Mastering Programming

From Testing to Performance in Go

Daniel Lemire

Contents

Introduction

When I learned programming as a teenager, I quickly felt like I had mastered it. There were loops, functions, and so on. Yet, every time I tried to embark on an ambitious programming project, I encountered difficulties. Computer programming is a rich activity that can require years of practice. My goal in this book is to help readers who are already familiar with computer programming to better master programming as a whole. In short, the manual aims to partially answer the following question: how can we quickly produce computer code that is correct and efficient to solve problems relevant to an organization? In particular, I want the reader to better connect concerns considered purely technical (low level) like the processor and memory, with more abstract concerns like algorithm design.

Programming languages

Computer programming normally involves a programming language. I assume that the reader is already familiar with an established language, such as Java, C# or C++. I also assume that the reader has a basic understanding of computer organization and algorithms. In this manual, we don't aim to get you to master any particular programming language. All languages have their strengths and weaknesses. We often choose a particular language based on the people we work with, our familiarity with it, or the task at hand. I want this book to be accessible to all readers,

regardless of their preferred programming language. Nevertheless, I want to use concrete, practical programming languages to express myself.

Instead of using a single programming language, I want to be able to use several, depending on the concepts covered. I choose to present many of my examples in a relatively neutral language, the Go programming language. I don't assume that the reader knows the Go language. In fact, I don't assume that the reader will choose to use the Go language in his or her reading. It's enough to know the basic syntax.

The curious reader may wish to take the A Tour of Go tutorial online (https://tour.golang.org/). The developers of the Go language offer an online tool (https://go.dev/play/) that lets you write and run programs in Go.

To get started with Go on Windows, visit the official Go website at https://go.dev/dl/ and download the latest Windows installer. Run the installer, which will add the go binary to your system PATH. To confirm the installation, open a Command Prompt or PowerShell and type `go version`; you should see the installed version, like `go version go1.21.x windows/amd64`. Next, create a directory for your Go projects, such as `C:\Users\YourName\go`.

To get started with Go on macOS, the process is the same. Visit the official Go website at https://go.dev/dl/ and download the latest macOS installer. To confirm the installation, open a Terminal and type `go version`; you should see the installed version. Next, create a directory for your Go projects, such as `/Users/YourName/go`.

Using a text editor, create a file name `hello.go` in your Go directory:

```
package main

import "fmt"

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

The default text editor on Windows is Notepad. The default text editor on macOS is TextEdit.

By default, you need to initialize a module each time you start a new project. Initialize a module by navigating to your project folder and running `go mod init hello`, which creates a `go.mod` file for dependency management. In your case, the module file could be as simple as:

```
module hello

go 1.24.2
```

To compile your code into an executable, use `go build`, which generates an executable file (e.g., `hello.exe`). To start a new project, you can create a new directory and repeat the same process. The Go documentation at https://go.dev/doc/ is an excellent resource for learning more, and the `go env` command can help troubleshoot configuration issues.

Using Visual Studio Code (VS Code) as your editor can make you more productive. After downloading and installing VS Code from https://code.visualstudio.com/, install the Go extension (golang.go) from the Extensions Marketplace. This extension provides features like code completion, formatting, and debugging. Upon installation, VS Code may prompt you to install additional tools. You can open your Go project folder in VS Code, and the editor will recognize .go files, offering inline error checking and suggestions. For example, writing a hello.go file in VS Code allows you to use the integrated terminal to run go run hello.go or set up debugging configurations.

Alternatively, GoLand is a dedicated development tools for Go development. It provides code completion, advanced refactoring tools, built-in debugging, and integration with Go-specific tools. You can purchase GoLand at https://www.jetbrains.com/go/

Here are a few quick syntax points concerning variables and control structures:

```
x := 1 // declaration of the integer x
x = 2 // assigned '2' to the variable
if x > 2 {
  x -= 1
}
for i := 0; i < 10; i++ {
  x += i
}
```

Functions are defined in Go using the keyword `func` followed by the function name. We then declare the function's parameters by sequencing the variable name and its type (in this case `int` for integer). The function declaration ends with a type declaration corresponding to the value returned by the function. We then proceed with the function definition:

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

In Go, a package is a fundamental organizational unit that groups related code together. Every Go source file begins with a package declaration, specifying the package it belongs to, such as package main for executable programs.

With these few syntax basics, you should be able to read code written in Go.

Another useful language is Python. Again, you don't need to master Python, just be familiar with the basic syntax. Languages such as JavaScript or Go allow you to indent your code for better readability: in Python, this indentation is required. Here are a few quick syntax points concerning variables and control structures:

```
x = 1
if x > 2 :
  x -= 1
```

```
for i in range(10):
  x += i
```

Functions in Go are defined with the `func` keyword followed by the function name. In Python, we use the `def` keyword instead. We then proceed with the function definition:

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

Unlike the Go programming language, variable types are not explicit in Python by default. However, you can declare them.

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

Interested readers will easily find tutorials and other guides to Python programming.

To install Python on Windows, visit the official Python website (python.org) and download the latest Python installer for Windows. Run the executable file, ensuring you check the box to "Add Python to PATH" during the setup process. Follow the prompts to complete the installation. Once installed, open the Command Prompt and type `python --version` to verify the installation. If the version number appears, Python is successfully installed and ready for use. It comes with the `pip` package manager.

To install Python on macOS, the process is similar. Go to python.org and download the latest Python installer for macOS. Open the downloaded .pkg file and follow the installer's instructions, which typically involve agreeing to the license and selecting the install location (default settings are usually fine). After installation, open the Terminal and type `python3 --version` to confirm Python is installed correctly. The corresponding package manager `pip3` should also have been isntalled.

Acknowledgements

I would like to thank professor Robert Godin and Wojciech Muła for their comments.

Chapter 1

Our most important goal in writing software is that it be correct. The software must do what the programmer wants it to do. It must meet the needs of the user.

In the business world, double-entry bookkeeping[1] is the idea that transactions are recorded in at least two accounts (debit and credit). One of the advantages of double-entry bookkeeping, compared to a more naive approach, is that it allows for some degree of auditing and error finding. If we compare accounting and software programming, we could say that double-entry accounting and its subsequent auditing is equivalent to software testing.

For an accountant, converting a naive accounting system into a double-entry system is a difficult task in general. In many cases, one would have to reconstruct it from scratch. In the same manner, it can be difficult to add tests to a large application that has been developed entirely without testing. And that is why testing should be first on your mind when building serious software.

A hurried or novice programmer can quickly write a routine, compile and run it and be satisfied with the result. A cautious or experienced programmer will know not to assume that the routine is correct.

Common software errors can cause problems ranging from a program that abruptly terminates to database corruption. The consequences can

[1] https://en.wikipedia.org/wiki/Double-entry_bookkeeping

be costly: a software bug caused the explosion of an Ariane 5 rocket in 1996 (Dowson, 1997[2]). The error was caused by the conversion of a floating point number to a signed integer represented with 16 bits. Only small integer values could be represented. Since the value could not be represented, an error was detected and the program stopped because such an error was unexpected. The irony is that the function that triggered the error was not required: it had simply been integrated as a subsystem from an earlier model of the Ariane rocket. In 1996 U.S. dollars, the estimated cost of this error is almost $400 million.

The importance of producing correct software has long been understood. The best scientists and engineers have been trying to do this for decades.

There are several common strategies. For example, if we need to do a complex scientific calculation, then we can ask several independent teams to produce an answer. If all the teams arrive at the same answer, we can then conclude that it is correct. Such redundancy is often used to prevent hardware-related faults (Yeh, 1996[3]). Unfortunately, it is not practical to write multiple versions of your software in general.

Many of the early programmers had advanced mathematical training. They hoped that we could prove that a program is correct. By putting aside the hardware failures, we could then be certain that we would not encounter any errors. And indeed, today we have sophisticated software that allows us to sometimes prove that a program is correct.

Let us consider an example of formal verification to illustrate our point. We can use the z3 library from Python (De Moura and Bjørner, 2008[4]). If you are not a Python user, don't worry: you don't have to be to follow the example. We can install the necessary library with the command `pip install z3-solver` or the equivalent. Suppose we want to be sure that the inequality `( 1 + y ) / 2 < y` holds for all 32-bit integers. We can use the following script:

[2]https://doi.org/10.1145/251880.251992
[3]https://doi.org/10.1109/AERO.1996.495891
[4]https://doi.org/10.1007/978-3-540-78800-3_24

```
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)
```

In this example we construct a 32-bit word (*BitVec*) to represent our example. By default, the z3 library interprets the values that can be represented by such a variable as ranging from -2147483648 to 2147483647 (from -2^{31} to $2^{31}-1$ inclusive). We enter the inequality opposite to the one we wish to show (`( 1 + y ) / 2 >= y`). If z3 does not find a counterexample, then we will know that the inequality `( 1 + y ) / 2 < y` holds.

When running the script, Python displays the integer value 2863038463 which indicates that z3 has found a counterexample. The z3 library always gives a positive integer and it is up to us to interpret it correctly. The number 2147483648 becomes -2147483648, the number 2147483649 becomes -2147483647 and so on. This representation is often called the two's complement[5]. Thus, the number 2863038463 is in fact interpreted as a negative number. Its exact value is not important: what matters is that our inequality (`( 1 + y ) / 2 < y`) is incorrect when the variable is negative. We can check this by giving the variable the value -1, we then get `0 < -1`. When the variable takes the value 0, the inequality is also false (`0<0`). We can also check that the inequality is false when the variable takes the value 1. So let us add as a condition that the variable is greater than 1 (`s.add( y > 1 )`):

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

[5]https://en.wikipedia.org/wiki/Two%27s_complement

```
s.add( y > 1 )

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

Since the latter script does not display anything on the screen when it is executed, we can conclude that the inequality is satisfied as long as the variable of variable is greater than 1.

Since we have shown that the inequality `( 1 + y ) / 2 < y` is true, perhaps the inequality `( 1 + y ) < 2 * y` is true too? Let's try it:

```
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)
```

This script will display 1412098654, half of 2824197308 which is interpreted by z3 as a negative value. To avoid this problem, let's add a new condition so that the double of the variable can still be interpreted as a positive value:

```
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)
```

This time the result is verified. As you can see, such a formal approach requires a lot of work, even in relatively simple cases. It may have been possible to be more optimistic in the early days of computer science, but by the 1970s, computer scientists like Dijkstra were expressing doubts:

> 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[6])

It is impractical to apply such a mathematical method on a large scale. Errors can take many forms, and not all of these errors can be concisely presented in mathematical form. Even when it is possible, even when we can accurately represent the problem in a mathematical form, there is no reason to believe that a tool like z3 will always be able to find a solution: when problems become difficult, computational times can become very long. An empirical approach is more appropriate in general.

Over time, programmers have come to understand the need to test their software. It is not always necessary to test everything: a prototype or an example can often be provided without further validation. However, any software designed in a professional context and having to fulfill an important function should be at least partially tested. Testing allows us to reduce the probability that we will have to face a disastrous situation.

There are generally two main categories of tests.

- There are unit tests. These are designed to verify a particular component of a software program. For example, a unit test can be performed on a single function. Most often, unit tests are automated:

[6] https://doi.org/10.1145/800027.808478

the programmer can execute them by pressing a button or typing a command. Unit tests often avoid the acquisition of valuable resources, such as creating large files on a disk or making network connections. Unit testing does not usually involve reconfiguring the operating system. They are often derived from the expectations of the programmers.

- Integration tests aim to validate a complete application. They often require access to networks and access to sometimes large amounts of data. They sometimes require manual intervention and specific knowledge of the application. Integration testing may involve reconfiguring the operating system and installing software. They can also be automated, at least in part. Most often, they are based on unit tests that serve as a foundation. Integration tests can validate several requirements such as security or environmental compliance. They are often closer to the user needs than the unit tests.

Unit tests are often part of a continuous integration process (Kaiser et al., 1989[7]). Continuous integration often automatically performs specific tasks including unit testing, backups, applying cryptographic signatures, and so on. Continuous integration can be done at regular intervals, or whenever a change is made to the code.

Unit tests are used to structure and guide software development. Tests can be written before the code itself, in which case we speak of *test-driven development*. Often, tests are written after developing the functions. Tests can be written by programmers other than those who developed the functions. It is sometimes easier for independent developers to provide tests that are capable of uncovering errors because they do not share the same assumptions.

It is possible to integrate tests into functions or an application. For example, an application may run a few tests when it starts. In such a case, the tests will be part of the distributed code. However, it is more

[7]https://doi.org/10.1109%2FCMPSAC.1989.65147

common not to publish unit tests. They are a component reserved for programmers and they do not affect the functioning of the application. In particular, they do not pose a security risk and they do not affect the performance of the application.

Experienced programmers often consider tests to be as important as the original code. It is therefore not uncommon to spend half of one's time on writing tests. The net effect is to substantially reduce the initial speed of writing computer code. Yet this apparent loss of time often saves time in the long run: setting up tests is an investment. Software that is not well tested is often more difficult to update. The presence of tests allows us to make changes or extensions with less uncertainty.

Tests should be readable, simple and they should run quickly. They often use little memory.

Unfortunately, it is difficult to define exactly how good tests are. There are several statistical measures. For example, we can count the lines of code that execute during tests. We then talk about test coverage. A coverage of 100% implies that all lines of code are concerned by the tests. In practice, this coverage measure can be a poor indication of test quality.

Consider this example[8]:

```
package main

import (
  "testing"
)

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

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

[8]https://play.golang.org/p/nwMUq2o_WlX

```
        t.Error(Average(2,4))
    }
}
```

Our test simply checks that the average of 2 and 4 is 3.

In the Go language, we can run tests with the command `go test`. We put the tests in files named with the suffix `_test.go` (e.g., `mycode_test.go`), the test functions have the prefix `Test` (e.g., `TestAverage`) which take as a parameter a value of type `*testing.T`: in case of error, we call the function `t.Error` where `t` is of type `*testing.T`. In our example, we have an `Average` function with a corresponding test. The test will run successfully. If you called the file `average_test.go`, you can run test tests as `go test average_test.go`. The coverage is 100%.

Unfortunately, the `Average` function may not be as correct as we would expect. If we pass the integers 40000 and 40000 as parameters, we would expect the average value of 40000 to be returned. But the integer 40000 added to the integer 40000 cannot be represented with a 16-bit integer (`uint16`): the result will be instead `(40000+4000)%65536=14464`. So the function will return 7232 which may be surprising. The following test will fail:

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

It is easy in Go to automatically generate a coverage report. It is more convenient to create a module first.
In a new directory, create the following three files.

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))
  }
}
```

While in the newly created directory, run the following commands:

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

Go will then inform you that the coverage is 50%. Indeed, the `Average2` function is not tested.

Code coverage is useful for identifying parts of a program that are not reached by tests. In the case of unfamiliar code, it provides an overview of what is happening. A lack of coverage may indicate dead code (code that serves no purpose).

When possible and fast, we can try to test the code more exhaustively, like in this example[9] where we include several values:

```
package main

import (
  "testing"
)


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

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

[9] https://play.golang.org/p/nlq_J_-Tw8F

```
        }
      }
    }
  }
}
```

Our test merely checks that the result falls within `[x,y]` when `y>x` and within `[y,x]` when `x>y`. In practice, one could be more careful and check that we compute the average. We leave this as an exercise to the reader.

With a bit of patience, it is possible to test billions of cases. For example, the following test verifies that all integers from `math.MinInt32` to `math.MaxInt32` can be converted to a string (using `strconv.Itoa`) and then have the original integer recovered from the produced string (`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("Parsing error for %d: %v", i, err)
      continue
    }
    if parsed != i {
      t.Errorf("For %d, got %d after conversion",
        i, parsed)
```

```
    }
  }
}
```

If you name this file `exhaustive_test.go`, you should be able to run `go test exhaustive_test.go` in a few minutes.

In practice, it is rare that we can do exhaustive tests. We can instead use pseudo-random tests. For example, we can generate pseudo-random numbers and use them as parameters. In the case of random tests, it is important to keep them deterministic: each time the test runs, the same values are tested. This can be achieved by providing a fixed *seed* to the random number generator as in this example[10]:

```
package main

import (
  "testing"
  "math/rand"
)

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

func TestAverage(t *testing.T) {
  rand.Seed(1234)
  for test := 0; test < 1000; test++ {
    x := rand.Intn(65536)
    y := rand.Intn(65536)
```

[10] https://play.golang.org/p/XGoxJoxfiEJ

```
    m := int(Average(uint16(x),uint16(y)))
    if x < y {
      if m < x || m > y {
        t.Error("error ", x, " ", y)
      }
    } else {
      if m < y || m > x {
        t.Error("error ", x, " ", y)
      }
    }
  }
}
```

Tests based on random exploration are part of a strategy often called *fuzzing* (Miller at al., 1990[11]).

The Go language lets you fuzz automatically, without having to generate your own random values:

```
// 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
  }
}
```

[11] https://doi.org/10.1145/96267.96279

```
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)
      }
    }
  })
}
```

We generally distinguish two types of tests. Positive tests aim at verifying that a function or component behaves in an agreed way. Thus, the first test of our `Average` function was a positive test. Negative tests verify that the software behaves correctly even in unexpected situations. We can produce negative tests by providing our functions with random data (*fuzzing*). Our second example can be considered a negative test if the programmer expected small integer values.

The tests should fail when the code is modified (Budd et al., 1978[12]). On this basis, we can also develop more sophisticated measures by testing for random changes in the code and ensuring that such changes often cause tests to fail.

Some programmers choose to generate tests automatically from the code. In such a case, a component is tested and the result is captured. For example, in our example of calculating the average, we could have captured the fact that `Average(40000,40000)` has the value 7232. If a subsequent change occurs that changes the result of the operation, the test will fail. Such an approach saves time since the tests are generated automatically.

[12]https://doi.org/10.1109/AFIPS.1978.195

We can quickly and effortlessly achieve 100% code coverage. On the other hand, such tests can be misleading. In particular, it is possible to capture incorrect behaviour. Furthermore, the objective when writing tests is not so much their number as their quality. The presence of several tests that do not contribute to validate the essential functions of our software can even become harmful. Irrelevant tests can waste programmers' time in subsequent revisions.

Regarding integration tests, the extensive external systems they often rely on, such as databases, network services, or third-party APIs, can frequently be simulated using mocks or stubs. In this context, a mock is a simulated version of an external component that exposes the same API as the real system but allows the test author to control its behavior directly within the test. This enables testing of the application's interactions with these systems without relying on their actual availability, which can be costly, slow, or prone to external failures. The ability to use mocks effectively hinges on a well-designed system architecture, particularly one that employs dependency injection to provide services rather than hardcoding dependencies, ensuring components are loosely coupled and easily replaceable.

Dependency injection is a design pattern in software engineering that promotes loose coupling between components by passing dependencies (services or objects a component needs) into a component rather than having the component create or directly reference them. This approach makes the system more modular, testable, and maintainable, as dependencies can be easily swapped or mocked during testing without altering the component's code. Instead of hardcoding dependencies (embedding them), they are "injected" from the outside, typically through constructors, setters, or interfaces.

For example, consider an application that integrates with a payment processing service. In a real-world scenario, the application sends requests to the service's API to process transactions. During integration testing, connecting to the actual payment service might be impractical due to costs, network issues, or the need for specific test scenarios (e.g., simulating a

failed payment). Instead, a mock can be created to mimic the payment service's API. The mock would respond to API calls in a controlled way, as defined by the test. For instance, a test might configure the mock to return a "payment successful" response for a valid request or a "payment declined" error for an invalid card number. This allows the test to verify how the application handles these responses without ever contacting the real payment service.

Suppose we have an application that processes orders and interacts with a payment service. We want to test the order processing logic without calling the real payment service, using a mock instead.

```
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) {
  // Simulate calling an external payment gateway.
  return "tx_12345", nil
}

type MockPaymentService struct {
  // Fields to control mock behavior.
  transactionID string
  err           error
}
```

```
func (m *MockPaymentService) ProcessPayment(orderID string,
  amount float64) (string, error) {
  // Return predefined values for testing.
  return m.transactionID, m.err
}

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("payment failed: %w", err)
  }
  return transactionID, nil
}

func main() {
  // Example usage in a real application.
  realPaymentService := &RealPaymentService{}
  processor := NewOrderProcessor(realPaymentService)
  transactionID, err := processor.ProcessOrder("order_001",
    99.99)
  if err != nil {
```

```
    fmt.Printf("Error: %v\n", err)
    return
  }
  fmt.Printf("Real service transaction ID: %s\n",
    transactionID)

  // Example usage in a test scenario with a mock.
  mockPaymentService := &MockPaymentService{
    transactionID: "mock_tx_999",
    err:           nil,
  }
  processor = NewOrderProcessor(mockPaymentService)
  transactionID, err = processor.ProcessOrder("order_002",
    49.99)
  if err != nil {
    fmt.Printf("Error: %v\n", err)
    return
  }
  fmt.Printf("Mock service transaction ID: %s\n",
    transactionID)
}
```

This Go code demonstrates dependency injection to facilitate integration testing by allowing a component, `OrderProcessor`, to work with either a real or mock implementation of a `PaymentService`. The `PaymentService` interface defines a `ProcessPayment` method, which is implemented by two structs: `RealPaymentService`, simulating interaction with an external payment gateway, and `MockPaymentService`, used for testing with configurable responses. The `OrderProcessor` struct depends on a `PaymentService`, which is injected via the `NewOrderProcessor` constructor, ensuring loose coupling. The `ProcessOrder` method calls the injected service's `ProcessPayment` method and handles its response, returning a transaction ID or an error. This design allows the `OrderProcessor`

to remain agnostic about whether it's interacting with a real or mock service, enhancing modularity and testability.

In the `main` function, the code showcases both real and test scenarios. First, it creates a `RealPaymentService`, injects it into an `OrderProcessor`, and processes an order, printing a transaction ID (e.g., `tx_12345`). Then, it demonstrates a test scenario by injecting a `MockPaymentService`, configured with a predefined `transactionID` (`mock_tx_999`) and no error. The `OrderProcessor` processes another order, printing the mock's transaction ID. The mock's fields allow test authors to simulate various outcomes (e.g., success or failure) without relying on an external system. This structure exemplifies how dependency injection enables seamless swapping of dependencies, making integration tests faster, more reliable, and independent of external services.

Table-based tests allow for verifying a function using a predefined set of inputs and expected outputs, organized in a data structure, typically a slice of structs. This approach is particularly effective for testing a function with multiple test cases, such as an empty string, zero, or a maximum value, while reducing code duplication. In the following example, we test a function that calculates the square of an integer, validating its behavior with varied inputs, including an empty string converted to an integer, zero, and a value exceeding the maximum integer.

```
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)
      }
    })
  }
}
```

As the code evolves, we aim to run tests frequently, potentially after every minor change. A software regression is a bug where a previously functioning feature stops working. In some cases, we may need to execute all tests, including unit and integration tests. This process is commonly referred to as regression testing.

Finally, we review the benefits of testing: tests help us organize our work, they are a measure of quality, they help us document the code, they avoid regression, they help debugging and they can produce more efficient code.

Organization

Designing sophisticated software can take weeks or months of work. Most often, the work will be broken down into separate units. It can be difficult, until you have the final product, to judge the outcome. Writing tests as we develop the software helps to organize the work. For example, a given component can be considered complete when it is written and tested. Without the test writing process, it is more difficult to estimate the progress of a project since an untested component may still be far from being completed.

Quality

Tests are also used to show the care that the programmer has put into his work. They also make it possible to quickly evaluate the care taken with the various functions and components of a software program: the presence of carefully composed tests can be an indication that the corresponding code is reliable. The absence of tests for certain functions can serve as a warning.

Some programming languages are quite strict and have a compilation phase that validates the code. Other programming languages (Python, JavaScript) leave more freedom to the programmer. Some programmers consider that tests can help to overcome the limitations of less strict programming languages by imposing on the programmer a rigour that the language does not require.

A related approach is the use of *linters*. These are tools that review code for potential errors. While they are not a form of testing per se, they can help us find errors. The Go language has a builtin linter that you can invoke with the command `go vet`.

Consider the following Go code.

```
package main

import (
  "fmt"
)

func main() {
  num := 42
  str := string(num) //strconv.Itoa
  fmt.Print(str)

  name := "Alice"
  fmt.Printf("%s %d\n", name)
}
```

There are two likely errors in this code. The first likely error is that, to convert a number into a string, the proper function is `strconv.Itoa` and not `string`. The expression `string(42)` returns the ASCII character corresponding to the codepoint value 42 which is `*`. The second error is that the expression `fmt.Printf("%s %d\n", name)` is missing a third parameter of a integer type. Running `go vet` in a directory with a the code in question results in the following warnings.

```
./bad.go:13:2: fmt.Printf format %d
reads arg #2, but call has 1 arg
./bad.go:9:9: conversion from int
to string yields a string of one rune,
not a string of digits
```

It is common practice to automatically execute linters like `go vet` as a form of quality control. However, we must be careful since these tools tend to produce false positives (report errors that are not errors).

Documentation

Software programming should generally be accompanied by clear and complete documentation. In practice, the documentation is often partial, imprecise, erroneous or even non-existent. Tests are therefore often the only technical specification available. Reading tests allows programmers to adjust their expectations of software components and functions. Unlike documentation, tests are usually up-to-date, if they are run regularly, and they are accurate to the extent that they are written in a programming language. Tests can therefore provide good examples of how the code is used. Even if we want to write high-quality documentation, tests can also play an important role. To illustrate computer code, examples are often used. Each example can be turned into a test. So we can make sure that the examples included in the documentation are reliable. When the code changes, and the examples need to be modified, a procedure to test our examples will remind us to update our documentation. In this way, we avoid the frustrating experience of readers of our documentation finding examples that are no longer functional.

Go supports examples as test. In a file containing tests, you can add a function that begins with the string `Example` and includes the name of a function we want document. The example function should take no parameter. The function should end with a comment starting with `Output:`. It automatically becomes a test. At testing time, function is executed and the result printed out by the function is compared with the comment starting with `Output:`. The following code illustrates the concept with a function called `Sum` and an example function called `ExampleSum`. At testing time, the `ExampleSum()` function is called, it prints `3` which matches the comment (`// Output: 3`) and, thus, the test passes.

```
package main

import (
  "fmt"
)
```

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

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

Regression

Programmers regularly fix flaws in their software. It often happens that the same problem occurs again. The same problem may come back for various reasons: sometimes the original problem has not been completely fixed. Sometimes another change elsewhere in the code causes the error to return. Sometimes the addition of a new feature or software optimization causes a bug to return, or a new bug to be added. When software acquires a new flaw, it is called a regression. To prevent such regressions, it is important to accompany every bug fix or new feature with a corresponding test. In this way, we can quickly become aware of regressions by running the tests. Ideally, the regression can be identified while the code is being modified, so we avoid regression. In order to convert a bug into a simple and effective test, it is useful to reduce it to its simplest form. For example, in our previous example with `Average(40000,40000)`, we can add the detected error in additional test[13]:

```
package main

import (
  "testing"
)
```

[13] https://play.golang.org/p/PH9y3ZqV2c9

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

func TestAverage(t *testing.T) {
   if Average(2,4) != 3 {
     t.Error("error 1")
   }
   if Average(40000,40000) != 40000 {
     t.Error("error 2")
   }
}
```

Bug fixing

In practice, the presence of an extensive test suite makes it possible to identify and correct bugs more quickly. This is because testing reduces the extent of errors and provides the programmer with several guarantees. To some extent, the time spent writing tests saves time when errors are found while reducing the number of errors. Furthermore, an effective strategy to identify and correct a bug involves writing new tests. It can be more efficient on the long run than other debugging strategies such as stepping through the code. Indeed, after your debugging session is completed, you are left with new unit tests in addition to a corrected bug.

Performance

The primary function of tests is to verify that functions and components produce the expected results. However, programmers are increasingly using tests to measure the performance of components. For example, the execution speed of a function, the size of the executable or the memory usage can be measured. It is then possible to detect a loss of performance following a modification of the code. You can compare the performance of your code against a reference code and check for differences using statistical tests.

Conclusion

All computer systems have flaws. Hardware can fail at any time. And even when the hardware is reliable, it is almost impossible for a programmer to predict all the conditions under which the software will be used. No matter who you are, and no matter how hard you work, your software will not be perfect. Nevertheless, you should at least try to write code that is generally correct: it most often meets the expectations of users. It is possible to write correct code without writing tests. Nevertheless, the benefits of a test suite are tangible in difficult or large-scale projects. Many experienced programmers will refuse to use a software component that has been built without tests. They might say: "If it is not tested, it does not work." The habit of writing tests probably makes you a better programmer. Psychologically, you are more aware of your human limitations if you write tests. When you interact with other programmers and with users, you may be better able to take their feedback into account if you have a test suite. We have said little about integration tests. They often depend more specifically on the application and the users. Your foundation should be the unit tests.

Suggested reading

- 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.

Exercises for chapter 1

Problem 1.

Write a function that converts floating point numbers (`float64` in Go) to signed integers represented with 16 bits. Then write a test to check that this function is correct: when the value is indeed an integer, it is returned, otherwise an error is generated.

Problem 2.

Write a function that adds two unsigned integers (`uint16`), but produces zero if the result cannot be represented in the same form (`uint16`). Then write a corresponding test.

Chapter 2

In practice, computer code is constantly being transformed. At the beginning of a project, the computer code often takes the form of sketches that are gradually refined. Later, the code can be optimized or corrected, sometimes for many years.

Soon enough, programmers realized that they needed to not only to store files, but also to keep track of the different versions of a given file. It is no accident that we are all familiar with the fact that software is often associated with versions. It is necessary to distinguish the different versions of the computer code in order to keep track of updates.

We might think that after developing a new version of a software, the previous versions could be discarded. However, it is practical to keep a copy of each version of the computer code for several reasons:

1. A change in the code that we thought was appropriate may cause problems: we need to be able to go back quickly.
2. Sometimes different versions of the computer code are used at the same time and it is not possible for all users to switch to the latest version. If an error is found in a previous version of the computer code, it may be necessary for the programmer to go back and correct the error in an earlier version of the computer code without changing the current code. In this scenario, the evolution of the software is not strictly linear. It is therefore possible to release version 1.0, followed by version 2.0, and then release version 1.1.

3. It is sometimes useful to be able to go back in time to study the evolution of the code in order to understand the motivation behind a section of code. For example, a section of code may have been added without much comment to quickly fix a new bug. The attentive programmer will be able to better understand the code by going back and reading the changes in context.
4. Computer code is often modified by different programmers working at the same time. In such a social context, it is often useful to be able to quickly determine who made what change and when. For example, if a problem is caused by a segment of code, we may want to question the programmer who last worked on that segment.

Furthermore, when many people work on the same software code at the same time, there are many different versions of the same code at the same time. Maybe each programmer in a team has their local copy and they are differ slightly. We need to regularly merge the work so that there is only one definitive version of the code. Ideally, we would like the code synchronization to be automated and safe. In particular, we do not want to lose valuable work in the process.

Version control systems

Programmers quickly realized that they needed version control systems. The basic functions that a version control system provides are rollback, the addition of new changes, and a history of changes made. Over time, the concept version control has spread. There are even several variants intended for the general public such as DropBox where various files, not only computer code, are stored.

The history of software version control tools dates back to the 1970s (Rochkind, 1975[14]). In 1972, Rochkind developed the SCCS (Source Code Control System) at Bell Laboratories. This system made it possible to create, update and track changes in a software project. SCCS remained a reference from the end of the 1970s until the 1980s. One of the constraints

[14]http://doi.org/10.1109/TSE.1975.6312866

of SCCS is that it does not allow collaborative work: only one person can modify a given file at a given time.

In the early 1980s, Tichy proposed the RCS (Revision Control System), which innovated with respect to SCCS by storing only the differences between the different versions of a file in backward order, starting from the latest file. In contrast, SCCS stored differences in forward order starting from the first version. For typical use where we access the latest version, RCS is faster.

In programming, we typically store computer code within text files. Text files most often use ASCII or Unicode (UTF-8 or UTF-16) encoding. Lines are separated by a sequence of special characters that identify the end of a line and the beginning of a new line. Two characters are often used for this purpose: "carriage return" (CR) and "line feed" (LF). In ASCII and UTF-8, these characters are represented with the byte having the value 13 and the byte having the value 10 respectively. In Windows, the sequence is composed of the CR character followed by the LF character, whereas in Linux and macOS, only the LF character is used. In most programming languages, we can represent these two characters with the escape sequences `\r` and `\n` respectively. So the string `"a\nb\nc"` has three lines in most programming languages under Linux or macOS: the lines "a", "b" and "c".

When a text file is edited by a programmer, usually only a small fraction of all lines are changed. Some lines may also be inserted or deleted. It is convenient to describe the differences as succinctly as possible by identifying the new lines, the deleted lines and the modified lines.

The calculation of differences between two text files is often done first by breaking the text files into lines. We then treat a text file as a list of lines. Given two versions of the same file, we want to associate as many lines in the first version as possible with an identical line in the second version. We also assume that the order of the lines is not reversed.

We can formalize this problem by looking for the longest common subsequence. Given a list, a subsequence simply takes a part of the list,

excluding some elements. For example, (a,b,d) is a subsequence of the list (a,b,c,d,e). Given two lists, we can find a common subsequence, e.g. (a,b,d) is a subsequence of the list (a,b,c,d,e) and the list (z,a,b,d). The longest common subsequence between two lists of text lines represents the list of lines that have not been changed between the two versions of a text file. It might be difficult to solve this program using brute force. Fortunately, we can compute the longest common subsequence by dynamic programming. Indeed, we can make the following observations.

1. If we have two strings with a longest subsequence of length k, and we add at the end of each of the two strings the same character, the new strings will have a longer subsequence of length k+1.
2. If we have two strings of lengths m and n, ending in distinct characters (for example, "abc" and "abd"), then the longest subsequence of the two strings is the longest subsequence of the two strings after removing the last character from one of the two strings. In other words, to determine the length of the longest subsequence between two strings, we can take the maximum of the length of the subsequence after amputating one character from the first string while keeping the second unchanged, and the length of the subsequence after amputating one character from the second string while keeping the first unchanged.

These two observations are sufficient to allow an efficient calculation of the length of the longest common subsequence. It is sufficient to start with strings comprising only the first character and to add progressively the following characters. In this way, one can calculate all the longest common subsequences between the truncated strings. It is then possible to reverse this process to build the longest subsequence starting from the end. If two strings end with the same character, we know that the last character will be part of the longest subsequence. Otherwise, one of the two strings is cut off from its last character, making our choice in such a way as to maximize the length of the longest common subsequence.

The following function illustrates a possible solution to this problem. Given two arrays of strings, the function returns the longest common

subsequence. If the first string has length `m` and the second `n`, then the algorithm runs in `O(m*n)` time.

```
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
}
```

Once the subsequence has been calculated, we can quickly calculate a description of the difference between the two text files. Simply move forward in each of the text files, line by line, stopping as soon as you

reach a position corresponding to an element of the longest sub-sequence. The lines that do not correspond to the subsequence in the first file are considered as having been deleted, while the lines that do not correspond to the subsequence in the second file are considered as having been added. The following function illustrates a possible solution.

```
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
}
```

The function we propose as an illustration for computing the longest subsequence uses `O(m*n)` memory elements. It is possible to reduce the memory usage of this function and simplify it (Hirschberg, 1975[15]). Several other improvements are possible in practice (Miller and Myers, 1985[16]). We can then represent the changes between the two files in a concise way.

Suggested reading: article Diff (wikipedia)[17]

Like SCCS, RCS does not allow multiple programmers to work on the same file at the same time. The need to own a file to the exclusion of all other programmers while working on it may have seemed a reasonable constraint at the time, but it can make the work of a team of programmers much more cumbersome.

In 1986, Grune developed the Concurrent Versions System (CVS). Unlike previous systems, CVS allows multiple programmers to work on the same file simultaneously. It also adopts a client-server model that allows a single directory to be present on a network, accessible by multiple programmers simultaneously. The programmer can work on a file locally, but as long as he has not transmitted his version to the server, it remains invisible to the other developers.

The remote server also serves as a de facto backup for the programmers. Even if all the programmers' computers are destroyed, it is possible to start over with the code on the remote server.

In a version control system, there is usually always a single latest version. All programmers make changes to this latest version. However, such a linear approach has its limits. An important innovation that CVS has updated is the concept of a branch. A branch allows to organize sets of versions that can evolve in parallel. In this model, the same file is virtually duplicated. There are then two versions of the file (or more than two) capable of evolving in parallel. By convention, there is

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

usually one main branch that is used by default, accompanied by several secondary branches. Programmers can create new branches whenever they want. Branches can then be merged: if a branch A is divided into two branches (A and B) which are modified, it is then possible to bring all the modifications into a single branch (merging A and B). The branch concept is useful in several contexts:

1. Some software development is speculative. For example, a programmer may explore a new approach without being sure that it is viable. In such a case, it may be better to work in a separate branch and merge with the main branch only if successful.
2. The main branch may be restricted to certain programmers for security reasons. In such a case, programmers with reduced access may be restricted to separate branches. A programmer with privileged access may then merge the secondary branch after a code inspection.
3. A branch can be used to explore a particular bug and its fix.
4. A branch can be used to update a previous version of the code. Such a version may be kept up to date because some users depend on that earlier version and want to receive certain fixes. In such a case, the secondary branch may never be integrated with the main branch.

While it is relatively easy to define algorithmically which lines changed between successive revisions of the same file, different simultaneous changes on the same file make the problem more difficult. In practice, these simultaneous changes may lead to *conflicts* which require users to make decisions. For example, suppose that we start with a file containing a single function:

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

User A adds a function `fa`...

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

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

User B adds a function `fb` in a different branch...

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

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

How we merge these changes is a suggestive choice. Should there be three functions (`f1`, `fa` and `fb`)? Or are the two users adding the same function, albeit using different names? In that case, we should have only two functions in the file file.

Thus, in practice, doing much work using different branches modifying the same files might become increasingly challenging, as merging the results could require more and more human intervention.

One of the drawbacks of CVS is poor performance when projects include multiple files and multiple versions. In 2000, Subversion (SVN) was proposed as an alternative to CVS that meets the same needs, but with better performance.

CVS and Subversion benefit from a client-server approach, which allows multiple programmers to work simultaneously with the same version directory. Yet programmers often want to be able to use several separate remote directories.

Distributed version control systems

To meet these needs, various "distributed version control systems" (DVCS) have been developed. The most popular one is probably the Git system developed by Torvalds (2005). Torvalds was trying to solve a problem of managing Linux source code. Git became the dominant version management tool. It has been adopted by Google, Microsoft, etc. It is free software.

In a distributed model, a programmer who has a local copy of the code can synchronize it with either one directory or another. They can easily create a new copy of the remote directory on a new server. Such flexibility is considered essential in many complex projects such as the Linux operating system kernel.

Several companies offer Git-based services including GitHub. Founded in 2008, GitHub has tens of millions of users. In 2018, Microsoft acquired GitHub for $7.5 billion.

For CVS and Subversion, there is only one set of software versions. With a distributed approach, multiple sets can coexist on separate servers. The net result is that a software project can evolve differently, under the responsibility of different teams, with possible future reconciliation.

In this sense, Git is distributed. Although many users rely on GitHub (for example), your local copy can be attached to any remote directory, and it can even be attached to multiple remote directories. The verb "clone" is sometimes used to describe the recovery of a Git project locally, since it is a complete copy of all files, changes, and branches.

If a copy of the project is attached to another remote directory, it is called a fork. We often distinguish between branches and forks. A branch always belongs to the main project. A fork is originally a complete copy of the project, including all branches. It is possible for a fork to rejoin the main project, but it is not essential.

Given a publicly available Git directory, anyone can clone it and start working on it and contributing to it. We can create a new fork. From

a fork, we can submit a pull request that invites people to integrate our changes. This allows a form of permissionless innovation. Indeed, it becomes possible to retrieve the code, modify it and propose a new version without ever having to interact directly with the authors.

Systems like CVS and subversion could become inefficient and take several minutes to perform certain operations. Git, in contrast, is generally efficient and fast, even for huge projects. Git is robust and does not get "corrupted" easily. However, it is not recommended to use Git for huge files such as multimedia content: Git's strength lies in text files. It should be noted that the implementation of Git has improved over time and includes sophisticated indexing techniques.

Git is often used on the command line. It is possible to use graphical clients. Services like GitHub make Git a little easier.

The basic logical unit of Git is the `commit`, which is a set of changes to multiple files. A commit contains a message, a time and an author. A `commit` includes a reference to at least one parent, except for the first `commit` which has no parent. A single `commit` can be the parent of several children: several branches can be created from a `commit` and each subsequent `commit` becomes a `child` of the initial `commit`. Furthermore, when several branches are merged, the resulting `commit` will have several parents. In this sense, the `commits` form an "acyclic directed graph".

With Git, we want to be able to refer to a `commit` in an easy way, using a unique identifier. That is, we want to have a short numeric value that corresponds to one `commit` and one `commit` only. We could assign each `commit` a version number (1.0, 2.0, etc.). Unfortunately, such an approach is difficult to reconcile with the fact that `commits` do not form a linear chain where a `commit` has one and only one parent. As an alternative, we use a hash function to compute the unique identifier. A hash function takes elements as parameters and calculates a numerical value (hash value). There are several simple hash functions. For example, we can iterate over the bytes contained in a message from a starting value `h`, by computing `h = 31 * h + b` where `b` is the byte value. For example, a message containing bytes 3 and 4 might have a hash value of `31 * (31`

`* 3) + 4` if we start `h = 0`. Such a simple approach is effective in some cases, but it allows malicious users to create collisions: it would be possible to create a fake `commit` that has the same hash value and thus create security holes. For this reason, Git uses more sophisticated hashing techniques (SHA-1, SHA-256) developed by cryptographic specialists. Commits are identified using a hash value (for example, the hexadecimal numeric value 921103db8259eb9de72f42db8b939895f5651489) which is calculated from the date and time, the comment left by the programmer, the user's name, the parents and the nature of the change. In theory, two `commits` could have the same hash value, but this is an unlikely event given the hash functions used by Git.

It may seem like Git relies on 'chance' by assuming that no two commits within the same project may have the hash value. In some sense, it does, but the probability of collision is so low that it can be neglected. Indeed, computer systems are never perfect in any case. We are constantly bombarded by cosmic rays that may flip bits in our memory systems. We are subject to random hardware failures. We accept algorithmic failure probabilities that are astronomically lower than hardware-related failure probabilities.

It's not always practical to reference a hexadecimal code. To make things easier, Git allows you to identify a commit with a label (e.g., v1.0.0). The following command will do: `git tag -a v1.0.0 -m "version 1.0.0"`. The command creates an annotated tag named `v1.0.0` in a git repository, marking the current commit with a version label. The flag `-m "version 1.0.0"` provides a descriptive message for the tag, here indicating the version.

Though tags can be any string, tags often contain sequences of numbers indicating a version. There is no general agreement among programmers on how to attribute version numbers to a version. However, tags sometimes take the form of three numbers separated by dots: MAJOR.MINOR.PATCH (for example, 1.2.3). With each new version, 1 is added to at least one of the three numbers. The first number often starts at 1 while the next two start at 0. - The first number (MAJOR) must

be increased when you make major changes to the code. The other two numbers (MINOR and PATCH) are often reset to zero. For example, you can go from version 1.2.3 to version 2.0.0. - The second number (MINOR) is increased for minor changes (for example, adding a function). When increasing the second number, the first number (MAJOR) is usually kept unchanged and the last number (PATCH) is reset to zero. - The last number (PATCH) is increased when fixing bugs. The other two numbers are not increased. There are finer versions of this convention like "semantic versioning[18]".

With Git, the programmer can have a local copy of the commit graph. They can add new `commits`. In a subsequent step, the programmer must "push" his changes to a remote directory so that the changes become visible to other programmers. The other programmers can fetch the changes using a 'pull'.

Git offers a set of advanced features that facilitate collaborative work within development teams by enabling precise traceability and efficient coordination of code changes. The `git blame` command is one of the most iconic tools in this regard. It allows for a line-by-line analysis of a file to identify the author of the last modification to each line, the date of that modification, and the associated commit. For example, running `git blame file_name` displays each line of the file alongside details such as the commit identifier (hash), the author's name, and the date, providing a detailed view of the contribution history. This feature is particularly useful in several contexts. First, it helps trace the origin of a bug or unexpected behavior in the code. If an error is detected, a developer can use `git blame` to identify who last modified the problematic line, enabling targeted questions to that person or an examination of the modification's context through the commit message. Additionally, `git blame` promotes accountability and transparency in collaborative projects, as each change is linked to a contributor. This can also be valuable during code reviews to assess individual contributions or to document a project's evolution. Beyond `git blame`, Git provides other

[18]https://semver.org

powerful collaborative tools. For instance, the `git log` command allows users to view the complete commit history, with options like `--author` to filter contributions by a specific developer or `--grep` to search for commits based on keywords in their messages.

Atomic commits

A commit should encapsulate a single, cohesive change or purpose in the codebase, making it easier to understand, track, and manage changes over time. The idea is to keep each commit focused on one specific aspect of the code, avoiding the bundling of unrelated changes. This approach enhances clarity, simplifies debugging, and facilitates collaboration by ensuring that each commit represents a logical, self-contained unit of work. We sometimes call these commits *atomic*.

An atomic commit should address one specific task, feature, or fix. It should include all relevant changes to complete that task (e.g., code, tests, and documentation). It should avoid mixing unrelated changes, such as combining a bug fix with a code style update or a new feature.

By keeping commits atomic, developers can more easily review the history of changes to understand why and how specific modifications were made. They can better revert or cherry-pick individual changes without affecting unrelated parts of the codebase. They can identify the source of bugs or issues by isolating changes to specific commits.

Suppose you are working on a project and need to add a new feature to calculate the square of a number in a Go program, reformat some code for consistency, and update the documentation. Instead of bundling all these changes into a single commit, you would create separate commits for each task. Consider the following starting point.

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

You notice the code lacks consistent indentation and formatting. You run a formatter (e.g., `gofmt`) to standardize the style. You commit this change separately.

Commit Message: Reformat code for consistent style

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

This commit only addresses code formatting and does not include functional changes.

You add a new function `square` to calculate the square of a number. This is a distinct feature, so it gets its own commit.

Commit Message: Implement square function

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

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

This commit focuses solely on adding the new function and its implementation.

You update the project's documentation (e.g., a README or code comments) to explain the new `square` function. This is a separate task, so it is committed independently.

Commit Message: Document square function in code comments

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

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

This commit only adds documentation, keeping it distinct from the implementation.

If all these changes were bundled into one commit with a vague message like "Update code," it would be harder to understand the purpose of each change when reviewing the commit history. By contrast, atomic commits make the history clear:

```
$ git log --oneline
c3f2e1d Document square function
  in code comments
a7b9c4f Implement square function
f1a2b3c Reformat code for consistent style
```

Each commit has a specific purpose, and the messages clearly describe what was done. If a bug is found in the `square` function, you can use `git blame` or `git bisect` to pinpoint the commit `a7b9c4f` and investigate or revert it without affecting the formatting or documentation changes.

A poor commit might look like the following example.

Commit Message: Add square function and fix stuff

```
// add returns the sum of two integers
func add(a, b int) int {
    return a + b
}
// square returns the square of an integer
func square(n int) int {
    return n * n
}
```

This commit combines formatting, feature implementation, and documentation into one. If you need to revert the feature but keep the

documentation, it becomes difficult because the changes are entangled. Additionally, the vague message doesn't clearly explain what *fix stuff* entails, reducing traceability. By using atomic commits, you create a more maintainable and transparent version history, making collaboration and debugging more efficient.

Branches in Git

In Git, a major innovation compared to previous version control systems like CVS or Subversion is the concept of branches. A branch in Git represents an independent line of development, allowing multiple versions of the same project to evolve in parallel. Unlike a linear model where all changes are applied to a single main version, branches enable developers to work on separate copies of the code without affecting the main branch, often called `main` or `master`.

A branch can be created with the command `git branch <branch-name>` to define a new branch, followed by `git switch <branch-name>` to switch to it. Once on a branch, changes made are isolated from other branches, allowing experimentation or fixes without risking the main codebase. A developer can work on an experimental feature with no guarantee of success. By using a dedicated branch, they can test new ideas and merge with the main branch only if the result is satisfactory. In some projects, access to the main branch may be restricted to a specific group of developers. Other contributors work on secondary branches, and their changes are merged after a code review by a maintainer. A branch can be created to isolate and fix a specific bug, particularly in an older version of the software still used by some users. A branch can be dedicated to maintaining a previous version of the code, receiving updates or fixes without being merged into the main branch.

Merging branches, done with the command `git merge <branch-name>`, integrates changes from one branch into another. For example, if a `feature` branch contains a new functionality, it can be merged into the `main` branch once completed. However, conflicts may arise if competing

changes have been made to the same file in different branches. For instance, if two developers modify the same line of code differently, Git will report a conflict that must be resolved manually by editing the file to choose or combine the changes.

Consider a simple example to illustrate a potential conflict. Suppose an initial file contains the following code.

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

Developer A, on a branch `branch-a`, adds a function `fa`.

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

Developer B, on a branch `branch-b`, adds a function `fb`.

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

When merging `branch-a` and `branch-b` into `main`, Git can usually combine the changes automatically since they affect different parts of the file. The result would be:

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

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

However, if both developers modify the same line, Git will report a conflict, and the developer must intervene to resolve it by choosing which version to keep or combining the changes.

Branches are often used in combination with services like GitHub, where developers can submit *pull requests* to propose integrating their changes. This facilitates large-scale collaboration, as anyone can clone a public repository, create a branch, make changes, and propose their integration without direct interaction with the project maintainers.

To list all branches in a repository, use `git branch`. To delete an unnecessary branch, run `git branch -d <branch-name>`. Git also ensures that branches deleted locally do not affect remote repositories, providing additional safety.

Conclusion

Version control in computing is a sophisticated approach that has benefited from many years of work. It is possible to store multiple versions of the same file at low cost and navigate from one version to another quickly.

If you develop code without using a version control tool like Git or the equivalent, you are bypassing proven practices. It's likely that if you want to work on complex projects with multiple programmers, your productivity will be much lower without version control.

Exercises for Chapter 2

Problem 1.

Design a function that represents the difference between two files in a concise way. For example, if only two lines are changed within a large file, it should be possible to store approximately only two lines of text.

Problem 2.

Given your solution to the first problem. Then write a function that can take the original file and your concise description and rebuild the second file.

Problem 3.

Read the Git tutorial[19]. Install Git on your machine, create a new directory, make changes to it, create a new branch, make changes to it, and merge the two branches.

[19] https://git-scm.com/docs/gittutorial

Chapter 3

Computer programming starts with the organization of the data into data structures. In almost all cases, we work with strings or numbers. It is critical to understand these building blocks to become an expert programmer.

Words

We often organize data using fixed blocks of memory. When these blocks are relatively small (e.g., 8 bits, 16 bits, 32 bits, 64 bits), we commonly call them 'words'.

The notion of 'word' is important because processors do not operate over arbitrary data types. For practical reasons, processors expect data to fit in hardware registers having some fixed size (usually 64-bit registers). Most modern processors accommodate 8-bit, 16-bit, 32-bit and 64-bit words with fast instructions. It is typical to have the granularity of the memory accesses to be no smaller than the 'byte' (8 bits) so bytes are, in a sense, the smallest practical words.

Variable-length data structures like strings might be made of a variable number of words. Historically, strings have been made of lists of bytes, but other alternatives are common (e.g., 16-bit or 32-bit words).

Boolean values

The simplest type is probably the Boolean type. A Boolean value can take either the false or the true value. Though a single bit suffices to represent a Boolean value, it is common to use a whole byte (or more). We can negate a Boolean value: the true value becomes the false value, and conversely. There are also binary operations: - The result of the OR operation between two Boolean values is false if and only if both inputs are false. The OR operation is often noted `|`. E.g., `1 | 0 == 1` where we use the convention that the symbol `==` states the equality between two values. - The result of the AND operation between two Boolean values is true if and only if both inputs are true. The AND operation is often noted `&`. E.g., `1 & 1 == 1`. - The result of the XOR operation is true if and only the two inputs differ in value. The XOR operation is often noted `^`. E.g., `1 ^ 1 == 0`. - The result of the AND NOT operation between two Boolean values is true if and only if the first Boolean value is true and the second one is false.

Integers

Integer data types are probably the most widely supported in software and hardware, after the Boolean types. We often represent integers using digits. E.g., the integer 1234 has 4 decimal digits. By extension, we use 'binary' digits, called bits, within computers. We often write an integer using the binary notation using the `0b` prefix. E.g., the integer `0b10` is two, the integer `0b10110` is equal to `2^1+2^2+2^4` or 22. After the prefix `0b`, we enumerate the bit values, starting with the most significant bit. We may also use the hexadecimal (base 16) notation with the `0x` prefix: in that case, we use 16 different digits in the list `0, 1, 2, 3,..., 9, A, B, C, D, E, F`. These digits have values `0, 1, 2, 3,..., 9, 10, 11, 12, 13, 14, 15`. For digits represented as letters, we may use either the lower or upper cases. Thus the number `0x5A` is equal to `5 * 16 + 10` or 90 in decimal. The hexadecimal notation is convenient when

working with binary values: a single digit represents a 4-bit value, two digits represent an 8-bit value, and so forth.

We might count the number of digits of an integer using the formula `ceil(log(x+1))` where the logarithm is the in the base you are interested in (e.g., base 2) and where `ceil` is the *ceiling* function: `ceil(x)` returns the smallest integer no smaller than `x`. The product between an integer having `d1` digits and an integer having `d2` digits has either `d1+d2-1` digits or `d1+d2` digits. To illustrate, let us consider the product between two integers having three digits. In base 10, the smallest product is 100 times 100 is 10,000, so it requires 5 digits. The largest product is 999 times 999 or 998,001 so 6 digits.

For speed or convenience, we might use a fixed number of digits. Given that we work with binary computers, we are likely to use binary digits. We also need a way to represent the sign of a number (negative and positive).

Unsigned integers

Possibly the simplest number type is the unsigned integer where we use a fixed number of bits used to represent non-negative integers. Most processors support arithmetic operations over unsigned integers. The term 'unsigned' in this instance is equivalent to 'non-negative': the integers can be zero or positive.

We can operate on binary integers using bitwise logical operations. For example, the bitwise AND (noted &) between `0b101` and `0b1100` is `0b100`: `0b101 & 0b1100 == 0b100`. The bitwise OR (noted |) is `0b1101`. The bitwise XOR (exclusive OR), noted ^, is `0b1001 : 0b101 ^ 0b1100 == 0b1001`.

The powers of two (1, 2, 4, 8,...) are the only numbers having a single 1-bit in their binary representation (`0b1`, `0b10`, `0b100`, `0b1000`, etc.). The numbers preceding powers of two (1,3,7,...) are the numbers made of consecutive 1-bits in the least significant positions (`0b1`, `0b11`, `0b111`,

`0b1111`, etc.). A unique characteristic of powers of two is that their bitwise AND with the preceding integer is zero: e.g., 4 AND 3 is zero, 8 AND 7 is zero, and so forth.

In the Go programming language, for example, we have 8-bit, 16-bit, 32-bit and 64-bit unsigned integer types: `uint8`, `uint16`, `uint32`, `uint64`. They can represent all numbers from 0 up to (but not including) 2 to the power of 8, 16, 32 and 64. For example, an 8-bit unsigned integer can represent all integers from 0 up to 255 inclusively.

Because we choose to use a fixed number of bits, we therefore can only represent a range of integers. The result of an arithmetic operation may exceed the range (an *overflow*). For example, 255 plus 2 is 257: though both inputs (255 and 2) can be represented using 8-bit unsigned integers, the result exceeds the range.

Regarding multiplications, the product of two 8-bit unsigned integers is at most 65025 which can be represented by a 16-bit unsigned integer. It is always the case that the product of two `n`-bit integers can be represented using `2n` bits. The converse is untrue: a given `2n`-bit integer is not the product of two `n`-bit integers. As `n` becomes large, only a small fraction of all `2n`-bit integers can be written as the product of two `n`-bit integers, a result first proved by Erdős.

Typically, arithmetic operations are "modulo" the power of two. That is, everything is as if we did the computation using infinite-precision integers and then we only kept the (positive) remainder of the division by the power of two.

Let us elaborate. Given two integers `a` and `b` (`b` being non-zero), there are unique integers `d` and `r` where `r` is in `[0,b)` such that `a = d * b + r`. The integer `r` is the remainder and the integer `d` is the quotient. In Go, the quotient is obtained with `a/b` whereas the remainder is given by `a%b`.

Euclid's division lemma tells us that the quotient and the remainder exist and are unique. We can check uniqueness. Suppose that there is another such pair of integers (`d'` and `r'`), `a = d' * b + r'`. We can

check that if `d'` is equal to `d`, then we must have that `r'` is equal to `r`, and conversely, if `r'` is equal to `r`, then `d'` is equal to `d`. Suppose that `r'` is greater than `r` (if not, just reverse the argument). Then, by subtraction, we have that `0 = (d'-d)*b + (r'-r)`. We must have that `r'-r` is in `[0,b)`. If `d'-d` is negative, then we have that `(d-d')*b = (r'-r)`, but that is impossible because `r'-r` is in `[0,b)` whereas `(d-d')*b` is greater or equal than `b`. A similar argument works when `d'-d` is positive.

In our case, the divisor (`b`) is a power of two. When the numerator (`a`) is positive, then the remainder amounts to a selection of the least significant bits. For example, the remainder of the division of 65 (or `0b1000001`) with 64 is 1.

When considering unsigned arithmetic, it often helps to think that we keep only the least significant bits (8, 16, 32 or 64) of the final result. Thus if we take 255 and we add 2, we get 257, but as an 8-bit unsigned integer, we get the number 1. Thus, using integers of the type `uint8`, we have that `255 + 2` is 1 (`255 + 2 == 1`). The power of two itself is zero: 256 is equal to zero as an `uint8` integer. If we subtract two numbers and the value would be negative, we effectively 'wrap' around: 10 - 20 in `uint8` arithmetic is the positive remainder of (-10) divided by 256 which is 246. Another way to think of negative numbers is that we can add the power of two (say 256) as many times as needed (size its value is effectively zero) until we get a value that is between 0 and the power of two. Thus if we must evaluate `1-5*250` as an 8-bit integer, we take the result (-1249) and we add 256 as many times as needed: we have that `-1249+5*256` is 31, a number between 0 and 256. Thus `1-5*250` is 31 as an unsigned 8-bit number.

We have that `0-1`, as an 8-bit number, is 255 or `0b11111111`. We have that `0-2` is 254 and that `0-3` is 253, and so forth.

Consider the set of integers.

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

As 8-bit integers, they are mapped to to the following.

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

Multiplication by a power of two is equivalent to shifting the bits left, possibly losing the leftmost bits. For example, 17 is `0b10001`. Multiplying it by 4, we get `0b1000100` or 68. If we were to multiply 68 by 4, we would get `0b100010000` or, as an 8-bit integer, `0b10000`. That is, as 8-bit unsigned integers, we have that `17 * 16` is 16. Thus we have that `17 * 16 == 1 * 16`.

The product of two non-zero integers may be zero. For example, `16*16` is zero as an 8-bit integer. It happens only when both integers are divisible by two. The product of two odd integers must always be odd.

We say that two numbers are 'coprime' if their largest common divisor is 1. Odd integers are coprime with powers of two. Even integers are never coprime with a power of two.

When multiplying a non-zero integer by an odd integer using finite-bit arithmetic, we never get zero. Thus, for example, `3 * x` as an 8-bit integer is zero if and only if `x` is zero when using fixed-bit unsigned integers. It means that `3 * x` is equal to `3 * y` if and only if `x` and `y` are equal. Thus we have that the following Go code will print out all values from 0 to 255, without repetition:

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

In mathematics, a permutation is a bijection from a set to itself, that is, a function that rearranges the elements of that set without adding or removing any. When you shuffle the cards in a deck, you permute them: no new card is added and no card is lost. Multiplying integers by an odd integer using finite-bit arithmetic permutes them.

If you consider powers of an odd integer, you similarly never get a zero result. However, you may eventually get the power to be one. For example, as an 8-bit unsigned integer, 3 to the power of 64 is 1. This number (64) is sometimes called the 'order' of 3. Since this is the smallest

exponent so that the result is one, we have that all 63 preceding powers give distinct results. We can show this result as follows. Suppose that 3 raised to the `p` is equal to 3 raised to the power `q`, and assume without loss of generality that `p>q`, then we have that `3` to the power of `p-q` must be 1, by inspection. And if both `p` and `q` are smaller than 64, then so must b `p-q`, a contradiction. Further, we can check that the powers of an odd integer repeat after the order is reached: we have that 3 to the power 64 is 1, 3 to the power of 65 is 3, 3 to the power of 66 is 9, and so forth. It follows that the order of any odd integer must divide the power of two (e.g., 256).

How large can the order of an odd integer be? We can check that all powers of an odd integer must be odd integers and there are only 128 distinct 8-bit integers. Thus the order of an 8-bit odd integer can be at most 128. Conversely, Euler's theorem tells us that any odd integer to the power of the number of odd integers (e.g., 3 to the power 128) must be one. Because the values of the power of an odd integer repeat cyclicly after the order is reached, we have that the order of any odd integer must divide 128 for 8-bit unsigned integers. Generally, irrespective of the width in bits of the words, the order of an odd integer must be a power of two.

Given two non-zero unsigned integers, `a` and `b`, we would expect that `a+b>max(a,b)` but it is only true if there is no overflow. When and only when there is an overflow, we have that `a+b<min(a,b)` using finite-bit unsigned arithmetic. We can check for an overflow with either conditions: `a+b<a` and `a+b<b`.

Typically, one of the most expensive operations a computer can do with two integers is to divide them. A division can require several times more cycles than a multiplication, and a multiplication is in turn often many times more expensive than a simple addition or subtraction. However, the division by a power of two and the multiplication by a power of two are inexpensive: we can compute the integer quotient of the division of an unsigned integer by shifting the bits *right*. For example, the integer 7 (0b111) divided by 2 is 0b011 or 3. We can further divide 7 (0b111) by 4 to get 0b001 or 1. The integer remainder is given by selecting the bits

that would be shifted out: the remainder of 7 divided by 4 is 7 AND 0b11 or 0b11. The remainder of the division by two is just the least significant bit. Even integers are characterized by having zero as the least significant bit. Similarly, the multiplication by a power of two is just a left shift: the integer 7 (0b111) multiplied by two is 14 (0b1110). More generally, an optimizing compiler may produce efficient code for the computation of the remainder and quotient when the divisor is fixed. Typically, it involves at least a multiplication and a shift[20].

Given an integer `x`, we say that `y` is its multiplicative inverse if `x * y == 1`. We have that every odd integer has a multiplicative inverse because multiplication by an integer creates a permutation of all integers. We can compute this multiplicative inverse using Newton's method. That is, we start with a guess and from the guess, we get a better one, and so forth, until we naturally converge to the right value. So we need some formula `f(y)`, so that we can repeatedly call `y = f(y)` until `y` converges. A useful recurrence formula is `f(y) = y * (2 - y * x)`. You can verify that if `y` is the multiplicative inverse of `x`, then `f(y) = y`. Suppose that `y` is not quite the inverse, suppose that `x * y = 1 + z * p` for some odd integer `z` and some power of two `p`. If the power of two is (say) 8, then it tells you that `y` is the multiplicative inverse over the first three bits. We get `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`. We can see from this result that if `y` is the multiplicative inverse over the first `n` bits, then f(y) is the multiplicative inverse over `2n` bits. That is, if `y` is the inverse "for the first `n` bits", then `f(y)` is the inverse "for the first `2n` bits". We double the precision each time we call the recurrence formula. It means that we can quickly converge on the inverse.

What should our initial guess for `y` be? If we use 3-bit words, then every number is its inverse. So starting with `y = x` would give us three bits of accuracy, but we can do better: `( 3 * x ) ^ 2` provides 5 bits of accuracy. The following Go program verifies the claim:

[20] https://arxiv.org/abs/1902.01961

```
package main

import "fmt"

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

Observe how we capture the 5 least significant bits using the expression `&0b11111`: it is a bitwise logical AND operation.

Starting from 5 bits, the first call to the recurrence formula gives 10 bits, then 20 bits for the second call, then 40 bits, then 80 bits. So, we need to call our recurrence formula 2 times for 16-bit values, 3 times for 32-bit values and 4 times for 64-bit values. The function `FindInverse64` computes the 64-bit multiplicative inverse of an odd integer:

```
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
}
```

We have that `FindInverse64(271) * 271 == 1`. Importantly, it fails if the provided integer is even.

We can use multiplicative inverses to replace the division by an odd integer with a multiplication. That is, if you precompute `FindInverse64(3)`, then you can compute the division by three for any multiple of three by computing the product: e.g., `FindInverse64(3) * 15 == 5`.

When we store multi-byte values such as unsigned integers in arrays of bytes, we may use one of two conventions: little- and big-endian. The little- and big-endian variants only differ by the byte order: we either start with the least significant bytes (little endian) or by the most significant bytes (big endian). Let us consider the integer 12345. An an hexadecimal value, it is 0x3039. If we store it as two bytes, we may either store it as the byte value 0x30 followed by the byte value 0x39 (big endian), or by the reverse (0x39 followed by 0x30). Most modern systems default on the little-endian convention, and there are relatively few big-endian systems. In practice, we rarely have to be concerned with the endianness of our system.

Signed integers and two's complement

Given unsigned integers, how do we add support for signed integers? At first glance, it is tempting to reserve a bit for the sign. Thus if we have 32 bits, we might use one bit to indicate whether the value is positive or negative, and then we can use 31 bits to store the absolute value of the integer.

Though this sign-bit approach is workable, it has downsides. The first obvious downside is that there are two possible zero values: `+0` and `-0`. The other downside is that it makes signed integers wholly distinct values as compared to unsigned integers: ideally, we would like hardware instructions that operate on unsigned integers to 'just work' on signed integers.

Thus modern computers use two's complement notation to represent signed integers. To simplify the exposition, we consider 8-bit integers. We represent all positive integers up to half the range (127 for 8-bit words) in the same manner, whether using signed or unsigned integers. Only when the most significant bit is set, do we differ: for the signed integers, it is as if the unsigned value derived from all but the most significant bit is subtracted by half the range (128). For example, as an 8-bit signed value, 0b11111111 is -1. Indeed, ignoring the most significant bit, we have 0b1111111 or 127, and subtracting 128, we get -1.

Binary	unsigned	signed
0b00000000	0	0
0b00000001	1	1
0b00000010	2	2
0b01111111	127	127
0b10000000	128	-128
0b10000001	129	-127
0b11111110	254	-2
0b11111111	255	-1

Observe how odd integers are mapped to odd integers and vice versa. Indeed, 254 is interpreted as -2, 255 as -1.

In Go, you can 'cast' unsigned integers to signed integers, and vice versa: Go leaves the binary values unchanged, but it simply reinterprets the value as unsigned and signed integers. If we execute the following code, we have that `x==z`:

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

Conveniently, whether we compute the multiplication, the addition or the subtraction between two values, the result is the same (in binary)

whether we interpret the bits as a signed or unsigned value. Thus we can use the same hardware circuits.

A downside of the two's complement notation is that the smallest negative value (-128 in the 8-bit case) cannot be safely negated. Indeed, the number 128 cannot be represented using 8-bit signed integers. This asymmetry is unavoidable because we have three types of numbers: zero, negative values and positive values. Yet we have an even number of binary values.

Like with unsigned integers, we can shift (right and left) signed integers. The left shift works like for unsigned integers at the bit level. We have that

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

However, right shift works differently for signed and unsigned integers. For unsigned integers, we shift in zeroes from the left; for signed integers, we either shift in zeroes (if the integer is positive or zero) or ones (if the integer and negatives). We illustrate this behaviour with an example:

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

When a signed integer is positive, then dividing by a power of two or shifting right has the same result (`10/4 == (10>>2)`). However, when the integer is negative, it is only true when the negative integer is divisible by the power of two. When the negative integer is not divisible by the power of two, then the shift is smaller by one than the division, as illustrated by the following code:

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

In Go, the remainder of the division between a negative integer and a positive integer is zero or negative. In effect, we have that `(-a)/b == -(a/b)` and `(-a)%b == -(a%b)`. For example, we have `(-5)//2 == -(5//2)` and `-1%2 == -(1%2) == -1`. For a positive integer, we can check if it is odd with the expression `x%2 == 1`, but this expression does not work anymore with negative integers since the remainder of the division will be negative: `-3%2 == -1%`. We can check if an integer is odd by checking the value of the least significant bit: `x&1==1` for odd numbers, whether they are negative or positive.

For positive or unsigned integers, we have that `(2*x + 1)/2==x`. However, for negative integers, it is false: `(2*-1 + 1)/2 == (-1)/2 == 0`. We have instead that `(2*x + 1)/2 == x+1`. More generally, we have that `(2*x - 1)/2 == x` when `x` is negative, and `(2*x + 1)/2 == x` when `x` is positive. The following program illustrates the observation:

```
package main

import "fmt"

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

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

However, we have the identity `2*(x>>1)+(x&1)` whether `x` is positive or negative.

Floating-point numbers

On computers, real numbers are typically approximated by binary floating-point numbers: a fixed-width integer `m` (the *significand*) multiplied by 2 raised to an integer exponent `p`: `m * 2**p` where `2**p` represents the number two raised to the power `p`. A sign bit is added so that both a positive and negative zero are available. Most systems today follow the IEEE 754 standard which means that you can get consistent results across programming languages and operating systems. Hence, it does not matter very much if you implement your software in C++ under Linux whereas someone else implements it in C# under Windows: if you both have recent systems, you can expect identical numerical outcomes when doing basic arithmetic and square-root operations.

A positive *normal* double-precision floating-point number is a binary floating-point number where the 53-bit integer `m` is in the interval `[2**52,2**53)` while being interpreted as a number in `[1,2)` by virtually dividing it by `2**52`, and where the 11-bit exponent `p` ranges from `-1022` to `+1023`. Thus we can represent all values between `2**-1022` and up to but not including `2**1024`. Some values smaller than `2**-1022` can be represented as *subnormal* values: they use a special exponent code which has the value `2**-1022` and the significand is then interpreted as a value in the interval $[0, 1)$.

In Go, a `float64` number can represent all decimal numbers made of a 15-digit significand from approximately `-1.8 * 10**308` to `1.8 *10**308`. The reverse is not true: it is not sufficient to have 15 digits of precision to distinguish any two floating-point numbers: we may need up to 17 digits.

The `float32` type is similar. It can represent all numbers between `2**-126` up to, but not including, `2**128`; with special handling for some numbers smaller than `2**-126` (subnormals). The `float32` type can represent exactly all decimal numbers made of a 6-digit decimal

significand but 9 digits are needed in general to identify uniquely a number.

Floating-point numbers also include the positive and negative infinity, as well as a special not-a-number value. They are identified by a reserved exponent value.

Numbers are typically serialized as decimal numbers in strings and then parsed back by the receiver. However, it is generally impossible to convert decimal numbers into binary floating-point numbers: the number `0.2` has no exact representation as a binary floating-point number. However, you should expect the system to choose the best possible approximation: `7205759403792794 * 2**-55` as a `float64` number (or about `0.20000000000000001110`). If the initial number was a `float64` (for example), you should expect the exact value to be preserved: it will work as expected in Go.

The positive and negative zeroes can be difficult to distinguish because they are considered to be equal values. Thankfully, the `math.Signbit` function returns `true` only when the value is negative. Consider the following program:

```
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)
}
```

It will output:

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

One might expect that multiplying any value by zero would result in the value zero, and that adding zero would never change the sign of a value. Unfortunately, we have that `0 * Inf` is `NaN`(not-a-number), a special value that can be consider as an error code. We have that `-0.0 + 0.0` is `0.0` so that adding `0.0` to `-0.0` changes its sign. The following program illustrates these identities:

```
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 / zero) * zero)
}
```

It can be inconvenient to represent binary floating-point numbers as decimal values since there is no exact conversion in general. For this purpose, Go and other programming languages allow us to print out

number in hexadecimal floating-point format. The hexadecimal floating-point notation is supported in the C (C99), C++ (C++17), Swift, Java, Julia and Go programming languages. As in the usual hexadecimal notation for integers, we start the string with 0x followed by the significand in hexadecimal form. Each hexadecimal character (0–9, A–F) represents 4 bits (a nibble). We append the suffix p followed by the exponent (e.g., p4 or p-4). Optionally, we can add an hexadecimal point in the significand. With a decimal point, we interpret the decimal fraction by dividing it by the appropriate power of ten. E.g, we write 1.45 = 145/100. The hexadecimal point works similarly. Thus 0x1.FCp17 means 0x1FC/256 times two to the power 17 or 260096 where we divide 0x1FC by 256 because there are two nibbles (16 * 16) after the binary point. Each nibble after the point requires a division by 16. When the value is a normal 64-bit floating-point number, the significand can be expressed as a most significant 1 followed by up to 52 bits, or 13 hexadecimal character. Thus 9 000 000 000 000 000 can be written as 0x1.ff973cafa8p+52. Unsurprisingly, the number 1 can be written as 0x1p0. The number 0.2 requires more care. We can approximate it most closely as a 64-bit floating-point number as 7205759403792794 divided by 2 to the power 55. The result is then slightly larger than 0.2: it is exactly .200000000000000038857805861880478914827108383178710937 5. It is not possible to exactly match 0.2 using binary floating-point numbers. The number 7205759403792794 is 0x1999999999999a in hexadecimal, so that 0.2 is approximated by 0x1.999999999999ap-03. The following program prints 0x1.999999999999ap-03 and 0x1p+00. Indeed, the function `strconv.FormatFloat` can translate any number into an hexadecimal floating-point number: in this case, we pass the parameters 'x' for hexadecimal, -1 to indicate that we want full accuracy and 64 to indicate that we are using a 64-bit representation. Importantly, the value 0x1.999999999999ap-03 is an exact representation of what is stored in your software when you input 0.2 and represent it as a standard binary floating-point number.

```
package main
```

```
import (
  "fmt"
  "strconv"
)

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

}
```

Arrays

In Go, as in most programming languages, you can organize data in an array. For example, you can create an array of 3 integers, like this:

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

In this case, we provide Go with the contents of the array with a list of integers. If we omit this list of integers, the elements are initialized with the value zero. Arrays can be created with different types. As long as the array is relatively small and its size is known when writing the code, one can easily use the declaration syntax containing a fixed size.

In Go, passing values to a function is usually done *by value* which means that Go makes a copy. The following function makes a copy of the three integer values on each call:

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

In Go, we prefer to use the notion of *slice*: an *slice* is used as an array, but its size is not fixed. In memory, it is only a reference to a section of an array which can be implicit. You can create an array with a size fixed at runtime with the `make` function:

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

The `make` function initializes the values with a zero. A *hidden* array is created. When passing an *slice* to a function, the data itself is not copied, just the reference:

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

The `append` function allows to add elements to a *slice*. This function adds the value '1' and returns the new *slice* including the additional value:

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

Calling the function `g3` does not change the initial *slice*. So we can have several *slice* sharing the same data. For example, the following array creates a *slice* `z` containing a single element (the value `3`):

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

Strings

One of the earliest string standards is ASCII: it was first specified in the early 1960s. The ASCII standard is still popular. Each character is a byte, with the most significant bit set to zero. There are therefore only 128 distinct ASCII characters. It is often sufficient for simple tasks like programming. Unfortunately, the ASCII standard could only ever represent up to 128 characters: far less than needed.

The following program prints all printable ASCII characters.

```
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)+"'")
  }
}
```

The program contains a for loop, which iterates from 32 to 126 (inclusive). We declare a variable buffer, which is a slice of bytes. A slice is a dynamically sized sequence of elements of the same type. A byte is an alias for uint8, which is an unsigned 8-bit integer. The make function creates a slice with a given length and capacity. In this case, the length and capacity are both 1, which means the slice has only one element. We assigns the value of i converted to a byte to the first element of the buffer slice. The byte function converts an integer to a byte, truncating if necessary. The eighth line prints the value of i, followed by a single quote, followed by the string representation of the buffer slice, followed by another single quote, followed by a newline. The fmt.Println function prints its arguments to the standard output, separated by spaces and ending with a newline. The string function converts a slice of bytes to a string, interpreting each byte as a UTF-8 encoded character. The output of the program is a table of ASCII codes and their corresponding characters, from 32 (space) to 126 (tilde). There are characters from 0 to 32 but they are control characters used for line endings and other special purposes. The last ASCII character (127) is also a control character (DEL).

Many diverging standards emerged for representing characters in software. The existence of multiple incompatible formats made the production of interoperable localized software challenging.

Engineers developed Unicode in the late 1980s as an attempt to provide a universal standard. Initially, it was believed that using 16 bits per character would be sufficient, but this belief was wrong. The Unicode standard was extended to include up to 1,114,112 characters. Only a small fraction of all possible characters have been assigned, but more are assigned over time with each Unicode revision. The Unicode standard is an extension of the ASCII standard: the first 128 Unicode characters match the ASCII characters.

Due to the original expectation that Unicode would fit in 16-bit space, a format based on 16-bit words (UTF-16) format was published in 1996. It may use either 16-bit or 32-bit per character. The UTF-16 format was adopted by programming languages such as Java, and became a default under Windows. Unfortunately, UTF-16 is not backward compatible with ASCII at a byte level. An ASCII-compatible format was proposed and formalized in 2003: UTF-8. Over time, UTF-8 became widely used for text interchange formats such as JSON, HTML or XML. Programming languages such as Go, Rust and Swift use UTF-8 by default. Both formats (UTF-8 and UTF-16) require validation: not all arrays of bytes are valid. The UTF-8 format is more expensive to validate.

ASCII characters require one byte with UTF-8 and two bytes with UTF-16. The UTF-16 format can represent all characters, except for the supplemental characters such as emojis, using two bytes. The UTF-8 format uses two bytes for Latin, Hebrew and Arabic alphabets, three bytes for Asiatic characters and 4 bytes for the supplemental characters.

UTF-8 encodes values in sequences of one to four bytes. We refer to the first byte of a sequence as a leading byte; the most significant bits of the leading byte indicates the length of the sequence: - If the most significant bit is zero, we have a sequence of one byte (ASCII). - If the three most significant bits are 0b110, we have a two-byte sequence. - If the four most significant bits are 0b1110, we have a three-byte sequence. - Finally, if the

five most significant bits are 0b11110, we have a four-byte sequence. All bytes following the leading byte in a sequence are continuation bytes, and they must have 0b10 as their most significant bits. Except for the required most significant bits, the numerical value of the character (between 0 to 1,114,112) is stored by starting with the most significant bits (in the leading byte) followed by the less significant bits in the other continuation bytes.

The following Go program will print all Unicode characters that use two bytes in 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)+"'")
    }
  }
}
```

This Go program generates and displays all Unicode characters encoded in UTF-8 using two bytes, corresponding to codepoints from U+0080 to U+07FF. It employs two nested loops: the outer loop (`i`) iterates over binary values from 0b00010 to 0b11111 (2 to 31), representing the 5 significant bits of the first byte, and the inner loop (`j`) iterates from 0 to 0b111111 (0 to 63), representing the 6 bits of the second byte. For each combination, a two-byte buffer is created. The first byte is formed by combining `i` with the prefix 0b11000000 (following the UTF-8 two-byte

format, 110xxxxx), and the second byte combines `j` with 0b10000000 (the continuation prefix, 10xxxxxx). The program then displays the Unicode codepoint calculated as `(i<<6)+j` (shifting `i` by 6 bits and adding `j`) and the corresponding character obtained by converting the buffer to a string. This allows it to display all 1,920 two-byte UTF-8 characters.

The following program illustrates how within the same UTF-8 string, some characters may need one byte ('L') while others need two bytes ('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)
}
```

In the UTF-16 format, characters in 0x0000-0xD7FF and 0xE000-0xFFFF are stored as single 16-bit words. Characters in the range 0x010000 to 0x10FFFF require two 16-bit words called a surrogate pair. The first word in the pair is in the range 0xd800 to 0xdbff whereas the second word is in the range from 0xdc00 to 0xdfff. The character value is made of the 10 least significant bits of the two words, using the second word as least significant, and adding 0x10000 to the result. There are two types of UTF-16 format. In the little-endian variant, each 16-bit word is stored

using the least significant bits in the first byte. The reverse is true in the big-endian variant.

When using ASCII, it is relatively easy to access the characters in random order. For UTF-16, it is possible if we assume that there are no supplemental characters, but since some characters might require 4 bytes while other 2 bytes, it is not possible to go directly to a character by its index without accessing the previous content. The UTF-8 is similarly not randomly accessible in general.

Software often depends on the chosen locale: e.g., US English, French Canadian, and so forth. Sorting strings is locale-dependent. It is not generally possible to sort strings without knowing the locale. However, it is possible to sort strings lexicographically as byte sequences (UTF-8) or as 16-bit word sequences (UTF-16). When using UTF-8, the result is then a string sort based on the characters' numerical value.

Pointers

At least conceptually, data in software is stored at a location in memory. Thus, each variable, each array element, is associated with a memory address. This abstraction is useful but not entirely accurate in practice. For example, a pointer can hold an invalid address. Nevertheless, a pointer is a useful concept in software programming when used with caution. The general syntax involves placing the ampersand character before a value (`&`) to obtain a pointer, and the resulting pointer type is created using the asterisk prefix `*`. Conversely, the asterisk prefix allows access to the pointed value. The following program outputs `1`:

```
package main
import (
    "fmt"
)
func f(v *int) {
    // f receives a pointer to an integer value
    *v = 1 // sets the pointed value to 1
```

```
}
func main() {
    x := make([]int, 10)
    f(&x[0]) // passes a pointer to the first element
    fmt.Println(x[0])
}
```

In the program, the function `f` modifies the value of `x[0]` using a pointer, avoiding passing a copy of the value. This is particularly useful for large data structures, such as a struct containing many fields, where copying the entire data would be costly in terms of performance. For example, in Go, a struct like `type Person struct { Name string; Age int }` can be passed by pointer to a function to modify its fields without duplicating the entire struct. However, pointers introduce risks, notably null pointers. In Go, an uninitialized pointer has the value `nil`, meaning it does not point to any valid memory address. If a function attempts to dereference a `nil` pointer, as in `*ptr = 5` where `ptr` is `nil`, the program will crash with a runtime error (panic). To avoid this, it is common to check if a pointer is `nil` before using it. For example, a function handling a pointer might include a check like `if v == nil { return }` to handle cases where no valid pointer is provided. For example, the following program will terminate with a fatal error.

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

To illustrate another difficulty, we will create a situation where a pointer references a value that is no longer accessible due to variable scope and memory management in Go.

```
package main
import "fmt"
```

```
func createPointer() *int {
    x := 42
    return &x
}
func main() {
    ptr := createPointer()
    fmt.Println(ptr)
    fmt.Println(*ptr)
}
```

In this program, the function `createPointer` declares a local variable `x` and returns its address as a pointer. However, since `x` is a local variable, it is allocated on the stack, and its scope is limited to the `createPointer` function. Once the function ends, the memory associated with `x` may be reused or considered invalid by Go's garbage collector. The pointer `ptr` in `main` becomes a dangling pointer, as it references a memory address that is no longer guaranteed to be valid. In Go, attempting to dereference `ptr` with `*ptr` in `main` may cause undefined behavior or, in some cases, a runtime error (panic), as the pointed memory is no longer accessible. You may notice that, although the program is incorrect, it may run without an apparent error. This is why errors caused by improper use of pointers are so challenging. A crash can occur randomly. To illustrate a more practical scenario, consider a case where a function returns a pointer to a locally allocated struct.

```
package main
import "fmt"
type Person struct {
    Name string
    Age  int
}
func createPerson() *Person {
    p := Person{Name: "Alice", Age: 30}
    return &p
}
```

```
func main() {
    ptr := createPerson()
    fmt.Println(ptr)
    fmt.Println(ptr.Name) // Undefined behavior or panic
}
```

Here, `createPerson` creates a `Person` struct locally and returns a pointer to it. Since `p` is a local variable, its memory may be freed or reused after the function ends. When `main` attempts to access `ptr.Name`, the program risks crashing or displaying corrupted data, as the address pointed to by `ptr` is no longer guaranteed to be valid. To avoid dangling pointers, a good practice in Go is to allocate memory on the heap using `new` or create objects that persist beyond the function's scope. Here is a corrected version of the second example.

```
package main
import "fmt"
type Person struct {
    Name string
    Age  int
}
func createPerson() *Person {
    p := new(Person)
    p.Name = "Alice"
    p.Age = 30
    return p
}
func main() {
    ptr := createPerson()
    fmt.Println(ptr)
    fmt.Println(ptr.Name)
}
```

In this version, `new(Person)` allocates the struct on the heap, and Go's garbage collector ensures the memory remains valid as long as `ptr` refer-

ences it. This eliminates the risk of a dangling pointer, as the memory is not freed prematurely. ## Structs, interfaces, and methods In Go, complex data structures are often built using structs, interfaces, and methods, which provide a flexible and modular approach to organizing and manipulating data. A struct allows grouping multiple fields of different types into a single entity. For example, to represent a node in a linked list, one could define:

```
package main
import "fmt"
type Node struct {
    Value  int
    Next   *Node
}
func main() {
    n := Node{Value: 10, Next: nil}
    fmt.Println(n.Value)
}
```

Here, `Node` is a struct representing a node with an integer value and a pointer to the next node. Structs are particularly useful for data structures like linked lists, trees, or graphs, where relationships between elements are defined by pointers. Methods in Go allow associating behavior with a struct. A method is a function with a special receiver, which can be a struct or a pointer to a struct. For example, to add a method to `Node` that modifies its value:

```
package main
import "fmt"
type Node struct {
    Value  int
    Next   *Node
}
func (n *Node) UpdateValue(newValue int) {
    n.Value = newValue
}
```

```
func main() {
    n := Node{Value: 10, Next: nil}
    n.UpdateValue(20)
    fmt.Println(n.Value) // Outputs 20
}
```

In this example, `UpdateValue` uses a pointer receiver (`*Node`) to directly modify the struct. If the receiver were `n Node`, the method would receive a copy of the struct, and changes would not affect the original. Pointer receivers are common in data structures to avoid costly copies and allow in-place modifications, such as in a linked list where a node needs to be updated. Interfaces in Go play a key role in creating generic and reusable data structures. An interface defines a set of methods a type must implement without specifying how. This allows working with different data structures uniformly. For example, for a data structure like a stack, one could define the following interface.

```
package main
import "fmt"
type Stack interface {
    Push(value int)
    Pop() (int, bool)
}
type ArrayStack struct {
    elements []int
}
func (s *ArrayStack) Push(value int) {
    s.elements = append(s.elements, value)
}
func (s *ArrayStack) Pop() (int, bool) {
    if len(s.elements) == 0 {
        return 0, false
    }
    value := s.elements[len(s.elements)-1]
    s.elements = s.elements[:len(s.elements)-1]
```

```
    return value, true
}
func main() {
    var stack Stack = &ArrayStack{}
    stack.Push(5)
    stack.Push(10)
    if value, ok := stack.Pop(); ok {
        fmt.Println("Popped value:", value) // Outputs 10
    }
}
```

In this example, the `Stack` interface defines the `Push` and `Pop` methods, and `ArrayStack` implements this interface using an array as the underlying structure. Another implementation could use a linked list, but as long as it implements `Push` and `Pop`, it would be compatible with the `Stack` interface. This allows manipulating different stack implementations interchangeably, a key concept for data structures like queues, trees, or hash tables. Interfaces also promote composition. For example, a data structure like a binary search tree could implement multiple interfaces, such as one for searching and one for insertion, enabling modular code reuse. Additionally, interfaces combined with pointers allow efficient modifications to data structures, as they avoid copies while providing a clear abstraction.

Exercises for Chapter 3

Problem 1

Write a Go function that computes the multiplicative inverse of a 16-bit unsigned integer. Verify that your function is correct by testing it over all 16-bit integers.

Problem 2

Given the 16-bit multiplicative inverse of `5`, write a program that computes `inverse(5) * x` for all 16-bit unsigned integers `x`. By examining the result, find a way to test whether `x` is a multiple of `5` using a single multiplication and a single comparison.

Problem 3

Cast the 16-bit unsigned multiplicative inverse of `5` to a signed 16-bit integer and call the result `z`. Write a program that computes `z * x` for all 16-bit signed integers `x`. Explain the result.

Problem 4

Given a potentially truncated utf-8 string as an array of bytes (`[]byte`), write a function which finds the location of last valid character.

Chapter 4

At a fundamental level, a programmer needs to manipulate bits. Modern processors operate over data by loading in 'registers' and not individual bits. Thus a programmer must know how to manipulate the bits within a register. Generally, we can do so while programming with 8-bit, 16-bit, 32-bit and 64-bit integers. For example, suppose that I want to set an individual bit to value 1. Let us pick the bit an index 12 in a 64-bit words. The word with just the bit at index 12 set is `1<<12`: the number 1 shifted to the left 12 times, or 4096. In Go, we format numbers using the `fmt.Printf` function: we use a string with formatting instructions followed by the values we want to print. We begin a formatting sequence with the letter `%` which has a special meaning (if one wants to print `%`, one most use the string `%%`). It can be followed by the letter `b` which stands for binary, the letter `d` (for decimal) or `x` (for hexadecimal). Sometimes we want to specify the minimal length (in characters) of the output, and we do so by a leading number: e.g, `fmt.Printf("%100d", 4096)` prints a 100-character string that ends with 4096 and begins with spaces. We can specify zero as a padding character rather than the space by adding it as a prefix (e.g., `"%0100d"`). In Go, we may print thus the individual bits in a word as in the following example:

```
package main

import "fmt"

func main() {
```

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

Running this program we get a binary string representing `1<<12`:

```
0000000000000000000000000000000000000000000000000001000000000000
```

The general convention when printing numbers is that the most significant digits are printed first followed by the least significant digits: e.g., we write 1234 when we mean `1000 + 200 + 30 + 4`. Similarly, Go prints the most significant bits first, and so the number `1<<12` has `64-13=51` leading zeros followed by a `1` with 12 trailing zeros.

We might find it interesting to revisit how Go represents negative integers. Let us take the 64-bit integer `-2`. Using two's complement notation, the number should be represented as the unsigned number `(1<<64)-2` which should be a word made entirely one ones, except for the second last bit. We can use the fact that a *cast* operation in Go (e.g., `uint64(x)`) preserves the binary representation:

```
package main

import "fmt"

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

This program will print `11...10` as expected.

Go has some relevant binary operators that we often use to manipulate bits:

```
&    bitwise AND
|    bitwise OR
```

```
^    bitwise XOR
&^   bitwise AND NOT
```

Furthermore, the symbol `^` is also used to flip all bits a word when used as an unary operation: `a ^ b` computes the bitwise XOR of `a` and `b` whereas `^a` flips all bits of `a`. We can verify that we have `a|b == (a^b) | (a&b) == (a^b) + (a&b)`.

We have other useful identities. For example, given two integers `a` and `b`, we have that `a+b = (a^b) + 2*(a&b)`. In the identity `2*(a&b)` represents the carries whereas `a^b` represents the addition without the carries. Consider for example `0b1001 + 0b10001`. We have that `0b1 + 0b1 == 0b10` and this is the `2*(a&b)` component, whereas `0b1000 + 0b10000 == 0b11000` is captured by `(a^b)`. We have that `2*(a|b) = 2*(a&b) + 2*(a^b)`, thus `a+b = (a^b) + 2*(a&b)` becomes `a+b = 2*(a|b) - (a^b)`. These relationships are valid whether we consider unsigned or signed integers, since the operations (bitwise logical, addition and subtraction) are identical at the bits level.

You may want to verify the following identities:

- `((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)`

Setting, clearing and flipping bits

We know how to create a 64-bit word with just one bit set to 1 (e.g., `1<<12`). Conversely, we can also create a word that is made of 1s except for a 0 at bit index 12 by flipping all bits: `^uint64(1<<12)`. Before flipping all bits of an expression, it is sometimes useful to specify its type (taking `uint64` or `uint32`) so that the result is unambiguous.

We can then use these words to affect an existing word:

1. If we want to set the 12th bit of word `w` to one: `w |= 1<<12`.
2. If we want to clear (set to zero) the 12th bit of word `w`: `w &^= 1<<12` (which is equivalent to `w = w & ^uint64(1<<12)`).
3. If we just want to flip (send zeros to ones and ones to zeros) the 12th bit: `w ^= 1<<12`.

We may also affect a range of bits. For example, we know that the word `(1<<12)-1` has all but the 11 least significant bits set to zeros, and the 11 least significant bits set to ones.

1. If we want to set the 11 least significant bits of the word `w` to ones: `w |= (1<<12)-1`.
2. If we want to clear (set to zero) the 11 least signficant bits of word `w`: `w &^= (1<<12)-1`.
3. If we want to flip the 11 least signficant bits: `w ^= (1<<12)-1`. The expression `(1<<12)-1` is general in the sense that if we want to select the 60 least significant bits, we might do `(1<<60)-1`. It even works with 64 bits: `(1<<64)-1` has all bits set to 1.

We can also generate a word that has an arbitrary range of bits set: the word `((1<<13)-1) ^ ((1<<2)-1)` has the bits from index 2 to index 12 (inclusively) set to 1, other bits are set to 0. With such a construction, we can set, clear or flip an arbitrary range of bits within a word, efficiently.

We can set any bit we like in a word. But what about querying the bit sets? We can check the 12th bit is set in the word `u` by checking whether `w & (1<<12)` is non-zero. Indeed, the expression `w & (1<<12)` has value `1<<12` if the 12th bit is set in `w` and, otherwise, it has value zero. We can extend such a check: we can verify whether any of the bits from index 2 to index 12 (inclusively) set to 1 by computing `w & ((1<<13)-1) ^ ((1<<2)-1)`. The result is zero if and only if no bit in the specified range is set to one.

Efficient and safe operations over integers

By thinking about values in terms of their bit representation, we can write more efficient code or, equivalent, have a better appreciation for what optimized binary code might look like. Consider the problem of checking if two numbers have the same sign: we want to know whether they are both smaller than zero, or both greater than or equal to zero. A naive implementation might look as follows:

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

However, let us think about what distinguishes negative integers from other integers: they have their last bit set. That is, their most significant bit as an unsigned value is one. If we take the exclusive or (xor) of two integers, then the result will have its last bit set to zero if their sign is the same. That is, the result is positive (or zero) if and only if the signs agree. We may therefore prefer the following function to determine if two integers have the same sign:

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

Suppose that we want to check whether `x` and `y` differ by at most 1. Maybe `x` is smaller than `y`, but it could be larger.

Let us consider the problem of computing the average of two integers. We have the following correct function:

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

```
  }
}
```

With a better knowledge of the integer representation, we can do better.

We have another relevant identity `x == 2*(x>>1) + (x&1)`. It means that `x/2` is within `[(x>>1), (x>>1)+1)`. That is, `x>>1` is the greatest integer no larger than `x/2`. Conversely, we have that `(x+(x&1))>>1` is the smallest integer no smaller than `x/2`.

We have that `x+y = (x^y) + 2*(x&y)`. Hence we have that `(x+y)>>1 == ((x^y)>>1) + (x&y)` (ignoring overflows in `x+y`). Hence, `((x^y)>>1) + (x&y)` is the greatest integer no larger than `(x+y)/2`. We also have that `x+y = 2*(x|y) - (x^y)` or `x+y + (x^y)&1= 2*(x|y) - (x^y) + (x^y)&1` and so `(x+y+(x^y)&1)>>1 == (x|y) - ((x^y)>>1)` (ignoring overflows in `x+y+(x^y)&1`). It follows that `(x|y) - ((x^y)>>1)` is the smallest integer no smaller than `(x+y)/2`. The difference between `(x|y) - ((x^y)>>1)` and `((x^y)>>1) + (x&y)` is `(x^y)&1`. Hence, we have the following two fast functions:

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

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

Though we use the type `uint16`, it works irrespective of the integer size (`uint8`, `uint16`, `uint32`, `uint64`) and it also applies to signed integers (`int8`, `int16`, `int32`, `int64`).

Efficient Unicode processing

In UTF-16, we may have surrogate pairs: the first word in the pair is in the range `0xd800` to `0xdbff` whereas the second word is in the range

from `0xdc00` to `0xdfff`. How may we detect efficiency surrogate pairs? If the values are stored using an `uint16` type, then it would seem that we could detect a value part of a surrogate pair with two comparisons: `(x>=0xd800) && (x<=0xdfff)`. However, it may prove more efficient to use the fact that subtractions naturally *wrap-around*: `0-0xd800==0x2800`. Thus `x-0xd800` will range between 0 and `0xdfff-0xd800` inclusively whenever we have a value that is part of a surrogate pair. However, any other value will be larger than `0xdfff-0xd800=0x7ff`. Thus, a single comparison is needed: `(x-0xd800)<=0x7ff`. Once we have determined that we have a value that might correspond to a surrogate pair, we may check that the first value `x1` is valid (in the range `0xd800` to `0xdbff`) with the condition `(x1-0xd800)<=0x3ff`, and similarly for the second value `x2`: `(x2-0xdc00)<=0x3ff`. Both checks may be combined in one expression: `((x1-0xd800)|(x2-0xdc00)) <=0x3ff`. We may then reconstruct the code point as `(1<<20) + ((x-0xd800)<<10) + x-0xdc00`. In practice, you may not need to concern yourself with such an optimization since your compiler might do it for you. Nevertheless, it is important to keep in mind that what might seem like multiple comparisons could actually be implemented as a single one.

Basic SWAR

Modern processors have specialized instructions capable of operating over multiple units of data with a single instruction (called SIMD for Single Instruction Multiple Data). We can do several operations using a single instruction (or few) instructions with a technique called SWAR (*SIMD within a register*) (Lamport, 1975[21]). Typically, we are given a 64-bit word `w` (`uint64`) and we want to treat it as a vector of eight 8-bit words (`uint8`).

Given a byte value (`uint8`), we can replicate it over all bytes of a word with a single multiplication: `x * uint64(0x0101010101010101)`. For example, we have `0x12 * uint64(0x0101010101010101) ==`

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

`0x1212121212121212`. This approach can be generalized in various ways. For example, we have that `0x7 * uint64(0x1101011101110101) == 0x7707077707770707`.

For convenience, let us define `b80` to be the `uint64` equal to `0x8080808080808080` and `b01` be the `uint64` equal to `0x0101010101010101`. We can check whether all bytes are smaller than 128. We first replicate the byte value with all but the most significant bit set to zero (`0x80 * b01` or `b80`) and then we compute the bitwise AND with our 64-bit word and check whether the result is zero: `(w & b80)) == 0`. It might compile to two or three instructions on a processor.

Thus we can verify whether a byte slice is made of ASCII characters by loading 8-byte words. The function IsAscii in the following program takes a byte slice as an argument and returns a boolean value indicating whether the byte slice contains only ASCII characters. It declares a variable x of type uint64 and initializes it to zero. This variable will store the bitwise OR of all the bytes in the slice. The variable is index to iterate over the byte slice. It uses a for loop to process the byte slice in chunks of 8 bytes at a time. For each chunk, it calls the binary.LittleEndian.Uint64 function to convert the 8 bytes into a uint64 value, and then performs a bitwise OR operation with x. This way, x will have a bit set to 1 if any of the bytes in the slice has that bit set to 1. It uses another for loop to process the remaining bytes in the slice, if any. For each byte, it converts it to a uint64 value and performs a bitwise OR operation with x. It performs a bitwise AND operation between x and the constant 0x8080808080808080. This constant has the most significant bit of each byte set to 1, and the rest to 0. The result of this operation will be zero if and only if none of the bytes in the slice has the most significant bit set to 1, which means they are all ASCII characters. It returns the result of comparing the bitwise AND operation with zero.

```
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)))
}
```

We can check whether any byte is zero, assuming that we have checked that they are smaller than 128, with an expression such as `((w - b01) & b80) == 0`. If this expression is false, then `w` contains at least one null byte. If we are not sure that they are smaller than 128, we can simply add an operation: `(((w - b01)|w) & b80) == 0`. Checking that a byte is zero allows us to check whether two words, `w1` and `w2`, have a matching byte value since, when this happens, `w1^w2` has a zero byte value.

We can also design more complicated operations if we assume that all byte values are no larger than 128. For example, we may check that all byte values are no larger than a 7-bit value (`t`) by the following routine:

`((w + (0x80 - t) *  b01) & b80) == 0`. If the value `t` is a constant, then the multiplication would be evaluated at compile time and it should be barely more expensive than checking whether all bytes are smaller than 128. In Go, we check that no byte value is larger than 77, assuming that all byte values are smaller than 128 by verifying thaat `b80 & (w+(128-77) * b01)` is zero. Similarly, we can check that all byte values are at least as large a 7-bit `t`, assuming that they are also all smaller than 128: `((b80 - w) + t * b01) & b80) == 0`. We can generalize further. Suppose we want to check that all bytes are at least as large at the 7-bit value `a` and no larger than the 7-bit value `b`. It suffices to check that `((w + b80 - a * b01) ^ (w + b80 - b * b01)) & b80 == 0`.

Rotating and reversing bits

Given a word, we say that we *rotate* the bits if we shift left or right the bits, while moving back the leftover bits at the beginning. To illustrate the concept, suppose that we are given the 8-bit integer `0b1111000` and we want to rotate it left by 3 bits. The Go language provides a function for this purpose (`bits.RotateLeft8` from the `math/bits` package): we get `0b10000111`. In Go, there is no *rotate right* operation. However, rotating left by 3 bits is the same as rotating right by 5 bits when processing 8-bit integers. Go provide rotation functions for 8-bit, 16-bit, 32-bit and 64-bit integers.

Suppose that you would like to know if two 64-bit words (`w1` and `w2`) have matching byte values, irrespective of the ordering. We know how to check that they have matching ordered byte values efficiently (e.g., `(((w1^w2 - b01)|(w1^w2)) & b80) == 0`). To compare all bytes with all other bytes, we can repeat the same operation as many times as they are bytes in a word (eight times for 64-bit integers), each time, we rotate one of the words by 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)
...
```

We recall that words can be interpreted as little-endian or big-endian depending on whether the first bytes are the least significant or the most significant. Go allows you to reverse the order of the bytes in a 64-bit word with the function `bits.ReverseBytes64` from the `math/bits` package. There are similar functions for 16-bit and 32-bit words. We have that `bits.ReverseBytes16(0xcc00) == 0x00cc`. Reversing the bytes in a 16-bit word, and rotating by 8 bits, are equivalent operations.

We can also reverse bits. We have that

```
bits.Reverse16(0b1111001101010101) == 0b1010101011001111`.
```

Go has functions to reverse bits for 8-bit, 16-bit, 32-bit and 64-bit words. Many processors have fast instructions to reverse the bit orders, and it can be a fast operation.

Fast bit counting

It can be useful to count the number of bits set to 1 in a word. This operation is sometimes called *population counting*. Go has fast functions for this purpose in the `math/bits` package for words having 8 bits, 16 bits, 32 bits and 64 bits. Thus we have that `bits.OnesCount16(0b1111001101010101) == 10`.

Similarly, we sometimes want to count the number of trailing or leading zeros. The number of trailing zeros is the number of consecutive zero bits appearing in the least significant positions.For example, the word `0b1` has no trailing zero, whereas the word `0b100` has two trailing zeros. When the input is a power of two, the number of trailing zeros is the logarithm in base two. We can use the Go functions `bits.TrailingZeros8`, `bits.TrailingZeros16` and so forth to compute the number of trailing zeros. The number of leading zeros is similar, but we start from the most significant positions. Thus the 8-bit integer `0b10000000` has zero

leading zeros, while the integer `0b00100000` has two leading zeros. We can use the functions `bits.LeadingZeros8`, `bits.LeadingZeros16` and so forth.

Given only the number a population-count function, we can count the number of trailing zeros efficiently by observing that `^x & (x-1)` is the word where all trailing zeros become ones, and other bits are set to zero. Hence a population count on `^x & (x-1)` is equivalent to the number of trailing zeros.

While the number of trailing zeros gives directly the logarithm of powers of two, we can use the number of leading zeros to compute the logarithm of any integer, rounded up to the nearest integer. For 32-bit integers, the following function provides the correct result:

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

We can also compute other logarithms. Intuitively, this ought to be possible because if $\log_b$ is the logarithm in base b, then $\log_b(x) = \log_2(x)/\log_2(b)$. In other words, all logarithms are within a constant factor (e.g., $1/\log_2(b)$).

For example, we might be interested in the number of decimal digits necessary to represent an integer (e.g., the integer `100` requires three digits). The general formula is `ceil(log(x+1))` where the logarithm should be taken in base 10. We can show that the following function (designed by an engineer called Kendall Willets) computes the desired number of digits for 32-bit integers:

```
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)
}
```

Though the function is a bit mysterious, its computation mostly involves computing the number of trailing zeros, and using the result to lookup a value in a table. It translates in only a few CPU instructions and is efficient.

Indexing Bits

Given a word, it is sometimes useful to compute the position of the set bits (bits set to 1). For example, given the word `0b11000111`, we would like to have the indexes 0, 1, 2, 6, 7 corresponding to the 5 bits with value 1. We can determine efficiently how many indexes we need to produce thanks to the `bits.OnesCount` functions. The `bits.TrailingZeros` functions can serve to identify the position of a bit. We may also use the fact that `x & (x-1)` set to zero the least significant 1-bit of `x`. The following Go function generates an array of indexes:

```
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
}
```

Given `0b11000111`, it produces the array `0, 1, 2, 6, 7`:

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

If we want to compute the bits in reverse order (`7, 6, 2, 1, 0`), we can do so with a bit-reversal function, like so:

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

Conclusion

As a programmer, you may access, set, copy, or move individual bit values efficiently. With some care, you can avoid arithmetic overflows without much of a performance penalty. With SWAR, you can use a single word as if it was made of several subwords. Though most of these operations are only rarely needed, it is important to know that they are available.

Exercises for Chapter 4

Question 1

Write an efficient Go expression that flips the bits from index 3 to index 15 inclusively in a 64-bit word. That is, it should leave the 3 least significant unchanged, flip the next 13 bits, and leave the 48 most significant bits unchanged.

Question 2

Write an efficient Go function that computes the average of two integers, rounding the result up. E.g., the average of an odd integer `x` and an even integer `y` is `x/2 + y/2 + 1`.

Question 3

Given an array of 16-bit integers (`uint16`), check that it is a valid UTF-16 sequence.

Question 4

Write a function that checks whether two 64-bit integers have a matching byte value, without assuming that the byte values are smaller than 128. Your solution should not require more than a handful of operations and no branching (if or for) should be required.

Question 5

Write a SWAR procedure that adds individual bytes from two 64-bit words. Use wrap-around arithmetic ($255 + 255 = 254$).

Chapter 5

Computer software is typically deterministic on paper: if you run twice the same program with the same inputs, you should get the same outputs. In practice, the complexity of modern computing makes it unlikely that you could ever run twice the same program and get exactly the same result, down to the exact same execution time. For example, modern operating systems randomize the memory addresses as a security precaution: a technique called *Address space layout randomization*. Thus if you run a program twice, you cannot be guaranteed that the memory is stored at the same memory addresses. In Go, you can print the address of a pointer with the `%p` directive. The following program will allocate a small array of integers, and print the corresponding address, using a pointer to the first value. If you run this program multiple times, you may get different addresses.

```
package main

import (
    "fmt"
)

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

Thus, in some sense, software programs are already *randomized* whether we like it or not. Randomization can make programming more challenging. For example, a bad program might behave correctly most of the time and only fail intermittently. Such unpredictable behavior is a challenge for a programmer.

Nevertheless, we can use randomization to produce better software: for example by testing our code with random inputs. Furthermore, randomness is a key element of security routines.

Though randomness is an intuitive notion, defining it requires more care. Randomness is usually tied to a lack of information. For example, it may be measured by our inability to predict an outcome. Maybe you are generating numbers, one every second, and after looking at the last few numbers you generated, I still cannot predict the next number you will generate. It does not imply that the approach you are using to generate numbers is magical. Maybe you are applying a perfectly predictable mathematical routine. Thus randomness is relative to the observer and their knowledge.

In software, we distinguish between pseudo-randomness and randomness. If I run a mathematical routine that generates random-looking numbers, but these numbers are perfectly determined, I will say that they are 'pseudo-random'. What *random looking* means is subjective and the concept of pseudo-randomness is likewise subjective.

It is possible, on a computer, to produce numbers that cannot be predicted by the programmer. For example, you might use a temperature sensor in your processor to capture physical 'noise' that can serve as a random-looking input. You might use the time of day when a program was started as a *random* input. We often refer to such values are random (as opposed to pseudo-random). We consider them random in the sense that, even in principle, it is not possible for the software to predict them: they are produced by a process from outside of the software system.

Hashing

Hashing is the process by which we design a function that takes various inputs (for example variable-length strings) and outputs a convenient value, often an integer value. Because hashing involves a function, given the same input, we always get the same output. Typically, hash functions produce a fixed number of bits: e.g., 32 bits, 64 bits, and so forth.

One application of hashing has to do with security: given a file recovered from the network, you may compute a hash value from it. You may then compare it with the hash value that the server provides. If the two hash values match, then it is likely that the file you recovered is a match for the file on the server. Systems such as git rely on this strategy.

Hashing can also be used to construct useful data structures. For example, you can create a hash table: given a set of key values, you compute a hash value representing an index in an array. You can then store the key and a corresponding value at the given index, or nearby. When a key is provided, you can hash it, seek the address in the array, and find the matching value. If the hashing is random-looking, then you should be able to hash `N` objects into an array of `M` elements for `M` slightly larger than `N` so that a few objects are hashed to the same location in the array. It is difficult to ensure that no two objects are ever mapped to the same array element: it requires that `M` be much, much larger than `N`. For `M` much larger than `N`, the probability of a collision is about `1 - exp(-N*N/(2*M))`. Though this probability falls to zero as M grows large, it requires `M` to be much larger than `N` for it to be practically zero. Solving for `p` in `1 - exp(-N*N/(2*M)) = p`, we get `M = -1/2 N*N / ln(1-p)`. That is, to maintain a probability `p`, then `M` must grow quadratically (proportionally to `N*N`) concerning `N`. Thus we should expect that there will be collisions in a hash table even if the hash function appears random. We can handle collisions in various ways. For example, you may use chaining: each element in the array stores a reference to a dynamic container that may contain several keys.

You can also use linear probing. When a collision occurs, that is, when two keys produce the same index after applying the hash function, linear probing involves examining the subsequent locations in the array until finding an empty position to store the key or locating the desired key. Suppose a hash table of size 10, with a simple hash function: `h(key) = key % 10`.
There are several variations of linear probing to optimize performance or handle collisions differently. Instead of advancing one step at a time (as in linear probing), quadratic probing uses a jump that increases quadratically. For example, if a collision occurs at index `h(key)`, it tries the locations `h(key) + 1²`, `h(key) + 2²`, `h(key) + 3²`, and so on. Quadratic probing can help distribute keys more evenly. Another approach is to use a second hash function to determine the jump step in case of a collision. If `h1(key)` gives the initial index and it is occupied, a step is calculated with a second function `h2(key)`, and it tries the locations `h1(key) + h2(key)`, `h1(key) + 2 * h2(key)`, and so forth.

A hash function might take each possible input and assign it to a purely random value given by an Oracle. Unfortunately, such hash functions are often impractical. They require the storage of large tables of input values and matching random values. In practice, we aim to produce hash functions that behave as if they were purely random while still being easy to implement efficiently.

A reasonable example to hash non-zero integer values is the murmur function. The murmur function consists of two multiplications and and three shift/xor operations. The following Go program will display random-looking 64-bit integers, using the murmur function:

```
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)))
    }
}
```

It is a reasonably fast function. One downside of the `murmur64` function is that zero is mapped to zero, so some care is needed.

In practice, your values might not be integers. If you want to hash a string, you might use a *recursive function*. You process the string character by character. At each character, you combine the character value with the hash value computed so far, generating a new hash value. Once the function is completed, you may then apply murmur to the result:

```
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"))
}
```

There are better and faster hash functions, but the result from recursive hashing with a murmur finalizer is reasonable.

Importantly, it is reasonably easy to generate two strings that hash to the same values, i.e., to create a collision. For example, you can verify that the strings `"Ace"`, `"BDe"`, `"AdF"`, `"BEF"` all have the same hash value:

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

When hashing arbitrarily long strings, collisions are always possible. However, we can use more sophisticated (and more computationally expensive) hash functions to reduce the probability that we encounter a problem.

Given a long strings, you may want to hash all sequences of N characters. A naive approach might be as follows:

```
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];
  }
  //...
}
```

You are visiting most character values N times. If N is large, it is inefficient.

You can do better using a rolling hash function: instead of recomputing the hash function anew each time, you just update it. It is possible to only access each character twice (instead of N times).

```
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)
    }
}
```

An interesting characteristic of the provided `murmur64` function is that it is invertible. If you consider the steps, you have two multiplication by odd integers. A multiplication by an odd integer is always invertible: the multiplicative inverse of 0xff51afd7ed558ccd is 0x4f74430c22a54005 and the multiplicative inverse of 0xc4ceb9fe1a85ec53 is 0x9cb4b2f8129337db, as 64-bit unsigned integers. It may be slightly less obvious that `h ^= h >> 33` is invertible. But if `h` is a 64-bit integer, we have that `h` and `h ^`

`(h >> 33)` are identical in their most significant 33 bits, by inspection. Thus if we are given `z = h ^ (h >> 33)`, we have that `z >> (64-33) == h >> (64-33)`. That is, we have identified the most significant 33 bits of `h` from `h ^ (h >> 33)`. Extending this reasoning, we have that `g` is the inverse of `f` in the following code, in the sense that `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
}
```

We often need hash values to fit within an interval starting at zero. E.g., you might want to get a hash value in `[0,max)`, you might use the following function:

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

This function outputs a value in `[0,max)` using a single multiplication. There are alternatives such as `random % max`, but an integer remainder operation may compile to a division instruction, and a division is typically more expensive than a multiplication. Whenever possible, you should avoid division instructions when performance is a factor.

Importantly, the `toIntervalBias` function introduces a slight bias: we start with 2^{64} distinct values and we map them to `N` distinct values. This means that out of 2^{64} original values, about $2^{64}/N$ values correspond to each output value. Let $\lceil x \rceil$ be the smallest integer no smaller than x and $\lfloor x \rfloor$ be the larger integer no larger than x. When $2^{64}/N$ is not an integer, then some output values match $\lceil 2^{64}/N \rceil$ original values, whereas others

match $\lfloor 2^{64}/N \rfloor$ original values. When N is small, it may be negligible, but as N grows, the bias is relatively more important. In some sense, it is the smallest possible bias if we are starting from original values that are uniformly distributed over a set of 2^{64} possible values.

Putting it all together, the following program will hash a string into a value in the interval `[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))
}
```

Though the `toIntervalBias` function is generally efficient, it is unnecessarily expensive when the range is a power of two. If `max` is a power of two (e.g., 32), then `random % max == random & (max-1)`. A bitwise AND with the decremented maximum is faster than even just a multiplication, typically. Thus the following function is preferable.

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

Estimating cardinality

One use case for hashing is to estimate the cardinality of the values in an array or stream of values. Suppose that your software receives billions of identifiers, how many distinct identifiers are there? You could build a database of all identifiers, but it could use a lot of memory and be relatively expensive. Sometimes, you only want a crude approximation, but you want to compute it quickly.

There are many techniques to estimate cardinalities using hashing: Probabilistic Counting (Flajolet-Martin), LOGLOG Probabilistic Counting, and so forth. We can explain the core idea and even produce a useful function without any advanced mathematics.

Suppose there are m distinct values. If you apply a hash function to these m distinct values, and the hash values fall within the range of integers $[0, n)$, it is as if you randomly selected m values from the range $[0, n)$. If you randomly choose m locations in an array of size n, the expected density (the fraction of the array elements selected) is $(1-(1-1/n)^m)$. If we measure this density D, we can calculate m from $(1-(1-1/n)^m) = D$:

$m = \log(1 - D)/\log(1 - 1/n)$. If n is chosen to be sufficiently large, $\log(1 - D)/\log(1 - 1/n)$ should provide a good estimate for the number of distinct values. The following function applies this formula to estimate cardinality:

```
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)
}
```

We can apply our function in the following program. The approximation is rather crude, but it can be good enough in some practical cases as long as the number of distinct elements is smaller than 20 million. We use about 0.1 bits per distinct value. If you expect larger sets, you should use more advanced techniques, or increase the memory allocated in the `estimateCardinality` function. It is able to estimate the number of distinct elements (19.5 million) in a set of a billion elements in maybe just one or two seconds.

```
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)
    }
}

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))
}
```

The Flajolet-Martin algorithm (Flajolet and Martin, 1985[22]) is a probabilistic method for estimating the number of distinct elements in a data stream. It uses a hash function to transform each element into a binary value, then observes the position of the rightmost 1 bit (r). The estimation is based on the maximum value of r; the number of distinct elements is approximately $2^{\max(r)}$. More precisely, the cardinality estimate is given by $2^{\max(r)}/\phi$, where $\phi \approx 0.77351$ is a correction constant to improve accuracy. This algorithm is particularly suitable for massive data streams, as it requires only a small amount of memory to store the maximum value of r. The following algorithm illustrates the Flajolet-Martin algorithm.

```
package main
import (
"fmt"
```

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

```
"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)
  }
}
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))
}
```

Integers

There are many ways to generate random integers, but a particularly simple approach is to rely on hashing. For example, we could start from an integer (e.g., 10) and return the random integer `murmur64(10)` and then increment the integer (e.g, to 11) and next return the integer `murmur64(10)`.

Steele et al. (2014)[23] propose a similar strategy which they call SplitMix: it is part of the Java standard library. It works much like what we just described but instead of incrementing the counter by one, they increment it by a large odd integer. They also use a slightly different version of the `murmur64` version. The following function prints 10 different random values, following the SplitMix formula:

```
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)
}

func main() {
```

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

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

Each time the `splitmix64` function is called, the hidden `seed` variable is advanced by a constant (`0x9E3779B97F4A7C15`). If you start from the same seed, you always get the same random values.

The function then performs a series of bitwise operations on z. First, it performs an XOR operation between z and z shifted right by 30 bits. It then multiplies the result by the constant value 0xBF58476D1CE4E5B9. Next, it performs another XOR operation between the result and the result shifted right by 27 bits. Finally, it multiplies the result by the constant value 0x94D049BB133111EB and returns the result XORed with the result shifted right by 31 bits.

It produces integers using the full 64-bit range. If one needs a random integer in an interval (e.g., `[0,N)`), then more work is needed. If the size of the interval is a power of two (e.g., `[0,32)`), then we may simply use the same technique as for hashing:

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

However, when the bound is arbitrary (not a power of two) and we want to avoid biases, a slightly more complicated algorithm is needed. Indeed, if we assume that the 64-bit integers are truly random, then all values are equally likely. However, if we are not careful, we can introduce a bias when converting the 64-bit integers to values in `[0,N)`. It is not a concern

when `N` is a power of two, but it becomes a concern when `N` is arbitrary. A fast routine was described by Lemire (2019)[24] to solve this problem:

```
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
}
```

The `toIntervalUnbiased` function takes two arguments: a pointer to a 64-bit unsigned integer `seed` and a 64-bit unsigned integer `max`. It returns a 64-bit unsigned integer. The function first calls the `splitmix64` function with the `seed` pointer as an argument to generate a random 64-bit unsigned integer `x`. It then multiplies `x` with `max` using the `bits.Mul64` function, which returns the product of two 64-bit unsigned integers as two 64-bit unsigned integers. The higher 64 bits of the product are stored in the variable `hi`, and the lower 64 bits are stored in the variable `lo`. If `lo` is less than `max`, the function enters a loop that generates new random numbers using `splitmix64` and recalculates the product of `x` and `max` until `lo` is greater than or equal to `-max % max`. This algorithm ensures that the distribution of random numbers is unbiased.

The general strategy used by this function is called the rejection method: we repeatedly try to generate a random integer until we can produce an unbiased result. However, when the interval is much smaller than 2^{64} (a common case), then we are very unlikely to use the rejection method or

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

to even have to compute an integer remainder. Most of the time, the function never enters in the rejection loop.

Testing that a random generator appears random is challenging. We can use many testing strategies, and each testing strategy can be more or less extensive. Thankfully, it is not difficult to think of some tests we can apply. For example, we want the distribution of values to be uniform: the probability that any one value is generated should be 1 over the number of possible values. When generating 2 to the 64 possible values, it is technically challenging to test for uniformity. However, we can conveniently restrict the size of the output with a function such as `toInterval`.

The following program computes the relative standard deviation of a frequency histogram based on 100 million values. The relative standard deviation is far smaller than 1% (0.05655%) which suggests that the distribution is uniform.

```
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, "%")
}
```

Random shuffle

Sometimes, you are given an array that you want to randomly shuffle. An elegant algorithm described by Knuth is the standard approach. The algorithm works by iterating over the array from the last element to the first element. At each iteration, it selects a random index between 0 and the current index (inclusive) and swaps the element at the current index with the element at the randomly generated index.

We can prove that it provides a fair random shuffle by an induction argument. There are $N!$ possible permutation of an array of size N and we want an algorithm that produces one out of these $N!$ permutations at random. When the array is of length 2, we can verify that it either keeps the default order, or it permutes the two values: each possibility has a 50% probability. Suppose that you want to randomly shuffle an array of size N, but you know how to randomly shuffle an array of size $N-1$. You begin by shuffling either the first or last $N-1$ elements of the array of size N. There are $(N-1)!$ such permutations, and you assume that they are equally likely by your induction argument. Then you permute the lone unshuffled element with any other element chosen (including itself) at random (uniformly). This creates $N!$ permutations, all equally likely. Thus the algorithm is correct. You may think that designing other random shuffling algorithms is easy, but we know of few such algorithms.

The following program shuffles randomly an array based on a seed. Changing the seed would change the order of the array. For large arrays, the number of possible permutations is likely to exceed the number of possible seeds: it implies that not all possible permutations are possible with such an algorithm using a simple fixed-length seed.

```
package main
```

```go
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)
}
```

Reservoir Sampling

We sometimes want to pick k distinct elements at random out of a set of N elements. An obvious solution would be to randomly shuffle the N elements and pick the first or last k elements. Unfortunately, such an algorithm requires much storage (N elements). A more efficient approach is to use a variation of the standard random shuffle that keeps only the first k elements in memory. The result is called reservoir sampling.

The algorithm also works by iterating over the array. Like for the random shuffle, at each iteration, we selects a random index between 0 and the current index (inclusive) and we virtual swaps the element at the current index with the element at the randomly generated index. However, we only need to physically swap if the randomly generated index points within the first k elements.

The code looks much like the random shuffle. An interesting feature of this algorithm is that it requires as little memory as possible. Unfortunately, if the input array is large, it may not be the best algorithm because it requires scanning the entire array.

```
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)
    }
}
```

In the scenario where you want to pick few elements from a large elements, you may simply keep on selecting an index at random, add it to a set of indexes, until you have reached k distinct indexes. There are several ways to implement a set data structure, but simplest one is an array which may work well if you want to pick few elements. A hash table or other such data structure might work better for larger sets. It is also possible to solve this problem using a bitset data structure. Yet for many use cases, the following simple code should prove useful:

```
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)
    }
}
```

Floats

It is often necessary to generate random floating-point numbers. Software systems typically use IEEE 754 floating-point numbers.

To generate 32-bit floating-point numbers in the interval $[0, 1)$, it may seem that we could generate a 32-bit integer (in $[0, 2^{32})$) and divide it by 2^{32} to get a random floating-point value in $[0, 1)$. That's certainly "approximately true", but we are making an error when doing so. How much of an error?

Floating-point (normal numbers) are represented as a sign bit, a mantissa, and an exponent as follows:

- There is a single sign bit. Because we only care about positive numbers, this bit is fixed and can be ignored.
- The mantissa of a 32-bit floating point number is 23 bits. It is implicitly preceded by the number 1.
- There are eight bits dedicated to the exponent. For normal numbers, the exponent ranges from -126 to 127. To represent zero, you need an exponent value of -127 and zero mantissa.

So how many normal non-zero numbers are there between 0 and 1? The negative exponents range from -1 to -126. In each case, we have 2^{23}

distinct floating-point numbers because the mantissa is made of 23 bits. So we have 126 x 2^{23} normal floating-point numbers in $[0, 1)$. If you don't have a calculator handy, that's 1,056,964,608. If we want to add the numbers 0 and 1, that's $126 \times 2^{23} + 2$ slightly over a billion distinct values. There are 2^{32} 32-bit words or slightly over 4 billion, so about a quarter of them are in the interval `[0,1]`. Of all the float-pointing point numbers your computer can represent, a quarter of them lie in `[0,1]`. By extension, half of the floating-point numbers are in the interval `[-1,1]`.

The number 2^{32} is not divisible by $126 \times 2^{23} + 2$, so we can't take a 32-bit non-negative integer, divide it by 2^{32} and hope that this will generate a number in `[0,1]` or $[0, 1)$ in an unbiased way.

We can use the fact that the mantissa uses 23 bits. This means in particular that you pick any integer in $[0, 2^{24})$, and divide it by 2^{24}, then you can recover your original integer by multiplying the result again by 2^{24}. This works with 2^{24} but not with 2^{25} or any other larger number. For 64-bit floating-point numbers, you have greater accuracy as you can replace 24 with 53.

So you can pick a random integer in $[0, 2^{24})$, divide it by 2^{24} and you will get a random number in $[0, 1)$ without bias, meaning that for every integer in `[0,2^{24})`, there is one and only one number in $[0, 1)$. Moreover, the distribution is uniform in the sense that the possible floating-point numbers are evenly spaced (the distance between them is a flat 2^{-24}).

So even though single-precision floating-point numbers use 32-bit words, and even though your computer can represent about 230 distinct and normal floating-point numbers in $[0, 1)$, chances are good that your random generator only produces 2^{24} distinct 32-bit floating-point numbers in the interval $[0, 1)$, and only 2^{53} distinct 64-bit floating-point numbers.

A common way to generate random integers in an interval `[0,N)` is to first generate a random floating-point number $[0, 1)$ and then multiply the result by `N`. Should `N` exceeds 2^{24} (or 2^{53}), then you are unable to generate all integers in the interval `[0,N)`. Similarly, to generate numbers in `[a,b)`, you would generate a random floating-point number $[0, 1)$ and

then multiply the result by `b-a` and add `a`. The result may not be ideal in general but it is convenient.

The following program generates random floating-point numbers:

```
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))
}
```

You may prefer to generate floating-point numbers in the interval `(0,1]`. You can modify the function accordingly, e.g.:

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

In some instances, you may want to sacrifice the fact that numbers are uniformly distributed, at fixed intervals, so you can generate more floating-point values in the interval. Cawley (2023)[25] provides a strategy. The general idea is to generate numbers in the interval `[0.5,1]` with probability 0.5 and a gap between the possible numbers of 2^{-54}, and then numbers in the interval `[0.25,0.5]` with probability 0.25 and a gap between the possible numbers of 2^{-55}, and so forth. Cawley proposes the following function:

```
// 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)
```

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

```
  return math.Float64frombits(x)
}
```

An amusing application of floating-point is to estimate the value of π. If we generate two floating-point numbers x, y in $[0, 1), [0, 1)$, then out of an area of 1 (the unit square), then the area was `x*x+y*y <= 1` should be $\pi/4$. The following program prints an estimate of the value of π.

```
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))
}
```

Of course, practical algorithms might require other distributions such as the normal distribution. We can generate high quality normally distributed floating-point values at high speed using the The Ziggurat Method (Marsaglia & Tsang, 2000[26]). The implementation is not difficult, but it is technical. In particular, it requires a precomputed table. Typically, we generate normally distributed values with a mean of zero and a standard deviation of one: we often multiply the result by the square root of the desired standard deviation, and we add the desired mean.

Discrete distributions

Sometimes we are given a collection of possible values and each value has a corresponding probability. For example, we might pick at random one of three colors (red, blue, green) with corresponding probabilities (20%, 40%, 40%). If there are few such values (for example three), a standard approach is a roulette wheel selection. We divide the interval from 0 to 1 into three distinct components, one for each colour: from 0 to 0.2, we pick red, from 0.2 to 0.6, we pick blue, from 0.6 to 1.0, we pick green.

The following program illustrates this algorithm:

[26] http://www.jstatsoft.org/v05/i08/paper

```
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())

    // Create a slice of cumulative probabilities
    cumulProb := make([]float64, len(probabilities))
    cumulProb[0] = probabilities[0]
    for i := 1; i < len(probabilities); i++ {
      cumulProb[i] = cumulProb[i-1] + probabilities[i]
    }
```

```
    // Generate a random number between 0 and 1
    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))
}
```

If you have to pick a value out of a large set, a roulette-wheel selection approach can become inefficient. In such cases, we may use the alias method[27].

[27] https://en.wikipedia.org/wiki/Alias_method

Cryptographic hashing and random numbers

We do not typically reimplement cryptographic functions. It is preferable to use well-tested implementations. They are typically reserved for cases where security is a concern because they often use more resources.

Cryptographic hashing of strings is designed so that it is difficult to find two strings that collide (have the same hash value). Thus if you receive a message, and you were given its hash value ahead of time, and you check that the hash value sent and the hash value computed from the sent message correspond, there are good chances that the message has not been corrupted. It is difficult (but not impossible) for an attacker to produce a message that matches the hash value you were given. To hash a string in Go cryptographically, you may use the following code:

```
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)
}
```

Similarly, you may want to generate random numbers in a cryptographical manner: in such cases, the produced random numbers are difficult to predict. Even if I were to give you the ten last numbers, it would be difficult to predict the next one. If you were to implement software for an online casino, you should probably use cryptographic random numbers.

```
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)
}
```

Exercises for Chapter 5

Question 1

I have 100 elements and I would like to hash them to 100 hash values, and I want to be sure that the probability that two hash values collide is low. Assuming that you have ideal (perfectly) random hash values, what is the probability of a collision if I produce 32-bit hash values? What if I use 128-bit hash values?

Question 2

Given the following function

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

```
  h ^= h >> 33
  return h
}
```

Find its inverse.

Question 2

Write a program that tests the uniformity of the SplitMix random generator, for various seeds. You may generate values within an interval. Are some seeds better than others? Are some intervals better or worse?

Question 3

The following function generates a single random 32-bit floating-point number. Modify the function so that it generates two 32-bit floating-point numbers from a single call to `splitmix64`.

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

Question 4

Modify the `estimateCardinality` function so that when the density is 1.0, it reruns the analysis using twice as much memory, and so forth, until it can measure a non-unitary density.

Chapter 6

When programming software, we are working over an abstraction over a system. The computer hardware may not know about your functions, your variables, and your data. It may only see bits and instructions. Yet to write efficient software, the programmer needs to be aware of the characteristics of the underlying system. Thankfully, we can also use the software itself to observe the behavior of the system through experiments.

Between the software and the hardware, there are several layers such as the compilers, the operating system, and the hardware. A good programmer should take into account these layers when needed. A good programmer must also understand the behavior of their software in terms of these layers.

Benchmarks in Go

To measure the performance, we often measure the time required to execute some function. Because most functions are fast, it can be difficult to precisely measure the time that takes a function if we run it just once. Instead, we can run the function many times, and record the total time. We can then divide the total time by the number of executions. It can be difficult to decide how many times we should execute the function: it depends in part on how fast a function is. If a function takes 6 seconds to run, we may not want or need to run it too often. An easier strategy

is to specify a minimum duration and repeatedly call a function until we reach or exceed the minimum duration.

When the function has a short execution time, we often call the benchmark a microbenchmark. We use microbenchmarks to compare different implementations of the same functionality or to better understand the system or the problem. We should always keep in mind that a microbenchmark alone cannot be used to justify a software optimization. Real-world performance depends on multiple factors that are difficult to represent in a microbenchmark.

Importantly, all benchmarks are affected by measurement errors, and by interference from the system. To make matters worse, the distribution of timings may not follow a normal distribution.

All programming languages provide the ability to run benchmarks. In Go, the tools make it easy to write benchmarks. You can import the testing package and create a function with the prefix Benchmark and a parameter of pointer type `testing.B`. For example, the following program benchmarks the time required to compute the factorial of 10 as an integer:

```
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)
}
```

If you put functions with such a signature (`BenchmarkSomething(b *testing.B)`) as part of your tests in a project, you can run them with the command `go test -bench .` where `.` refers to the current package. To run just one of them, you can specify a pattern such as `go test -bench Factorial` which would only run benchmark functions containing the word `Factorial`.

The `b.N` field indicates how many times the benchmark function runs. The testing package adjusts this value by increasing it until the benchmark runs for at least one second.

Measuring memory allocations

In Go, each function has its own 'stack memory'. As the name suggests, stack memory is allocated and deallocated in a last-in, first-out (LIFO) order. This memory is typically only usable within the function, and it is often limited in size. The other type of memory that a Go program may use is heap memory. Heap memory is allocated and deallocated in a random order. There is only one heap shared by all functions.

With the stack memory, there is no risk that the memory may get lost or misused since it belongs to a specific function and can be reclaimed at the end of the function. Heap memory is more of a problem: it is sometimes unclear when the memory should be reclaimed. Programming languages like Go rely on a garbage collector to solve this problem. For example, when we create a new slice with the `make` function, we do not need to worry about reclaiming the memory. Go automatically reclaims it. However, it may still be bad for performance to constantly allocate and

deallocate memory. In many real-world systems, memory management becomes a performance bottleneck.

Thus it is sometimes interesting to include the memory usage as part of the benchmark. The Go testing package allows you to measure the number of heap allocation made. Typically, in Go, it roughly corresponds to the number of calls to `make` and to the number of objects that the garbage collector must handle. The following extended program computers the factorial by storing its computation in dynamically allocated slices:

```
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
```

The last function allocates 802816 bytes per operation, unlike the first two. In this instance, if Go determines that data is not referenced after

the function returns (a process called 'escape analysis'), and if the amount of memory used is sufficiently small, it will avoid allocating the memory to the heap, preferring instead stack memory. In the case of the last function, the memory usage is too high, hence there is allocation on the heap rather than the stack.

Measuring memory usage

Your operating system provides memory to a running process in units of pages. The operating system cannot provide memory in smaller units than a page. Thus if you allocate memory in a program, it may either cost no additional memory if there are enough pages already; or it may force the operating system to provide more pages.

The size of a page depends on the operating system and its configuration. It can often vary between 4 kilobytes and 16 kilobytes although much larger pages are also possible (e.g., 1 gigabyte).

A page is a contiguous array of virtual memory addresses. A page may also represent actual physical memory. However, operating systems tend to only map used pages to physical memory. An operating system may provide a nearly endless supply of pages to a process, without ever mapping it to physical memory. Thus it is not simple to ask how much memory a program uses. A program may appear to use a lot of (virtual) memory, while not using much physical memory, and inversely.

The page size impacts both the performance and the memory usage. Allocating pages to a process is not free, it takes some effort. Among other things, the operating system cannot just reuse a memory page from another process as is. Doing so would be a security threat because you could have indirect access to the data stored in memory by another process. This other process could have held in memory your passwords or other sensitive information. Typically an operating system has to initialize (e.g., set to zero) a newly assigned page. As a general rule, an operating system must initialize (e.g., set to zero) a newly allocated page. Additionally, mapping virtual memory pages (used by programs) to actual

physical memory (the hardware) is a complex process that takes time. Each time a program accesses a virtual memory address, the system must translate that address into a corresponding physical address. This process, called page mapping, relies on page tables maintained by the operating system. To speed up this mapping, modern processors use a mechanism called the *translation lookaside buffer* (TLB), or address translation cache. The TLB is a small cache integrated into the processor that stores recent mappings between virtual and physical addresses. When a virtual address needs to be translated, the processor first checks the TLB. If the mapping is found, the translation is fast. However, if the address is not in the TLB, the processor must access the page tables in memory, which is a much slower process, as it typically involves RAM accesses and additional computations. The TLB has limited capacity, meaning it can fill up quickly, especially in systems running multiple processes or handling large amounts of data. When there is no more space in the TLB, the page mapping must be recalculated. This operation can involve multiple memory accesses and complex computations, significantly slowing down system performance. Larger pages can therefore improve the performance of certain programs because they reduce the number of pages required. However, large pages force the operating system to provide memory in larger chunks to a process, potentially wasting precious memory. You can write a Go program which prints out the page size of your system:

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

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

Go makes it relatively easy to measure the number of pages allocated to a program by the operating system. Nevertheless, some care is needed. Because Go uses a garbage collector to free allocated memory, there might be a delay between the moment you no longer need some memory, and the actual freeing of the memory. You may force Go to call immediately the garbage collector with the function call `runtime.GC()`. You should rarely deliberately invoke the garbage collector in practice, but for our purposes (measuring memory usage), it is useful.

There are several memory metrics. In Go, some of the most useful are `HeapSys` and `HeapAlloc`. The first indicates how much memory (in bytes) has been given to the program by the operating system. The second value, which is typically lower indicates how much of that memory is actively in used by the program.

The following program allocates ever larger slices, and then ever smaller slices. In theory, the memory usage should first go up, and then go down:

```
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),
    )
}
```

The program calls `os.Getpagesize()` to get the underlying system's memory page size in bytes as an integer, and assigns it to a variable `pageSize`. It declares a variable `m` of type `runtime.MemStats`, which is a struct that holds various statistics about the memory allocator and the garbage collector. The program repeatedly calls `runtime.GC()` to trigger a garbage collection cycle manually, which may free some memory and make it available for release. It calls `runtime.ReadMemStats(&m)` to populate the `m` variable with the current memory statistics. We can reuse the same variable `m` from call to call. The purpose of this program is to demonstrate how the memory usage of a Go program changes depending on the size and frequency of memory allocations and deallocations, and how the garbage collector and the runtime affect the memory release. The program prints the memory usage before and after each allocation, and shows how the `m.HeapSys`, `m.HeapAlloc`, and `m.HeapSys / pageSize` values grow and shrink accordingly.

If you run this program, you may observe that a program tends to hold on to the memory you have allocated and later released. It is partly a matter of optimization: acquiring memory takes time and we wish to avoid giving back pages only to soon request them again. It illustrates that it can be challenging to determine how much memory a program uses.

The program may print something like the following:

```
$ 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
```

Observe how, at the very beginning and at the very end, over a third of a megabyte of memory (238 pages) is repeated as being in used. Furthermore, over 68,000 pages remain allocated to the program at the very, even though no data structure remains in scope within the `main` function.

Inlining

One of the most powerful optimization technique that a compile may do is function inlining: the compiler brings some of the called functions directly into the calling functions.

Go makes it easy to tell which functions are inlined. We can also easily request that the compiles does not inline by adding the line `//go:noinline` right before a function.

Let us consider this program which contains two benchmarks were we sum all odd integers in a range.

```
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)
}
```

In Go, the flag `-gcflags=-m` tells the compiler to report the main optimizations it does. If you call this program `simpleinline.go` and compile it with the command `go build -gcflags=-m simpleinline.go`, you may see the following:

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

If you run the benchmark, you should see that the inlined version is much faster:

```
$ go run simpleinline.go
```

```
BenchmarkCountOddInline   3716786        294.6 ns/op
BenchmarkCountOddNoinline  1388792        864.8 ns/op
```

Inlining is not always beneficial: in some instances, it can generate large binaries and it may even slow down the software. Modern CPU has usually have a separate cache for code, but its size is limited and cache misses might negate the benefits of inlining. However, when it is applicable, inlining can have a large beneficial effect.

Go tries as hard as possible to inline functions, but it has limitations. For example, compilers often find it difficult to inline recursive functions. Let benchmark two factorial functions, one that is recursive, and one that is not.

```
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()))
}
```

Though both `FactorialLoop` and `Factorial` are equivalent, running this program, you should find that the non-recursive function (`FactorialLoop`) is much faster. A possible output of this program

is as follows. In this instance, the non-recursive function is more than three times faster.

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

Hardware prefetchers

Loading data from memory often takes several nanoseconds. While the processor waits for the data, it may be forced to wait without performing useful work. Hardware prefetchers in modern processors anticipate memory accesses by loading data into the cache before it is requested, thereby optimizing performance. Their effectiveness varies depending on the access pattern: sequential reads benefit from efficient prefetching, unlike random accesses.

To test the impact of prefetchers, we propose a Go program that uses a single array access function, with an index array configured for either sequential or random accesses. The execution time is measured to compare performance. The following program initializes a large array and performs accesses using an index array, either sequential or random, with repeated measurements for greater reliability.

```go
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)
}
```

This program uses a single access function taking an array of indices, configured to be either sequential or random. Sequential accesses should be faster due to hardware prefetchers, while random accesses, being less predictable, will be slower. Execution times are measured over 10 iterations, displaying the minimum, maximum, and average values in milliseconds. Results depend on the processor architecture and cache characteristics. Here is a possible result.

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

In this case, we observe that the sequential access approach is approximately ten times faster than the random access approach.

Cache line

Our computers read and write memory using small blocks of memory called "cache lines". The cache line size is usually fixed and small (e.g., 64 or 128 bytes). To attempt to measure the cache-line size, we may use a strided copy. From a large array, we copy every N bytes to another large array. We repeat this process N times. Thus if the original array contains a 1000 bytes, we always copy 1024 bytes, whether r $N = 1$, $N = 2$, $N = 4$, or $N = 8$.

When N is sufficiently large (say $N = 16$), the problem should be essentially memory bound: the performance is not limited by the number of instructions, but by the system's ability to load and store cache lines. If N is larger than twice the cache line, then we can effectively skip one cache line out of two. You expect a large stride to be significant faster because you skip many cache lines. If N is smaller than the cache line, then every cache line must be accessed. Thus if N is smaller than the cache line, then speed should not be sensitive to the exact value of N.

One limitation to this approach is that processors may fetch more cache lines than needed so we may overestimate the size of the cache line.

However, unless memory bandwidth is overly abundant, we should expect processors to try to limit the number of cache lines fetched.

Let us run an experiment. For each stride size, we repeat 10 times and record the maximum, the minimum and the average. Consider the following program.

```
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)
    }
}
```

We may get the following result:

```
$ 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]
```

We see that the performance increases substantially when the stride goes from 127 to 255. It suggests that the cache line has 128 bytes. If you run this same benchmark on your own system, you may get a different result.

The results need to be interpreted with care: we are not measuring a copy speed of 1247.4 GB/s. Rather, we can copy large arrays at such a speed if we only copy one byte out of every 4095 bytes.

CPU cache

When programming, we often do not think directly about memory. When we do consider that our data uses memory, we often think of it as homogeneous: memory is like a large uniform canvas upon which the computer writes and reads it data. However, your main memory (RAM) is typically buffered using a small amount of memory that resides close to the processor core (CPU cache). We often have several layers of cache memory (e.g., L1, L2, L3): L1 is is typically small but very fast whereas, for example, L3 is larger but slower.

You can empirically measure the effect of the cache. If you take a small array and shuffle it randomly, will be moving data primarily in the CPU cache, which is fast. If you take a larger array, you will move data in memory without much help from the cache, a process that is much slower. Thus shuffling ever larger arrays is a way to determine the size of your cache. It may prove difficult to tell exactly how many layers of cache you have and how large each layer is. However, you can usually tell when your array is significantly larger than the CPU cache.

We are going to write a random shuffle function: `Shuffle(arr []uint32)`. It uses an algorithm called Fisher-Yates shuffle, which involves going through the array in reverse and swapping each element with another randomly chosen from those preceding it. The function uses a seed variable to generate random numbers from a mathematical formula. For our purposes, we use a simplistic number generator: we multiply the seed by the index. The function bits.Mul64 calculates the product of two 64-bit numbers and returns the result as two 32-bit numbers: the most significant (hi) and the least significant. The most significant value is necessarily between 0 and i (inclusively). We use this most significant value as the random index. The function then exchanges the elements using multiple assignment. We call this shuffle function several times, on inputs of different sizes. We report the time normalized by the size of the input.

```
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)
    }
}
```

A possible output of running this program might be:

```
$ 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]
```

We see between 16 KB and 16384 KB, the time per element shuffle does not increase much even though we repeatedly double the input size. However, between 16384 KB and 32768 KB, the time per element doubles. And then it consistently doubles each time the size of the array doubles. It suggests that the size of the CPU cache is about 16384 KB in this instance.

Memory bandwidth

You can only read and write memory up to a maximal speed. It can be difficult to measure such limits. In particular, you may need several cores in a multi-core system to achieve the best possible memory. For simplicity, let us consider maximal read memory.

Many large systems do not have a single bandwidth number. For example, many large systems rely on NUMA: NUMA stands for Non-Uniform Memory Access. In a NUMA system, each processor has its own local memory, which it can access faster than the memory of other processors.

The bandwidth also depends to some extend on the amount of memory requested. If the memory fits in CPU cache, only the first access may be expensive. A very large memory region may not fit in RAM and may require disk storage. Even if it fits in RAM, an overly large memory region might require many memory pages, and accessing all of them may cause page walking due to the limits of the translation lookaside buffer.

If the memory is accessed at random locations, it might be difficult for the system to sustain a maximal bandwidth because the system cannot predict easily where the next memory load occurs. To get the best bandwidth, you may want to access the memory linearly or according to some predictable pattern.

Let us consider the following code:

```
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())
    }
}
```

The code defines two functions: run and main. The main function is the entry point for the program, and it calls the run function 10 times, printing the result each time. The run function is a custom function that measures the memory bandwidth of the system. It does this by performing the following steps:

It declares a variable called bestbandwidth and initializes it to 0.0. This variable stores the highest bandwidth value obtained during the execution of the function. It creates a slice of bytes (uint8) called arr, with a length equivalent to 4 GB. The slice is initialized with 1s. The loop will only access every 64th element of the slice, skipping the rest. Given that most systems have a cache-line size of 64 bytes or more, it is enough to touch each cache line. It calculates the bandwidth by dividing the size of the slice (in bytes) by the difference between the end and start times (in seconds), and then dividing by 1024 three times to convert the result to gigabytes per second (GB/s). The code repeats the measurement 20 times and returns the best result, to account for possible variations in the system performance. The code prints the result 10 times, to show the consistency of the measurement.

Memory latency and parallelism

Latency is often described as the time delay between the beginning of a request and the moment when you are served. Thus if you go to a restaurant, the latency you might be interested in is the time it will take before you can start eating. The latency is distinct from the throughput: a restaurant might be able to serve hundreds of customers at once, but still have high latency (long delays for each customer). If you put a lot of data on a very large disk, you can put this disk in a truck and drive the truck between two cites. It could represent a large bandwidth (much data is moved per unit of time), but the latency could be quite poor (hours). Similarly, you could shine a laser at your partner when supper is ready: the information could arrive without much delay even if you are very far away, but you are communicating little information (low throughput). One way to express this trade-off between latency and throughput is with Little's Law: $L = \lambda W$ where L is the average number of elements in the system, λ is the throughput (long-term average arrival rate of new elements), and W is the latency, or the average amount of time that elements spend waiting. Thus if you want to have L customers at all times in your restaurant, and fewer customers arrive, you should serve the

customers with greater delays. And so forth. Little's law work with our memory subsystems as well: computers can sustain a maximum number of memory requests, each memory request has a latency, and there is an overall bandwidth. If latency does not improve, we can still improve bandwidth or throughput by increasing the number of requests that can be sustained concurrently. Unfortunately, system designers are often forced to make this choice, and so it is not common to see stagnant or worsening memory latencies despite fast improving memory bandwidths. A common illustration of the concept of memory latency is the traversal of a linked list. In computer science, a linked list is a data structure made of nodes, and each node is *linked* (by a pointer) to the next node. The nodes may not be laid out in memory consecutively, but even if they are, accessing each successive node requires a at least a small delay. On current processors, it can often take at least 3 cycles to load data from memory, even if the memory is in cache. Thus determining the length of the list by traversing the whole linked list can take time, and most of this time is just made of the successive delays. The following code benchmarks the time required to traverse a linked list made of a million nodes. Though the time varies depending on your system, it may represent a sizeable fraction of a millisecond.

```
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))
}
```

In this code, a Node struct is defined with two fields: `data` is an integer representing the value stored in the node, `next` is a pointer to the next

node in the linked list. We could also add a pointer to the previous node, but that is not necessary in our case. The build function creates a singly linked list of nodes from an integer volume as an argument. It initializes an empty linked list (`head` is initially `nil`). It iterates from `0` to `volume-1`, creating a new node with value `i` and pointing its next to the current head. The new node becomes the new head. The function returns the final head of the linked list. The main function initializes two global variables (`list` and `N`) storing respectively the head of the list and the expected length. These values are used by the `BenchmarkLen` function. This code demonstrates how to create a linked list, calculate its length, and benchmark the performance of the length calculation. Our length computation is almost entirely bounded (limited) by the memory latency, the time it takes to access the memory. The computations that we are doing (comparisons, increments) are unimportant to the performance. To illustrate our observation, we can try traversing two linked lists simultaneously, as in this example:

```
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))
}
```

If you run this new code, you might find that the benchmark results are close to the single-list ones. It is not surprising: the processor is mostly just waiting for the next node, and waiting for two nodes is not much more expensive. For this reason, when programming, you should limit memory accesses as much as possible. Use simple arrays when you can instead of linked lists or node-based tree structures. We would would like to work with arbitrarily large data structures, so that we can stress the memory access outside of the cache. Sattolo's algorithm is a variant of the well-known random shuffle that generates a random cyclic permutation of an array or list. Sattolo's algorithm ensures that the data is permuted using a single cycle. That is, starting with one element in a list of size `n`, we find that this element is moved to another position, which is itself moved to another position, and so forth, until after n moves, we end up back at our initial position. To apply Sattolo's algorithm, given an array or list of elements, we start with an index i from 0 to `n-1`, where `n` is the length of the array. For each index i, we choose a random index `j` such that `i < j < n`. We swap the elements at indices `i` and `j`. E.g., suppose we have an array [0, 1, 2, 3, 4]. The algorithm might produce a cyclic permutation like [2, 0, 3, 1, 4]. With this algorithm, we can visit all values in an array exactly once in random order. From an array contain indexes 0 to n-1 permuted with Sattolo's algorithm, we first load the first element, read its value, move to the corresponding index, and so forth. After `n` operation, we should come back at the initial position. Because each operation involves a memory load, it is limited by memory latency. We can try to go faster with *memory-level parallelism*: we can pick k positions spread out in the cycle and move from these k initial positions n/k times through the cycle. Because computers can load many values in parallel, this algorithm should be faster for larger values of k. However, as k increases, we may see fewer and fewer gains because systems have limited memory-level parallelism and bandwidth.The following program implements this idea.

```
package main

import (
```

```
    "fmt"
    "math/rand"
    "time"
)

func makeCycle(length int) ([]uint64, []uint64) {
    array := make([]uint64, length)
    index := make([]uint64, length)
    // Create a cycle of maximum length
    // within the big array
    for i := 0; i < length; i++ {
      array[i] = uint64(i)
    }

    // Sattolo shuffle
    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
}

// setupPointers sets up pointers
// based on the given 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)))
     }
}
```

The function makeCycle creates a cycle of a specified length starting at element 0. It initializes two slices: array and index, both of type []uint64. The array slice represents the elements in the cycle. The index slice stores the indices of the elements in the cycle, so that we can more easily access a position in the cycle. The function performs the following steps. It initializes array with values from 0 to length-1. It applies Sattolo's shuffle algorithm to the array to create a random permutation. The function returns both array and index. The function setupPointers: the function calculates the increment value (totalInc) based on the length and the number of lanes (mlp). It assigns the indices from index to sp based on the calculated increments. The function runBench benchmarks the execution time for a given number of lanes (mlp). It initializes a slice sp using setupPointers. The function iterates through the pointers in sp and updates them by following the indices in `array`. It measures the execution time and returns it as a time.Duration instance. The main function first computes the running time for 1 lane, and then it reports the gains when using multiple lanes. Overall, this code generates a cycle of specified length, sets up pointers, and benchmarks the execution time for different numbers of lanes. The primary purpose seems to be exploring parallelization using multiple lanes. The runBench function simulates parallel execution by updating pointers concurrently. The speedup is calculated by comparing the execution time for different numbers of lanes. The larger the speedup, the more efficient the memory-level parallel execution. The general principle is that you can often improve the performance of a system that faces high latencies by breaking the data dependencies. Instead of putting all your data in a long chain, try to

break to have no chain at all or, if you must have chains, use several smaller chains.

Superscalarity and data dependency

Most current processors are superscalar (as opposed to 'scalar'), meaning that they can execute and retire several instructions per CPU cycles. That is, even if you have a single CPU core, there is much parallelism involved. Some processors can retire 8 instructions per cycle or more. Not all code routines benefit equally from superscalar execution. Several factors can limit your processors to few instructions per cycle. Having to wait on memory accesses is one such factor. Another common factor is data dependency: when the next instruction depends on the result of a preceding instruction, it may have to wait before it starts executing. To illustrate consider functions that compute successive differences between elements of an array (e.g., given 5,7,6, you might get the initial value 5 followed by 2 and -1), and the reverse operation which sums up all the differences to recover the original value. You may implement these functions as such:

```
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]
    }
}
```

Assuming that the compiler does not optimize these functions in a non-trivial manner (e.g., using SIMD instructions), we can reason relatively

simply about the performance. For the successive differences, we need approximately one subtraction per element in the array. For the prefix sum, you need approximately one addition per element in the array. It looks quite similar at a glance. However, the data dependency is different. To compute the difference between any two values in the array, you do not need to have computed the preceding differences. However, the prefix sum, as we implemented it, requires us to have computed all preceding sums before the next can be computed. Let us write a small benchmarking program to test the performance difference:

```
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)

}
```

Your result will vary depending on your system. However, you should not be surprised if the prefix sum takes more time. On an Apple system, we go the following results:

```
BenchmarkSuccessiveDifferences 39742334     30.04 ns/op
BenchmarkPrefixSum  8307944      142.8 ns/op
```

The prefix sum can be several times slower, even though it appears at a glance that it should use a comparable number of instructions.

Let us consider another example. We can compute the greatest common divisor between two integers using the Euclidean algorithm. A reasonable implementation in Go is as follows:

```
func gcd(a, b uint) uint {
    for b != 0 {
      quotient := a / b
      a, b = b, a-quotient*b
    }
    return a
}
```

There is an extension of this algorithm which computes not only the greatest common divisor, but also the Bézout coefficients. That is, given two integers `a` and `b`, we wish to find integers `s` and `t` such that `a * s + b * t = gcd(a,b)`: `s` and `t` are called Bézout coefficients. One application of Bézout coefficients is in the computation of a multiplicative inverse: suppose that `a` and `b` are such that `gcd(a,b)=1` then we have that `s` is the multiplicative inverse of `a` modulo `b`: `(a * s) % b = 1`. The extended algorithm is quite similar to the regular Euclidean algorithm:

```
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
}
```

This function returns the greatest common divisor as the first returned value and then the two Bézout coefficients. It would at first glance that the extended Euclidean algorithm is about three times slower than the regular Euclidean algorithm. It does seem to do three times as much computation. We can write a small benchmark to test this hypothesis:

```
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)
}
```

Observe that we ask Go not to inline the functions: these are relatively simple functions and inlining could trigger optimizations that make our analysis complicated. If you run this benchmark, you are likely to find though the extended Euclidean algorithm might be slower, the difference can be small. When we ran this benchmark, we found that the extended Euclidean algorithm was about 35% slower. It is quite a small difference compared to one's expectation that it might be three times slower.

```
GCD  12838      91481 ns/op
EGCD   8866      124004 ns/op
```

In general, you cannot trust a hasty analysis. Just because two functions appear to do a similar amount of work, does not mean that they have the same performance. Similarly, a function that appears to do more work

may not be much slower. Several factors must be taken into account, including data dependencies.

Branch prediction

In part because the processors are multiscalar, they have been designed to execute speculatively: when facing a branch, the processor tries to guess the direction that will be taken, and it begins the computation optimistically. When the processor makes the correct prediction, it usually improves the performance, sometimes by a large amount. However, when the processor is unable to predict accurately the branch, branch prediction may become a net negative. Indeed, when the branch is mispredicted, the processor may have to restart the computation from the point where it made the wrong prediction, an expensive process that can waste several CPU cycles. To illustrate, let us first consider a function that copies the content of an slice into another slice of the same size:

```
func Copy(dest []uint, arr []uint) {
    if len(dest) < len(arr) {
      panic("dest is too small")
    }
    for i, v := range arr {
      dest[i] = v
    }
}
```

A more sophisticated function may copy only the odd elements:

```
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
```

```
      }
    }
}
```

We may try to copy an array that contains random integers (both odd and even), only odd integers, or only even integers. The following program illustrates:

```
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)
}
```

On an Apple system, we got the following results:

```
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
```

The last three timings involve the same function, only the input data differs. We find that all timings are similar in this case, except for benchmark that copies random data: it is several times slower in our tests. The much longer running time is due to the presence of an unpredictable branch in our inner loop. Observe that the same function, subject to the same volume of data, can have vastly different performance characteristics, even though the computational complexity of the function does not change: in all instances, we have linear time complexity. If we expect our data to lead to poor branch prediction, we may reduce the number of branches in the code. The resulting code might be nearly branch free or branchless. For example, we can use an arithmetic and logical expression to replace a condition copy:

```
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])
    }
}
```

Let us review the complicated expression:

- `v & 1`: This operation checks if the least significant bit of v is set (i.e., if v is odd).

- `-(v & 1)`: This negates the result of the previous operation. If v is odd, this becomes -1; otherwise, it becomes 0. However, `-1` as an unsigned integer is becomes the maximal value, the one with all of the bits set to 1.
- `v ^ dest[i]`: This XORs the value of v with the corresponding element in the dest slice.
- `uint(-(v & 1)) & (v ^ dest[i])`: If v is odd, it returns the XOR of v with dest[i]; otherwise, it returns 0.
- Finally, `dest[i] ^= uint(-(v & 1)) & (v ^ dest[i])` leaves `dest[i]` unchanged if v is even, otherwise it replaces with `v` using the fact that `dest[i] ^ (v ^ dest[i]) == v`.

We can put this function to good use in a 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)
}
```

On an Apple system, we got:

```
BenchmarkCopyOdd (random)  60782       19254 ns/op
```

```
BenchmarkCopyOddBranchless (random)   166863      7124 ns/op
```

In this test, the branchless approach is much faster. We should stress that it is not always the case that branchless code is faster. In fact, we observe that in our overall test results, the branchless function is significantly slower than the original when the results are predictable (e.g., 2896 ns/op vs 7124 ns/op). In actual software, you should try to recognize where you have poorly predicted branches and act in these cases to see if a branchless approach might be faster. Thankfully, most branches are well predicted in practice in most projects.

Exercises for Chapter 6

Question 1

Write a program that allocates a slice containing `b` bytes, for `b` ranging from 10,000,000 to 10,000,100. Your program should estimate the memory usage of a newly allocated slice of `b` bytes.

Question 2

Modify your program from the previous question to try to determine the size of a memory page using `runtime.MemStats.HeapInuse`.

Question 3

Write a program that computes the length of k different linked lists for an arbitrary integer $k > 0$.

Chapter 7

A data structure in programming is a specific way of organizing and storing data in a computer so that it can be accessed and used efficiently.

In woodworking or metalworking, a jig holds a piece of work and guides the tools operating on it. It helps to produce consistent results. The simplest jig is probably a straight edge. For example, a straight metal bar can help you cut wood following a straight line. Another simple jig is the mitre box which helps us cut wood at fixed angles.

Data structures can be thought of as jigs for programmers. Much like how a jig guides the tool to make precise cuts or shapes, data structures in programming provide a framework for organizing and accessing data. Just as a jig guides the tool, data structures guide how data should be stored, accessed, and manipulated. Data structures help us ensure that operations on the data are performed in a predictable and efficient manner.

Conventional computer science courses often introduce a wide range of data structures: linked lists, red-black trees, and so forth. Yet, in practice, the overwhelming majority of programming problems can be solved using little more than arrays, the standard composite type (`struct`), and the occasional hash table. The analogy with the woodworking jig still holds: the most popular woodworking jigs are so simple that we often even forget that they are an actual tool.

Arrays

Some data structures are particularly useful due to their simplicity and efficiency. The simplest data structure is the array. Arrays are fixed-size collections of elements. E.g., in Go you can declare an array of 5 integers like so:

```
var myArray [5]int
```

Typically arrays are stored in a contiguous manner in memory. Thus arrays are ideal if you need to traverse all of the elements as quickly as possible: you access the memory in a predictable manner with little overhead. Accessing any given element in an array typically just involves taking the index and converting it to a memory address. It may require only a few inexpensive operations.

Generally speaking, arrays generate fast code in part because compilers find it easy to optimize the functions. For example, in Go, array accesses are typically subject to a bound checker: Go would prevent access to an array at index 10 if the array size is 5. However, the compiler can often optimize away such checks. For example, consider the following function where we sum the five elements in an array. It should compile to efficient code with hardly any overhead (e.g., bound checking).

```
func Sum(x [5]int) int {
    sum := 0
    for i := 0; i < 5; i++ {
      sum += x[i]
    }
    return sum
}
```

An array can contain different values (including arrays!). It is even possible to store Boolean values (true/false) in an array although in such cases, at least a byte per element is required. You may replace an array with a bitset in such cases. You may implement a bitset over an array as follows (using 128 bits as an example).

```
// BitSet represents a fixed-size bitset
type BitSet struct {
    bytes [16]byte
}

// Set sets the bit at 'index' to 1
func (bs *BitSet) Set(index int) {
  bs.bytes[index/8] |= 1 << uint(7-(index%8))
}

// Clear sets the bit at 'index' to 0
func (bs *BitSet) Clear(index int) {
  bs.bytes[index/8] &^= 1 << uint(7-(index%8))
}

// Get returns the value of the bit at 'index'
func (bs *BitSet) Get(index int) bool {
  return bs.bytes[index/8]&(1<<uint(7-(index%8))) != 0
}
```

A natural extension of the array is the multidimensional array. Even though the Go language supports natively multidimensional arrays, we can still implement them from scratch using a standard array. Typically, we use a row-major implementation. For example, we can implement a simple 3x3 matrix using a 9-element array. The first three elements of the array represent the first row, the next three elements represent the second row, and so forth.

```
// Matrix represents a 3x3 matrix
type Matrix struct {
    data [9]float64
}

// Set sets the value
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
}

// Get returns the value
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, like many programming languages, allows you to create efficient composite data structures (`struct`). In particular, you can put arrays inside these data structures. Suppose, for example, that we want to represent points in space, we might do it as follows:

```
type Point struct {
    x float64
    y float64
}
```

We can put these points inside an array of ten elements:

```
var list [10]Point
```

This pattern is often called an *array of structs*. We can also represent the same data using a *struct of arrays*:

```
type Points struct {
    x [10]float64
    y [10]float64
}
```

While both the array of structs and the struct of arrays can contain the same information, they have different performance characteristics. For example, if you needed to sum the x values, the struct of arrays might lead to faster code.

Dynamic arrays and slices

A slightly more sophisticated data structure is the dynamic array. A dynamic array is an array that can grow or shrink in size. In Go, they are implemented as slices. They are one of the most fundamental data structures in Go.

```
var mySlice []int
mySlice = append(mySlice, 1, 2, 3)
```

You access arrays by integer indexes. E.g., you access an array of size 3 with the indexes 0, 1, 2. Each index gives you access to a value. Thus the array constitutes a key-value data structure where the keys are the indexes.

Because the size of the slice is dynamic, Go provides a function to query it: `len` (length). E.g., `len(mySlice)` might be 3.

In Go, you can have several slices share the same underlying array. We use the operator `[begin:end]` to create a new slice from an existing slice. Importantly, this operation does not copy the underlying data.

In the following example, the second slice will be of length 1 and contain the second element of the first slice.

```
var mySlice []int
mySlice = append(mySlice, 1, 2, 3)
secondSlice := mySlice[1:2]
```

Go provides some shorthand notations. Instead of writing `mySlice[2:len(mySlice)]`, you may write `mySlice[2:]` and instead of writing `mySlice[0:1]`, you may write `mySlice[:1]`. To create a copy of the slice, you may simply write `mySlice[:]`.

Slices share many of the same characteristics as arrays. For example, we can write functions that take slices as parameters and compile into similarly efficient code, such as this example where we sum the first 5 elements of the slice:

```
func Sum(x []int) int {
    sum := 0
  if len(x) < 5 {
  return -1
  }
    for i := 0; i < 5; i++ {
      sum += x[i]
    }
    return sum
}
```

In fact, dynamic arrays, that is, slices, are typically implemented as a thin wrapper over an array. The slice points at a region inside an array. When we shrink or extend the region pointed at by the slice, the underlying array may remain the same.

Typically, the underlying array has more storage capacity than the length of the slice. You may query the size of the underlying array with the `cap` (capacity) function. It may seem wasteful to have more capacity than needed. However, consider the case where we regularly add elements to a

given slice, and the size of the underlying array is set to match exactly the length of the slice. In such a case, each addition (append) to the slice might require allocating a new array and copying the elements. Counting only the copies of elements, we get that we need $1+2+3+4+...+n-1 = (n-1)n/2$ copies to populate a slice with n values. Instead, we typically grow the size of the underlying array using exponential steps. For example, whenever the capacity becomes insufficient, we could allocate memory to the next power of two: we allocate 16 elements of capacity if we need 7 elements, we allocate 1024 elements if we need 600 elements, and so forth. We can verify that with such a strategy, if we need to repeatedly add an element to a slice, we will never need more than $2n$ copies to create a slice with n values: $2n$ is much smaller than $(n-1)n/2$ when n is large. By default, Go does not recover excess capacity when making the slice smaller (e.g., `s = s[:newlength]`). However, you can use the extended syntax `s[low:high:max]` to tell Go to reduce the size of the underlying array to `max-low`. E.g., `s = s[::len(s)]` would create a copy of the slice while adjusting the underlying capacity to `len(s)`.

The following program will add elements to a slice and prints the capacity (size of the underlying array) and length of the slice at each increment. At the end of the program, we show that we can shrink the slice with or without adjusting the capacity.

```
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))
}
```

The following is a possible output:

```
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
```

Sometimes it is convenient to store the data in sorted order within an array. We can then use the array as an efficient set data structure even if the array is large. When searching for a value, we may use a binary search. The algorithm is relatively straight-forward: given the value that we are looking for, we first compare it against the value in the middle, that is, the median value. If the value we are searching for is greater than the median value, we search in the later half of the array, otherwise we search in the early half of the array. We repeat the division recursively until we either find the value that we are looking for, or we determine that the value cannot be found. We need about $log_2(N)$ iterations to complete the search over an array of size N, and the logarithmic function grows very slowly (e.g., $log_2(10^6) \approx 20$).

```
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
}
```

Sometimes we only want to be able to access the smallest or largest value. Suppose for example that you have a stream of requests that you need to process, and you always want to pick next the request that has been waiting the longest. You could repeatedly scan the array, or repeatedly sort it. However, if you constantly append and remove data from the array, it could become expensive. Instead, we can order the values in the array according to a *binary heap*. The root of the heap is stored at index 0, and it is either the smallest or the largest value depending on the type of heap you need. So we either have a max-heap or a min-heap. Within the array, the values are then organized as a tree. For any node at index i, its left child is at $2 * i + 1$ and its right child is at $2 * i + 2$. At the end of the array, some node may only have a left child, while others may have no child. If we have a max-heap, each node must be no smaller than its children. If we have a min-heap, each node must be no greater than its children. Operations like insertion and deletion are performed by manipulating elements in the array while maintaining the heap property. To insert a new value, you place the new element at the array's end. It then becomes the child of an existing node if the array was not empty: if you put it at index i, then you can compute the index of the parent as the integer $(i - 1)/2$. You then permute it with its parent to preserve the heap property (e.g., if we have a max-heap, each node must be no smaller

than its children). You may then need to check against the parent once more and so forth until you get to the top of the heap. The following function modifies a slice according to this algorithm using the max-heap property.

```
func Insert(heap *[]int, value int) {
    // Append the new value to the end of the heap
    *heap = append(*heap, value)
    heapSize := len(*heap)

    // Start with the last element's index
    i := heapSize - 1

    for i > 0 {
      parent := (i - 1) / 2

      if (*heap)[i] > (*heap)[parent] {
          // Swap the elements
          (*heap)[i], (*heap)[parent]
      = (*heap)[parent], (*heap)[i]

          // Move up to the parent's index
          i = parent
      } else {
          break
      }
    }
}
```

When removing the top element, you replace it with the last element and then propagate this element downward by swapping it with the larger (or smaller) child if it violates the heap property. We may implement it as follows for a max-heap:

```
func RemoveTop(heap *[]int) int {
    if len(*heap) == 0 {
```

```
    return 0
  }

  // Store the root value to return later
  top := (*heap)[0]

  // Replace the root with the last element
  heapSize := len(*heap)
  (*heap)[0] = (*heap)[heapSize-1]
  *heap = (*heap)[:heapSize-1]

  // Start heapifying down from the root
  i := 0
  for {
    // Calculate indices of children
    leftChild := 2*i + 1
    rightChild := 2*i + 2
    largest := i

    // Check left child
    if leftChild < len(*heap) && (*heap)[leftChild] >
        (*heap)[largest] {
        largest = leftChild
    }

    // Check right child
    if rightChild < len(*heap) && (*heap)[rightChild] >
        (*heap)[largest] {
        largest = rightChild
    }

    if largest != i {
        (*heap)[i],
          (*heap)[largest] = (*heap)[largest],
```

```
            (*heap)[i]
        i = largest
    } else {
        // If largest is the root, we are done
        break
    }
  }

  return top
}
```

Thus with only two relatively simple functions, we can maintain a binary heap. These functions require at most $\lceil \log_2 N \rceil$ comparisons where N is the number of elements in the array. Interestingly, a binary heap provides a sensible algorithm to sort an array sometimes called a heapsort: insert all elements in a binary heap, and then repeatedly remove the top value. The result is a $O(N \log N)$ algorithm.

Just like we can implement a bitset over an array, we can implement a dynamic bitset over a dynamic array. The approach is much the same, but the array is replaced by a slice. Importantly, we need to check for the need to extend the bitset. When the need arises, we grow the bitset by allocating a larger array and copying our existing data. Observe that we choose to grow the slice by twice the amount needed: this prevents degenerate cases where the user might access the bits one by one (0, 1, ...), leading to repeated copies and allocations.

```
type BitSet struct {
    bytes []byte
}

func (bs *BitSet) ensureCapacity(index int) {
    requiredBytes := (index + 1 + 7) / 8
    if len(bs.bytes) < requiredBytes {
      // Grow the slice if necessary
```

```
        newBytes := make([]byte, requiredBytes*2)
        copy(newBytes, bs.bytes)
        bs.bytes = newBytes
    }
}

// Set sets the bit at 'index' to 1
func (bs *BitSet) Set(index int) {
    bs.ensureCapacity(index)
    bs.bytes[index/8] |= 1 << uint(7-(index%8))
}

// Clear sets the bit at 'index' to 0
func (bs *BitSet) Clear(index int) {
    bs.ensureCapacity(index)
    bs.bytes[index/8] &^= 1 << uint(7-(index%8))
}

// Get returns the value of the bit at 'index'
func (bs *BitSet) Get(index int) bool {
    bs.ensureCapacity(index)
    return bs.bytes[index/8]&(1<<uint(7-(index%8))) != 0
}
```

Hash tables and maps

It is common that you need a data structure where you can access values using either non sequential integers (e.g., 10, 1000, 100000), or other types of values such as a string. E.g., maybe you want a map from names to phone numbers. In computer science, a map (also known as a dictionary or associative array) is a function from keys to values, such that given a key, you can efficiently get the value.

One of the most useful data structures is the hash table. Hash tables build on top of arrays, but instead of using consecutive integer values as indexes, they can take nearly any type of values (e.g., string) as keys.

The essential trick of the hash table is to use a hash function which maps arbitrary keys to integer values. The integer values are used as indexes inside an array. The array is typically picked to be of a size comparable to the number of distinct keys. Thus a hash table is a generalization of the array, but one where you replace the simple accesses with a hash function. A hash function takes a key and converts it into an index of the hash table array.

We typically hope that the hash function distributes keys evenly across the array to minimize collisions. When two keys are mapped to the same index inside the array, we must resolve the issue. One possibility is to create a larger array. But this would cause the arrays to grow too quickly in practice. Thus we must handle collisions efficiently. There are two broad strategies. One possibility is to have the ability to store several key-value pairs in each element of the array. It is often called a bucket approach. Another possibility is to store colliding key-value pairs elsewhere: e.g., you can store it to the next available slot in the array.

Many hash table implementations exist with varying properties. Go provides a built-in map type that implements a hash table. E.g., to create a map from strings to integers, we might proceed as follows:

```
myMap := make(map[string]int)
myMap["key"] = 10
```

Go uses a *bucket* approach: each element in the array can store several values. Each bucket in Go's map implementation contains multiple key-value pairs. The exact number of pairs per bucket can vary, but it's designed to handle a small number of entries efficiently. E.g., for example, we could use this type of data structure with slices:

```
type bucket struct {
    keys  []string // Keys
```

```
    values  []int  // Values
}
```

When two keys hash to the same index, they are placed in the same bucket.

When looking up a key, we compute the hash of the key, use the hash to find the correct bucket, check each entry in the bucket for a matching key. As long as the buckets are small enough, the performance is going to be acceptable.

We often compute the number of keys stored in the hash table and divide it by the size of the array. The result is called the load factor. When there are too many keys compared to the size of the underlying array, the array is grown, and the hash table is reconstructed. This will typically reduce the average size of the buckets. Similarly, we can reduce the size of the array if there are too few keys left.

Let us consider a complete example:

```
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 {
    // A very simple hash function
    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")) // Should print 1
}
```

Our example is simplified. However, it illustrates the basic principles of how hash tables work.

The approach we described, with buckets, is relatively simple to implement, but it may not always offer the best performance because of the overhead of maintaining buckets as separate dynamic arrays. Instead, we may consider a variation where we have one key-value per slot in the array. This alternative model is sometimes called open addressing. When two keys hash to the same slot, we can move one of the key-value pairs to another available slot, or increase the size of the underlying array. At query time, we begin by searching for the key-value according to the hash value of the key. If another key-value is found, we visit the next slot, until either we find the value we are looking for, or an empty slot is found. As long as we keep the underlying array large enough so that a sizeable fraction of slots are empty, the performance will be acceptable.

The following code illustrates the main idea behind open addressing, with the exception that the underlying array does not grow as more values are added. To add this functionality, we should track the number of keys added, and grow the array as needed. As a *sentinel* to indicate an available slot, we use the empty key. In practice, we would need to add

additional checks to make sure that the user does not try to add a key with an empty string, and to handle the scenario appropriately.

```
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},
    }
    // Initialize all slots with emptyValue
    for i := 0; i < size; i++ {
      ht.items[i] = ht.emptyValue
    }
    return ht
}

func hash(key string) int {
    // A very simple hash function
    hash := 0
    for i := 0; i < len(key); i++ {
      hash += 31 * int(key[i])
    }
    return hash
}

// Put adds a key-value pair to the hash table
```

```
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
      }
      // Linear probing to find next available slot
      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 // key not found
      }
      hashValue = (hashValue + 1) % len(ht.items)
    }
    return -1, false // key not found after full cycle
}
```

There are many alternatives that could be even faster than open addressing in some instances, such as Cuckoo hashing. In practice, programmers rarely implement their own hash tables, but they should be aware that there are different implementations with various advantages.

A hash table that only has keys can be used to implement a set data structure. That is, we do not always need to have values. A straightforward variation worth considering is to allow keys to be mapped to several different values. This variation is sometimes called a multimap.

Conclusion

Though there are countless data structures in computer science, much can be achieved with arrays, and a few simple functions. When organizing your data, you should first try to use arrays. In some cases, a hash table might be needed if you have a set or a map. It should cover the vast majority of your data structures.

To illustrate the idea, consider JSON. JSON (JavaScript Object Notation) is a standard text-based data interchange format that is commonly used to exchange data online. It has primitive values (strings, numbers, Booleans, the null value) and composite types (arrays and objects). Objects are maps from strings to primitive values or to other composite types (arrays and objects). We represent objects as comma-separated key-value pairs within curly braces `{}`, using the colon as a separator between the key and the value. We use square brackets `[]` for arrays, separating values with commas. Consider the following example representing a list of employees.

```
{
   "employees":[
    {
     "name":"Richard",
     "salary":56000,
     "function":"CEO"
    },
    {
     "name":"Lisa",
     "salary":55000,
     "function":"accountant"
    }
```

```
    ]
}
```

Empirically, JSON is found to be sufficient to represent most data, despite its simplicity. A key insight is that by combining arrays and maps, with a few standard types for numbers and strings, you can solve most problems.

In some cases, specialized data structures can provide superior performance. Yet you should be mindful that it is often easier to optimize code when the underlying data structure is simpler.

Exercises for Chapter 7

Question 1

Implement from scratch an `Insert` and `RemoveTop` function for a min-heap of strings.

Question 2

We know how to implement a binary search. It divides up the sorted arrays in two at each iteration, after a single comparison. We can generalize the binary search to a k-ary search where, at each iteration, we divide the array into k sections. How many comparisons do we need to make at each iteration?

Question 3

Write a function in Go which finds, given a positive integer `n`, the smallest power of two that is at least as large as `n`.

Chapter 8

In practice, the software we write runs on several processors. Unfortunately, much of what we take for granted on a single processor becomes false when there are more than one processor. For example, if two processors modify the same piece of memory, what is the state of the memory after the modifications? It is difficult to tell in general. It is possible that the modification of one processor could overwrite any modification done by the other processor. The reverse could be true: the modification done by the other processor could win out. Or, maybe both processors will attempt the modification and the result will be a confused state that corresponds to nothing we would like to see. We call such accesses a 'data race': a situation where two or more processors in a program access the same memory location simultaneously, and at least one of those accesses is a write operation, without proper synchronization. It gets more difficult when you want two or more processors to meaningfully modify the same memory. For example, suppose that you have a variable that counts the number of products sold. Maybe you have different processors incrementing this one variable.

Threads and goroutines

A thread is the smallest unit of execution within a process that can be scheduled and run independently by a computer's operating system. It represents a single sequence of instructions that the CPU executes, allowing a program to perform multiple tasks concurrently within the

same process. A thread exists within a larger entity called a process, which is essentially a running program with its own memory space, resources, and state. A process can contain one or more threads, all sharing the same memory and resources (like open files or global variables) allocated to that process. There is a limit to how many threads a program can manage efficiently. To enable even more parallelism, the Go programming language has its own concept of a thread called a goroutine. While a goroutine is not a thread in the traditional sense, it maps to conventional threads under the hood. The Go runtime uses a scheduler to map many goroutines onto a smaller number of threads. These threads are the actual threads recognized by the operating system—kernel-level entities with their own stack and execution context, as described in general computing. A single thread in Go can execute multiple goroutines by switching between them efficiently. This makes goroutines much cheaper than OS threads—creating thousands or even millions of goroutines is practical, whereas spawning that many threads would exhaust system resources due to their larger memory footprint. In some sense, Go blurs the distinction between concurrency and parallelism. Concurrency is about managing multiple tasks so they can make progress independently. Parallelism, however, involves executing multiple tasks simultaneously across multiple resources. While concurrency focuses on software design for task coordination and can work with or without multiple cores, parallelism relies on hardware to achieve true simultaneous execution, and the two can combine when a concurrent system leverages parallel resources for efficiency.

To start a goroutine, you only need to type the keyword ‘go’ followed by a function:

```
go func() {
  fmt.Println("Canada")
}()
```

This spawns a goroutine, but the Go runtime decides which thread it runs on, potentially sharing that thread with other goroutines. Unfortunately, a program made of only this goroutine could be disappointing:

```
package main

import (
    "fmt"
)

func main() {
    go func() {
      fmt.Println("Canada")
    }()
}
```

The problem is that the main function might end before the goroutine can terminate. In Go, goroutines run concurrently (at the same time), and the main function (which is the main goroutine) does not automatically wait for other goroutines to complete. If the main goroutine exits, the program terminates, potentially before other goroutines finish. To ensure a goroutine terminates before the program ends, the simplest approach is to synchronize the main goroutine with the spawned goroutine using a mechanism like a channel or a WaitGroup. In Go, a channel is a built-in construct that provides a way for goroutines (concurrent functions) to communicate with each other and synchronize their execution. A channel has a type and it is created with the `make` function:

```
ch := make(chan int) // A channel that carries integers
```

The keyword `chan` is the keyword for declaring a channel. The type after `chan` (e.g., `int`) defines what kind of data the channel can transport.

We use the `<-` operator to send a value into a channel.

```
ch <- 42 // Send the value 42 into the channel
```

We use the `<-` operator to receive a value from a channel.

```
value := <-ch // Receive a value from
// the channel and store it in 'value'
```

We use the close function to indicate no more data will be sent: `close(ch)`. Sending to a closed channel causes a panic. The following program would print 'Canada':

```
package main

import "fmt"

func main() {
  ch := make(chan string) // Create a channel for strings

  go func() {
  ch <- "Canada" // Send a message to the channel
  }()

  msg := <-ch // Receive the message in the main goroutine
  fmt.Println(msg)
}
```

The following program illustrates how we might use a channel to wait for a goroutine to terminate:

```
package main

import (
    "fmt"
)

func main() {
    channel := make(chan bool)

    go func() {
      fmt.Println("Canada")
      channel <- true // Signal that the goroutine is done
    }()
```

```
    <-channel // Wait for the goroutine
}
```

The goroutine sends a value (true) to the channel when it finishes. The main function blocks at `<-done`, waiting to receive from the channel, ensuring it does not exit until the goroutine completes. By default a channel is unbuffered: it can contain at most one value. So if you try to write to it more than one value, it will block until at least one value is read.

```
ch := make(chan int, 2)
ch <- 1 // Does not block (buffer has space)
ch <- 2 // Does not block (buffer is now full)
ch <- 3 // Blocks until a value is received
```

In Go, you can pass multiple channels to a function just like any other arguments. Channels are first-class values in Go, meaning they can be passed as parameters, returned from functions, or stored in variables. When passing several channels to a function, you simply include them in the function's parameter list, specifying their types. Let us consider an example where we access two URLs:

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

type Response struct {
    url  string
    status string
    err  error
}
```

```
func fetchURL(url string, ch chan Response) {
    // Create HTTP client with timeout
    client := &http.Client{
      Timeout: 10 * time.Second,
    }

    // Make HTTP GET request
    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() {
    // Record start time
    startTime := time.Now()
    // Create channel for responses
    ch := make(chan Response)

    // URLs to fetch
    urls := []string{
      "https://www.google.com",
      "https://www.github.com",
    }

    // Start goroutines for each URL
    for _, url := range urls {
      go fetchURL(url, ch)
    }
```

```
    // Collect responses
    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 the channel (optional since program ends here)
    close(ch)

    // Calculate and print elapsed time
    elapsed := time.Since(startTime)

    fmt.Printf("\nTotal time taken: %s\n", elapsed)
}
```

This program defines a Response struct to hold the URL, its status, and any error that occurred. It implements a fetchURL function that takes a URL and a channel as parameters, uses an HTTP client with a 10-second timeout, makes a GET request to the URL, sends the result through the channel. It uses defer to ensure the response body is closed. In this instance, the channel can be written to or read from in the function: to ensure that it can only be written to, we could declare it as `ch chan<- Response` instead as `ch chan Response` when passing it. In the main function, we create a channel to receive responses, we define two URLs to fetch, we launch a goroutine for each URL, we collect responses from the channel and we print the results. When we run this program, it will fetch both URLs simultaneously using separate goroutines, it will use channels to communicate results back to the main goroutine, and it will print the

status (like “200 OK”) or any errors for each URL. We can rewrite this program so that it is simpler, without goroutines, like so:

```
package main

import (
    "fmt"
    "net/http"
    "time"
)

type Response struct {
    url  string
    status string
    err  error
}

func fetchURLSynchro(url string) Response {
    // Create HTTP client with timeout
    client := &http.Client{
      Timeout: 10 * time.Second,
    }

    // Make HTTP GET request
    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 to fetch
```

```
    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)
}
```

The two programs do the same work, but one uses two goroutines (in addition to the main goroutine) while the other uses only the main goroutine. Testing these programs, you may find that the one using two goroutines completes faster: network accesses are typically expensive and easily parallelizable. That is, the two tasks can be done almost independently on your computer, even if executed simultaneously. Hence, you may find that we can query two URLs using HTTP requests in 250 ms whereas 400 ms is needed if the requests are consecutive, using a single goroutine. However, you should not assume that using more goroutines always makes software run faster. It often does not. Furthermore, additional goroutines might trigger the use of additional processors which increases the cost or power usage of your software. Adding more goroutines makes your software more complicated, more difficult to maintain and debug. Formally speaking, you do not need parallelism (i.e., many physical

processors) to execute two network requests concurrently. Executing such requests does not require much processing time and has much to do with waiting for the network response. Therefore, it is a case where using goroutines is likely appropriate. When splitting up more computational tasks into goroutines, you are less certain to get a performance boost. To illustrate the point, let us consider the case where we are summing all values in an array. We consider two cases, first a small array (100k elements) and then a large array with millions of elements. For both cases, we can either use a simple function (with one goroutine) or a function that uses multiple goroutines. To maximize parallelism, we set the number of goroutines to the number of processors detected on the system by Go (`runtime.NumCPU()`).

```
package main

import (
    "fmt"
    "runtime"
    "testing"
)

// sequentialSum calculates the sum
// of an array sequentially
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++
    }

    // Collect partial sums from the channel
    total := 0
    for i := 0; i < activeGoroutines; i++ {
      total += <-resultChan
    }
    close(resultChan)

    return total
}

// Benchmark functions
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)
    }
}

// Benchmark functions
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)
}
```

On a system with a large number of processors, we might get the following result:

```
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
```

We see that when summing up the modest array, we get that the approach using 128 goroutines takes five times longer. If it does end up using 128 processors, then it might be `128 * 5 = 640` times less efficient! The lesson is that if the task is sufficiently inexpensive, such as summing up thousands of integers, you should not use more than one goroutine. In the instance where we are summing 10 million integers, the parallelized task is more interesting: it goes 3.6 times faster. Again, the single-routine approach is likely much more efficient: a single processor takes 3.6 longer than over one hundred goroutine. The problem with a simple sum is that it is driven by memory accesses and not especially computational. What if we consider a more expensive task? Let us sum the sine of the values of an array using various numbers of goroutines (1, 2, ...). We use one million values in the array.

```
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])
    }

    // Collect results
    total := 0.0
    activeGoroutines := (len(numbers) + chunkSize - 1)
  / chunkSize
    for i := 0; i < activeGoroutines; i++ {
      total += <-resultChan
    }
    close(resultChan)
    return total
```

```
}

// Benchmarks
func BenchmarkSequential(b *testing.B) {
    numbers := make([]int, 1000000)
    for i := range numbers {
      numbers[i] = i % 1000 // Keep numbers manageable
    }

    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)
}
```

On a powerful machine with many cores, we might get the following results:

```
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
```

Going from one goroutine to two improves the speed by a factor of 1.5. Going from one goroutine to 16 goroutines improves the speed by a factor of 11. Increasing the number of goroutines beyond 16 brings no further gain. This pattern is sublinear gains with an upper limit is rather typical.

Yet goroutines and channels can be remarkably efficient in their own right. Let us create a chain of channels. Each goroutine has an input channel and an output channel. As soon as data is received in the input channel, data is written to the input channel. We link hundreds of goroutines in a chain of input and output channels:

```
package main

import (
    "fmt"
    "time"
)

func relay(input <-chan int, output chan<- int) {
    // Wait for value from input channel
    value := <-input
```

```
    // Send value to output channel
    output <- value
}

func main() {
    // Number of goroutines in the chain
    const chainLength = 10000

    // Create slice to hold all channels
    channels := make([]chan int, chainLength+1)

    // Initialize all channels
    for i := range channels {
      channels[i] = make(chan int)
    }

    // Start timing
    startTime := time.Now()

    // Create the chain of goroutines
    for i := 0; i < chainLength; i++ {
      go relay(channels[i], channels[i+1])
    }

    // Send initial value into the first channel
    go func() {
      channels[0] <- 42
    }()

    // Wait for and receive the value from the last channel
    result := <-channels[chainLength]

    // Calculate elapsed time
    elapsed := time.Since(startTime)
```

```
    // Print results
    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))
}
```

Running this program, you may get the following result:

```
Value 42 successfully passed through 10000 goroutines
Time taken: 13.987416ms
Average time per hop: 1.398µs
```

Effectively, you can traverse nearly a million goroutines per second in this manner.

Channels are commonly used in the context of file and network access. Employing multiple goroutines is often beneficial in this scenario, as the processor can execute different tasks concurrently, thereby enhancing the program's efficiency and responsiveness.

The following program creates ten text files, writes the word "Canada" to each, and then reads each file to count its byte size, displaying the results. File operations are carried out using Go's standard library. Initially, the `os.WriteFile` function creates or overwrites each file listed in the `files` array (f1.txt to f10.txt) and writes the string "Canada" as bytes, with file permissions set to 0644 (read/write for the owner, read for others). Subsequently, the `countBytes` function opens each file using `os.Open`, establishing a connection for reading. If the file cannot be opened (e.g., if it does not exist), an error is returned. The `defer file.Close()` statement ensures the file is closed after use, freeing system resources. Then, `io.ReadAll` reads the entire file content into memory as a byte array, and the length of this array (obtained via `len(data)`) provides the file's byte count. These operations are performed concurrently for each

file using goroutines, with results (filename and byte count) collected through a channel (`results`) for display.

```
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)
    }
}
```

Wait groups

Another common approach to managing multiple goroutines is the use of `sync.WaitGroup`. Before exploring an example, it is helpful to understand the role of wait groups in Go programming. A wait group is a synchronization mechanism provided by the `sync` package, which allows a program to wait for all launched goroutines to complete their execution before proceeding. In practice, a wait group maintains an internal counter. Each goroutine increments this counter at startup and decrements it upon completion. The main function can then wait for the counter to reach zero, ensuring that all asynchronous tasks are completed.

To use a wait group, three key functions are employed: `wg.Add`, `wg.Done`, and `wg.Wait`. The `wg.Add(n)` function increments the wait group counter by `n`, typically called before launching goroutines to indicate the number

of tasks to wait for. The `wg.Done()` function decrements the counter by 1, signaling that a goroutine has completed its execution. Finally, `wg.Wait()` blocks the program's execution until the counter reaches zero, indicating that all associated goroutines have finished. These functions, used together, ensure robust and predictable synchronization.

To ensure that `wg.Done()` is called even in case of an error, we use `defer`. In Go, the `defer` keyword is used to schedule the execution of a function just before the enclosing function (the one containing the `defer` statement) terminates. This approach is particularly useful for ensuring that cleanup operations, such as closing a file or decrementing a wait group counter, are performed even in case of an error or early return. Consider the following example where a division by zero causes a panic, but `defer` ensures the execution of a statement before termination.

```
package main
import "fmt"

func f(x int) int {
    defer fmt.Println("!!!")
    return 1 / x
}

func main() {
    f(0)
}
```

In this example, when `f(0)` is called, a division by zero causes a panic. However, thanks to `defer`, the `fmt.Println("!!!")` statement is executed before the program crashes, illustrating how `defer` ensures code execution even in the case of a fatal error.

The use of `defer` becomes particularly relevant when an error may occur in a goroutine, requiring systematic cleanup. Consider the following example where an error is generated, but `defer` ensures that the wait group counter is decremented.

```
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("---")
}
```

In this example, the `defer wg.Done()` statement ensures that the wait group counter is decremented, even if the goroutine terminates early due to an error, thus preventing a program deadlock. The following code would not work because `wg.Done()` would never be called, and the main function would never be notified that the goroutine has completed.

```
go func() {
    y, err := divide(0)
    if err != nil {
        fmt.Println("Error:", err)
        return
    }
    fmt.Println(y)
    wg.Done()
}()
```

To illustrate a more sophisticated use of wait groups and `defer`, consider a program that increments values in an array. In the `main` function, an array `arr` of integers is created. Four goroutines are used to divide the work, each goroutine processing a portion of the array, calculated via `chunkSize` to distribute the indices evenly. The `incrementChunk` function increments each element in a given range of the array, from `start` to `end`, and uses a `sync.WaitGroup` to synchronize the goroutines. Each goroutine calls `wg.Done()` via `defer` once completed, and `wg.Wait()` in `main` ensures that all goroutines finish before the program ends, guaranteeing that all array elements are incremented concurrently and safely.

```
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()
}
```

Atomics

If you need to read data from different goroutines, that is not a problem as long as the data remains constant. If nobody writes to the data, there is no problem. Unfortunately, we often need to change the data, while reading it from different goroutines. Sometimes you can use channels to communicate. But that is sometimes not enough. Let us consider an example. We take an array of 10 integers, and goroutines randomly decrement one array element and then increment another array element. Initially, the sum of all elements should be 1000 and it should remain 1000 unless there is a bug. We can implement our code like so:

```
package main

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

```
    "sync"
    "time"
)

func main() {

    // Initialize array with 10 elements, each set to 100
    arr := [10]int{100, 100, 100, 100, 100,
   100, 100, 100, 100, 100}
    var wg sync.WaitGroup

    // Function for goroutine behavior
    worker := func() {
      defer wg.Done()
      wg.Add(1)
      r := rand.New(rand.NewSource(time.Now().UnixNano()))

      // Run for 200000000 iterations as an example
      for i := 0; i < 200000000; i++ {
          // Pick first random index
          idx1 := r.Intn(10)
          // Only proceed if value > 0
          if arr[idx1] > 0 {
            // Decrement first element
            arr[idx1]--

            // Pick second random index
            idx2 := r.Intn(10)
            // Increment second element
            arr[idx2]++

          }
      }
    }
```

```
    // Launch two goroutines
    go worker()
    go worker()
    fmt.Println("waiting...")
    wg.Wait()
    fmt.Println("waiting...ok")

    fmt.Println("\nFinal array state:", arr)
    // Verify total sum should still be 1000 (10 * 100)
    sum := 0
    for _, val := range arr {
      sum += val
    }
    fmt.Println("Total sum:", sum)
}
```

This program is wrong: it contains data races because we are writing and reading data from different goroutines without synchronization. A possible ouput of this program is the following:

```
Final array state: [3001 644 880 324 2319 2845 3664
 160 232 1741]
Total sum: 15810
```

Observe how the sum is higher than expected.

In Go you can avoid such a bug with the guarantee of atomicity provided by the `sync/atomic` package, which ensures that operations like increments are executed as indivisible steps, preventing race conditions. Functions like `atomic.AddInt32(&x, 1)` or `atomic.AddInt64(&x, 1)` ensure that the increment operation (read-modify-write) is performed atomically. This means that even if two threads execute the increment concurrently, the operations are serialized at the hardware level, and no updates are lost.

```
package main

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

func main() {

    // Initialize array with 10 elements, each set to 100
    arr := [10]int32{100, 100, 100,
   100, 100, 100, 100, 100, 100, 100}
    var wg sync.WaitGroup

    // Function for goroutine behavior
    worker := func() {
      defer wg.Done()
      wg.Add(1)
      r := rand.New(rand.NewSource(time.Now().UnixNano()))

      // Run for 200000000 iterations as an example
      for i := 0; i < 200000000; i++ {
          // Pick first random index
          idx1 := r.Intn(10)
          // Only proceed if value > 0
          val := atomic.LoadInt32(&arr[idx1])
          if val > 0 {
            if atomic.CompareAndSwapInt32(&arr[idx1],
       val, val-1) {
                // Pick second random index
                idx2 := r.Intn(10)
```

```
                // Increment second element
                atomic.AddInt32(&arr[idx2], 1)
            }

        }
    }
}

// Launch two goroutines
go worker()
go worker()
fmt.Println("waiting...")
wg.Wait()
fmt.Println("waiting...ok")

fmt.Println("\nFinal array state:", arr)
// Verify total sum should still be 1000 (10 * 100)
sum := 0
for _, val := range arr {
  sum += int(val)
}
fmt.Println("Total sum:", sum)
}
```

The expression `atomic.LoadInt32(&arr[idx1])` atomically reads the value at array position `idx1`. The value is stored in a local variable (`val`): data races are not possible with a local variable. We then use a Compare-And-Swap (CAS) operation: `atomic.CompareAndSwapInt32(&arr[idx1], val, val-1)`. It checks if `arr[idx1]` still equals `val` (the previously loaded value) and if true, it sets `arr[idx1]` to `val-1`. It returns true if successful, false if the value changed since the load. Importantly, it executes as a single atomic operation. Finally, we use `atomic.AddInt32(&arr[idx2], 1)` to atomically add 1 to `arr[idx2]`.

If you run the new program, the sum of the values in the array is maintained. The program is safe.

Mutex

Atomic operations like `AddInt32` or `CompareAndSwapInt32` are designed for single, indivisible operations on a single variable. They become insufficient when we have more complex data structures.

In these more complex cases, we use a mutex. A mutex (short for "mutual exclusion") is a synchronization primitive used in concurrent programming to prevent multiple threads or processes from simultaneously accessing or modifying a shared resource. It ensures that only one thread (or goroutine) can enter a critical section of code at a time, thus avoiding race conditions and maintaining data consistency. Essentially, only one 'lock' can be held at any given time.

To illustrate, let us create a program where money is transferred between two accounts, and we need to ensure that the withdrawal from one account and deposit to another happen together without interference from other goroutines. This requires protecting a multi-step operation, which goes beyond what atomic operations can do.

```
package main

import (
  "fmt"
  "sync"
  "time"
)

type Bank struct {
  accounts map[string]int // Map of account IDs to balances
  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()

  // Check if source account has sufficient funds
  if b.accounts[from] >= amount {
  // Perform the transfer: two related operations
  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

    // Launch multiple concurrent transfers
    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"))
}
```

If no other goroutines are attempting to acquire the same mutex, the lock operation is fast. The mutex state is updated using atomic operations, such as compare-and-swap, which are highly optimized at the hardware level and typically complete in a few nanoseconds. When several goroutines compete for the same mutex at the same time, the speed of the lock will vary. The goroutine might enter into a spin lock: it will repeatedly check the mutex state in a tight loop for a short period, attempting to acquire the lock without immediately yielding to the scheduler. This can be efficient if the mutex is expected to be released quickly, as it avoids the overhead of parking the goroutine. However, if the mutex remains locked for an extended time, the spinning goroutine may waste CPU cycles before being parked by the Go scheduler, leading to increased latency and potential performance degradation. Once parked, the goroutine is placed in a wait queue, and the scheduler will wake it when the mutex becomes available, incurring additional overhead due to context switching. In complex cases, it is also possible to trigger a deadlock. A deadlock is a concurrency failure where threads are trapped in a circular wait for resources, unable to proceed due to mutual dependencies. We can modify our example to include a deadlock. Instead of a global mutex, we create

a mutex per account. If the goroutine acquires the source account and then the destination account, a deadlock becomes 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()

    // Get the accounts
    fromAccount := b.accounts[from]
```

```
    toAccount := b.accounts[to]

    // Lock the "from" account first
    fromAccount.mutex.Lock()
    fmt.Printf("Locked %s for transfer of %d to %s\n",
   from, amount, to)

    time.Sleep(100 * time.Millisecond)

    // Then try to lock the "to" account
    toAccount.mutex.Lock()
    fmt.Printf("Locked %s for transfer of %d from %s\n",
   to, amount, from)

    // Perform the transfer
    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)
    }

    // Unlock both accounts
    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)
  // Alice -> Bob
    go bank.Transfer("Bob", "Alice", 100, &wg)
  // Bob -> Alice

    wg.Wait() // This will never complete due to deadlock

    fmt.Printf("Final balances: Alice=%d, Bob=%d\n",
      bank.GetBalance("Alice"), bank.GetBalance("Bob"))
}
```

The deadlock in this code occurs because two goroutines acquire mutexes in different orders, leading to a circular wait. One strategy to avoid such a deadlock is to use ordered mutexes. E.g., if accounts are numbered, we always lock the account with the lesser number first.

To achieve better performance, it is sometimes preferable to replace a mutex with an RWMutex. An RWMutex, or *read-write mutex*, is a synchronization primitive that manages concurrent access to a shared resource by distinguishing between read and write operations. Unlike a classic mutex, which exclusively locks access for any operation, an RWMutex allows multiple goroutines to access the resource simultaneously for reads, as long as no goroutine is writing. This improves performance in scenarios where reads are frequent and writes are rare. When a goroutine

needs to write, it must acquire an exclusive lock (*Lock*), which prevents any other reads or writes during that operation. Reads, on the other hand, use a shared lock (*RLock*), allowing multiple goroutines to read the resource in parallel without interference.

To illustrate the use of an RWMutex, consider a program simulating a shared cache where multiple goroutines frequently read data, but updates are less common. In this example, we use an RWMutex to allow multiple goroutines to read data simultaneously while ensuring that writes are exclusively protected.

```
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)
}
```

This code implements a shared cache using an `RWMutex` to manage concurrent access by multiple goroutines. A Cache structure contains a map associating keys with values and a `sync.RWMutex` to synchronize access. The NewCache function initializes the cache with two key-value pairs. The Read method allows multiple goroutines to simultaneously read a value using `RLock`, which locks in shared read mode, displaying the read value or an error message if the key does not exist, with a simulated delay of 100 ms. The `Write` method uses Lock for an exclusive lock, enabling a cache update with a simulated delay of 200 ms. In `main`, three goroutines read the `"key1"` key in parallel, one goroutine writes a new `"key3:value3"` pair, and another reads `"key2"`. A `WaitGroup` synchronizes execution, and the final cache state is displayed.

False sharing

False sharing is a performance issue that occurs in multicore systems when multiple processors access data located in the same cache line, even if the data is distinct. In a multicore processor, data in memory is transferred to the cache in blocks called cache lines (typically 64 or 128 bytes). If two goroutines, running on different cores, modify distinct variables located in the same cache line, the cache coherence system invalidates and reloads that line with each modification, leading to significant performance losses.

In Go, false sharing can occur when using data structures shared between goroutines without considering data alignment in the cache. For example,

an array of counters incremented by different goroutines may seem independent, but if the counters are contiguous in memory, they may share the same cache line, causing conflicts.

Here is an example illustrating the false sharing issue in Go, followed by a solution to avoid it.

```
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)
}
```

In this example, we have two structures: `Counter` and `PaddedCounter`. In `Counter`, the four `counts` counters are stored contiguously in memory,

increasing the likelihood that they share the same cache line. Each goroutine increments a different counter, but since they are in the same cache line, the cores must constantly synchronize that line, slowing down execution.

In `PaddedCounter`, we add padding (an array of 15 unused `int64`, totaling 120 bytes) between each counter, ensuring that each counter resides in a distinct cache line (cache lines are typically up to 128 bytes). This eliminates cache conflicts, improving performance.

When running the code, you might obtain results like these on a multicore machine:

```
No padding - Time: 24.979625ms
Padding - Time: 9.303167ms
```

The test with padding is significantly faster because it avoids unnecessary cache line invalidations. To mitigate false sharing, it is crucial to understand data alignment in memory and use techniques like padding or separate data structures to ensure that variables accessed by different goroutines reside in distinct cache lines.

Conclusion

Concurrency is a powerful tool in modern software development, enabling programs to leverage multiple processors for improved performance. However, it introduces significant complexities that must be carefully managed. Data races, where unsynchronized access to shared memory leads to unpredictable outcomes, underscore the need for robust synchronization mechanisms. Go's goroutines and channels offer an elegant, lightweight approach to concurrency, allowing developers to efficiently parallelize tasks like network requests or data processing while avoiding the overhead of traditional threads. Yet, the performance benefits of parallelism are not guaranteed—simple tasks may suffer from excessive goroutine overhead, while computationally intensive operations can see substan-

tial gains, albeit with diminishing returns as the number of goroutines increases.

Synchronization tools like `sync.WaitGroup`, atomic operations from `sync/atomic`, and mutexes (`sync.Mutex`) provide essential safeguards against concurrency pitfalls. Atomics excel for single-variable updates, ensuring thread safety with minimal overhead, while mutexes protect multi-step operations on complex data structures. However, mutexes come with risks, such as deadlocks, which arise from circular dependencies and require careful design—like consistent lock ordering—to avoid. Choosing the right concurrency strategy depends on the task's nature, scale, and performance requirements. Ultimately, effective concurrent programming in Go demands a balance between leveraging parallelism for speed and maintaining simplicity, correctness, and efficiency in the face of shared resource contention.

Exercises for Chapter 8

Question 1

Write a Go program that counts the total number of words across multiple text strings concurrently. The program should: Take a slice of strings as input (e.g., `[]string{"Hello world", "Go is awesome", "Concurrency is fun"}`). Use a goroutine for each string to count the words in that string (a word is any sequence of characters separated by whitespace). Use a channel to collect the word counts from each goroutine. Sum the results in the main goroutine and print the total word count.

Question 2

Description: Modify the following program, which has a data race, to ensure correct behavior using the sync/atomic package:

```
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() {
            defer wg.Done()
            for j := 0; j < 1000; j++ {
                counter++ // Data race here
            }
        }()
    }

    wg.Wait()
    fmt.Println("Final counter value:", counter)
}
```

Exercise 3

Extend the bank transfer example from the chapter to support multiple accounts and prevent deadlocks. The program should define a Bank struct with a map of account names to Account structs, where each Account has a balance (int) and a mutex (sync.Mutex). You must implement a Transfer method that moves money from one account to another, locking both accounts safely.