Intro to property based tests

You can find all the code for this chapter here

Some companies will ask you to do the Roman Numeral Kata as part of the interview process. This chapter will show how you can tackle it with TDD.

We are going to write a function which converts an Arabic number (numbers 0 to 9) to a Roman Numeral.

If you haven't heard of Roman Numerals they are how the Romans wrote down numbers.

You build them by sticking symbols together and those symbols represent numbers

So `I` is "one". `III` is three.

Seems easy but there's a few interesting rules. `V` means five, but `IV` is 4 (not `IIII`).

`MCMLXXXIV` is 1984. That looks complicated and it's hard to imagine how we can write code to figure this out right from the start.

As this book stresses, a key skill for software developers is to try and identify "thin vertical slices" of *useful* functionality and then **iterating**. The TDD workflow helps facilitate iterative development.

So rather than 1984, let's start with 1.

Write the test first

```
1 func TestRomanNumerals(t *testing.T) {
2         got := ConvertToRoman(1)
3         want := "I"
4
5         if got != want {
6                 t.Errorf("got %q, want %q", got, want)
7         }
8 }
```

If you've got this far in the book this is hopefully feeling very boring and routine to you. That's a good thing.

Try to run the test

`./numeral_test.go:6:9: undefined: ConvertToRoman`

Let the compiler guide the way

Write the minimal amount of code for the test to run and check the failing test output

Create our function but don't make the test pass yet, always make sure the tests fails how you expect

```
1 func ConvertToRoman(arabic int) string {
2         return ""
3 }
```

It should run now

```
1 === RUN   TestRomanNumerals
2 --- FAIL: TestRomanNumerals (0.00s)
3     numeral_test.go:10: got '', want 'I'
4 FAIL
```

Write enough code to make it pass

```
1 func ConvertToRoman(arabic int) string {
2         return "I"
```

```
}
```

Refactor

Not much to refactor yet.

I know it feels weird just to hard-code the result but with TDD we want to stay out of "red" for as long as possible. It may *feel* like we haven't accomplished much but we've defined our API and got a test capturing one of our rules; even if the "real" code is pretty dumb.

Now use that uneasy feeling to write a new test to force us to write slightly less dumb code.

Write the test first

We can use subtests to nicely group our tests

```
func TestRomanNumerals(t *testing.T) {
	t.Run("1 gets converted to I", func(t *testing.T) {
		got := ConvertToRoman(1)
		want := "I"

		if got != want {
			t.Errorf("got %q, want %q", got, want)
		}
	})

	t.Run("2 gets converted to II", func(t *testing.T) {
		got := ConvertToRoman(2)
		want := "II"

		if got != want {
			t.Errorf("got %q, want %q", got, want)
		}
	})
}
```

Try to run the test

```
=== RUN   TestRomanNumerals/2_gets_converted_to_II
    --- FAIL: TestRomanNumerals/2_gets_converted_to_II (0.00s)
        numeral_test.go:20: got 'I', want 'II'
```

Not much surprise there

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {
        if arabic == 2 {
                return "II"
        }
        return "I"
}
```

Yup, it still feels like we're not actually tackling the problem. So we need to write more tests to drive us forward.

Refactor

We have some repetition in our tests. When you're testing something which feels like it's a matter of "given input X, we expect Y" you should probably use table based tests.

```
func TestRomanNumerals(t *testing.T) {
        cases := []struct {
                Description string
                Arabic      int
                Want        string
        }{
                {"1 gets converted to I", 1, "I"},
```

```
                {"2 gets converted to II", 2, "II"},
        }

        for _, test := range cases {
                t.Run(test.Description, func(t *testing.T) {
                        got := ConvertToRoman(test.Arabic)
                        if got != test.Want {
                                t.Errorf("got %q, want %q", got, test.Want)
                        }
                })
        }
}
```

We can now easily add more cases without having to write any more test boilerplate.

Let's push on and go for 3

Write the test first

Add the following to our cases

```
{"3 gets converted to III", 3, "III"},
```

Try to run the test

```
=== RUN   TestRomanNumerals/3_gets_converted_to_III
    --- FAIL: TestRomanNumerals/3_gets_converted_to_III (0.00s)
        numeral_test.go:20: got 'I', want 'III'
```

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {

        if arabic == 3 {
                return "III"
        }
        if arabic == 2 {
                return "II"
        }
        return "I"
}
```

Refactor

OK so I'm starting to not enjoy these if statements and if you look at the code hard enough you can see that we're building a string of `I` based on the size of `arabic`.

We "know" that for more complicated numbers we will be doing some kind of arithmetic and string concatenation.

Let's try a refactor with these thoughts in mind, it *might not* be suitable for the end solution but that's OK. We can always throw our code away and start afresh with the tests we have to guide us.

```
func ConvertToRoman(arabic int) string {

        var result strings.Builder

        for i := 0; i < arabic; i++ {
                result.WriteString("I")
        }

        return result.String()
}
```

You may not have used `strings.Builder` before

> A Builder is used to efficiently build a string using Write methods. It minimizes memory copying.

Normally I wouldn't bother with such optimisations until I have an actual performance problem but the amount of code is not much larger than a "manual" appending on a string so we may as well use the faster approach.

The code looks better to me and describes the domain *as we know it right now*.

The Romans were into DRY too...

Things start getting more complicated now. The Romans in their wisdom thought repeating characters would become hard to read and count. So a rule with Roman Numerals is you can't have the same character repeated more than 3 times in a row.

Instead you take the next highest symbol and then "subtract" by putting a symbol to the left of it. Not all symbols can be used as subtractors; only I (1), X (10) and C (100).

For example `5` in Roman Numerals is `V`. To create 4 you do not do `IIII`, instead you do `IV`.

Write the test first

```
1 {"4 gets converted to IV (can't repeat more than 3 times)", 4, "IV"},
```

Try to run the test

```
1 === RUN   TestRomanNumerals/4_gets_converted_to_IV_(cant_repeat_more_than_3_times)
2     --- FAIL: TestRomanNumerals/4_gets_converted_to_IV_(cant_repeat_more_than_3_times
3         numeral_test.go:24: got 'IIII', want 'IV'
```

Write enough code to make it pass

```
func ConvertToRoman(arabic int) string {

        if arabic == 4 {
                return "IV"
        }

        var result strings.Builder

        for i := 0; i < arabic; i++ {
                result.WriteString("I")
        }

        return result.String()
}
```

Refactor

I don't "like" that we have broken our string building pattern and I want to carry on with it.

```
func ConvertToRoman(arabic int) string {

        var result strings.Builder

        for i := arabic; i > 0; i-- {
                if i == 4 {
                        result.WriteString("IV")
                        break
                }
                result.WriteString("I")
        }

        return result.String()
}
```

In order for 4 to "fit" with my current thinking I now count down from the Arabic number, adding symbols to our string as we progress. Not sure if this will work in the long run but let's see!

Let's make 5 work

Write the test first

```
{"5 gets converted to V", 5, "V"},
```

Try to run the test

```
=== RUN   TestRomanNumerals/5_gets_converted_to_V
    --- FAIL: TestRomanNumerals/5_gets_converted_to_V (0.00s)
        numeral_test.go:25: got 'IIV', want 'V'
```

Write enough code to make it pass

Just copy the approach we did for 4

```
func ConvertToRoman(arabic int) string {

        var result strings.Builder

        for i := arabic; i > 0; i-- {
                if i == 5 {
                        result.WriteString("V")
                        break
                }
                if i == 4 {
                        result.WriteString("IV")
                        break
                }
                result.WriteString("I")
        }
```

```
16
17          return result.String()
18  }
```

Refactor

Repetition in loops like this are usually a sign of an abstraction waiting to be called out. Short-circuiting loops can be an effective tool for readability but it could also be telling you something else.

We are looping over our Arabic number and if we hit certain symbols we are calling `break` but what we are *really* doing is subtracting over `i` in a ham-fisted manner.

```
1  func ConvertToRoman(arabic int) string {
2
3          var result strings.Builder
4
5          for arabic > 0 {
6                  switch {
7                  case arabic > 4:
8                          result.WriteString("V")
9                          arabic -= 5
10                 case arabic > 3:
11                         result.WriteString("IV")
12                         arabic -= 4
13                 default:
14                         result.WriteString("I")
15                         arabic--
16                 }
17         }
18
19         return result.String()
20 }
```

- Given the signals I'm reading from our code, driven from our tests of some very basic scenarios I can see that to build a Roman Numeral I need to subtract from `arabic` as I apply symbols
- The `for` loop no longer relies on an `i` and instead we will keep building our string until we

have subtracted enough symbols away from `arabic`.

I'm pretty sure this approach will be valid for 6 (VI), 7 (VII) and 8 (VIII) too. Nonetheless add the cases in to our test suite and check (I won't include the code for brevity, check the github for samples if you're unsure).

9 follows the same rule as 4 in that we should subtract `I` from the representation of the following number. 10 is represented in Roman Numerals with `X`; so therefore 9 should be `IX`.

Write the test first

```
1 {"9 gets converted to IX", 9, "IX"}
```

Try to run the test

```
1 === RUN   TestRomanNumerals/9_gets_converted_to_IX
2     --- FAIL: TestRomanNumerals/9_gets_converted_to_IX (0.00s)
3         numeral_test.go:29: got 'VIV', want 'IX'
```

Write enough code to make it pass

We should be able to adopt the same approach as before

```
1 case arabic > 8:
2     result.WriteString("IX")
3     arabic -= 9
```

Refactor

It *feels* like the code is still telling us there's a refactor somewhere but it's not totally obvious to me, so let's keep going.

I'll skip the code for this too, but add to your test cases a test for `10` which should be `X` and make it pass before reading on.

Here are a few tests I added as I'm confident up to 39 our code should work

```
{"10 gets converted to X", 10, "X"},
{"14 gets converted to XIV", 14, "XIV"},
{"18 gets converted to XVIII", 18, "XVIII"},
{"20 gets converted to XX", 20, "XX"},
{"39 gets converted to XXXIX", 39, "XXXIX"},
```

If you've ever done OO programming, you'll know that you should view `switch` statements with a bit of suspicion. Usually you are capturing a concept or data inside some imperative code when in fact it could be captured in a class structure instead.

Go isn't strictly OO but that doesn't mean we ignore the lessons OO offers entirely (as much as some would like to tell you).

Our switch statement is describing some truths about Roman Numerals along with behaviour.

We can refactor this by decoupling the data from the behaviour.

```
type RomanNumeral struct {
        Value  int
        Symbol string
}

var allRomanNumerals = []RomanNumeral{
        {10, "X"},
        {9, "IX"},
        {5, "V"},
        {4, "IV"},
        {1, "I"},
}

func ConvertToRoman(arabic int) string {

```

```
	var result strings.Builder

	for _, numeral := range allRomanNumerals {
		for arabic >= numeral.Value {
			result.WriteString(numeral.Symbol)
			arabic -= numeral.Value
		}
	}

	return result.String()
}
```

This feels much better. We've declared some rules around the numerals as data rather than hidden in an algorithm and we can see how we just work through the Arabic number, trying to add symbols to our result if they fit.

Does this abstraction work for bigger numbers? Extend the test suite so it works for the Roman number for 50 which is `L`.

Here are some test cases, try and make them pass.

```
{"40 gets converted to XL", 40, "XL"},
{"47 gets converted to XLVII", 47, "XLVII"},
{"49 gets converted to XLIX", 49, "XLIX"},
{"50 gets converted to L", 50, "L"},
```

Need help? You can see what symbols to add in this gist.

And the rest!

Here are the remaining symbols

Arabic	Roman
100	C
500	D
1000	M

Take the same approach for the remaining symbols, it should just be a matter of adding data to both the tests and our array of symbols.

Does your code work for `1984`: `MCMLXXXIV` ?

Here is my final test suite

```
func TestRomanNumerals(t *testing.T) {
        cases := []struct {
                Arabic int
                Roman  string
        }{
                {Arabic: 1, Roman: "I"},
                {Arabic: 2, Roman: "II"},
                {Arabic: 3, Roman: "III"},
                {Arabic: 4, Roman: "IV"},
                {Arabic: 5, Roman: "V"},
                {Arabic: 6, Roman: "VI"},
                {Arabic: 7, Roman: "VII"},
                {Arabic: 8, Roman: "VIII"},
                {Arabic: 9, Roman: "IX"},
                {Arabic: 10, Roman: "X"},
                {Arabic: 14, Roman: "XIV"},
                {Arabic: 18, Roman: "XVIII"},
                {Arabic: 20, Roman: "XX"},
                {Arabic: 39, Roman: "XXXIX"},
                {Arabic: 40, Roman: "XL"},
                {Arabic: 47, Roman: "XLVII"},
                {Arabic: 49, Roman: "XLIX"},
                {Arabic: 50, Roman: "L"},
                {Arabic: 100, Roman: "C"},
                {Arabic: 90, Roman: "XC"},
                {Arabic: 400, Roman: "CD"},
                {Arabic: 500, Roman: "D"},
                {Arabic: 900, Roman: "CM"},
                {Arabic: 1000, Roman: "M"},
                {Arabic: 1984, Roman: "MCMLXXXIV"},
                {Arabic: 3999, Roman: "MMMCMXCIX"},
                {Arabic: 2014, Roman: "MMXIV"},
                {Arabic: 1006, Roman: "MVI"},
                {Arabic: 798, Roman: "DCCXCVIII"},
        }
```

```
36          for _, test := range cases {
37                  t.Run(fmt.Sprintf("%d gets converted to %q", test.Arabic, test.Roman)
38                          got := ConvertToRoman(test.Arabic)
39                          if got != test.Roman {
40                                  t.Errorf("got %q, want %q", got, test.Roman)
41                          }
42                  })
43          }
44  }
```

- I removed `description` as I felt the *data* described enough of the information.
- I added a few other edge cases I found just to give me a little more confidence. With table based tests this is very cheap to do.

I didn't change the algorithm, all I had to do was update the `allRomanNumerals` array.

```
1  var allRomanNumerals = []RomanNumeral{
2          {1000, "M"},
3          {900, "CM"},
4          {500, "D"},
5          {400, "CD"},
6          {100, "C"},
7          {90, "XC"},
8          {50, "L"},
9          {40, "XL"},
10         {10, "X"},
11         {9, "IX"},
12         {5, "V"},
13         {4, "IV"},
14         {1, "I"},
15 }
```

Parsing Roman Numerals

We're not done yet. Next we're going to write a function that converts *from* a Roman Numeral to an `int`

Write the test first

We can re-use our test cases here with a little refactoring

Move the `cases` variable outside of the test as a package variable in a `var` block.

```
func TestConvertingToArabic(t *testing.T) {
        for _, test := range cases[:1] {
                t.Run(fmt.Sprintf("%q gets converted to %d", test.Roman, test.Arabic)
                        got := ConvertToArabic(test.Roman)
                        if got != test.Arabic {
                                t.Errorf("got %d, want %d", got, test.Arabic)
                        }
                })
        }
}
```

Notice I am using the slice functionality to just run one of the tests for now (`cases[:1]`) as trying to make all of those tests pass all at once is too big a leap

---

Try to run the test

```
./numeral_test.go:60:11: undefined: ConvertToArabic
```

Write the minimal amount of code for the test to run and check the failing test output

Add our new function definition

```
func ConvertToArabic(roman string) int {
```

```
3 }       return 0
```

The test should now run and fail

```
1 --- FAIL: TestConvertingToArabic (0.00s)
2     --- FAIL: TestConvertingToArabic/'I'_gets_converted_to_1 (0.00s)
3         numeral_test.go:62: got 0, want 1
```

Write enough code to make it pass

You know what to do

```
1 func ConvertToArabic(roman string) int {
2         return 1
3 }
```

Next, change the slice index in our test to move to the next test case (e.g. `cases[:2]`). Make it pass yourself with the dumbest code you can think of, continue writing dumb code (best book ever right?) for the third case too. Here's my dumb code.

```
1 func ConvertToArabic(roman string) int {
2         if roman == "III" {
3                 return 3
4         }
5         if roman == "II" {
6                 return 2
7         }
8         return 1
9 }
```

Through the dumbness of *real code that works* we can start to see a pattern like before. We need to iterate through the input and build *something*, in this case a total.

```
1 func ConvertToArabic(roman string) int {
```

```
        total := 0
        for range roman {

                total++
        }
        return total
}
```

Write the test first

Next we move to `cases[:4]` (`IV`) which now fails because it gets 2 back as that's the length of the string.

Write enough code to make it pass

```
// earlier..
type RomanNumerals []RomanNumeral

func (r RomanNumerals) ValueOf(symbol string) int {
        for _, s := range r {
                if s.Symbol == symbol {
                        return s.Value
                }
        }

        return 0
}

// later..
func ConvertToArabic(roman string) int {
        total := 0

        for i := 0; i < len(roman); i++ {
                symbol := roman[i]

                // look ahead to next symbol if we can and, the current symbol is bas
                if i+1 < len(roman) && symbol == 'I' {
```

```
                        nextSymbol := roman[i+1]
                        // build the two character string
                        potentialNumber := string([]byte{symbol, nextSymbol})

                        // get the value of the two character string
                        value := allRomanNumerals.ValueOf(potentialNumber)

                        if value != 0 {
                                total += value
                                i++ // move past this character too for the next loop
                        } else {
                                total++
                        }
                } else {
                        total++
                }
        }
        return total
}
```

This is horrible but it does work. It's so bad I felt the need to add comments.

- I wanted to be able to look up an integer value for a given roman numeral so I made a type from our array of `RomanNumeral`s and then added a method to it, `ValueOf`
- Next in our loop we need to look ahead *if* the string is big enough *and the current symbol is a valid subtractor*. At the moment it's just `I` (1) but can also be `X` (10) or `C` (100).
  - If it satisfies both of these conditions we need to lookup the value and add it to the total *if* it is one of the special subtractors, otherwise ignore it
  - Then we need to further increment `i` so we don't count this symbol twice

---

Refactor

I'm not entirely convinced this will be the long-term approach and there's potentially some interesting refactors we could do, but I'll resist that in case our approach is totally wrong. I'd rather make a few more tests pass first and see. For the meantime I made the first `if` statement slightly less horrible.

```
func ConvertToArabic(roman string) int {
	total := 0

	for i := 0; i < len(roman); i++ {
		symbol := roman[i]

		if couldBeSubtractive(i, symbol, roman) {
			nextSymbol := roman[i+1]

			// build the two character string
			potentialNumber := string([]byte{symbol, nextSymbol})

			// get the value of the two character string
			value := allRomanNumerals.ValueOf(potentialNumber)

			if value != 0 {
				total += value
				i++ // move past this character too for the next loop
			} else {
				total++
			}
		} else {
			total++
		}
	}
	return total
}

func couldBeSubtractive(index int, currentSymbol uint8, roman string) bool {
	return index+1 < len(roman) && currentSymbol == 'I'
}
```

Write the test first

Let's move on to `cases[:5]`

```
=== RUN   TestConvertingToArabic/'V'_gets_converted_to_5
    --- FAIL: TestConvertingToArabic/'V'_gets_converted_to_5 (0.00s)
        numeral_test.go:62: got 1, want 5
```

Write enough code to make it pass

Apart from when it is subtractive our code assumes that every character is a `I` which is why the value is 1. We should be able to re-use our `ValueOf` method to fix this.

```
func ConvertToArabic(roman string) int {
        total := 0

        for i := 0; i < len(roman); i++ {
                symbol := roman[i]

                // look ahead to next symbol if we can and, the current symbol is bas
                if couldBeSubtractive(i, symbol, roman) {
                        nextSymbol := roman[i+1]

                        // build the two character string
                        potentialNumber := string([]byte{symbol, nextSymbol})

                        if value := allRomanNumerals.ValueOf(potentialNumber); value
                                total += value
                                i++ // move past this character too for the next loop
                        } else {
                                total++ // this is fishy...
                        }
                } else {
                        total += allRomanNumerals.ValueOf(string([]byte{symbol}))
                }
        }
        return total
}
```

Refactor

When you index strings in Go, you get a `byte`. This is why when we build up the string again we have to do stuff like `string([]byte{symbol})`. It's repeated a couple of times, let's just move

that functionality so that `ValueOf` takes some bytes instead.

```
func (r RomanNumerals) ValueOf(symbols ...byte) int {

        symbol := string(symbols)
        for _, s := range r {
                if s.Symbol == symbol {
                        return s.Value
                }
        }

        return 0
}
```

Then we can just pass in the bytes as is, to our function

```
func ConvertToArabic(roman string) int {
        total := 0

        for i := 0; i < len(roman); i++ {
                symbol := roman[i]

                if couldBeSubtractive(i, symbol, roman) {
                        if value := allRomanNumerals.ValueOf(symbol, roman[i+1]); val
                                total += value
                                i++ // move past this character too for the next loop
                        } else {
                                total++ // this is fishy...
                        }
                } else {
                        total += allRomanNumerals.ValueOf(symbol)
                }
        }
        return total
}
```

It's still pretty nasty, but it's getting there.

If you start moving our `cases[:xx]` number through you'll see that quite a few are passing now. Remove the slice operator entirely and see which ones fail, here's some examples from my suite

```
=== RUN   TestConvertingToArabic/'XL'_gets_converted_to_40
    --- FAIL: TestConvertingToArabic/'XL'_gets_converted_to_40 (0.00s)
        numeral_test.go:62: got 60, want 40
=== RUN   TestConvertingToArabic/'XLVII'_gets_converted_to_47
    --- FAIL: TestConvertingToArabic/'XLVII'_gets_converted_to_47 (0.00s)
        numeral_test.go:62: got 67, want 47
=== RUN   TestConvertingToArabic/'XLIX'_gets_converted_to_49
    --- FAIL: TestConvertingToArabic/'XLIX'_gets_converted_to_49 (0.00s)
        numeral_test.go:62: got 69, want 49
```

I think all we're missing is an update to `couldBeSubtractive` so that it accounts for the other kinds of subtractive symbols

```
func couldBeSubtractive(index int, currentSymbol uint8, roman string) bool {
        isSubtractiveSymbol := currentSymbol == 'I' || currentSymbol == 'X' || currer
        return index+1 < len(roman) && isSubtractiveSymbol
}
```

Try again, they still fail. However we left a comment earlier...

```
total++ // this is fishy...
```

We should never be just incrementing `total` as that implies every symbol is a `I`. Replace it with:

```
total += allRomanNumerals.ValueOf(symbol)
```

And all the tests pass! Now that we have fully working software we can indulge ourselves in some refactoring, with confidence.

Refactor

Here is all the code I finished up with. I had a few failed attempts but as I keep emphasising, that's fine and the tests help me play around with the code freely.

```
import "strings"

func ConvertToArabic(roman string) (total int) {
	for _, symbols := range windowedRoman(roman).Symbols() {
		total += allRomanNumerals.ValueOf(symbols...)
	}
	return
}

func ConvertToRoman(arabic int) string {
	var result strings.Builder

	for _, numeral := range allRomanNumerals {
		for arabic >= numeral.Value {
			result.WriteString(numeral.Symbol)
			arabic -= numeral.Value
		}
	}

	return result.String()
}

type romanNumeral struct {
	Value  int
	Symbol string
}

type romanNumerals []romanNumeral

func (r romanNumerals) ValueOf(symbols ...byte) int {
	symbol := string(symbols)
	for _, s := range r {
		if s.Symbol == symbol {
			return s.Value
		}
	}

	return 0
```

```
}
func (r romanNumerals) Exists(symbols ...byte) bool {
	symbol := string(symbols)

	for _, s := range r {
		if s.Symbol == symbol {
			return true
		}
	}
	return false
}

var allRomanNumerals = romanNumerals{
	{1000, "M"},
	{900, "CM"},
	{500, "D"},
	{400, "CD"},
	{100, "C"},
	{90, "XC"},
	{50, "L"},
	{40, "XL"},
	{10, "X"},
	{9, "IX"},
	{5, "V"},
	{4, "IV"},
	{1, "I"},
}

type windowedRoman string

func (w windowedRoman) Symbols() (symbols [][]byte) {
	for i := 0; i < len(w); i++ {
		symbol := w[i]
		notAtEnd := i+1 < len(w)

		if notAtEnd && isSubtractive(symbol) && allRomanNumerals.Exists(symbo
			symbols = append(symbols, []byte{symbol, w[i+1]})
			i++
		} else {
			symbols = append(symbols, []byte{symbol})
		}
	}
	return
}
```

```
84 func isSubtractive(symbol uint8) bool {
85         return symbol == 'I' || symbol == 'X' || symbol == 'C'
86 }
```

My main problem with the previous code is similar to our refactor from earlier. We had too many concerns coupled together. We wrote an algorithm which was trying to extract Roman Numerals from a string *and* then find their values.

So I created a new type `windowedRoman` which took care of extracting the numerals, offering a `Symbols` method to retrieve them as a slice. This meant our `ConvertToArabic` function could simply iterate over the symbols and total them.

I broke the code down a bit by extracting some functions, especially around the wonky if statement to figure out if the symbol we are currently dealing with is a two character subtractive symbol.

There's probably a more elegant way but I'm not going to sweat it. The code is there and it works and it is tested. If I (or anyone else) finds a better way they can safely change it - the hard work is done.

An intro to property based tests

There have been a few rules in the domain of Roman Numerals that we have worked with in this chapter

- Can't have more than 3 consecutive symbols
- Only I (1), X (10) and C (100) can be "subtractors"
- Taking the result of `ConvertToRoman(N)` and passing it to `ConvertToArabic` should return us `N`

The tests we have written so far can be described as "example" based tests where we provide the tooling some examples around our code to verify.

What if we could take these rules that we know about our domain and somehow exercise them against our code?

Property based tests help you do this by throwing random data at your code and verifying the rules you describe always hold true. A lot of people think property based tests are mainly about random

Enough words, let's see some code

```
func TestPropertiesOfConversion(t *testing.T) {
        assertion := func(arabic int) bool {
                roman := ConvertToRoman(arabic)
                fromRoman := ConvertToArabic(roman)
                return fromRoman == arabic
        }

        if err := quick.Check(assertion, nil); err != nil {
                t.Error("failed checks", err)
        }
}
```

Rationale of property

Our first test will check that if we transform a number into Roman, when we use our other function to convert it back to a number that we get what we originally had.

- Given random number (e.g `4`).
- Call `ConvertToRoman` with random number (should return `IV` if `4`).
- Take the result of above and pass it to `ConvertToArabic`.
- The above should give us our original input (`4`).

This feels like a good test to build us confidence because it should break if there's a bug in either. The only way it could pass is if they have the same kind of bug; which isn't impossible but feels unlikely.

Technical explanation

We're using the testing/quick package from the standard library

Reading from the bottom, we provide `quick.Check` a function that it will run against a number of random inputs, if the function returns `false` it will be seen as failing the check.

Our `assertion` function above takes a random number and runs our functions to test the property.

Run our test

Try running it; your computer may hang for a while, so kill it when you're bored :)

What's going on? Try adding the following to the assertion code.

```
assertion := func(arabic int) bool {
    if arabic < 0 || arabic > 3999 {
         log.Println(arabic)
         return true
    }
    roman := ConvertToRoman(arabic)
    fromRoman := ConvertToArabic(roman)
    return fromRoman == arabic
}
```

You should see something like this:

```
=== RUN   TestPropertiesOfConversion
2019/07/09 14:41:27 6849766357708982977
2019/07/09 14:41:27 -7028152357875163913
2019/07/09 14:41:27 -6752532134903680693
2019/07/09 14:41:27 4051793897228170080
2019/07/09 14:41:27 -1111868396280600429
2019/07/09 14:41:27 8851967058300421387
2019/07/09 14:41:27 562755830018219185
```

Just running this very simple property has exposed a flaw in our implementation. We used `int` as our input but:

- You can't do negative numbers with Roman Numerals
- Given our rule of a max of 3 consecutive symbols we can't represent a value greater than 3999 (well, kinda) and `int` has a much higher maximum value than 3999.

This is great! We've been forced to think more deeply about our domain which is a real strength of property based tests.

Clearly `int` is not a great type. What if we tried something a little more appropriate?

`uint16`

Go has types for *unsigned integers*, which means they cannot be negative; so that rules out one class of bug in our code immediately. By adding 16, it means it is a 16 bit integer which can store a max of `65535`, which is still too big but gets us closer to what we need.

Try updating the code to use `uint16` rather than `int`. I updated `assertion` in the test to give a bit more visibility.

```
assertion := func(arabic uint16) bool {
        if arabic > 3999 {
                return true
        }
        t.Log("testing", arabic)
        roman := ConvertToRoman(arabic)
        fromRoman := ConvertToArabic(roman)
        return fromRoman == arabic
}
```

If you run the test they now actually run and you can see what is being tested. You can run multiple times to see our code stands up well to the various values! This gives me a lot of confidence that our code is working how we want.

The default number of runs `quick.Check` performs is 100 but you can change that with a config.

```
if err := quick.Check(assertion, &quick.Config{
        MaxCount: 1000,
}); err != nil {
        t.Error("failed checks", err)
}
```

Further work

- Can you write property tests that check the other properties we described?
- Can you think of a way of making it so it's impossible for someone to call our code with a number greater than 3999?
  - You could return an error
  - Or create a new type that cannot represent > 3999

- What do you think is best?

Wrapping up

More TDD practice with iterative development

Did the thought of writing code that converts 1984 into MCMLXXXIV feel intimidating to you at first? It did to me and I've been writing software for quite a long time.

The trick, as always, is to **get started with something simple** and take **small steps**.

At no point in this process did we make any large leaps, do any huge refactorings, or get in a mess.

I can hear someone cynically saying "this is just a kata". I can't argue with that, but I still take this same approach for every project I work on. I never ship a big distributed system in my first step, I find the simplest thing the team could ship (usually a "Hello world" website) and then iterate on small bits of functionality in manageable chunks, just like how we did here.

The skill is knowing *how* to split work up, and that comes with practice and with some lovely TDD to help you on your way.

Property based tests

- Built into the standard library
- If you can think of ways to describe your domain rules in code, they are an excellent tool for giving you more confidence
- Force you to think about your domain deeply
- Potentially a nice complement to your test suite

Postscript

This book is reliant on valuable feedback from the community. Dave is an enormous help in practically every chapter. But he had a real rant about my use of 'Arabic numerals' in this chapter so, in the interests of full disclosure, here's what he said.

>

Just going to write up why a value of type `int` isn't really an 'arabic numeral'. This might be me being way too precise so I'll completely understand if you tell me to f off.
A *digit* is a character used in the representation of numbers - from the Latin for 'finger', as we usually have ten of them. In the Arabic (also called Hindu-Arabic) number system there are ten of them. These Arabic digits are:

```
0 1 2 3 4 5 6 7 8 9
```

A *numeral* is the representation of a number using a collection of digits. An Arabic numeral is a number represented by Arabic digits in a base 10 positional number system. We say 'positional' because each digit has a different value based upon its position in the numeral. So

```
1337
```

The `1` has a value of one thousand because its the first digit in a four digit numeral.

Roman are built using a reduced number of digits (`I`, `V` etc...) mainly as values to produce the numeral. There's a bit of positional stuff but it's mostly `I` always representing 'one'.

So, given this, is `int` an 'Arabic number'? The idea of a number is not at all tied to its representation - we can see this if we ask ourselves what the correct representation of this number is:

```
255
11111111
two-hundred and fifty-five
FF
377
```

Yes, this is a trick question. They're all correct. They're the representation of the same number in the decimal, binary, English, hexadecimal and octal number systems respectively.

The representation of a number as a numeral is *independent* of its properties as a number - and we can see this when we look at integer literals in Go:

```
        0xFF == 255 // true
```

And how we can print integers in a format string:

```
n := 255
fmt.Printf("%b %c %d %o %q %x %X %U", n, n, n, n, n, n, n, n)
// 11111111 ÿ 255 377 'ÿ' ff FF U+00FF
```

We can write the same integer both as a hexadecimal and an Arabic (decimal) numeral.
So when the function signature looks like `ConvertToRoman(arabic int) string` it's making a bit of an assumption about how it's being called. Because sometimes `arabic` will be written as a decimal integer literal

```
1 ConvertToRoman(255)
```

But it could just as well be written

```
1 ConvertToRoman(0xFF)
```

Really, we're not 'converting' from an Arabic numeral at all, we're 'printing' - representing - an `int` as a Roman numeral - and `int`s are not numerals, Arabic or otherwise; they're just numbers. The `ConvertToRoman` function is more like `strconv.Itoa` in that it's turning an `int` into a `string`.

But every other version of the kata doesn't care about this distinction so :shrug: