Lua 102

Did you ask for more awesome?

Contents

Functions

Functions

A *function* is a named block of code, which can be executed from somewhere else in the program. A function can have *parameters* (inputs) and *return values* (outputs).

Functions are useful for:

- code organisation (separate various problems into distinct pieces of code)
- code reuse (write once, use many times)
- abstraction (solve a problem for a generic input)

A simple function

To *define* a function, we use the keyword `function`:

```
-- definition

function printHello()
    print("Hello!")
end
```

Defining the function doesn't execute the code, it only gives it a name.

To *call* a function, we use the function name, followed by `()`:

```
-- call

printHello()
print("We did it!")
```

When calling a function, control moves inside the function, executes all its instructions, then comes back to where it was called and continues with the next instruction.

Functions: syntax

Definition:

```
function name()
    -- block
end
```

Call:

```
name()
```

Reminder: defining the function doesn't execute it, it saves it for later use. Calling the function is when it is executed.

Note: function names: same rules as variable names (letters, numbers, underscores, cannot start with a number).

Function body

The block inside a function (it's *body*) can be any number of instructions, including `if`s, loops, assignments, `print()`, etc.

```
function printHello5Times()
    print('Printing "Hello" five times:')
    for i = 1,5 do
        print("Hello")
    end
    print("Done!")
end
```

Calling functions

A function call is an instruction like any other, it can be done anywhere (in any block of code):

```
printHello5Times()

for i = 1,10 do
    printHello5Times()
end

if 10 < 100 then
    printHello5Times()
end
```

Mini exercise: functions

- Define a function that outputs your name 10 times. Then call this function a few times.
- Define a function that prints "Foo", and a function that prints "Bar". Then complete the program so that it outputs the following 10 times:

```
Foo
Foo
Bar
```

Calling other functions

Like any other instructions, functions can call other functions:

```
function foo()
    print("Foo")
end

function bar()
    print("Bar")
    foo()
    print("Bar2")
end

bar()   -- ouputs: Bar Foo Bar2
```

Side effects

Functions can also modify variables, and output stuff. In general, anything that modifies memory or peripherals is called a *side effect*:

```
function foo()
    a = 5
    print("I did it!")
end

print(a)    -- nil
foo()       -- I did it!
print(a)    -- 5
```

Mini exercice: side effect

Write a function `increment()` that increases the value of variable `a` by 1, so that the following program:

```
-- your code here

a = 10
increment()
increment()
print(a)
```

would output 12.

Function parameter

A function can be defined to have a *parameter* (input). When we call that function, we will *pass* a value, that the function can use for computations or other stuff:

```
-- definition

function printThisNumber(number)
    print("You asked me to print: " .. number)
end

-- call

printThisNumber(5)
```

When we call `printThisNumber(5)`, not only control passes to the function, but also the value 5 is assigned to the variable `number` inside the function.

Parameter: syntax

Definition:

```
function name(parameterName)
    -- block
end
```

The parameter name follows the same rule as variable (it *is* in fact a variable).

Call:

```
name(expression)
```

In the call, we put an expression inside the parentheses:

- a literal: `-3.5 "Hello" true`...
- a variable
- an operation: `3 + 5`

Parameter

Parameters can be of any type:

```
function printThisTwice(s)
    print(s)
    print(s)
end

printThisTwice("Bob")
printThisTwice(200)
```

But remember that some operations work only with certain types:

```
function printDouble(n)
    print(n * 2)
end

printDouble(5)       -- 10
printDouble("Foo")   -- ERROR!!
```

Mini exercices: parameter

Write a function that:

- prints whether the parameter is a positive number (bigger than 0) or not
- prints "Hello world" `N` times, where `N` is the function parameter
- prints the 10 first multiples of the parameter `N` (eg. given 3, it would print 3, 6, 9, 12, etc.)
- prints whether the parameter is a multiple of 7

Multiple parameters

A function can be defined to have multiple parameters (separated with commas: `,`)

```
function multiply(a, b)
    print(a * b)
end
```

A function with multiple parameters is called by passing multiple values (comma separated expressions)

```
multiply(10, 4)

num = 129
multiply(num, 8 + 1)
```

When calling the function, all the values passed are assigned to the parameters (in the same order) inside the function, which can use them for whatever.

Mini exercices: multiple parameters

Write a function that:

- takes 2 parameters, and prints each on one line
- takes 2 parameters (strings or numbers), and prints them both on one line
- takes one string `s` and one number `N`, and prints the string `s`, `N` times
- takes three numbers, and prints their sum

Return value

A function can have a *return value* (output): a result that is *returned* to the caller. We use the keyword `return`, followed by an expression:

```
function double(n)
    return n * 2
end
```

When a function has a return value, the function call is itself an expression, that can be used in `print()`, operations, assigned to variables, etc.

```
print(double(10))

a = double(12)
print(a)

print(double(12) + 5)
```

Return instruction

The `return` instruction can be at the end of any block inside the function.
Mostly useful once you have the result of a computation:

```
function weird(n)
    if n % 2 == 0 then
        return n + 1
    else
        return n * 3
    end
end

function solve(n)
    for i = 1,n do
        -- some computation
        if foundSolution then
            return solution
        end
    end
end
```

When encountering `return`, control will return *immediately* to the caller.

Return type

You can return any value you want, and the type of the return value becomes the type of the function call expression:

```
function concatFoo(s)
    return s .. "Foo"
end

print(concatFoo("Hi"))              -- HiFoo
print(concatFoo("Hi") .. "Bar")     -- HiFooBar
```

Here, the call `concatFoo("Hi")` is an expression of type string and can be used anywhere a string can be used.

Mini exercises: returning values

Write a function that:

- takes two numbers A and B, and returns A × A + B × B
- takes a string `s`, a number `N`, and returns `s` concatenated to itself `N` times (for instance passing "Ben" and 3 would return "BenBenBen")
- takes a number `N`, and returns true is `N` is even, or false if `N` is odd

Details about functions

Mixing side effects

You can use side effects and return values in the same functions:

```
function multiply(a,b)
    print("Let's multiply " .. a .. " and " .. b)
    return a * b
end

result = multiply(3,6)
print(result)
```

Ignoring the return value

If a function returns a value, but you don't use it (assign it to a variable, or print it, etc.), it is discarded. The function is still executed (side effects will happen).

```
function foo(n)
    print("This is foo " .. n)
    return n * n + 2
end

foo(3)  -- outputs: This is foo 3
```

Here, the return value (11) is computed, but discarded immediately.

Forgetting the return value

Conversely, trying to save the return value of a function that doesn't return anything will result in `nil`:

```
function doSomething()
    print("Doing something")
end

print(doSomething())
a = doSomething()
print(a)
```

We say that the function doesn't return anything, or "returns nil".

Forgetting the return value

A more likely scenario:

```
function half(n)
    if n % 2 == 0 then
        return n / 2
    end
end

print(half(10))     -- 5
print(half(11))     -- nil
```

It can happen:

- when you forgot one possible case
- on purpose: `nil` can mean "no result"

Returning nil

You can also force the function to return immediately using `return` alone.

```
function blah()
    for i = 1,1000 do
        -- something
        if someReason then
            return
        end
    end
end
```

Note: it will return nil. In fact you can even explicitly write `return nil`.
Same thing.

A well known function

```
print(5)
print("Hello")
```

`print()` is a built-in function: it takes arguments of any type, and outputs them to the console.

It has only side effects though (it doesn't return anything):

```
a = print("Hi")      -- Hi
print(a)             -- nil
```

In fact, it can take multiple arguments (prints them on one line, separated with tabs):

```
print(1, "Hello", true)     -- 1    Hello   true
```

Convenient for quick output where you don't care about formatting.

Lua's standard library

Complete documentation in the reference (we'll see later what the dot in the function names means).

```
-- absolute value
print(math.abs(-4))          -- 4

-- pi (a variable, not a function)
print(math.pi)               -- 3.141592

-- cosine and sine
print(math.cos(math.pi))     -- -1
print(math.sin(0))           -- 0

-- square root
print(math.sqrt(81))         -- 9

-- min and max
print(math.max(4,19,11))     -- 19
print(math.min(4,19,11))     -- 4

-- random numbers
print(math.random())         -- eg. 0.0012512588885159
```

Lua's standard library (2)

Error function:

```
if somethingWrong then
    error("AARGH??")          -- panic: exit program
end
```

OS:

```
print(os.clock())  -- seconds since the start of the program
print(os.time())   -- seconds since 1 January 1970 00:00 UTC
print(os.date())   -- the current date as a string
os.exit()          -- immediately exit program
```

Exercise: palindrome

A palindrome is a phrase that can be read in both directions, like:

> A man, a plan, a canal, Panama!

In Lua, the function `string.sub` lets you extract substrings. Without going into the details, know that `string.sub(s, i, i)` will return the `i`-th letter of string `s`:

```
print(string.sub("Great!", 3, 3))   -- outputs: e
```

Write a function that recognizes if a string is a palindrome. To simplify, assume there is no punctuation, capitals or spaces, so that the function would only need recognize `"amanaplanacanalpanama"` as a palindrome.

Hint: there are many ways to do this. Try to analyze how you would do it "in real life", and then translate that algorithm into Lua.

Bonus question: modify your function so that it ignores spaces, and recognizes `"a man a plan a canal panama"` as a palindrome.

Scope

Lexical scoping

> *Scope*: the region of program source in which an identifier is meaningful.

Scoping is the fact that variables can be visible or not to other parts of the program, depending on where they are defined.

Lexical scoping means that these visibility rules depend only of the source code, and not of the state of the program at execution (the standard in most languages, much easier to visualize, understand, and maintain).

In Lua: exactly two scopes: *local* and *global*.

Global variables

The ones we have used so far:

- no need to declare (just use them)
- visible from *everywhere*: can read them and write them in any block of code

```
function setIt()
    a = 5
end

function printIt()
    print(a)
end

printIt()       -- nil
setIt()
print(a)        -- 5
printIt()       -- 5
a = 10
printIt()       -- 10
```

Local variables

- must be declared using `local` keyword
- exist *only* in the block of code they were declared in

```
a = 6

if a > 3 then
    local double = a * 2
    print(double)           -- 12

    -- double goes "out of scope"
end

print(double)   -- nil, because trying to access a global
```

Declaring a local

No need to initialize immediately:

```
local zob       -- declared, still nil
if something then
    zob = 5
else
    zob = 10
end

print(zob)
```

Shadowing

Local variables *shadow* variables of outer scopes:

```
a = 10
local b = 11

if 10 < 100 then
    local a = 100
    local b = 200
    print(a,b)      -- 100    200

    -- locals a and b go out of scope
end

print(a,b)      -- 10    11
```

Implicit locals

We've been using locals already!

Implicitely declared local variables:

- iterator in `for` loops
- parameters in functions

```
i = 50
for i = 1,10 do
    print(i)    -- 1, 2, 3...
end
print(i)        -- 50

function printTwice(i)
    print(i)
    print(i)
end

printTwice("lol")   -- lol, lol
print(i)            -- 50
```

Blocks and scopes

Every new block of code opens a new scope:

- `then`, `elseif` and `else` blocks in `if` statement
- the body of `for` loops
- the body of a function
- a Lua chunck (because it's actually a function)

```lua
local foo = 5
for i = 1,10 do
    local foo = i * 2
    print(foo)
    if i < 5 then
        local foo = "Smaller"
        print(foo)
    else
        local foo = "Bigger"
        print(foo)
    end
end
print(foo)
```

Note: one more reason to indent your code correctly: a visual reminder of the blocks, and therefore, of the scopes of your variables.

Locals: tips

- Always use locals inside a function. Have functions act on their arguments instead of globals.
  - cleaner code: input -> output, no side effects which are more difficult to track
  - more independent logical unit

```
-- compare:
doSomething()   -- what did you do, did you change anything?

-- to:
output = treatData(input)   -- that's quite clear
```

- Small, temporary results should be locals too: use them freely, throw them away.
- In fact, very few cases where globals are necessary, or useful.
- As programs grow, everything will be functions, so all variables will be locals.

Exercises: locals

Rewrite these previous exercises using functions and *only* local variables:

- write a function that checks if a number `N` is prime (it should return a boolean), then:
  - print all primes under 1000
  - print the first prime bigger than one million
  - print the 2000th prime number
- write a function that returns the `N`-th Fibonacci number
- fix your palindrome functions so that they don't use any globals

Don't hesitate to:

- experiment with global and local variables
- make sure you understand the scoping rules

In the future:

- think "one problem, one function"
- break down problems in smaller problems: functions that call other functions
- use locals!
- avoid globals!

More loops

The `while` loop

Repeat a block of code as long as a condition is true:

```
i = 1
while i < 1000 do
    print(i)
    i = i + 1
end
```

Or typically:

```
continue = true

while continue do
    -- computations
    if foundSolution then
        continue = false
    end
end
```

`while` loop: syntax

```
while condition do
    -- block
end
```

where `condition` should be an expression of type boolean.

How to choose between `while` or `for`:

- `for` loop when you already know exactly how many times to loop
- `while` loop when you don't know yet how many times to loop

Infinite loops!

Make sure that the body of the loop actually has an effect, otherwise, the loop might repeat forever!

```
i = 1
while i < 100 do
    print(i)
    -- OH NO! I forgot to increment i.
end
```

Mini exercises: `while`

- Write a program that counts down from 1000, using a `while` loop
- Test the Collatz conjecture: starting from an integer N, if N is even, divide it by 2, if it is odd, multiply by 3 and add 1. Repeat the process until you get 1. Write a function that, given an integer N, prints all the steps of its Collatz sequence.

Example sequence, starting from 10: 10, 5, 16, 4, 2, 1

Bonus loop: the `repeat` loop

Similar to `while`, but:

- the condition is a *stopping* condition
- we test the condition *after* executing the block (the block is always executed at least once)

```
repeat
    -- block
until condition
```

Used fairly rarely, but can result in more compact and elegant code.

Mini exercise: use a `repeat` loop to write all numbers from 1 to 100.

Breaking out, bis

The `break` instruction also works with `while` and `repeat` loops.

Interesting idiom:

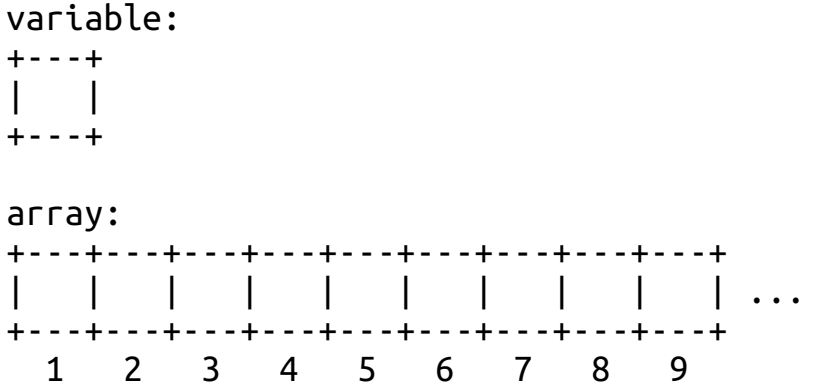
```
while true do
    -- OH NO? Is this an infinite loop?

    if weAreDoneHere then
        break
    end
end
```

Arrays

Arrays

The first non-basic type in Lua: a *collection* of values identified by a numerical *index*.

- so far, each variable contained *one* value
- in an array, we can store *many* values



Creating arrays

Created using an array literal:

```
foo = {}    -- create empty array, stores it in variable foo
bar = {"lol", 5 , true, 2}   -- create array with 4 elements
```

```
foo:
++
||
++

bar:
+-------+---+------+---+
| "lol" | 5 | true | 2 |
+-------+---+------+---+
    1     2    3     4
```

Array indexing

Values are stored and retrieved using integer indices and the square brackets syntax:

```
bar = {"a", "b", "c", "d"}
print(bar[2])   -- b
bar[3] = "lol"
print(bar[3])   -- lol
```

Note: Indices start at 1.

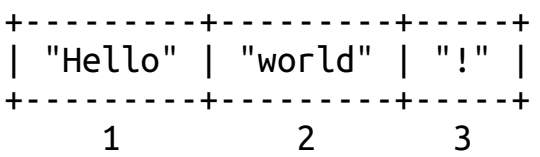
Can use any expression as index, as long as it is a positive integer:

```
index = 3
foo = {11, 12, 13}
print(foo[1 + 1], foo[index])   -- 12    13
```

Variable size

An array's size change as you add elements to it:

```
array = {}
array[1] = "Hello"
array[2] = "world"
array[3] = "!"
```



Mini exercises: arrays

- create an array that contains all numbers from 1 to 100
- create an array that contains the string "What" 50 times
- create an array that contains the 30 first multiples of 7 (7, 14, 21, ...)

Missing elements

Trying to access an element that is not set returns `nil`:

```
array = {"hello"}
print(array[1000])  -- nil
```

Likewise, you can set an element to `nil` to erase it:

```
array = {1, 2, 3, 4, 5}
array[3] = nil

for i = 1,5 do
    print(array[i])     -- 1, 2, nil, 4, 5
end
```

Sparse arrays

You don't have to set elements contiguously (the array can have gaps):

```lua
array = {}
array[100] = "foo"
for i = 1,200 do
    print(array[i]) -- nil, nil, nil, ..., foo, nil, nil...
end
```

In that sense, the Lua array sort of has infinite size, but all its elements start as `nil`.

Array length operator

When an array does not have gaps, the `#` operator returns its length (the number of elements in it):

```
array = {"foo", "bar", 1, true}
print(#array)   -- 4
array[4] = nil
print(#array)   -- 3
```

If some elements are `nil`, it may or may not work: unless you have a good reason to, don't put `nil` in the middle of an array.

Common array idioms

```lua
-- iterate over all elements of array
for i = 1,#array do
    -- set or get array[i]
end

-- add element at the end of array
array[#array + 1] = "foo"

-- remove last element of array
array[#array] = nil

-- print array contents with separator
print(table.concat(array, ","))
```

Mini exercises: arrays

Complete the following program:

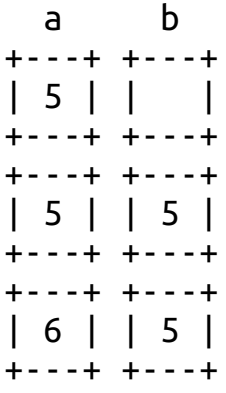
```
array = {1,2,3,4,5,6,7,8,9,10}

-- your code here
```

- to print all of `array`'s elements (one by line)
- to replace each of `array`'s elements with the string "Haha"
- without using `table.concat()`, print all of `array`'s elements on one line, separated with commas

Reference type

All basic types (number, string, boolean, nil) are *value* types: the variable *contains* the value. Assignment copies it:

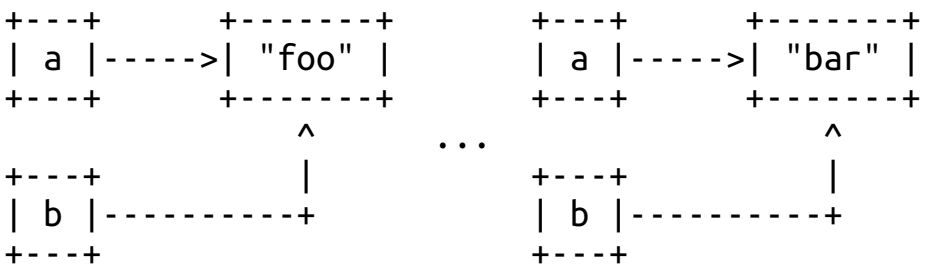
```
a = 5
b = a
a = 6
print(a,b)  -- 6    5
```



Reference type

The array is a *reference* type. When creating an array, some memory is allocated "somewhere", and a *reference* to it is returned. A variable can only contain the reference to the array, not the array itself. Several variables can hold a reference to the same array:

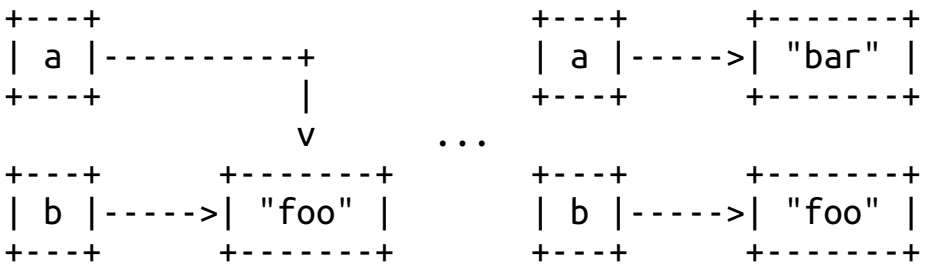
```
a = {"foo"}
b = a
print(a[1], b[1])     -- foo    foo
a[1] = "bar"
print(a[1], b[1])     -- bar    bar
```



Losing references

When an array reference is erased or replaced, it doesn't affect the array itself:

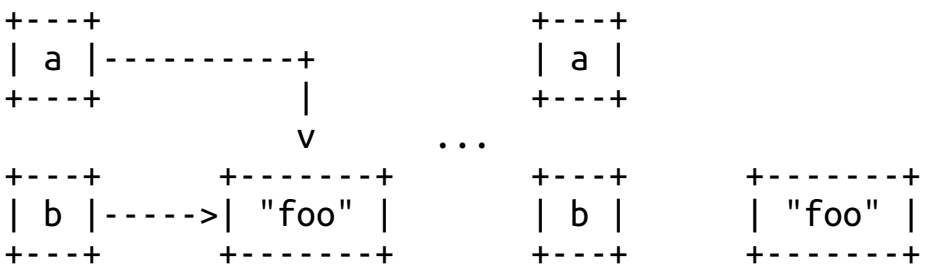
```
a = {"foo"}
b = a
a = {"bar"}
print(a[1], b[1])   -- bar    foo
```



Deleting arrays

An array is deleted only if *all* the references to it are lost:

```lua
a = {"foo"}
b = a
a = nil
print(b[1])     -- foo
b = nil
-- can't access it anymore!
```



The *garbage collection* takes care of freeing the memory.

Passing arrays

Passing "an array" (ie. a reference to an array) to a function allows the function to modify it:

```
function foo(array)
    array[1] = "foo"
end

a = {"a", "b", "c"}
foo(a)
print(table.concat(a, ",")) -- foo,b,c
```

Likewise, functions can "return arrays" (ie. return a reference to an array):

```
function makeArray()
    local array = {}
    -- do something with it
    return array
end
```

Vocabulary

By abuse of language:

- assign an array (assign a reference to an array)
- the variable contains an array (refers to an array)
- pass an array to a function (pass a reference to an array)
- return an array (return a reference to an array)
- variable of array type (variable of array reference type)
- ...

It's ok to take shortcuts as long as the semantics are clear! In Lua, *all* arrays are by-reference, unlike some other languages.

Mini exercises: array functions

- write a function that reverses the order of elements in an array (`{1,2,3,4}` would become `{4,3,2,1}`)
- write a function that returns the value of the smallest element of an array (assume the array contains numbers only)
- write a function that takes an array and another value, and returns true if the array contains that value
- write a function that prints all pairs of elements of an array (for instance, passing `{"a",1,true}` would print `(a,1)(a,true)(1,true)`)
- write a function that returns a copy of an array (a new array, but with the same elements)

Insert/remove

Insert element in array (pushing all following elements up):

```
array = {1,2,3,4,5,6,7,8,9,10}
table.insert(array, 5, "lol")
print(table.concat(array, ","))     -- 1,2,3,4,lol,5,6,7,8,9,10

-- with no argument: add at the end
table.insert(array, "bar")
print(table.concat(array, ","))     -- 1,2,3,4,lol,5,6,7,8,9,10,ba
```

Remove element in array (pushing all following elements down):

```
array = {1,2,3,4,5,6,7,8,9,10}
table.remove(array, 5)
print(table.concat(array, ","))     -- 1,2,3,4,6,7,8,9,10

-- with no argument: remove from the end
table.remove(array)
print(table.concat(array, ","))     -- 1,2,3,4,6,7,8,9
```

Mini exercise: insert/remove

Adding/removing elements at the *end* of the array is efficient. Doing it somewhere else may result in up to `#array` operations internally (the pushing up/down of elements)

Write functions equivalent to `table.insert` and `table.remove` yourself, to understand why.

Exercices

- write a function `fib(n)` that returns an array containing the `n` first Fibonacci numbers
- write a function that given a integer, returns an array containing its digits (for instance, passing 2098 would return `{2,0,9,8}`)
- write a function that, given a integer up to 999999, returns its spelling in Finnish (passing 7654 would return the string "seitsemäntuhattakuusisataaviisikymmentäneljä")
  - start with numbers up to 99
  - then with numbers up to 999
  - then complete the exercise

Hint: remember to break down the problems in smaller problems, and to reuse functions or techniques you already know!

Exercise: sorting an array

To prepare for next lesson (algorithmics): write a function `sort(arr)` that, given an array of numbers, returns an array containing the same numbers, but sorted in increasing order.

```
result = sort({3,20,500,8,3,45,91,31,23})
print(table.concat(result, ","))

-- should output: 3,3,8,20,23,31,45,91,500
```

- there are many, many ways
- if stuck, get a shuffled pack of cards, sort them by hand and try to break down your algorithm into its essential parts
- if you find several solutions, try to compare their efficiency: how many operations happen in each, for an array of size N (comparisons, copies, swaps, etc.)