error) {
```

```
  if x == 0 {
    return 0, fmt.Errorf("x == zero")
  }
  return 1 / x, nil
}

func main() {
  var wg sync.WaitGroup
  wg.Add(1)
  go func() {
    defer wg.Done()
    y, err := divide(0)
    if err != nil {
      fmt.Println("Error:", err)
      return
    }
    fmt.Println(y)
  }()
  wg.Wait()
  fmt.Println("---")
}
```

Dans cet exemple, l'instruction `defer wg.Done()` garantit que le compteur du *wait group* est décrémenté, même si la goroutine se termine prématurément à cause d'une erreur, évitant ainsi un blocage du programme. Le code suivant ne fonctionnerait pas parce que `wg.Done()` ne serait jamais appelé. Le fonction principale ne serait donc jamais informée que la goroutine s'est terminée.

```
  go func() {
    y, err := divide(0)
    if err != nil {
      fmt.Println("Error:", err)
      return
    }
```

```
    fmt.Println(y)
    wg.Done()
  }()
```

Pour illustrer une utilisation plus sophistiquée des *wait groups* et de `defer`, considérons un programme qui incrémente les valeurs dans un tableau. Dans la fonction `main`, un tableau `arr` d'entiers est créé. Quatre goroutines sont utilisées pour diviser le travail, chaque goroutine traitant une portion du tableau, calculée via `chunkSize` pour répartir équitablement les indices. La fonction `incrementChunk` incrémente chaque élément dans une plage donnée du tableau, de `start` à `end`, et utilise un `sync.WaitGroup` pour synchroniser les goroutines. Chaque goroutine appelle `wg.Done()` via `defer` une fois terminée, et `wg.Wait()` dans `main` garantit que toutes les goroutines se terminent avant la fin du programme, assurant ainsi que tous les éléments du tableau sont incrémentés de manière concurrente et sûre.

```
package main

import (
  "sync"
)

func incrementChunk(arr []int, start, end int,
  wg *sync.WaitGroup) {
  defer wg.Done()
  for i := start; i < end && i < len(arr); i++ {
    arr[i]++
  }
}

func main() {
  size := 100000
  arr := make([]int, size)
```

```
  numGoroutines := 4
  chunkSize := (size + numGoroutines - 1) / numGoroutines

  var wg sync.WaitGroup
  wg.Add(numGoroutines)

  for i := 0; i < numGoroutines; i++ {
    start := i * chunkSize
    end := start + chunkSize
    if end > size {
      end = size
    }
    go incrementChunk(arr, start, end, &wg)
  }

  wg.Wait()
}
```

Atomicité

Si vous avez besoin de lire des données à partir de différentes goroutines, ce n'est pas un problème tant que les données restent constantes. Si personne n'écrit sur les données, il n'y a pas de problème. Malheureusement, nous avons souvent besoin de modifier les données, tout en les lisant à partir de différentes goroutines. Il est parfois possible d'utiliser des canaux pour communiquer. Mais cela ne suffit pas toujours. Prenons un exemple. Nous prenons un tableau de 10 entiers, et les goroutines décrémentent aléatoirement un élément du tableau, puis incrémentent un autre élément du tableau. Initialement, la somme de tous les éléments doit être de 1000 et doit rester de 1000 à moins qu'il n'y ait un bug. Nous pouvons implémenter notre code comme suit:

```
package main
```

```
import (
  "fmt"
  "math/rand"
  "sync"
  "time"
)

func main() {


  arr := [10]int{100, 100, 100, 100, 100,
   100, 100, 100, 100, 100}
  var wg sync.WaitGroup


  worker := func() {
    defer wg.Done()
    wg.Add(1)
    r := rand.New(rand.NewSource(time.Now().UnixNano()))


    for i := 0; i < 200000000; i++ {

      idx1 := r.Intn(10)

      if arr[idx1] > 0 {

        arr[idx1]--


        idx2 := r.Intn(10)

        arr[idx2]++
```

```
        }
      }
    }


    go worker()
    go worker()
    fmt.Println("waiting...")
    wg.Wait()
    fmt.Println("waiting...ok")

    fmt.Println("\nFinal array state:", arr)

    sum := 0
    for _, val := range arr {
      sum += val
    }
    fmt.Println("Total sum:", sum)
}
```

Ce programme est erroné: il contient des courses de données parce que nous écrivons et lisons des données à partir de différentes goroutines sans synchronisation. Un résultat possible de ce programme est le suivant:

```
Final array state: [3001 644 880 324 2319 2845 3664
160 232 1741]
Total sum: 15810
```

Observez comment la somme est plus élevée que prévu.

En Go, vous pouvez éviter un tel bug grâce à la garantie d'atomicité fournie par le paquet `sync/atomic`, qui assure que les opérations telles que les incréments sont exécutées en tant qu'étapes indivisibles, empêchant les conditions de course. Des fonctions comme `atomic.AddInt32(&x, 1)` ou `atomic.AddInt64(&x, 1)` assurent que l'opération d'incrémentation (lecture-modification-écriture) est exécutée de manière atomique. Cela

signifie que même si deux threads exécutent l'incrémentation simultanément, les opérations sont sérialisées au niveau matériel et aucune mise à jour n'est perdue.

```
package main

import (
  "fmt"
  "math/rand"
  "sync"
  "sync/atomic"
  "time"
)

func main() {


  arr := [10]int32{100, 100, 100, 100,
   100, 100, 100, 100, 100, 100}
  var wg sync.WaitGroup


  worker := func() {
    defer wg.Done()
    wg.Add(1)
    r := rand.New(rand.NewSource(time.Now().UnixNano()))


    for i := 0; i < 200000000; i++ {

      idx1 := r.Intn(10)

      val := atomic.LoadInt32(&arr[idx1])
      if val > 0 {
        if atomic.CompareAndSwapInt32(&arr[idx1],
```

```
      val, val-1) {

            idx2 := r.Intn(10)

            atomic.AddInt32(&arr[idx2], 1)
          }

        }
      }
    }


    go worker()
    go worker()
    fmt.Println("waiting...")
    wg.Wait()
    fmt.Println("waiting...ok")

    fmt.Println("\nFinal array state:", arr)

    sum := 0
    for _, val := range arr {
      sum += int(val)
    }
    fmt.Println("Total sum:", sum)
}
```

L'expression `atomic.LoadInt32(&arr[idx1])` lit atomiquement la valeur à la position `idx1` du tableau. La valeur est stockée dans une variable locale (`val`): les courses aux données ne sont pas possibles avec une variable locale. Nous utilisons ensuite une opération de comparaison et d'échange (CAS): `atomic.CompareAndSwapInt32(&arr[idx1], val, val-1)`. Elle vérifie si `arr[idx1]` est toujours égal à `val` (la valeur précédemment chargée) et si c'est vrai, elle met `arr[idx1]` à `val-1`. Elle

retourne true en cas de succès, false si la valeur a changé depuis le chargement. Il est important de noter qu'elle s'exécute comme une seule opération atomique. Enfin, nous utilisons `atomic.AddInt32(&arr[idx2], 1)` pour ajouter atomiquement 1 à `arr[idx2]`.

Si vous exécutez le nouveau programme, la somme des valeurs du tableau est maintenue. Le programme est sûr.

Mutex

Les opérations atomiques (comme `atomic.AddInt32` ou `atomic.CompareAndSwapInt32`) sont conçues pour des opérations uniques et indivisibles sur une seule variable. Elles deviennent insuffisantes lorsque nous avons des structures de données plus complexes.

Dans ces cas plus complexes, nous utilisons un mutex. Un mutex (abréviation de *mutual exclusion*) est une primitive de synchronisation utilisée dans la programmation concurrente pour empêcher plusieurs threads ou processus d'accéder simultanément à une ressource partagée ou de la modifier. Il garantit qu'un seul thread (ou goroutine) peut entrer dans une section critique du code à la fois, ce qui permet d'éviter les conditions de course et de maintenir la cohérence des données. Essentiellement, un seul *verrou* peut être détenu à un moment donné.

Pour illustrer notre propos, créons un programme dans lequel de l'argent est transféré entre deux comptes, et nous devons nous assurer que le retrait d'un compte et le dépôt sur un autre se produisent simultanément sans interférence d'autres goroutines. Cela nécessite de protéger une opération en plusieurs étapes, ce qui va au-delà de ce que les opérations atomiques peuvent faire.

```
package main

import (
  "fmt"
  "sync"
```

```
  "time"
)

type Bank struct {
  accounts map[string]int
  mutex  sync.Mutex
}

func NewBank() *Bank {
  return &Bank{
    accounts: map[string]int{
      "Alice": 1000,
      "Bob":   500,
    },
  }
}

func (b *Bank) Transfer(from, to string, amount int,
 wg *sync.WaitGroup) {
  defer wg.Done()

  b.mutex.Lock()
  defer b.mutex.Unlock()


  if b.accounts[from] >= amount {
    b.accounts[from] -= amount
    b.accounts[to] += amount
  } else {
    fmt.Printf("Failed transfer\n")
  }
}

func (b *Bank) GetBalance(account string) int {
```

```
  b.mutex.Lock()
  defer b.mutex.Unlock()
  return b.accounts[account]
}

func main() {
  bank := NewBank()
  var wg sync.WaitGroup


  wg.Add(4)
  go bank.Transfer("Alice", "Bob", 200, &wg)
  go bank.Transfer("Bob", "Alice", 100, &wg)
  go bank.Transfer("Alice", "Bob", 300, &wg)
  go bank.Transfer("Bob", "Alice", 50, &wg)

  wg.Wait()

  fmt.Printf("Final balances: Alice=%d, Bob=%d\n",
    bank.GetBalance("Alice"), bank.GetBalance("Bob"))
}
```

Si aucun autre goroutine n'essaie d'acquérir le même mutex, l'opération de verrouillage est rapide. L'état du mutex est mis à jour à l'aide d'opérations atomiques, telles que le compare-and-swap, qui sont hautement optimisées au niveau matériel et s'exécutent généralement en quelques nanosecondes. Lorsque plusieurs goroutines rivalisent pour le même mutex en même temps, la vitesse du verrouillage varie. Une goroutine peut entrer dans une boucle de spin : elle vérifie répétitivement l'état du mutex dans une boucle serrée pendant une courte période, essayant d'acquérir le verrou sans céder immédiatement au planificateur. Cela peut être efficace si le mutex devrait être libéré rapidement, car cela évite le coût de mise en attente de la goroutine. Cependant, si le mutex reste verrouillé pendant une durée prolongée, la goroutine en spin peut gaspiller des cycles CPU

avant d'être mise en attente par le planificateur Go, ce qui entraîne une latence accrue et une potentielle dégradation des performances. Une fois en attente, la goroutine est placée dans une file d'attente, et le planificateur la réveillera lorsque le mutex sera disponible, ce qui engendre un surcoût supplémentaire en raison du changement de contexte. Dans des cas complexes, il est également possible de déclencher un blocage. Une impasse est une défaillance concurrentielle où les threads sont piégés dans une attente circulaire de ressources, incapables de procéder en raison de dépendances mutuelles. Nous pouvons modifier notre exemple pour inclure une impasse. Au lieu d'un mutex global, nous créons un mutex par compte. Si la goroutine acquiert le compte source puis le compte destination, un blocage devient possible.

```
package main

import (
  "fmt"
  "sync"
  "time"
)

type Account struct {
  balance int
  mutex   sync.Mutex
}

type Bank struct {
  accounts map[string]*Account
}

func NewBank() *Bank {
  return &Bank{
    accounts: map[string]*Account{
      "Alice": {balance: 1000},
      "Bob":   {balance: 500},
```

```
    },
  }
}

func (b *Bank) Transfer(from, to string, amount int,
 wg *sync.WaitGroup) {
  defer wg.Done()


  fromAccount := b.accounts[from]
  toAccount := b.accounts[to]


  fromAccount.mutex.Lock()
  fmt.Printf("Locked %s for transfer of %d to %s\n",
   from, amount, to)

  time.Sleep(100 * time.Millisecond)


  toAccount.mutex.Lock()
  fmt.Printf("Locked %s for transfer of %d from %s\n",
   to, amount, from)


  if fromAccount.balance >= amount {
    fromAccount.balance -= amount
    toAccount.balance += amount
    fmt.Printf("Transferred %d from %s to %s."
  +" New balances: %s=%d, %s=%d\n",
      amount, from, to, from, fromAccount.balance,
     to, toAccount.balance)
  } else {
    fmt.Printf("Failed transfer of %d from %s to %s:"
```

```
  +" insufficient funds\n",
      amount, from, to)
  }


  toAccount.mutex.Unlock()
  fromAccount.mutex.Unlock()
}

func (b *Bank) GetBalance(account string) int {
  acc := b.accounts[account]
  acc.mutex.Lock()
  defer acc.mutex.Unlock()
  return acc.balance
}

func main() {
  bank := NewBank()
  var wg sync.WaitGroup

  wg.Add(2)
  go bank.Transfer("Alice", "Bob", 200, &wg)

  go bank.Transfer("Bob", "Alice", 100, &wg)


  wg.Wait()

  fmt.Printf("Final balances: Alice=%d, Bob=%d\n",
    bank.GetBalance("Alice"), bank.GetBalance("Bob"))
}
```

L'impasse dans ce code se produit parce que deux goroutines acquièrent des mutex dans des ordres différents, ce qui conduit à une attente circulaire.

Une stratégie pour éviter ce type d'impasse consiste à utiliser des mutex ordonnés. Par exemple, si les comptes sont numérotés, nous verrouillons toujours en premier le compte dont le numéro est le plus petit.

Pour de meilleurs performances, il est parfois préférable de remplacer le mutex par un RWMutex. Un RWMutex, ou *read-write mutex*, est une primitive de synchronisation qui permet de gérer l'accès concurrent à une ressource partagée en distinguant les opérations de lecture (*read*) des opérations d'écriture (*write*). Contrairement à un mutex classique, qui verrouille exclusivement l'accès pour toute opération, un RWMutex permet à plusieurs goroutines d'accéder simultanément à la ressource pour des lectures, tant qu'aucune goroutine n'effectue d'écriture. Cela améliore les performances dans les scénarios où les lectures sont fréquentes et les écritures rares. Lorsqu'une goroutine souhaite écrire, elle doit acquérir un verrou exclusif (*Lock*), ce qui empêche toute autre lecture ou écriture pendant cette opération. Les lectures, quant à elles, utilisent un verrou partagé (*RLock*), permettant à plusieurs goroutines de lire la ressource en parallèle sans interférence.

Pour illustrer l'utilisation d'un RWMutex, considérons un programme simulant une cache partagée où plusieurs goroutines lisent des données fréquemment, mais les mises à jour sont moins courantes. Dans cet exemple, nous utilisons un RWMutex pour permettre à plusieurs goroutines de lire les données simultanément, tout en garantissant que les écritures sont protégées de manière exclusive.

```
package main

import (
  "fmt"
  "sync"
  "time"
)

type Cache struct {
  data  map[string]string
```

```
  mutex sync.RWMutex
}

func NewCache() *Cache {
  return &Cache{
    data: map[string]string{
      "key1": "value1",
      "key2": "value2",
    },
  }
}

func (c *Cache) Read(key string, readerID int,
   wg *sync.WaitGroup) {
  defer wg.Done()
  c.mutex.RLock()
  defer c.mutex.RUnlock()
  value, exists := c.data[key]
  if exists {
    fmt.Printf("Reader %d read %s: %s\n",
      readerID, key, value)
  } else {
    fmt.Printf("Reader %d: key %s not found\n",
      readerID, key)
  }
  time.Sleep(100 * time.Millisecond)
}

func (c *Cache) Write(key, value string, writerID int,
  wg *sync.WaitGroup) {
  defer wg.Done()
  c.mutex.Lock()
  defer c.mutex.Unlock()
  c.data[key] = value
```

```
  fmt.Printf("Writer %d wrote %s: %s\n",
    writerID, key, value)
  time.Sleep(200 * time.Millisecond)
}

func main() {
  cache := NewCache()
  var wg sync.WaitGroup

  for i := 1; i <= 3; i++ {
    wg.Add(1)
    go cache.Read("key1", i, &wg)
  }

  wg.Add(1)
  go cache.Write("key3", "value3", 1, &wg)

  wg.Add(1)
  go cache.Read("key2", 4, &wg)

  wg.Wait()
  fmt.Println("Final cache state:", cache.data)
}
```

Ce code implémente une cache partagée utilisant un `RWMutex` pour gérer l'accès concurrent par plusieurs goroutines. Une structure `Cache` contient une map associant des clés à des valeurs et un `sync.RWMutex` pour synchroniser les accès. La fonction `NewCache` initialise la cache avec deux paires clé-valeur. La méthode `Read` permet à plusieurs goroutines de lire simultanément une valeur à l'aide de `RLock`, qui verrouille en mode lecture partagée, affichant la valeur lue ou un message d'erreur si la clé n'existe pas, avec un délai simulé de 100 ms. La méthode `Write` utilise `Lock` pour un verrou exclusif, permettant une mise à jour de la cache, avec un délai simulé de 200 ms. Dans `main`, trois goroutines lisent la clé `"key1"` en

parallèle, une goroutine écrit une nouvelle paire `"key3:value3"`, et une autre lit `"key2"`. Un `WaitGroup` synchronise l'exécution, et l'état final de la cache est affiché.

Faux partages

Un faux partage est un problème de performance qui survient dans les systèmes multicœurs lorsque plusieurs processeurs accèdent à des données situées dans la même ligne de cache, même si ces données sont distinctes. Dans un processeur multicœur, les données en mémoire sont transférées vers le cache par blocs appelés lignes de cache (généralement 64 ou 128 octets). Si deux goroutines, exécutées sur différents cœurs, modifient des variables distinctes mais situées dans la même ligne de cache, le système de cohérence de cache invalide et recharge cette ligne à chaque modification, entraînant des pertes de performance significatives.

En Go, un faux partage peut se produire lorsqu'on utilise des structures de données partagées entre goroutines sans prendre en compte l'alignement des données dans le cache. Par exemple, un tableau de compteurs incrémentés par différentes goroutines peut sembler indépendant, mais si les compteurs sont contigus en mémoire, ils peuvent partager la même ligne de cache, provoquant des conflits.

Voici un exemple illustrant le problème de faux partage en Go, suivi d'une solution pour l'éviter.

```
package main

import (
  "fmt"
  "sync"
  "time"
)

const iterations = 10000000
```

```
type Counter struct {
  counts [4]int64
}

type PaddedCounter struct {
  counts [4]struct {
    value int64
    _     [15]int64
  }
}

func runBenchmark(wg *sync.WaitGroup,
  counter *Counter, index int) {
  defer wg.Done()
  for i := 0; i < iterations; i++ {
    counter.counts[index]++
  }
}

func runPaddedBenchmark(wg *sync.WaitGroup,
  counter *PaddedCounter, index int) {
  defer wg.Done()
  for i := 0; i < iterations; i++ {
    counter.counts[index].value++
  }
}

func main() {
  var wg sync.WaitGroup

  counter := Counter{}
  wg.Add(4)
  start := time.Now()
  for i := 0; i < 4; i++ {
```

```
    go runBenchmark(&wg, &counter, i)
  }
  wg.Wait()
  elapsed := time.Since(start)
  fmt.Printf("No padding - Time: %v\n", elapsed)

  paddedCounter := PaddedCounter{}
  wg.Add(4)
  start = time.Now()
  for i := 0; i < 4; i++ {
    go runPaddedBenchmark(&wg, &paddedCounter, i)
  }
  wg.Wait()
  elapsed = time.Since(start)
  fmt.Printf("Padding - Time: %v\n", elapsed)
}
```

Dans cet exemple, nous avons deux structures : `Counter` et `PaddedCounter`. Dans `Counter`, les quatre compteurs `counts` sont stockés de manière contiguë en mémoire, ce qui augmente la probabilité qu'ils partagent la même ligne de cache. Chaque goroutine incrémente un compteur différent, mais comme ils sont dans la même ligne de cache, les cœurs doivent constamment synchroniser cette ligne, ce qui ralentit l'exécution.

Dans `PaddedCounter`, nous ajoutons du remplissage (un tableau de 15 `int64` inutilisés, soit 120 octets) entre chaque compteur, garantissant que chaque compteur se trouve dans une ligne de cache distincte (les lignes de cache font typiquement d'au plus 128 octets). Cela élimine les conflits de cache, améliorant les performances.

Lors de l'exécution, vous pourriez obtenir des résultats comme ceux-ci sur une machine multicœur.

```
No padding - Time: 24.979625ms
Padding - Time: 9.303167ms
```

Le test avec du remplissage est nettement plus rapide, car il évite les invalidations inutiles des lignes de cache. Pour atténuer le faux partage, il est crucial de comprendre l'alignement des données en mémoire et d'utiliser des techniques comme le remplissage ou des structures de données séparées pour s'assurer que les variables accédées par différentes goroutines résident dans des lignes de cache distinctes.

Conclusion

Le parallélisme est un outil puissant dans le développement des logiciels modernes, permettant aux programmes d'exploiter plusieurs processeurs pour améliorer les performances. Cependant, elle introduit des complexités importantes qui doivent être gérées avec soin. Les courses aux données, où l'accès non synchronisé à la mémoire partagée conduit à des résultats imprévisibles, soulignent la nécessité de mécanismes de synchronisation robustes. Les goroutines et les canaux de Go offrent une approche élégante et légère de la concurrence, permettant aux développeurs de paralléliser efficacement des tâches telles que les requêtes réseau ou le traitement des données, tout en évitant la surcharge des threads traditionnels. Cependant, les avantages du parallélisme en termes de performances ne sont pas garantis - les tâches simples peuvent souffrir d'une surcharge excessive des goroutines, tandis que les opérations à forte intensité de calcul peuvent bénéficier de gains substantiels, bien que les rendements diminuent à mesure que le nombre de goroutines augmente.

Les outils de synchronisation comme `sync.WaitGroup`, les opérations atomiques de `sync/atomic`, et les mutex (`sync.Mutex`) fournissent des garanties essentielles contre les pièges de la concurrence. Les opérations atomiques sont excellentes pour les mises à jour d'une seule variable, assurant la sécurité des threads avec une surcharge minimale, tandis que les mutex protègent les opérations en plusieurs étapes sur les structures de données complexes. Cependant, les mutex comportent des risques, tels que les blocages, qui résultent de dépendances circulaires et nécessitent une conception soignée, comme un ordre cohérent des verrous, pour être évités. Le choix de la bonne stratégie de concurrence dépend de la nature

de la tâche, de son échelle et des exigences de performance. En fin de compte, une programmation concurrente efficace en Go exige un équilibre entre l'exploitation du parallélisme pour la vitesse et le maintien de la simplicité, de la correction et de l'efficacité face à la contention des ressources partagées.

Exercices du chapitre 8

Question 1

Ecrivez un programme Go qui compte le nombre total de mots dans plusieurs chaînes de texte simultanément. Le programme doit: Prendre une tranche de chaînes en entrée (par exemple, `[]string{"Hello world", "Go is awesome", "Concurrency is fun"}`). Utiliser une goroutine pour chaque chaîne afin de compter les mots de cette chaîne (un mot est une séquence de caractères séparés par un espace). Utilisez un canal pour collecter le nombre de mots de chaque goroutine. Additionnez les résultats dans la goroutine principale et affichez le nombre total de mots.

Question 2

Modifiez le programme suivant, qui a une course de données, pour assurer un comportement correct en utilisant le package sync/atomic:

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var counter int
    var wg sync.WaitGroup
```

```
    for i := 0 ; i < 100 ; i++ {
        wg.Add(1)
        go func() {
            différer wg.Done()
            for j := 0 ; j < 1000 ; j++ {
                counter++
            }
        }()
    }

    wg.Wait()
    fmt.Println("Valeur finale du compteur : ", counter)
}
```

Question 3

Étendez l'exemple de transfert bancaire du chapitre pour prendre en charge plusieurs comptes et éviter les blocages. Le programme doit définir une structure Bank avec une correspondance entre les noms de comptes et les structures Account, où chaque Account a un solde (int) et un mutex (sync.Mutex). Vous devez implémenter une méthode Transfer qui déplace de l'argent d'un compte à l'autre, en verrouillant les deux comptes en toute sécurité.