Introduction to Programming

Learn programming algorithms and data structures using Golang

Youri Ackx

Introduction to Programming

Introduction to Programming with Go

Learn data structures and algorithms with Go

Youri Ackx

13 Apr. 2023 (preview)

Learn programming algorithms and data structures using Golang

Youri Ackx

This book is about computer programming and algorithms, without the formal academic approach. Learn about recursion, complexity, data structures, and solve classical computer science problems like the Towers of Hanoi, the Eight Queens and Conway's Game Of Life.

Contents

1 Introduction

1.1 It's about programming

This book is first about **programming, algorithms and data structures**. Of course, Go will be our reference language, and for sure, you will learn Go along the way. But the techniques presented in this book will also be transferable to a large extent to other programming languages like Python, Java or C.

We will cover one single programming paradigm: **imperative programming**. Other paradigms such as object oriented programming are not in the scope of this book.

We will take the time to understand what is going on under the hood. A tutorial will usually give you a recipe to solve a specific problem, without necessarily discuss the underlying algorithm. For instance, you can be given instructions on how to sort a list using a library or a built-in function, but it will probably not discuss how sorting a list actually works.

This book covers the equivalent of one semester or more of first year computer science class. It is **designed for beginners** who want to acquire a good grasp on algorithms and data structures. **Teachers** on the other hand can use it as a classroom support and leverage its numerous exercises, while focusing on teaching the material.

Go may seem a peculiar choice to learn programming. We will discuss the reasons of this choice and its merits, with background information on the language.

1.2 Approach

We will take a **non-formal, intuitive and practical approach** to programming, heavily based on exercises of increasing complexity, adding the minimum amount of theory necessary as we progress to solve them.

For **beginners**, the first part lays the foundation of programming. Variables, loops, conditional statements and functions found in most languages will be presented. At this stage, you will be able to solve simple problems, like checking if a paper fits in an enveloppe.

We will continue with more advanced data structures. Slices and maps are the bread and butter of Go programs. We will read data from files and manipulate them. From there, you can already go a long way and write useful programs. At that point, implementing games like Blackjack or Hangman is in reach.

We will extend these data structures to form lists, linked lists, queues, stack and trees to mention the most common ones. At this **intermediate** level, we will talk about recursion, backtracking, space and time complexity, finite automatons. We will solve both fun and classical academic problems like the eight queens problem, the towers of Hanoi, or implement the classical Conway's game of life;, and more.

1.3 Exercises

Solving exercises is a key part of the learning process.

This book offers numerous exercises. The most efficient method is to solve them completely. First, write your resolution steps on paper in pseudo-code. Then, once you are confident with what you have written on paper, write the corresponding program in Go and run it. Improve it until it is satisfactory.

Heading straight to the solution and thinking that "you get it" does not bear the same teaching value as actually resolving the exercise yourself, which may involve sweat and tears, and sometimes getting stuck. If you find yourself blocked, take a break, or try another exercise before going back to the challenging part. Only through hard personal work will you progress. Do not despair if you spend one hour on a single exercise: this is perfectly normal.

You don't have to resolve all the exercices; however you are advised to a least test your skills on some of them, including the hardest ones. You should at least be confident you *could* solve the others as well before proceeding to the next chapters.

Figure 1: Some deep thinking moments required

1.4 Vocabulary

Programmng means writing a *program* that resolves a *problem*. A program is said to be *correct* if, for every *input*, it terminates consistently with the correct *output* and solves the given problem. An *algorithm* contains a sequence of steps to solve a problem. A program formally describes an algorithm.The program is *executed* by a computer *processor*. The program is written in a *programming language* according to a predefined *syntax*. The computer processor (CPU) cannot execute it directly. It needs tools like an *interpreter* or a *compiler*.

1.5 Abstraction

In order to write programs, we need **abstractions**.

Consider a hard disk drive (HDD). It is an enclosure containing rapidly rotating disks or platters, each of which is coated with magnetic material in order to store data. Arms with magnetic heads move above the platters in order to read (and write) data that can represent an image, a song or a poem, or more generally anything we call a *file*.

Figure 2: A hard drive needs an abstraction

You could read the poem stored on the hard drive and display it on screen by sending commands to move the heads above the right platter on the hard drive, communicating with the CPU directly. But the task would be incredibly difficult to achieve. Every program you or someone else writes would have to repeat the same operations, for every single data you would want to read, and for every possible disk geometries. This would be at best a tedious task. Above all, it would be pointless.

Maybe the hard disk drive can be seen as *cylinders* and *sectors* that form some logical organization. Which it does in reality. This is a first level of **abstraction**. But this is still not the abstraction we are looking for — unless we are writing specific parts of an operating system, or a HDD controller.

For sure, we have a more casual purpose. Say we want to read the poem from the disk with as little knowledge about the underlying hardware as possible. We are interessted in retrieving the poem, not about the low-level hardware details.

Now consider the following program fragment:

```go
poem, err := ioutil.ReadFile("poem.txt")
defer poem.Close()
if e != nil {
    panic(e)
}
fmt.Print(string(poem))
```

There are several things that require an explanation. What is `err` or `panic` for instance. All in due time. For now, using a high-level programming language, we have *abstracted* the process of retrieving the information on the disk magnetic surface. In fact, the abstraction is layered: the Go compiler provides you with a first abstraction from the operating system, which in turn abstracts you for the processor, the memory and the hard disk. This is what we were looking for: a mean to **express ideas and concepts while hiding the underlying complexity and details**. It is still nice to know how a hard drive or an operating system work though. But suffice to say it is much more efficient and confortable to rely on the work done by others to access a storage medium, and focus on our program.

> *Note*: Nowadays hard disk drives (HDD) are getting replaced by solid state drives (SDD) that have no mechaninal moving parts. Moving parts or not, you still want to be abstracted from the bare medium. By virtue of abstraction, our program above will still work, no matter the actual underlying physical media, HDD, SSD or other.

1.6 First program

To get acquainted with a programming language, it is customary to write a program that displays a friendly message. For the first chapters, you do not necessarily need to have Go installed on your computer. You can simply enter an execute your program in the Go Playground:

```
package main

import "fmt"

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

This simple program already raises several questions. What is a `package`, why is there an `import` with `"fmt"` in it? One thing at the time. For now, remember that `Println` prints a line on the screen. The term “print” and its variations are found in many programming languages, more often than “display” and comes from the ancient times where the primary interface with the computer was a printer rather than a screen. `main` is the program entry point. It is where the first instruction will be executed.

This program is found by default when you open the the playground, so you don’t even have to copy-paste it. Mark this page, as this program will be the skeleton for many exercises we will solve in the next chapters.

1.7 A paper and a pen

… is all you need to write a program. For instance, let’s play hide and seek. By that, we mean to write *pseudo-code* that describes a game of hide and seek:

```
close eyes
count from 20 to 0
while buddy not found:
    seek buddy
```

Pseudo-code is an informal high-level description of the operating principle of a computer program or other algorithm. It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading.[1]

You can read this program almost literally:

- Close your eyes
- Count from 20 to 0

[1]https://en.wikipedia.org/wiki/Pseudocode

- Search for your hidden buddy
- The game ends when you find them

The `while` section is a **loop**. It means to keep looking _while_fski you haven't found your buddy.

It is helpful to describe an algorithm coarsly. But we can take the same algorithm and add more details, for instance when we countdown:

```
counter <- 20
while counter is not 0:
    counter <- counter - 1
```

Here we give more details on how to count down. Instead of just saying "countdown" and leaving the details to the reader, we explicitely say that we need to decrement our "counter" at each step. Notice that we still lack some details, like for instance the delay between each step. Is it a second, should we count as fast as we can? It is not described, but we can be more specific:

```
counter <- 20
while counter is not 0:
    counter <- counter - 1
    wait 1 second
```

Same algorithm, different levels of details. From informal to more formal. The first version is lightweight and easy to read. It leaves out the details. The thrid version leaves less room to interpretation. The second is in-between.

Both have their use. In all cases, we have left many step aside. For instance, think about what would happen if your buddy was too well hidden. As it is described, the game would go on forever as you would keep searching for them endlessly. A compiled program will gleefully execute the sequence of instructions you entered, regardless of their implications, provided that they are syntactically correct, and no matter the aberration they can produce.

Paper and pen are great tools to learn programming, but may not be very motivating. A program run on a computer gives you more feedback, interaction and gratification than paper. This being said, pseudo-code on paper or on a drawing board[2] remains a key tool before jumping to your keyboard.

[2] Of course you can use your tablet and wireless pencil.

Figure 3: Start with a paper and a pen

1.8 About Go

Before we roll up our sleeves, it is important to understand why the language we will use was developed. Let's have a look at a quote[3] [video] from Rob Pike[4], one of its creators.

> "Go was conceived as an answer to some of the problems we were seeing developing software infrastructure at Google. The computing landscape today is almost unrelated to the environment in the languages being used, mostly C++, Java and Python have been created. The problems introduced by multicore processors, networked systems, massive computation clusters, and the web programming model were being worked around rather than addressed head-on."

> "Moreover, the scale has changed: todays's server programs comprise ten of millions of lines of code, are worked on by hundred or even thousands of programmers, and are updated literally every day. To make matters worse, build time has stretched to many minutes, even hours, on large compilation clusters".

[3]Video: https://www.youtube.com/watch?v=bmZNaUcwBt4 "Go at Google: Language Design in the Service of Software Engineering" https://talks.golang.org/2012/splash.article

[4]Rob Pike https://en.wikipedia.org/wiki/Rob_Pike

The **language creators** all have major contributions to our field. Robert Griesemer[5] has worked on the V8 Javascript engine and on the Java VM hotspot; Rob Pike was on the Unix team at the Bell Lab and created the first windowing system for Unix; Ken Thompson designed and implemented the original Unix. Thompson and Pike the co-creator of UTF-8.

Go has a very **simple programming model**, something that can fill in your head easily. They purposefully avoided complex and advanced constructions. You won't find exceptions, which can be surprising at first. Generics were added only recently. They have a "less is better" approach and they are extremely wary not to add unnecessary features to the language.

As a result, you can learn Go in only a few days.

Some of the **cruft** accumulated by older languages has been removed, like the need to specifically declare a data type, although a modern compiler should not need you to do so. Back then Java did not have type inference for instance, although now it does.

Go also strongly favours certain **idioms**, and that's an excellent thing to achieve higher consistency, and to avoid endless debates on personal tastes. The `go fmt` tool settles the debate about formatting. You just do it the Go way so to speak.

It is a **garbage collected** language, so you don't have manage memory allocation yourself.

It is built with **concurrent programming** in mind, with its *coroutines* and *channels*. **Testing** also has a key place with dedicated constructs.

Go compiles to **native code** and allows **cross compilation** to another platform. So for instance one can create a executable for GNU/Linux on a macOS machine.

The tools are well designed and well thought, and so is the language. There is a decent amount of libraries available. The **memory footprint** is extremely small. It is very capable to handle heavy loads.

It was designed to run "server-side" programs, and although solutions exist to plug Go to a GUI, its sweet spot is on the server side. Youtube relies amongst other things on a Google project called vitess, and vitess is written in Go[6]. This gives you an indication that Go delivers.

From a career perspective, Go is also is a sensible choice[7].

[5]Robert Griesemer https://de.wikipedia.org/wiki/Robert_Griesemer
[6]FOSDEM 2014 https://blog.golang.org/fosdem14
[7]Survey on StackOverflow https://insights.stackoverflow.com/survey/2017#top-paying-technologie

2 Basic data types

2.1 Definition

A data type is a classification that specifies which type of value a data has and what type of operations can be applied to it.

In our first program, `"Hello, world!"` data type is a string of characters, or simply a *string*. `42` is a number.

There are many ways to categorize and group data types. For instance, *numbers* can be further subdived in *integers* (whole numbers like `1`, `42`) and *floating points* or *floats* (like `1.234`). But you may encounter the more mathematical term *Real* to designate them. It depends on the context and… the language.

Every programming languages has **primitives**. They are the data types from which all other data types are constructed. To make matters more confusing, some primitive data types may be considered *derived* primitive data types.

The Go language specification[8] should settle the debate.

In this chapter, we shall have a look at some basic datatypes.

2.2 Bits and bytes

It is a well-known fact that computers only understand "ones and zeros". The most basic unit of information that the computer in store is called a **bit**. A bit can hold two possible values: `1` or `0`. Actually, one and zero are merely conventions. Instead we could use `on` and `off`, or `true` and `false`. We'll stick to the long standing convention though.

Dealing with `1` or `0` exclusively would be extermely cumbersome, even for a computer processor. Bits are usually[9] grouped by **8** to form a **byte**.

[8]https://go.dev/ref/spec#Types

[9]It is actually more complex than that. Abate can technically be of any size. But the common definition is to use eight bits.

1	1	1	0	0	1	0	1
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

It gives us 2^8 or 256 possible combinations.

What does the above byte represents? Typically, an positive number (unsigned integer). To convert it to a decimal value, multiply each bit by 2^k, with k equal to the bit position. In our example:

1	1	1	0	0	1	0	1

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$$1.2^7 + 1.2^6 + 1.2^5 + 0.2^4 + 0.2^3 + 1.2^2 + 0.2^1 + 1.2^0$$
$$= 1.128 + 1.64 + 1.32 + 0.16 + 0.8 + 1.4 + 0.2 + 1.1$$
$$= 128 + 64 + 32 + 4 + 1$$
$$= 229$$

2.3 Numeric

Numeric types include *integer*, *floating-point* and *complex* types.

Consider the following program:

```
package main

import "fmt"

func sumOfSquares(a, b int) int {
    return a*a + b*b
}

func main() {
        fmt.Println(sumOfSquares(3, 2))
}
```

It computes and prints the sum of the squares of two *integers*.

$$a^2 + b^2, a = 3, b = 23$$

$$3^2 + 2^2 = 3.3 + 2.2 = 13$$

The function `sumOfSquares` deals with the **`int`** type to compute the sum of the squares of to integers. It makes sense. You can compute the square of `3` but not the square of an orange. `sumOfSquares` does not accept types other than **`int`**. Try to replace the main call by:

```
sumOfSquares("hello", 2)
```

If we try to call the function with `"hello"` as a first argument, Go won't even run our program. The compiler will analyze it and rightfully complain:

```
cannot use "hello" (untyped string constant) as int value in argument to
    sumOfSquares
```

It will not go any better if we use a floating point number:

```
sumOfSquares(2.5, 2)
```

```
cannot use 2.5 (untyped float constant) as int value in argument to
    sumOfSquares
```

Next to the **`int`** type, we have discovered the **`string`** (more on that later) and the `float` data types simply by looking at the compiler error message. There are actually many more predeclared types[10] in Go. There is a granularity to integer types, ranging from 8 to 64 bits representation. An 8 bit integer can hold 2^8 or 256 different values, as we have already seen.

```
         the set of all...
uint8    unsigned  8-bit integers (0 to 255)
uint16   unsigned 16-bit integers (0 to 65535)
```

[10]https://golang.org/ref/spec#Types

```
uint32   unsigned 32-bit integers (0 to 4294967295)
uint64   unsigned 64-bit integers
           (0 to 18446744073709551615)
```

Next to these types that can hold **unsigned** integers, there are corresponding **signed** integers. They have the same number of bytes, can hold as many different values, only the range differ. For instance, while the value of an unsigned integer `uint8` range from 0 to 255, a signed integer `int8` hold values ranging from -128 to $+127$.

```
         the set of all...
int8     signed  8-bit integers (-128 to 127)
int16    signed 16-bit integers (-32768 to 32767)
int32    signed 32-bit integers (-2147483648 to 2147483647)
int64    signed 64-bit integers
           (-9223372036854775808 to 9223372036854775807)
```

Type `int` alone is an odd duck, as it will be either 32 or 64 bits, depending on your platform.

The situation is slightly simpler with “non integers non complex” numbers, as there are only two types `float32` and `float64` IEEE-754 32-bit floating-point representations. You can represent very large and very small numbers (in absolute value) but… with a finite precision. This is not a Go-specific limitation as floating-point aritmethic is very common among programming languages.

WARNING: Never use a floating point values to manipulate financial amounts, whatever the language.

2.4 Strings

In Go, a *string* is a defined as a *slice of bytes*. For now, consider it as a (possibly empty) **sequence of bytes**. The package `strings` contains many utility functions to manipulate them, and the `fmt` package contains numerous functions to format them.

2.4.1 Formatting

Package fmt offers formatting functions analogous to C’s `printf` and `scanf`. The format “verbs” are derived from C. Suppose you execute the following code:

```
fmt.Println(1.0/3)
```

The output would be:

```
0.3333333333333333
```

It is convenient to be able to round to a certain number of digits, while keeping the original arbitrary precision of the variable. Let's say we want to display the number with 2 digits. This can be achieved as follows:

```
fmt.Printf("%.2f", 1.0/3)
```

Which would display:

```
0.33
```

Notice we've used `Printf` (print formatted) instead of `Println` (print line). There is something missing however. If we were to attempt to print the value of a second expression, it would be displayed on the same line as the first. With `Printf`, we need to explicitely add a *new line* at the end, with the `\n` symbol.

```
fmt.Printf("%.2f\n", 1.0/3)
```

Go offers a large amount of formatting options as you can read in the `fmt` documentation.

2.4.2 Length and indices

Let us declare a variable to hold a message:

```
message := "1, 2, 3, Go"
```

The number of bytes is called the length of the string and is never negative. It can be determined by using the built-in `len` function.

```
fmt.Println(len(message))
```

Will output:

```
11
```

The string's bytes can be access by **indices**. Like in most programming languages, Go starts counting from 0 rather than from 1. The first character is therefore located at index `0`. If the length of the string is `n`, the last byte index is `n-1`. Our `message` indices range from `0` to `10` included – `10` being the 11th and last byte.

Individual bytes are accessed using square bracket notation. For instance, the "G" of "Go" is the 10th character, located at index `9`:

```
g := message[9]
fmt.Println(g)
```

```
71
```

The result is somewhat unexpected. Why would the program output `71` rather than `G`? It turns out that 71 is the ASCII code for `G`. Check out the ASCII table[11]. If we want the character `G` to be displayed, we need to print formatted with `Printf` and the appropriate verb `c`, rather than using `Println`.

```
fmt.Printf("%c\n", g)
```

Attempts to access an index below `0` or beyond the last byte result in an error:

```
panic: runtime error: index out of range [100] with length 11
```

We can use a loop to iterate over the bytes in the string. The *for* loops starts at index *0*, and keeps running as long as the index *i* is less than the length of the string. At each iteration, we increment our index *i*.

```
message := "Hello"
for i := 0; i < len(message); i++ {
    fmt.Printf("%c", message[i])
}
```

There is a different form of **`for`** loop that does not require to specify that we start at *0* and terminate at *n* while incrementing (adding 1) at each step:

```
message := "Hello"
for i, c := range message {
    fmt.Printf("%d -> %c\n", i, c)
}
```

This construct with the help of **`range`** will assign the current index to `i` and the current character to `c`. We can then display the current index and the corresponding character.

[11]ASCII is an old is still widely used way to encore characters. See this table: http://www.asciitable.com/

```
0 -> H
1 -> e
2 -> l
3 -> l
4 -> o
```

This shorter form is to be preferred over the former whenever suitable.

2.4.3 Package strings

The package "strings" (with an *s* at the end, not to be mixed with the type *string*) offers a plethora of functions to manipulate functions[12]. A few of them will come very handy in this book.

Function	Description
`Contains`	Reports whether a substring is within s
`Count`	Counts the number of times a substring appears in s
`Index`	Returns the index (position) of the first instance (occurrence) of a substring in s
`Repeat`	Returns a new string consisting of count copies of the string s
`Replace`	Replaces a substring in a given string
`Split`	Splits s into substrings separated by a given separator
`ToLower`	Returns s with all letters mapped to their lower case
`ToUpper`	Returns s with all letters mapped to their upper case

We have slightly simplified the original definitions for readability. Refer to the official documentation for a more accurate and formal description. Let's see them in action.

```go
package main

import (
    "fmt"
    "strings"
```

[12] https://golang.org/pkg/strings/

```
)

func main() {
    fmt.Println("Contains:  ", strings.Contains("Hello", "llo"))
    fmt.Println("Count:     ", strings.Count("Hello", "l"))
    fmt.Println("Index:     ", strings.Index("test", "e"))
    fmt.Println("Repeat:    ", strings.Repeat("*", 5))
    fmt.Println("Replace:   ", strings.Replace("Hello", "l", "r", -1))
    fmt.Println("Split:     ", strings.Split("a-b-c-d-e", "-"))
    fmt.Println("ToLower:   ", strings.ToLower("Hello"))
    fmt.Println("ToUpper:   ", strings.ToUpper("World"))
}
```

```
Contains:   true
Count:      2
Index:      1
Repeat:     *****
Replace:    Herro
Split:      [a b c d e]
ToLower:    hello
ToUpper:    WORLD
```

Take a moment to browse the *strings* package documentation.

2.4.4 Immutability

Strings are immutable: once created, it is impossible to change the contents of a string. Therefore, the following attempt to modify the string's first character.

```
func main() {
    message := "hello"
    message[0] = "H"
}
```

Will result in an error:

```
./prog.go:6:13: cannot assign to message[0]
```

2.5 Overflow

If you attempt to create a numeric value that is outside of the range that can be represented with a given number of digits, an **overflow** will occur. Typically, the result will "wrap around" the maximum.

```go
func add(a, b uint8) uint8 {
    return a + b
}

func main() {
    fmt.Println(add(250, 10))
}
```

```
4
```

(From now on, we shall omit the obvious `package main` and `import "fmt"` from our examples.)

In this example, we are trying to add 10 to 250. The resulting 260 exceeds the capacity of `uint8`. When computing, Go reaches 255 and wraps back to 0, hence a result of 4. This can be shown with the following binary addition. It works like a the decimal addition you know, with carry, only with 2 digits. Bits are grouped by 4 for readability. As you can see, the result has a 9th bit on the left that won't fit in `uint8`. It gets dropped, resutling in 00000100 or 4.

```
    --uint8--
    1111 1010 (250)
+   0000 1010 (5)
= 1 0000 0100 (260)
    0000 0100 (4)
```

It is up to the program author to make provisions to avoid such errors. For instance, by modifying the return type so that it can always hold the result.

> Exercise: modify the program to return a larger integer, and test it.

That condition will be most likely unanticipated, leading to an incorrect or undefined behavior of your program. This can have dire consequences.

On June 4th, 1996, an Ariane 5 rocket bursted into flames 39 seconds after liftoff[13]. The explo-

[13] https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure

sion was caused by a **buffer overflow** when a program tried to stuff a 64-bit number into a 16-bit space. Sounds familiar? It is estimated that the explosion resulted in a loss of US$ 370m. Fortunately there was no crew on board.

Figure 4: Ariane explosion was caused by buffer overflow

Sometimes, the result is less harmful. In 2014, a popular video clip caused the Youtube view counter to overflow[14], forcing Google to switch from 32 to 64 bits integer. A number of views greater than 2 billions had not been anticipated.

The consequences were far greater in the former case than in the latter. Depending on your context, you may need to be extremelly wary, or you can afford to remain relatively casual while designing and testing your program.

We can reason about a the safety of a datatype (or of a data structure) based on its purpose.

An application dealing with “small” amounts may be confortable with `int32`. The numbers supported by `uint64` seem to go even beyond a banker’s wildest dreams, although today sky-high numbers in the financial world cast a doubt on the safest assumptions. Imagine you manipulate cents rather than units in order to avoid dealing with decimal numbers. You would multiply

[14]https://arstechnica.com/information-technology/2014/12/gangnam-style-overflows-int_max-forces-youtube-to-go-64-bit/

every number by 100. Or even by 10000 to safely manipulate 4 decimals. And suppose you do computations on a currency like japanese yen currently at 1 JPY for 0,008 EUR, leading to further multiply values by about 1000. Say you have to deal with consolidated results in the billions of euros, converted to yens, counting in cents.

How safe is your initial assumption now?

To safelhy manipulate large numbers, as Go has dedicated implementations for big numbers[15]. But they come at the cost of convenience, readability and performance. That is why they are not your go-to solution in all contexts.

2.6 Abstraction vs low level

Why not simply manipulate "integers"? Why "floating point arithmetic" and different integers? After all, we mentionned the importance to abstract ourselves from the underlyting platform. Some languages only expose "integers" and "decimals", but it comes with a substantial performance cost. Go integers types are closer to the hardware architecture. That is a trade-off the languages authors decided to do, based on the intent and purpose of the language, where high performance is key.

From a teaching perspective, this design choice gets a bit into our way as it clutters the explanations, at least at the begining. On the bright side, as far as learning goes, you are exposed to technical underlying details that would otherwise remain hidden, and you can already get a grasp at them.

[15] https://golang.org/pkg/math/big/

3 Programming blocks

3.1 Variables and constants

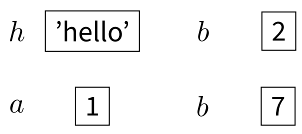
A **variable** is a storage place in the computer memory used by a program. It has a name that identifies it. For instance, we could perform the following sequence of instructions in pseudo-code:

```
h <- 'hello'
a <- 1
b <- 2
score <- a + b + 4
```

Which would result in:

- The variable `h` contains the string of characters `'hello'`
- The variable `a` contains the value `1`
- The variable `b` contains the value `2`
- The variable `score` is the sum of `a`, `b` and `4`, that is `7`.

Or visually:



The equivalent in Go is to declare the variables, and to assign them a value.

```go
func main() {
    h := "hello"
    a := 1
    b := 2
    score := a + b + 4
}
```

Notice the `:=` operator to assign a value to a variable.

Assigning variables does not do much. If you were to enter this code snippet in the program main method, the program would execute but nothing would be displayed. Go ahead and try it in the Go Playground.

Play with this small program and try to add `"hello"` and `2`. You will get an error message, as one can of course only perform additions on numbers.

Of course, we can print the results. We can modify our program so that it displays the sum of three numbers.

```
fmt.Println(sum)
```

Our modified program would unsurprisingly produce the following output:

```
7.5
```

As its name implies, a variable can *vary*. Or more precisely, the value it holds can vary.

```
a := 7
a = a + 2
```

What is going on?

- After the first instruction, `a` holds the value `7`
- After the second instruction, `a` holds the value `a+2`, that is `7+2`, which evaluates to `9`.

As opposed to variables, we can define **constants** whose values cannot change once they are set.

```
const a := 5
a = 6   // won't compile, already set
```

3.2 Conditional statements

3.2.1 If-else

Programs perform different computations or actions depending on whether a condition evaluates to true or false via a mechanism called **conditional statement**. We can express this in pseudo-code:

```
if condition:
    action1
    action2
else:
    action3
    action4
```

Notice the indent, as it is not accidental. The `action1` and `action2` will only be executed **`if`** the condition is met, a.k.a if it evaluates to **`true`**. Or **`else`**, `action3` and `action4` will be executed.

What would our `condition` actually be? We could for instance check that “a is geater than 10”, expressed as `if a > 10`.

You can have any number of instructions in the “if” and in the “else” block. Let’s see an example in Go:

```
a := 10
b := 5
if isDivisible(a, b) {
    fmt.Printf("%d is divisible by %d\n", a, b)
} else {
    fmt.Printf("%d is not divisible by %d\n", a, b)
}
```

Provided some `isDivisible()` function we haven’t defined here, the program would display “10 is divisible by 5” if 10 is actually divisible by 5 (which it is). The part of the program displaying “10 is not divisible by 5” would never be executed.

Go uses curly braces { and } to define a block of actions. In this case, each block contains a single statement that prints a message.

The *else* part is optional. We could simply do nothing if the “if” condition was not met.

3.2.2 Example - coffee

Here is a more practical example, in pseudo-code:

```
if no coffee:
    exit house
    buy coffee
    enter house
pour coffee
drink coffee
```

In this example, we will pour the cofee and drink it whether we have some left in the first place. The condition `if no coffee` applies to the block represented by the 3 instructions below it, **in-**

dented. So we will exit house, buy coffee and enter the house only if there is no coffee left. The two last actions *pour coffee* and *drink coffee* are outside the "if" block.

Coffee is easy. Let's combine it with bread.

```
if no bread or no coffee:
    exit house
    if no bread:
        buy bread
    if no cofee:
        buy cofee
    enter house
slide bread
pour coffee
eat bread
drink coffee
```

Several blocks can be constructed. Copy-paste the following code in the Go Playground and run it with different values for `a` to get different output.

```go
a := 10
if a > 10 {
    fmt.Printf("%d is greater than 10\n", a)
} else if a < 0 {
    fmt.Printf("%d is less than 0\n", a)
} else {
    fmt.Printf("%d is between 0 and 10\n", a)
}
```

Blocks can also be nested:

```go
a := 1000
if a < 0 {
    fmt.Printf("%d is negative\n", a)
} else {
    if a > 100 {
        fmt.Printf("%d is a large number\n", a)
    }
}
```

3.2.3 Example - bakery

Let's see a variation of the coffee examples.

```
if no bread:
    exit house
    timer <- 10 minutes
    while timer > 0
        search bakery
    if barkery found:
        buy bread
    enter house
if bread:
    slide bread
    eat bread
```

In this fictive example, we have two “big” chuncks: one where there is no bread, and one where there is.

If there is no bread, we exit the house and start searching for a bakery. We have put a limit of 10 minutes on the “search bakery” action, expressed casually with a “*while*” loop, so that we do not end up roaming the streets endlessly. This means we may not have found a bakery after the time is up. Therefore, we put a condition “bakery found” on the “buy bread” action.

If we omitted this condition, and if there was no bakery found after 10 minutes, our program would attempt to buy bread out of thin air, and would *terminate abnormally*, a.k.a. it would crash to say things colloquially.

In any case, bakery found and bread found or not, we return to our house afterwards.

The second part is were there is bread, in which case we slice it and eat it. This can happen in two cases:

- Either we have bread from the beginning. We did not have to step out of the house, search and buy bread.
- Or we had no bread upfront, we went out, found the bakery and bought the bread.

There is one case were we don’t slide and eat the bread: if we did not have any bread to start with, went out, did not find a bakery, and returned home wihtout bread.

That’s it! We have covered explored all the cases, and hopefully we don’t run the streets forever, we don’t try to buy bread out of nowhere and we don’t try to slice bread we don’t have. Failing to cover any of these cases would result in a **program failure**.

Note: we intentionnally left the case where there is no bread in the bakery. It is a valid assumption for this illustration, as we are free to state that a bakery will always have bread in store. In real-life, you would have to take care of that case too.

3.3 Loops

A **loop** repeats an action or a set of actions *while* a condition is met.

Sometimes, it can be a negative condition, e.g. “while countdown *is not* 0”. A program often combines loops and conditional statements, like the coffee example we just saw.

In Go, all loops are performed using the `for` instruction. The following program count downs from 10 to 0.

```
n := 10
for n >= 0 {
    fmt.Println(n)
    n = n - 1
}
```

On each step (called **iteration**), the program checks if the condition `n >= 0` (*is n greater or equal to 0?*) holds. If it does, it executes the instructions in the “for” block, that is it prints the current value of the counter and decrements it.

A general form of a `for` statement is the following:

```
"for" [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] Block
```

where

- `InitStmt` is an initialization statement, performed once before starting the loop;
- `Condition` is the loop condition that must hold true to continue to iterate, for instance `i < 10`;
- `PostStmt` is a statement executed after each execution of the loop. Typically we increment a counter.

For example, the following program will compute the sum of all integers from 0 to 10 excluded, and print it.

```
sum := 0
for i := 0; i < 10; i++ {
    sum += i
}
```

```
fmt.Println(sum)
```

- Init statetment is `i := 0`. It is executed once;
- End condition is `i < 10`, aka we will perform the loop while `i` is less than 10;
- After each execution of the block, `i` is increased by 1 (**incremented**) with the instruction `i++`, a shortcut for `i = i + 1`.

3.4 Functions

Suppose we want to display the sum of the squares of any two numbers.

$$f(a, b) \to a^2 + b^2$$

We could do the following:

```
func main() {
    a := 2
    b := 3
    fmt.Println(a*a + b*b)
    a := 5
    b := 4
    fmt.Println(a*a + b*b)
}
```

You can see we are repeating ourselves. Instead of writing the same formula `a*a + b*b` over and over again, we could write a **function**. We have already used functions, namely the one called `main`. It is easy to create our own. We shall call it `sumOfSquares`:

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

We say `sumOfSquares` is a function that take takes two **parameters** `a` and `b`. They are of **type** `int`, which stands for *integer*, and it returns another *integer*. Indeed, as our **arguments** `2` and `3` are integer values. The function **returns** another integer, which the value that was computed (the sum of squares). We can invoke it and print its output:

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

func main() {
    fmt.Println(sumOfSquares(2, 3))
    fmt.Println(sumOfSquares(5, 4))
}
```

With a predictable result:

```
13
41
```

Our function `sumOfSquares` is reusable. Our `main` methods nately calls twice, hiding the calculation details and providing a higher level of abstraction.

3.5 Arithmetic operators

3.5.1 Standard operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators (+, -, *, /) apply to integers and floating-points.

`+`	sum	integers, floats, complex values, strings
`-`	difference	integers, floats, complex values
`*`	product	integers, floats, complex values
`/`	quotient	integers, floats, complex values
`%`	remainder	integers

You are familiar with the standard arithmetic operators. However the remainder operator may be new to you. If we divide 7 by 2, the result will be 3, and the remainder will be 1, because `7 == 2*3 + 1`.

```
7 / 2
3
```

```
7 % 2
1
```

It gets tricky with negative numbers; refer to the Go documentation and experiment if you want to know more.

3.5.2 Concatenation

The `+` operator can also be applied to two strings: `"Hello, "+ "world!"`. In this case, rather than being *added*, the strings are *concatenated*.

3.5.3 Bitwise and shift

In a later chapter, we will present *bitwise logical operators* and *shifts*, after we cover binary representation.

3.6 Expressions

3.6.1 Examples

An intuitive way to start programming is to have a look at expressions. `"Hello, world!"` we used in our first program is an expression. `42` is another one. `3+4` a third. The basic arithmetic operations can be performed with `+`, `-`, `\*` and `/`. You can perform complex operations, with parenthesis if you need. The `*` and `/` operators take precedence over `+` and `-`.

We can go further. All of the following are expressions:

```
sumOfSquare(2, 3)
sumOfSquare(2 * 4, 3 * 2)
sumOfSquare(sumOfSquare(2, 3), 3 * 2)
```

```
7 + 2
42
```

3.6.2 Definitions

Expressions in Go are formally defined in the language specification. In the **expression** `7+2`, `+` is the **operand**. `7` and `2` are its **operators**. The operator's **arity** is 2.

Expression:

> An expression specifies the computation of a value by applying operators and functions to operands.

Operand:

> Operands denote the elementary values in an expression.

Arity:

> The number of arguments or operands that a function takes.

Binary operator:

> An operator that applies to two operands. Its **artity** is 2.

Unary operator:

> An operator that applies to one operand. The expression `-7` has an operator - (negate) with one operand `7`. Its **arity** is 1. In this case, the operator - (of arity 1) is not to be confused with the minus mathematical operator (of arity 2).

3.6.3 Evaluation

Let's take a non trivial yet simple case and evaluate `3+4`. The expression is evaluated to `7` after the following steps:

```
3 + 4
7
```

Our `sumOfSquare` with two integers will be evaluated as follows:

```
sumOfSquare(2, 3)
(2 * 2) + (3 * 3)
4 + (3 * 3)
4 + 9
13
```

NOTE: Notice the use of parenthesis. Although mathematically not mandatory in this case, as multiplation takes precendence over addition, they denote a group to be evaluated regardless of arithmetic precedence considerations.

In order to evaluate `sumOfSquare(2 * 4, 3 * 2)`, `2*4` and `3*2` must be evaluated first. Then we fall back on the case of `sumOfSquare` with two integers as parameters we already know.

```
sumOfSquare(2 * 4, 3 * 2)
sumOfSquare(8, 3 * 2)
sumOfSquare(8, 6)
(8 * 8) + (6 * 6)
64 + 36
100
```

Expression evaluation will come in handy at a later stage, when we examine the complexity of an algorithm.

3.6.4 Boolean expressions

A Boolean expression is **a logical statement that is either true or false**.

The expression `3 < 5` is evaluated as **`true`** while `2 < 0` is evaluated as **`false`**. You can assign a boolean expression to a variable, for instance `a = 3 < 5`. In that case, `a` will be evaluated to **`true`**.

You can test two values for equality with `==` as in `3 == 3` which of course is **`true`**. The double equal is used to avoid confusion with a variable assignment, as in `a = 3`. Which can be confusing in itself. Actually, most languages use `==` to perform equality comparisons, for historical reasons. If you want to check that two expressions are different, as in "≠", you would use `!=`, like so: `3 != 5` which evaluates to **`true`** as 3 is not equal to 5.

Math	Go	Meaning
>	`>`	Greater than
≥	`>=`	Greater or equal
<	`<`	Less than
≤	`<=`	Less than or equal
=	`==`	Equal
≠	`!=`	Not equal

Although it would be pretty useless, for illustration purpose, we can write a function `isGreaterThan` that checks if its first argument is greater than its second.

```
func isGreaterThan(a int, b int) bool {
  return a > b
}

a := 3
```

With for instance the following evaluation steps:

```
isGreaterThan(5, 3)
5 > 3
true
```

3.7 Bitwise logical operators

Bitwise logical operators perform logical operations on individual bits of binary numbers. These operators work at the binary level, manipulating individual bits. In binary, each bit can have one of two possible values, represented as `1` and `0`, or `I` and `O`, or **`true`** or **`false`**.

Before we work with numbers, let's first work on single bits. Let's have two of them: `p` and `q`. The operators are:

&	bitwise AND

|	bitwise OR
^	bitwise XOR
&^	bit clear (AND NOT)

3.7.1 Tables of truth

p	q	p & q	p | q	p ^ q	p &^ q
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	0

Now that you understand how binary operators work on single bits, it is easy to see how they operate on integers once you represent the integer in their binary form.

3.7.2 Binary numbers

Given 12 and 10 in decimal, represented in binary, one above the other:

```
1 1 0 0 (12)
1 0 1 0 (10)
```

We can perform the bitwise operations on each pair of bits taken vertically. We will say that two bits “have the same place” if they have the same significance or weight, that is, if they both are on the same vertical imaginary line.

Bitwise AND `&` will look for bits with the same significance (same place) only to keep bits that are “1” in both integers:

```
  1 1 0 0
& 1 0 1 0
= 1 0 0 0
```

The binary result `1000` is `8` in decimal.

Bitwise OR will keep bits that are "1" *at the same place* in either integer, or in both:

```
  1 1 0 0
| 1 0 1 0
= 1 1 1 0
```

The binary result `1110` is `14` in decimal.

Bitwise XOR will keep bits that are "1" *at the same place* in either integers, but not in both:

```
  1 1 0 0
^ 1 0 1 0
= 0 1 1 0
```

Bitwise AND NOT will keep bits that are "1" for the first bit and "0" for the second:

```
   1 1 0 0
&^ 1 0 1 0
=  0 1 0 0
```

3.7.3 Shift

```
<<   left shift              integer << unsigned integer
>>   right shift             integer >> unsigned integer
```

This is simply a matter of moving bits to the left (`<<`) or to the right (`>>`) as specified number of times.

`5<<2` means:

- Take the binary representation of `5`: `101`
- Move the bits to the left, two times, filling with `0` on the right as you go: `10100`
- The result is `10100` in binary or `20` in decimal.

`5>>2` means:

- Take the binary representation of `5`: `101`
- Move the bits to the right, two times, dropping the bit on the right each time: `10` then `1`
- The result is `1` in binary or `1` in decimal.

Attempting to shift a negative number of times results in an error.

An interresting property of shifting `<<1` is that it multiplies the original value by 2. Conversly, `>>1` divides by 2.

3.7.4 Exercises

Calculate the following:

- `128^255`
- `128>>4`
- `64|32`

4 Lists, Arrays and Slices

4.1 Arrays

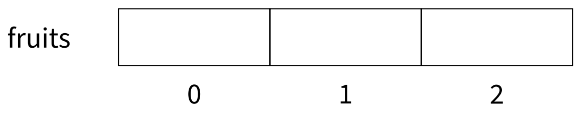
Suppose you want to manipulate several strings of characters. For instance, a shopping list, where each string is an item. So far we have used variables to contain numeric (integer or floating point) and string values. They were single values. You can declare several variables to hold several values, but this will quickly become cumbersome if the number of items on our list is not known in advance, or if there are a large amount of items to purchase.

All high-level languages offer some data structure to that effect. In many instances, the basic building block is called an **array**, defined as a *collection of elements* (values or variables), each identified by an *index* (plurals *indices*).

The following expression declares a variable `fruits` as an array of 3 strings:

```
var fruits [3]string
```

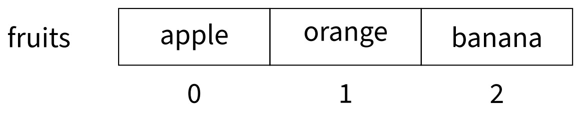
There are no fruits in the array yet, or more precisely, each fruit is the empty string.



You can assign actual values to an index:

```
fruits[0] = "apple"
fruits[2] = "banana"
fruits[1] = "orange"
```

Which can be represented as:



The first element is at position 0. Using 0 rather than 1 as the index of the first element is a widely spread convention in computer programming. There is no obligation to fill all the slots, nor to fill them in any particular order, as shown in the example above. You can also declare the array with its elements:

```
fruits := [3]string{"apple", "banana", "orange"}
```

The compiler can count the elements for you, eliminating the need to explicitly declare how many of them are present by using `...`, like so:

```
fruits := [...]string{"apple", "banana", "organge"}
```

The array has a length, accessible with `len`

```
fmt.Println(len(fruits))
```

```
3
```

An array cannot be resized. Not to worry, in Go there is a more potent data structure at our disposal called *slice*.

4.2 Slices

Arrays are a key construct but they are a bit limited in flexibility. In Go, *slices* are usually preferred. They are built on top of arrays, leveraging their performance, while adding a lot of convenience.

A slice can be declared like an array, without counting the elements. Notice the subtle difference: we use `[]` for slices when we had `[...]` or `[3]` for arrays. So the following is a *slice*, not an *array*:

```
fruits := []string{"apple", "banana", "orange"}
```

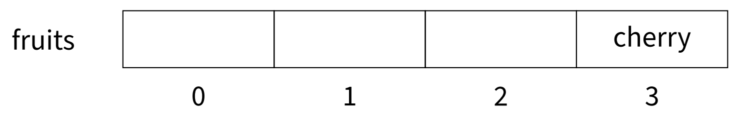
If you do not want to declare the elements yet, a slice can be created with the `make` function.

```
fruits := make([]string, 3)
```

This declares a slice of 3 elements, each initialized to the empty string. Other than that, you can access and modify the slice elements just like we did with arrays. In addition, you can **add** an element at the end of a slice thanks to aptly-named the built-in `append` function.

```
fruits = append(fruits, "cherry")
// len(fruits) == 4
// fruits[3] == "cherry"
```

Resutling in a slice of 4 elements, of which the three first are empty.



Notice that we assigned the result of the **append** expression back to `fruits`. The **append** statement alone does not modify our slice of fruits ; instead, it returns a modified slice. On top of that, the Go compliler rightfully prevents any attempt to execute a statement if it is not used. Therefore, any attempt to compile the following code would fail, because the result of the **append** call would be left unused:

```
append(fruits, "cherry")

./prog.go:9:8: append(fruits, "cherry") evaluated but not used
```

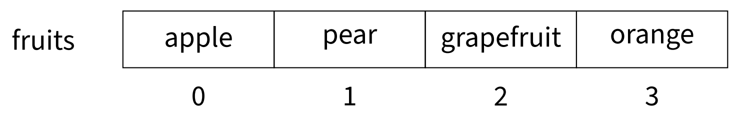
Go will however allow a statement without effect if the result is explicitly discarded.

```
_ = append(fruits, "cherry")
```

Of course, this last expression does probably not make sense in a program, but you are still free to shoot yourself in the foot. The compiler will detect many errors made in "good faith", but not gross or even intentional errors.

If you have two slices `a` and `b`, you can append one to the other, using ... to expand the second slice to a list of arguments.

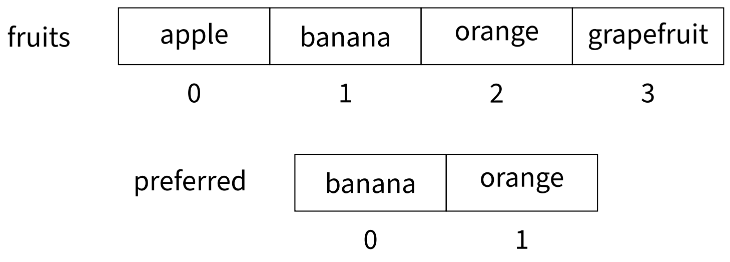
```
a := []string{"apple", "pear"}
b := []string{"grapefruit", "orange"}
a = append(a, b...)
// a == []string{"apple", "pear", "grapefruit", "orange"}
```



A slice can be created by "**slicing**" another slice (or array), that is to say, by taking a portion of it. Slicing is achieved by specifying a half-open range $[low, high)$ where low is the index of the first element (included) and $high$ is the index of the last element (excluded). For example, `fruits[1:3]` will create a new slice including the fruits at indices from `1` included to `3` excluded — that is, fruits at positions 1 and 2.

The newly created slice will have indices `0` and `1`, not the indices `1` and `2` it has in the original slice.

```
fruits := []string{"apple", "banana", "orange", "grapefruit"}
preferred := fruits[1:2]
// preferred == []string{"banana", "orange"}
// preferred[0] == "banana"
// fruits[1] == "banana"
```



Start and end indices can be omitted. In `s[low:high]`, the default value are `0` for `low` and `len(s)` for `high`:

```
// fruits[2:] == []string{"orange", "grapefruit"}
// fruits[:2] == []string{"apple", "banana"}
// fruits[:] == fruits
```

WARNING: Beware that slicing **does not copy data**. The makes the slicing operation efficient, but it may not be what you expect.

Take the following example. Notice how, by modifying `letters`, we also modified `twoFirst`:

```
letters := []byte{'a', 'b', 'c', 'd', 'e'}
twoFirst := letters[:2]
fmt.Printf("%c\n", twoFirst[0])
letters[0] = 'z'
fmt.Printf("%c\n", twoFirst[0])
```

The built-in `copy`, as its name suggests, **copies** a slice from source to destination, and returns the number of elements copied. Watch out for the order of the arguments. The `destination` comes first.

```
func copy(dst, src []T) int
```

The length of the destination should be equal or greater than the length of the source, otherwise only the smaller number of elements will be copied. Example:

```
a := []string{"apple", "banana", "orange", "grapefruit"}
b := make([]string, len(a))
howMany := copy(b, a)
fmt.Println(b)
fmt.Println(howMany)
tooShort := make([]string, 2)
notEnough := copy(tooShort, a)
fmt.Println(tooShort)
fmt.Println(howMany)
```

Will produce:

```
[apple banana orange grapefruit]
4
[apple banana]
2
```

4.2.1 Iterating

Remember the chapter on loops. Back then, we wrote a simple sum function.

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

The very same construct can be used to iterate over an array or a slice. At each iteration, instead of displaying the value of `i` (our index), we display the fruit at index `i`. We keep iterating while our index is strictly less than the number of fruits, so `i < len(fruits)`. The index will therefore vary from 0 to 3, and never reach 4, otherwise we would attempt to access the fruit at index 4, which would cause an error.

```
fruits := []string{"apple", "banana", "orange", "grapefruit"}
for i := 0; i < len(fruits); i++ {
    fmt.Println(fruits[i])
}
```

This loop construct is generic but somewhat verbose, as we need to explicitly declare an index, declare the starting point at 0, the end condition and the post-statement. Whenever possible, simpler forms of loops using `range` are preferred.

```go
fruits := []string{"apple", "banana", "orange", "grapefruit"}
for i := range fruits {
    fmt.Println(fruits[i])
}
```

Or the alternative form that allows you to retrieve both the index and the element at the same time:

```go
fruits := []string{"apple", "banana", "orange", "grapefruit"}
for i, fruit := range fruits {
    fmt.Printf("%d -> %s\n", i, fruit)
}
```

```
0 -> apple
1 -> banana
2 -> orange
3 -> grapefruit
```

What if you are not interested in the index `i`, but still want to retrieve the fruit? The index can be discarded with underscore `_`:

```go
fruits := []string{"apple", "banana", "orange", "grapefruit"}
for _, fruit := range fruits {
    fmt.Println(fruit)
}
```

The generic, more verbose construct has its place. If you need to skip some elements, or loop over them backward for instance, it can accomodate a wider range of scenarios. Let's display the list of fruits in reverse order:

```go
fruits := []string{"apple", "banana", "orange", "grapefruit"}
for i := len(fruits) - 1; i >= 0; i-- {
    fmt.Println(fruits[i])
}
```

Notice that we had to start at `len(fruits)- 1` with -1 because the length is 4, but the highest possible index is 3.

```
grapefruit
orange
banana
apple
```

4.2.2 Reference

Slice intro on golang blog

4.3 Filtering

A common operation on lists is to filter values from a slice that match a certain criteria. For instance, we have a list of scores, ranging from 0 to 20. We want to keep all scores equal or above 12. Putting together what we already know about loops, slices, and conditional statements, we can write the following program:

```
scores := []int{12, 14, 20, 3, 10, 16}
success := make([]int, 0)
for _, score := range scores {
    if score >= 12 {
        success = append(success, score)
    }
}
fmt.Println(success)
```

```
[12 14 20 16]
```

Back to the kitchen. Let's say we want to skip every other fruits, and store the result in another slice called `skimmed`. To skip fruits in our iteration, instead of incrementing with `i++`, we will increase `i` by `2`. For the sake of simplicity, we will do something inefficient by declaring `skimmed` with a length of `0` and relying only on **`append`** to expand the slice.

```
fruits := []string{"apple", "banana", "orange", "grapefruit"}
skimmed := make([]string, 0)
for i := 0; i < len(fruits); i = i + 2 {
    skimmed = append(skimmed, fruits[i])
}
fmt.Printf("%d fruits in %v\n", len(skimmed), skimmed)
```

A more efficient technique would be to declare `skimmed` with the proper length, that is, half of `fruits` length. On top of that, we would need a second index to remember where we stand in `skimmed`, and that index would be different than the index in `fruits`. We could also have leveraged the slice **capacity**, but we have left out that characterstic of slices in order to focus on algorithms. Hopefully our naive and simple approach does the trick.

```
2 fruits in [apple orange]
```

4.4 Min and max

Finding the maximum of slice is a form of filter, that yields a single element. In order to find the greatest element of a slice, we first initialize the `max` as the first element of the slice.

```
12   34   6   13
^^
max == 12
```

Then, we iterate the slice. We compare each element we encounter with `max`. If the element is greater, it becomes the new `max`. Otherwise, we keep the current max… by doing nothing.

```
func maxInt(values []int) int {
    max := values[0]
    for _, i := range values {
        if i > max {
            max = i
        }
    }
    return max
}
```

Let's call this function:

```
func main() {
    a := []int{12, 34, 6, 13}
    fmt.Println(maxInt(a))
}
```

The output is of course `34`. The same logic can be applied to find the smallest element.

4.5 Generics

Let's revisit our `sumOfSquare` function.

```go
package main

import "fmt"

func sumOfSquares(a, b int) int {
    return a*a + b*b
}

func main() {
        fmt.Println(sumOfSquares(3, 2))
}
```

We have seen that it won't work with a type other than a `int`. It makes sense not to square another type like `string`, but we could perfectly want to compute the square of to floating-point numbers.

Go allows to define a **generic** function[16].

First, we declare a type with constraints:

```go
type Number interface {
    int | float32 | float64
}
```

Then, we make `sumOfSquare` generic by allowing it to accept any `Number` symbolized by `T`, and return the same type `T`.

```go
func sumOfSquares[T Number](a, b T) T {
    return a*a + b*b
}
```

We can now call it with different numeric types.

```go
fmt.Println(sumOfSquares(1, 2))
fmt.Println(sumOfSquares(1.5, 3.2))
```

Remark: we cannot call our function with mixed types, for instance `sumOfSquare(2.5, 2)`. The following error occurs: `default type int of 2 does not match inferred type float64`

[16] https://go.dev/doc/tutorial/generics#add_generic_function

`for T`. The second argument must be turned into a `float64` to be compatible with the first one:

```go
sumOfSquares(2.5, float64(2))
```

4.6 Exercices

1. Write a function `minInt` that accepts an array of integeters as parameter, and returns the smallest element in the array.
2. Write a function `sumInt` that accepts an array of integeters as parameter, and returns the sum of all element in the array.
3. Write a function `posWord` that takes an array of strings as parameter, and returns the position of the word in the array if found, or `-1` otherwise.
4. Make the function `maxInt` generic to accept any type of number.

5 Complex data types

5.1 Maps

5.1.1 Definition

A **map** is a data type composed of a collection of **key-value pairs**. Each key can appear at most once.

With a map, you can for instance associate fruits with their energy value, or the quantity remaining in your kitchen. Conceptually, it looks like this:

```
fruit -> quantity
```

With some content:

```
apple -> 2
orange -> 3
pear -> 1
```

5.1.2 Operations

A map offers operations to add a new key-pair, to retrieve the value associated to a key, to check the existence of the key and therefore of the value, to delete a key-value pair, and to iterate through its elements.

In Go, the corresponding type is aptly named `Map`.

```
map[KeyType]ValueType
```

We can declare a variable `fruits` as a map in which keys are `string` (the name of the fruit) and the values are `int` (the quantity). It is up to the programmer to give meaning to the keys and values. Here, we intended to count fruits.

```
var fruits map[string]int
```

This map is not initialized. It will behave like an empty map if you attempt to read from it, but any attempt to modify it will cause a runtime panic. Instead, you can declare the same map and initialize it with the built-in `make`.

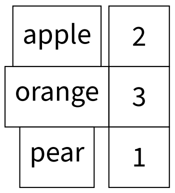
```
fruits = make(map[string]int)
// fruits == map[]
```

Now that the map is initialized, we can write to it.

```
fruits["apple"] = 2
// fruits == map[apple: 2]
```

Alternatively, the map can be initialized with a *map literal*.

```
fruits := map[string]int{
    "apple":  2,
    "orange": 3,
    "pear":   1,
}
// fruits == map[apple:2 orange:3 pear:1]
```



The *map literal* can also be empty, using the same syntaxm only with an empty literal `{}`. In this case, the empty map will be readable, just like an unitialized map, but it will also be writable (add, modify or delete entries).

```
fruits = map[string]int{}
```

A value can be retrieved and assigned to a variable. Supposing we set `apple` to `2`:

```
applesCount = fruits["apple"]
// applesCount == 2
```

You can obtain the number of elements (keys) in the map with the built-in **`len`**

```
typesOfFruits := len(fruits)
// typesOfFruits == 3
```

Adding a new key-pair does not require any built-in. Assign the value to the corresponding key like so:

```
fruits["banana"] = 5
```

There can not be any duplicate key. If you assign an existing key, its value will be replaced. Duplicate values are allowed. It makes sense: there could not be two different counts of *orange* for example. But two fruits can have the same count.

```
fruits["banana"] = 5
// map[apple:2 orange:6 pear:1, banana:5]
fruits["banana"] = 1
// "banana" value replaced, no new key
// map[apple:2 orange:6 pear:1, banana:1]
```

To remove an entry from the map, use `delete`. It does not return any value and is safe to invoke even if the value is not present in the map.

```
// fruits == {"apple": 2, "orange": 3, "pear": 1}
delete(fruits, "apple")
// fruits == {"orange": 3, "pear": 1}
delete(fruits, "foo")
// fruits == {"orange": 3, "pear": 1}
```

In a typical Go way, you can perform a two-values assignement that tests for the presence of the key, and retrieve it. Starting from our original map `orangeCount` will contain `3` and `present` will be `true`.

```
orangeCount, present = fruits["orange"]
// orangeCount == 3
// present == true
```

Using a similar construct, you can test a key for existence by discarding the first value with `_` (underscore).

```
_, present = fruits["orange"]
// present == true
```

5.1.3 Iteration

Iteration is achieved with the well-known `range`. Each iteration returns a key and a value, so in our case, a fruit and its count.

```
for fruit, count := range fruits {
    fmt.Println("Fruit:", fruit, "Count:", count)
}
```

```
Fruit: apple Count: 2
Fruit: orange Count: 3
Fruit: pear Count: 1
```

> *Note*: The order of iteration is not guaranteed.

5.1.4 Exercices

1. Declare an unitialized map holding name-age pairs. Attempt to read the key `john`, which is not present. What happens? Attempt to write the pair `steve`: `32`. Observe the behavior.
2. Declare a map with keys=country and value=capitals, using a map litteral with two entries: `Belgium`: `Brussels` and `Spain`: `Madrid`. Read the values of the following countries: `Spain`, `Italy`.
3. Using the same map, add the entry `Chile`: `Santiago`.
4. Iterate the map to display all countries and their capitals, one per line, in the following format: "The capital of Belgium is Brussels".
5. Delete a country of your choice.
6. Display how many countries you have in your map (should be 2).

5.2 struct

A **struct** is a typed collection of fields. It allows to group data together to form records.

5.2.1 Motivation

Suppose we want to represent a person. What data would we capture? What characteristics represent a person? It depends on the context. The medical department of a hospital may need data like first name, last name, date of birth, blood type, gender, weight and height and plenty of medical information. The administrative department of the same hospital is likely to be interested in data such as name, address (street, postal code, city, country), and social security number.

In our examples, we will limit ourselves to name and age.

So far, we have used “atomic” data types, like `string` and `int`. A `string` is suitable to store a person’s name. An `int` is suitable to store their age. It would however be unconvenient to use two independent variables to store those two characteristics.

```
name := "John"
age := 32
```

Arguably, this is not bad, but what if we want to represents two persons?

```
person1Name := "John"
person1Age := 32
person2Name := "Marge"
person2Age := 43
```

We only have two persons, each described by two fields, and things are already messy. Notice how these four variables are unrelated. Nothing expresses the fact that `person1Name` and `person1Age` are information related to the same person. Sure, they bear a common prefix `person1` that gives the reader some clue. But this alone does not create the representation of a person. It is just a good variable naming. We are left with a bunch of unrelated charactersitics. Imagine handling five persons, each with ten fields using this (lack of) technique. It would not be good programming design.

Instead, we can declare a **struct** that acts as a **blueprint** for a person – any person.

```
type Person struct {
    Name string
    Age  int
}
```

> *Note*: Using `uint8` instead of `int` will use up less memory. This is not a concern here. Also, in actual an information system, the date of birth will be used instead of the age.

We can instantiate our persons, and assign them to variables. Notice how the variables `john` and `marge` each conveniently represent a “whole” person, with all their relevant characteristic bundled in the object.

```
john := Person{"John", 32}
marge := Person{"Marge", 43}
```

5.2.2 Arguments

We can however be more explicit, by specifying each field name.

```
john := Person{Name: "John", Age: 32}
marge := Persom{Name: "Marge", Age: 43}
```

What is the difference between these two ways to declare a struct? The second one does not rely on the field position, which prevents mismatching fields. Sure enough, there is little risk of error when declaring our persons with our trivial struct. Indeed, the compiler would refuse inverted name and age, as they are of different types. But suppose we store the first name and the last name. In that case, which one comes first? You would have to look it up. If you accidentally invert them, the program will still compile. If someone decides to re-arrange the field order in the struct for any reason, the program would also still compile, but the names would be mixed up. Spelling out each field is more verbose but avoids mistakes and makes your program more robust if you are coding anything non trivial.

5.2.3 Constructor

This being said, it is idomatic to encaspulate new struct creation in a function. This function's name starts with `New` by convention, followed by the struct name. In our case, it would be named `NewPerson`. It is also idomatic to return a **pointer** to the newly created struct, denoted by a `*`. More on pointers later. Here is how the constructor would look like:

```
func NewPerson(name string, age int) *Person {
    return &Person{Name: name, Age: age}
}

robert := NewPerson("Robert", 67)
// robert.Name == "Robert"
// robert.Age == 67
```

Unfortunately, we loose the explict naming of arguments. On the other hand, this *constructor* can ensure consistency between fields and check for invalid input. For instance, it could check for negative age value, and reject it.

5.2.4 Behaviors

Functions can be "associated" with structs. For instance, let us say we want to add two functions: one to determine if a senior discount is applicable, and another to determine if a junior account is applicable. We could proceed with as follows:

```go
func IsJuniorDiscountApplicable(p *Person) bool {
    return p.age <= 12
}

fmt.Println(IsJuniorDiscountApplicable(robert))
// false
```

There is another way to declare this function, in a tighter manner, as a "behavior" for a person.

```go
func (p *Person) IsJuniorDiscountApplicable bool {
    return p.age <= 12
}
fmt.Println(robert.IsJuniorDiscountApplicable())
// false
```

Observe the difference. The second version may not seem to bring much to the table, if anything at all, but it will prove useful in the chapter about **interfaces**.

5.2.5 String representation

What if we try to print a person?

```go
john := &Person{"John", 34}
fmt.Println(john)
```

```
&{John 34}
```

The output is readable enough, but it can customized.

```go
func (p *Person) String() string {
    return fmt.Sprintf("%s (%d)", p.Name, p.Age)
}
```

Let's decompose this function.

- It is called `String`.

- It "acts" on a person[17] aliased by `p`, as per `(p *Person)`.
- It returns a **`string`**.
- We use `fmt.Sprintf` to build our custom representation. The first argument of `Sprintf` is the format. `"%s (%d)"` denoted a **`string`** as per `%s`, followed by a number as per `%d`. The number is between bracket.

`fmt.Println(person)` will look for a `String` function applicable to `person` and will find the one we have just defined. If we print `john`, the output is now formatted as we requested.

```
John (34)
```

Every person will be formatted that way when output is requested.

5.2.6 Variable naming

We already discussed the need to have meaningful variable names. Excessive abbreviations are hard to read. In a toy example, names like `p1` and `p2` may be acceptable, but we took the good habit of properly naming variables, e.g. `john` and `robert`.

Therefore, how come we wrote `(p *Person)` rather than `(person *Person)`? Isn't the second version less vague? The former is more compact and arguable, there is no risk of confusion, as the type is defined. The latter version introduces a form a stuttering. Furthermore, this `p` is only used "privately" by the function, as opposed to a possible argument that could be used by the caller. It does not "leak" outside.

It is customary in Go code to use the compact form, if not idiomatic. In any case, neither form should be considered "wrong".

> *Note*: Always be consistent in your coding style. Nothing is more annoying than unconsistent coding conventions. It slows down even seasoned programmers.

5.2.7 Multiple structs

A struct instance does not have to live alone. It can be combined with slices. Everything we have described in the Arrays and Slices chapter remains applicable to structs. Instead of declare slices of **`string`** or **`int`**, we can manipulate slices of `*Person` (a pointer to a person).

[17]More precisely, a **pointer** to a `Person`.

```go
robert := NewPerson("Robert", 67)
myla := NewPerson("Myla", 8)
persons := []*Person{robert, myla}
for _, person := range persons {
    fmt.Println(person)
}
```

```
Robert (67)
Myla (8)
```

5.2.8 Exercices

1. Declare a struct `Bike` to hold the following information: brand, model, color, number in stock.
2. Ensure than when a bike is printed, the color is between brackets and the number of items in stock is between square brackers. For instance `Cowboy Mk2 (black)[4]`.
3. Declare a slice `inventory` to contain an arbitrary number of bikes. Initiliaze it with the following products: Blue-Grey Trek Powerfly (3 in stock), Black-Orange BTwin Triban 540 (2 in stock).
4. Write a function `countInStock` that receives our inventory and returns the total count of bikes (in our example, 5 in stock).

5.3 Interface

An interface is a **named collection of method signatures** that a type can implement.

This definition is very abstract, so let us work out a canonical example: shapes. A triangle, a square, a rectangle, and a circle are shapes. They are geometrically different, but they share similarities. They all have an area and a perimeter, although each is calculated differently, based on the shape's characteristics.

Shape	Area	Perimeter	Terms
Triangle	$\frac{1}{2}bh$		b=base; h=height

Shape	Area	Perimeter	Terms
Square	a^2		a=side
Rectangle	lw		l=length; w=width
Circle	πr^2		r=radius

If you are familiar with other object-oriented languages, notice that interfaces in Go do not enforce a type to implement methods.

An interface `Shape` would allow us to treat triangles, squares, rectangles and circles uniformly, and to call their functions to compute area and perimeter without knowing the actual shape.

Let us declare structs for the different shapes. This is nothing different than what we saw earlier, when we used `Person` as an example. To limit the lines of codes, we shall limit ourselves to `Square` and `Rectangle`.

```
type Square struct {
    Side float64
}

type Rectangle struct {
    Width float64
    Height float64
}
```

We now declare an interface `Shape`. The keyword **`type`** is used. Notice the similarities and the differences between an **`interface`** and a **`struct`**.

```
type Shape interface {
    Area() float64
    Perimeter() float64
}
```

As we know, functions can be added to structs. Guess what. We are going to add `Area()` and `Perimeter()` functions to our two shapes. The actual computation depend on the shape, but the function declarations are the same, only applied to different types.

```
func (s *Square) Area() float64 {
    return s.Side * s.Side
}
```

```go
func (s *Square) Perimeter() float64 {
    return 4 * s.Side
}

func (r *Rectangle) Area() float64 {
    return r.Width * r.Height
}

func (r *Rectangle) Perimeter() float64 {
    return 2 * (r.Width + r.Height)
}
```

Do not mind the `*`. They denote *pointers*, e.g. we are manipulating *something that points to a square* rather than a square. We will come back to pointers later.

For the sake of readability, let us add the much needed `String` functions, otherwise our shapes will be represented as numbers, e.g. `&{5}` or `&{4. 8}`

```go
func (s *Square) String() string {
    return fmt.Sprintf("Square. s=%f", s.Side)
}

func (r *Rectangle) String() string {
    return fmt.Sprintf("Rectangle. w=%f, h=%f", r.Width, r.Height)
}
```

Now that `Square` and `Rectangle` implement the methods defined in `Shape`, they *are* shapes and can be treated uniformly as such ; for instance in a slice of `[]Shape`. Notice how, after the slice declaration, there are no references to `Square` or `Triangle`. There is no use of the original structs, only of the interface `Shape`. We have a achieved a form of **polymorphism**.

```go
square := &Square{Side: 5}
rectangle := &Rectangle{Width: 4, Height: 8}
shapes := []Shape{square, rectangle}
for _, shape := range shapes {
    fmt.Printf("%s => area=%f, perimeter=%f\n",
        shape, shape.Area(), shape.Perimeter())
}
```

```
Square. s=5.000000 => area=25.000000, perimeter=20.000000
Rectangle. w=4.000000, h=8.000000 => area=32.000000, perimeter=24.000000
```

> *Note*: Make sure you implement the **exact methods** from the interface. A simple typo and you would just be declaring an unrelated function, failing to adhere to the interface.

5.3.1 The empty interface

The interface type that specifies zero methods is known as the empty interface.

```
interface{}
```

An empty interface may hold values of any type, because every type implements “at least” zero methods. In other words, all types implement the empty interface *implicitly*.

Consider the following declaration[18]:

```
var i interface{}
fmt.Printf("(%v, %T)\n", i, i)
```

The output is `(<nil>, <nil>)`, meaning both value and type are `nil`. Let us assign something to `i`.

```
i = 42
fmt.Printf("(%v, %T)\n", i, i)
```

`(42, int)` will be printed, meaning `i` is an `int` and its value is `42`. One last for the road:

```
i = "hello"
fmt.Printf("(%v, %T)\n", i, i)
```

Prints `(hello, string)` and, as you figured out, `i` is now a `string` and its value is `"hello"`.

> *Note*: These code snippets are for demonstration purpose only. Their appearance in actual code would be dubious.

Empty interfaces are used by code that handles values of **unknown type**. For instance, the well-known `fmt.Println`[19] takes any number of arguments of type `interface{}`. All that time, you have been using it intuitively. Now you can understand the mechanism that lies behind.

[18] Example comes from A Tour of Go https://tour.golang.org/methods/14
[19] https://golang.org/pkg/fmt/#Println

5.3.2 Exercices

1. Complete the `Shape` example by adding `Triangle`. Implement the requires functions so that it becomes a shape. Make sure the output is readable when printed.
2. Declare an array containing the string `"hello"` and the integer `42`.

5.4 Sets

A set is an abstract data type that can store **unique** values, **without any particular order**.

Unlike many high level languages, Go does not have a data type for set. It has however another data type that can serve the same purpose: a map. Each key can appear at most once in a map, so the first property of a set (unique keys) can be met. There is no order either so the second property is met as well.

The trick is to ignore the values in the map.

The idiomatic way (or one of the idiomatic ways) to implement a set in Go is the following, assuming a set of strings:

```
set := make(map[string]bool)
```

Suppose we want to store the folling values in a set: $orange, apple, raspberry$. Each fruit would be a key, with `true` as the associated value.

```
fruits := map[string]bool{
    "apple":  true,
    "orange": true,
    "pear":   true,
}
```

All the concepts we have seen about map (add, delete, iterate) apply here.

The boolean value is a dummy value, serving no purpose other than to "make the map work". It is slightly inefficient (a bit of memory is used to hold a fake value) and arguably inelegant, so you may run into other implementations using a `struct{}` as the value.

```
type void struct{}
var member void
fruits := map[string]void{
    "apple":  void,
    "orange": void,
    "pear":   void,
}
```

The first version is more straighforward if you need to declare a set "on the spot". In practice, and in this chapter, we will prefer a dedicated structure that **encapsulate** (hide the underlying details) the data type *set* and its operations.

So instead of manipulating a map acting as a set, we will be manipulating a new data type Set. We will know that underneath, there is a map, but it won't appear in our interactions with the set *from the outside*.

First, let us create a set of strings.

```go
package myset

type StringSet struct {
    set map[string]bool
}
```

The `set` lower case is not accessible from outside the package `myset`. This is as desired, because want to deal with a `StringSet` rather than with the underlying map. We need in return to implement some basic functionalities, otherwise there would be no way to interact with the set. Let's start with a constructor.

```go
func NewStringSet() StringSet {
    return &StringSet{make(map[string]bool)}
}
```

`NewStringSet` builds a new `StringSet` structure by means of `StringSet{...}`. The unique parameter `make(map[string]bool)` creates the underlying map called `set` (because functionally it acts as a set, even if it is technically a map) in the `StringSet` structure.

If this does not make sense, consider a more verbose version:

```go
func NewStringSet() StringSet {
    // new structure, empty shell
    newStringSet := StringSet{}
    // actual set (yes, it's a map, don't tell anyone)
    actualSet := make(map[string]bool)
    // assign it to `set` inside the struct
    newStringSet.set = actualSet
    // return the new structure, set included
    return newStringSet
}
```

This second version is equivalent to the first one, but a trained programmer would prefer the first one. Try to figure out how they match.

We can create a `StringSet` with `set := NewStringSet()` but we must implement a few more operations before we can do something usefull with it.

Operation	Description
Add	Add an element to the set
Remove	Remove an element from the set
Contains	Check if the set contains a value

We already know how to add or delete an element from a map.

```
func (ss StringSet) Add(value string)  {
    ss.set[value] = true
}

func (ss StringSet) Remove(value string) {
    delete(ss.set, value)
}

func (ss StringSet) Contains(value string) bool {
    _, found := ss.set[value]
    return found
}
```

All these functions are self-explanatory. After all, our structure is merely a map in disguise. Notice the repeating `(ss StringSet)`. It means the function is applicable to a `StringSet` using a dot, like so:

```
fruits := NewStringSet()
fruits.Add("blackberry")
fruits.Contains("orange")  // false
fruits.Remove("blackberry")
```

> Tip: in real life, you should consider using a package implementing a set, and avoid re-inventing the whell. This advise is applicable to any data type or structure.

Finally, let us add two usual operations.

Operation	Description
Intersection	Return a new set containing the elements common to this set and another set
Union	Return a new set containing the elements of this set and another set

```go
func (ss StringSet) Union(otherSet StringSet) StringSet {
    union := NewStringSet()
    for k, _ := range ss.set {
        union.set[k] = true
    }
    for k, _ := range otherSet.set {
        union.set[k] = true
    }
    return union
}
```

We can say that `Union` is a function on a `StringSet` that accepts one argument of type `StringSet` and returns a `StringSet`. There are 3 sets at play: the "target" `ss`, the `otherSet` and the returned set. The original sets `ss` and `otherSet` are left unchanged. In action:

```go
    vegetables := NewStringSet()
    vegetables.Add("eggplant")
    vegetables.Add("salad")
    union := fruits.Union(vegetables)
```

For `Intersection`, we will start from the target set. For each key in `ss`, we search for the same key in `otherSet`.

```go
func (ss StringSet) Intersection(otherSet StringSet) StringSet {
    intersection := NewStringSet()
    for k, _ := range ss.set {
        if _, found := otherSet.set[k]; found {
            intersection.set[k] = true
        }
    }
    return intersection
}
```

We have used an idiomatic construct in the `if` statement: we assign and check `found` on the same line. It saves one line and express the intent that we want to check if the key is present and discard `found` afterwards.

To illustrate the function, I know of one fruit that can be considered as a vegetable.

```
    fruits.Add("tomato")
    vegetables.Add("tomato")
    interestion := fruits.Intersection(vegetables)  // tomato
```

> Exercise: implement the following operations.

Operation	Description
Pick	Return an arbitrary element (any element) from this set
Pop	Return an arbitrary element (any element) from this set, deleting it from this set
Count	Count the number of elements in the set
Difference	Return the difference of this set and another set
IsSubset	Test whether this set is a subset of another set

5.5 Stack

A stack is an abstract data type that serves as a collection of elements[20], with 2 main operations:

- **push** adds an element to the collection
- **pop** removes the most recently added element from the collection

And optional operations:

[20] https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

- **len** returns the number of elements in the collection
- **peek** returns the most recently added element (without removing it)

It is a **LIFO** (last in, first out) data structure.

Stacks are used for implementing function calls, storing program variables, parsing expressions, evaluating arithmetic expressions, …

Go includes a package that implements a stack[21]. It uses an optimized `type` that contains a node, chained with pointers. It maintains a `length` to avoid navigating the chain when `Len()` is called, for performance reasons.

5.5.1 A simple implementation

As an exercise, we can implement a simpler stack. We will use:

- a slice to store the values in the collection
- a generic type `T` (rather than the legacy `interface{}` in the Go library)
- and all functions will operate on a `*Stack[T]`.

This translates to:

```go
package stack

type Stack[T any] []T

func New[T any]() *Stack[T] {
    return &Stack[T]{}
}

func (s *Stack[T]) Len() int {
    return len(*s)
}

func (s *Stack[T]) Push(elem T) {
    *s = append(*s, elem)
}

func (s *Stack[T]) Pop() (elem T, ok bool) {
    if s.Len() == 0 {
```

[21] https://pkg.go.dev/github.com/golang-collections/collections/stack

```go
        return elem, false
    }
    index := len(*s) - 1
    elem = (*s)[index]
    *s = (*s)[:index]
    return elem, true
}
```

Remarks:

- We rely on `New()` to create a `*Stack` rather than forcing the client to declare a slice.
- `Pop()` returns the element (if found) and a flag. This is a common and convenient way to return an possibly missing value in Go.

Let's add a main package to demonstrate the stack capabilities, and how to pop a value.

```go
package main

import (
    "fmt"
    "be.sugoi.ipgo/stack"
)

func main() {
    s := stack.New[int]()
    s.Push(1)
    s.Push(2)
    s.Push(3)
    for s.Len() > 0 {
        elem, ok := s.Pop()
        if !ok {
            panic("stack is empty")
        }
        fmt.Println(elem)
    }
    fmt.Println(s.Pop()) // 0 false
}
```

Refer to the appendix "Modules" for instructions on how to store those files and execute the program.

5.5.2 Exercise

1. Make this code run on your computer.
2. Add a `Peek()` function. Test that it does not remove the element it returns.

6 Go techniques

In this chapter, we introduce techniques that are not shared by all imperative programming languages, like pointers, or that are the hallmarks of Go, like channels.

6.1 Pointers

A variable holds a value. A **pointer** holds the memory address of a value.

6.1.1 Declaration

```go
// Declare an integer variable with value 10
val := 10

// Declare a pointer to the integer variable
ptr := &val

// Print the value of the integer variable by dereferencing pointer
fmt.Println(*ptr) // Output: 10
```

The following syntax is more compact. It also shows `ptr` is not a mere **`int`** and that it requires the use of **`new`**.

```go
ptr := new(int)
*ptr = 10
```

10
val
10
ptr

Figure 5: Illustration of a variable (left) and a pointer (right)

A pointer is represented by an asterisk `*` followed by the variable name. `*ptr` is a *pointer to an int*.

The asterisk is also used to **dereference** the pointer. Dereferencing a pointer gives you access to the value the pointer points to.

6.1.2 Usage

Pointers in Go are useful for:

- **Modifying values in-place**. When you pass a value to a function, it is passed by value, which means that any changes made to the value within the function will not affect the original value. However, if you pass a *pointer* to the value, you can modify the value in-place, which means that the changes made within the function will also be reflected in the original value.
- **Data structures**. Pointers are essential for implementing many data structures such as linked lists, trees, and graphs. These data structures require dynamic allocation of memory and pointers allow you to allocate and manage memory dynamically.
- **Performance**. When a large data structure such as a struct is passed to a function, it is copied to a new memory location. This copy can be expensive in terms of time and memory. By passing a pointer to the original data structure, you can avoid copying the entire data structure.

6.1.3 Modify in-place

Consider the following function:

```go
func AddOne(x int) {
  x = x + 1
}

func main() {
  x := 10
  AddOne(x)
  fmt.Println(x) // x is still 10
}
```

The function `AddOne` will not modify `x` because it is passed **by value**. Of course, we don't need no fancy pointer to add 1. We don't even need a function. That is just an example. Let's modify our function to actually add 1 to `x`.

```go
func AddOne(x *int) {
  *x = *x + 1
}
```

```go
func main() {
  x := 10
  AddOne(&x)
  fmt.Println(x) // x is 11
}
```

By using a pointer (`*int`) the `AddOne` function is able to modify the original `x`.

The `&` operator to obtain the **address** of a variable. `&x` returns a `*int` (pointer to an int) because `x` is an `int`. This is what allows us to modify the original variable.

6.1.4 Data structures

We have already seen an example of pointers with the `struct Person`. Quick refresher:

```go
type Person struct {
    Name string
    Age  int
}

func NewPerson(name string, age int) *Person {
    return &Person{Name: name, Age: age}
}

robert := NewPerson("Robert", 67)
// robert.Name == "Robert"
// robert.Age == 67

func (p *Person) IsJuniorDiscountApplicable bool {
    return p.age <= 12
}
// robert.IsJuniorDiscountApplicable()) == false
```

6.1.5 Performance

Even if a function won't modify your structure, it is still beneficial to pass a pointer for performance reason when the struct contains a lot of data.

6.1.6 Safety

Go provides a higher level of safety than C with regard to pointers. This is because Go has built-in garbage collection, which makes it easier to manage memory allocation and deallocation. Go also has a strong type system that helps prevent common pointer-related errors in C, such as dereferencing null pointers or accessing memory that has already been freed.

Go also provides some additional safety features like bounds checking for slices and arrays.

Unlike C, Go has no pointer arithmetic.

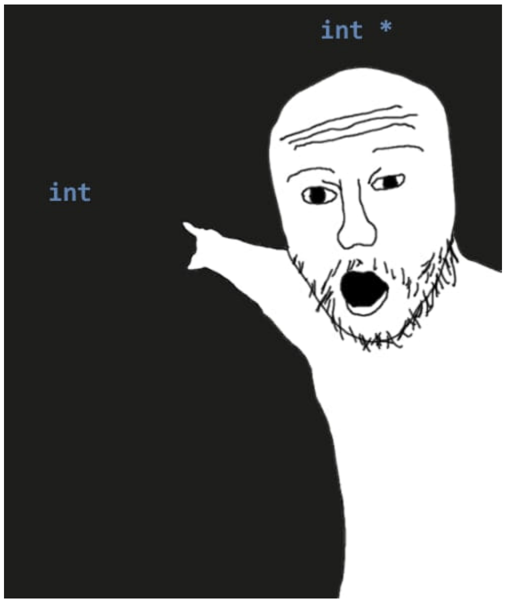


Figure 6: A meme on C pointers

6.2 Concurrency (goroutines)

Concurrency is property of a program, when two or more tasks can start, run, and complete in overlapping time periods.

For instance, a web crawler that scans pages on the internet can be made of a scanner that downloads pages, and a parser that interprets them. They may run at the same time, but it is not mandatory to achieve concurrency. The parser could run out of work if the scanner is stuck because of a broken internet connection.

Go has outstanding built-in support for concurrency via **goroutines**.

A goroutine is a function that can run concurrently with other functions. It is a plain old function invoked with the `go` keyword.

```go
package main

import "fmt"

func foo() {
    for i := 0; i < 100; i++ {
        fmt.Println(i)
    }
}

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

If you run this program, the result may not be what you expect. The program displays "Terminated" and… terminates. It does not print number from 0 to 99. This is because the goroutine had no chance to make any progress before the end of the program.

Let's modify this program so that waits half a second after printing the end message, but before it terminates.

```go
import (
    "fmt"
    "time"
)

func main() {
    go foo()
    fmt.Println("Terminated")
    time.Sleep(time.Millisecond * 500)
}
```

This time, the numbers are printed… after the message. We gave time to the goroutine to complete, but there is **no guarantee on the execution order**.

Besides, adding `time.Sleep` is a bad practice (how do you even decide on the "right" sleep duration? How much time does it need to display the number on different machines?). We will need another approach in the next section.

6.3 Channels

Channels are the Go way for goroutines to communicate with one another, and to synchronize their work.

As Rob Pike would put it: "don't let computations communicate by sharing memory, let them share memory by communicating".

The communication can be uni- or bi-directional. Assuming `T` is the data type of the data passed through the channel:

- `chan T` is a bidirectional channel
- `chan<- T` is a unidirectional channel, send-only
- `<-chan T` is a unidirectional channel, receive-only

6.3.1 Fake web crawler

Let's implement a fake version of the web crawler we mentionned earlier. We first make a channel called `pages`. It will be used by our two workers. We shall separate fetching from parsing a page. The program terminates when the user presses Enter. `fetch` and `parse` are invoked as goroutine with `go`.

We pass the channel as a parameter. Notice the slight difference in declaration:

- `chan string`, bidirectional channel in `main`
- `chan<- string`, unidirectional channel, send-only in `fetch`
- `<-chan string`, unidirectional channel, receive-only in `parse`

The bidirectional channel declared in `main` can be "specialized" into a send-only or receive-only channel when passed as an argument. This cannot be done the other way around; you cannot turn you unidirectional channel into a bidirectional channel.

```go
package main

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

```go
func main() {
    urls := []string{
        "https://www.google.com",
        "https://www.yahoo.com",
        "https://www.bing.com",
    }
    pages := make(chan string)

    go fetch(urls, pages)
    go parse(pages)

    fmt.Println("Press ENTER to exit")
    var input string
    fmt.Scanln(&input)
}

func fetch(urls []string, pages chan<- string) {
    for _, url := range urls {
        // Pretend to fetch the page
        fmt.Printf("Fetching %s\n", url)
        pages <- url
    }
}

func parse(pages <-chan string) {
    for {
        // Pretend to parse the page
        url := <-pages
        fmt.Printf("Parsing %s\n", url)
        sleepTime := time.Duration(rand.Intn(2000)) * time.Millisecond
        time.Sleep(sleepTime)
    }
}
```

6.4 Producer-consumer

We can generlize the webcrawler to a pattern called **producer-consumer**.

The main function waits for a timeout. A channel comes in handy. Instead of sleeping, it uses the `select` construct to match a particular event.

```go
package main

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

func producer(items chan<- int) {
    i := 0
    for {
        // Pretend to produce an item
        sleepTime := time.Duration(rand.Intn(2000)) * time.Millisecond
        time.Sleep(sleepTime)

        // Send the item on the channel
        fmt.Println("Produced", i)
        items <- i

        i++
    }
}

func consumer(items <-chan int) {
    for item := range items {
        // Pretend to consume the item
        sleepTime := time.Duration(rand.Intn(2000)) * time.Millisecond
        time.Sleep(sleepTime)
        fmt.Println("Consumed", item)
    }
}

func main() {
    items := make(chan int)

    // Launch the producer and consumer goroutines
    go producer(items)
    go consumer(items)

    // Wait for 5 seconds
    select {
    case <-time.After(5 * time.Second):
        fmt.Println("Timeout reached, exit")
    }
}
```

There are many ways to organize your program when it comes to channels, and Go offers many mechanisms beyond the scope of this chapter [22].

[22] See https://go.dev/tour/concurrency/2 https://go101.org/article/channel.html

7 Programming techniques

7.1 Recursion

Recursion is the technique of making a function call itself. It is a way to break down a problem into smaller and easier problems, until the problem becomes trivial. A function (or a procedure) that goes through recursion is said to be *recursive*.

The canonical use case for teaching recursion is the calculation of factorial.

$$n! = \prod_{i}^{n} i = 1 \times 2 \times 3 \times ... \times (n-1) \times n \quad n \in \mathbb{N}$$

By convention, factorial of 0 is 1 ; $0! = 1$. For example:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

The factorial of a number can be calculated with an **iterative** version and a loop construct you already know.

```go
func factorial(n uint) uint {
    result := 1
    for i := 1; i <= n; i++ {
        result *= i
    }
    return result
}
```

Instead, we can use recursion to solve this problem. Let's see how it works with a methaphore. Suppose you are asked to compute $fact(3)$. But that is too much work. You can multiply 2 numbers. You also know Alice can calculate $fact(2)$. So you ask her to give you the value of $fact(2)$, and you multiply the result by 3.

```
3! = 3 * 2!  // ask Alice for 2!
```

How does Alice calculate $fact(2)$? She too can multiply 2 numbers. Alice is lazy. She can't bothered to put too much thoughts into $fact(1)$. She calls Bob. He knows how to calculate $fact(1)$. She will simply multiply $2 \times fact(1)$.

```
2! = 2 * 1!  // Alice asks Bob for 1!
```

So far we have:

```
3! = 3 * (2 * 1!)
```

Bob does not need to call anyone. He knows $fact(1) = 1$ and that's it. He deals with the **terminal condition** or **base case**, upon which the solution stops recurring. We end up with:

```
3 * 2!
3 * (2 * 1!)
3 * (2 * (1 * 1))
3 * (2 * 1)
3 * 2
6
```

In a computer program, you, Alice and Bob roles will be played by a single function. A recursive function. More formally, the recursive definition of factorial is:

```
0! = 1.
n > 0, n! = (n - 1)! × n.
```

The definition can be translated to Go:

```
func factorial(n uint) uint {
    if n == 0 {
        return 1
    }
    return n * factorial(n-1)
}
```

The general form of recursion is:

$$f(x) = e_0 \text{ if } x \in D_0$$

$$f(x) = F(f(e_1), f(e_2), ..., f(e_k)) \text{ if } x \in D_v$$

where:

- f is the name of the function
- $e_i (i = 0..k)$ are expressions that depend solely on x
- $F(y_1, ..., y_k)$ is an expression that depends solely on x and $(i = 1..k)$

- D_0 and D_v are disjoint sets

There is a cost to this approach. When we decompose the chain of calls made by Alice and friends, we realise how each recursive call cannot be evaluated before the whole chain completes. Each call maintains a record on the program stack.

$$f(5) \rightarrow f(4) \rightarrow f(3) \rightarrow f(2) \rightarrow f(1) \rightarrow f(0)$$

The stack space needed for the recursive calls is proportional to the number. We say the space complexity is $\mathcal{O}(n)$. Compared to the iterative version, for which the space complexity is constant in $\mathcal{O}(1)$. For small numbers, keeping calls on the stack has limited adverse effect. For larger numbers however, the stack may **overflow**.

A stack overflow can be avoided by using **tail recursion**, in which the recursive call is the *last call*. We use an `accumulator` to carry on the intermediate result. An **auxiliary** function is used to perform the actual recursion. The auxiliary function is not meant to be called directly.

```go
func factorial_rec(n, accumulator uint) uint {
    if n == 0 {
        return accumulator
    } else {
        return factorial_rec(n-1, n*accumulator)
    }
}

func factorial(n uint) uint {
    return factorial_rec(n, 1)
}
```

Unfortunately, like many imperative languages, Go does not optimize tail recursion, unlike most functional languages (Haskell, Scheme etc). But we wanted to demontrate the technique nonetheless.

Arguablty, this last version is less readable than the initial, iterative version. Although factorial is a traditional example to explain recursion, it may not be the preferred approach in practive. However recursion is a key technique when solving classic problems like Fibonacci numbers or the Tower of Hanoi. Some problems are way easier to solve with recursion.

8 Classic computer problems

8.1 Fibonacci numbers

The Fibonacci numbers (F_n) form a sequence called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1.

$$n > 1 : F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

The sequence starts with

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144$$

8.1.1 Recursive method

It is possible to translate the mathematical definition into Go code without much effort.

```go
func Fib(n int) int {
    if n <= 1 {
        return n
    }
    return Fib(n - 1) + Fib(n - 2)
}
```

This implementation is trivial, albeit very inefficient. Let's visualise the successive recursive calls for $Fib(4)$.

$$\begin{tikzcd}
 & & & Fib(4) \arrow[dl] \arrow[dr] & & \\
 & & Fib(3) \arrow[dl] \arrow[d] & & Fib(2) \arrow[d] \arrow[dr] & \\
 & Fib(2) \arrow[dl] \arrow[d] & Fib(1) & & Fib(1) & Fib(0) \\
Fib(1) & Fib(0) & & & &
\end{tikzcd}$$

As you can see, even for a simple case where n is low, there is a lot of repetition. The same $Fib(n)$ is evaluated several times. It only gets worse as n becomes bigger. A program cannot "remember" the previously calculated values if it has not been written to do so, and will therefore performs the recursive calls as many time as needed.

> Exercise: draw the top of the recursive calls tree for $Fib(5)$.

8.1.2 Memoization

Memoization is an optimization technique used primarily to speed up programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again[23].

Our cache will be of type `map[int]int`, where each key is the sequence, and each value the corresponding Fibonacci number. So as $Fib(6) = 8$, the cache will be populated with `{6: 8}` *the first time* this number is calculated.

We start with trivial cases `{0: 0, 1: 1}` to populate our cache.

We want to keep the function signature tidy, that is we do not want any concept of memoization to appear to the calling program. `FibMemoize(int)int` will simply initilize the cache, and call the auxiliary function `fibMemoizeRecurse`, which does the actual recursive work, passing our cache along.

When we enter `fibMemoizeRecurse`, we first check if we have already computed the number by accessing our cache. In case of a hit, we return the cached value directly. Otherwise, we perform the recursive calls for $F(n-1)$ and $F(n-2)$, passing the cache along.

Before we return the computed number, we store it in the cache, so other callers can benefit from it.

```go
func FibMemoize(n int) int {
    cache := map[int]int{0: 0, 1: 1}
    return fibMemoizeRecurse(n, cache)
}

func fibMemoizeRecurse(n int, cache map[int]int) int {
    cached, hit := cache[n]
    if hit {
```

[23]https://en.wikipedia.org/wiki/Memoization

```
        return cached
    }
    n1 := fibMemoizeRecurse(n-1, cache)
    n2 := fibMemoizeRecurse(n-2, cache)
    f := n1 + n2
    cache[n] = f
    return f
}
```

The recursive call tree is greatly simplified. The cache hits are denoted by a $*$. It might not look as much on a small tree like $Fib(4)$ as used here to illustrate, but beyond that number, entire branches of the tree are pruned.

$$\begin{tikzcd}
 & & & Fib(4) \arrow[dl] \arrow[dr] & \\
 & & Fib(3) \arrow[dl] \arrow[d] & & Fib(2)* \\
 & Fib(2) \arrow[dl] \arrow[d] & Fib(1)* & & \\
Fib(1) & Fib(0) & & &
\end{tikzcd}$$

> Exercise: extend the recursive calls tree above for $Fib(5)$ with memoization.

8.1.3 Iterative method

With the iterative approach, we loop over integers if n is greater than 2. We retain only the two previous numbers $F(n-1)$ and $F(n-2)$ as we progress.

Notice the use of an interesting Go property: `a, b = b, a` will swap the values of the variables `a` and `b`. This is *not* equivalent to `a = b; b = a` as we would loose the value of `a` after the first assignment.

```
func Fibo(n int) int {
    if n <= 1 {
        return n
    }
    n1 := 1
    n2 := 0
    for i := 2; i <= n; i++ {
        n2, n1 = n1, n1 + n2
    }

    return n1
}
```

There is no growing space complexity as n increases. This solution is arguably less readable than the naive recursive version.

8.1.4 Related

Fibonacci numbers are used in the Fibonacci search technique, and can form a tiling.

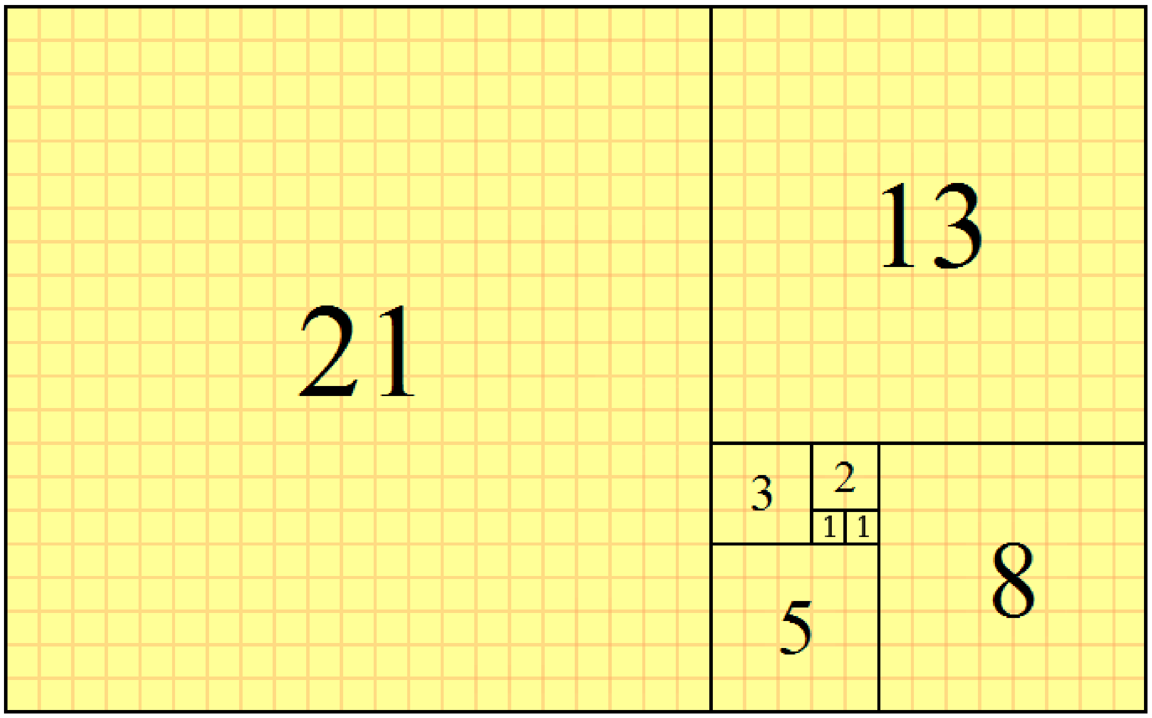


Figure 7: Tiling pattern using Fibonacci numbers

8.2 Hangman

Guess a secret word by suggesting letters.

8.2.1 The game

In this classic game, one person picks a random word, and another attempts to guess it. Each letter of the word to guess is hidden and replaced by a dash. At each turn, the guessing player suggests a letter. If it occurs in the word, the other player writes it at the correct position. If the letter does not occur in the word, the other player draws one limb of a hanged man stick figure on a gallow. The guessing player wins if they find out out the word before the hangman is drawn. The other player wins otherwise.

There are many variations of the game ; we will stick to a simple one. In our implementation:

- The computer will pick the word, and the human player will try to guess it.
- We play with uppercase letters only.
- If a letter occuring several times is guessed, it will be revealed at all the places it occurs.
- The program terminates after the word is guessed, or after the stickman is complete.

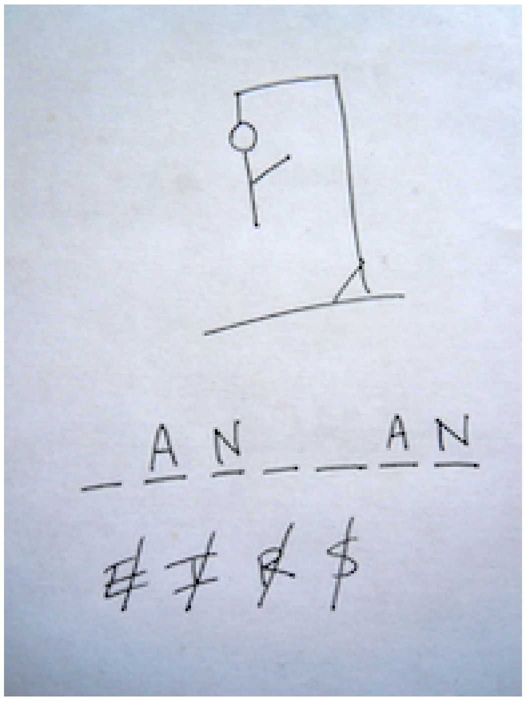


Figure 8: Hangman

> *Note*: Constructing a hanged stickman limb by limb is somewhat unsettling.

8.2.2 Building blocks

The game is straightforward to implement with the basic programming elements we have described so far. There is no complex algorithm or data structure involved. Our first step is to decompose the game in building blocks we can eventually assemble:

- Pick a random word from a dictionary
- Ask player for their guess
- Represent the game's state
- Start the game
- Process the guess
- Draw the game state
- Check if game is over

Even though the entire task may be daunting, each individual function is simple enough. By taking a divide and conquer approach, a seemingly difficult task becomes simpler.

8.2.3 Pick a random word

We will pick a random word from a dictionary stored as a file, *one word per line*. We will read and store the dictionary in memory in order to choose a random entry. This simple approach is not memory-efficient, but even thousands of words should not cause performance concerns on a modern machine.

> *Note*: There is no such thing as a `len(file)` function that would return the number of lines in the file without reading it. This is not a Go specific limitation. From the operating system point of view, a file is made of bytes and there is no concept of "lines" or "words". The linebreak is just a byte (or two bytes), like any other. We cannot know how many "lines" the file contains without reading it entirely.

A different approach would be to first count the number of words in the file (by scanning the file entierely), generate a random number smaller than the number of lines, and then rescan the file to choose the corresponding word. It would avoid loading the whole dictionary in memory, at the cost of two file scans instead of one. So the efficiency gained on memory would be mitigated by the double file scan (worst case, if we pick the last word in the file).

A more effcient but also fairly involved technique would improve the first scanning phase by reading large chuncks of bytes, rather than relying on the slower `scanner.Scan()` and `scanner.Text()` to read lines that will will use.

These improvements are however beyond the scope of our little program.

Here is a simple implementation to read a dictionary.

```go
const dictionaryPath = "dict.txt"

// Choose a random word from a dictionary
func randomWord(path string) string {
    file, err := os.Open(path)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
    lines := make([]string, 0)
    scanner := bufio.NewScanner(file)
    for scanner.Scan() {
        lines = append(lines, scanner.Text())
    }

    word := lines[rand.Intn(len(lines))]
    return strings.ToUpper(word)
}
```

8.2.4 Ask player's guess

We leniently read from the standard input, ignoring errors. Don't make a habit of ignoring errors. It is fine in a toy program, but not adequate for a production system. Incorrect input tolerance and error handling can take a fair amount of work, so we will that aside.

We then check that the input length is `1`, that it is a letter, and convert it to upper case.

```go
// Ask the player to enter their guess.
// Single letter only. Will be upper cased
func askGuess() string {
    for {
        fmt.Print("Your guess? ")
        var guess string
        _, _ = fmt.Scan(&guess)
        if len(guess) != 1 {
            continue
        }
        for _, c := range guess {
            if !unicode.IsLetter(c) {
                continue
            }
        }
        return strings.ToUpper(guess)
    }
}
```

Notice the endless `for` loop. In structured programming, one would declare a stop condition on the iteration. Arguably the extra variable and nesting in conditional statements would not improve readability. Idiomatic Go favors explicit exits via `break` (or in this case `continue`). The function `askGuess` is short, so there is no risk of confusion when reading the code flow.

8.2.5 Game state

A `struct` holds the game state. It is better than having free variables “floating” around.

```go
type Hangman struct {
    secret  string      // Secret word to guess
    word    string      // Current player's word.
                        // Letters not found yet will be '-'
    guesses []string    // Player's guesses
    limb    Limb        // Current limb
}
```

The `Limb` will be an enumeration. In idomatic Go, enums are composed of:

1. A custom type declaration
2. Several constants with `iota`
3. A string representation (for humans)

```go
type Limb int

const (
    Empty = iota
    Head
    Torso
    LeftArm
    RightArm
    LeftLeg
    RightLeg
)

func (limb Limb) String() string {
    return [...]string{
        "Empty", "Head", "Torso", "Left Arm",
        "Right Arm", "Left Leg", "Right Leg"}[limb]
}
```

We have included a fake limb called `Empty` to treat the starting point of the game like the rest of the game.

8.2.6 Start the game

When starting the game, we instantiate a consistent `Hangman` by performing the following actions:

- Pick a random `secret` word, using the function we declared above
- Initialize the `guesses` to an empty slice
- Set the `word` to dashes
- Set the current `limb` to `Empty`

```go
// Instantiate a new game
func newHangman() *Hangman {
    return &Hangman{
        guesses: make([]string, 0),
        secret:  randomWord(dictionaryPath),
        word:    strings.Repeat("-", len(secret)),
        limb:    Empty}
}
```

The pseudo-random generator must also be intitialized. It has to be done once in the program, no matter how many games are played. Initiliazation has been therefore kept out of `newHangman`
.

```go
func initGame() {
    rand.Seed(time.Now().UnixNano())
}
```

8.2.7 Process player's guess

The bulk of the game's logic is to process the player's guess.

If the `letter` occurs in the `secret`, a loop will replace the dash(es) to reveal it. Notice how we leverage *slicing* to construct a new `word` in 3 parts: 1. everything before the letter's position `[:i]`, 2. the `letter` itself and 3. what is left after the letter's position `[i+1:]`.

If the `letter` does not occur, we simply increase to the next `limb`.

At this stage, no check is performed to find out if one of the players won, or if the game continues.

```go
// Attempt to guess a letter.
// If the letter exists in the secret, it is revealed in the word.
// Otherwise, limb is incremented.
func (h *Hangman) guess(letter string) {
    h.guesses = append(h.guesses, letter)
    found := false
    for i, c := range h.secret {
        s := fmt.Sprintf("%c", c)
        if s == letter {
            h.word = h.word[:i] + letter + h.word[i+1:] // reveal
            found = true
        }
    }

    if !found {
        h.limb++
    }
}
```

8.2.8 Display the game state

You are free to go as fancy as you want for the display. We will opt for the utmost sobriety.

```go
func (h *Hangman) draw() {
    fmt.Printf("%s %v %s\n", h.word, h.guesses, h.limb)
}
```

Admitedly, calling this function `draw()` may be an overstatement since it merely displays or prints the game state in a raw form.

```
----E----E [A E P] Torso
```

> *Note:* Displaying a gallow and an actual stickman will require more lines of code of their own than the complete implementation presented here. There is nothing wrong with that ; we simply prefer to keep things simple.

8.2.9 Check game over

The guessing player wins if the `word` matches the `secret`. They loose if the `limb` is the last one. Give them the courtesy to reveal the secret in that case. If neither `won()` or `lost()` are **`true`**, the game goes on!

```
// Check if player won
func (h *Hangman) won() bool {
    return h.word == h.secret
}

// Check if player lost
func (h *Hangman) lost() bool {
    return h.limb == RightLeg
}

// Display game result
func (h *Hangman) displayResult() {
    if h.won() {
        fmt.Println("You won")
    } else if h.lost() {
        fmt.Printf("You lost. Secret word was: %s\n", h.secret)
    }
}
```

8.2.10 Main

Finally we assemble our building blocks in neat `main` function.

```
func main() {
    initGame()
    hangman := newHangman()
    hangman.draw()

    for !hangman.won() && !hangman.lost() {
        letter := askGuess()
        hangman.guess(letter)
        hangman.draw()
    }

    hangman.displayResult()
}
```

8.2.11 Sample

```
---------- [] -
```

```
Your guess? a
---------- [A] Head
Your guess? e
----E----E [A E] Head
Your guess? p
----E----E [A E P] Torso
Your guess? d
----ED---E [A E P D] Torso
Your guess? i
I---EDI--E [A E P D I] Torso
Your guess? n
IN--EDI--E [A E P D I N] Torso
Your guess? c
INC-EDI--E [A E P D I N C] Torso
Your guess? r
INCREDI--E [A E P D I N C R] Torso
Your guess? b
INCREDIB-E [A E P D I N C R B] Torso
Your guess? l
INCREDIBLE [A E P D I N C R B L] Torso
You won
```

8.2.12 Dictionary

You can easily find and download dictionaries from the internet, in English or in any other language. A short dictionary is however useful for testing purpose, like this one.

```
panda
cactus
dog
cat
incredible
```

8.3 Eight Queens

Place eight queens on a chessboard without any threat.

8.3.1 Statement

The eight queens puzzle requires to place 8 queens on a 8 by 8 chessboard so that no queen threatnens another. In other words no two queens can be found on the same row, column (called *file* in chess) or diagonal.

The puzzle admits 92 solutions, of which 12 *fundamental* that differ other than by symetry, reflection or rotation.

The problem can be extended to n queens on a $n \times n$ chessboard.

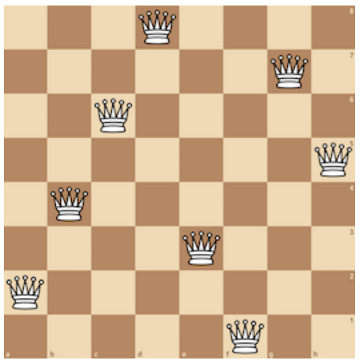


Figure 9: One solution to the puzzle

8.3.2 Data structure

An array of integers will suffice, with each index corresponding to column and each value of the array representing a row. There is no need to declare a two-dimensional array.

8.3.3 Backtracking

The puzzle serves as a good exercise to illustrate the **backtracking depth-first search** algorithm. Backtracking incrementally builds candidates to the solutions, and discards them (*backtracks*) as soon as the candidate cannot lead to valid solution. This approach is much more effecient than brute-force, where we would try to build all combinations on the board, even if they are doomed from the start, for instance by placing two queens on the first row and attempting to place a third one although this could never lead to a valid position because of the two first queens in check.

To solve an n queens board, we start by creating a slice of n integers. Then we solve boards of increasing size k, starting at $k = 0$ to $k = n$.

```go
func SolveNQueens(n int) {
    board := make([]int, n)
    solve(board, 0, n)
}
```

An auxiliary `solve()` functions does the heavy lifting. When $k = n$, it means we have managed to put n queens on the board, and we have found a valid solution that we can print. Otherwise, we first test if the board with the new queen is safe. In the affirmative, we try all possibles rows for column k.

```go
func solve(board []int, k, n int) {
    if k == n {
        fmt.Println(board)
    } else {
        for i := 0; i < n; i++ {
            if isSafe(board, k, i) {
                board[k] = i
                solve(board, k+1, n)
            }
        }
    }
}
```

Checking if a new queen is valid on the board is done in `isSafe()`.

```go
func isSafe(board []int, column int, row int) bool {
    for i := column - 1; i >= 0; i-- {
        d := column - i
        if board[i] == row || board[i] == row-d || board[i] == row+d {
            return false
        }
    }
    return true
}
```

Now we can solve a board of size `4` for instance:

```go
func main() {
    solveNQueens(4)
}
```

With the solutions:

```
[1 3 0 2]
[2 0 3 1]
```

8.3.4 Caveats

- Our solver **prints** a solution when it founds one, rather than returning a list of all solutions. This can be considered as a side-effect. It limits the ability of the caller to print the solutions the way is sees fit, for instance in chess notation. It brings simplicity however as a list of solutions does not have to be carried around by the solving function.
- For the sake of simplicity, we did not declare a custom type `Board` to abstract the more low level `[]int`. Again, this is fine in the context of a simple exercise, and we may argue that introducing a custom type would be overengineering the solution.

8.3.5 Other methods

- **Iterative** – an iterative solver would require more code and would be less intuitive.
- **Channels** – Go has a built-in mechansim called *channels*. It is used in the context of concurrent programming but it can act as an elegant communication mechanism between the solver and the caller, passing solutions from the former to the latter.
- **Representation** – The solutions could be displayed in more natural chessboard coordinates.

8.4 Conway's Game of Life

Cellular automaton.

8.4.1 Presentation

The Game of Life[24] is a cellular automaton devised by John Horton Conway[25] in 1970. It is a zero-player game, meaning that its evolution is determined by its initial state (*seed*), requiring

[24] https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
[25] https://en.wikipedia.org/wiki/John_Horton_Conway

no further input. One interacts with the game by creating an initial configuration and observing how it evolves. It is Turing complete[26].

8.4.2 Rules

The simulation evolves on an infinite, two-dimensional grid of cells, each of which is either live or dead. Every cell interacts with its eight neighbours. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if by underpopulation.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overpopulation.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

8.4.3 Example of patterns

This simple set of rules can lead to remarkable patterns. Common patterns include:

- **Oscillators** – return to their initial state after a finite number of generations called **periods** (*blinker*, *toad*, *beacon*, *pulsar*) ;
- **Still lifes** – remain unchanged from one generation to the next (*block*, *beehive*, *loaf*, *boat*, *tub*) ;
- **Spaceships** – travel accross the grid (*glider* and other spaceships).

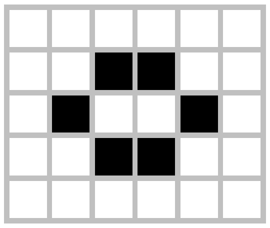


Figure 10: Beehive, one form of still life

Decades after Conway released his game, a self-replicating pattern called *Gemini* was discovered.

[26]https://en.wikipedia.org/wiki/Turing_completeness

It creates a copy of itself while destroying its parent. Other astonishing patterns include *gilder guns*.

8.4.4 Data structure

Some implementations use a two-dimensional array-like structures, for instace `[][]bool` to represent the grid. This is however inefficient and not practical. It cannot represent an infinite grid and can be a huge waste of memory on large grids. Instead, we represent a **cell** as pair of integers, and the **grid** as a set of cells. We shall use a `map[Cell]bool` to represent the grid.

```
// Cell represents a cell, living or dead
type Cell struct {
    X, Y int
}

// CellSet represents a set of cells
// used as a grid or a set of neighbors
type CellSet struct {
    cells map[Cell]bool
}
```

Let's add:

- A `String()` method to display a cell in a human-readable format ;
- An empty grid constructor ;
- A cell-based grid constructor.

```
func (c *Cell) String() string {
    return fmt.Sprintf("(%d, %d)", c.X, c.Y)
}

func NewCellSet() *CellSet {
    return &CellSet{make(map[Cell]bool)}
}

func CellSetFrom(cells ...Cell) *CellSet {
    cs := NewCellSet()
    for _, cell := range cells {
        cs.Add(cell)
    }
    return cs
}
```

```go
func (cs *CellSet) String() string {
    s := "["
    for cell := range cs.cells {
        s = s + cell.String() + ", "
    }
    s = strings.TrimRight(s, ", ")
    s = s + "]"
    return s
}
```

8.4.5 Implementation

Go does not offer a built-in type for set, so we need to bring our own helper functions like `Add`, `Contains` or `Intersect` to the table.

```go
func (cs *CellSet) Add(cell Cell) {
    cs.cells[cell] = true
}

func (cs *CellSet) Contains(cell Cell) bool {
    _, found := cs.cells[cell]
    return found
}

func (cs *CellSet) Intersect(other *CellSet) *CellSet {
    intersect := NewCellSet()
    for _, cell := range cs.Cells() {
        for _, otherCell := range other.Cells() {
            if cell == otherCell {
                intersect.Add(cell)
            }
        }
    }
    return intersect
}

func (cs *CellSet) Cells() []Cell {
    var cells []Cell
    for cell := range cs.cells {
        cells = append(cells, Cell{cell.X, cell.Y})
    }
    return cells
}
```

```go
func (cs *CellSet) Len() int {
    return len(cs.Cells())
}
```

With this setup, we can focus on the simulation itself. We will need a function that returns the neighbours of a cell.

```go
// Returns all the neighbours of the given cell, living or dead
func neighbours(cell *Cell) *CellSet {
    n := NewCellSet()
    for i := -1; i <= 1; i++ {
        for j := -1; j <= 1; j++ {
            if i != 0 || j != 0 {
                n.Add(Cell{i + cell.X, j + cell.Y})
            }
        }
    }
    return n
}
```

Lastly, `Next()` allows to transition from one state to the next. We first check if a cell survive to the next generation according to the rules. Then we examine all neighbor cells to determine if a new cell will appear (it must have 3 neighbors so the possible newborns always come next to living cell).

If the function returns an empty grid, we have reached a terminal state.

```go
// Next moves a grid to its next state
func Next(grid *CellSet) *CellSet {
    newGrid := NewCellSet()

    // All neighbours of all living cells.
    // They are all candidate newborns
    candidates := NewCellSet()

    // Survivors and current neighbours
    for _, cell := range grid.Cells() {
        neighbours := neighbours(&cell)
        countLivingNeighbours := grid.Intersect(neighbours).Len()
        if countLivingNeighbours == 2 || countLivingNeighbours == 3 {
            newGrid.Add(cell)
        }
        // Neighbors of this cell are newborn candidates
        for _, neighbour := range neighbours.Cells() {
            candidates.Add(neighbour)
        }
    }

    // Add eligible newborns
    for _, candidate := range candidates.Cells() {
        if !grid.Contains(candidate) {  // It's a empty cell
            neighbours := neighbours(&candidate)
            countLivingNeighbours := grid.Intersect(neighbours).Len()
            if countLivingNeighbours == 3 {
                newGrid.Add(candidate)
            }
        }
    }

    return newGrid
}
```

Create a `main()` method to run the simulation with different input, for instance

- Trival case: an empty grid that remains empty.
- A single cell that dies after the first generation.
- A beehive (still life) that remains unchanged, no matter how many times you call the next generation.
- A blinker of period 2.

For example:

```go
NewCellSet()
CellSetFrom(Cell{0, 0})
CellSetFrom(
    Cell{2, 1}, Cell{3, 1}, Cell{1, 2},
    Cell{4, 2}, Cell{2, 3}, Cell{3, 3})
CellSetFrom(Cell{1, 0}, Cell{1, 1}, Cell{1, 2})
```

8.4.6 Exercise

The `String()` grid and cell representations we provided are accurate, although a bit dull. As an exercise, display the automaton in a grid-like manner.

8.5 Tower of Hanoi

Learn recursion with this classical problem.

8.5.1 Presentation

The Tower of Hanoi[27] is a mathematical game or puzzle consisting of three rods (or sticks) and a number of disks (or pegs) of different diameters, which can slide onto any rod. The puzzle starts with the disks stacked on one rod in order of decreasing size, the smallest at the top.

For instance, let us label the 3 rods `A`, `B` and `C` and use 3 disks `{3, 2, 1}`, `3` being the largest:

```
A = {3, 2, 1}
B = {}
C = {}
```

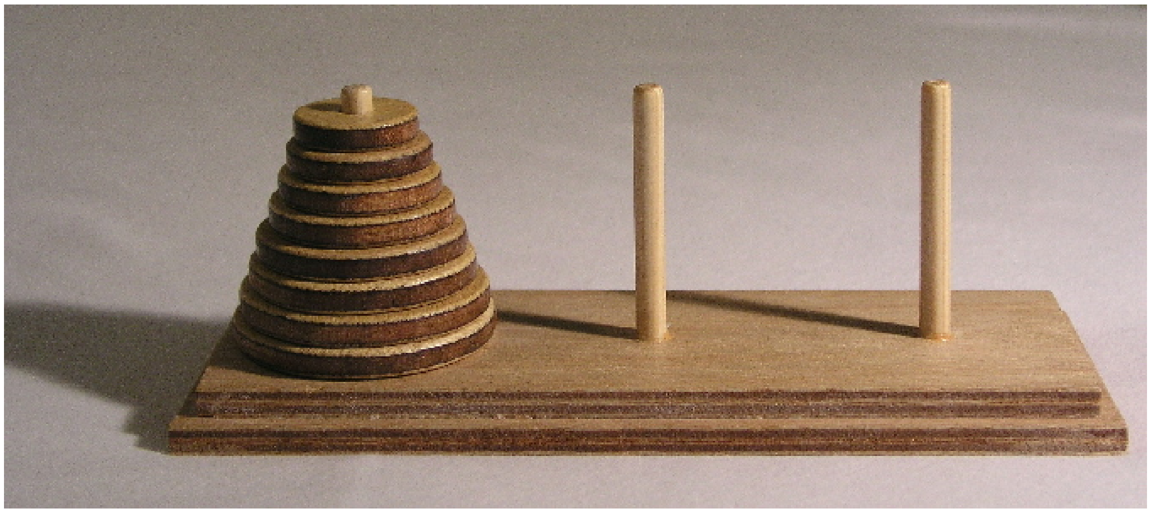


Figure 11: A model set of the Tower of Hanoi with 8 disks

8.5.2 Rules

The objective is to move the entire stack to the last rod, obeying the following simple rules:

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

1. Only one disk may be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No disk may be placed on top of a disk that is smaller than it.

The minimal number of moves required to solve a Tower of Hanoi puzzle is $2^n - 1$, where n is the number of disks.

8.5.3 Recursive method

If we only display the moves and not the rods successive states, we do not even need a data structure to represent the rods and the disks. This solution is straightforward, at the cost of testability.

To $solve(n, A, B, C)$ for n disks using pegs A (**source**), B (**auxiliary**) and C (**destination**) with $n > 0$:

1. $solve(n-1, A, C, B)$
2. $move(A, C)$
3. $solve(n-1, B, A, C)$

To phrase it in a more casual form, say you are asked to solve for n pegs, but you don't know how to do it. But you know a friend that can solve it for $n-1$. How? That's none of your concern, but it turns out that your friend delegates for $n-2$ to yet another person, and so on. Eventually, there's that special someone that can solve it for $n = 1$. It is a trivial case: they must simply move the top peg.

So to solve for (n, A, B, C), you call your friend to solve for $(n-1, A, C, B)$. Then you call the special someone to move the top peg from A to C. You call your friend again, this time to solve $(n-1, B, A, C)$.

```go
func solve(n int, source, auxiliary, destination string) {
    if n == 0 {
        return
    }
    solve(n-1, source, destination, auxiliary)
    fmt.Printf("Move from %s to %s\n", source, destination)
    solve(n-1, auxiliary, source, destination)
}
```

```go
func main() {
    solve(4, "A", "B", "C")
}
```

And that's it! Problem solved, no sweat.

8.5.4 Iterative method

The iterative approach is much more involved than the recursive one. It requires to create a type `Tower`. Each rod is an `[]int` labelled by a string.

```go
type Tower struct {
    n int
    rods map[string][]int
}

func (tower *Tower) String() string {
    return fmt.Sprintf("%v", tower.rods)
}
```

Provide a new tower constructor based on the number of disks:

```go
func NewTower(n int) *Tower {
    var disks []int
    for i := n; i > 0; i-- {
        disks = append(disks, i)
    }
    tower := &Tower{n,make(map[string][]int)}
    tower.rods["A"] = disks  // descending [3, 2, 1]
    tower.rods["B"] = make([]int, 0)
    tower.rods["C"] = make([]int, 0)
    return tower
}
```

We will need to `swap` disks "smartly", depending on the rod's configuration.

```go
func (tower *Tower) swap(x, y string) {
    topDiskX := tower.topDisk(x)
    topDiskY := tower.topDisk(y)
    if topDiskX == 0 && topDiskY == 0 {
        return
    }
    if (topDiskX < topDiskY && topDiskX != 0) || topDiskY == 0 {
        tower.move(x, y)
    } else {
        tower.move(y, x)
    }
}
```

Return the top disk of the given rod:

```go
func (tower *Tower) topDisk(rod string) int {
    count := len(tower.rods[rod])
    if count == 0 {
        return 0
    }
    return tower.rods[rod][count-1]
}
```

We need to be able to move a disk from one rod to another:

```go
func (tower *Tower) move(from, to string) {
    fmt.Println(tower)
    fmt.Printf("Move from %s to %s\n", from, to)
    countFrom := len(tower.rods[from])
    topFrom := tower.rods[from][countFrom - 1]
    tower.rods[from] = tower.rods[from][:countFrom - 1]
    tower.rods[to] = append(tower.rods[to], topFrom)
}
```

We know we are done when the destination disk `C` contains all the disks, that is, its size is the size `n` of the tower.

```go
func (tower *Tower) isDone() bool {
    return len(tower.rods["C"]) == tower.n
}
```

At last, to put it all together, we have two sets of rules:

- Even number of disks
- Odd number of disks

For an even number of disks:

2. Make the legal move between rods A and C (in either direction),
3. Make the legal move between rods A and B (in either direction),
4. Make the legal move between rods B and C (in either direction),

For an odd number of disks:

1. make the legal move between rods A and C (in either direction),
2. make the legal move between rods A and B (in either direction),
3. make the legal move between rods B and C (in either direction),

Repeat the steps until complete.

The part "in either direction" explains why we designed a generic function `swap()` that is able to `move(x, y)` or `move(y, x)`.

We can now solve according to our algorithm above.

```go
func (tower *Tower) solve() {
    var steps [][]string

    if tower.n % 2 == 0 {
        steps = [][]string{{"A", "B"}, {"A", "C"}, {"B", "C"}}
    } else {
        steps = [][]string{{"A", "C"}, {"A", "B"}, {"B", "C"}}
    }
    for {
        for _, step := range steps {
            tower.swap(step[0], step[1])
            if tower.isDone() {
                return
            }
        }
    }
}
```

The implementation requires us to retain the tower's state after each step. A `main` function will help our algorithm come to life.

```go
func main() {
    tower := NewTower(4)
    tower.solve()
    fmt.Println(tower)
}
```

8.6 Blackjack (guided exercise)

8.6.1 Rules

Blackjack is a casino banking game. We will play a simplified version of the game: single player, single deck of cards, no bets, no hole card.

At the table, the dealer faces the player. The card deck is shuffled. The dealer deals 2 cards to the player, face-down. The dealer's hand is also two cards face-down.

The player's goal is to create a card total higher than those of the dealer's hand but not exceeding 21.

On their turn, the player chooses to "hit" (take a card) or "stand" (end their turn and stop without taking a card). Number cards count as their number. Jack, queen and king count as 10. Aces count as either 1 or 11 according to the player's choice. If the total exceeds 21 points, it busts, and the player or dealer immediately loses. The winner scores one point.

Cards are left aside after each hand. After seven hands, the dealer grabs all the cards and reshuffle them.

8.6.2 Play

The player will be a human interacting with the program. The dealer will be played by the computer.

The dealer's decision process is simple: if their total is below the player's, they hit. If it's above, they stand. If it's equal, to make things simple, the dealer stands.

8.6.3 Design

Make sure you have completed the Hangman chapter before starting this exercise. You first task is to decompose the problem in smaller, manageable chuncks. The main design ideas are identical or very similar.

- Accept player's input when it's their turn
- Validate it
- Perform dealer's logic

- Keep track of the score
- Detect a bust
- Detect the end of the hand
- Compute total
- Handle aces (1 or 11)
- …

The game runs indefinitely. It is an endless loop. You can allow the player to enter `'q'` to quit the game at any time. Inside that loop live **nested loops** running for each hand, handling the player's and the dealer's decisions.

Here is a simple suggested algoirthm, where the deck is shuffled before each hand (rather than after 7 hands).

```
scores = 0, 0
while not quit {
    shuffle
    deal
    while player not standing and player not bust {
        ask decision
        if hit {
            add to hand
        }
    }
    while dealer score < player score {
        dealder hit
        add to hand
    }
    decide winner
}
display scores
```

To make your life easier, you ignore the aces double value in your first version, counting them as 11.

Good luck!

Figure 12: A table of blackjack

9 Appendix A - Install and run

9.1 Go playground

For the impatient, or when you are working from a computer where Go has not been installed, the Go Playground allows you to run code in the browser.

https://play.golang.org/

9.2 Install

Go to the installation page and follow the instructions.

https://go.dev/doc/install

On macOS, consider using Homebrew.

https://formulae.brew.sh/formula/go

9.3 Edit

For this book, any decent editor will do.

Visual Studio Code is recommanded on all platforms.

https://code.visualstudio.com/

9.4 Run

```go
// cat main.go

package main

import "fmt"

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

Alternatively, you can open the file from your file explorer or navigator as well.

Run with `go run`:

```
$ go run main.go
Hello, world!
```

9.5 Modules

In some chapters, we make the code more modular by using modules.

A good example is found in the ch. 5.5 Stack. You don't need to understand the details of the code if you just began reading this book.

In this example, we have:

- a module file `go.mod`
- a stack implementation in `package stack`
- a demo in `package main`.

From a directory of your choosing, the file structure is:

```
+-- ipgo
    +-- exercise
        +-- stack
        |   +-- stack.go
        +-- main.go
        +-- go.mod
```

First, create a module:

```
$ go mod init be.sugoi.ipgo
```

Go will create the following file:

```
$ cat go.mod
module be.sugoi.ipgo

go 1.20
```

I chose `be.sugoi.ipgo` because it matches this book's namespace, but you can choose whatever you like.

The stack:

```go
// stack.go

package stack

type Stack[T any] []T

func New[T any]() *Stack[T] {
    return &Stack[T]{}
}

func (s *Stack[T]) Len() int {
    return len(*s)
}

// rest of the code omitted
```

The demo:

```go
// main.go

package main

import (
    "fmt"
    "be.sugoi.ipgo/stack"
)

func main() {
    s := stack.New[int]()
    fmt.Println(s.Len()) // 0
}
```

10 Appendix B - Credits

- Eisvogel pandoc LaTeX template, copyright 2017 - 2020, Pascal Wagler, copyright 2014 - 2020, John MacFarlane
- Hard drive, Wikipedia, licensed under CC BY-SA 3.0.
- Gopher mascot by Takuya Ueda, licensed under the Creative Commons 3.0 Attributions license.
- Ariane 501, Copyright ESA
- Thinking monkey, photo by Juan Rumimpunu, licensed under Unsplah license. https://unsplash.com/photos/nLXOatvTaLo
- Chessboard, Lichess.
- Tower of Hanoi, Wikipedia, licensed under CC BY-SA 3.0.
- Fibonnaci tiles, Wikipedia, licensed under Creative Commons Attribution-Share Alike 4.0 International
- Blackjack table, Wikipedia, licensed under Creative Commons CC0 1.0 Universal Public Domain Dedication
- Pointer meme, u/tuunraq, Reddit, all rights reserved.